Check for updates

Computer Science Curricula 2023

January 2024

The Joint Task Force on
Computer Science Curricula

Association for Computing Machinery (ACM)

IEEE-Computer Society (IEEE-CS)

Association for the Advancement of
Artificial Intelligence (AAAI)

Association for Computing Machinery

IEEE COMPUTER SOCIETY

Association for the Advancement of Artificial Intelligence

Steering Committee members:

ACM members:

- Amruth N. Kumar, Ramapo College of NJ, Mahwah, NJ, USA (ACM Co-Chair)
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Brett A. Becker, University College Dublin, Ireland
- Richard L. Blumenthal, Regis University, Denver, CO, USA
- Michael Goldweber, Denison University, Granville, OH, USA
- Pankaj Jalote, Indraprastha Institute of Information Technology, Delhi, India
- Susan Reiser, University of North Carolina Asheville, Asheville, NC, USA
- Titus Winters, Google Inc., New York, NY, USA

IEEE-CS members:

- Rajendra K. Raj, Rochester Institute of Technology (RIT), Rochester, NY, USA (IEEE-CS Co-Chair)
- Sherif G. Aly, American University in Cairo, Egypt
- Douglas Lea, SUNY Oswego, NY, USA
- Michael J. Oudshoorn, High Point University, High Point, NC, USA
- Marcelo Pias, Federal University of Rio Grande (FURG), Brazil
- Christian Servin, El Paso Community College, El Paso, TX, USA
- Qiao Xiang, Xiamen University, China

AAAI members:

- Eric Eaton, University of Pennsylvania, Philadelphia, PA, USA
- Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, New York, NY, USA

Copyright © 2024 by ACM, IEEE, AAAI

ALL RIGHTS RESERVED

Copyright and Reprint Permissions: Permission is granted to use these curriculum guidelines for the development of educational materials and programs. Other use requires specific permission. Permission requests should be addressed to: ACM Permissions Dept. at permissions@acm.org, the IEEE Copyrights Manager at copyrights@ieee.org, or AAAI at info@aaai.org.

ISBN: 979-8-4007-1033-9
DOI: 10.1145/3664191
Web link: https://doi.org/10.1145/3664191

Cite as:
Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. 2023. Computer Science Curricula 2023. ACM Press, IEEE Computer Society Press and AAAI Press. DOI: https://doi.org/10.1145/3664191

Sponsoring Societies

This report was made possible by
financial support from the following societies:

Association for Computing Machinery (ACM)
IEEE Computer Society (IEEE-CS)
Association for the Advancement of Artificial Intelligence (AAAI)

The CS2023 Final Report has been endorsed by ACM, IEEE-CS, and AAAI

Cover art by Robert Vizzini

Contents

Executive Summary 21
Overview 23
Introduction to CS2023 25
History 25
Vision Statement 26
References 26
Introduction to Knowledge Model 29
Definitions and Terminology 29
CS2023 Knowledge Model 29
Structure of CS2023 Knowledge Area Explained 32
Changes since CS2013 34
Designing/Revising a Curriculum Based on the Knowledge Model 38
References 39
Introduction to Competency Framework 41
Competence Model - Definitions and Terminology 41
CS2023 Competency Model 42
CS2023 Framework for Systematically Identifying Tasks 43
Designing/Revising a Curriculum Using the Competency Framework 44
References 45
The Design 47
Principles and Processes 49
The Task Force 49
The Steering Committee 49
Knowledge Area Committees 49
Guiding Principles for the Process 49
The Process 50
Guiding Concerns for the Curricular Recommendations 51
The Knowledge Model Revision Process 52
Characteristics of Computer Science Graduates 55
Professional knowledge and skills—for a technical solution 55
Professional Responsibilities – for the whole solution 55

Professional dispositions – the whole person view 56
References 56
Challenges and Opportunities for Computer Science 57
The Details 61
Body of Knowledge 63
Artificial Intelligence (AI) 65
Preamble 65
Changes since CS2013 66
Consider recent AI advances when using this curriculum 66
Core Hours 66
Knowledge Units 67
AI-Introduction: Fundamental Issues 67
AI-Search: Search 68
AI-KRR: Fundamental Knowledge Representation and Reasoning 70
AI-ML: Machine Learning 71
AI-SEP: Applications and Societal Impact 75
AI-LRR: Logical Representation and Reasoning 76
AI-Probability: Probabilistic Representation and Reasoning 77
AI-Planning: Planning 78
AI-Agents: Agents and Cognitive Systems 79
AI-NLP: Natural Language Processing 79
AI-Robotics: Robotics 80
AI-Vision: Perception and Computer Vision 82
Professional Dispositions 83
Mathematics Requirements 83
Course Packaging Suggestions 84
Committee 85
Algorithmic Foundations (AL) 87
Preamble 87
Changes since CS2013 87
Core Hours 88
Knowledge Units 88
AL-Foundational: Foundational Data Structures and Algorithms 88

AL-Strategies: Algorithmic Strategies 90
AL-Complexity: Complexity 91
AL-Models: Computational Models and Formal Languages 93
AL-SEP: Society, Ethics, and the Profession 95
Professional Dispositions 96
Mathematics Requirements 96
Course Packaging Suggestions 96
Committee 99
Architecture and Organization (AR) 101
Preamble 101
Changes since CS2013 101
Core Hours 101
Knowledge Units 102
AR-Logic: Digital Logic and Digital Systems 102
AR-Representation: Machine-Level Data Representation 103
AR-Assembly: Assembly Level Machine Organization 103
AR-Memory: Memory Hierarchy 104
AR-IO: Interfacing and Communication 105
AR-Organization: Functional Organization 106
AR-Performance-Energy: Performance and Energy Efficiency 106
AR-Heterogeneity: Heterogeneous Architectures 107
AR-Security: Secure Processor Architectures 107
AR-Quantum: Quantum Architectures 108
AR-SEP: Sustainability Issues 109
Professional Dispositions 109
Mathematics Requirements 110
Course Packaging Suggestions 110
Committee 111
Data Management (DM) 113
Preamble 113
Changes since CS2013 114
Core Hours 114
Knowledge Units 115

DM-Data: The Role of Data and the Data Life Cycle 115
DM-Core: Core Database System Concepts 115
DM-Modeling: Data Modeling 116
DM-Relational: Relational Databases 117
DM-Querying: Query Construction 118
DM-Processing: Query Processing 118
DM-Internals: DBMS Internals 119
DM-NoSQL: NoSQL Systems 120
DM-Security: Data Security and Privacy 121
DM-Analytics: Data Analytics 121
DM-Distributed: Distributed Databases/Cloud Computing 122
DM-Unstructured: Semi-structured and Unstructured Databases 122
DM-SEP: Society, Ethics, and the Profession 123
Professional Dispositions 123
Mathematics Requirements 124
Course Packaging Suggestions 124
Committee 125
References 125
Foundations of Programming Languages (FPL) 127
Preamble 127
Changes since CS2013 127
Core Hours 129
Knowledge Units 130
FPL-OOP: Object-Oriented Programming 130
FPL-Functional: Functional Programming 131
FPL-Logic: Logic Programming 133
FPL-Scripting: Shell Scripting 133
FPL-Event-Driven: Event-Driven and Reactive Programming 134
FPL-Parallel: Parallel and Distributed Computing 135
FPL-Aspect: Aspect-Oriented Programming 136
FPL-Types: Type Systems 136
FPL-Systems: Systems Execution and Memory Model 138
FPL-Translation: Language Translation and Execution 139

FPL-Abstraction: Program Abstraction and Representation 140
FPL-Syntax: Syntax Analysis 141
FPL-Semantics: Compiler Semantic Analysis 141
FPL-Analysis: Program Analysis and Analyzers 142
FPL-Code: Code Generation 142
FPL-Run-Time: Run-time Behavior and Systems 143
FPL-Constructs: Advanced Programming Constructs 144
FPL-Pragmatics: Language Pragmatics 144
FPL-Formalism: Formal Semantics 145
FPL-Methodologies: Formal Development Methodologies 146
FPL-Design: Design Principles of Programming Languages 146
FPL-SEP: Society, Ethics, and the Profession 147
Professional Dispositions 147
Mathematics Requirements 147
Course Packaging Suggestions 148
Committee 149
Graphics and Interactive Techniques (GIT) 151
Preamble 151
Changes since CS2013 154
Core Hours 154
Knowledge Units 155
GIT-Fundamentals: Fundamental Concepts 155
GIT-Visualization: Visualization 156
GIT-Rendering: Applied Rendering and Techniques 157
GIT-Modeling: Geometric Modeling 158
GIT-Shading: Shading and Advanced Rendering 159
GIT-Animation: Computer Animation 160
GIT-Simulation: Simulation 161
GIT-Immersion: Immersion 161
GIT-Interaction: Interaction 162
GIT-Image: Image Processing 163
GIT-Physical: Tangible/Physical Computing 164
GIT-SEP: Society, Ethics, and the Profession 165

Professional Dispositions 166
Mathematics Requirements 166
Course Packaging Suggestions 167
Committee 170
References 171
Human-Computer Interaction (HCI) 173
Preamble 173
Changes since CS2013 173
Core Hours 174
Knowledge Units 174
HCI-User: Understanding the User: Individual goals and interactions with others 174
HCI-Accountability: Accountability and Responsibility in Design 176
HCI-Accessibility: Accessibility and Inclusive Design 177
HCI-Evaluation: Evaluating the Design 178
HCI-Design: System Design 179
HCI-SEP: Society, Ethics, and the Profession 181
Professional Dispositions 182
Mathematics Requirements 182
Course Packaging Suggestions 182
Committee 183
Mathematical and Statistical Foundations (MSF) 185
Preamble 185
Changes since CS2013 185
Core Hours 185
Acknowledging some tensions 185
Rationale for recommended hours 186
Knowledge Units 187
MSF-Discrete: Discrete Mathematics 187
MSF-Probability: Probability 188
MSF-Statistics: Statistics 190
MSF-Linear: Linear Algebra 191
MSF-Calculus 192
Professional Dispositions 194

Mathematics Requirements....194
Course Packaging Suggestions....195
Committee....195
References....196
Networking and Communication (NC)....197
Preamble....197
Changes since CS2013....197
Core Hours....198
Knowledge Units....198
NC-Fundamentals: Fundamentals....198
NC-Applications: Networked Applications....199
NC-Reliability: Reliability Support....199
NC-Routing: Routing and Forwarding....199
NC-SingleHop: Single Hop Communication....200
NC-Security: Network Security....200
NC-Mobility: Mobility....201
NC-Emerging: Emerging Topics....201
Professional Dispositions....201
Mathematics Requirements....202
Course Packaging Suggestions....202
Committee....203
Operating Systems (OS)....205
Preamble....205
Changes since CS2013....205
Core Hours....205
Knowledge Units....206
OS-Purpose: Role and Purpose of Operating Systems....206
OS-Principles: Principles of Operating System....207
OS-Concurrency: Concurrency....208
OS-Protection: Protection and Safety....209
OS-Scheduling: Scheduling....209
OS-Process: Process Model....210
OS-Memory: Memory Management....210

OS-Devices: Device management 211
OS-Files: File Systems API and Implementation 212
OS-Advanced-Files: Advanced File systems 212
OS-Virtualization: Virtualization 213
OS-Real-time: Real-time and Embedded Systems 214
OS-Faults: Fault tolerance 214
OS-SEP: Society, Ethics, and the Profession 215
Professional Dispositions 215
Mathematics Requirements 215
Course Packaging Suggestions 215
Committee 216
Parallel and Distributed Computing (PDC) 217
Preamble 217
Changes since CS2013 217
Overview 217
Core Hours 218
Knowledge Units 219
PDC-Programs: Programs 219
PDC-Communication: Communication 220
PDC-Coordination: Coordination 222
PDC-Evaluation: Evaluation 224
PDC-Algorithms: Algorithms 225
Professional Dispositions 227
Mathematics Requirements 227
Course Packaging Suggestions 227
Committee 228
References 228
Software Development Fundamentals (SDF) 229
Preamble 229
Changes since CS 2013 229
Overview 229
Core Hours 230
Knowledge Units 230

SDF-Fundamentals: Fundamental Programming Concepts and Practices 230
SDF-Data-Structures: Fundamental Data Structures 231
SDF-Algorithms: Algorithms 232
SDF-Practices: Software Development Practices 232
SDF-SEP: Society, Ethics, and the Profession 233
Professional Dispositions 233
Mathematics Requirements 234
Course Packaging Suggestions 234
Committee 235
Software Engineering (SE) 237
Preamble 237
Changes since CS 2013 238
Overview 238
Core Hours 239
Knowledge Units 240
SE-Teamwork: Teamwork 240
SE-Tools: Tools and Environments 241
SE-Requirements: Product Requirements 242
SE-Design: Software Design 243
SE-Construction: Software Construction 245
SE-Validation: Software Verification and Validation 246
SE-Refactoring: Refactoring and Code Evolution 249
SE-Reliability: Software Reliability 250
SE-Formal: Formal Methods 251
Professional Dispositions 251
Mathematics Requirements 252
Course Packaging Suggestions 252
Committee 252
Security (SEC) 255
Preamble 255
Changes since CS2013 255
Differences between CS2023 Security knowledge area and Cybersecurity 256
Core Hours 257

Knowledge Units 257
SEC-Foundations: Foundational Security 257
SEC-SEP: Society, Ethics, and the Profession 258
SEC-Coding: Secure Coding 259
SEC-Crypto: Cryptography 260
SEC-Engineering: Security Analysis, Design, and Engineering 261
SEC-Forensics: Digital Forensics 262
SEC-Governance: Security Governance 263
Professional Dispositions 263
Mathematics Requirements 264
Course Packaging Suggestions 264
Committee 267
References 267
Society, Ethics, and the Profession (SEP) 269
Preamble 269
Changes Since CS2013 272
Core Hours 272
Knowledge Units 273
SEP-Context: Social Context 273
SEP-Ethical-Analysis: Methods for Ethical Analysis 274
SEP-Professional-Ethics: Professional Ethics 276
SEP-IP: Intellectual Property 277
SEP-Privacy: Privacy and Civil Liberties 278
SEP-Communication: Communication 279
SEP-Sustainability: Sustainability 281
SEP-History: Computing History 282
SEP-Economies: Economies of Computing 283
SEP-Security: Security Policies, Laws and Computer Crimes 284
SEP-DEIA: Diversity, Equity, Inclusion, and Accessibility 285
Professional Dispositions 286
Course Packaging Suggestions 287
Committee 289
References 290

Systems Fundamentals (SF) 291
Preamble 291
Changes since CS2013 291
Core Hours 292
Knowledge Units 292
SF-Overview: Overview of Computer Systems 292
SF-Foundations: Basic Concepts 293
SF-Resource: Resource Management 294
SF-Performance: System Performance 294
SF-Evaluation: Performance Evaluation 295
SF-Reliability: System Reliability 296
SF-Security: System Security 297
SF-Design: System Design 297
SF-SEP: Society, Ethics, and the Profession 297
Professional Dispositions 298
Mathematics Requirements 298
Course Packaging Suggestions 298
Committee 299
Specialized Platform Development (SPD) 301
Preamble 301
Changes since CS2013 301
Core Hours 301
Knowledge Units 302
SPD-Common: Common Aspects/Shared Concerns 302
SPD-Web: Web Platforms 303
SPD-Mobile: Mobile Platforms 303
SPD-Robot: Robot Platforms 304
SPD-Embedded: Embedded Platforms 305
SPD-Game: Game Platforms 306
SPD-Interactive: Interactive Computing Platforms 308
SPD-SEP/Mobile 309
SPD-SEP/Web 310
SPD-SEP/Game 310

SPD-SEP/Robotics 311
SPD-SEP/Interactive 312
Professional Dispositions 312
Mathematics Requirements 312
Course Packaging Suggestions 313
Committee 315
Core Topics Table 317
Software Competency Area 317
SDF: Software Development Fundamentals 317
FPL: Foundations of Programming Languages 322
SE: Software Engineering 328
Systems Competency Area 330
AR: Architecture and Organization 331
OS: Operating Systems 335
NC: Networking and Communication 339
PDC: Parallel and Distributed Computing 342
SF: Systems Fundamentals 350
DM: Data Management 352
SEC: Security 356
Applications Competency Area 360
AI: Artificial Intelligence 361
GIT: Graphics and Interactive Techniques 367
HCI: Human-Computer Interaction 370
SPD: Specialized Platform Development 375
Crosscutting Core Topics 380
SEP: Society, Ethics, and the Profession 380
MSF: Mathematical and Statistical Foundations 387
Curricular Packaging 391
8 Course Model 391
10 Course Model 391
12 Course Model 392
16 Course Model 392
Model 1: 392

Model 2: 393
Model 3: 394
Competency Framework Examples 395
Sample Tasks 395
Software Competency Area 395
Systems Competency Area 396
Applications Competency Area 397
Sample Competency Specifications 398
Software Competency Area 398
Systems Competency Area 400
Applications Competency Area 403
Pedagogy and Practices 407
Pedagogical Considerations 409
Introduction 409
Curriculum-Wide Considerations 409
Considerations by Knowledge Area 410
Artificial Intelligence (AI) 410
Algorithmic Foundations (AL) 410
Architecture and Organization (AR) 410
Data Management (DM) 410
Foundations of Programming Languages (FPL) 411
Mathematical and Statistical Foundations (MSF) 411
Networking and Communication (NC) 412
Operating Systems (OS) 412
Parallel and Distributed Computing (PDC) 412
Software Development Fundamentals (SDF) 412
Software Engineering (SE) 413
Security (SEC) 413
Society, Ethics, and the Profession (SEP) 413
Specialized Platform Development (SPD) 414
Systems Fundamentals (SF) 414
Curricular Practices in Computer Science 417
Introduction 417

Social Aspects 417
Pedagogical Considerations 418
Educational Practices 419
Teaching about Accessibility in Computer Science Education 420
Computing for Social Good in Education 422
Multiple Approaches for Teaching Responsible Computing 424
Making ethics at home in Global CS Education: Provoking stories from the Souths 429
CS + X: Approaches, Challenges, and Opportunities in Developing Interdisciplinary Computing Curricula 431
The Role of Formal Methods in Computer Science Education 433
Quantum Computing Education: A Curricular Perspective 435
Generative AI in Introductory Programming 438
The 2022 Undergraduate Database Course in Computer Science: What to Teach? 441
Computer Science Curriculum Guidelines: A New Liberal Arts Perspective 443
Computer Science Education in Community Colleges 446
Generative AI and the Curriculum 449
Introduction 449
Implications by Competency Area - Software 449
Algorithmic Foundations (AL) 449
Foundations of Programming Languages (FPL) 449
Software Engineering (SE) 450
Implications by Competency Area - Systems 450
Architecture and Organization (AR) 450
Data Management (DM) 450
Networking and Communication (NC) 450
Operating Systems (OS) 451
Systems Fundamentals (SF) 451
Security (SEC) 451
Implications by Competency Area - Applications 451
Artificial Intelligence (AI) 451
Graphics and Interactive Techniques (GIT) 451
Human-Computer Interaction (HCI) 452
Specialized Platform Development (SPD) 452

Implications for Crosscutting Areas 452
Society, Ethics, and the Profession (SEP) 453
Implications for the Curriculum 453
Acknowledgments 455
Organization 455
Reviewers 455
Contributors 457

Executive Summary

CS2023 is the latest version of computer science curricular guidelines, produced by a joint task force of the ACM, IEEE Computer Society, and AAAI. The following is a summary of significant issues of the day and how they have been addressed in CS2023 curricular guidelines:

- **The discipline continues to evolve.** The *Body of Knowledge* consisting of seventeen knowledge areas has been revised and updated.
- **The discipline continues to grow.** Topics that every graduate must know have been circumscribed as CS Core and kept to a minimum. Topics recommended for in-depth study have been labeled KA Core.
- **It is increasingly difficult for programs to be all things to all people.** Programs can now select the knowledge areas on which to focus. The knowledge areas, when coherently chosen, define the competency area(s) of the program.
- **Societal and ethical concerns have risen sharply.** The Society, Ethics, and the Profession (SEP) knowledge area is now an integral part of most knowledge areas of the curriculum.
- **The role of mathematics has increased.** Additional hours have been allocated to mathematics and flexibility has been provided for coverage of the requirements in the curriculum.
- **The need for professional dispositions is increasingly being recognized.** Professional dispositions appropriate for each knowledge area have been listed and justified.
- **Interest is growing among educators in a competency model of the curriculum.** A *Competency Framework* has been provided for programs to create their own competency model of the curriculum tailored to local needs.
- **Generative AI is poised to impact computer science education.** A chapter has been included that addresses how *Generative AI* could propel further innovation in computer science education.

The curricular guidelines build towards the *Characteristics of Graduates* enumerated in the report and take into consideration the *Challenges and Opportunities for Computer Science Education* identified in the report. The guidelines have been supplemented with articles on *Pedagogical Considerations* and *Curricular Practices*.

The process used by the CS2023 task force has been collaborative (over ninety task force members), international (six continents), data-driven (five large- and over seventy small-scale surveys) and transparent (csed.acm.org). Community engagement included numerous conference panels and presentations along with regular postings to over a dozen ACM Special Interest Group (SIG) mailing lists and the full membership of AAAI. The iterative process has included at least two review and revision cycles for most of the knowledge areas. The resulting curricular guidelines are the culmination of three years of effort using the outlined *Principles and Processes*.

Overview

1. Introduction to CS2023
2. Introduction to Knowledge Model
3. Introduction to Competency Framework

Introduction to CS2023

History

Several successive curricular guidelines for computer science have been published over the years as the discipline has continued to evolve. They are identified here.

- Curriculum 68 [1]: The first curricular guidelines were published by the Association for Computing Machinery (ACM) over 50 years ago as a classification of subject areas and courses.
- Curriculum 78 [2]: The curriculum was revised and presented in terms of core and elective courses.
- Computing Curricula 1991 [3]: The ACM teamed up with the Institute of Electrical and Electronics Engineers – Computer Society (IEEE-CS) for the first time to produce revised curricular guidelines.
- Computing Curricula 2001 [4]: For the first time, the guidelines focused only on Computer Science, with other disciplines such as computer engineering and software engineering being spun off into their own distinct curricular guidelines.
- Computer Science Curriculum 2008 [5]: This was presented as an interim revision of Computing Curricula 2001.
- Computer Science Curricula 2013 [6]: This was the most recent version of the curricula published by the ACM and IEEE-CS.

CS2023 is the latest revision of computer science curricular guidelines. It is a joint effort among the ACM, IEEE-CS, and, for the first time, the Association for the Advancement of Artificial Intelligence (AAAI).

Since 2013, the focus of curricular design has moved from what is taught (a knowledge model) to what is learned (a competency model). All prior versions of computer science guidelines used a knowledge model where related topics were grouped into a knowledge unit, and related knowledge units were grouped into a knowledge area. Computer Science Curricula Guidelines 2013 [6] contained 163 knowledge units grouped into 18 knowledge areas. Learning outcomes were identified for each knowledge unit. Distinction was made between core topics that every computer science graduate must know and elective topics that were considered optional. Core topics were further divided into Tier 1 topics that were to be covered completely and Tier 2 topics, at least 80% of which had to be covered.

Some early efforts to design a competency model of a curriculum were for Software Engineering [14] and Information Technology [7]. The broader Computing Curricula CC2020 report [8] proposed a competency model for various computing disciplines, including Computer Science, Information Systems, and Data Science. Competency models followed for Information Systems [9], Associate-degree CyberSecurity [13] and Data Science [10].

A knowledge model with its initial emphasis on content and a competency model with its primary emphasis on outcomes are complementary views of the same learning continuum. For computer science, neither model is a substitute for the other. The two models complement each other and work better together than apart. So, the CS2023 task force has both revised the CS2013 knowledge model [6] and proposed a framework for the competency model that maintains consistency with it [15].

Other recent model undergraduate curricula for computer science include that of the All India Council for Technical Education [11] and the "101 plan" of the Ministry of Education in China [16]. Similarly, professional bodies have drafted curricular guidelines on specific areas of computer science such as parallel and distributed computing [12].

This report limits itself to computer science curricula. But a holistic view requires consideration of the interrelatedness of computer science with other computing disciplines such as Software Engineering, Security, and Data Science. For an overview of the landscape of computing education, please see the section "Computing Interrelationships" in the CC 2020 report [8: pp. 29-30].

Vision Statement

The vision for CS2023 curricular revision includes the following:

- An updated knowledge model of the computer science curriculum—explained in the chapter *Introduction to Knowledge Model* in this section, (Section 1);
- An appropriate competency model for computer science—explained in the chapter *Introduction to Competency Framework* in this section;
- Consistency between the knowledge model and the competency model—as explained in [15];
- Well-researched articles by experts on curricular practices—included in Section 4;
- A live online version of the curriculum in addition to a hardcopy version—presented at the csed.acm.org website.

References

[1] Atchison, W. F., Conte, S. D., Hamblen, J. W., Hull, T. E., Keenan, T. A., Kehl, W. B., McCluskey, E. J., Navarro, S. O., Rheinboldt, W. C., Schweppe, E. J., Viavant, W., and Young, D. "Curriculum 68: Recommendations for academic programs in computer science." *Communications of the ACM*, 11, 3 (1968): 151-197.

[2] Austing, R. H., Barnes, B. H., Bonnette, D. T., Engel, G. L., and Stokes, G. "Curriculum '78: Recommendations for the undergraduate program in computer science." *Communications of the ACM*, 22, 3 (1979): 147-166.

[3] ACM/IEEE-CS Joint Curriculum Task Force. "Computing Curricula 1991." (New York, USA: ACM Press and IEEE Computer Society Press, 1991).

[4] ACM/IEEE-CS Joint Curriculum Task Force. "Computing Curricula 2001 Computer Science." (New York, USA: ACM Press and IEEE Computer Society Press, 2001).

[5] ACM/IEEE-CS Interim Review Task Force. "Computer Science Curriculum 2008: An interim revision of CS 2001." (New York, USA: ACM Press and IEEE Computer Society Press, 2008).

[6] ACM/IEEE-CS Joint Task Force on Computing Curricula. "Computing Science Curricula 2013." (New York, USA: ACM Press and IEEE Computer Society Press, 2013).

[7] Sabin, M., Alrumaih, H., Impagliazzo, J., Lunt, B., Zhang, M., Byers, B., Newhouse, W., Paterson, W., Tang, C., van der Veer, G. and Viola, B. Information Technology Curricula 2017: Curriculum Guidelines for Baccalaureate Degree Programs in Information Technology. Association for Computing Machinery, New York, NY, USA, (2017).

[8] Clear, A., Parrish, A., Impagliazzo, J., Wang, P., Ciancarini, P., Cuadros-Vargas, E., Frezza, S., Gal-Ezer, J., Pears, A., Takada, S., Topi, H., van der Veer, G., Vichare, A., Waguespack, L. and Zhang, M. Computing Curricula 2020 (CC2020): Paradigms for Future Computing Curricula. Technical Report. Association for Computing Machinery / IEEE Computer Society, New York, NY, USA, (2020).

[9] Leidig, P. and Salmela, H. A Competency Model for Undergraduate Programs in Information Systems (IS2020). Technical Report. Association for Computing Machinery, New York, NY, USA, (2021).

[10] Danyluk, A. and Leidig, P. Computing Competencies for Undergraduate Data Science Curricula (DS2021). Technical Report. Association for Computing Machinery, New York, NY, USA, (2021).

[11] https://iiitd.ac.in/sites/default/files/docs/aicte/AICTE-CSE-Curriculum-Recommendations-July2022.pdf, accessed July 2023.

[12] Prasad, S. K., Estrada, T., Ghafoor, S., Gupta, A., Kant, K., Stunkel, C., Sussman, A., Vaidyanathan, R., Weems, C., Agrawal, K., Barnas, M., Brown, D. W., Bryant, R., Bunde, D. P., Busch, C., Deb, D., Freudenthal, E., Jaja, J., Parashar, M., Phillips, C., Robey, B., Rosenberg, A., Saule, E., Shen, C. 2020. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version II-beta, Online: http://tcpp.cs.gsu.edu/curriculum/, 53 pages, accessed March 2024.

[13] https://ccecc.acm.org/files/publications/Cyber2yr2020.pdf, accessed July 2023.

[14] https://www.computer.org/volunteering/boards-and-committees/professional-educational-activities/software-engineering-competency-model, accessed July 2023.

[15] Kumar, A. N., Becker, B. A., Pias, M., Oudshoorn, M., Jalote, P., Servin, C., Aly, S.G., Blumenthal, R. L., Epstein, S. L., and Anderson, M.D. 2023. A Combined Knowledge and Competency (CKC) Model for Computer Science Curricula. *ACM Inroads* 14, 3 (September 2023), 22–29. https://doi.org/10.1145/3605215

[16] Liu, Y., Xiang, Q., Chen, J., Zhang, M., Xu, J., and Luo, Y. Undergraduate Computer Science Education in China. *ACM Inroads*, Vol 15, 1, March 2024, 28-36.

Introduction to Knowledge Model

Definitions and Terminology

A **knowledge model** of a curriculum is structured as a set of knowledge areas:

Knowledge model = { Knowledge areas }

A **knowledge area** is a set of related knowledge units:

Knowledge area = { Knowledge units }

A **knowledge unit** is a set of related topics and a set of learning outcomes for those topics:

Knowledge unit = { Topics } + { Learning outcomes }

Learning outcomes are used for assessment. Each topic in a knowledge unit is categorized as either core or elective:

Topic $\in$ { core } $\cup$ { elective }

Core topics are topics that *every* graduate **must** know. Every curriculum is typically expected to cover all the core topics.

Elective topics are those that are not required of every graduate. Nevertheless, they complement the coverage of core topics. In addition to covering all the core topics, a curriculum is expected to cover a considerable percentage of elective topics.

Students may be expected to demonstrate proficiency in topics at different **skill levels**. Typically, Bloom's taxonomy [8] is used to describe skill levels. The instructional time needed to cover a topic is determined by the skill level, e.g., instructing how to apply a concept may take longer than instructing how to explain the concept.

The expected skill levels and, therefore, the time needed to cover topics are salient because they determine how many topics can be packaged into a typical **course** and how many courses are needed in a **curriculum** to cover all the required topics. Thus, the list of core topics and the skill levels at which students must demonstrate proficiency in those topics determine the minimum size of a curriculum.

CS2023 Knowledge Model

Knowledge areas: The CS2023 knowledge model consists of 17 knowledge areas, listed in alphabetical order of their abbreviation:

• Artificial Intelligence (AI) • Algorithmic Foundations (AL) • Architecture and Organization (AR) • Data Management (DM) • Foundations of Programming Languages (FPL) • Graphics and Interactive Techniques (GIT)	• Networking and Communication (NC) • Operating Systems (OS) • Parallel and Distributed Computing (PDC) • Software Development Fundamentals (SDF) • Software Engineering (SE) • Security (SEC) • Society, Ethics, and the Profession (SEP)

● Human-Computer Interaction (HCI) ● Mathematical and Statistical Foundations (MSF)	● Systems Fundamentals (SF) ● Specialized Platform Development (SPD)

Details of the knowledge areas are in Section 3 of this report. In CS2023, knowledge areas have been expanded to include professional dispositions in addition to knowledge units:

Knowledge area = { Knowledge units } + { Professional dispositions }

Professional dispositions are malleable values, beliefs, and attitudes that enable consistent behaviors desirable in the workplace, e.g., *persistent*, *self-directed*. When dispositions seem indistinguishable from skills (e.g., *communicative*, *collaborative*), they refer to the willingness and intent to apply the skills to complete a task. They are sought by employers and are essential for succeeding in the workplace. Most recent curricular guidelines have emphasized the need for and the value of professional dispositions in computing education, including Information Technology 2017 [4], Computing Curricula 2020 [5], Information Systems 2020 [6], and Data Science 2021 [7]. These guidelines have proposed competency models of the curricula in which knowledge, skills, and professional dispositions needed to carry out a task are bundled together.

Dispositions vary by knowledge area. Some dispositions are more important at certain stages in a student's development than others, e.g., being *persistent* is essential at introductory levels, whereas being *self-directed* is expected at advanced levels of study. Group projects call for *collaborative* disposition whereas mathematical foundations demand *meticulous* disposition. So, associating dispositions with knowledge areas instead of individual tasks makes it easier for educators to repeatedly and consistently promote dispositions during the accomplishment of tasks relevant to the knowledge area. In CS2023, the professional dispositions most relevant for each knowledge area have been listed.

In CS2023, core topics are categorized as either CS Core or KA Core. Elective topics are renamed Non-core:

Topic $\in$ { CS Core } $\cup$ { KA Core } $\cup$ { Non-core }

CS (Computer Science) Core topics are topics that *every* computer science graduate **must** know. Every computer science curriculum is expected to cover all the CS Core topics. **KA (Knowledge Area) Core** topics are topics *recommended* for more in-depth study. Additional nuances to the concept of KA Core topics are:

- Topics in Mathematical and Statistical Foundations (MSF) knowledge area are designated as KA Core to indicate that they are required for KA Core topics in other knowledge areas, e.g., many KA Core topics in statistics are needed for KA Core topics in the Artificial Intelligence (AI) knowledge area.
- Some knowledge areas have more than one subset of KA Core topics, e.g., Specialized Platform Development (SPD) has KA Core topics on web platforms, mobile platforms, embedded platforms, etc. with minimal overlap among them. The hours for these disparate KA Core topics are not to be considered additive for the knowledge area.

Over 50% of the knowledge units in the 17 knowledge areas of CS2023 include CS Core topics and over 75% of the knowledge units include KA Core topics. A curriculum may choose to focus on the KA Core topics of some knowledge areas in greater depth/breadth than others. The subset of knowledge

areas on which a curriculum is focused, when coherently chosen, defines the **competency area**(s) of the curriculum.

Competency area ⊆ Knowledge model

Three *representative* competency areas are presented in CS2023:

- **Software** Development – the knowledge areas that prepare a student to be a journeyman programmer. These include Software Development Fundamentals (SDF), Algorithmic Foundations (AL), Foundations of Programming Languages (FPL), and Software Engineering (SE) knowledge areas.
- **Systems** Development – the knowledge areas that prepare a student to provide essential services including non-functional requirements. These include Systems Fundamentals (SF), Architecture and Organization (AR), Operating Systems (OS), Parallel and Distributed Computing (PDC), Networking and Communication (NC), Security (SEC), and Data Management (DM).
- **Applications** Development – the knowledge areas that prepare a student with problem-specific or solution-specific knowledge in addition to software development. These include Graphics and Interactive Techniques (GIT), Artificial Intelligence (AI), Specialized Platform Development (SPD), Human-Computer Interaction (HCI), Security (SEC), and Data Management (DM).

Society, Ethics, and the Profession (SEP) and Mathematical and Statistical Foundations (MSF) are part of all competency areas. Note that the Software competency area is a prerequisite of the other two competency areas. This list of competency areas is meant to be neither prescriptive nor comprehensive. Other competency area(s) may be based on institutional mission and local needs. Examples include *Computing for the social good, Scientific computing,* and *Secure computing*.

Skill levels: Four **skill levels** were adopted for use in CS2023: Explain, Apply, Evaluate, and Develop. They are loosely aligned with the revised Bloom's taxonomy [8] as shown in Table 1. *Explain* is the prerequisite skill for the other three levels. The verbs corresponding to each of the four skill levels were adopted from the work of ACM and CCECC [9].

<table>
<tr><th>Revised Bloom's Taxonomy</th><th>Skill level with applicable verbs</th></tr>
<tr><td>Remember</td><td rowspan="2">Explain: define, describe, discuss, enumerate, express, identify, indicate, list, name, select, state, summarize, tabulate, translate</td></tr>
<tr><td>Understand</td></tr>
<tr><td>Apply</td><td rowspan="2">Apply: backup, calculate, compute, configure, debug, deploy, experiment, install, iterate, interpret, manipulate, map, measure, patch, predict, provision, randomize, recover, restore, schedule, solve, test, trace, train, virtualize</td></tr>
<tr><td rowspan="2">Analyze</td></tr>
<tr><td></td></tr>
</table>

Evaluate	**Evaluate:** analyze, compare, classify, contrast, distinguish, categorize, differentiate, discriminate, order, prioritize, criticize, support, decide, recommend, assess, choose, defend, predict, rank
Create	**Develop:** combine, compile, compose, construct, create, design, generalize, integrate, modify, organize, plan, produce, rearrange, rewrite, refactor, write

Table 1. Skill levels and corresponding verbs and levels in revised Bloom's taxonomy.

In CS2023, desired skill levels were identified for CS Core and KA Core topics. The desired skill levels were then used to estimate the time needed for the instruction of CS Core and KA Core topics. These details are presented in tabular format in Section 3 of this report. The skill levels identified for core topics should be treated as recommended, not prescriptive.

The time needed to cover CS Core and KA Core topics is expressed in terms of **instructional hours**. Instructional hours are hours spent in the classroom imparting knowledge regardless of the pedagogy used. Students are expected to spend additional time after class practicing related skills and exercising professional dispositions.

In CS2023, CS Core topics at the desired skill levels are estimated to need 270 hours of instructional time. Since every computer science graduate must know CS Core topics, every computer science curriculum is expected to include all 270 hours of instruction. While CS Core topics set the *minimum* for a computer science curriculum, a typical curriculum is expected to include many more KA Core topics in a competency area of choice as well as complementary Non-core topics.

A curriculum is structured in terms of courses, not knowledge areas. A knowledge area is not necessarily a course. Many courses may be carved out of a knowledge area and a course may contain topics from multiple knowledge areas. In CS2023, recommendations have been provided in each knowledge area for **packaging course(s)** from its topics. For packaging purposes, a course is assumed to meet for around 40 instructional hours.

The number of courses in a computer science curriculum varies among educational institutions. In acknowledgment of this variety, in CS2023, **curricular packaging** recommendations have been provided in Section 3 for programs that require 8, 10, 12, and 16 computer science courses. Both course and curricular packaging recommendations are suggestive, not prescriptive. It is expected that curriculum designers will adapt them to suit their local needs, resources, and constraints. The packaging recommendations may also be used to compare courses and curricula of different educational institutions.

Structure of CS2023 Knowledge Area Explained

Each of the 17 knowledge areas in Section 3 is structured as follows.

- **A Preamble** that describes the knowledge area and includes:
    - Changes in the knowledge area since CS2013;
    - An optional overview of the various knowledge units in the knowledge area.
- A table listing CS Core and KA Core hours assigned to the knowledge area.

- A listing of knowledge units, each of which in turn consists of:
    - CS Core, KA Core and Non-core topics, enumerated, and
    - CS Core and KA Core illustrative learning outcomes, enumerated.
- Professional dispositions most relevant to the knowledge area.
- Mathematics requirements for the knowledge area—both required and desired.
- Suggestions for packaging courses from the knowledge area.
- The committee members who reviewed and revised the knowledge area recommendations.

Table of Core Hours: The table lists the number of CS Core and KA Core hours assigned to each knowledge unit in the knowledge area. Where applicable, the table also lists the CS/KA Core hours shared with other knowledge areas.
Knowledge units: Each knowledge unit has an abbreviated name in addition to its complete name. For example, the abbreviated name of the knowledge unit "Computational Models and Formal Languages" in Algorithmic Foundations (AL) is "Models." The abbreviated name is used to refer to the knowledge unit in the rest of the document, as in "AL-Models."
Topics: Within each knowledge unit, topics are categorized as CS Core, KA Core or Non-core. They are consecutively numbered through all three categories so that each topic can be uniquely identified. The enumeration should not be construed as implying any order or dependency among the topics. The recommended syntax for referring to a topic is:
<Knowledge area abbreviation>--<Knowledge unit abbreviation>: Decimal.alphabetic.roman
For example, "Local vs global solutions" in the following is **AI-Search: 3.c.i**:

3. Heuristic graph search for problem-solving
    c. Local minima and the search landscape
        i. Local vs global solutions

Even though knowledge areas are groupings of related topics, they are not insular. Making connections among the various knowledge areas is an essential part of maturing as a computer science graduate. So, whenever possible, relationships between topics in different knowledge areas have been pointed out with "See also:" annotation in CS2023. For example, for the topic "Attacks and antagonism" in OS-Protection, the reader is directed to see also SEC-Foundations.
Typically, when a topic appears in two knowledge areas, the two knowledge areas present different perspectives on the topic. When one knowledge area is the primary repository for a topic and the other knowledge area refers to the topic from the repository, "See also" annotations are skipped in the knowledge area that serves as the primary repository. The knowledge areas that serve as primary repositories include Software Development Fundamentals (SDF – all the introductory programming topics), Algorithmic Foundations (AL – all the algorithms), Mathematical and Statistical Foundations (MSF), and Society, Ethics, and the Profession (SEP).
Learning Outcomes: In each knowledge unit, learning outcomes have been identified primarily for CS Core and KA Core topics. Furthermore, they have been listed under CS Core and KA Core subtitles and have been consecutively numbered for ease of referencing. The learning outcomes are meant to be descriptive, not prescriptive. So, they have been re-named **Illustrative Learning Outcomes**.

Professional Dispositions: The professional dispositions most relevant to each knowledge area have been listed, along with a brief explanation of why they are relevant to the knowledge area. These dispositions are desirable for most of the tasks that require a knowledge of this knowledge area.
The list of dispositions is not meant to be comprehensive, but rather, to provide a starting point for adaptation. Dispositions, by definition, vary with the context—type of institution, academic level of students, demographic profile of the student body, etc. Educators will want to adapt this list of dispositions to suit their local context.
Dispositions are not meant to be used to exclude students based on their background or demographics. Instead, they are listed in the spirit that explicit acknowledgment of their role will provide impetus for educators to foster them in their curricula and level the playing field for the success of all computer science graduates regardless of their background or prior preparation.
Mathematics requirements: The mathematics requirements have been individually noted for each knowledge area. Where applicable, the requirements have been further categorized as either required or desirable. This serves several purposes: 1) it acknowledges the essential role played by mathematics for success in computer science; 2) it provides for the possibility of covering the mathematics required for a computer science course within the course instead of as a prerequisite gatekeeper mathematics course; and 3) it helps educators provide access ramps for computer science students underprepared in mathematics.
Course packaging suggestions: Each course has been specified in terms of knowledge units, both from within and outside the knowledge area. Instructional hours have been suggested for each knowledge unit. These hours represent the weight suggested for the knowledge unit in a *typical* course of around 40 instructional hours. A course that meets for fewer or more hours may scale the hours accordingly and/or include fewer/more knowledge units in its coverage, as long as the course covers all the listed CS Core topics. Finally, objectives have been specified for each course that describe what a student must be able to do once the student has completed the course.
Committee: The committee that reviewed and revised the knowledge area has been listed along with its chair. The committee has typically included international experts from academia and industry who met regularly over the course of the curricular revision. In addition, non-committee experts who contributed significantly to the knowledge area have been listed as Contributors. Experts who reviewed the drafts of the knowledge area have been listed as Reviewers in the Acknowledgments section.

Changes since CS2013

The CS2023 knowledge model was developed by revising the CS2013 curricular guidelines [3]. Significant changes from CS2013 to CS2023 are described below in terms of the components of a knowledge model.
Knowledge Areas: Several CS2013 knowledge areas were renamed, either to better emphasize their focus or to incorporate changes in the area since 2013.

- Algorithms and Complexity (AL) as Algorithmic Foundations (AL)
- Discrete Structures (DS) as Mathematical and Statistical Foundations (MSF)
- Graphics and Visualization (GV) as Graphics and Interactive Techniques (GIT)
- Information Assurance and Security (IAS) as Security (SEC)

- Information Management (IM) as Data Management (DM)
- Intelligent Systems (IS) as Artificial Intelligence (AI)
- Platform Based Development (PBD) as Specialized Platform Development (SPD)
- Programming Languages (PL) as Foundations of Programming Languages (FPL)
- Social Issues and Professional Practice (SP) as Society, Ethics, and the Profession (SEP)

Computational Science (CN) from CS2013 was dropped as a knowledge area because it had very little computer science content that was not also included in other knowledge areas. In CS2013, it was allocated just one core hour for modeling and simulation. Modeling was considered and rejected as an alternative to Computational Science: while applying modeling is a crosscutting theme in computer science, studying modeling for its own sake more appropriately belongs in the CS + X space.
Given the increased role played by mathematics in computer science today, the mathematical component of computer science was expanded from Discrete Structures in CS2013 to also include probability, statistics, and linear algebra. The expanded knowledge area was renamed Mathematical and Statistical Foundations (MSF) to reflect this change.
Systems Fundamentals (SF) from CS2013 was considered for elimination, but ultimately retained since it provides a system-wide perspective not also available in any of the other systems knowledge areas, that is, Architecture and Organization (AR), Operating Systems (OS), Networking and Communication (NC) or Parallel and Distributed Computing (PDC).
Data Science was considered for inclusion as a new knowledge area in CS2023, but ultimately rejected since all of Data Science's computer science-specific topics are already covered by Artificial Intelligence (AI), Graphics and Interactive Techniques (GIT), Data Management (DM) and Mathematical and Statistical Foundations (MSF).
SEP Knowledge Unit: Given that the work of computer science graduates affects all aspects of everyday life, computer science as a discipline can no longer ignore or treat as incidental, social, ethical, and professional issues. In recognition of this pervasive nature and influence of computing, effort was made to include a separate knowledge unit on Society, Ethics, and the Profession (SEP) in every other knowledge area. Topics and learning outcomes at the intersection of the knowledge area and SEP were explicitly listed in the knowledge unit to help educators call attention to these issues across the curriculum.
Topics: In CS2023, as stated earlier, every topic has been provided a unique identifier so that any topic can be unambiguously referenced using the notation:
<Knowledge area abbreviation>--<Knowledge unit abbreviation>: Decimal.alphabetic.roman
Skill Levels: The three skill levels used in CS2013 have been expanded to four. The skill levels used in CS2013 were: Familiarity ("What do you know about this?"), Usage ("What do you know how to do?") and Assessment ("Why would you do that?"). In CS2023, in order to reflect the learner's thinking processes and actions rather than behaviors, verbs were used for skill levels instead of nouns: Familiarity was renamed Explain; and Assessment was renamed Evaluate. Usage was split into two: Apply and Develop. Apply was introduced as a skill level because of the increased emphasis placed on it in online courseware. It is also a skill level of increasing importance in light of the availability of generative AI for development tasks. The four skill levels, that is, Explain, Apply, Develop, and Evaluate, are loosely aligned with the revised Bloom's taxonomy [8] as shown earlier in Table 1.

Learning Outcomes: In CS2013, each learning outcome was labeled with a skill level. Since the skill level of a learning outcome can typically be inferred from the verb used in the learning outcome statement, in CS2023, skill level labels were dropped from learning outcomes. Since the learning outcomes in CS2023 are meant to be descriptive, not prescriptive, they have been renamed *Illustrative Learning Outcomes* to acknowledge that additional learning outcomes can be specified at other skill levels for each topic.

Professional Dispositions: CS2013 guidelines [3] emphasized the importance of dispositions in passing (Professional Practice, pp. 15-16). In addition to the knowledge model of CS2013, a framework for a competency model was also attempted in CS2023. A competency model demands a more integrated treatment of dispositions. So, in CS2023, professional dispositions appropriate for each knowledge area were identified.

Core Topics: In CS2013 [3], core topics were identified at two levels: Tier 1 accounting for 165 instructional hours and Tier 2 accounting for 143 hours. Computer science programs were expected to cover 100% of Tier 1 core topics and at least 80% of Tier 2 topics as shown in Figure 1. While proposing this scheme, CS2013 was mindful that the number of core hours has been steadily increasing in curricular recommendations, from 280 hours in CC2001 [1] to 290 hours in CS2008 [2] and 308 hours in CS2013 [3]. Accommodating the increasing number of core hours poses a challenge for computer science programs that may want to restrict the size of the program either by design or due to necessity.

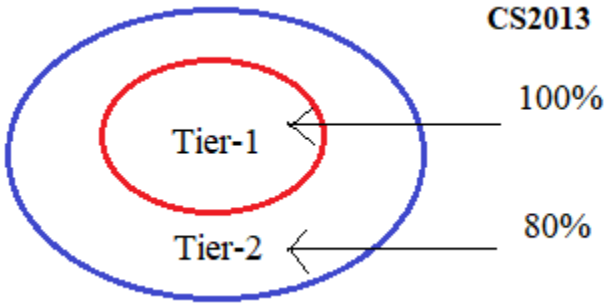


Figure 1: CS2013 Core Topics

In CS2023, instead of Tier 1 and Tier 2, core topics were categorized as CS Core and KA Core: every computer science program must cover all the CS Core topics. But a program may choose to cover KA Core topics in great detail in some knowledge areas and minimally or not at all in other knowledge areas as illustrated by highlighting in Figure 2. The result is a sunflower model that acknowledges that often, the design of curricula in smaller programs is dictated by curricular emphasis based on regional needs, the local availability of instructional expertise, and evolutionary history of the programs.

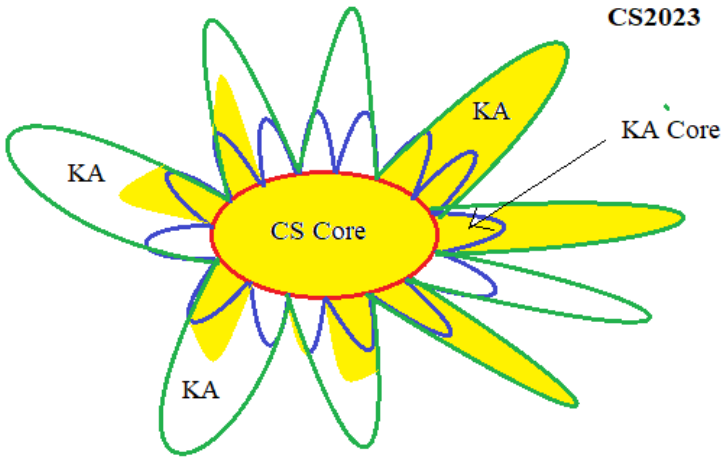


Figure 2: Sunflower model of core topics used in CS2023

Core hours: Table 2 shows how the distribution of core hours among the knowledge areas has changed from CS2013 to CS2023.

Knowledge Area	**CS2013**		**CS2023**	
	Tier-1	**Tier-2**	**CS Core**	**KA Core**
Artificial Intelligence (AI)	0	10	12	18
Algorithmic Foundations (AL)	19	9	32	32
Architecture and Organization (AR)	0	16	9	16
Data Management (DM)	1	9	10	26
Foundations of Programming Languages (FPL)	8	20	21	19
Graphics and Interactive Techniques (GIT)	2	1	4	70
Human-Computer Interaction (HCI)	4	4	8	16
Mathematical and Statistical Foundations (MSF)	37	4	55	145
Networking and Communication (NC)	3	7	7	24
Operating Systems (OS)	4	11	8	13
Parallel and Distributed Computing (PDC)	5	10	9	26
Software Development Fundamentals (SDF)	43	0	43	
Software Engineering (SE)	6	22	6	21
Security (SEC)	3	6	6	35
Society, Ethics, and the Profession (SEP)	11	5	18	14
Systems Fundamentals (SF)	18	9	18	8
Specialized Platform Development (SPD)	0	0	4	
Computational Science (CN)	1	0	Dropped	
Total	**165**	**143**	**270**	**N/A**

Table 2. Change in the distribution of core hours from CS2013 to CS2023.

Even though most knowledge areas in CS2023 contain a knowledge unit on Society, Ethics, and the Profession (SEP), core topics and hours for SEP issues were identified only in the Society, Ethics, and the Profession (SEP) knowledge area and not in the SEP knowledge units of other knowledge areas.

This omission is meant to give educators the flexibility to decide how to distribute the coverage of the core SEP topics across the various computer science courses in their curriculum.

Course and Curricular Packaging: In CS2013, course and curricular exemplars were included from various institutions. But exemplars encapsulate institutional context such as the level of preparedness of students, the availability of teaching expertise, the availability of pre- and co-requisite courses, etc. that hinders their adoption across institutions. So, in CS2023, canonical packaging of courses has been provided in terms of knowledge areas and knowledge units as was done in CC2001 [1].

Designing/Revising a Curriculum Based on the Knowledge Model

A process for designing or revising a computer science curriculum using the CS2023 knowledge model is as follows.

1. Identify the **competency area(s)** to be targeted by the curriculum based on local needs. Some representative competency areas are Software, Systems, and Applications.
2. Based on the competency area(s) identified in step 1, select the **knowledge areas** whose **KA Core** topics will be covered in greater depth, while considering the availability of local resources (instruction, laboratory, etc.).
3. For each knowledge area identified in step 2, start with one or more **course packaging** suggestions. For each course add/subtract/scale **knowledge units** as appropriate.
4. Within each knowledge unit identified in step 3:
    a. Ensure that all the **CS Core** topics are included;
    b. Maximize the KA Core topics covered;
    c. Eliminate duplicate topics shared with other courses in the curriculum;
    d. Identify the desired **skill level** for each core topic—use the skill levels recommended in the *Table of Core Topics* in Section 3 as a benchmark for comparison;
    e. Create an assessment plan for each course:
        i. Aggregate the **illustrative learning outcomes** of the knowledge units and topics covered by the course;
        ii. Adapt the learning outcomes to the skill levels identified in step 4d.
5. For the knowledge areas not selected for in-depth coverage in step 2 and the knowledge units not selected in step 3:
    a. Design coherent course(s) that cover all the **CS Core** topics in these knowledge areas/units;
    b. Eliminate duplicate topics shared with other courses in the curriculum;
    c. Identify the desired **skill level** for each core topic—use the skill levels recommended in the *Table of Core Topics* in Section 3 as a benchmark for comparison;
    d. Create an assessment plan for each course:

i. Aggregate the **illustrative learning outcomes** of the knowledge units and topics covered by the course;

ii. Adapt the learning outcomes to the skill levels identified in step 5d.

6. Complete the curriculum design loop:
    a. Aggregate the course assessment plans from steps 4e and 5d;
    b. Reconcile the aggregation with the competency area(s) identified in step 1.
7. Sequence the courses in the curriculum with prerequisites and corequisites based on the following:
    a. **Mathematics requirements** listed for the courses identified in steps 4 and 5;
    b. Software competency is a prerequisite for most other competency areas.

References

1. ACM/IEEE-CS Joint Curriculum Task Force. “Computing Curricula 2001 Computer Science.” (New York, USA: ACM Press and IEEE Computer Society Press, 2001).
2. ACM/IEEE-CS Interim Review Task Force. “Computer Science Curriculum 2008: An interim revision of CS 2001.” (New York, USA: ACM Press and IEEE Computer Society Press, 2008).
3. ACM/IEEE-CS Joint Task Force on Computing Curricula. “Computing Science Curricula 2013.” (New York, USA: ACM Press and IEEE Computer Society Press, 2013).
4. Sabin, M., Alrumaih, H., Impagliazzo, J., Lunt, B., Zhang, M., Byers, B., Newhouse, W., Paterson, W., Tang, C., van der Veer, G. and Viola, B. Information Technology Curricula 2017: Curriculum Guidelines for Baccalaureate Degree Programs in Information Technology. Association for Computing Machinery, New York, NY, USA, (2017).
5. Clear, A., Parrish, A., Impagliazzo, J., Wang, P., Ciancarini, P., Cuadros-Vargas, E., Frezza, S., Gal-Ezer, J., Pears, A., Takada, S., Topi, H., van der Veer, G., Vichare, A., Waguespack, L. and Zhang, M. Computing Curricula 2020 (CC2020): Paradigms for Future Computing Curricula. Technical Report. Association for Computing Machinery / IEEE Computer Society, New York, NY, USA, (2020).
6. Leidig, P. and Salmela, H. A Competency Model for Undergraduate Programs in Information Systems (IS2020). Technical Report. Association for Computing Machinery, New York, NY, USA, (2021).
7. Danyluk, A. and Leidig, P. Computing Competencies for Undergraduate Data Science Curricula (DS2021). Technical Report. Association for Computing Machinery, New York, NY, USA, (2021).
8. Anderson, L. W. and Krathwohl, D. R., eds. (2001). A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives. New York: Longman. ISBN 978-0-8013-1903-7.
9. Bamkole, A., Geissler, M., Koumadi, K., Servin, C., Tang, C., and Tucker, C. S., "Bloom’s for Computing: Enhancing Bloom's Revised Taxonomy with Verbs for Computing Disciplines". The Association for Computing Machinery. (January 2023). https://ccecc.acm.org/files/publications/Blooms-for-Computing-20230119.pdf, accessed March 2024.

Introduction to Competency Framework

Competence Model - Definitions and Terminology

Competency is defined as the sum of knowledge, skills, and dispositions in IT2017 [1], wherein dispositions are defined as *cultivable behaviors desirable in the workplace* [3].

Competency = Knowledge + Skills + Dispositions *in context*

In CC 2020 [2], competency was further elaborated as the sum of the three within the performance of a task. Instead of the additive model of IT 2017, CC2020 defined competency as the intersection of the three:

Competency = Knowledge ∩ Skills ∩ Dispositions

In CS2023, competency is treated as a point in a 3D space with knowledge, skills, and dispositions as the three axes of the space (Figure 1): all three are required for proper execution of a task.

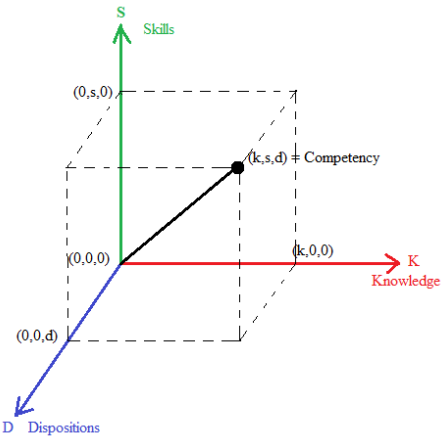


Figure 1. Competency as a point in a 3D space [3].

For a given learner at a given moment and a given task, competency is represented as a point in this 3D space. For a set of tasks, the competency of a learner is a point cloud in this 3D space.

A competency model of a curriculum is a set of competency specifications:

Competency Model = { Competency Specifications }

A competency specification consists of a competency statement and enumeration of the knowledge, skills, and dispositions needed to complete the task stated in the competency statement.

Competency Specification = Competency Statement + Knowledge + Skills + Dispositions

An ITiCSE working group has tried to design sample competency statements for computer science [4]. A process has also been proposed for converting a knowledge model to a competency model for computer science [5].

CS2023 Competency Model

In the specification of a competency, the task is the sole independent variable. The knowledge, skills, and dispositions needed to complete a task depend on the task and vary from one task to another. So, in CS2023, the description of the task was separated from the competency statement in a competency specification:

Competency Specification = Task + Competency Statement + Knowledge + Skills + Dispositions

In a competency specification:

- a task is a statement of what someone in a given role and context might be expected to do and is expressed in layman terms;
- the competency statement describes how a graduate might go about completing the task and is expressed in technical terms;
- knowledge is specified in terms of knowledge units in knowledge areas in the *Body of Knowledge* (Section 3);
- Skills are one or more of *Explain*, *Apply*, *Evaluate,* and *Develop*; and
- Dispositions are one or more of those identified as appropriate for the knowledge areas needed to complete the task.

The format used for competency specifications in CS2023 is as follows.

- **Task:** *What* someone in a given role and context might be expected to do.
- **Competency statement:** *What* a graduate might bring to bear in terms of knowledge and skills to attempt the task.
- **Required knowledge:** List of knowledge area-knowledge unit pairs needed to complete the task.
- **Required skills:** Explain / Apply / Evaluate / Develop
- **Desirable professional dispositions:** Adaptable / Collaborative / Inventive / Meticulous / Persistent / Proactive / Responsive / Self-Directed / Other

The first step in designing a competency model is to identify the tasks for which competency specifications will be written. But computer science being a general-purpose discipline of the study of solving problems with computers, the range of tasks for which it prepares graduates is vast. Not only is a comprehensive enumeration of tasks intractable, but it is also further complicated by the variability of the granularity and composition of tasks. Finally, local customization is an essential ingredient of competency models. So, a complete competency model of computer science, even if it were tractable, would be of limited off-the-shelf utility. Given these reasons, in CS2023, a competency framework has been proposed instead of an exhaustive competency model. The framework consists of 1) a framework for systematically identifying tasks (described next); 2) a format for competency specifications (see previous box); and 3) an algorithm to design a customized competency model using the competency framework (described last in this Introduction to Competency Framework). Examples of sample tasks and competency specifications are included in Section 3.

CS2023 Framework for Systematically Identifying Tasks

The framework for systematically identifying tasks focuses on atomic tasks that can be combined to create compound tasks to suit local needs. The framework consists of three dimensions: component, activity, and constraint. In a task statement, the component is typically the noun, the activity the verb, and the constraint either an adjective or adverb.

The framework is elaborated for the three competency areas proposed in the knowledge model, that is, Software, Systems, and Applications. The following is an initial set of **components** in these three competency areas.

- Software: Program, algorithm, and language/paradigm
- Systems: Processor, storage, communication, architecture, I/O, data, and service
- Applications: Input, computation, output, and platform

An initial list of **activities** includes design, develop, document, evaluate, maintain, improve, humanize, and research. While most of the activities are self-explanatory, humanize refers to activities that address issues of society, ethics, and the profession, and research refers to activities that study theoretical underpinnings.

Constraints are categorized as follows:

- Problem constraints, e.g., task size (small versus large), problem (well-defined versus open-ended), task agent (solo versus in a team);
- Solution constraints, e.g., functional and non-functional requirements such as performance, availability, security, scalability, efficiency, reliability, cost; and
- Implementation constraints, e.g., parallel, distributed, virtualized.

The components, activities, and constraints listed above are representative, not prescriptive, or comprehensive. They may be visualized in three-dimensional space as shown in Figure 1 for the three competency areas. In the figure, the three axes use nominal scale, with no ordinality implied.

Each atomic task is a point in the three-dimensional space of component x activity x constraint. At the bottom-right of Figure 1 are the following three tasks mapped on software competency area:

- Develop a program for an open-ended problem (blue star);
- Evaluate the efficiency of a parallel algorithm (green star);
- Research language features for writing secure code (red star).

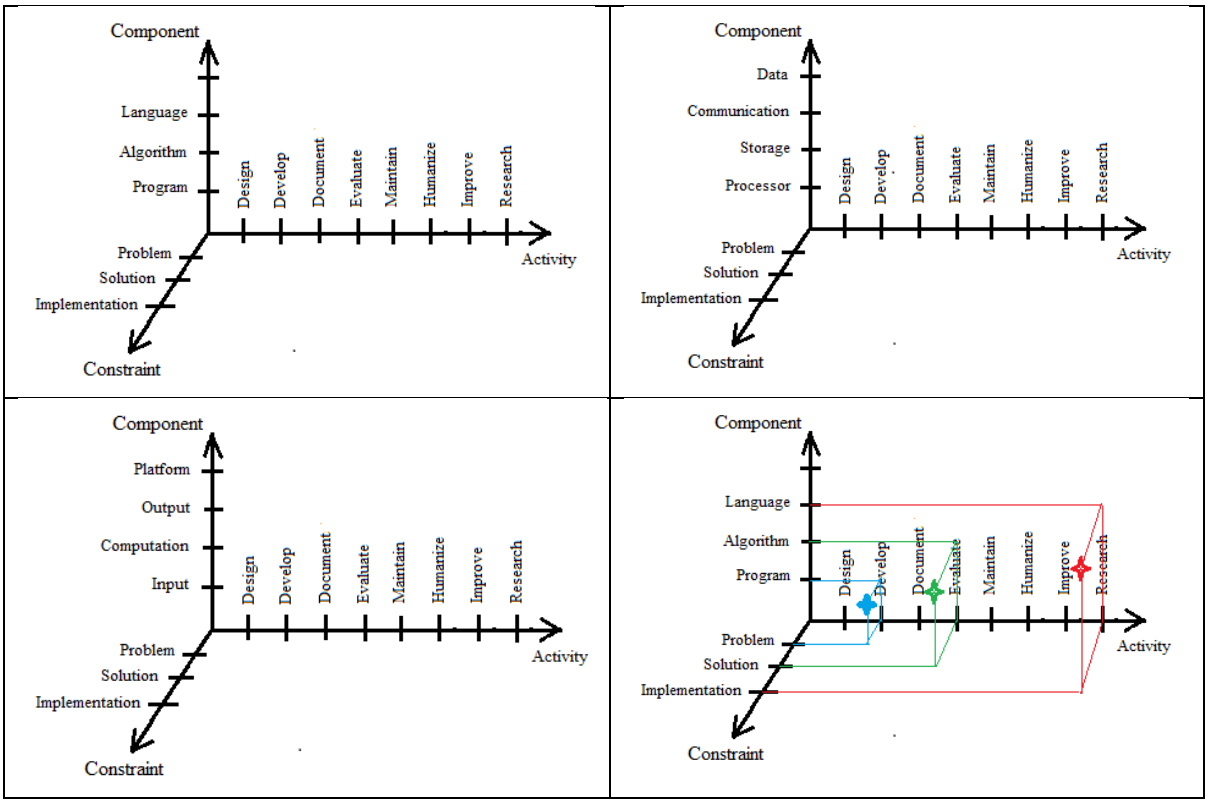


Figure 1: Task design space: Software competency area (top-left); Systems competency area (top-right); Applications competency area (bottom-left) and three tasks mapped on software competency area (bottom-right)

The framework is offered as a starting point for **generating** atomic tasks, not "classifying" them. One may want to add other components, activities, and constraints to the framework as appropriate for their local needs. It is expected that most competency specifications will be written for compound tasks created by combining two or more atomic tasks, e.g., "Design, implement, and document a parallelized scheduling program."

Designing/Revising a Curriculum Using the Competency Framework

A process for designing or revising a computer science curriculum using the CS2023 competency framework is as follows.

1. Identify the **competency area(s)** to be targeted by the curriculum in consultation with local stakeholders (academics, industry representatives, policy makers, etc.).
2. For each targeted competency area, use the component x activity x constraint task design space of the competency area to identify the atomic tasks for which the curriculum must prepare graduates – the task design spaces of Software, Systems, and Applications competency areas are shown in Figure 1. The targeted atomic tasks will each be a point in the three-dimensional task design space as illustrated at the bottom-right in Figure 1.
3. Create compound tasks by combining two or more related atomic tasks.

4. Write a **competency specification** for each atomic or compound task identified in the previous two steps. In each specification,
    a. Exhaustively identify all the knowledge areas and knowledge units needed for the task.
    b. Identify the **skills** needed to complete the task.
        i. For the tasks that require CS and KA Core topics, re-design the tasks to require the skill levels recommended in the *Table of Core Topics* in Section 3.
    c. Identify the **dispositions** needed for the task – use the dispositions listed for the knowledge areas in step 4a as the starting point.

    The set of competency specifications for all the identified atomic/compound tasks constitutes the **competency model** of the curriculum.
5. Aggregate the required **knowledge areas** and **knowledge units** identified in the competency specifications of the competency model.
    a. Eliminate duplicate knowledge unit entries in the aggregation.
    b. Include additional knowledge units that are prerequisites for the knowledge units listed in the aggregation.
    c. Include additional knowledge units in the aggregation as necessary to ensure that all the **CS Core** topics are covered.
6. Package the knowledge units in the aggregation into courses. Adapt the **course packaging suggestions** of knowledge areas.
7. Sequence the courses to form a curriculum with prerequisites and co-requisites based on the following:
    a. **Mathematics requirements** listed for the courses identified in step 6;
    b. Software competency is a prerequisite for most other competency areas.
8. Complete the curriculum design loop.
    a. Verify that the knowledge areas aggregated in step 5 include all the knowledge areas that are part of the competency area(s) identified in Step 1.
    b. Add to the list of tasks identified in step 2, the tasks supported by the knowledge units added in steps 5b and 5c.

References

1. Sabin, M., Alrumaih, H., Impagliazzo, J., Lunt, B., Zhang, M., Byers, B., Newhouse, W., Paterson, W., Tang, C., van der Veer, G. and Viola, B. Information Technology Curricula 2017: Curriculum Guidelines for Baccalaureate Degree Programs in Information Technology. Association for Computing Machinery, New York, NY, USA, (2017).
2. Clear, A., Parrish, A., Impagliazzo, J., Wang, P., Ciancarini, P., Cuadros-Vargas, E., Frezza, S., Gal-Ezer, J., Pears, A., Takada, S., Topi, H., van der Veer, G., Vichare, A., Waguespack, L. and Zhang, M. Computing Curricula 2020 (CC2020): Paradigms for Future Computing Curricula. Technical Report. Association for Computing Machinery / IEEE Computer Society, New York, NY, USA, (2020).
3. Kumar, A. N., Becker, B. A., Pias, M., Oudshoorn, M., Jalote, P., Servin, C., Aly, S.G., Blumenthal, R. L., Epstein, S. L., and Anderson, M.D. 2023. A Combined Knowledge and Competency (CKC) Model for Computer Science Curricula. *ACM Inroads* 14, 3 (September 2023), 22–29. https://doi.org/10.1145/3605215

4. Clear, A., Clear, T., Vichare, A., Charles, T., Frezza, S., Gutica, M., Lunt, B., Maiorana, F., Pears, A., Pitt, F., Riedesel, C. and Szynkiewicz, J. Designing Computer Science Competency Statements: A Process and Curriculum Model for the 21st Century. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '20). Association for Computing Machinery, New York, NY, USA, (2020), 211–246.
5. Clear, A., Clear, T., Impagliazzo, J. and Wang, P. From Knowledge-based to Competency-based Computing Education: Future Directions. In 2020 IEEE Frontiers in Education Conference (FIE). IEEE, New York, (2020), 1–7.

The Design

1. Principles and Processes
2. Characteristics of Computer Science Graduates
3. Challenges and Opportunities for Computer Science

Principles and Processes

The Task Force

The CS2023 task force consisted of a Steering Committee of 17 members and a committee for each of the 17 knowledge areas. In all, the task force consisted of 94 members from 17 countries.

The Steering Committee

The ACM and IEEE-Computer Society each appointed a co-chair. The rest of the Steering Committee consisted of three members nominated by IEEE-CS co-chair, two members nominated by AAAI, one member nominated by the ACM Committee for Computing Education in Community Colleges (CCECC) and the remaining nine members selected through interviews in April 2021 from among the educators who nominated themselves in response to a Call for Participation posted to multiple ACM Special Interest Group (SIG) mailing lists. The requirements for the Steering Committee members were that they were subject experts willing to work on a volunteer basis, willing to commit to at least ten hours a month to CS2023 activities, willing to commit to attending at least two in-person meetings a year; and were aligned with the CS2023 vision of both revising the CS2013 knowledge model and producing an appropriate competency model.

Knowledge Area Committees

In June 2021, each Steering Committee member took charge of a knowledge area and assembled a committee of 5 to 10 subject experts drawn from: 1) individuals who had nominated themselves in response to the Call for Participation posted to ACM SIG mailing lists; 2) industry experts; and 3) other Steering Committee members who shared interest in the knowledge area. Knowledge Area committee members met monthly to discuss curricular revisions. While the revision effort was in progress, additional subject experts who expressed interest in volunteering were added to the committees.

Guiding Principles for the Process

The guiding principles for the CS2023 curricular revision process were the following.

- **Collaboration:** Each knowledge area was revised by a committee of experts.
- **Diversity:** At every level of activity (Steering Committee, knowledge area committees, knowledge area reviewers, survey participants), participation was solicited and obtained from academia and industry, from different types of academic institutions, and from all over the world.
- **Data-driven:** Data was collected through surveys of academics and industry practitioners to inform the work of the task force.
- **Community outreach:** The work of the task force was presented at multiple conferences including the annual SGCSE Technical Symposium every year. Its work was publicized through regular postings to over a dozen ACM Special Interest Group (SIG) mailing lists.

- **Community input:** Multiple channels were provided for the community to contribute, including feedback forms and email addresses for knowledge areas and for earlier versions of the curricular guidelines.
- **Continuous review and revision:** Each version of the curricular draft was anonymously reviewed by multiple outside experts. Revision reports were produced to document how the reviews were addressed in subsequent versions of the drafts.
- **Transparency:** The work of CS2023 was documented for review and comments by the community on the csed.acm.org website. Available information included composition of knowledge area committees, results of surveys, and the process used to form the task force.

The Process

The objectives of documenting the process are several.

- Knowledge of the process informs interpretation of the product.
- Future curricular revisions can benefit from knowledge of the process, particularly how the process can be improved to produce better curricular guidelines.
- Curricular guidelines are a community effort. Documenting the process helps the community understand how it has contributed to the effort and how it can have a greater voice in curricular design going forward.

The overall curricular revision process was as follows.

- In 2021, surveys were conducted of the current use of CS2013 curricular guidelines and the importance of various components of curricula. The surveys were filled out by 212 educators in the United States, 215 educators from abroad and 865 industry respondents. The summaries of the surveys were incorporated.
- In May 2022, Version Alpha of the curricular guidelines was released. It contained a revised version of the CS2013 knowledge model. It was publicized internationally, and feedback was solicited. The draft of each knowledge area was sent out to reviewers suggested by the knowledge area committee. Their reviews were incorporated into the subsequent version of the curricular draft. In September-November 2022, a survey of the mathematical requirements of computer science was filled out by 597 educators.
- In March 2023, Version Beta of the curricular guidelines was released. It contained a preliminary competency model. This draft was again sent out to reviewers suggested by the knowledge area committee as well as educators who had nominated themselves through online forms. Their reviews were incorporated into the subsequent version of the curricular draft. In all, 99 reviewers from 18 countries were involved in the two review cycles.
- Over July and August of 2023, 182 educators from 30 countries filled out 70 surveys on the list of core topics. One hundred and ten educators filled out a survey of the characteristics of graduates and 65 educators filled out a survey of the challenges for computer science programs.
- In August 2023, Version Gamma of the curricular guidelines was posted online for a final round of comments and suggestions. It contained course and curricular packaging information, core topics and hours, a framework for identifying tasks to build a competency model and summaries of articles on curricular practices.

- The report was finalized in January 2024 and copy-edited in March 2024.

This process is illustrated in Figure 1.

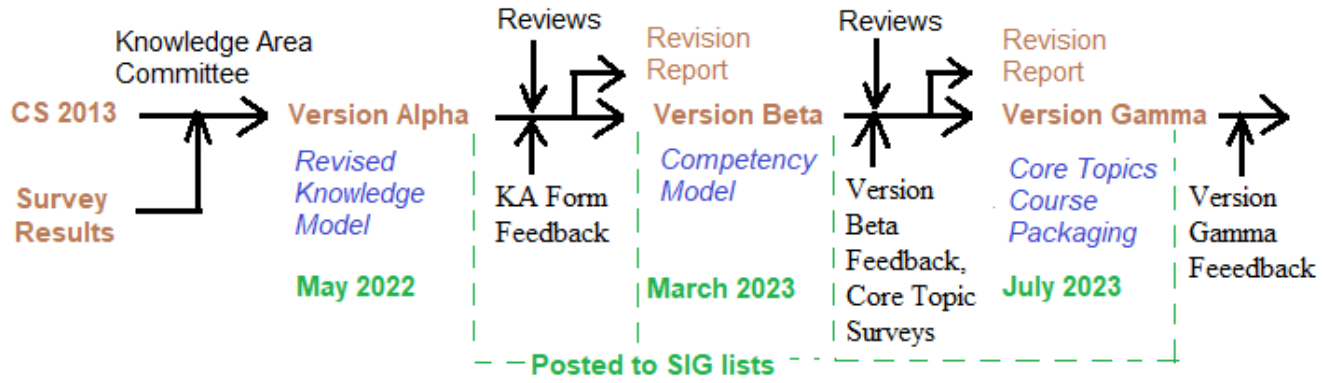


Figure 1. CS2023 Curricular revision process

Guiding Concerns for the Curricular Recommendations

The concerns guiding the CS2023 curricular recommendations are the following.

.

- Computer Science is a rapidly changing discipline. The curriculum should be designed to prepare graduates to not only keep up, but also thrive in the discipline.
  In CS2023, emerging areas have been introduced (e.g., quantum computing) or expanded (e.g., machine learning). Self-directed learning has been listed as a characteristic of graduates.
- One size does not fit all with computer science curricula. A curriculum must be responsive to the needs of its students and the industries that hire them.
  In CS2023, programs are offered the flexibility to select the knowledge areas on which they want to focus. When the knowledge areas are coherently chosen, they define the **competency area(s)** of the program. This design caters to a variety of institutions, department sizes, and student populations.
- Computer Science is rapidly growing as a discipline. A curriculum that covers everything that could be considered computer science would be too onerous with limited broad-based utility.
  In CS2023, the size of CS Core, i.e., the topics that all graduates must know, has been reduced so that programs will have more room to specialize in the competency area(s) of their choice. Programs are encouraged to go beyond CS Core topics to include as many KA Core topics and Non-core topics as possible. But the lack of coverage of any non-CS Core topic should be interpreted as a choice of focus, not lack of value.
- Computer science is more than the sum of the 17 knowledge areas specified by CS2023. The connections between these knowledge areas are important. A curriculum should help students mature as computer science practitioners by helping them make lateral connections among the knowledge areas.

In CS2023, "See also" annotations have been inserted throughout the Body of Knowledge to help educators make connections for their students between knowledge areas.

- Given the pervasiveness of computing applications in every walk of life, a curriculum must address issues of society, ethics, and the profession as integrally and widely as possible. Dealing with these issues is integral to the **whole solution** to a problem that every graduate must be prepared to deliver.
  In CS2023, these issues have been specified in the Society, Ethics, and the Profession (SEP) knowledge area. In addition, SEP topics specific to other knowledge areas have been specified in separate knowledge units within those knowledge areas.
- A curriculum should strive to educate the **whole person**. In computer science, this includes creating opportunities for students to develop professional dispositions (often called soft skills) valued in the workplace.
  In CS2023, the professional dispositions most relevant for each knowledge area have been identified.
- The application of mathematics has increased in computer science. At the same time, mathematics should not be the reason why otherwise well-qualified students are kept away from computer science.
  In CS2023, mathematical needs have been individually identified for each knowledge area. This gives students the flexibility to negotiate a curriculum based on their level of mathematical maturity. It also provides educators the option to cover the necessary mathematics as part of a computer science course, thereby foregoing mathematics prerequisites that may pose barriers for wider participation of students.
- Computer science is at an inflection point with the advent of generative AI. Given that generative AI is only about a year old in its current form and its capabilities are expected to increase rapidly in the near future, it is too early to predict the effects of generative AI on computer science education.
  In CS2023, a speculative exercise was conducted on the implications of generative AI for the various knowledge areas. The results are to be treated as thought exercises, not predictions. A curricular practice article on the implications of generative AI for introductory programming has also been included.

The Knowledge Model Revision Process

Each knowledge area was reviewed and revised by a committee of experts who met regularly and invited contributors who provided input as needed. The committee used CS2013 as the starting point. For each knowledge area, a form was posted on the csed.acm.org website that could be used by the computer science education community to provide targeted feedback. A presentation at the SIGCSE Technical Symposium in Providence, RI, USA in March 2022 was used to publicize the CS2023 effort and invite the community to contribute.

Version Alpha draft was released in May 2022. The draft was posted on the csed.acm.org website.

- The computer science education community was invited to provide feedback through postings on the mailing lists of over a dozen ACM Special Interest Groups (SIGs) including SIGPLAN, SIGOPS, SIGMOBILE, SIGCHI, SIGCAS, SIGCSE, SIGARCH, SIGAI, PODC, SIGACCESS, SIGCSEIRE, UK-SIGCSE, and SIGGRAPH in May 2022 and again in September 2022.

- The draft was sent out for anonymous review to outside experts suggested by the committee as shown in Table 1.

The committee incorporated the feedback from the community and the reviewers to produce Version Beta. It also produced a revision report documenting how it had addressed the comments and suggestions of the community and the reviewers. Both Version Beta draft and knowledge area revision reports were posted on the website.

Version Beta draft was released in March 2023. Educators were invited to nominate themselves to review it.

- The computer science education community was again invited to provide feedback through postings on the SIG mailing lists mentioned earlier in March 2023. The draft was also publicized at the SIGCSE Technical Symposium in Toronto, Canada in March 2023.
- The draft was sent out for anonymous review to additional outside experts suggested by the committee as well as qualified self-nominated educators as shown in Table 1.

Again, the committee incorporated the feedback from the community and the reviewers to produce **Version Gamma** in August 2023. It also produced a Version Beta revision report. Both Version Gamma draft and Version Beta revision report were posted on the website. Their availability was publicized through postings on the SIG mailing lists mentioned earlier in September 2023. Subsequent feedback received from the community was incorporated to produce the final version of the report.

Table 1 lists the number of formal reviews solicited and received for each knowledge area on its Version Alpha and Beta drafts. The formal reviewers, both invited and self-nominated, have been listed in the Acknowledgments section at the end of this report. Note that Table 1 does not include statistics about informal feedback provided by the community to various drafts.

Knowledge Area	**Version Alpha**		**Version Beta**		
	Invited	**Reviewed**	**Invited**	**Self-Nominated**	**Reviewed**
Artificial Intelligence (AI)	1		10		2
Algorithmic Foundations (AL)	6	3	9	5	5
Architecture and Organization (AR)	15	1		3	2
Data Management (DM)	10	2		2	2
Foundations of Programming Languages (FPL)	10	3	16	4	4
Graphics and Interactive Techniques (GIT)	3	3	3		3
Human-Computer Interaction (HCI)	9	3	15	2	2
Mathematics & Statistical Foundations (MSF)			15		4
Networking and Communication (NC)	9	1	10	3	4
Operating Systems (OS)	7	3	6	1	2

Parallel and Distributed Computing (PDC)	4	4		1	1
Software Development Fundamentals (SDF)	4	2	10	2	8
Software Engineering (SE)	4	3	10	1	4
Security (SEC)			6	2	3
Society, Ethics, and the Profession (SEP)	8		7	1	3
Systems Fundamentals (SF)	5	1	5	2	2
Specialized Platform Development (SPD)	9	3	8		

Table 1. The number of formal reviews solicited and received for each knowledge area.

CS Core and KA Core topics were identified by knowledge area committees and instructional hours needed to cover the topics were estimated as follows.

1. Tier 1 and Tier 2 topics from CS2013 were reallocated into CS Core and KA core topics. Some CS2013 core topics were dropped, and others were newly added, in CS2023.
2. The skill level recommended for each core topic was identified, as listed in *Core Topics Table* in Section 3. Based on the skill level, the instructional hours needed to cover each topic were estimated.
3. Seventy surveys were conducted covering all the CS Core topics. In the surveys, for each CS Core topic, respondents were asked whether every computer science graduate must know the topic and if so, the skill level at which they must know the topic. The surveys were filled out by 182 computer science educators. The results of the surveys were used to revise the list of CS Core topics in each knowledge area.
4. Finally, core topics and hours shared between knowledge areas were identified and documented.

Characteristics of Computer Science Graduates

The characteristics that will help computer science graduates realize their full potential while meeting the current and future needs of society are central to the design of computer science programs. Every recent iteration of computer science curricula (2001, 2008, 2013) has attempted to identify them. Given the dynamic nature of computer science, these characteristics have evolved over time. After obtaining input from surveys of 110 academics and 865 industry practitioners, the CS2023 Task Force identified characteristics along three dimensions.

- **Professional knowledge and skills** demonstrating technical expertise
- **Professional responsibilities** toward society, ethics, and the profession
- **Professional dispositions** such as persistence and life-long learning

Professional knowledge and skills—for a technical solution

A computer science graduate must have foundational technical knowledge and skills described as CS Core in CS2023, supplemented by more advanced knowledge of the KA Core and Non-core topics selected by each institution. A computer science graduate should be able to apply the knowledge and skills to develop complete and correct solutions to problems.

A computer science graduate must have minimally acquired the following **knowledge**.

- Fundamentals of software, systems, and applications development
- Current tools, libraries, and frameworks for developing solutions
- Mathematical and theoretical underpinnings of computing

A graduate must have developed essential **skills** including problem-solving (decomposition, recognition of solution patterns), algorithmic thinking, analytical reasoning, and working at multiple levels of abstraction to formulate computing problems and their solutions.

Other desirable characteristics include the ability to quickly learn the essentials of new problem domains and apply computing solutions to them, the ability to handle ambiguity and uncertainty, and the ability to work in teams.

Professional Responsibilities – for the whole solution

A computer science graduate must be committed to the **whole solution:** not just to the technical aspects but also to issues of the society, ethics, and the profession summarized in the CS2023 knowledge area of the same name (SEP) and elaborated as a separate knowledge unit in most other knowledge areas. To that end, a graduate must:

- Demonstrate knowledge of a code of ethics and conduct appropriate for computing professionals (e.g., ACM [1], AAAI [2], or IEEE [3]) and commitment to abide by such a code.

- Demonstrate awareness of responsibilities beyond those captured in a professional code (e.g., global and cultural competence and the priorities and impact of local values and practices across the world).
- Work to maximize the benefits of computing for the society at large while preventing harm to individuals.

Professional dispositions – the whole person view

Professional dispositions are essential for not just succeeding in the workplace but also thriving as a professional over the long run. The dispositions identified by multiple CS2023 knowledge areas as essential for computer science graduates include:

- **Adaptable**, as the discipline is continually evolving;
- **Collaborative**, as most real-world applications are team efforts;
- **Inventive** in order to devise new solutions and apply existing solutions to new contexts;
- **Meticulous** to ensure the correctness and completeness of solutions;
- **Persistent,** since computational problem-solving is an iterative process;
- **Proactive** to anticipate issues pertaining to usability, security, ethics, etc.;
- **Responsible** in all aspects of a solution including design, implementation, and maintenance;
- **Self-directed**, as commitment to life-long learning is required due to rapid evolution of the discipline.

These characteristics change in importance over the career of a graduate: some characteristics are more important during early career while others are essential for success over the long run [4]. Moreover, given the dynamic nature of computer science, the desirable characteristics of computer science graduates will also continue to evolve.

References

1. https://www.acm.org/code-of-ethics; accessed March 2024.
2. https://www.ieee.org/about/corporate/governance/p7-8.html; accessed March 2024.
3. https://aaai.org/about-aaai/ethics-and-diversity/#ethics-conduct; accessed March 2024.
4. Simha, R., Kumar, A.N., and Raj. R. K. 2024. Undergraduate Computer Science Curricula. *Commun. ACM* 67, 2 (February 2024), 29–31; https://doi.org/10.1145/3624729.

Challenges and Opportunities for Computer Science

What are the challenges and opportunities facing undergraduate computer science education today? What are their implications for the adoption/adaptation of CS2023 curricular guidelines? How can CS2023 help address them? We attempt to explore these questions from several institutional perspectives – students, faculty, curricular content, instructional resources, assessment methods, packaging and delivery, and ensuring the long-term vitality of computer science education.

Students

- Computer science has had a long-term problem with diversity of participation. Given that computer science, as a discipline, touches all walks of life and all populations, it benefits from vigorous participation by all populations regardless of their demographic identities. *Everyone belongs in computer science.* Programs should not only make every effort to send out this message but also promote the active participation of all populations in the discipline.

  In CS2023, a curricular practice article has been included on accessibility in computer science education. CS2023 commissioned a special issue of *ACM Inroads* on the practice of computer science education in various geographic regions of the world. (See *ACM Inroads*, Special Issue, 15, 1 (March 2024)).
- Employers everywhere seek professional dispositions (often called soft skills) among graduates (e.g., persistence, being self-directed, adaptive). In response, it has become necessary to make explicit what has always been implicit—the need for students to appreciate the importance of professional dispositions to their future professional success and develop them while still students.

  In CS2023, within each knowledge area, the professional dispositions most relevant to the knowledge area have been identified.

Faculty

- In many institutions, explosive growth in enrollment has put significant strain on faculty resources in terms of class sizes, course loads, etc. Managing faculty load is critical for the vitality of faculty, both in terms of the quality of their teaching and their professional development.
- Recruitment and retention of faculty has been a challenge, given the demand for graduates with advanced degrees in computer science. A trend that has gathered momentum in the last decade is the creation of teaching-track faculty. Institutions must strike the right balance between meeting instructional needs and supporting the professional development of teaching faculty to ensure that they stay current.

Curricular content:

- Computer Science is a rapidly evolving discipline. This has been both a boon and a bane – boon because of new opportunities and bane because of the accompanying challenges. Updating courses and curricula to stay current places significant demands on the resources of computer science educators and should be so acknowledged and supported.

  In CS2023, emerging technologies have been highlighted (e.g., quantum computing) and rapidly evolving areas have been significantly expanded (e.g., machine learning). A curricular practice article has been included on quantum computing education.

- Generative AI, like other emerging technologies, has the potential to revolutionize computer science education. It will impact course content, pedagogy, and assessment techniques. Harnessing generative AI in service of the goals of formal education will be one of the most significant challenges for the community over the next few years.

  In CS2023, a chapter has been included that lists educated guesses on the implications of generative AI for the various knowledge areas. The capabilities of generative AI are expected to rapidly improve. So, how well these speculations are borne out in the future remains to be seen. A curricular practice article has been included on the implications of generative AI for introductory programming.
- Theoretical and mathematical underpinnings make computer science a science. They are essential for long-term career success whereas tools and technologies prepare students for immediate employability. Striking the right balance between these dual objectives will continue to be a challenge, given the increasing need for mathematics in computer science (e.g., in machine learning) and often inadequate mathematical preparation of students entering computer science programs.

  In CS2023, mathematical requirements have been individually identified for each knowledge area. This provides for flexibility: mathematically underprepared students can better navigate the curriculum and faculty can either require a prerequisite mathematics course or cover the necessary mathematics as part of a computer science course.
- Given the pervasiveness of computing applications, a computing solution is not just technical in nature. It must incorporate issues related to society, ethics, and the profession as well. Interweaving these issues into technical coverage so as to make them unavoidable in a curriculum is a challenge every educator must take up in fulfillment of responsible citizenry.

  In CS2023, issues of society, ethics, and the profession (SEP) have been explicitly enumerated in as many knowledge areas as possible to highlight their importance across the curriculum and help educators incorporate them into their courses. Curricular practice articles have also been included on responsible computing, ethics, and CS for good.
- Computational thinking is now considered the fourth basic skill alongside reading, writing and arithmetic. This provides an opportunity for computer science programs to offer courses for non-majors, both as a service and a recruiting tool. Similarly, interdisciplinary options (CS + X) provide opportunities for computer science educators to collaborate and create programs that will also enhance the learning experience of computer science students. Resource availability is the primary constraint for availing both these worthwhile opportunities.

  In CS2023, a curricular practice article has been included on CS + X.

Instructional resources:

- Increasingly, entire computer science courses and curricula are being moved onto the cloud and to using freely available software and services online. While the benefits of such moves are many, the pitfalls are many as well, including loss of control over the resources and data, privacy issues, etc. A careful consideration of both benefits and pitfalls by all stakeholders should precede such moves.
- The free availability of a variety of big data presents an invaluable opportunity for educators to scale assignments and projects and use real-life problems in their courses to better motivate students.

In CS2023, a curricular practice article has been included on “CS for Good,” a great example of using computing to solve real-life problems.

- Some of the critical resources relevant to emerging areas in computing may not be equitably available to the global computer science community. For example, quantum resources are export-regulated, which hinders quantum computing education.
- The availability of cutting-edge textbooks in languages other than English and the affordability of textbooks are ongoing challenges for computer science educators.

Assessment Methods:

- Generative AI tools are rendering existing assessment methods ineffective. In addition, tools that detect plagiarism based on generative AI are still evolving.
- Current assessment techniques are not well-suited to provide personalized feedback efficiently and at scale.

Packaging and Delivery:

- Online delivery of courses has matured since the COVID-19 pandemic. The success of computer science courses delivered online, whether synchronously or asynchronously, critically rests on the level of maturity of the student. Taking this into account is not only in the best interests of the student but also the discipline and the profession.
- Computer science has been rapidly fragmenting, spinning off Software Engineering, Data Science, Security, and lately, Artificial Intelligence, as separate disciplines. This should be seen as both an opportunity and a challenge: an opportunity to amortize costs by sharing resources and costs; and a challenge to differentiate the disciplines sufficiently from each other so that students can make educated choices. A variety of options are available for differentiation, from certificates and minors all the way up to distinct majors.

Computer Science education research has lately been gathering momentum. It is now a mainstream area of doctoral research. Professional conferences catering to it are increasing in number and ranking. This portends well for computer science education by providing a feedback loop for improvement that could not have come sooner. It signals the maturing of computer science education.

The Details

1. Body of Knowledge
2. Core Topics Table
3. Curricular Packaging
4. Competency Framework Examples

Body of Knowledge

	Knowledge Area	# Knowledge Units	CS Core Hours	KA Core Hours
AI	Artificial Intelligence	12	12	18
AL	Algorithmic Foundations	5	32	32
AR	Architecture and Organization	11	9	16
DM	Data Management	13	10	26
FPL	Foundations of Programming Languages	22	21	19
GIT	Graphics and Interactive Techniques	12	4	70
HCI	Human-Computer Interaction	6	8	16
MSF	Mathematical and Statistical Foundations	5	55	145
NC	Networking and Communication	8	7	24
OS	Operating Systems	14	8	13
PDC	Parallel and Distributed Computing	5	9	26
SDF	Software Development Fundamentals	5	43	
SE	Software Engineering	9	6	21
SEC	Security	7	6	35
SEP	Society, Ethics, and the Profession	11	18	14
SF	Systems Fundamentals	9	18	8
SPD	Specialized Platform Development	8	4	
	Total	162	270	N/A

Artificial Intelligence (AI)

Preamble

Artificial intelligence (AI) studies problems that are difficult or impractical to solve with traditional algorithmic approaches. These problems are often reminiscent of those considered to require human intelligence, and the resulting AI solution strategies typically generalize over classes of problems. AI techniques are now pervasive in computing, supporting everyday applications such as email, social media, photography, financial markets, and intelligent virtual assistants (e.g., Siri, Alexa). These techniques are also used in the design and analysis of autonomous agents that perceive their environment and interact rationally with it, such as self-driving vehicles and other robots.

Traditionally, AI has included a mix of symbolic and subsymbolic approaches. The solutions it provides rely on a broad set of general and specialized knowledge representation schemes, problem solving mechanisms, and optimization techniques. These approaches deal with perception (e.g., speech recognition, natural language understanding, computer vision), problem solving (e.g., search, planning, optimization), generation (e.g., narrative, conversation, images, models, recommendations), acting (e.g., robotics, task-automation, control), and the architectures needed to support them (e.g., single agents, multi-agent systems). Machine learning may be used within each of these aspects and can even be employed end-to-end across all of them. The study of Artificial Intelligence prepares students to determine when an AI approach is appropriate for a given problem, identify appropriate representations and reasoning mechanisms, implement them, and evaluate them with respect to both performance and their broader societal impact.

Over the past decade, the term “artificial intelligence” has become commonplace within businesses, news articles, and everyday conversation, driven largely by a series of high-impact machine learning applications. These advances were made possible by the widespread availability of large datasets, increased computational power, and algorithmic improvements. In particular, there has been a shift from engineered representations to representations learned automatically through optimization over large datasets. The resulting advances have put such terms as “neural networks” and “deep learning” into everyday vernacular. Businesses now advertise AI-based solutions as value-additions to their services, so that “artificial intelligence” is now both a technical term and a marketing buzzword. Other disciplines, such as biology, art, architecture, and finance, increasingly use AI techniques to solve problems within their disciplines.

For the first time in our history, the broader population has access to sophisticated AI-driven tools, including tools to generate essays or poems from a prompt, artwork from a description, and fake photographs or videos depicting real people. AI technology is now in widespread use in stock trading, curating our news and social media feeds, automated evaluation of job applicants, detection of medical conditions, and influencing prison sentencing through recidivism prediction. Consequently, AI technology can have significant societal impacts and ethical considerations that must be understood and considered when developing and applying it.

Changes since CS2013

To reflect this recent growth and societal impact, the knowledge area has been revised from CS2013 in the following ways.

- The name has changed from “Intelligent Systems” to “Artificial Intelligence,” to reflect the most common terminology used for these topics within the field and its more widespread use outside the field.
- An increased emphasis on neural networks and representation learning reflects the recent advances in the field. Given its key role throughout AI, search is still emphasized but there is a slight reduction on symbolic methods in favor of understanding subsymbolic methods and learned representations. It is important, however, to retain knowledge-based and symbolic approaches within the AI curriculum because these methods offer unique capabilities, are used in practice, ensure a broad education, and because more recent neurosymbolic approaches integrate both learned and symbolic representations.
- There is an increased emphasis on practical applications of AI, including a variety of areas (e.g., medicine, sustainability, social media). This includes explicit discussion of tools that employ deep generative models (e.g., ChatGPT, DALL-E, Midjourney) and are now in widespread use, covering how they work at a high level, their uses, and their shortcomings/pitfalls.
- The curriculum reflects the importance of understanding and assessing the broader societal impacts and implications of AI methods and applications, including issues in AI ethics, fairness, trust, and explainability.
- The AI knowledge area includes connections to data science through 1) cross-connections with the Data Management and other knowledge areas and 2) a sample Data Science model course.
- There are explicit goals to develop basic AI literacy and critical thinking in every computer science student, given the breadth of interconnections between AI and other knowledge areas in practice.

Consider recent AI advances when using this curriculum

The field of AI is undergoing rapid development and increasingly widespread applications. Since the first draft of this document, several new techniques (e.g., generative networks, large language models) have become widely used and so were added to the CS or KA Cores. This document is as current as we can make it in 2023. However, we expect such rapid changes to continue in the subfield of AI during the expected life of this document. Consequently, it is imperative that faculty teaching AI understand current advances and consider whether these advances should be taught in order to keep the curriculum current.

Core Hours

Knowledge Unit	CS Core	KA Core
Fundamental Issues	2	1
Search	2 + 3 (AL)	6

Fundamental Knowledge Representation and Reasoning	1 + 1 (MSF)	2
Machine Learning	4	6
Applications and Societal Impact	3	3
Probabilistic Representation and Reasoning		
Planning		
Logical Representation and Reasoning		
Agents and Cognitive Systems		
Natural Language Processing		
Robotics		
Perception and Computer Vision		
Total	**12**	**18**

The CS Core includes 3 hours that are shared with and counted under Algorithm Foundations (Uninformed search) and 1 hour that is shared with and counted under Mathematical Foundations (Probability).

Knowledge Units

AI-Introduction: Fundamental Issues

CS Core:

1. Overview of AI problems, Examples of successful recent AI applications
2. Definitions of agents with examples (e.g., reactive, deliberative)
3. What is intelligent behavior?
   a. The Turing test and its flaws
   b. Multimodal input and output
   c. Simulation of intelligent behavior
   d. Rational versus non-rational reasoning
4. Problem characteristics
   a. Fully versus partially observable
   b. Single versus multi-agent
   c. Deterministic versus stochastic
   d. Static versus dynamic
   e. Discrete versus continuous
5. Nature of agents

a. Autonomous, semi-autonomous, mixed-initiative autonomy
b. Reflexive, goal-based, and utility-based
c. Decision making under uncertainty and with incomplete information
d. The importance of perception and environmental interactions
e. Learning-based agents
f. Embodied agents
    i. sensors, dynamics, effectors
6. Overview of AI Applications, growth, and impact (economic, societal, ethics)

KA Core:

7. Practice identifying problem characteristics in example environments
8. Additional depth on nature of agents with examples
9. Additional depth on AI Applications, Growth, and Impact (economic, societal, ethics, security)

Non-core:

10. Philosophical issues
11. History of AI

Illustrative Learning Outcomes:

1. Describe the Turing test and the “Chinese Room” thought experiment.
2. Differentiate between optimal reasoning/behavior and human-like reasoning/behavior.
3. Differentiate the terms: AI, machine learning, and deep learning.
4. Enumerate the characteristics of a specific problem.

AI-Search: Search

CS Core:

1. State space representation of a problem
    a. Specifying states, goals, and operators
    b. Factoring states into representations (hypothesis spaces)
    c. Problem solving by graph search
        i. e.g., Graphs as a space, and tree traversals as exploration of that space
        ii. Dynamic construction of the graph (not given upfront)
2. Uninformed graph search for problem solving (See also: AL-Foundational)
    a. Breadth-first search
    b. Depth-first search
        i. With iterative deepening
    c. Uniform cost search
3. Heuristic graph search for problem solving (See also: AL-Strategies)
    a. Heuristic construction and admissibility
    b. Hill-climbing
    c. Local minima and the search landscape
        i. Local vs global solutions
    d. Greedy best-first search
    e. A* search

4. Space and time complexities of graph search algorithms

KA Core:

5. Bidirectional search
6. Beam search
7. Two-player adversarial games
    a. Minimax search
    b. Alpha-beta pruning
        i. Ply cutoff
8. Implementation of A* search
9. Constraint satisfaction

Non-core:

10. Understanding the search space
    a. Constructing search trees
    b. Dynamic search spaces
    c. Combinatorial explosion of search space
    d. Search space topology (e.g., ridges, saddle points, local minima)
11. Local search
12. Tabu search
13. Variations on A* (IDA*, SMA*, RBFS)
14. Two-player adversarial games
    a. The horizon effect
    b. Opening playbooks/endgame solutions
    c. What it means to “solve” a game (e.g., checkers)
15. Implementation of minimax search, beam search
16. Expectimax search (MDP-solving) and chance nodes
17. Stochastic search
    a. Simulated annealing
    b. Genetic algorithms
    c. Monte-Carlo tree search

Illustrative Learning Outcomes:

1. Design the state space representation for a puzzle (e.g., N-queens or 3-jug problem)
2. Select and implement an appropriate uninformed search algorithm for a problem (e.g., tic-tac-toe), and characterize its time and space complexities.
3. Select and implement an appropriate informed search algorithm for a problem after designing a helpful heuristic function (e.g., a robot navigating a 2D gridworld).
4. Evaluate whether a heuristic for a given problem is admissible/can guarantee an optimal solution.
5. Apply minimax search in a two-player adversarial game (e.g., connect four), using heuristic evaluation at a particular depth to compute the scores to back up. [KA Core]
6. Design and implement a genetic algorithm solution to a problem.
7. Design and implement a simulated annealing schedule to avoid local minima in a problem.

8. Design and implement A*/beam search to solve a problem, and compare it against other search algorithms in terms of the solution cost, number of nodes expanded, etc.
9. Apply minimax search with alpha-beta pruning to prune search space in a two-player adversarial game (e.g., connect four).
10. Compare and contrast genetic algorithms with classic search techniques, explaining when it is most appropriate to use a genetic algorithm to learn a model versus other forms of optimization (e.g., gradient descent).
11. Compare and contrast various heuristic searches vis-a-vis applicability to a given problem.
12. Model a logic or Sudoku puzzle as a constraint satisfaction problem, solve it with backtrack search, and determine how much arc consistency can reduce the search space.

AI-KRR: Fundamental Knowledge Representation and Reasoning

CS Core:

1. Types of representations
    a. Symbolic, logical
        i. Creating a representation from a natural language problem statement
    b. Learned subsymbolic representations
    c. Graphical models (e.g., naive Bayes, Bayesian network)
2. Review of probabilistic reasoning, Bayes theorem (See also: MSF-Probability)
3. Bayesian reasoning
    a. Bayesian inference

KA Core:

4. Random variables and probability distributions
    a. Axioms of probability
    b. Probabilistic inference
    c. Bayes' Rule (derivation)
    d. Bayesian inference (more complex examples)
5. Independence
6. Conditional Independence
7. Markov chains and Markov models
8. Utility and decision making

Illustrative Learning Outcomes:

1. Given a natural language problem statement, encode it as a symbolic or logical representation.
2. Explain how we can make decisions under uncertainty, using concepts such as Bayes theorem and utility.
3. Compute a probabilistic inference in a real-world problem using Bayes' theorem to determine the probability of a hypothesis given evidence.
4. Apply Bayes' rule to determine the probability of a hypothesis given evidence.
5. Compute the probability of outcomes and test whether outcomes are independent.

AI-ML: Machine Learning

CS Core:

1. Definition and examples of a broad variety of machine learning tasks
    a. Supervised learning
        i. Classification
        ii. Regression
    b. Reinforcement learning
    c. Unsupervised learning
        i. Clustering
2. Fundamental ideas:
    a. No free lunch theorem: no one learner can solve all problems; representational design decisions have consequences.
    b. Sources of error and undecidability in machine learning
3. A simple statistical-based supervised learning such as linear regression or decision trees
    a. Focus on how they work without going into mathematical or optimization details; enough to understand and use existing implementations correctly
4. The overfitting problem/controlling solution complexity (regularization, pruning – intuition only)
    a. The bias (underfitting) – variance (overfitting) tradeoff
5. Working with Data
    a. Data preprocessing
        i. Importance and pitfalls of preprocessing choices
    b. Handling missing values (imputing, flag-as-missing)
        i. Implications of imputing vs flag-as-missing
    c. Encoding categorical variables, encoding real-valued data
    d. Normalization/standardization
    e. Emphasis on real data, not textbook examples
6. Representations
    a. Hypothesis spaces and complexity
    b. Simple basis feature expansion, such as squaring univariate features
    c. Learned feature representations
7. Machine learning evaluation
    a. Separation of train, validation, and test sets
    b. Performance metrics for classifiers
    c. Estimation of test performance on held-out data
    d. Tuning the parameters of a machine learning model with a validation set
    e. Importance of understanding what a model is doing, where its pitfalls/shortcomings are, and the implications of its decisions
8. Basic neural networks
    a. Fundamentals of understanding how neural networks work and their training process, without details of the calculations
    b. Basic introduction to generative neural networks (e.g., large language models)
9. Ethics for Machine Learning (See also: SEP-Context)
    a. Focus on real data, real scenarios, and case studies
    b. Dataset/algorithmic/evaluation bias and unintended consequences

KA Core:

10. Formulation of simple machine learning as an optimization problem, such as least squares linear regression or logistic regression
    a. Objective function
    b. Gradient descent
    c. Regularization to avoid overfitting (mathematical formulation)
11. Ensembles of models
    a. Simple weighted majority combination
12. Deep learning
    a. Deep feed-forward networks (intuition only, no mathematics)
    b. Convolutional neural networks (intuition only, no mathematics)
    c. Visualization of learned feature representations from deep nets
    d. Other architectures (generative NN, recurrent NN, transformers, etc.)
13. Performance evaluation
    a. Other metrics for classification (e.g., error, precision, recall)
    b. Performance metrics for regressors
    c. Confusion matrix
    d. Cross-validation
        i. Parameter tuning (grid/random search, via cross-validation)
14. Overview of reinforcement learning methods
15. Two or more applications of machine learning algorithms
    a. E.g., medicine and health, economics, vision, natural language, robotics, game play
16. Ethics for Machine Learning
    a. Continued focus on real data, real scenarios, and case studies (See also: SEP-Context)
    b. Privacy (See also: SEP-Privacy)
    c. Fairness (See also: SEP-Privacy)
    d. Intellectual property
    e. Explainability

Non-core:

17. General statistical-based learning, parameter estimation (maximum likelihood)
18. Supervised learning
    a. Decision trees
    b. Nearest-neighbor classification and regression
    c. Learning simple neural networks / multi-layer perceptrons
    d. Linear regression
    e. Logistic regression
    f. Support vector machines (SVMs) and kernels
    g. Gaussian Processes
19. Overfitting
    a. The curse of dimensionality
    b. Regularization (mathematical computations, L_2 and L_1 regularization)
20. Experimental design

    a. Data preparation (e.g., standardization, representation, one-hot encoding)
    b. Hypothesis space
    c. Biases (e.g., algorithmic, search)
    d. Partitioning data: stratification, training set, validation set, test set
    e. Parameter tuning (grid/random search, via cross-validation)
    f. Performance evaluation
        i. Cross-validation
        ii. Metric: error, precision, recall, confusion matrix
        iii. Receiver operating characteristic (ROC) curve and area under ROC curve
21. Bayesian learning (Cross-Reference AI/Reasoning Under Uncertainty)
    a. Naive Bayes and its relationship to linear models
    b. Bayesian networks
    c. Prior/posterior
    d. Generative models
22. Deep learning
    a. Deep feed-forward networks
    b. Neural tangent kernel and understanding neural network training
    c. Convolutional neural networks
    d. Autoencoders
    e. Recurrent networks
    f. Representations and knowledge transfer
    g. Adversarial training and generative adversarial networks
    h. Attention mechanisms
23. Representations
    a. Manually crafted representations
    b. Basis expansion
    c. Learned representations (e.g., deep neural networks)
24. Unsupervised learning and clustering
    a. K-means
    b. Gaussian mixture models
    c. Expectation maximization (EM)
    d. Self-organizing maps
25. Graph analysis (e.g., PageRank)
26. Semi-supervised learning
27. Graphical models (See also: AI-Probability)
28. Ensembles
    a. Weighted majority
    b. Boosting/bagging
    c. Random forest
    d. Gated ensemble
29. Learning theory
    a. General overview of learning theory / why learning works
    b. VC dimension
    c. Generalization bounds

30. Reinforcement learning
    a. Exploration vs exploitation tradeoff
    b. Markov decision processes
    c. Value and policy iteration
    d. Policy gradient methods
    e. Deep reinforcement learning
    f. Learning from demonstration and inverse RL
31. Explainable / interpretable machine learning
    a. Understanding feature importance (e.g., LIME, Shapley values)
    b. Interpretable models and representations
32. Recommender systems
33. Hardware for machine learning
    a. GPUs / TPUs
34. Application of machine learning algorithms to:
    a. Medicine and health
    b. Economics
    c. Education
    d. Vision
    e. Natural language
    f. Robotics
    g. Game play
    h. Data mining (Cross-reference DM/Data Analytics)
35. Ethics for Machine Learning
    a. Continued focus on real data, real scenarios, and case studies (See also: SEP-Context)
    b. In depth exploration of dataset/algorithmic/evaluation bias, data privacy, and fairness (See also: SEP-Privacy, SEP-Context)
    c. Trust / explainability

Illustrative Learning Outcomes:

1. Describe the differences among the three main styles of learning (supervised, reinforcement, and unsupervised) and determine which is appropriate to a particular problem domain.
2. Differentiate the terms of AI, machine learning, and deep learning.
3. Frame an application as a classification problem, including the available input features and output to be predicted (e.g., identifying alphabetic characters from pixel grid input).
4. Apply two or more simple statistical learning algorithms to a classification task and measure the classifiers' accuracy.
5. Identify overfitting in the context of a problem and learning curves and describe solutions to overfitting.
6. Explain how machine learning works as an optimization/search process.
7. Implement a statistical learning algorithm and the corresponding optimization process to train the classifier and obtain a prediction on new data.
8. Describe the neural network training process and resulting learned representations.

9. Explain proper ML evaluation procedures, including the differences between training and testing performance, and what can go wrong with the evaluation process leading to inaccurate reporting of ML performance.
10. Compare two machine learning algorithms on a dataset, implementing the data preprocessing and evaluation methodology (e.g., metrics and handling of train/test splits) from scratch.
11. Visualize the training progress of a neural network through learning curves in a well-established toolkit (e.g., TensorBoard) and visualize the learned features of the network.
12. Compare and contrast several learning techniques (e.g., decision trees, logistic regression, naive Bayes, neural networks, and belief networks), providing examples of when each strategy is superior.
13. Evaluate the performance of a simple learning system on a real-world dataset.
14. Characterize the state of the art in learning theory, including its achievements and shortcomings.
15. Explain the problem of overfitting, along with techniques for detecting and managing the problem.
16. Explain the triple tradeoff among the size of a hypothesis space, the size of the training set, and performance accuracy.
17. Given a real-world application of machine learning, describe ethical issues regarding the choices of data, preprocessing steps, algorithm selection, and visualization/presentation of results.

AI-SEP: Applications and Societal Impact

Note: There is substantial benefit to studying applications and ethics/fairness/trust/explainability in a curriculum alongside the methods and theory that they apply to, rather than covering ethics in a separate, dedicated class session. Whenever possible, study of these topics should be integrated alongside other modules, such as exploring how decision trees could be applied to a specific problem in environmental sustainability such as land use allocation, then assessing the social/environmental/ethical implications of doing so.

CS Core:

1. At least one application of AI to a specific problem and field, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose at least one for the CS Core).
    a. Formulating and evaluating a specific application as an AI problem
        i. How to deal with underspecified or ill-posed problems
    b. Data availability/scarcity and cleanliness
        i. Basic data cleaning and preprocessing
        ii. Data set bias
    c. Algorithmic bias
    d. Evaluation bias
    e. Assessment of societal implications of the application
2. Deployed deep generative models
    a. High-level overview of deep image generative models (e.g., as of 2023, DALL-E, Midjourney, Stable Diffusion, etc.), their uses, and their shortcomings/pitfalls.
    b. High-level overview of large language models (e.g., as of 2023, ChatGPT, Bard, etc.), their uses, and their shortcomings/pitfalls.
3. Overview of societal impact of AI

a. Ethics (See also: SEP-Context)
b. Fairness (See also: SEP-Privacy, SEP-DEIA)
c. Trust/explainability (See also: SEP-Context)
d. Privacy and usage of training data (See also: SEP-Privacy)
e. Human autonomy and oversight/regulations/legal requirements (See also: SEP-Context)
f. Sustainability (See also: SEP-Sustainability)

KA Core:

4. One or more additional applications of AI to a broad set of problems and diverse fields, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose a different area from that chosen for the CS Core).
    a. Formulating and evaluating a specific application as an AI problem
        i. How to deal with underspecified or ill-posed problems
    b. Data availability/scarcity and cleanliness
        i. Basic data cleaning and preprocessing
        ii. Data set bias
    c. Algorithmic bias
    d. Evaluation bias
    e. Assessment of societal implications of the application
5. Additional depth on deployed deep generative models
    a. Introduction to how deep image generative models work, (e.g., as of 2023, DALL-E, Midjourney, Stable Diffusion) including discussion of attention
    b. Introduction to how large language models work, (e.g., as of 2023, ChatGPT, Bard) including discussion of attention
    c. Idea of foundational models, how to use them, and the benefits/issues with training them from big data
6. Analysis and discussion of the societal impact of AI
    a. Ethics (See also: SEP-Context)
    b. Fairness (See also: SEP-Privacy, SEP-DEIA)
    c. Trust/explainability (See also: SEP-Context)
    d. Privacy and usage of training data (See also: SEP-Privacy)
    e. Human autonomy and oversight/regulations/legal requirements (See also: SEP-Context)
    f. Sustainability (See also: SEP-Sustainability)

Illustrative Learning Outcomes:

1. Given a real-world application domain and problem, formulate an AI solution to it, identifying proper data/input, preprocessing, representations, AI techniques, and evaluation metrics/methodology.
2. Analyze the societal impact of one or more specific real-world AI applications, identifying issues regarding ethics, fairness, bias, trust, and explainability.
3. Describe some of the failure modes of current deep generative models for language or images, and how this could affect their use in an application.

AI-LRR: Logical Representation and Reasoning

Non-core:

1. Review of propositional and predicate logic (See also: MSF-Discrete)
2. Resolution and theorem proving (propositional logic only)
    a. Forward chaining, backward chaining
3. Knowledge representation issues
    a. Description logics
    b. Ontology engineering
4. Semantic web
5. Non-monotonic reasoning (e.g., non-classical logics, default reasoning)
6. Argumentation
7. Reasoning about action and change (e.g., situation and event calculus)
8. Temporal and spatial reasoning
9. Logic programming
    a. Prolog, Answer Set Programming
10. Rule-based Expert Systems
11. Semantic networks
12. Model-based and Case-based reasoning

Illustrative Learning Outcomes:

1. Translate a natural language (e.g., English) sentence into a predicate logic statement.
2. Convert a logic statement into clausal form.
3. Apply resolution to a set of logic statements to answer a query.
4. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses.
5. Identify the components of non-monotonic reasoning and its usefulness as a representational mechanism for belief systems.
6. Compare and contrast the basic techniques for representing uncertainty.
7. Compare and contrast the basic techniques for qualitative representation.
8. Apply situation and event calculus to problems of action and change.
9. Explain the distinction between temporal and spatial reasoning, and how they interrelate.
10. Explain the difference between rule-based, case-based, and model-based reasoning techniques.
11. Define the concept of a planning system and how it differs from classical search techniques.
12. Describe the differences between planning as search, operator-based planning, and propositional planning, providing examples of domains where each is most applicable.
13. Explain the distinction between monotonic and non-monotonic inference.

AI-Probability: Probabilistic Representation and Reasoning

Non-core:

1. Conditional Independence review
2. Knowledge representations
    a. Bayesian Networks
        i. Exact inference and its complexity
        ii. Markov blankets and d-separation
        iii. Randomized sampling (Monte Carlo) methods (e.g., Gibbs sampling)
    b. Markov Networks

   c. Relational probability models
   d. Hidden Markov Models
3. Decision Theory
   a. Preferences and utility functions
   b. Maximizing expected utility
   c. Game theory

Illustrative Learning Outcomes:

1. Compute the probability of a hypothesis given the evidence in a Bayesian network.
2. Explain how conditional independence assertions allow for greater efficiency of probabilistic systems.
3. Identify examples of knowledge representations for reasoning under uncertainty.
4. State the complexity of exact inference. Identify methods for approximate inference.
5. Design and implement at least one knowledge representation for reasoning under uncertainty.
6. Describe the complexities of temporal probabilistic reasoning.
7. Design and implement an HMM as one example of a temporal probabilistic system.
8. Describe the relationship between preferences and utility functions.
9. Explain how utility functions and probabilistic reasoning can be combined to make rational decisions.

AI-Planning: Planning

Non-core:

1. Review of propositional and first-order logic
2. Planning operators and state representations
3. Total order planning
4. Partial-order planning
5. Plan graphs and GraphPlan
6. Hierarchical planning
7. Planning languages and representations
   a. PDDL
8. Multi-agent planning
9. MDP-based planning
10. Interconnecting planning, execution, and dynamic replanning
    a. Conditional planning
    b. Continuous planning
    c. Probabilistic planning

Illustrative Learning Outcomes:

1. Construct the state representation, goal, and operators for a given planning problem.
2. Encode a planning problem in PDDL and use a planner to solve it.
3. Given a set of operators, initial state, and goal state, draw the partial-order planning graph and include ordering constraints to resolve all conflicts.
4. Construct the complete planning graph for GraphPlan to solve a given problem.

AI-Agents: Agents and Cognitive Systems

Non-core:

1. Agent architectures (e.g., reactive, layered, cognitive)
2. Agent theory (including mathematical formalisms)
3. Rationality, Game Theory
    a. Decision-theoretic agents
    b. Markov decision processes (MDP)
    c. Bandit algorithms
4. Software agents, personal assistants, and information access
    a. Collaborative agents
    b. Information-gathering agents
    c. Believable agents (synthetic characters, modeling emotions in agents)
5. Learning agents
6. Cognitive systems
    a. Cognitive architectures (e.g., ACT-R, SOAR, ICARUS, FORR)
    b. Capabilities (e.g., perception, decision making, prediction, knowledge maintenance)
    c. Knowledge representation, organization, utilization, acquisition, and refinement
    d. Applications and evaluation of cognitive systems
7. Multi-agent systems
    a. Collaborating agents
    b. Agent teams
    c. Competitive agents (e.g., auctions, voting)
    d. Swarm systems and biologically inspired models
    e. Multi-agent learning
8. Human-agent interaction (See also: HCI-User, HCI-Accessibility)
    a. Communication methodologies (verbal and non-verbal)
    b. Practical issues
    c. Applications
        i. Trading agents, supply chain management
        ii. Ethical issues of AI interactions with humans
        iii. Regulation and legal requirements of AI systems for interacting with humans

Illustrative Learning Outcomes:

1. Characterize and contrast the standard agent architectures.
2. Describe the applications of agent theory to domains such as software agents, personal assistants, and believable agents, and discuss associated ethical implications.
3. Describe the primary paradigms used by learning agents.
4. Demonstrate using appropriate examples how multi-agent systems support agent interaction.
5. Construct an intelligent agent using a well-established cognitive architecture (ACT-R, SOAR) for solving a specific problem.

AI-NLP: Natural Language Processing

Non-core:

1. Deterministic and stochastic grammars
2. Parsing algorithms
    a. CFGs and chart parsers (e.g., CYK)
    b. Probabilistic CFGs and weighted CYK
3. Representing meaning/Semantics
    a. Logic-based knowledge representations
    b. Semantic roles
    c. Temporal representations
    d. Beliefs, desires, and intentions
4. Corpus-based methods
5. N-grams and HMMs
6. Smoothing and backoff
7. Examples of use: POS tagging and morphology
8. Information retrieval (See also: DM-Unstructured)
    a. Vector space model
        i. TF & IDF
    b. Precision and recall
9. Information extraction
10. Language translation
11. Text classification, categorization
    a. Bag of words model
12. Deep learning for NLP (See also: AI-ML)
    a. RNNs
    b. Transformers
    c. Multi-modal embeddings (e.g., images + text)
    d. Generative language models

Illustrative Learning Outcomes:

1. Define and contrast deterministic and stochastic grammars, providing examples to show the adequacy of each.
2. Simulate, apply, or implement classic and stochastic algorithms for parsing natural language.
3. Identify the challenges of representing meaning.
4. List the advantages of using standard corpora. Identify examples of current corpora for a variety of NLP tasks.
5. Identify techniques for information retrieval, language translation, and text classification.
6. Implement a TF/IDF transform, use it to extract features from a corpus, and train an off-the-shelf machine learning algorithm using those features to do text classification.

AI-Robotics: Robotics

(See also: SPD-Robot)

Non-core:

1. Overview: problems and progress
    a. State-of-the-art robot systems, including their sensors and an overview of their sensor processing

   b. Robot control architectures, e.g., deliberative vs reactive control and Braitenberg vehicles
   c. World modeling and world models
   d. Inherent uncertainty in sensing and in control
2. Sensors and effectors
   a. Sensors: e.g., LIDAR, sonar, vision, depth, stereoscopic, event cameras, microphones, haptics,
   b. Effectors: e.g., wheels, arms, grippers
3. Coordinate frames, translation, and rotation (2D and 3D)
4. Configuration space and environmental maps
5. Interpreting uncertain sensor data
6. Localization and mapping
7. Navigation and control
8. Forward and inverse kinematics
9. Motion path planning and trajectory optimization
10. Manipulation and grasping
11. Joint control and dynamics
12. Vision-based control
13. Multiple-robot coordination and collaboration
14. Human-robot interaction (See also: HCI-User, HCI-Accessibility)
   a. Shared workspaces
   b. Human-robot teaming and physical HRI
   c. Social assistive robots
   d. Motion/task/goal prediction
   e. Collaboration and communication (explicit vs implicit, verbal or symbolic vs non-verbal or visual)
   f. Trust
15. Applications and Societal, Economic, and Ethical Issues
   a. Societal, economic, right-to-work implications
   b. Ethical and privacy implications of robotic applications
   c. Liability in autonomous robotics
   d. Autonomous weapons and ethics
   e. Human oversight and control

Illustrative Learning Outcomes:

(Note: Due to the expense of robot hardware, all of these could be done in simulation or with low-cost educational robotic platforms.)

1. List capabilities and limitations of today's state-of-the-art robot systems, including their sensors and the crucial sensor processing that informs those systems.
2. Integrate sensors, actuators, and software into a robot designed to undertake a specific task.
3. Program a robot to accomplish simple tasks using deliberative, reactive, and/or hybrid control architectures.
4. Implement fundamental motion planning algorithms within a robot configuration space.
5. Characterize the uncertainties associated with common robot sensors and actuators; articulate strategies for mitigating these uncertainties.
6. List the differences among robots' representations of their external environment, including their strengths and shortcomings.

7. Compare and contrast at least three strategies for robot navigation within known and/or unknown environments, including their strengths and shortcomings.
8. Describe at least one approach for coordinating the actions and sensing of several robots to accomplish a single task.
9. Compare and contrast a multi-robot coordination and a human-robot collaboration approach and attribute their differences to differences between the problem settings.
10. Analyze the societal, economic, and ethical issues of a real-world robotics application.

AI-Vision: Perception and Computer Vision

Non-core:

1. Computer vision
    a. Image acquisition, representation, processing, and properties
    b. Shape representation, object recognition, and segmentation
    c. Motion analysis
    d. Generative models
2. Audio and speech recognition
3. Touch and proprioception
4. Other modalities (e.g., olfaction)
5. Modularity in recognition
6. Approaches to pattern recognition (See also: AI-ML)
    a. Classification algorithms and measures of classification quality
    b. Statistical techniques
    c. Deep learning techniques

Illustrative Learning Outcomes:

1. Summarize the importance of image and object recognition in AI and indicate several significant applications of this technology.
2. List at least three image-segmentation approaches, such as thresholding, edge-based and region-based algorithms, along with their defining characteristics, strengths, and weaknesses.
3. Implement 2d object recognition based on contour-based and/or region-based shape representations.
4. Distinguish the goals of sound-recognition, speech-recognition, and speaker-recognition and identify how the raw audio signal will be handled differently in each of these cases.
5. Provide at least two examples of a transformation of a data source from one sensory domain to another, e.g., tactile data interpreted as single-band 2d images.
6. Implement a feature-extraction algorithm on real data, e.g., an edge or corner detector for images or vectors of Fourier coefficients describing a short slice of audio signal.
7. Implement an algorithm combining features into higher-level percepts, e.g., a contour or polygon from visual primitives or phoneme hypotheses from an audio signal.
8. Implement a classification algorithm that segments input percepts into output categories and quantitatively evaluates the resulting classification.
9. Evaluate the performance of the underlying feature-extraction, relative to at least one alternative possible approach (whether implemented or not) in its contribution to the classification task (8), above.

10. Describe at least three classification approaches, their prerequisites for applicability, their strengths, and their shortcomings.
11. Implement and evaluate a deep learning solution to problems in computer vision, such as object or scene recognition.

Professional Dispositions

- **Meticulousness:** Since attention must be paid to details when implementing AI and machine learning algorithms, students must be meticulous about detail.
- **Persistence:** AI techniques often operate in partially observable environments and optimization processes may have cascading errors from multiple iterations. Getting AI techniques to work predictably takes trial and error, and repeated effort. These call for persistence on the part of the student.
- **Inventive:** Applications of AI involve creative problem formulation and application of AI techniques, while balancing application requirements and societal and ethical issues.
- **Responsible:** Applications of AI can have significant impacts on society, affecting both individuals and large populations. This calls for students to understand the implications of work in AI to society, and to make responsible choices for when and how to apply AI techniques.

Mathematics Requirements

Required:

- Algebra
- Precalculus
- Discrete Math (See also: MSF-Discrete)
    - sets, relations, functions, graphs
    - predicate and first-order logic, logic-based proofs
- Linear Algebra (See also: MSF-Linear)
    - Matrix operations, matrix algebra
    - Basis sets
- Probability and Statistics (See also: MSF-Statistics)
    - Basic probability theory, conditional probability, independence
    - Bayes theorem and applications of Bayes theorem
    - Expected value, basic descriptive statistics, distributions
    - Basic summary statistics and significance testing
    - All should be applied to real decision-making examples with real data, not “textbook” examples.

Desirable:

- Calculus-based probability and statistics
- Calculus: single-variable and partial derivatives
- Other topics in probability and statistics
    - Hypothesis testing, data resampling, experimental design techniques
- Optimization
- Linear algebra (all other topics)

Course Packaging Suggestions

Artificial Intelligence to include the following:

- AI-Introduction (4 hours)
- AI-Search (9 hours)
- AI-KRR (4 hours)
- AI-ML (12 hours)
- AI-Probability (5 hours)
- AI-SEP (4 hours –integrated throughout the course)

Prerequisites:

- SDF-Fundamentals
- SDF-Data-Structures
- SDF-Algorithms
- MSF-Discrete
- MSF-Probability

Course objective: A student who completes this course should understand the basic areas of AI and be able to understand, develop, and apply techniques in each. They should be able to solve problems using search techniques, basic Bayesian reasoning, and simple machine learning methods. They should understand the various applications of AI and associated ethical and societal implications.

Machine Learning to include the following:

- AI-ML (32 hours)
- AI-KRR (4 hours)
- AI-NLP (4 hours – selected topics, e.g., TF-IDF, bag of words, and text classification)
- AI-SEP (4 hours – should be integrated throughout the course)

Prerequisites:

- SDF-Fundamentals
- SDF-Data-Structures
- SDF-Algorithms
- MSF-Discrete
- MSF-Probability
- MSF-Statistics
- MSF-Linear (optional)

Course objective: A student who completes this course should be able to understand, develop, and apply mechanisms for supervised, unsupervised, and reinforcement learning. They should be able to select the proper machine learning algorithm for a problem, preprocess the data appropriately, apply proper evaluation techniques, and explain how to interpret the resulting models, including the model's shortcomings. They should be able to identify and compensate for biased data sets and other sources of error and be able to explain ethical and societal implications of their application of machine learning to practical problems.

Robotics to include the following:

- AI-Robotics (25 hours)

- SPD-Robot (4 hours – focusing on hardware, constraints/considerations, and software architectures; other topics in SPD/Robot Platforms that overlap with AI/Robotics)
- AI-Search (4 hours – selected topics well-integrated with robotics, e.g., A* and path search)
- AI-ML (6 hours – selected topics well-integrated with robotics, e.g., neural networks for object recognition)
- AI-SEP (3 hours – integrated throughout the course; robotics is already a huge application, so this really should focus on societal impact and specific robotic applications).

Prerequisites:

- SDF-Fundamentals
- SDF-Data-Structures
- SDF-Algorithms
- MSF-Linear

Course objective: A student who completes this course should be able to understand and use robotic techniques to perceive the world using sensors, localize the robot based on features and a map, and plan paths and navigate in the world in simple robot applications. They should understand and be able to apply simple computer vision, motion planning, and forward and inverse kinematics techniques.

Introduction to Data Science to include the following:

- GIT-Visualization (6 hours) – types of visualization, libraries, foundations
- GIT-SEP (2 hours) – ethically responsible visualization
- DM-Core (2 hours) – Parallel and distributed processing (MapReduce, cloud frameworks, etc.)–
- DM-Modeling (2 hours) – Graph representations, entity resolution
- DM-Querying (4 hours) – SQL, query formation
- DM-NoSQL (2 hours) – Graph DBs, data lakes, data consistency
- DM-Security (2 hours) – privacy, personally identifying information and its protection
- DM-Analytics (1 hour) – exploratory data techniques, data science lifecycle
- DM-SEP (2 hours) – Data provenance
- AI-ML (15 hours) – Data preprocessing, missing data imputation, supervised/semi-supervised/unsupervised learning, text analysis, graph analysis and PageRank, experimental methodology, evaluation, and ethics
- AI-SEP (3 hours) – Applications specific to data science, interspersed throughout the course
- MSF-Statistics (3 hours) – Statistical analysis, hypothesis testing, experimental design

Prerequisites:

- SDF-Fundamentals

Course objective: A student who completes this course should be able to formulate questions as data analysis problems, understand and use statistical techniques to achieve that analysis from real data, apply visualization techniques to convey the results, and analyze the ethical and societal implications of data science applications. Students should also be able to understand and effectively use data management techniques for preprocessing, storage, security, and retrieval of data in current systems.

Committee

Chair: Eric Eaton, University of Pennsylvania, Philadelphia, PA, USA

Members:

- Zachary Dodds, Harvey Mudd College, Claremont, CA, USA
- Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, New York, NY, USA
- Laura Hiatt, US Naval Research Laboratory, Washington, DC, USA
- Amruth N. Kumar, Ramapo College of New Jersey, Mahwah, NJ, USA
- Peter Norvig, Google, Mountain View, CA, USA
- Meinolf Sellmann, GE Research, Niskayuna, NY, USA
- Reid Simmons, Carnegie Mellon University, Pittsburgh, PA, USA

Contributors:

- Nate Derbinsky, Northeastern University, Boston, MA, USA
- Eugene Freuder, Insight Centre for Data Analytics, University College Cork, Cork, Ireland
- Ashok Goel, Georgia Institute of Technology, Atlanta, GA, USA
- Claudia Schulz, Thomson Reuters, Zurich, Switzerland

Algorithmic Foundations (AL)

Preamble

Algorithms and data structures are fundamental to computer science, since every theoretical computation and applied program consists of algorithms that operate on data elements possessing some underlying structure. Selecting appropriate computational solutions to real-world problems benefits from understanding the theoretical and practical capabilities and limitations of available algorithms and paradigms, including their impact on the environment and society. Moreover, this understanding provides insight into the intrinsic nature of computation, computational problems, and computational problem-solving as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or other implementation aspects.

This knowledge area focuses on the nature of computation including the concepts and skills required to design and analyze algorithms for solving real-world computational problems. It complements the implementation of algorithms and data structures found in the Software Development Foundations (SDF) knowledge area. As algorithms and data structures are essential in all advanced areas of computer science, this area provides the algorithmic foundations that every computer science graduate is expected to know. Exposure to the breadth of these foundational AL topics is designed to provide students with the basis for studying these topics in more depth, for studying additional computation and algorithm topics, and for learning advanced algorithms across a variety of CS knowledge areas and CS+X disciplines.

Changes since CS2013

This area has been renamed from Algorithms and Complexity to better reflect its foundational scope since topics in this area focus on the practical and theoretical foundations of algorithms, complexity, and computability. These topics also provide the foundational prerequisites for advanced study in computer science. Additionally, topics focused on complexity and computability have been cleanly separated into respective knowledge units. To reinforce the important impact of computation on society, a Society, Ethics, and the Profession (AL-SEP) knowledge unit has been added with the expectation that SEP implications be addressed in some manner during every lecture hour of focus in this AL knowledge area.

The increase of four CS Core hours acknowledges the importance of this foundational area in the CS curriculum and returns it to the 2001 level (less than one course). Despite this increase, there is a significant overlap in hours with the Software Development Fundamentals (SDF) and Mathematical Foundations (MSF) areas. There is also a complementary nature of the units in this area since, for example, while linear search of an array covers topics in AL-Foundational, it can be used to simultaneously explain AL-Complexity $O(n)$ and AL-Strategies Brute-Force topics.
The KA topics and hours primarily reflect topics studied in a stand-alone computational theory course and the availability of additional hours when such a course is included in the curriculum.

Core Hours

Knowledge Unit	CS Core	KA Core
Foundational Data Structures and Algorithms	11	6
Algorithmic Strategies	6	
Complexity Analysis	6	3
Computational Models and Formal Languages	9	23
Society, Ethics, and the Profession	Included in SEP hours	
Total	**32**	**32**

The 11 CS Core hours in Foundational Data Structures and Algorithms are in addition to 9 hours counted in SDF and 3 hours counted in MSF.

Knowledge Units

AL-Foundational: Foundational Data Structures and Algorithms

CS Core: (See also: SDF-Data-Structures, SDF-Algorithms)

1. Abstract Data Type (ADT) and operations on an ADT (See also: FPL-Types)
   a. Dictionary operations (insert, delete, find)
2. Arrays
   a. Numeric vs non-numeric, character strings
   b. Single (vector) vs multidimensional (matrix)
3. Records/Structs/Tuples and Objects (See also: FPL-OOP)
4. Linked lists (for historical reasons)
   a. Single vs Double and Linear vs Circular
5. Stacks
6. Queues and deques
   a. Heap-based priority queue
7. Hash tables/maps
   a. Collision resolution and complexity (e.g., probing, chaining, rehash)
8. Graphs (e.g., [un]directed, [a]cyclic, [un]connected, and [un]weighted) (See also: MSF-Discrete)
   a. Graph representation: adjacency list vs matrix
9. Trees (See also: MSF-Discrete)
   a. Binary, n-ary, and search trees
   b. Balanced (e.g., AVL, Red-Black, Heap)
10. Sets (See also: MSF-Discrete)

11. Search algorithms
    a. O(n) complexity (e.g., linear/sequential array/list search)
    b. $O(\log_2 n)$ complexity (e.g., binary search)
    c. $O(\log_b n)$ complexity (e.g., uninformed depth/breadth-first tree search)
12. Sorting algorithms (e.g., stable, unstable)
    a. $O(n^2)$ complexity (e.g., insertion, selection),
    b. $O(n \log n)$ complexity (e.g., quicksort, merge, timsort)
13. Graph algorithms
    a. Shortest path (e.g., Dijkstra's, Floyd's)
    b. Minimal spanning tree (e.g., Prim's, Kruskal's)

KA Core:

14. Sorting algorithms
    a. $O(n \log n)$ complexity heapsort
    b. Pseudo $O(n)$ complexity (e.g., bucket, counting, radix)
15. Graph algorithms
    a. Transitive closure (e.g., Warshall's)
    b. Topological sort
16. Matching
    a. Efficient string matching (e.g., Boyer-Moore, Knuth-Morris-Pratt)
    b. Longest common subsequence matching
    c. Regular expression matching

Non-core:

17. Cryptography algorithms (e.g., SHA-256) (See also: SEC-Crypto)
18. Parallel algorithms (See also: PDC-Algorithms, FPL-Parallel)
19. Consensus algorithms (e.g., Blockchain) (See also: SEC-Crypto)
    a. Proof of work vs proof of stake (See also: SEP-Sustainability)
20. Quantum computing algorithms (See also: AL-Models, AR-Quantum)
    a. Oracle-based (e.g., Deutsch-Jozsa, Bernstein-Vazirani, Simon)
    b. Superpolynomial speed-up via QFT (e.g., Shor's)
    c. Polynomial speed-up via amplitude amplification (e.g., Grover's)
21. Fast-Fourier Transform (FFT) algorithm
22. Differential evolution algorithm

Illustrative Learning Outcomes:

CS Core:

1. For each ADT/Data-Structure in this unit
    a. Explain its definition, properties, representation(s), and associated ADT operations.
    b. Explain step-by-step how the ADT operations associated with the data structure transform it.
2. For each algorithm in this unit explain step-by-step how the algorithm operates.
3. For each algorithmic approach (e.g., sorting) in this unit apply a prototypical example of the approach (e.g., merge sort).

4. Given requirements for a problem, develop multiple solutions using various data structures and algorithms. Subsequently, evaluate the suitability, strengths, and weaknesses selecting an approach that best satisfies the requirements.
5. Explain how collision avoidance and collision resolution is handled in hash tables.
6. Explain factors beyond computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data.
7. Explain the heap property and the use of heaps as an implementation of a priority queue.

KA Core:

8. For each of the algorithms and algorithmic approaches in the KA Core topics:
    a. Explain a prototypical example of the algorithm, and
    b. Explain step-by-step how the algorithm operates.

Non-core:

9. An appreciation of quantum computation and its application to certain problems.

AL-Strategies: Algorithmic Strategies

CS Core:

1. Paradigms
    a. Brute-Force (e.g., linear search, selection sort, traveling salesperson, knapsack)
    b. Decrease-and-Conquer
        i. By a Constant (e.g., insertion sort, topological sort),
        ii. By a Constant Factor (e.g., binary search),
        iii. By a Variable Size (e.g., Euclid's)
    c. Divide-and-Conquer (e.g., binary search, quicksort, mergesort, Strassen's)
    d. Greedy (e.g., Dijkstra's, Kruskal's, Knapsack)
    e. Transform-and-Conquer
        i. Instance simplification (e.g., find duplicates via list presort)
        ii. Representation change (e.g., heapsort)
        iii. Problem reduction (e.g., least-common-multiple, linear programming)
        iv. Dynamic programming (e.g., Floyd's, Warshall, Bellman-Ford)
    f. Space vs time tradeoffs (e.g., hashing)
2. Handling exponential growth (e.g., heuristic A*, branch-and-bound, backtracking)
3. Iteration vs recursion (e.g., factorial, tree search)

KA Core:

4. Paradigms
    a. Approximation algorithms
    b. Iterative improvement (e.g., Ford-Fulkerson, simplex)
    c. Randomized/Stochastic algorithms (e.g., max-cut, balls and bins)

Non-core:

5. Quantum computing

Illustrative Learning Outcomes:

CS Core:

1. For each of the paradigms in this unit,
   a. Explain its definitional characteristics,
   b. Explain an example that demonstrates the paradigm including how this example satisfies the paradigm's characteristics.
2. For each of the algorithms in the AL-Foundational unit, explain the paradigm used by the algorithm and how it exemplifies this paradigm.
3. Given an algorithm, explain the paradigm used by the algorithm and how it exemplifies this paradigm.
4. Give a real-world problem, evaluate appropriate algorithmic paradigms and algorithms from these paradigms that address the problem including evaluating the tradeoffs among the paradigms and algorithms selected.
5. Give examples of iterative and recursive algorithms that solve the same problem, explain the benefits and disadvantages of each approach.
6. Evaluate whether a greedy approach leads to an optimal solution.
7. Explain various approaches for addressing computational problems whose algorithmic solutions are exponential.

AL-Complexity: Complexity

CS Core:

1. Complexity Analysis Framework
   a. Best, average, and worst-case performance of an algorithm
   b. Empirical and relative (Order of Growth) measurements
   c. Input size and primitive operations
   d. Time and space efficiency
2. Asymptotic complexity analysis (average and worst-case bounds)
   a. Big-O, Big-Omega, and Big-Theta formal notations
   b. Foundational Complexity Classes and Representative Examples/Problems
      i. $O(1)$ *Constant* (e.g., array access)
      ii. $O(\log_2 n)$ *Logarithmic* (e.g., binary search)
      iii. $O(n)$ *Linear* (e.g., linear search)
      iv. $O(n \log_2 n)$ *Log Linear* (e.g., mergesort)
      v. $O(n^2)$ *Quadratic* (e.g., selection sort)
      vi. $O(n^c)$ *Polynomial* (e.g., $O(n^3)$ Gaussian elimination)
      vii. $O(2^n)$ *Exponential* (e.g., Knapsack, Satisfiability (SAT), Traveling Sales-Person (TSP), all subsets)
      viii. $O(n!)$ *Factorial* (e.g., Hamiltonian circuit, all permutations)
3. Empirical measurements of performance
4. Tractability and intractability
   a. P, NP, and NP-Complete Complexity Classes
   b. NP-Complete Problems (e.g., SAT, Knapsack, TSP)
   c. Reductions
5. Time and space tradeoffs in algorithms

KA Core:

6. Little-o, Little-Omega, and Little Theta notations
7. Formal recursive analysis
8. Amortized analysis
9. Turing Machine-based models of complexity
    a. Time complexity
        i. P, NP, NP-C, and EXP classes
        ii. Cook-Levin theorem
    b. Space Complexity
        i. NSpace and PSpace
        ii. Savitch's theorem

Illustrative Learning Outcomes:

CS Core:

1. Prepare a presentation that explains to first year students the basic concepts of algorithmic complexity including best, average, and worst-case algorithm behavior, Big- O, Omega, and Theta notations, complexity classes, time and space tradeoffs, empirical measurement, and impact on practical problems.
2. Using examples, explain each of the foundational complexity classes in this unit.
3. For each foundational complexity class in this unit, explain an algorithm that demonstrates the associated runtime complexity.
4. For each algorithm in the AL-Foundational unit, explain its runtime complexity class and why it belongs to this class.
5. Informally evaluate the foundational complexity class of simple algorithms.
6. Given a problem to program for which there may be several algorithmic approaches, evaluate them and determine which are feasible, and select one that is optimal in implementation and run-time behavior.
7. Develop empirical studies to determine and validate hypotheses about the runtime complexity of various algorithms by running algorithms on input of various sizes and comparing actual performance to the theoretical analysis.
8. Explain examples that illustrate time-space tradeoffs of algorithms.
9. Explain how tree balance affects the efficiency of binary search tree operations.
10. Explain to a non-technical audience the significance of tractable versus intractable algorithms using an intuitive explanation of Big-O complexity.
11. Explain the significance of NP-Completeness.
12. Explain how NP-Hard is a lower bound and NP is an upper bound for NP-Completeness.
13. Explain examples of NP-complete problems.

KA Core:

14. Use recurrence relations to evaluate the time complexity of recursively defined algorithms.
15. Apply elementary recurrence relations using a form of the Master Theorem.
16. Apply Big-O notation to give upper case bounds on time/space complexity of algorithms.
17. Explain the Cook-Levin Theorem and the NP-Completeness of SAT.

18. Explain the classes P and NP.
19. Prove that a problem is NP-Complete by reducing a classic known NP-C problem to it (e.g., 3SAT and Clique).
20. Explain the P-space class and its relation to the EXP class.

AL-Models: Computational Models and Formal Languages

CS Core:

1. Formal automata
    a. Finite State
    b. Pushdown
    c. Linear Bounded
    d. Turing Machine
2. Formal languages, grammars and Chomsky Hierarchy
    (See also: FPL-Translation, FPL-Syntax)
    a. Regular (Type-3)
        i. Regular Expressions
    b. Context-Free (Type-2)
    c. Context-Sensitive (Type-1)
    d. Recursively Enumerable (Type-0)
3. Relations among formal automata, languages, and grammars
4. Decidability, (un)computability, and halting
5. The Church-Turing thesis
6. Algorithmic correctness
    a. Invariants (e.g., in iteration, recursion, tree search)

KA Core:

7. Deterministic and nondeterministic automata
8. Pumping Lemma proofs
    a. Proof of Finite State/Regular-Language limitation
    b. Pushdown Automata/Context-Free-Language limitation
9. Decidability
    a. Arithmetization and diagonalization
10. Reducibility and reductions
11. Time complexity based on Turing Machine
12. Space complexity (e.g., Pspace, Savitch's Theorem)
13. Equivalent models of algorithmic computation
    a. Turing Machines and Variations (e.g., multi-tape, non-deterministic)
    b. Lambda Calculus (See also: FPL-Functional)
    c. Mu-Recursive Functions

Non-core:

14. Quantum computation (See also: AR-Quantum)
    a. Postulates of quantum mechanics
        i. State space

ii. State evolution
iii. State composition
iv. State measurement

b. Column vector representations of qubits
c. Matrix representations of quantum operations
d. Simple quantum gates (e.g., XNOT, CNOT)

Illustrative Learning Outcomes:

CS Core:

1. For each formal automaton in this unit:
   a. Explain its definition comparing its characteristics with this unit's other automata,
   b. Using an example, explain step-by-step how the automaton operates on input including whether it accepts the associated input,
   c. Explain an example of inputs that can and cannot be accepted by the automaton.
2. Given a problem, develop an appropriate automaton that addresses the problem.
3. Develop a regular expression for a given regular language expressed in natural language.
4. Explain the difference between regular expressions (Type-3 acceptors) and the regular expressions (Type-2 acceptors) used in programming languages.
5. For each formal model in this unit:
   a. Explain its definition comparing its characteristics with the others in this unit,
   b. Explain example inputs that are and cannot be accepted by the language/grammar.
6. Explain a universal Turing Machine and its operation.
7. Present to an audience of co-workers and managers the impossibility of providing them a program that checks all other programs, including some seemingly simple ones, for infinite loops including an explanation of the Halting problem, why it has no algorithmic solution, and its significance for real-world algorithmic computation.
8. Explain examples of classic uncomputable problems.
9. Explain the Church-Turing Thesis and its significance for algorithmic computation.
10. Explain how (loop) invariants can be used to prove the correctness of an algorithm.

Illustrative Learning Outcomes:

KA Core:

11. For each formal automaton in this unit explain (compare/contrast) its deterministic and nondeterministic capabilities.
12. Apply pumping lemmas, or alternative means, to prove the limitations of Finite State and Pushdown automata.
13. Apply arithmetization and diagonalization to prove the Halting Problem for Turing Machines is Undecidability.
14. Given a known undecidable language, apply a mapping reduction or computational history to prove that another language is undecidable.
15. Convert among equivalently powerful notations for a language, including among DFAs, NFAs, and regular expressions, and between PDAs and CFGs.
16. Explain Rice's theorem and its significance.

17. Explain an example proof of a problem that is uncomputable by reducing a classic known uncomputable problem to it.
18. Explain the Primitive and General Recursive functions (zero, successor, selection, primitive recursion, composition, and Mu), their significance, and Turing Machine implementations.
19. Explain how computation is performed in Lambda Calculus (e.g., Alpha conversion and Beta reduction)

Non-core:

20. For a quantum system give examples that explain the following postulates.
    a. State Space – system state represented as a unit vector in Hilbert space,
    b. State Evolution – the use of unitary operators to evolve system state,
    c. State Composition – the use of tensor product to compose systems states,
    d. State Measurement – the probabilistic output of measuring a system state.
21. Explain the operation of a quantum XNOT or CNOT gate on a quantum bit represented as a matrix and column vector, respectively.

AL-SEP: Society, Ethics, and the Profession

CS Core: (See also: SEP-Context, SEP-Sustainability)

1. Social, ethical, and secure algorithms
2. Algorithmic fairness
3. Anonymity (e.g., Differential Privacy)
4. Accountability/Transparency
5. Responsible algorithms
6. Economic and other impacts of inefficient algorithms
7. Sustainability

KA Core:

8. Context aware computing

Illustrative Learning Outcomes:

CS Core:

1. Develop algorithmic solutions to real-world societal problems, such as routing an ambulance to a hospital.
2. Explain the impact that an algorithm may have on the environment and society when used to solve a real-world problem while considering its sustainability and that it can affect different societal groups in different ways.
3. Prepare a presentation that justifies the selection of appropriate data structures and/or algorithms to solve a given real-world problem.
4. Explain an example that articulates how differential privacy protects knowledge of an individual's data.
5. Explain the environmental impacts of design choices that relate to algorithm design.
6. Explain the tradeoffs involved in proof-of-work and proof-of-stake algorithms.

Professional Dispositions

- **Meticulous:** As an algorithm is a formal solution to a computational problem, attention to detail is important when developing and combining algorithms.
- **Persistent:** As developing algorithmic solutions to computational problems can be challenging, computer scientists must be resolute in pursuing such solutions.
- **Inventive:** As computer scientists develop algorithmic solutions to real-world problems, they must be inventive in developing solutions to these problems.

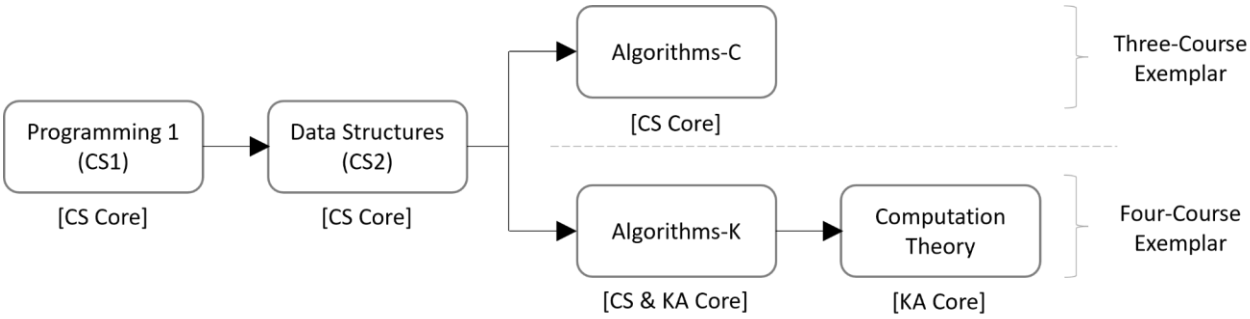
Mathematics Requirements

Required:

- MSF-Discrete

Course Packaging Suggestions

As depicted in the following figure, the committee envisions two common approaches for addressing foundational AL topics in CS courses. Both approaches included the required introductory Programming (CS1) and Data Structures (CS2) courses. In a three-course approach, all CS Core topics are covered with additional unused hours to cover other topics. Alternatively, in the four-course approach, the AL-Model knowledge unit CS and KA Core topics are addressed in a Computational Theory focused course, which leaves room to address additional KA topics in the third Algorithms course. Both approaches assume Big-O analysis is introduced in the Data Structures (CS2) course and that graphs are taught in the third Algorithms course. The committee recognizes that there are many different approaches for packaging AL topics into courses including, for example, introducing graphs in CS2 Data Structures, backtracking in an AI course, and AL-Model topics in a theory course that also addresses, for instance, FPL topics. The given example is simply one way to cover the entire AL CS Core in three introductory courses with additional lecture hours to spare.



Courses Common to Three and Four Course Exemplars

Programming 1 (CS1)

- AL-Foundational (2 hours)
  - Arrays and Strings
  - Search Algorithms (e.g., $O(n)$ Linear Search)

- AL-SEP (In SEP hours)

Note: the following AL topics are demonstrated in CS1, but not explicitly taught as such:

- AL-Strategies (less than hour)
    - Brute Force (e.g., linear search)
    - Iteration (e.g., linear search)
- AL-Complexity (less than 1 hour)
    - Foundational Complexity Classes
        - $O(1)$ *Constant* and $O(n)$ *Linear* runtime complexities

Course objectives: Students should be able to explain, evaluate, and apply arrays in a variety of problem-solving contexts including using linear search for elements in an array. They should also be able to begin to explain the impact algorithmic design and use has on society.

Data Structures (CS2)

- AL-Foundational (12 hours)
    - Abstract Data Types and Operations (ADTs)
    - Binary Search
    - Multi-dimensional Arrays
    - Linked Lists
    - Hash Tables/Maps including conflict resolution strategies
    - Records/Structs/Tuples and Objects
    - Sets
    - Stacks, Queues, and Deques
    - Trees: Binary, Ordered, Breadth- and Depth-first search
    - Search Algorithms (e.g., $O(n^2)$ Selection Sort, $O(\log_2 n)$ binary search)
    - Sorting Algorithms (e.g., $O(n \log n)$ Mergesort, $O(\log_b n)$ tree search)
- AL-Strategies (3 hours)
    - Brute Force (e.g., selection sort)
    - Decrease-and-Conquer (e.g., depth/breadth tree search)
    - Divide-and-Conquer (e.g., mergesort, quicksort)
    - Iteration vs Recursion (e.g., factorial, tree search)
    - Space vs Time tradeoff (e.g., hashing)
- AL-Complexity (3 hours)
    - Complexity Analysis Framework
    - Foundational Complexity Classes
        - $O(\log_2 n)$ *Logarithmic*, $O(n \log_2 n)$ *Log Linear,* and $O(n^2)$ *Quadratic*
    - Time and Space Tradeoffs in Algorithms
- AL-SEP (In SEP hours)

Course objectives: Students should be able to explain, evaluate, and apply the specified data structures and algorithms in a variety of problem-solving contexts. Additionally, they should be able demonstrate the use of different data structures, algorithms, and algorithmic strategies (paradigms) to

solve the same problem. Also, they will continue to enhance and refine their understanding of the impact that algorithmic design and use has on society.

Three Course Exemplar Approach
Algorithms-C

- AL-Foundational (3 hours)
  - Graphs including Graph Algorithms
- AL-Complexity (3 hours)
  - Asymptotic Complexity Analysis
  - Foundational Complexity Classes
    - $O(2^n)$ *Exponential* and $O(n!)$ *Factorial*
  - Empirical Measurements of Performance
  - Tractability and Intractability
- AL-Strategies (3 hours)
  - Brute Force (e.g., traveling salesperson, knapsack)
  - Decrease-and-Conquer (e.g., topological sort)
  - Divide-and-Conquer (e.g., Strassen's)
  - Greedy (e.g., Dijkstra's, Kruskal's)
  - Transform-and-Conquer/Reduction (e.g., heapsort, trees (2-3, AVL, Red-Black))
    - Dynamic Programming (e.g., Warshall's, Floyd's, Bellman-Ford)
  - Handling Exponential Growth (e.g., heuristic A*, branch-and-bound, backtracking)
- AL-Models (9 hours)
  - All CS Core topics
- AL-SEP (In SEP hours)

Course objectives: Students should be able to explain, evaluate, and apply the specified data structures and algorithms in a variety of problem-solving contexts. Additionally, they should be able to formally explain complexity analysis and the importance of tractability including approaches for handling intractable problems. Finally, they should also be able to summarize formal models of computation, grammars, and languages including the definition of a computer as a Turing Machine and the undecidability of the Halting problem.

Four Course Exemplar Approach
Algorithms-C (third course)

- AL-Foundational (3 hours)
  - Graphs including Graph Algorithms
  - Sorting Algorithms
    - $O(n \log n)$ heapsort
    - Pseudo $O(n)$ complexity (e.g., bucket, counting, radix)
  - Graph Algorithms
    - Transitive closure (e.g., Warshall's)
    - Topological sort
  - Matching
    - Efficient String Matching (e.g., Boyer-Moore, Knuth-Morris-Pratt)

        - Longest common subsequence matching
        - Regular expression matching
- AL-Complexity (3 hours)
    - Asymptotic Complexity Analysis
    - Foundational Complexity Classes
        - $O(2^n)$ *Exponential* and $O(n!)$ *Factorial*
    - Empirical Measurements of Performance
    - Tractability and Intractability
- AL-Strategies (3 hours)
    - Brute Force (e.g., traveling salesperson, knapsack)
    - Decrease-and-Conquer (e.g., topological sort)
    - Divide-and-Conquer (e.g., Strassen's algorithm)
    - Greedy (e.g., Dijkstra's, Kruskal's)
    - Transform-and-Conquer/Reduction (e.g., heapsort, trees (2-3, AVL, Red-Black))
        - Dynamic Programming (e.g., Warshall's, Floyd's, Bellman-Ford)
    - Handling Exponential Growth (e.g., heuristic A*, branch-and-bound, backtracking)

Course objectives: Students should be able to explain, evaluate, and apply the specified data structures and algorithms in a variety of problem-solving contexts. Additionally, they should be able to formally explain complexity analysis and the importance of tractability including approaches for handling intractable problems.

Computation Theory (fourth course)

- AL-Complexity (3 hours)
    - Turing Machine-based models of complexity (P, NP, and NP-C classes)
    - Space complexity (NSpace, PSpace Savitch' Theorem)
- AL-Models (29 hours)
    - All CS and KA Core topics
- AL-SEP (In SEP hours)

Course objectives: Students should be able to explain, evaluate, and apply models of computation, grammars, and languages. Additionally, they should be able to explain formal proofs that demonstrate the capability and limitations of various automata. Students should be able to relate the complexity of Random Access Models of Computation to Turing Machine models. Finally, students should be able to summarize decidability and reduction proofs.

Committee

Chair: Richard Blumenthal, Regis University, Denver, CO, USA

Members:

- Cathy Bareiss, Bethel University, Mishawaka, MN, USA
- Tom Blanchet, SciTec, Inc., Boulder, CO, USA
- Doug Lea, State University of New York at Oswego, Oswego, NY, USA

- Sara Miner More, John Hopkins University, Baltimore, MD, USA
- Mia Minnes, University of California San Diego, San Diego, CA, USA
- Atri Rudra, University at Buffalo, Buffalo, NY, USA
- Christian Servin, El Paso Community College, El Paso, TX, USA

Architecture and Organization (AR)

Preamble

Computing professionals spend considerable time writing efficient code to solve a particular problem in an application domain. As the shift from sequential to parallel processing occurs, a deeper understanding of the underlying computer architectures is necessary. Architecture can no longer be viewed as a black box where principles from one architecture can be applied to another. Instead, programmers should look inside the black box and use specific components to enhance system performance and energy efficiency.

The Architecture and Organization (AR) knowledge area aims to develop a deeper understanding of the hardware environments upon which almost all computing is based, and the relevant interfaces provided to higher software layers. The target hardware comprises low-end embedded system processors up to high-end enterprise multiprocessors.

The topics in this knowledge area will benefit students by enabling them to appreciate the fundamental architectural principles of modern computer systems, including the challenge of harnessing parallelism to sustain performance and energy improvements into the future. This KA will help computer science students depart from the black box approach and become more aware of the underlying computer system and the efficiencies specific architectures can achieve.

Changes since CS2013

Changes and additions are summarized as follows.

- Topics have been revised, particularly AR/Memory Hierarchy and AR/Performance and Energy Efficiency. This update brings recent advances in memory caching and energy consumption.
- The newly created AR/Heterogeneous Architectures covers emerging topics in Computer Architecture: Processing In-Memory (PIM) and domain-specific architectures (e.g., neural network processors).
- The new AR/Quantum Architectures offers a "toolbox" covering introductory topics in quantum computing.
- Knowledge units have been merged to better deal with overlaps:
    - AR/Multiprocessing and Alternative Architectures were merged into newly created AR/Heterogeneous Architectures.
- The new AR/Secure Processor Architectures covers hardware support for multi-stack security applications.

Core Hours

Knowledge Unit	CS Core		KA Core
Digital Logic and Digital Systems			2 + 1 (SF)

Machine-Level Data Representation	1		
Assembly Level Machine Organization	1		1 + 1 (PDC)
Memory Hierarchy	4+2 (OS)		
Interfacing and Communication	1		
Functional Organization			2
Performance and Energy Efficiency			3
Heterogeneous Architectures			2
Secure Processor Architectures			2
Quantum Architectures			2
Sustainability Issues	Included in SEP hours		
Total	**9**		**16**

The hours shared with OS include overlapping topics and are counted here.

Knowledge Units

AR-Logic: Digital Logic and Digital Systems

KA Core:

1. Combinational vs sequential logic/field programmable gate arrays (FPGAs) (See also: SF-Overview, SF-Foundations, SPD-Embedded)
   a. Fundamental combinational
   b. Sequential logic building block
2. Computer-aided design tools that process hardware and architectural representations
3. High-level synthesis
   a. Register transfer notation
   b. Hardware description language (e.g., Verilog/VHDL/Chisel)
4. System-on-chip (SoC) design flow
5. Physical constraints
   a. Gate delays
   b. Fan-in and fan-out
   c. Energy/power
   d. Speed of light

Illustrative Learning Outcomes:

KA Core:

1. Discuss the progression of computer technology components from vacuum tubes to VLSI, from mainframe computer architectures to the organization of warehouse-scale computers.
2. Describe parallelism and data dependencies between and within components in a modern heterogeneous computer architecture.
3. Explain the relationship between parallelism and power consumption.
4. Construct the design of basic building blocks for a computer: arithmetic-logic unit (gate-level), registers (gate-level), central processing unit (register transfer-level), and memory (register transfer-level).
5. Evaluate simple building blocks (e.g., arithmetic-logic unit, registers, movement between registers) of a simple computer design.
6. Analyze the timing behavior of a pipelined processor, identifying data dependency issues.

AR-Representation: Machine-Level Data Representation

CS Core:

1. Overview and history of computer architecture (See also: SPD-Game)
2. Bits, bytes, and words
3. Unsigned, signed and two's complement representations
4. Numeric data representation and number bases
   a. Fixed-point
   b. Floating-point
5. Representation of non-numeric data
6. Representation of records, arrays and UTF data types (See also: AL-Foundational)

Illustrative Learning Outcomes:

CS Core:

1. Discuss why everything in computers are data, including instructions.
2. Explain how fixed-length number representations can affect accuracy and precision.
3. Describe how negative integers are stored in sign-magnitude and two's-complement representations.
4. Discuss how different formats can represent numerical data.
5. Explain the bit-level representation of non-numeric data, such as characters, strings, records, and arrays.
6. Translate numerical data from one format to another.
7. Describe how a single adder (without overflow detection) can handle both signed (two's complement) and unsigned (binary) input without "knowing" which format a given input is using.

AR-Assembly: Assembly Level Machine Organization

CS Core:

1. von Neumann machine architecture
2. Control unit: instruction fetch, decode, and execution (See also: OS-Principles)
3. Introduction to SIMD vs MIMD and the Flynn taxonomy (See also: PDC-Programs, OS-Scheduling, OS-Process)

4. Shared memory multiprocessors/multicore organization (See also: PDC-Programs, OS-Scheduling)

KA Core:

5. Instruction set architecture (ISA) (e.g., x86, ARM and RISC-V)
    a. Fixed vs variable-width instruction sets
    b. Instruction formats
    c. Data manipulation, control, I/O
    d. Addressing modes
    e. Machine language programming
    f. Assembly language programming
6. Subroutine call and return mechanisms (See also: FPL-Translation, OS-Principles)
7. I/O and interrupts (See also: OS-Principles)
8. Heap, static, stack, and code segments (See also: FPL-Translation, OS-Process)

Illustrative Learning Outcomes:

CS Core:

1. Discuss how the classical von Neumann functional units are implemented in embedded systems, particularly on-chip and off-chip memory.
2. Describe how instructions are executed in a classical von Neumann machine, with extensions for threads, multiprocessor synchronization, and SIMD execution.
3. Assess an example diagram with instruction-level parallelism and hazards to describe how they are managed in typical processor pipelines.

KA Core:

4. Discuss how instructions are represented at the machine level and in the context of a symbolic assembler.
5. Map an example of high-level language patterns into assembly/machine language notations.
6. Contrast different instruction formats considering aspects such as addresses per instruction and variable-length vs fixed-length formats.
7. Analyze a subroutine diagram to comment on how subroutine calls are handled at the assembly level.
8. Describe basic concepts of interrupts and I/O operations.
9. Write a simple assembly language program for string/array processing and manipulation.

AR-Memory: Memory Hierarchy

CS Core:

1. Memory hierarchy: the importance of temporal and spatial locality (See also: SF-Performance, OS-Memory)
2. Main memory organization and operations (See also: OS-Memory)
3. Persistent memory (e.g., SSD, standard disks)
4. Latency, cycle time, bandwidth, and interleaving (See also: SF-Performance)
5. Cache memories (See also: SF-Performance)
    a. Address mapping
    b. Block size

    c. Replacement and store policy
    d. Prefetching
6. Multiprocessor cache coherence (See also: OS-Scheduling)
7. Virtual memory (hardware support) (See also: OS-Memory)
8. Fault handling and reliability (See also: SF-Reliability)
9. Reliability (See also: SF-Reliability, OS-Faults)
    a. Error coding
    b. Data compression
    c. Data integrity

KA Core:

10. Processing In-Memory (PIM)

Illustrative Learning Outcomes:

CS Core:

1. Using a memory system diagram, identify the main types of memory technology (e.g., SRAM, DRAM) and their relative cost and performance.
2. Measure the effect of memory latency on running time.
3. Enumerate the functions of a system with virtual memory management.
4. Compute average memory access time under various cache and memory configurations and mixes of instruction and data references.

AR-IO: Interfacing and Communication

CS Core:

1. I/O fundamentals (See also: OS-Devices, PDC-Communication)
    a. Handshaking and buffering
    b. Programmed I/O
    c. Interrupt-driven I/O (See also: OS-Principles)
2. Interrupt structures: vectored and prioritized, interrupt acknowledgment (See also: OS-Principles)
3. I/O devices (e.g., mouse, keyboard, display, camera, sensors, actuators) (See also: GIT-Fundamentals, GIT-Interaction, OS-Advanced-Files, PDC-Programs)
4. External storage, physical organization, and drives
5. Buses fundamentals (See also: OS-Devices)
    a. Bus protocols
    b. Arbitration
    c. Direct-memory access (DMA)

Illustrative Learning Outcomes:

CS Core:

1. Analyze an interrupt control diagram to comment on how interrupts are used to implement I/O control and data transfers.
2. Enumerate various types of buses in a computer system.
3. List the advantages of magnetic disks and contrast them with those of solid-state disks.

AR-Organization: Functional Organization

KA Core:

1. Implementation of simple datapaths, including instruction pipelining, hazard detection, and resolution (e.g., stalls, forwarding)
2. Control unit
   a. Hardwired implementation
   b. Microprogrammed realization
3. Instruction pipelining (See also: SF-Overview)
4. Introduction to instruction-level parallelism (ILP) (See also: PDC-Programs)

Illustrative Learning Outcomes:

KA Core:

1. Compare alternative implementation of datapaths in modern computer architectures.
2. Produce a set of control signals for adding two integers using hardwired and microprogrammed implementations.
3. Discuss instruction-level parallelism using pipelining and significant hazards that may occur.
4. Design a complete processor, including datapath and control.
5. Compute the average cycles per instruction for a given processor and memory system implementation.

AR-Performance-Energy: Performance and Energy Efficiency

KA Core:

1. Performance-energy evaluation (introduction): performance, power consumption, memory, and communication costs (See also: SF-Evaluation, OS-Scheduling, SPD-Game)
2. Branch prediction, speculative execution, out-of-order execution, Tomasulo's algorithm
3. Enhancements for vector processors and GPUs (See also: SPD-Game)
4. Hardware support for multithreading (See also: OS-Concurrency, OS-Scheduling, PDC-Programs)
   a. Race conditions
   b. Lock implementations
   c. Point-to-point synchronization
   d. Barrier implementation
5. Scalability
6. Alternative architectures including VLIW/EPIC, accelerators, and other special purpose processors
7. Dynamic voltage and frequency scaling (DVFS)
8. Dark Silicon

Illustrative Learning Outcomes:

KA Core:

1. Discuss performance and energy efficiency evaluation metrics.
2. Analyze a speculative execution diagram and write about the decisions that can be made.
3. Create a GPU performance-watt benchmarking diagram.
4. Write a multithreaded program that adds (in parallel) elements of two integer vectors.
5. Recommend a set of design choices for alternative computer architectures.

6. Enumerate key concepts associated with dynamic voltage and frequency scaling.
7. Measure energy savings improvement for an 8-bit integer quantization compared to a 32-bit quantization.

AR-Heterogeneity: Heterogeneous Architectures

KA Core:

1. SIMD and MIMD architectures (e.g., General-Purpose GPUs, TPUs, and NPUs) (See also: PDC-Programs, SPD-Embedded, GIT-Shading, SPD-Game)
2. Heterogeneous memory systems (See also: OS-Process, PDC-Communication)
   a. Shared memory versus distributed memory
   b. Volatile vs non-volatile memory
   c. Coherence protocols
3. Domain-Specific Architectures (DSAs) (See also: HCI-Accountability, GIT-Shading)
   a. Machine Learning Accelerator
   b. In-networking computing (See also: NC-Applications)
   c. Embedded systems for emerging applications
   d. Neuromorphic computing
   e. Edge computing devices
4. Packaging and integration solutions such as 3DIC and chiplets
5. Machine learning in architecture design
   a. AI algorithms for workload analysis
   b. Optimization of architecture configurations for performance and power efficiency

Illustrative Learning Outcomes:

KA Core

1. Analyze a system diagram with alternative parallel architectures, e.g., SIMD and MIMD, and identify the key differences.
2. Discuss what memory-management issues are found in multiprocessors that are not present in uniprocessors and how these issues might be resolved.
3. Indicate the differences between memory backplane, processor memory interconnect, and remote memory via networks, their implications for access latency, and their impact on program performance.
4. Discuss how you would determine when to use a domain-specific accelerator instead of a general-purpose CPU.
5. Enumerate key differences in architectural design principles between a vector and scalar-based processing unit.
6. List the advantages and disadvantages of a PIM architecture.

AR-Security: Secure Processor Architectures

KA core:

1. Principles of Secure Hardware
   a. Security Risk Analysis, Asset Protection, and Threat Model
   b. Cryptographic Acceleration with Hardware (See also: SEC-Crypto)

    c. Support for virtualization (e.g., OS isolation)
2. Roots of trust in hardware, Physically Unclonable Functions (PUF)
3. Hardware Random Number Generators
4. Memory protection extensions
    a. Runtime pointer bounds checking (e.g., buffer overflow)
    b. Protection at the microarchitectural level
    c. Protection at the ISA level
5. Trusted Execution Environment (TEE)
    a. Trusted Computer Base Protections
    b. Protecting virtual machines
    c. Protecting containers
    d. Trusted software modules (Enclaves)
6. Homomorphic encryption for privacy-preserving data processing

Illustrative Learning Outcomes

KA Core:

1. Discuss principles of secure hardware, exploring a framework for risk analysis and asset protection.
2. Summarize how Physically Unclonable Functions (PUF) can be a unique device identifier in security applications.
3. Distinguish a random number generator with dedicated hardware support from generators without hardware dedicated to generating entropy.
4. List the advantages and disadvantages of memory protection at the ISA level.
5. Describe key design issues of a trusted execution environment (TEE) to support virtual machines.

AR-Quantum: Quantum Architectures

KA Core:

1. Principles (See also: AL-Models: 8)
    a. The wave-particle duality principle
    b. The uncertainty principle in the double-slit experiment
    c. What is a Qubit? Superposition, interference, and measurement. Photons as qubits
    d. Systems of two qubits, Entanglement, Bell states, The No-Signaling theorem
2. Axioms of QM: superposition principle, measurement axiom, unitary evolution
3. Single qubit gates for the circuit model of quantum computation: X, Z, H
4. Two qubit gates and tensor products, working with matrices
5. The No-Cloning Theorem. The Quantum Teleportation protocol
6. Algorithms (See also: AL-Foundational)
    a. Simple quantum algorithms: Bernstein-Vazirani, Simon's algorithm
    b. Implementing Deutsch-Josza with Mach-Zehnder Interferometers
    c. Quantum factoring (Shor's Algorithm)
    d. Quantum search (Grover's Algorithm)
7. Implementation aspects (See also: SPD-Interactive)
    a. The physical implementation of qubits
    b. Classical control of a Quantum Processing Unit (QPU)
    c. Error mitigation and control, NISQ and beyond

   d. Measurement approaches
8. Emerging Applications
   a. Post-quantum encryption
   b. The Quantum Internet
   c. Adiabatic quantum computation (AQC) and quantum annealing

Illustrative Learning Outcomes:
KA Core:
1. Discuss how a quantum object produced as a particle propagates like a wave and is detected as a particle with a probability distribution corresponding to the wave.
2. Discuss the quantum-level nature that is inherently probabilistic.
3. Express your view on entanglement that can be used to create non-classical correlations, but there is no way to use quantum entanglement to send messages faster than the speed of light.
4. Describe quantum parallelism and the role of constructive vs destructive interference in quantum algorithms given the probabilistic nature of measurement(s).
5. Analyze a code snippet providing the role of quantum Fourier transform (QFT) in Shor's algorithm.
6. Write a program to implement Shor's algorithm in a simulator, highlighting the classical components and aspects of Shor's algorithm.
7. Enumerate the specifics of each qubit modality (e.g., trapped ion, superconducting, silicon spin, photonic, quantum dot, neutral atom, topological, color center, electron-on-helium).
8. Contrast AQC with the gate model of quantum computation and the problems each is better suited to solve.

AR-SEP: Sustainability Issues

Non-core:
1. Environmental impacts of implementation decisions
   a. Sustainability goals, resource consumption, and economic viability
   b. Carbon footprint, hardware electronic waste
   c. The energy footprint of data centers at various workloads (e.g., AI model training and use)
   d. Guidelines for sustainable design standards

Illustrative Learning Outcomes:
Non-core:
1. Assess the environmental impacts of a given project's deployment (e.g., the energy consumption of CPUs and GPUs, contribution to e-waste, and effect of hardware virtualization in data centers).

Professional Dispositions

- **Self-directed:** Students should increasingly become self-motivated to acquire complementary knowledge.
- **Proactive:** Students should exercise control and anticipate issues related to the underlying computer system.

Mathematics Requirements

- MSF-Discrete, MSF-Linear, MSF-Statistics, MSF-Calculus, MSF-Probability

Course Packaging Suggestions

Computer Architecture - Introductory Course to include the following:

- SEP-History (2 hours)
- AR-Representation (2 hours)
- AR-Assembly (2 hours)
- AR-Memory (10 hours)
- OS-Memory (10 hours)
- AR-IO (4 hours)
- AR-Heterogeneity (5 hours)
- PDC-Programs (4 hours)
- SEP-Ethical-Analysis (3 hours)

Course objectives: Students should understand the fundamentals of modern computer architectures, including the challenges associated with memory caches, memory management, and pipelining.

Prerequisites:

- MSF-Discrete

Computer Architecture - Advanced Topics Course to include the following:

- AR-Logic (4 hours)
- AR-Representation (2 hours)
- AR-Assembly (2 hours)
- AR-Memory (10 hours)
- AR-IO (2 hours)
- SF-Performance (4 hours)
- AR-Heterogeneity (4 hours)
- AR-Performance-Energy (5 hours)
- AR-Security (4 hours)
- AR-Quantum (4 hours)

Course objectives: Students should understand how computer architectures evolved into today's heterogeneous systems and to what extent choices made in the past can influence the design of future high-performance computing systems.

Prerequisites:

- MSF-Discrete

Systems Course to include the following:

- SEP-History (2 hours)

- SF-Design (2 hours)
- SF-Reliability (2 hours)
- OS-Purpose (2 hours)
- AR-Representation (2 hours)
- AR-Assembly (2 hours)
- AR-Memory (8 hours)
- AR-IO (2 hours)
- PDC-Algorithms (4 hours)
- AR-Heterogeneity (4 hours)
- AR-Performance-Energy (5 hours)
- NC-Applications (5 hours)

Course objectives: Students should understand the advanced architectural aspects of modern computer systems, including heterogeneous architectures and the required hardware and software interfaces to improve the performance and energy footprint of applications.
Prerequisites:

- MSF-Discrete, MSF-Statistics

Committee

Chair: Marcelo Pias, Federal University of Rio Grande (FURG), Rio Grande-RS, Brazil
Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Mohamed Zahran, New York University, New York, NY, USA
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Qiao Xiang, Xiamen University, Xiamen, China
- Adrian German, Indiana University, Bloomington, IN, USA

Data Management (DM)

Preamble

Since the mid-1970s, the study of Data Management (DM) has meant an almost exclusive study of relational database systems. Depending on institutional context, students have studied, in varying proportions, the following.

- Data modeling and database design: for example, E-R Data model, relational model, normalization theory
- Query construction: e.g., relational algebra, SQL
- Query processing: e.g., indices (B+tree, hash), algorithms (e.g., external sorting, select, project, join), query optimization (transformations, index selection)
- DBMS internals: e.g., concurrency/locking, transaction management, buffer management

Today's graduates are expected to possess DBMS user (rather than implementor) skills. These primarily include data modeling and query construction; ability to take an unorganized collection of data, organize it using a DBMS, and access/update the collection via queries.

Additionally, students need to study the following.

- The role data plays in an organization. This includes the Data Life Cycle: Creation-Processing-Review/Reporting-Retention/Retrieval-Destruction.
- The social/legal aspects of data collection: e.g., scale, data privacy, database privacy (compliance) by design, de-identification, ownership, reliability, database security, and intended and unintended applications.
- Emerging and advanced technologies that are augmenting/replacing traditional relational systems, particularly those used to support (big) data analytics, including NoSQL (e.g., JSON, XML, key-value store databases), cloud databases, MapReduce, and dataframes.
- The existing and emerging roles for those involved with data management, which include the following.
    - Product feature engineers: those who use both SQL and NoSQL operational databases.
    - Analytical engineers/data engineers: those who write analytical SQL, Python, and Scala code to build data assets for business groups.
    - Business analysts: those who build/manage data most frequently with Excel spreadsheets.
    - Data infrastructure engineers: those who implement a data management system in a variety of data applications (e.g., OLTP).
    - “Everyone” who produces or consumes data must understand the associated social, ethical, and professional issues.

One role that transcends all the above categories is that of data custodian. Previously, data were seen as a resource to be managed (Information Systems Management) just like other enterprise resources. Today, data are seen in a larger context. Data about customers can now be seen as belonging to (or in some national contexts, as owned by) those customers. There is now an accepted understanding that the safe and ethical storage, and use, of institutional data is part of being a responsible data custodian.

Furthermore, we acknowledge the tension between a curricular focus on professional preparation versus the study of a knowledge area as a scientific endeavor. This is particularly true with Data Management. For example, proving (or at least knowing) the completeness of Armstrong's Axioms is fundamental in functional dependency theory. However, most computer science graduates will never utilize this concept during their professional careers. The same can be said for many other topics in the Data Management canon. Conversely, if our graduates can only normalize data into Boyce-Codd normal form (using an automated tool) and write SQL queries, without understanding the role that indices play in efficient query execution, we have done them and society a disservice.

To this end, the number of CS Core hours is relatively small relative to the KA Core hours. This approach is designed to allow institutions with differing contexts to customize their curricula appropriately. An institution that focuses on OLTP implementation, for example, would prioritize efficient storage and data access, while an institution that focuses on product features would prioritize programmatic access to extant databases.

However, an institution manages this tension, we wish to give voice to one of the ironies of computer science curricula. Students typically spend much of their educational life reading (and writing) data from a file or interactively, while outside of the academy the predominant data comes from databases accessed programmatically. Perhaps in the not-too-distant future students will learn programmatic database access early on and then continue this practice as they progress through their curriculum.

Finally, we understand that while the Data Management KA may be orthogonal to the SEC (Security) and SEP (Society, Ethics, and the Profession) KAs, it is also ground zero for these (and other) knowledge areas. When designing persistent data stores, the question of what should be stored must be examined from both legal and ethical perspectives. Are there privacy concerns? And just as importantly, how well protected is the data?

Changes since CS2013

- Rename the knowledge area from Information Management to Data Management. This renaming does not represent any kind of philosophical shift. It is simply an effort to avoid confusion with the similar definitions used in Information Systems and Information Technology curricula.
- Inclusion of NoSQL approaches and MapReduce as CS Core topics.
- Increased attention to SEP and SEC topics in both the CS Core and KA Core areas.

Core Hours

Knowledge Unit	CS Core Hours	KA Core Hours
The Role of Data	2	
Core Database Systems Concepts	2	1

Data Modeling	2	3
Relational Databases	1	3
Query Construction	2	4
Query Processing		4
DBMS Internals		4
NoSQL Systems		2
Data Security & Privacy	1	2
Data Analytics		3
Distributed Databases/Cloud Computing		
Semi-structured and Unstructured Databases		
Society, Ethics, and the Profession	Included in SEP hours	
Total	**10**	**26**

The CS Core hour in Data Security & Privacy is shared with SEC and is counted here.

Knowledge Units

DM-Data: The Role of Data and the Data Life Cycle

CS Core:

1. The Data Life Cycle: Creation-Processing-Review/Reporting-Retention/Retrieval-Destruction (See also: SEP-Context, SEP-Ethical-Analysis, SEP-Professional-Ethics, SEP-Privacy, SEP-Security, SEC-Foundations)

Illustrative Learning Outcomes:

CS Core:

1. Identify the five stages of the Data Life Cycle.

DM-Core: Core Database System Concepts

CS Core:

1. Purpose and advantages of database systems
2. Components of database systems

3. Design of core DBMS functions (e.g., query mechanisms, transaction management, buffer management, access methods)
4. Database architecture, data independence, and data abstraction
5. Transaction management
6. Normalization
7. Approaches for managing large volumes of data (e.g., NoSQL database systems, use of MapReduce) (See also: PDC-Algorithms)
8. How to support CRUD-only applications
9. Distributed databases/cloud-based systems
10. Structured, semi-structured, and unstructured data
11. Use of a declarative query language

KA Core:

12. Systems supporting structured and/or stream content

Illustrative Learning Outcomes:

CS Core:

1. Identify at least four advantages that using a database system provides.
2. Enumerate the components of a (relational) database system.
3. Follow a query as it is processed by the components of a (relational) database system.
4. Defend the value of data independence.
5. Compose a simple select-project-join query in SQL.
6. Enumerate the four properties of a correct transaction manager.
7. Describe the advantages for eliminating duplicate repeated data.
8. Outline how MapReduce uses parallelism to process data efficiently.
9. Evaluate the differences between structured and semi/unstructured databases.

DM-Modeling: Data Modeling

CS Core:

1. Data modeling (See also: SE-Requirements)
2. Relational data model (See also: MSF-Discrete)

KA Core:

3. Conceptual models (e.g., entity-relationship, UML diagrams)
4. Semi-structured data models (expressed using DTD, XML, or JSON Schema, for example)

Non-core:

5. Spreadsheet models
6. Object-oriented models (See also: FPL-OOP)
    a. GraphQL
7. New features in SQL
8. Specialized Data Modeling topics
    a. Time series data (aggregation, join)
    b. Graph data (link traversal)

c. Techniques for avoiding inefficient raw data access (e.g., “avg daily price”): materialized views and special data structures (e.g., Hyperloglog, bitmap)
d. Geo-Spatial data (e.g., GIS databases) (See also: SPD-Interactive)

Illustrative Learning Outcomes:
CS Core:
1. Describe the components of the relational data model.
2. Model 1:1, 1:n, and n:m relationships using the relational data model.

KA Core:
3. Describe the components of the E-R (or some other non-relational) data model.
4. Model a given environment using a conceptual data model.
5. Model a given environment using the document-based or key-value store-based data model.

DM-Relational: Relational Databases

CS Core:
1. Entity and referential integrity: Candidate key, superkeys
2. Relational database design

KA Core:
3. Mapping conceptual schema to a relational schema
4. Physical database design: file and storage structures (See also: OS-Files)
5. Introduction to Functional dependency theory
6. Normalization Theory
    a. Decomposition of a schema; lossless-join, and dependency-preservation properties of a decomposition
    b. Normal forms (BCNF)
    c. Denormalization (for efficiency)

Non-core:
7. Functional dependency theory
    a. Closure of a set of attributes
    b. Canonical Cover
8. Normalization theory
    a. Multi-valued dependency (4NF)
    b. Join dependency (PJNF, 5NF)
    c. Representation theory

Illustrative Learning Outcomes:
CS Core:
1. Describe the defining characteristics behind the relational data model.
2. Comment on the difference between a foreign key and a superkey.
3. Enumerate the different types of integrity constraints.

KA Core:

4. Compose a relational schema from a conceptual schema which contains 1:1, 1:n, and n:m relationships.
5. Map appropriate file structure to relations and indices.
6. Describe how functional dependency theory generalizes the notion of key.
7. Defend a given decomposition as lossless and or dependency preserving.
8. Detect which normal form a given decomposition yields.
9. Comment on reasons for denormalizing a relation.

DM-Querying: Query Construction

CS Core:

1. SQL Query Formation
    a. Interactive SQL execution
    b. Programmatic execution of an SQL query

KA Core:

2. Relational Algebra
3. SQL
    a. Data definition including integrity and other constraint specifications
    b. Update sublanguage

Non-core:

4. Relational Calculus
5. QBE and 4th-generation environments
6. Different ways to invoke non-procedural queries in conventional languages
7. Introduction to other major query languages (e.g., XPATH, SPARQL)
8. Stored procedures

Illustrative Learning Outcomes:

CS Core:

1. Compose SQL queries that incorporate select, project, join, union, intersection, set difference, and set division.
2. Determine when a nested SQL query is correlated or not.
3. Iterate over data retrieved programmatically from a database via an SQL query.

KA Core:

4. Define, in SQL, a relation schema, including all integrity constraints and delete/update triggers.
5. Compose an SQL query to update a tuple in a relation.

DM-Processing: Query Processing

KA Core:

1. Page structures
2. Index structures
    a. B+ trees (See also: AL-Foundational)

    b. Hash indices: static and dynamic (See also: AL-Foundational, SEC-Foundations)
    c. Index creation in SQL
3. File structures (See also: OS-Files)
    a. Heap files
    b. Hash files
4. Algorithms for query operators
    a. External Sorting (See also: AL-Foundational)
    b. Selection
    c. Projection; with and without duplicate elimination
    d. Natural Joins: Nested loop, Sort-merge, Hash join
    e. Analysis of algorithm efficiency (See also: AL-Complexity)
5. Query transformations
6. Query optimization
    a. Access paths
    b. Query plan construction
    c. Selectivity estimation
    d. Index-only plans
7. Parallel Query Processing (e.g., parallel scan, parallel join, parallel aggregation) (See also: PDC-Algorithms)
8. Database tuning/performance
    a. Index selection
    b. Impact of indices on query performance (See also: SF-Performance, SEP-Sustainability)
    c. Denormalization

Illustrative Learning Outcomes:

KA Core:

1. Describe the purpose and organization of both B+ tree and hash index structures.
2. Compose an SQL command to create an index (any kind).
3. Specify the steps for the various query operator algorithms: external sorting, projection with duplicate elimination, sort-merge join, hash-join, block nested-loop join.
4. Derive the run-time (in I/O requests) for each of the above algorithms.
5. Transform a query in relational algebra to its equivalent appropriate for a left-deep, pipelined execution.
6. Compute selectivity estimates for a given selection and/or join operation.
7. Describe how to modify an index structure to facilitate an index-only operation for a given relation.
8. For a given scenario decide on which indices to support for the efficient execution of a set of queries.
9. Describe how DBMSs leverage parallelism to speed up query processing by dividing the work across multiple processors or nodes.

DM-Internals: DBMS Internals

KA Core:

1. DB Buffer Management (See also: OS-Memory, SF-Resource)

2. Transaction Management (See also: PDC-Coordination)
    a. Isolation Levels
    b. ACID
    c. Serializability
    d. Distributed Transactions
3. Concurrency Control: (See also: OS-Concurrency)
    a. 2-Phase Locking
    b. Deadlocks handling strategies
    c. Quorum-based consistency models
4. Recovery Manager
    a. Relation with Buffer Manager

Non-core:

5. Concurrency Control:
    a. Optimistic concurrency control
    b. Timestamp concurrency control
6. Recovery Manager
    a. Write-Ahead logging
    b. ARIES recovery system (Analysis, REDO, UNDO)

Illustrative Learning Outcomes:

KA Core:

1. Describe how a DBMS manages its Buffer Pool.
2. Describe the four properties for a correct transaction manager.
3. Outline the principle of serializability.

DM-NoSQL: NoSQL Systems

KA Core:

1. Why NoSQL? (e.g., Impedance mismatch between Application [CRUD] and RDBMS)
2. Key-Value and Document data model

Non-core:

3. Storage systems (e.g., Key-Value systems, Data Lakes)
4. Distribution Models (Sharding and Replication) (See also: PDC-Communication)
5. Graph Databases
6. Consistency Models (Update and Read, Quorum consistency, CAP theorem) (See also: PDC-Communication)
7. Processing model (e.g., Map-Reduce, multi-stage map-reduce, incremental map-reduce) (See also: PDC-Communication)
8. Case Studies: Cloud storage system (e.g., S3); Graph databases; “When not to use NoSQL” (See also: SPD-Web)

Illustrative Learning Outcomes:

KA Core:

1. Develop a use case for the use of NoSQL over RDBMS.
2. Describe the defining characteristics behind Key-Value and Document-based data models.

DM-Security: Data Security and Privacy

CS Core:

1. Differences between data security and data privacy (See also: SEC-Foundations)
2. Protecting data and database systems from attacks, including injection attacks such as SQL injection (See also: SEC-Foundations)
3. Personally identifying information (PII) and its protection (See also: SEC-Foundations, SEP-Security, SEP-Privacy)
4. Ethical considerations in ensuring the security and privacy of data (See also: SEC-SEP, SEP-Ethical-Analysis, SEP-Security, SEP-Privacy)

KA Core:

5. Need for, and different approaches to securing data at rest, in transit, and during processing (See also: SEC-Foundations, SEC-Crypto)
6. Database auditing and its role in digital forensics (See also: SEC-Forensics)
7. Data inferencing and preventing attacks (See also: SEC-Crypto)
8. Laws and regulations governing data security and data privacy (See also: SEP-Security, SEP-Privacy, SEC-Foundations, SEC-Governance)

Non-core:

9. Typical risk factors and prevention measures for ensuring data integrity (See also: SEC-Governance)
10. Ransomware and prevention of data loss and destruction (See also: SEC-Coding, SEC-Forensics)

Illustrative Learning Outcomes:

CS Core:

1. Describe the differences in the goals for data security and data privacy.
2. Identify and mitigate risks associated with different approaches to protecting data.
3. Describe legal and ethical considerations of end-to-end data security and privacy.

KA Core:

4. Develop a database auditing system given risk considerations.
5. Apply several data exploration approaches to understanding unfamiliar datasets.

DM-Analytics: Data Analytics

KA Core:

1. Exploratory data techniques (motivation, representation, descriptive statistics, visualizations)
2. Data science lifecycle: business understanding, data understanding, data preparation, modeling, evaluation, deployment, and user acceptance (See also: AI-ML)

3. Data mining and machine learning algorithms: e.g., classification, clustering, association, regression (See also: AI-ML)
4. Data acquisition and governance (See also: SEC-Governance)
5. Data security and privacy considerations (See also: SEP-Security, SEP-Privacy, SEC-Foundations)
6. Data fairness and bias (See also: SEP-Security, AI-SEP)
7. Data visualization techniques and their use in data analytics (See also: GIT-Visualization)
8. Entity Resolution

Illustrative Learning Outcomes:

KA Core:

1. Describe several data exploration approaches, including visualization, to understanding unfamiliar datasets.
2. Apply several data exploration approaches to understanding unfamiliar datasets.
3. Describe basic machine learning/data mining algorithms and when they are appropriate for use.
4. Apply several machine learning/data mining algorithms.
5. Describe legal and ethical considerations in acquiring, using, and modifying datasets.
6. Describe issues of fairness and bias in data collection and usage.

DM-Distributed: Distributed Databases/Cloud Computing

Non-core:

1. Distributed DBMS (See also: PDC-Communications)
    a. Distributed data storage
    b. Distributed query processing
    c. Distributed transaction model
    d. Homogeneous and heterogeneous solutions
    e. Client-server distributed databases (See also: NC-Fundamentals)
2. Parallel DBMS (See also: PDC-Algorithms)
    a. Parallel DBMS architectures: shared memory, shared disk, shared nothing;
    b. Speedup and scale-up, e.g., use of the MapReduce processing model (See also: PDC-Programs, SF-Foundations)
    c. Data replication and weak consistency models (See also: PDC-Coordination)

DM-Unstructured: Semi-structured and Unstructured Databases

Non-core:

1. Vectorized unstructured data (text, video, audio, etc.) and vector storage
    a. TF-IDF Vectorizer with ngram
    b. Word2Vec
    c. Array database or array data type handling
2. Semi-structured databases (e.g., JSON)
    a. Storage
        i. Encoding and compression of nested data types
    b. Indexing
        i. Btree, skip index, Bloom filter

ii. Inverted index and bitmap compression
iii. Space filling curve indexing for semi-structured geo-data

c. Query processing for OLTP and OLAP use cases
i. Insert, Select, update/delete tradeoffs
ii. Case studies on Postgres/JSON, MongoDB, and Snowflake/JSON

DM-SEP: Society, Ethics, and the Profession

CS Core:

1. Issues related to scale (See also: SEP-Economies)
2. Data privacy overall (See also: SEP-Privacy, SEP-Ethical-Analysis)
    a. Privacy compliance by design (See also: SEP-Privacy)
3. Data anonymity (See also: SEP-Privacy)
4. Data ownership/custodianship (See also: SEP-Professional-Ethics)
5. Intended and unintended applications of stored data (See also: SEP-Professional-Ethics, SEC-Foundations)

KA Core:

6. Reliability of data (See also: SEP-Security)
7. Provenance, data lineage, and metadata management (See also: SEP-Professional-Ethics)
8. Data security (See also: DM-Security, SEP-Security)

Illustrative Learning Outcomes:

CS Core:

1. Enumerate three social and three legal issues related to large data collections.
2. Describe the value of data privacy.
3. Identify the competing stakeholders with respect to data ownership.
4. Enumerate three negative unintended consequences from a given (well known) data-centric application (e.g., Facebook, LastPass, Ashley Madison).

KA Core:

5. Describe the meaning of data provenance and lineage.
6. Identify how a database might contribute to data security as well as how it may introduce insecurities.

Professional Dispositions

- **Meticulous:** Those who either access or store data collections must be meticulous in fulfilling data ownership responsibilities.
- **Responsible**: In conjunction with the professional management of (personal) data, it is equally important that data are managed responsibly. Protection from unauthorized access as well as prevention of irresponsible, though legal, use of data is paramount. Furthermore, data custodians need to protect data not only from outside attack, but from crashes and other foreseeable dangers.

- **Collaborative**: Data managers and data users must behave in a collaborative fashion to ensure that the correct data are accessed and are used only in an appropriate manner.
- **Responsive**: The data that get stored and are accessed are always in response to an institutional need/request.

Mathematics Requirements

Required:

- Discrete Mathematics: Set theory (union, intersection, difference, cross-product) (See also: MSF-Discrete)

Desired:

- Probability and Statistics for those studying DM-Analytics. (See also: MSF-Probability, MSF-Statistics)

Desirable Data Structures:

- Hash functions and tables (See also: AL-Foundational)
- Balanced (binary) trees (e.g., AVL, 2-3-4, Red-Black) (See also: AL-Foundational)
- B and B+-trees

Course Packaging Suggestions

For those implementing a single course on Database Systems, there are a variety of options. As described in [1], there are four primary perspectives from which to approach databases:

- Database design/modeling
- Database use
- Database administration
- Database development, which includes implementation algorithms

Course design proceeds by focusing on topics from each perspective in varying degrees according to one's institutional context. For example, in [1], one of the courses described can be characterized as design/modeling (20%), use (20%), development/internals (30%), and administration/tuning/advanced topics (30%). The topics might include the following.

- DM-SEP (3 hours)
- DM-Data (1 hour)
- DM-Core (3 hours)
- DM-Modeling (5 hours)
- DM-Relational (4 hours)
- DM-Querying (6 hours)
- DM-Processing (5 hours)
- DM-Internals (5 hours)
- DM-NoSQL (4 hours)
- DM-Security (3 hours)

- DM-Distributed (2 hours)

The more interesting question may be how to cover the CS Core concepts in the absence of a dedicated database course. The key to accomplishing this may be to *normalize* database access. Starting with the introductory course, students could access a database instead of using file I/O or interactive data entry to acquire the data needed for introductory-level programming. As students progress through their curriculum, additional CS Core topics could be introduced. For example, introductory students could be given the code to access the database along with the SQL query. At the intermediate level, they could be writing their own queries. Finally, in a Software Engineering or capstone course, they could practice database design. One advantage of this approach, *databases across the curriculum,* is that it allows for the inclusion of database-related SEP topics to also be spread across the curriculum.

In a similar vein one might have a whole course on the Role of Data from either a Security (SEC) perspective, or an Ethics (SEP) perspective.

Committee

Chair: Mikey Goldweber, Denison University, Granville, OH, USA

Members:

- Sherif Aly, The American University in Cairo, Cairo, Egypt
- Sara More, Johns Hopkins University, Baltimore, MD, USA
- Mohamed Mokbel, University of Minnesota, Minneapolis, MN, USA
- Rajendra K. Raj, Rochester Institute of Technology, Rochester, NY, USA
- Avi Silberschatz, Yale University, New Haven, CT, USA
- Min Wei, Microsoft, Seattle, WA, USA
- Qiao Xiang, Xiamen University, Xiamen, China

References

1. The 2022 Undergraduate Database Course in Computer Science: What to Teach?. Michael Goldweber, Min Wei, Sherif Aly, Rajendra K. Raj, and Mohamed Mokbel. *ACM Inroads*, 13, 3, 2022.

Foundations of Programming Languages (FPL)

Preamble

The foundations of programming languages are rooted in discrete mathematics, logic, and formal languages, and provide a basis for the understanding of complex modern programming languages. Although programming languages vary according to the language paradigm and the problem domain and evolve in response to both societal needs and technological advancement, they share an underlying abstract model of computation and program development. This remains true even as processor hardware and their interface with programming tools become increasingly intertwined and progressively more complex. An understanding of the common abstractions and programming paradigms enables faster learning of programming languages.

The Foundations of Programming Languages knowledge area is concerned with articulating the underlying concepts and principles of programming languages, the formal specification of a programming language and the behavior of a program, explaining how programming languages are implemented, comparing the strengths and weaknesses of various programming paradigms, and describing how programming languages interface with entities such as operating systems and hardware. The concepts covered here are applicable to several languages and an understanding of these principles assists a learner to move readily from one language to another, as well as select a programming paradigm and language that best suits the problem at hand.

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. Over the course of a career, a computer scientist will learn and work with many different languages, separately or together. Software developers must understand different programming models, programming features and constructs, and underlying concepts to make informed design choices among languages that support multiple complementary approaches. It would be useful to know how programming language features are defined, composed, and implemented to improve execution efficiency and long-term maintenance of developed software. Also useful is a basic knowledge of language translation, program analysis, run-time behavior, memory management and interplay of concurrent processes communicating with each other through message-passing, shared memory, and synchronization. Finally, some developers and researchers will need to design new languages, an exercise which requires greater familiarity with basic principles.

Changes since CS2013

Changes since 2013 include a change in name of the KA from Programming Languages to Foundations of Programming Languages to reflect the fact that the KA is about the fundamentals underpinning programming languages, and related concepts, not about any specific programming language. Changes also include a redistribution of content formerly identified as core Tier-1 and core Tier-2 within the Programming Language Knowledge Area (KA). In CS2013, graduates were expected to complete all Tier-1 topics and 80% of Tier-2 topics, for a total of 24 required hours. These 24 hours are designated as CS Core topics in CS2023. The remaining Tier-2 topics are designated as KA Core topics in CS2023. The change in core topics (Tier-1 plus 80% of Tier-2 hours) from 2013 reflects the

change in importance or relevance of topics over the past decade. The inclusion of new topics was driven by their current prominence in the programming language landscape, or the anticipated impact of emerging areas on the profession in general. Specifically, the changes are:

- Object-Oriented Programming -4 CS Core hours
- Functional Programming -2 CS Core hours
- Event-Driven and Reactive Programming +1 CS Core hour
- Parallel and Distributed Computing +3 CS Core hours
- Type Systems -1 CS Core hour
- Program Representation -1 CS Core hour

In addition, a number of knowledge units from CS2013 were renamed to reflect their content more accurately, as noted here.

- Static Analysis was renamed Program Analysis and Analyzers.
- Concurrency and Parallelism was renamed Parallel and Distributed Computing.
- Program Representation was renamed Program Abstraction and Representation.
- Runtime Systems was renamed Runtime Behavior and Systems.
- Basic Type Systems and Type Systems were merged into a single topic and named Type Systems.

Six new knowledge units were added to reflect their continuing and growing importance as we look toward the 2030s:

- Shell Scripting +2 CS Core hours
- Systems Execution and Memory Model +3 CS Core hours
- Formal Development Methodologies
- Design Principles of Programming Languages
- Fundamentals of Programming Languages
- Society, Ethics, and the Profession

Notes:

- Several topics within this knowledge area either build on or overlap content covered in other knowledge areas such as the Software Development Fundamentals knowledge area in a curriculum's introductory courses. Curricula will differ on which topics are integrated in this fashion and which are postponed until later courses on software development and programming languages.
- Different programming paradigms correspond to different problem domains. Most languages have evolved to integrate more than one programming paradigm such as imperative with object-oriented, functional programming with object-oriented, logic programming with object-oriented, and event and reactive modeling with object-oriented programming.
  Hence, the emphasis is not on just one programming paradigm but on a balance of all major programming paradigms.
- While the number of CS Core and KA Core hours is identified for each major programming paradigm (object-oriented, functional, logic), the distribution of hours across the paradigms may differ depending on the curriculum and programming languages students have been exposed to

leading up to coverage of this knowledge area. This document assumes that students have exposure to an object-oriented programming language leading into this knowledge area.

- Imperative programming is not listed as a separate paradigm to be examined. Instead, it is treated as a subset of the object-oriented paradigm.
- With multicore computing, cloud computing, and computer networking becoming commonly available in the market, it has become critical to understand the integration of "distribution, concurrency, parallelism" along with other programming paradigms as a core area. This paradigm is integrated with almost all other major programming paradigms.
- With ubiquitous computing and real-time temporal computing applications increasing in daily human life within domains such as health, transportation, smart homes, it has become important to cover the software development aspects of event-driven and reactive programming as well as parallel and distributed computing. A number of topics covered will require and overlap with concepts in knowledge areas such as Architecture and Organization, Operating Systems, and Systems Fundamentals.
- Some topics from the Parallel and Distributed Computing knowledge unit are likely to be integrated within the curriculum with topics from the Parallel and Distributed Programming knowledge area.
- There is an increasing interest in formal methods to prove program correctness and other properties. To support this, additional coverage of topics related to formal methods has been included, but all these topics are identified as Non-core.
- When introducing these topics, it is also important that an instructor provides context for this material including why we have an interest in programming languages and what they do for us in terms of providing a human readable version of instructions for a computer to execute.

Core Hours

Knowledge Unit	CS Core	KA Core
Object-Oriented Programming	4 + 1 (SDF)	1
Functional Programming	4	3
Logic Programming		2 + 1 (MSF)
Shell Scripting	2	
Event-Driven and Reactive Programming	2	2
Parallel and Distributed Computing	2 + 1 (PDC)	2
Aspect-Oriented Programming		
Type Systems	3	3
Systems Execution and Memory Model	2 + 1 (AR and OS)	

Language Translation and Execution	2	3
Program Abstraction and Representation		3
Syntax Analysis		
Compiler Semantic Analysis		
Program Analysis and Analyzers		
Code Generation		
Runtime Behavior and Systems		
Advanced Programming Constructs		
Language Pragmatics		
Formal Semantics		
Formal Development Methodologies		
Design Principles of Programming Languages		
Society, Ethics, and the Profession	Included in SEP hours	
Total	**21**	**19**

The CS and KA Core totals do not include the shared hours that have been counted in other knowledge areas.

Knowledge Units

FPL-OOP: Object-Oriented Programming

CS Core:

1. Imperative programming as a subset of object-oriented programming.
2. Object-oriented design:
   a. Decomposition into objects carrying state and having behavior.
   b. Class-hierarchy design for modeling.
3. Definition of classes: fields, methods, and constructors. (See also: SDF-Fundamentals)
4. Subclasses, inheritance (including multiple inheritance), and method overriding.
5. Dynamic dispatch: definition of method-call.
6. Exception handling. (See also: SDF-Fundamentals, PDC-Coordination, SE-Construction)
7. Object-oriented idioms for encapsulation:
   a. Privacy, data hiding, and visibility of class members.
   b. Interfaces revealing only method signatures.
   c. Abstract base classes, traits and mixins.

8. Dynamic vs static properties.
9. Composition vs inheritance.
10. Subtyping:
    a. Subtype polymorphism; implicit upcasts in typed languages.
    b. Notion of behavioral replacement: subtypes acting like supertype.
    c. Relationship between subtyping and inheritance.

KA Core:

11. Collection classes, iterators, and other common library components.
12. Metaprogramming and reflection.

Illustrative Learning Outcomes:

CS Core:

1. Enumerate the differences between imperative and object-oriented programming paradigms.
2. Compose a class through design, implementation, and testing to meet behavioral requirements.
3. Build a simple class hierarchy utilizing subclassing that allows code to be reused for distinct subclasses.
4. Predict and validate control flow in a program using dynamic dispatch.
5. Compare and contrast how computational solutions to a problem differ in procedural, functional, and object-oriented approaches.
6. Compare and contrast mechanisms to define and protect data elements within procedural, functional, and object-oriented approaches.
7. Compare and contrast the benefits and costs/impact of using inheritance (subclasses) and composition (specifically, how to base composition on higher order functions).
8. Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype).
9. Use object-oriented encapsulation mechanisms such as interfaces and private members.
10. Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. (See also: FPL-Functional)

KA Core:

11. Use collection classes and iterators effectively to solve a problem.

FPL-Functional: Functional Programming

CS Core:

1. Lambda expressions and evaluation: (See also: AL-Models, FPL-Formalism)
    a. Variable binding and scope rules. (See also: SDF-Fundamentals)
    b. Parameter-passing. (See also: SDF-Fundamentals)
    c. Nested lambda expressions and reduction order.
2. Effect-free programming:
    a. Function calls have no side effects, facilitating compositional reasoning.
    b. Immutable variables and data copying vs reduction.
    c. Use of recursion vs loops vs pipelining (map/reduce).

3. Processing structured data (e.g., trees) via functions with cases for each data variant:
   a. Functions defined over compound data in terms of functions applied to the constituent pieces.
   b. Persistent data structures.
4. Using higher-order functions (taking, returning, and storing functions).

KA Core:

5. Metaprogramming and reflection.
6. Function closures (functions using variables in the enclosing lexical environment).
   a. Basic meaning and definition – creating closures at run-time by capturing the environment.
   b. Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments.
   c. Using a closure to encapsulate data in its environment.
   d. Delayed versus eager evaluation.

Non-core:

7. Graph reduction machine and call-by-need.
8. Implementing delayed evaluation.
9. Integration with logic programming paradigm using concepts such as equational logic, narrowing, residuation and semantic unification. (See also: FPL-Logic)
10. Integration with other programming paradigms such as imperative and object-oriented.

Illustrative learning outcomes:

CS Core:

1. Develop basic algorithms that avoid assigning to mutable states or considering reference equality.
2. Develop useful functions that take and return other functions.
3. Compare and contrast how computational solutions to a problem differ in procedural, functional, and object-oriented approaches.
4. Compare and contrast mechanisms to define and protect data elements within procedural, functional, and object-oriented approaches.

KA Core:

5. Explain a simple example of lambda expression being implemented using a virtual machine, such as a SECD machine, showing storage and reclaim of the environment.
6. Correctly interpret variables and lexical scope in a program using function closures.
7. Use functional encapsulation mechanisms such as closures and modular interfaces.
8. Compare and contrast stateful vs stateless execution.
9. Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. (See also: FPL-OOP)

Non-core:

10. Illustrate graph reduction using a λ-expression using a shared subexpression.
11. Illustrate the execution of a simple nested λ-expression using an abstract machine, such as an ABC machine.
12. Illustrate narrowing, residuation, and semantic unification using simple illustrative examples.

13. Illustrate the concurrency constructs using simple programming examples of known concepts such as a buffer being read and written concurrently or sequentially. (See also: FPL-OOP)

FPL-Logic: Logic Programming

KA Core:

1. Universal vs existential quantifiers. (See also: AI-LRR, MSF-Discrete)
2. First order predicate logic vs higher order logic. (See also: AI-LRR, MSF-Discrete)
3. Expressing complex relations using logical connectives and simpler relations.
4. Definitions of Horn clause, facts, goals and subgoals.
5. Unification and unification algorithm; unification vs assertion vs expression evaluation.
6. Mixing relations with functions. (See also: MSF-Discrete)
7. Cuts, backtracking, and non-determinism.
8. Closed-world vs open-world assumptions.

Non-core:

9. Memory overhead of variable copying in handling iterative programs.
10. Programming constructs to store partial computation and pruning search trees.
11. Mixing functional programming and logic programming using concepts such as equational logic, narrowing, residuation, and semantic unification. (See also: FPL-Functional)
12. Higher-order, constraint, and inductive logic programming. (See also: AI-LRR)
13. Integration with other programming paradigms such as object-oriented programming.
14. Advance programming constructs such as difference-lists, creating user defined data structures, set of, etc.

Illustrative learning outcomes:

KA Core:

1. Use a logic language to implement a conventional algorithm.
2. Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts.
3. Use a simple illustrative example to show correspondence between First Order Predicate Logic (FOPL) and logic programs using Horn clauses.
4. Use examples to illustrate the unification algorithm and its role of parameter-passing in query reduction.
5. Use simple logic programs interleaving relations, functions, and recursive programming such as factorial and Fibonacci numbers and simple complex relationships between entities and illustrate execution and parameter-passing using unification and backtracking.

Non-core:

6. Illustrate computation of simple programs such as Fibonacci and show overhead of recomputation, and then show how to improve execution overhead.

FPL-Scripting: Shell Scripting

CS Core:

1. Error/exception handling
2. Piping (See also: AR-Organization, SF-Overview, OS-Process)
3. System commands (See also: SF-Overview)
    a. Interface with operating systems (See also: SF-Overview, OS-Principles)
4. Environment variables (See also: SF-Overview)
5. File abstraction and operators (See also: SDF-Fundamentals, OS-Files, SF-Resource)
6. Data structures, such as arrays and lists. (See also: AL-Foundational, SDF-Fundamentals, SDF-Data-Structures)
7. Regular expressions (See also: AL-Models)
8. Programs and processes (See also: OS-Process)
9. Workflow

Illustrative learning outcomes:

CS Core:

1. Create and execute automated scripts to manage various system tasks.
2. Solve various text processing problems through scripting.

FPL-Event-Driven: Event-Driven and Reactive Programming

CS Core:

1. Procedural programming vs reactive programming: advantages of reactive programming in capturing events.
2. Components of reactive programming: event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers. (See also: GIT-Interaction, SPD-Web, SPD-Mobile, SPD-Robot, SPD-Embedded, SPD-Game, SPD-Interactive)
3. Stateless and state-transition models of event-based programming.
4. Canonical uses such as GUIs, mobile devices, robots, servers. (See also: GIT-Interaction, GIT-Image, SPD-Web, SPD-Mobile, SPD-Robot, SPD-Embedded, SPD-Game, SPD-Interactive)

KA Core:

5. Using a reactive framework:
    a. Defining event handlers/listeners
    b. Parameterization of event senders and event arguments
    c. Externally generated events and program-generated events
6. Separation of model, view, and controller
7. Event-driven and reactive programs as state-transition systems

Illustrative learning outcomes:

CS Core:

1. Implement event handlers for use in reactive systems, such as GUIs.
2. Examine why an event-driven programming style is natural in domains where programs react to external events.

KA Core:

3. Define and use a reactive framework.

4. Describe an interactive system in terms of a model, a view, and a controller.

FPL-Parallel: Parallel and Distributed Computing

CS Core:

1. Safety and liveness (See also: PDC-Evaluation)
   a. Race conditions (See also: OS-Concurrency)
   b. Dependencies/preconditions
   c. Fault models (See also: OS-Faults)
   d. Termination (See also: PDC-Coordination)
2. Programming models (See also: PDC-Programs)
   One or more of the following:
   a. Actor models
   b. Procedural and reactive models
   c. Synchronous/asynchronous programming models
   d. Data parallelism
3. Properties (See also: PDC-Programs, PDC-Coordination)
   a. Order-based properties
      i. Commutativity
      ii. Independence
   b. Consistency-based properties
      i. Atomicity
      ii. Consensus
4. Execution control: (See also: PDC-Coordination, SF-Foundations)
   a. Async await
   b. Promises
   c. Threads
5. Communication and coordination (See also: OS-Process, PDC-Communication, PDC-Coordination)
   a. Mutexes
   b. Message-passing
   c. Shared memory
   d. Cobegin-coend
   e. Monitors
   f. Channels
   g. Threads
   h. Guards

KA Core:

6. Futures
7. Language support for data parallelism such as forall, loop unrolling, map/reduce
8. Effect of memory-consistency models on language semantics and correct code generation
9. Representational State Transfer Application Programming Interfaces (REST APIs)
10. Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing
11. Overheads of message-passing

12. Granularity of program for efficient exploitation of concurrency
13. Concurrency and other programming paradigms (e.g., functional)

Illustrative learning outcomes:

CS Core:

1. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result.
2. Implement correct concurrent programs using multiple programming models, such as shared memory, actors, futures, synchronization constructs, and data-parallelism primitives.
3. Use a message-passing model to analyze a communication protocol.
4. Use synchronization constructions such as monitor/synchronized methods in a simple program.
5. Modeling data dependency using simple programming constructs involving variables, read and write.
6. Modeling control dependency using simple constructs such as selection and iteration.

KA Core:

7. Explain how REST API's integrate applications and automate processes.
8. Explain benefits, constraints and challenges related to distributed and parallel computing.

FPL-Aspect: Aspect-Oriented Programming

Non-core:

1. Aspects
2. Join points
3. Advice
   a. Before
   b. After (as finally, returning or throwing)
   c. Around
4. Point cuts
   a. Designators
5. Weaving – static and dynamic
6. Alternatives including annotations and IDEs

FPL-Types: Type Systems

CS Core:

1. A type as a set of values together with a set of operations
   a. Primitive types (e.g., numbers, Booleans) (See also: SDF-Fundamentals)
   b. Compound types built from other types (e.g., records/structs, unions, arrays, lists, functions, references using set operations) (See also: SDF-Data-Structures)
2. Association of types to variables, arguments, results, and fields
3. Type safety as an aspect of program correctness (See also: FPL-Formalism)
4. Type safety and errors caused by using values inconsistently given their intended types
5. Goals and limitations of static and dynamic typing: detecting and eliminating errors as early as possible.

6. Generic types (parametric polymorphism)
    a. Definition and advantages of polymorphism: parametric, subtyping, overloading, and coercion
    b. Comparison of monomorphic and polymorphic types
    c. Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism
    d. Generic parameters and typing
    e. Use of generic libraries such as collections
    f. Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
    g. Prescriptive vs descriptive polymorphism
    h. Implementation models of polymorphic types
    i. Subtyping

KA Core:

7. Type equivalence: structural vs name equivalence
8. Complementary benefits of static and dynamic typing:
    a. Errors early vs errors late/avoided
    b. Enforce invariants during code development and code maintenance vs postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections.
    c. Typing rules for function, product, and sum types
    d. Avoiding misuse of code vs allowing more code reuse
    e. Detect incomplete programs vs allow incomplete programs to run
    f. Relationship to static analysis
    g. Decidability

Non-core:

9. Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
10. Type checking
11. Subtyping: (See also: FPL-OOP)
    a. Subtype polymorphism; implicit upcasts in typed languages
    b. Notion of behavioral replacement: subtypes acting like supertype
    c. Relationship between subtyping and inheritance
12. Type safety as preservation plus progress
13. Type inference
14. Static overloading
15. Propositions as types (implication as a function, conjunction as a product, disjunction as a sum) (See also: FPL-Formalism)
16. Dependent types (universal quantification as dependent function, existential quantification as dependent product). (See also: FPL-Formalism)

Illustrative learning outcomes:

CS Core:

1. Describe, for both a primitive and a compound type, the values that have that type.

2. Describe, for a language with a static type system, the operations that are forbidden statically, such as passing the wrong type of value to a function or method.
3. Describe examples of program errors detected by a type system.
4. Identify program properties, for multiple programming languages, that are checked statically and program properties that are checked dynamically.
5. Describe an example program that does not type-check in a particular language and yet would have no error if run.
6. Use types and type-error messages to write and debug programs.

KA Core:

7. Explain how typing rules define the set of operations that are legal for a type.
8. List the type rules governing the use of a particular compound type.
9. Explain why undecidability requires type systems to conservatively approximate program behavior.
10. Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections.
11. Discuss the differences among generics, subtyping, and overloading.
12. Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software.

Non-core:

13. Define a type system precisely and compositionally.
14. For various foundational type constructors, identify the values they describe and the invariants they enforce.
15. Precisely describe the invariants preserved by a sound type system.
16. Prove type safety for a simple language in terms of preservation and progress theorems.
17. Implement a unification-based type-inference algorithm for a simple language.
18. Explain how static overloading and associated resolution algorithms influence the dynamic behavior of programs.

FPL-Systems: Systems Execution and Memory Model

CS Core:

1. Data structures for translation, execution, translation, and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string
2. Direct, indirect, and indexed access to memory location
3. Run-time representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects
4. Abstract low-level machine with simple instruction, stack, and heap to explain translation and execution
5. Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap (See also: AR-Memory, OS-Memory)
    a. Translating selection and iterative constructs to control-flow diagrams
    b. Translating control-flow diagrams to low level abstract code
    c. Implementing loops, recursion, and tail calls

   d. Translating function/procedure calls and return from calls, including different parameter-passing mechanisms using an abstract machine
6. Memory management: (See also: AR-Memory, OS-Memory)
   a. Low level allocation and accessing of high-level data structures such as basic data types, n-dimensional array, vector, record, and objects
   b. Return from procedure as automatic deallocation mechanism for local data elements in the stack
   c. Manual memory management: allocating, de-allocating, and reusing heap memory
   d. Automated memory management: garbage collection as an automated technique using the notion of reachability
7. Green computing. (See also: SEP-Sustainability)

Illustrative learning outcomes:

CS Core:

1. Explain how a core language construct, such as data abstractions and control abstractions, is executed.
2. Explain how programming language implementations typically organize memory into global data, text, heap, and stack sections and how features such as recursion and memory management map to this memory model.
3. Explain why memory leaks and dangling pointer problems occur, and what can be done by a programmer to avoid/fix them.

FPL-Translation: Language Translation and Execution

CS Core:

1. Execution models for JIT (Just-In-Time), compiler, interpreter
2. Use of intermediate code, e.g., bytecode
3. Limitations and benefits of JIT, compiler, and interpreter
4. Cross compilers/transpilers
5. BNF and extended BNF representation of context-free grammar
6. Parse tree using a simple sentence such as arithmetic expression or if-then-else statement
7. Execution as native code or within a virtual machine
8. Language translation pipeline: syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution

KA Core:

9. Run-time representation of core language constructs such as objects (method tables) and functions that can be passed as parameters to and returned from functions (closures)
10. Secure compiler development (See also: SEC-Foundations, SEC-Coding)

Illustrative learning outcomes:

CS Core:

1. Explain and understand the differences between compiled, JIT, and interpreted language implementations, including the benefits and limitations of each.
2. Differentiate syntax and parsing from semantics and evaluation.

3. Use BNF and extended BNF to specify the syntax of simple constructs such as if-then-else, type declaration and iterative constructs for known languages such as C++ or Python.
4. Illustrate the parse tree using a simple sentence/arithmetic expression.
5. Illustrate translation of syntax diagrams to BNF/extended BNF for simple constructs such as if-then-else, type declaration, iterative constructs, etc.
6. Illustrate ambiguity in parsing using nested if-then-else/arithmetic expression and show resolution using precedence order.

KA-Core:

7. Discuss the benefits and limitations of garbage collection, including the notion of reachability.

FPL-Abstraction: Program Abstraction and Representation

KA Core:

1. BNF and regular expressions
2. Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators
3. Components of a language:
   a. Definitions of alphabets, delimiters, sentences, syntax, and semantics
   b. Syntax vs semantics
4. Program as a set of non-ambiguous meaningful sentences
5. Basic programming abstractions: constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling (See also: SDF-Fundamentals)
6. Mutable vs immutable variables: advantages and disadvantages of reusing existing memory location vs advantages of copying and keeping old values; storing partial computation vs recomputation
7. Types of variables: static, local, nonlocal, global; need and issues with nonlocal and global variables.
8. Scope rules: static vs dynamic; visibility of variables; side-effects.
9. Side-effects induced by nonlocal variables, global variables and aliased variables.

Non-core:

10. L-values and R-values: mapping mutable variable-name to L-values; mapping immutable variable-names to R-values
11. Environment vs store and their properties
12. Data and control abstraction
13. Mechanisms for information exchange between program units such as procedures, functions, and modules: nonlocal variables, global variables, parameter-passing, import-export between modules
14. Data structures to represent code for execution, translation, or transmission.
15. Low level instruction representation such as virtual machine instructions, assembly language, and binary representation (See also: AR-Representation, AR-Assembly)
16. Lambda calculus, variable binding, and variable renaming. (See also: AL-Models, FPL-Formalism)
17. Types of semantics: operational, axiomatic, denotational, behavioral; define and use abstract syntax trees; contrast with concrete syntax.

Illustrative learning outcomes:

KA Core:

1. Illustrate the scope of variables and visibility using simple programs.
2. Illustrate different types of parameter-passing using simple pseudo programming language.
3. Explain side-effect using global and nonlocal variables and how to fix such programs.
4. Explain how programs that process other programs treat the other programs as their input data.
5. Describe a grammar and an abstract syntax tree for a small language.
6. Describe the benefits of having program representations other than strings of source code.
7. Implement a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator.

FPL-Syntax: Syntax Analysis

Non-core:

1. Regular grammars vs context-free grammars (See also: AL-Models)
2. Scanning and parsing based on language specifications
3. Lexical analysis using regular expressions
4. Tokens and their use
5. Parsing strategies including top-down (e.g., recursive descent, or LL) and bottom-up (e.g., LR or GLR) techniques
    a. Lookahead tables and their application to parsing
6. Language theory:
    a. Chomsky hierarchy (See also: AL-Models)
    b. Left-most/right-most derivation and ambiguity
    c. Grammar transformation
7. Parser error recovery mechanisms
8. Generating scanners and parsers from declarative specifications

Illustrative learning outcomes:

Non-core:

1. Use formal grammars to specify the syntax of languages.
2. Illustrate the role of lookahead tables in parsing.
3. Use declarative tools to generate parsers and scanners.
4. Recognize key issues in syntax definitions: ambiguity, associativity, precedence.

FPL-Semantics: Compiler Semantic Analysis

Non-core:

1. Abstract syntax trees; contrast with concrete syntax
2. Defining, traversing, and modifying high-level program representations
3. Scope and binding resolution
4. Static semantics
    a. Type checking.
    b. Define before use

   c. Annotation and extended static checking frameworks.
5. L-values/R-values (See also: SDF-Fundamentals)
6. Call semantics
7. Types of parameter-passing with simple illustrations and comparison: call by value, call by reference, call by value-result, call by name, call by need and their variations
8. Declarative specifications such as attribute grammars and their applications in handling limited context-base grammar

Illustrative learning outcomes:

Non-core:

1. Draw the abstract syntax tree for a small language.
2. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences.
3. Describe semantic analyses using an attribute grammar.

FPL-Analysis: Program Analysis and Analyzers

Non-core:

4. Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, and static single assignment
5. Undecidability and consequences for program analysis
6. Flow-insensitive analysis, such as type-checking and scalable pointer and alias analysis
7. Flow-sensitive analysis, such as forward and backward dataflow analyses
8. Path-sensitive analysis, such as software model checking and software verification
9. Tools and frameworks for implementing analyzers
10. Role of static analysis in program optimization and data dependency analysis during exploitation of concurrency (See also: FPL-Code)
11. Role of program analysis in (partial) verification and bug-finding (See also: FPL-Code)
12. Parallelization:
    a. Analysis for auto-parallelization
    b. Analysis for detecting concurrency bugs

Illustrative learning outcomes:

Non-core:

1. Explain the difference between dataflow graph and control flow graph.
2. Explain why non-trivial sound program analyses must be approximate.
3. Argue why an analysis is correct (sound and terminating).
4. Explain why potential aliasing limits sound program analysis and how alias analysis can help.
5. Use the results of a program analysis for program optimization and/or partial program correctness.

FPL-Code: Code Generation

Non-core:

1. Instruction sets (See also: AR-Assembly)
2. Control flow

3. Memory management (See also: AR-Memory, OS-Memory)
4. Procedure calls and method dispatching
5. Separate compilation; linking
6. Instruction selection
7. Instruction scheduling (e.g., pipelining)
8. Register allocation
9. Code optimization as a form of program analysis (See also: FPL-Analysis)
10. Program generation through generative AI

Illustrative learning outcomes:

Non-core:

1. Identify all essential steps for automatically converting source code into assembly or other low-level languages.
2. Explain the low-level code necessary for calling functions/methods in modern languages.
3. Discuss why separate compilation requires uniform calling conventions.
4. Discuss why separate compilation limits optimization because of unknown effects of calls.
5. Discuss opportunities for optimization introduced by naive translation and approaches for achieving. optimization, such as instruction selection, instruction scheduling, register allocation, and peephole optimization.

FPL-Run-Time: Run-time Behavior and Systems

Non-core:

1. Process models using stacks and heaps to allocate and deallocate activation records and recovering environments using frame pointers and return addresses during a procedure call including parameter-passing examples
2. Schematics of code lookup using hash tables for methods in implementations of object-oriented programs
3. Data layout for objects and activation records
4. Object allocation in heap
5. Implementing virtual entities and virtual methods; virtual method tables and their application
6. Run-time behavior of object-oriented programs
7. Compare and contrast allocation of memory during information exchange using parameter-passing and non-local variables (using chain of static links).
8. Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)
9. Just-in-time compilation and dynamic recompilation
10. Interface to operating system (e.g., for program initialization)
11. Interoperability between programming languages including parameter-passing mechanisms and data representation (See also: AR-Representation)
    a. Big endian, little endian
    b. Data layout of composite data types such as arrays
12. Other common features of virtual machines, such as class loading, threads, and security checking
13. Sandboxing

Illustrative learning outcomes:

Non-core:

1. Discuss benefits and limitations of automatic memory management.
2. Explain the use of metadata in run-time representations of objects and activation records, such as class pointers, array lengths, return addresses, and frame pointers.
3. Compare and contrast static allocation vs stack-based allocation vs heap-based allocation of data elements.
4. Explain why some data elements cannot be automatically deallocated at the end of a procedure/method call (need for garbage collection).
5. Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation.
6. Discuss the use of sandboxing in mobile code.
7. Identify the services provided by modern language run-time systems.

FPL-Constructs: Advanced Programming Constructs

Non-core:

1. Encapsulation mechanisms
2. Delayed evaluation and infinite streams
3. Compare and contrast delayed evaluation vs eager evaluation
4. Unification vs assertion vs expression evaluation
5. Control abstractions: exception handling, continuations, monads.
6. Object-oriented abstractions: multiple inheritance, mixins, traits, multimethods
7. Metaprogramming: macros, generative programming, model-based development
8. String manipulation via pattern-matching (regular expressions)
9. Dynamic code evaluation ("eval")
10. Language support for checking assertions, invariants, and pre/post-conditions
11. Domain specific languages, such as database languages, data science languages, embedded computing languages, synchronous languages, hardware interface languages
12. Massive parallel high performance computing models and languages

Illustrative learning outcomes:

Non-core:

1. Use various advanced programming constructs and idioms correctly.
2. Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity.
3. Discuss how various advanced programming constructs interact with the definition and implementation of other language features.

FPL-Pragmatics: Language Pragmatics

Non-core:

1. Effect of technology needs and software requirements on programming language development and evolution
2. Problem domains and programming paradigm
3. Criteria for good programming language design

   a. Principles of language design such as orthogonality
   b. Defining control and iteration constructs
   c. Modularization of large software
4. Evaluation order, precedence, and associativity
5. Eager vs delayed evaluation
6. Defining control and iteration constructs
7. External calls and system libraries

Illustrative learning outcomes:

Non-core:

1. Discuss the role of concepts such as orthogonality and well-chosen defaults in language design.
2. Objectively evaluate and justify language-design decisions.
3. Implement an example program whose result can differ under different rules for evaluation order, precedence, or associativity.
4. Illustrate uses of delayed evaluation, such as user-defined control abstractions.
5. Discuss the need for allowing calls to external calls and system libraries and the consequences for language implementation.

FPL-Formalism: Formal Semantics

Non-core:

1. Syntax vs semantics
2. Approaches to semantics: axiomatic, operational, denotational, type-based
3. Axiomatic semantics of abstract constructs such as assignment, selection, iteration using pre-condition, post-conditions, and loop invariant
4. Operational semantics analysis of abstract constructs and sequence of such as assignment, expression evaluation, selection, iteration using environment and store
   a. Symbolic execution
   b. Constraint checkers
5. Denotational semantics
   a. Lambda Calculus. (See also: AL-Models, FPL-Functional)
6. Proofs by induction over language semantics
7. Formal definitions and proofs for type systems (See also: FPL-Types)
   a. Propositions as types (implication as a function, conjunction as a product, disjunction as a sum)
   b. Dependent types (universal quantification as dependent function, existential quantification as dependent product)
   c. Parametricity

Illustrative learning outcomes:

Non-core:

1. Construct formal semantics for a small language.
2. Write a lambda-calculus program and show its evaluation to a normal form.
3. Discuss the different approaches of operational, denotational, and axiomatic semantics.
4. Use induction to prove properties of all programs in a language.

5. Use induction to prove properties of all programs in a language that is well-typed according to a formally defined type system.
6. Use parametricity to establish the behavior of code given only its type.

FPL-Methodologies: Formal Development Methodologies

1. Formal specification languages and methodologies
2. Theorem provers, proof assistants, and logics
3. Constraint checkers (See also: FPL-Formalism)
4. Dependent types (universal quantification as dependent function, existential quantification as dependent product) (See also: FPL-Types, FPL-Formalism)
5. Specification and proof discharge for fully verified software systems using pre/post conditions, refinement types, etc.
6. Formal modeling and manual refinement/implementation of software systems.
7. Use of symbolic testing and fuzzing in software development.
8. Model checking.
9. Understanding of situations where formal methods can be effectively applied and how to structure development to maximize their value.

Illustrative learning outcomes:

Non-core:

1. Use formal modeling techniques to develop and validate architectures.
2. Use proof assisted programming languages to develop fully specified and verified software artifacts.
3. Use verifier and specification support in programming languages to formally validate system properties.
4. Integrate symbolic validation tooling into a programming workflow.
5. Discuss when and how formal methods can be effectively used in the development process.

FPL-Design: Design Principles of Programming Languages

Non-core:

1. Language design principles
    a. Simplicity
    b. Security (See also: SEC-Coding)
    c. Fast translation
    d. Efficient object code
    e. Orthogonality
    f. Readability
    g. Completeness
    h. Implementation strategies
2. Designing a language to fit a specific domain or problem
3. Interoperability between programming languages
4. Language portability
5. Formal description of a programming language
6. Green computing principles (See also: SEP-Sustainability)

Illustrative Learning Outcomes:

Non-core:

1. Understand what constitutes good language design and apply that knowledge to evaluate a real programming language.

FPL-SEP: Society, Ethics, and the Profession

Non-core:

1. Impact of English-centric programming languages
2. Enhancing accessibility and inclusivity for people with disabilities – Supporting assistive technologies
3. Human factors related to programming languages and usability
   a. Impact of syntax on accessibility
   b. Supporting cultural differences (e.g., currency, decimals, dates)
   c. Neurodiversity
4. Etymology of terms such as “class,” “master,” and “slave” in programming languages
5. Increasing accessibility by supporting multiple languages within applications (UTF)

Illustrative learning outcomes:

Non-core:

1. Consciously design programming languages to be inclusive and non-offensive.

Professional Dispositions

1. **Professional:** Students must demonstrate and apply the highest standards when using programming languages and formal methods to build safe systems that are fit for their purpose.
2. **Meticulous:** Attention to detail is essential when using programming languages and applying formal methods.
3. **Inventive:** Programming and approaches to formal proofs is inherently a creative process, students must demonstrate innovative approaches to problem solving. Students are accountable for their choices regarding the way a problem is solved.
4. **Proactive:** Programmers are responsible for anticipating all forms of user input and system behavior and to design solutions that address each one.
5. **Persistent:** Students must demonstrate perseverance since the correct approach is not always self-evident and a process of refinement may be necessary to reach the solution.

Mathematics Requirements

Required:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar. (See also: MSF-Discrete)
- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction). (See also: MSF-Discrete)

- Mathematics – Matrices, probability, statistics. (See also: MSF-Probability, MSF-Statistics)

Course Packaging Suggestions

Two example courses are presented illustrating how the content may be covered. The first is an introductory course which covers the CS Core and KA Core content. This course focuses on the different programming paradigms and ensures familiarity with each to a level sufficient to be able to decide which paradigm is appropriate in each circumstance.

The second course is an advanced course focused on the implementation of a programming language, the formal description of a programming language and a formal description of the behavior of a program.

While these two courses have been the predominant way to cover this knowledge area over the past decade, it is by no means the only way that this content can be covered. Institutions can, for example, choose to cover only the CS Core content (24 hours) as part of one or spread over multiple courses (e.g., Software Engineering). Natural combinations are easily identifiable since they are the areas in which the Foundations of Programming Languages knowledge area overlaps with other knowledge areas. Such overlaps have been identified throughout this knowledge area.

Programming Language Concepts (Introduction) Course to include the following:

- FPL-OOP: Object-Oriented Programming (6 hours)
- FPL-Functional: Functional Programming (7 hours)
- FPL-Logic: Logic Programming (3 hours)
- FPL-Scripting: Shell Scripting (2 hours)
- FPL-Event-Driven: Event-Driven and Reactive Programming (4 hours)
- FPL-Parallel: Parallel and Distributed Computing (5 hours)
- FPL-Types: Type Systems (6 hours)
- FPL-Systems: Systems Execution and Memory Model (3 hours)
- FPL-Translation: Language Translation and Execution (5 hours)
- FPL-Abstraction: Program Abstraction and Representation (3 hours)
- FPL-SEP: Society, Ethics, and the Profession (1 hour)

Prerequisites:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar. (See also: MSF-Discrete).

Programming Language Implementation (Advanced) Course to include the following:

- FPL-Types: Type Systems (3 hours)
- FPL-Translation: Language Translation and Execution (2 hours)
- FPL-Syntax: Syntax Analysis(3 hours)
- FPL-Semantics: Compiler Semantic Analysis (5 hours)
- FPL-Analysis: Program Analysis and Analyzers (5 hours)

- FPL-Code: Code Generation(5 hours)
- FPL-Run-Time: Run-time Systems (4 hours)
- FPL-Constructs: Advanced Programming Constructs (4 hours)
- FPL-Pragmatics: Language Pragmatics (3 hours)
- FPL-Formalism: Formal Semantics (5 hours)
- FPL-Methodologies: Formal Development Methodologies (5 hours)

Prerequisites:

- Discrete mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar (See also: MSF-Discrete).
- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction). (See also: MSF-Discrete).
- Introductory programming course (See also: SDF-Fundamentals).
- Programming proficiency in programming concepts such as: (See also: SDF-Fundamentals):
  - Type declarations such as basic data types, records, indexed data elements such as arrays and vectors, and class/subclass declarations, types of variables
  - Scope rules of variables
  - Selection and iteration concepts, function and procedure calls, methods, object creation
- Data structure concepts such as: (See also: SDF-DataStructures):
  - Abstract data types, sequence and string, stack, queues, trees, dictionaries (See also: SDF-Data-Structures)
  - Pointer-based data structures such as linked lists, trees, and shared memory locations (See also: SDF-Data-Structures, AL-Foundational)
  - Hashing and hash tables (See also: SDF-Data-Structures, AL-Foundational)
- System fundamentals and computer architecture concepts such as (See also: SF-Foundations):
  - Digital circuits design, clocks, bus (See also: OS-Principles)
  - registers, cache, RAM, and secondary memory (See also: OS-Memory)
  - CPU and GPU (See also: AR-Heterogeneity)
- Basic knowledge of operating system concepts such as
  - Interrupts, threads and interrupt-based/thread-based programming (See also: OS-Concurrency)
  - Scheduling, including prioritization (See also: OS-Scheduling)
  - Memory fragmentation (See also: OS-Memory)
  - Latency

Committee

Chair: Michael Oudshoorn, High Point University, High Point, NC, USA
Members:

- Annette Bieniusa, TU Kaiserslautern, Kaiserslautern, Germany
- Brijesh Dongol, University of Surrey, Guildford, UK
- Michelle Kuttel, University of Cape Town, Cape Town, South Africa
- Doug Lea, State University of New York at Oswego, Oswego, NY, USA

- James Noble, Victoria University of Wellington, Wellington, New Zealand
- Mark Marron, Microsoft Research, Seattle, WA, USA and University of Kentucky, Lexington, KY, USA
- Peter-Michael Osera, Grinnell College, Grinnell, IA, USA
- Michelle Mills Strout, University of Arizona, Tucson, AZ, USA

Contributors:

- Alan Dearle, University of St. Andrews, St. Andrews, Scotland

Graphics and Interactive Techniques (GIT)

Preamble

Computer graphics is the term used to describe the computer generation and manipulation of images and can be viewed as the science of enabling visual communication through computation. Its application domains include animation, Computer Generated Imagery (CGI) and Visual Effects (VFX); engineering; machine learning; medical imaging; scientific, information, and knowledge visualization; simulators; special effects; user interfaces; and video games. Traditionally, graphics at the undergraduate level focused on rendering, linear algebra, physics, the graphics pipeline, interaction, and phenomenological approaches. Today's graphics courses increasingly include data science, physical computing, animation, and haptics. Thus, the knowledge area (KA) expanded beyond core image-based computer graphics. At the advanced level, undergraduate institutions are more likely to offer one or several courses specializing in a specific graphics knowledge unit (KU) or topic: e.g., gaming, animation, visualization, tangible or physical computing, and immersive courses such as Augmented Reality (AR)/Virtual Reality (VR)/eXtended Reality (XR). There is considerable connection with other computer science knowledge areas (KAs): Algorithmic Foundations, Architecture and Organization, Artificial Intelligence; Human-Computer Interaction; Parallel and Distributed Computing; Specialized Platform Development; Software Engineering; and Society, Ethics, and the Profession.

For students to become adept at the use and generation of computer graphics and interactive techniques, many issues must be addressed, such as human perception and cognition, data and image file formats, display specifications and protocols, hardware interfaces, and application program interfaces (APIs). Unlike other knowledge areas, knowledge units within Graphics and Interactive Techniques may be included in a variety of elective courses. Alternatively, graphics topics may be introduced in an applied project in courses primarily covering human computer interaction, embedded systems, web development, introductory programming courses, etc. Undergraduate computer science students who study the knowledge units specified below through a balance of theory and applied instruction will be able to understand, evaluate, and/or implement the related graphics and interactive techniques as users and developers. Because technology changes rapidly, the Graphics and Interactive Techniques subcommittee attempted to avoid being overly prescriptive. Any examples of APIs, programs, and languages should be considered as appropriate examples in 2023. In effect, this is a snapshot in time.

Graphics as a knowledge area has expanded and become pervasive since the CS2013 report. AR/VR/XR, artificial intelligence, computer vision, data science, machine learning, and interfaces driven by embedded sensors in everything from cars to coffee makers use graphics and interactive techniques. The now ubiquitous smartphone has made much of the world's population regular users and creators of graphics, digital images, and the interactive techniques to manipulate them. Animations, games, visualizations, and immersive applications that ran on desktops in 2013, now can run on mobile devices. The amount of stored digital data grew exponentially since 2013, and both data and visualizations are now published by myriad sources including news media and scientific organizations. Revenue from mobile video games now exceeds that of music and movies combined [1]. CGI and VFX are employed in almost all films, animations, TV productions, advertising, and business graphics. The

number of people who create graphics has skyrocketed, as have the number of applications and generative tools used to produce graphics.

It is critical that students and faculty confront the ethical issues, questions, and conundrums that have arisen and will continue to arise in and because of applications in computer graphics. Today's headlines unfortunately already provide examples of inequity and/or wrong-doing in autonomous navigation, deepfakes, computational photography, generative images, and facial recognition.

Overview of Knowledge Units

The following knowledge units are included in Graphics and Interactive Techniques. Descriptions are included below where they are not explicitly evident from the title. Graphics as a knowledge area is unique in that many of its knowledge units can and are taught as stand-alone courses where implementation projects are critical to student mastery. Apart from Applied Rendering and Techniques which scaffolds the typical undergraduate interactive computer graphics course, the other knowledge unit are more specialized. To be consistent with the design of CS2023 those knowledge unit are limited to two weeks of instruction, corresponding roughly to 6 hours of instruction. This limitation allows a two-week knowledge unit to be added to a course. Due to that time restriction, we list their Core Topic skill levels in most of the knowledge units as "Explain." However, if one of the thematic knowledge units is implemented as a full-term course, our expectation is that the skill levels will rise to "Apply" or "Develop" which is reflected in many of the practical illustrative learning outcomes. This is not meant to be prescriptive but to encourage customization. How to implement a knowledge unit is left to the discretion of the instructor. If given a two-week constraint to teach one of the thematic knowledge units, many of us would choose to limit the topics and include an applied project. Our hope is that an inclusive list of topics will help faculty design a course that best meets their and their department's pedagogical goals.

- **GIT-Fundamentals: Fundamental Concepts.** For nearly every computer scientist and software developer, understanding of how humans interact with machines is essential.
- **GIT-Visualization: Visualization.** Visualization seeks to determine and present underlying correlated structures and relationships in data sets from a wide variety of application areas. The prime objective is to communicate the information in a way which enhances understanding.
- **GIT-Rendering: Applied Rendering and Techniques.** This unit includes basic rendering and fundamental graphics techniques that nearly every undergraduate course in graphics will cover and that are essential for further study in most graphics-related courses.
- **GIT-Modeling: Geometric Modeling.** Graphics must be encoded in computer memory, often in the form of a mathematical specification of shape and form.
- **GIT-Shading: Shading and Advanced Rendering.** This unit contains more in-depth coverage of rendering topics.
- **GIT-Animation: Computer Animation.** Computer Animation is concerned with the generation of moving imagery.
- **GIT-Simulation: Simulation.** Simulation has strong ties to Computational Science. However, in the graphics domain, simulation techniques are re-purposed to a different end. Rather than creating predictive models, the goal instead is to achieve a mixture of physical plausibility and

artistic intention. To illustrate, the goals of “model surface tension in a liquid” and “produce a crown splash” are related, but different. Depending on the simulation goals, covered topics may vary as shown.

- Particle systems
  - Integration methods (Forward Euler, Midpoint, Leapfrog)
- Rigid Body Dynamics
  - Particle systems
  - Collision Detection
  - Triangle/point
  - Edge/edge
- Cloth
  - Particle systems
  - Mass/spring networks
  - Collision Detection
- Particle-Based Water
  - Integration methods
  - Smoother Particle Hydrodynamics (SPH) Kernels
  - Signed Distance Function-Based Collisions
- Grid-Based Smoke and Fire
  - Semi-Lagrangian Advection
  - Pressure Projection
- Grid and Particle-Based Water
  - Particle-Based Water
- Grid-Based Smoke and Fire
  - Semi-Lagrangian Advection
  - Pressure Projection
- Grid and Particle-Based Water
  - Particle-Based Water
  - Grid-Based Smoke, and Fire

- **GIT-Immersion: Immersion.** Immersion includes Augmented Reality (AR), Virtual Reality (VR), and Mixed Reality (MR).
- **GIT-Interaction: Interaction.** Interactive computer graphics is a requisite part of real-time applications ranging from the utilitarian-like word processors to virtual and/or augmented reality applications.
- **GIT-Image: Image Processing.** Image Processing consists of the analysis and processing of images for multiple purposes, but most frequently to improve image quality and to manipulate imagery. It lies at the cornerstone of computer vision.
- **GIT-Physical: Tangible/Physical Computing.** Tangible/Physical Computing refers to microcontroller-based interactive systems that detect and respond to sensor input.
- **GIT-SEP: Society, Ethics, and the Profession.**

Changes since CS2013

In order to align CS2013's Graphics and Visualization areas with the ACM Special Interest Group on Graphic and Interactive Techniques (SIGGRAPH) and to reflect the natural expansion of the field to include haptic and physical computing in addition to images, we have renamed it Graphics and Interactive Techniques (GIT). To capture the expanded footprint of the knowledge area , the following five knowledge units have been added to the original list consisting of Fundamental Concepts, Visualization, Basic Rendering (renamed Rendering), Geometric Modeling, Advanced Rendering (renamed Shading), and Computer Animation.

- Immersion (MR, AR, VR)
- Interaction
- Image Processing
- Tangible/Physical Computing
- Simulation

Core Hours

Knowledge Unit	CS Core	KA Core
Fundamental Concepts	4	3
Visualization		6
Applied Rendering and Techniques		15
Geometric Modeling		6
Shading and Advanced Rendering		6
Computer Animation		6
Simulation		6
Immersion (MR, AR, VR)		6
Interaction		4
Image Processing		6
Tangible/Physical Computing		6
Society, Ethics, and the Profession	Included in SEP hours	

Total	**4**	

Knowledge Units

GIT-Fundamentals: Fundamental Concepts

CS Core:

1. Uses of computer graphics and interactive techniques and their potential risks and abuses.
    a. Entertainment, business, and scientific applications: e.g., visual effects, generative imagery, computer vision, machine learning, user interfaces, video editing, games and game engines, computer-aided design and manufacturing, data visualization, and virtual/augmented/mixed reality
    b. Intellectual property, deep fakes, facial recognition, privacy (See also: SEP-DEIA, SEP-Privacy, SEP-IP, SEP-Professional-Ethics)
2. Graphic output
    a. Displays (e.g., LCD)
    b. Printers
    c. Analog film
    d. Concepts
        i. Resolution (e.g., pixels, dots)
        ii. Aspect ratio
        iii. Frame rate
3. Human vision system
    a. Tristimulus reception (RGB)
    b. Eye as a camera (projection)
    c. Persistence of vision (frame rate, motion blur)
    d. Contrast (detection, Mach banding, dithering/aliasing)
    e. Non-linear response (dynamic range, tone mapping)
    f. Binocular vision (stereo)
    g. Accessibility (color deficiency, strobing, monocular vision, etc.) (See also: SEP-DEIA, HCI-User)
4. Standard image formats
    a. Raster
        i. Lossless (e.g., TIF)
        ii. Lossy (e.g., JPG, GIF, etc.)
    b. Vector (e.g., SVG, Adobe Illustrator)
5. Digitization of analog data
    a. Rasterization
    b. Resolution
    c. Sampling and quantization
6. Color models: additive (RGB), subtractive (CMYK), and color perception (HSV)
7. Tradeoffs between storing image data and re-computing image data
8. Spatialization: coordinate systems, absolute and relative positioning
9. Animation as a sequence of still images

KA Core:

10. Applied interactive graphics (e.g., processing, python)
11. Display characteristics (protocols and ports)

Illustrative Learning Outcomes:

CS Core:

1. Identify common uses of digital presentation to humans (e.g., computer graphics, sound).
2. Describe how analog signals can be reasonably represented by discrete samples, for example, how images can be represented by pixels.
3. Compute the memory requirement for storing a color image given its resolution.
4. Create a graphic depicting how the limits of human perception affect choices about the digital representation of analog signals.
5. Indicate when and why you should use each of the following common file formats: JPG, PNG, MP3, MP4, and GIF.
6. Describe color models and their use in graphics display devices.
7. Compute the memory requirements for a multi-second movie (lasting *n* seconds) displaying at a specific framerate (*f* frames per second) at a specified resolution (*r* pixels per frame)
8. Compare and contrast digital video to analog video.
9. Describe the basic process of producing continuous motion from a sequence of discrete frames (sometimes called “flicker fusion”).
10. Describe a possible visual misrepresentation that could result from digitally sampling an analog world.
11. Compute memory space requirements based on resolution and color coding.
12. Compute time requirements based on refresh rates and rasterization techniques.

KA Core:

13. Design a user interface and an alternative for persons with color perception deficiency.
14. Construct a simple graphical user interface using a graphics library.

GIT-Visualization: Visualization

KA Core:

1. Scientific Data Visualization and Information Visualization
2. Visualization techniques
    a. Statistical visualization (e.g., scatterplots, bar graphs, histograms, line graphs, pie charts, trees, and graphs)
    b. Text visualization
    c. Geospatial visualization
    d. 2D/3D scalar fields
    e. Vector fields
    f. Direct volume rendering
3. Visualization pipeline
    a. Structuring data
    b. Mapping data to visual representations (e.g., scales, grammar of graphics)
    c. View transformations (e.g., pan, zoom, filter, select)

4. Common data formats (e.g., HDF, netCDF, geotiff, GeoJSON, shape files, raw binary, JSON, CSV, plain text)
5. High-dimensional data handling techniques
    a. Statistical (e.g., averaging, clustering, filtering)
    b. Perceptual (e.g., multi-dimensional vis, parallel coordinates, trellis plots)
6. Perceptual and cognitive foundations that drive visual abstractions.
    a. Human optical system
    b. Color theory
    c. Gestalt theories
7. Design and evaluation of visualizations
    a. Purpose (e.g., analysis, communication, aesthetics)
    b. Accessibility
    c. Appropriateness of encodings
    d. Misleading visualizations

Illustrative Learning Outcomes:

KA Core:

1. Compare and contrast data visualization and information visualization.
2. Deploy basic algorithms for visualization.
3. Compare the tradeoffs of visualization algorithms in terms of accuracy and performance.
4. Design a suitable visualization for a particular combination of data characteristics, application tasks, and audience.
5. Analyze the effectiveness of a given visualization for a particular task.
6. Design a process to evaluate the utility of a visualization algorithm or system.
7. Identify a variety of applications of visualization including representations of scientific, medical, and mathematical data; flow visualization; and spatial analysis.

GIT-Rendering: Applied Rendering and Techniques

KA Core: (See also: SPD-Game)

1. Object and scene modeling
    a. Object representations: polygonal, parametric, etc.
    b. Modeling transformations: affine and coordinate-system transformations
    c. Scene representations: scene graphs
2. Camera and projection modeling
    a. Pinhole cameras, similar triangles, and projection model
    b. Camera models
    c. Projective geometry
3. Radiometry and light models
    a. Radiometry
    b. Rendering equation
    c. Rendering in nature – emission and scattering, etc.
4. Rendering
    a. Simple triangle rasterization
    b. Rendering with a shader-based API

c. Visibility and occlusion, including solutions to this problem (e.g., depth buffering, Painter's algorithm, and ray tracing)
d. Texture mapping, including minification and magnification (e.g., trilinear MIP mapping)
e. Application of spatial data structures to rendering.
f. Ray tracing
g. Sampling and anti-aliasing

Illustrative Learning Outcomes:

KA Core:

1. Describe and illustrate the light transport problem (i.e., light is emitted, scatters around the scene, and is measured by the eye).
2. Describe the basic rendering pipeline.
3. Compare and contrast how forward and backwards rendering factor into the graphics pipeline.
4. Create a program to display 2D shapes in a window.
5. Create a program to display 3D models.
6. Produce linear perspective from similar triangles by converting points (x, y, z) to points (x/z, y/z, 1).
7. Compute two-dimensional and 3-dimensional points by applying affine transformations.
8. Indicate the changes required to extend 2D transformation operations to handle transformations in 3D.
9. Define texture mapping, sampling, and anti-aliasing, and describe examples of each.
10. Compare ray tracing and rasterization for the visibility problem.
11. Construct a program that performs transformation and clipping operations on simple two-dimensional shapes.
12. Implement a simple real-time renderer using a rasterization API (e.g., OpenGL, webGL) using vertex buffers and shaders.
13. Compare and contrast the different rendering techniques.
14. Compare and contrast the difference in transforming the camera vs the models.

GIT-Modeling: Geometric Modeling

KA Core:

1. Basic geometric operations such as intersection calculation and proximity tests on 2D objects
2. Surface representation/model
   a. Tessellation
   b. Mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, and marching cubes/tetrahedrons
   c. Parametric polynomial curves and surfaces
   d. Implicit representation of curves and surfaces
   e. Spatial subdivision techniques
3. Volumetric representation/model
   a. Volumes, voxels, and point-based representations.
   b. Signed Distance Fields
   c. Sparse Volumes, i.e., VDB
   d. Constructive Solid Geometry (CSG) representation
4. Procedural representation/model

   a. Fractals
   b. L-Systems
5. Multi-resolution modeling (See also: SPD-Game)
6. Reconstruction, e.g., 3D scanning, photogrammetry

Illustrative Learning Outcomes:

KA Core:

1. Contrast representing curves and surfaces in both implicit and parametric forms.
2. Create simple polyhedral models by surface tessellation.
3. Create a mesh representation from an implicit surface.
4. Create a fractal model or terrain using a procedural method.
5. Create a mesh from data points acquired with a laser scanner.
6. Create CSG models from simple primitives, such as cubes and quadric surfaces.
7. Contrast modeling approaches with respect to space and time complexity and quality of image.

GIT-Shading: Shading and Advanced Rendering

KA Core:

1. Solutions and approximations to the rendering equation, for example
   a. Distribution ray tracing and path tracing
   b. Photon mapping
   c. Bidirectional path tracing
   d. Metropolis light transport
2. Time (motion blur), lens position (focus), and continuous frequency (color) and their impact on rendering
3. Shadow mapping
4. Occlusion culling
5. Bidirectional Scattering Distribution function (BSDF) theory and microfacets
6. Subsurface scattering
7. Area light sources
8. Hierarchical depth buffering
9. Image-based rendering
10. Non-photorealistic rendering
11. Realtime rendering
12. GPU architecture (See also: AR-Heterogeneity)
13. Human visual systems including adaptation to light, sensitivity to noise, and flicker fusion (See also: HCI-Accessibility, SEP-DEIA)

Illustrative Learning Outcomes:

KA Core:

1. Describe how an algorithm estimates a solution to the rendering equation.
2. Discuss the properties of a rendering algorithm (e.g., complete, consistent, and unbiased).
3. Analyze the bandwidth and computation demands of a simple shading algorithm.
4. Implement a non-trivial shading algorithm (e.g., toon shading, cascaded shadow maps) under a rasterization API.

5. State how a particular artistic technique might be implemented in a renderer.
6. Describe how one might recognize the shading techniques used to create a particular image.
7. Write a program that implements any of the specified graphics techniques using a primitive graphics system at the individual pixel level.
8. Write a ray tracer for scenes using a simple (e.g., Phong's) Bidirectional Reflection Distribution Function (BRDF) plus reflection and refraction.

GIT-Animation: Computer Animation

KA Core:

1. Principles of Animation: Squash and Stretch, Timing, Anticipation, Staging, Follow Through and Overlapping Action, Straight Ahead Action, and Pose-to-Pose Action, Slow In and Out, Arcs, Exaggeration, and Appeal
2. Types of animation
    a. 2- and 3-dimensional animation
    b. Motion graphics
    c. Motion capture
    d. Motion graphics
    e. Stop animation
3. Key-frame animation
    a. Keyframe Interpolation Methods: Lerp/Slerp/Spline
4. Forward and inverse kinematics (See also: SPD-Robot, AI-Robotics)
5. Skinning algorithms
    a. Capturing
    b. Linear blend, dual quaternion
    c. Rigging
    d. Blend shapes
    e. Pose space deformation
6. Motion capture
    a. Set up and fundamentals
    b. Blending motion capture clips
    c. Blending motion capture and keyframe animation
    d. Ethical considerations (See also: SEP-DEIA, SEP-Privacy)
        i. Avoidance of "default" captures - there is no typical human walk cycle.
        ii. Accessibility

Illustrative Learning Outcomes:

KA Core:

1. Using a simple open-source character model and rig, describe visually why each of the principles of animation is fundamental to realistic animation.
2. Compute the location and orientation of model parts using a forward kinematic approach.
3. Compute the orientation of articulated parts of a model from a location and orientation using an inverse kinematic approach.
4. Compare the tradeoffs in different representations of rotations.

5. Write a script that implements the spline interpolation method for producing in-between positions and orientations.
6. Deploy off-the-shelf animation software to construct, rig, and animate simple organic forms.

GIT-Simulation: Simulation

KA Core:

1. Collision detection and response
   a. Signed Distance Fields
   b. Sphere/sphere
   c. Triangle/point
   d. Edge/edge
2. Procedural animation using noise
3. Particle systems
   a. Integration methods (e.g., forward Euler, midpoint, leapfrog)
   b. Mass/spring networks
   c. Position-based dynamics
   d. Rules (e.g., boids, crowds)
   e. Rigid bodies
4. Grid-based fluids
   a. Semi-Lagrangian advection
   b. Pressure projection
5. Heightfields
   a. Terrain: transport, erosion
   b. Water: ripple, shallow water.
6. Rule-based systems (e.g., L-systems, space-colonizing systems, Game of Life)

Illustrative Learning Outcomes:

KA Core:

1. Implement algorithms for physical modeling of particle dynamics using simple Newtonian mechanics (e.g., Witkin & Kass, snakes and worms, symplectic Euler, Stormer/Verlet, or midpoint Euler methods)
2. Contrast the basic ideas behind fluid simulation methods for modeling ballistic trajectories (e.g., for splashes, dust, fire, or smoke).
3. Implement a smoke solver with user interaction.

GIT-Immersion: Immersion

KA Core: (See also: SPD-Game, SPD-Mobile, HCI-Design)

1. Immersion levels (i.e., Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR))
2. Definitions of and distinctions between immersion and presence
3. 360 Video
4. Stereoscopic display
   a. Head-mounted displays
   b. Stereo glasses

5. Viewer tracking
    a. Inside out and outside In
    b. Head/Body/Hand/tracking
6. Time-critical rendering to achieve optimal Motion To Photon (MTP) latency
    a. Multiple Levels Of Details (LOD)
    b. Image-based VR
    c. Branching movies
7. Distributed VR, collaboration over computer network
8. Presence and factors that impact level of immersion
9. 3D interaction
10. Applications in medicine, simulation, training, and visualization
11. Safety in immersive applications
    a. Motion sickness
    b. VR obscures the real world, which increases the potential for falls and physical accidents

Illustrative Learning Outcomes:

KA Core:

1. Create a stereoscopic image.
2. Design and write an AR or VR application.
3. Summarize the pros and cons of different types of viewer tracking.
4. Compare and contrast the differences between geometry- and image-based virtual reality.
5. Analyze the design issues of user action synchronization and data consistency in a networked environment.
6. Create the specifications for an augmented reality application to be used by surgeons in the operating room.
7. Assess an immersive application's accessibility (See also: HCI-Accessibility, SEP-DEIA)
8. Identify the most important technical characteristics of a VR system/application that should be controlled to avoid motion sickness and explain why.

GIT-Interaction: Interaction

KA Core:

1. Event Driven Programming (See also: FPL-Event-Driven)
    a. Mouse or touch events
    b. Keyboard events
    c. Voice input
    d. Sensors
    e. Message passing communication
    f. Network events
2. Graphical User Interface (Single Channel)
    a. Window
    b. Icons
    c. Menus
    d. Pointing Devices
3. Accessibility (See also: SEP-DEIA)

Non-core:

4. Gestural Interfaces (See also: SPD-Game)
    a. Touch screen gestures
    b. Hand and body gestures
5. Haptic Interfaces
    a. External actuators
    b. Gloves
    c. Exoskeletons
6. Multimodal Interfaces
7. Head-worn Interfaces
    a. Brain-computer interfaces, e.g., Electroencephalography (EEG) electrodes and Multi-Electrode Arrays (MEAs)
    b. Headsets with embedded eye tracking
    c. AR glasses
8. Natural Language Interfaces (See also: AI-NLP)

Illustrative Learning Outcomes:

KA Core:

1. Create a simple game that responds to single channel mouse and keyboard events.
2. Create a mobile app that responds to touch events.
3. Design and create an application that responds to different event triggers.

None-core:

4. Assess the consistency or lack of consistency in cross-platform touch screen gestures.
5. Design and create an application that provides haptic feedback.
6. Write a program that is controlled by gestures.

GIT-Image: Image Processing

KA Core: (See also: AI-Vision)

1. Morphological operations
    a. Connected components
    b. Dilation
    c. Erosion
    d. Computing region properties (area, perimeter, centroid, etc.)
2. Color histograms
    a. Representation
    b. Contrast enhancement through normalization
3. Image enhancement
    a. Convolution
    b. Blur (e.g., Gaussian)
    c. Sharpen (e.g., Laplacian)
    d. Frequency filtering (e.g., low-pass, high-pass)
4. Image restoration

   a. Noise, degradation
   b. Inpainting and other completion algorithms
   c. Wiener filter
5. Image coding
   a. Redundancy
   b. Compression (e.g., Huffman coding)
   c. Discrete Cosine Transform (DCT), wavelet transform, Fourier transforms (See also: SPD-Interactive)
   d. Nyquist Theorem
   e. Watermarks
6. Connections to deep learning (e.g., Convolutional Neural Networks) (See also: AI-ML)

Illustrative Learning Outcomes:

KA Core:

1. Write a program that uses dilation and erosion to smooth the edges of a binary image.
2. Manipulating the hue of an image.
3. Write a program that applies a high-pass filter to an image. (The advanced variation would be to filter an image using a high-pass filter in the frequency domain.)
4. Write a program that restores missing parts of an image using an in-paint algorithm (e.g., Poisson image editing)
5. Assess the results of selectively filtering an image in the frequency domain.

GIT-Physical: Tangible/Physical Computing

KA Core:

1. Interaction with the physical world (See also: SPD-Embedded)
   a. Acquisition of data from sensors
   b. Driving external actuators
2. Connection to physical artifacts
   a. Computer-Aided Design (CAD)
   b. Computer-Aided Manufacturing (CAM)
   c. Fabrication (See also: HCI-Design)
      i. Prototyping
      ii. Additive (3D printing)
      iii. Subtractive (Computer Numerical Control (CNC) milling)
      iv. Forming (vacuum forming)
3. Internet of Things (See also: SPD-Interactive)
   a. Network connectivity
   b. Wireless communication

Illustrative Learning Outcomes:

KA Core:

1. Construct a simple virtual switch or application button and use it to turn on an LED.
2. Construct a simple system to move a servo in response to sensor data.

3. Create a circuit and accompanying microcontroller code that uses a light sensor to vary a property of something else (e.g., color or brightness of an LED or graphic, position of an external actuator).
4. Create a circuit with a variable resistor and write a microcontroller program that reads and responds to the resistor's changing values.
5. Create a 3D form in a CAD package.
    a. Show how affine transformations are achieved in the CAD program.
    b. Show an example of instances of an object.
    c. Create a fabrication plan. Provide a cost estimate for materials and time. How will you fabricate it?
    d. Fabricate it. How closely did your actual fabrication process match your plan? Where did it differ?
6. Write the G- and M-Code to construct a 3D maze and use a CAD/CAM package to check your work.
7. Decide and defend your decision to use Ethernet, WiFi, Bluetooth, RFID/NFC, or something else for internet connectivity when designing an IoT pill dispenser. Create an IoT pill dispenser.
8. Distinguish between the different types of fabrication and describe when you would use each.

GIT-SEP: Society, Ethics, and the Profession

KA Core:

1. Accessibility in immersive, interactive, and physical computing applications (See also: SEP-DEIA)
    a. Accessible to people with mobility impairments
    b. Accessible to people with vision and/or hearing impairments
2. Ethics/privacy in graphics applications. (See also: SEP-Privacy, SEP-Professional-Ethics, and SEP-Security)
    a. Acquisition of private data (room scans, body proportions, active cameras, etc.)
    b. Can't look away from immersive applications easily
    c. Danger to self/surroundings while immersed
    d. Ethical pitfalls of facial recognition
    e. Misleading visualizations
        i. Due to incorrect data because of exaggeration, hole filling, smoothing, data cleanup, etc.
        ii. Even correct data can mislead (e.g., aliasing can cause back moving or stopped fan blades)
    f. Privacy regarding health and other personal information
    g. Bias in image processing
        i. Deep fakes
        ii. Applications that misidentify people based on skin color or hairstyle
3. Intellectual Property law as it relates to computer graphics and interactive techniques (See also: SEP-IP)
    a. images used to train generative AI
    b. images produced by generative AI
4. Current and past contributors to the field (See also: SEP-DEIA)

Illustrative Learning Outcomes:

KA Core:

1. Discuss the security issues inherent in location tags.
2. Describe the ethical pitfalls of facial recognition. Can facial recognition be used ethically? If so, how?
3. Discuss the copyright issues of using watermarked images to train a neural network.

Professional Dispositions

- **Self-directed:** Graphics hardware and software evolves rapidly. Students need to understand the importance of being a life-long learner.
- **Collaborative:** Graphics developers typically work in diverse teams composed of people with disparate subject matter expertise. Students should understand the value of being a good team member, and their teamwork skills should be cultivated and evaluated with constructive feedback.
- **Effective communicator:** Communication is critical. Students' technical communication—verbal, written, and in code—should be practiced and evaluated.
- **Creative:** Creative problem-solving lies at the core of computer graphics.

Mathematics Requirements

Required:

1. Coordinate geometry
2. Trigonometry
3. MSF-Linear*
    a. Points (coordinate systems & homogeneous coordinates), vectors, and matrices
    b. Vector operations: addition, scaling, dot and cross products
    c. Matrix operations: addition, multiplication, determinants
    d. Affine transformations
4. MSF-Calculus*
    a. Continuity

*Note, if students enroll in a graphics class without linear algebra or calculus, graphics faculty can teach what is needed. To wit, many graphics textbooks cover the requisite mathematics in the appendix.

Desirable:

1. MSF-Linear
    a. Eigenvectors and Eigen decomposition
    b. Gaussian elimination and lower upper factorization
    c. Singular value decomposition
2. MSF-Calculus
    a. Quaternions
    b. Differentiation
    c. Vector calculus
    d. Tensors
    e. Differential geometry
3. MSF-Probability

4. MSF-Statistics
5. MSF-Discrete
    a. Numerical methods for simulation

Necessary and Desirable Data Structures:

1. Data Structures necessary for this knowledge area (See also: AL-Foundational, SDF-Algorithms, SDF-Data-Structures)
    a. Directed Acyclic Graphs
    b. Tuples (points / vectors / matrices of fixed dimension)
    c. Dense 1D, 2D, 3D arrays.
2. Data Structures desirable for this knowledge area (See also: AL-Foundational, SDF-Algorithms, SDF-Data-Structures, SDF-Practices)
    a. Array Structures and Structure of Arrays
    b. Trees (e.g., K-trees, quadtrees, Huffman Trees)

Course Packaging Suggestions

Interactive Computer Graphics to include the following:

- GIT-Rendering (20 hours)
- GIT-Modeling (6 hours)
- GIT-Interaction (4 hours)
- SEP-Professional-Ethics, SEP-DEIA (3 hours)

Prerequisites:

- AL-Foundational
- AL-Strategies
- SDF-Algorithms
- SDF-Data-Structures
- SDF-Practices
- MSF-Linear as a prerequisite or cover relevant topics in class

Course objectives: Students should understand and be able to create basic computer graphics using an API. They should know how to position and orient models, the camera, and distant and local lights. Note: depending on the instructor, this course can be customized to include topics from another graphics knowledge unit, for example a two-week unit on image processing or advanced rendering.

Media Computation to include the following:

- GIT-Fundamentals (4 hours)
- GIT-Rendering (6 hours)
- GIT-Interaction (3 hours)
- SDF-Fundamentals (10 hours)
- AL-Foundational (7 hours)
- HCI-User (5 hours)
- GIT-SEP (4 hours)

Course objectives: In this introductory programming class, students should be able to explain, evaluate, and apply algorithms and arrays that use and produce digital media.

User-Centered Development to include the following:

- GIT-Fundamentals (4 hours)
- GIT-Rendering (6 hours)
- GIT-Interaction (3 hours)
- HCI-User (8 hours)
- HCI-Accessibility (3 hours)
- HCI-SEP (4 hours)
- SE-Construction (4 hours)
- SPD-Web, SPD-Game, SPD-Mobile (8 hours)

Students should be able to develop applications that are usable and useful for people. Graphical user interface (GUI) designs will be implemented and analyzed using rapid prototyping.

Tangible Computing to include the following:

- GIT-Physical (14 hours)
- GIT-Interaction (4 hours)
- SPD-Embedded (10 hours)
- HCI-User (3 hours)
- HCI-Design (3 hours)
- SEP-Privacy and SEP-DEIA (3 hours)

Prerequisites:

- AL-Foundational

Course objectives: Students should be able to use human-centered design to build circuits and program a networked microcontroller. Additionally, they will learn to work with real time sensors and understand polarity, Ohm's law, and how to work with electronics safely.

Image Processing to include the following:

- GIT-Image (20 hours)
- GIT-Interaction (4 hours)
- SEP-Privacy, SEP-DEIA and SEP-IP (3 hours)

Prerequisites:

- AL-Foundational
- AL-Strategies
- SDF-Algorithms
- SDF-Data-Structures
- SDF-Practices

Course objectives: Students should understand and be able to appropriately acquire, process, display, and save digital images.

Data Visualization to include the following:

- GIT-Visualization (20 hours)
- GIT-Interaction (4 hours)
- GIT-Fundamentals (4 hours)

- HCI-User (3 hours)
- HCI-Design (3 hours)
- SEP-Privacy, SEP-DEIA, and SEP-Professional-Ethics (3 hours)

Prerequisites:

- AL-Foundational
- AL-Strategies
- SDF-Algorithms
- SDF-Data-Structures
- SDF-Practices
- MSF-Probability
- MSF-Statistics

Course objectives: Students should understand how to select a dataset; ensure the data are accurate and appropriate; design, develop and test a usable visualization program that depicts the data; and be able to read and evaluate existing visualizations.

Simulation to include the following:

- GIT-Simulation (10 hours)
- GIT-Rendering (15 hours)
- GIT-Shading: (6 hours)
- SEP-Professional-Ethics (3 hours)

Prerequisites:

- AL-Foundational
- AL-Strategies
- SDF-Algorithms
- SDF-Data-Structures
- SDF-Practices
- MSF-Linear as a prerequisite or cover relevant topics in class
- MSF-Probability

Course objectives: Students should understand and be able to create directable simulations, both of physical and non-physical systems.

Introduction to AR and VR to include the following:

- GIT-Immersive (15 hours)
- GIT-Fundamentals (4 hours)
- GIT-Interactive (8 hours)
- SEP-Privacy, SEP-DEIA, and SEP-Professional-Ethics (3 hours)

Course objectives: Students should understand and be able to develop VR and AR applications.

Computer Animation to include the following:

- GIT-Animation (30 hours)
- SEP-Privacy, SEP-DEIA, and SEP-Professional-Ethics (3 hours)

Prerequisites:

- Interactive Computer Graphics course

Course objectives: Students should understand and be able to create short animations employing the principles of animation.

Lighting and Shading to include the following:
- GIT-Shading (12 hours)
- GIT-Modeling (6 hours)
- GIT-Interaction (4 hours)
- SEP-IP, SEP-DEIA, and SEP-Professional-Ethics (3 hours)

Prerequisites:
- Interactive Computer Graphics course

Course objectives: Students should be able to create realistic and non-photorealistic lighting and understand the underlying theory of shading and lighting.

Committee

Chair: Susan Reiser, University of North Carolina Asheville, Asheville, NC, USA

Members:
- Erik Brunvand, University of Utah, Salt Lake City, UT, USA
- Kel Elkins, NASA/GSFC Scientific Visualization Studio, Greenbelt, MD, USA
- Jeff Lait, SideFX, Toronto, Canada
- Amruth Kumar, Ramapo College, Mahwah, NJ, USA
- Paul Mihail, Valdosta State University, Valdosta, GA, USA
- Tabitha Peck, Davidson College, Davidson, NC, USA
- Ken Schmidt, NOAA NCEI, Asheville, NC, USA
- Dave Shreiner, UnityTechnologies & Sonoma State University, San Francisco, CA, USA

Contributors:
- Ginger Alford, Southern Methodist University, University Park, TX, USA
- Christopher Andrews, Middlebury College, Middlebury, VT, USA
- A. J. Christensen, NASA/GSFC Scientific Visualization Studio – SSAI, Champaign, IL, USA
- Roger Eastman, University of Maryland, College Park, MD, USA
- Ted Kim, Yale University, New Haven, CT, USA
- Barbara Mones, University of Washington, Seattle, WA, USA
- Greg Shirah, NASA/GSFC Scientific Visualization Studio, Greenbelt, MD, USA
- Beatriz Sousa Santos, University of Aveiro, Portugal
- Anthony Steed, University College, London, UK

References

1. Jon Quast, Clay Bruning, and Sanmeet Deo. "Markets: This Opportunity for Investors Is Bigger Than Movies and Music Combined." https://www.nasdaq.com/articles/this-opportunity-for-investors-is-bigger-than-movies-and-music-combined-2021-10-03. Accessed March 2024.

Human-Computer Interaction (HCI)

Preamble

Computational systems not only enable users to solve problems, but also foster social connectedness and support a broad variety of human endeavors. Thus, these systems should work well with their users and solve problems in ways that respect individual dignity, social justice, and human values and creativity. Human-computer interaction (HCI) addresses those issues from an interdisciplinary perspective that includes computer science, psychology, business strategy, and design principles.

Each user is different and, from the perspective of HCI, the design of every system that interacts with people should anticipate and respect that diversity. This includes not only accessibility, but also cultural and societal norms, neural diversity, modality, and the responses the system elicits in its users. An effective computational system should evoke trust while it treats its users fairly, respects their privacy, provides security, and abides by ethical principles.

These goals require design-centric engineering that begins with intention and with the understanding that design is an iterative process, one that requires repeated evaluation of its usability and its impact on its users. Moreover, technology evokes user responses, not only by its output, but also by the modalities with which it senses and communicates. This knowledge area heightens the awareness of these issues and should influence every computer scientist.

Changes since CS2013

Driven by this broadened perspective, the HCI knowledge area has revised the CS2013 document in several ways:

- Knowledge units have been renamed and reformulated to reflect current practice and to anticipate future technological development.
- There is increased emphasis on the nature of diversity and the centrality of design focused on the user.
- Modality (e.g., text, speech) is still emphasized given its key role throughout HCI, but with a reduced emphasis on specific modalities in favor of a more timely and empathetic approach.
- The curriculum reflects the importance of understanding and evaluating the impacts and implications of a computational system on its users, including issues in ethics, fairness, trust, and explainability.
- Given its extensive interconnections with other knowledge areas, we believe HCI is itself a cross-cutting knowledge area with connections to Artificial Intelligence, Graphics and Interactive Techniques, Software Development Fundamentals, Software Engineering, and Society, Ethics, and the Profession.

Core Hours

Knowledge Unit	CS Core	KA Core
Understanding the User	2	5
Accountability and Responsibility in Design	2	2
Accessibility and Inclusive Design	2	2
Evaluating the Design	1	2
System Design	1	5
Society, Ethics, and the Profession	Included in SEP hours	
Total Hours	**8**	**16**

Knowledge Units

HCI-User: Understanding the User: Individual goals and interactions with others

CS Core:

1. User-centered design and evaluation methods. (See also: SEP-Context, SEP-Ethical-Analysis, SEP-Professional-Ethics)
    a. "You are not the users"
    b. User needs-finding
    c. Formative studies
    d. Interviews
    e. Surveys
    f. Usability tests

KA Core:

2. User-centered design methodology. (See also: SE-Tools)
    a. Personas/persona spectrum
    b. User stories/storytelling and techniques for gathering stories
    c. Empathy maps
    d. Needs assessment (techniques for uncovering needs and gathering requirements - e.g., interviews, surveys, ethnographic and contextual enquiry) (See also: SE-Requirements)
    e. Journey maps
    f. Evaluating the design (See also: HCI-Evaluation)
    g. Interfacing with stakeholders, as a team
    h. Risks associated with physical, distributed, hybrid and virtual teams
3. Physical and cognitive characteristics of the user
    a. Physical capabilities that inform interaction design (e.g., color perception, ergonomics)

   b. Cognitive models that inform interaction design (e.g., attention, perception and recognition, movement, memory)
   c. Topics in social/behavioral psychology (e.g., cognitive biases, change blindness)
4. Designing for diverse user populations. (See also: SEP-DEIA, HCI-Accessibility)
   a. How differences (e.g., in race, ability, age, gender, culture, experience, and education) impact user experiences and needs
   b. Internationalization
   c. Designing for users from other cultures
   d. Cross-cultural design
   e. Challenges to effective design evaluation. (e.g., sampling, generalization; disability and disabled experiences)
   f. Universal design
5. Collaboration and communication (See also: AI-SEP, SE-Teamwork, SEP-Communication, SPD-Game)
   a. Understanding the user in a multi-user context
   b. Synchronous group communication (e.g., chat rooms, conferencing, online games)
   c. Asynchronous group communication (e.g., email, forums, social networks)
   d. Social media, social computing, and social network analysis
   e. Online collaboration
   f. Social coordination and online communities
   g. Avatars, characters, and virtual worlds

Non-core:

6. Multi-user systems

Illustrative Learning Outcomes:

CS Core:

1. Conduct a user-centered design process that is integrated into a project.

KA Core:

2. Compare and contrast the needs of users with those of designers.
3. Identify the representative users of a design and discuss who else could be impacted by it.
4. Describe empathy and evaluation as elements of the design process.
5. Carry out and document an analysis of users and their needs.
6. Construct a user story from a needs assessment.
7. Redesign an existing solution to a population whose needs differ from those of the initial target population.
8. Contrast the different needs-finding methods for a given design problem.
9. Reflect on whether your design would benefit from low-tech or no-tech components.

Non-core:

10. Recognize the implications of designing for a multi-user system/context.

HCI-Accountability: Accountability and Responsibility in Design

CS Core: (See also: SEP-Context)

1. Design impact
    a. Sustainability (See also: SEP-Sustainability)
    b. Inclusivity (See also: SEP-DEIA)
    c. Safety, security and privacy (See also: SEP-Security, SEC-Foundations)
    d. Harm and disparate impact (See also: SEP-DEIA)
2. Ethics in design methods and solutions (See also: SEP-Ethical-Analysis, SEP-Context, SEP-Intellectual Property)
    a. The role of artificial intelligence (See also: AI-SEP)
    b. Responsibilities for considering stakeholder impact and human factors (See also: SEP-Professional-Ethics)
    c. Role of design to meet user needs
3. Requirements in design (See also: SEP-Professional-Ethics)
    a. Ownership responsibility
    b. Legal frameworks, compliance requirements
    c. Consideration beyond immediate user needs, including via iterative reconstruction of problem analysis and "digital well-being" features

KA Core:

4. Value-sensitive design (See also: SEP-Ethical-Analysis, SEP-Context, SEP-Communication)
    a. Identify direct and indirect stakeholders
    b. Determine and include diverse stakeholder values and value systems.
5. Persuasion through design (See also: SEP-Communication)
    a. Assess the persuasive content of a design
    b. Employ persuasion as a design goal
    c. Distinguish persuasive interfaces from manipulative interfaces

Illustrative Learning Outcomes:

CS Core:

1. Identify and critique the potential impacts of a design on society and relevant communities to address such concerns as sustainability, inclusivity, safety, security, privacy, harm, and disparate impact.

KA Core:

2. Identify the potential human factor elements in a design.
3. Identify and understand direct and indirect stakeholders.
4. Develop scenarios that consider the entire lifespan of a design, beyond the immediately planned uses that anticipate direct and indirect stakeholders.
5. Identify and critique the potential factors in a design that impact direct and indirect stakeholders and broader society (e.g., transparency, sustainability of the system, trust, artificial intelligence).
6. Assess the persuasive content of a design and its intent relative to user interests.
7. Critique the outcomes of a design given its intent.
8. Understand the impact of design decisions.

HCI-Accessibility: Accessibility and Inclusive Design

CS Core:

1. Background (See also: SEP-DEIA, SEP-Security)
    a. Societal and legal support for and obligations to people with disabilities
    b. Accessible design benefits everyone
2. Techniques
    a. Accessibility standards (e.g., Web Content Accessibility Guidelines) (See also: SPD-Web)
3. Technologies (See also: SE-Tools)
    a. Features and products that enable accessibility and support inclusive development by designers and engineers
4. IDFs (Inclusive Design Frameworks) (See also: SEP-DEIA)
    a. Recognizing differences
5. Universal design

KA Core:

6. Background
    a. Demographics and populations (permanent, temporary, and situational disability)
    b. International perspectives on disability (See also: SEP-DEIA)
    c. Attitudes towards people with disabilities (See also: SEP-DEIA)
7. Techniques
    a. UX (user experience) design and research
    b. Software engineering practices that enable inclusion and accessibility. (See also: SEP-DEIA)
8. Technologies
    a. Examples of accessibility-enabling features, such as conformance to screen readers
9. Inclusive Design Frameworks
    a. Creating inclusive processes such as participatory design
    b. Designing for larger impact

Non-core:

10. Background (See also: SEP-DEIA)
    a. Unlearning and questioning
    b. Disability studies
11. Technologies: the Return On Investment (ROI) of inclusion
12. Inclusive Design Frameworks: user-sensitive inclusive design (See also: SEP-DEIA)
13. Critical approaches to HCI (e.g., inclusivity) (See also: SEP-DEIA)

Illustrative Learning Outcomes:

CS Core:

1. Identify accessibility challenges faced by people with different disabilities and specify the associated accessible and assistive technologies that address them. (See also: AI-Agents, AI-Robotics)
2. Identify appropriate inclusive design approaches, such as universal design and ability-based design.
3. Identify and demonstrate understanding of software accessibility guidelines.
4. Demonstrate recognition of laws and regulations applicable to accessible design.

KA Core:

5. Apply inclusive frameworks to design, such as universal design and usability and ability-based design, and demonstrate accessible design of visual, voice-based, and touch-based UIs.
6. Demonstrate understanding of laws and regulations applicable to accessible design.
7. Demonstrate understanding of what is appropriate and inappropriate high level of skill during interaction with individuals from diverse populations.
8. Analyze web pages and mobile apps for current standards of accessibility.

Non-core:

9. Biases towards disability, race, and gender have historically, either intentionally or unintentionally, informed technology design.
    a. Find examples.
    b. Consider how those experiences (learnings?) might inform design.
10. Conceptualize user experience research to identify user needs and generate design insights.

HCI-Evaluation: Evaluating the Design

CS Core:

1. Methods for evaluation with users
    a. Formative (e.g., needs-finding, exploratory analysis) and summative assessment (e.g., functionality and usability testing)
    b. Elements to evaluate (e.g., utility, efficiency, learnability, user satisfaction, affective elements such as pleasure and engagement)
    c. Understanding ethical approval requirements before engaging in user research (See also: SE-Tools, SEP-Ethical-Analysis, SEP-Security, SEP-Privacy, SEP-Professional-Ethics)

KA Core:

2. Methods for evaluation with users (See also: SE-Validation)
    a. Qualitative methods (qualitative coding and thematic analysis)
    b. Quantitative methods (statistical tests)
    c. Mixed methods (e.g., observation, think-aloud, interview, survey, experiment)
    d. Presentation requirements (e.g., reports, personas)
    e. User-centered testing
    f. Heuristic evaluation
    g. Challenges and shortcomings to effective evaluation (e.g., sampling, generalization)
3. Study planning
    a. How to set study goals
    b. Hypothesis design
    c. Approvals from Institutional Research Boards and ethics committees (See also: SEP-Ethical-Analysis, SEP-Security, SEP-Privacy)
    d. How to pre-register a study
    e. Within-subjects vs between-subjects design
4. Implications and impacts of design with respect to the environment, material, society, security, privacy, ethics, and broader impacts. (See also: SEC-Foundations)

a. The environment
b. Material
c. Society
d. Security
e. Privacy
f. Ethics
g. Broader impacts

Non-core:

5. Techniques and tools for quantitative analysis
    a. Statistical packages
    b. Visualization tools
    c. Statistical tests (e.g., ANOVA, t-tests, post-hoc analysis, parametric vs non-parametric tests)
    d. Data exploration and visual analytics; how to calculate effect size.
6. Data management
    a. Data storage and data sharing (open science)
    b. Sensitivity and identifiability.

Illustrative Learning Outcomes:

CS Core:

1. Discuss the differences between formative and summative assessment and their role in evaluating design

KA Core:

2. Select appropriate formative or summative evaluation methods at different points throughout the development of a design.
3. Discuss the benefits of using both qualitative and quantitative methods for evaluation.
4. Evaluate the implications and broader impacts of a given design.
5. Plan a usability evaluation for a given user interface, and justify its study goals, hypothesis design, and study design.
6. Conduct a usability evaluation of a given user interface and draw defensible conclusions given the study design.

Non-core:

7. Select and run appropriate statistical tests on provided study data to test for significance in the results.
8. Pre-register a study design, with planned statistical tests.

HCI-Design: System Design

CS Core:

1. Prototyping techniques and tools
    a. Low-fidelity prototyping
    b. Rapid prototyping
    c. Throw-away prototyping

   d. Granularity of prototyping
2. Design patterns
   a. Iterative design
   b. Universal design (See also: SEP-DEIA)
   c. Interaction design (e.g., data-driven design, event-driven design)
3. Design constraints
   a. Platforms (See also: SPD-Game)
   b. Devices
   c. Resources
   d. Balance among usability, security and privacy (See also: SEC-Foundations)

KA Core:

4. Design patterns and guidelines
   a. Software architecture patterns
   b. Cross-platform design
   c. Synchronization considerations
5. Design processes (See also: SEP-Communication)
   a. Participatory design
   b. Co-design
   c. Double-diamond
   d. Convergence and divergence
6. Interaction techniques (See also: GIT-Interaction)
   a. Input and output vectors (e.g., gesture, pose, touch, voice, force)
   b. Graphical user interfaces
   c. Controllers
   d. Haptics
   e. Hardware design
   f. Error handling
7. Visual UI design (See also: GIT-Visualization)
   a. Color
   b. Layout
   c. Gestalt principles

Non-core:

8. Immersive environments (See also: GIT-Immersion)
   a. XR (encompasses virtual reality, augmented reality, and mixed reality)
   b. Spatial audio
9. 3D printing and fabrication
10. Asynchronous interaction models
11. Creativity support tools
12. Voice UI designs

Illustrative Learning Outcomes:

CS Core:

1. Propose system designs tailored to a specified appropriate mode of interaction.
2. Follow an iterative design and development process that incorporates the following:
   a. Understanding the user
   b. Developing an increment
   c. Evaluating the increment
   d. Feeding those results into a subsequent iteration
3. Explain the impact of changing constraints and design tradeoffs (e.g., hardware, user, security.) on system design.

KA Core:

4. Evaluate architectural design approaches in the context of project goals.
5. Identify synchronization challenges as part of the user experience in distributed environments.
6. Evaluate and compare the privacy implications behind different input techniques for a given scenario.
7. Explain the rationale behind a UI design based on visual design principles.

Non-core:

8. Evaluate the privacy implications within a VR/AR/MR scenario

HCI-SEP: Society, Ethics, and the Profession

CS Core:

1. Universal and user-centered design (See also: HCI-User, SEP-DEIA)
2. Accountability (See also: HCI-Accountability)
3. Accessibility and inclusive design (See also: SEP-DEIA, SEP-Security)
4. Evaluating the design (See also: HCI-Evaluation)
5. System design (See also: HCI-Design)

KA Core:

6. Participatory and inclusive design processes
7. Evaluating the design: Implications and impacts of design: with respect to the environment, material, society, security, privacy, ethics, and broader impacts (See also: SEC-Foundations, SEP-Privacy)

Non-core:

8. VR/AR/MR scenarios

Illustrative Learning Outcomes:

CS Core:

1. Conduct a user-centered design process that is integrated into a project.
2. Identify and critique the potential impacts of a design on society and relevant communities to address such concerns as sustainability, inclusivity, safety, security, privacy, harm, and disparate impact.

KA Core:

2. Critique a recent example of a non-inclusive design choice, its societal implications, and propose potential design improvements.
3. Evaluating the design: Identify the implications and broader impacts of a given design.

Non-core:

4. Evaluate the privacy implications within a VR/AR/MR scenario.

Professional Dispositions

- **Adaptable:** An HCI practitioner should be adaptable to address dynamic changes in technology, user needs, and design challenges.
- **Meticulous:** An HCI practitioner should be meticulous in ensuring that their products are both user-friendly and meet the objectives of the project.
- **Empathetic:** An HCI practitioner must demonstrate understanding of the user's needs.
- **Team-oriented:** The successful HCI practitioner should focus on the success of the team.
- **Inventive:** An HCI practitioner should design solutions that are informed by past practice, the needs of the audience, and HCI fundamentals. Creativity is required to blend these into something that solves the problem appropriately and elegantly.

Mathematics Requirements

Required:

- Basic statistics (MSF-Statistics) to support the evaluation and interpretation of results, including central tendency, variability, frequency distribution.

Course Packaging Suggestions

Introduction to HCI for CS majors and minors, to include the following:

- HCI-User: Understanding the User (7 hours)
- HCI-Accountability: Accountability and Responsibility in Design: (2 hours)
- HCI-Accessibility: Accessibility and Inclusive Design: (4 hours)
- HCI-Evaluation: Evaluating the Design: (3 hours)
- HCI-Design: System Design: (10 hours)
- HCI-SEP: Society, Ethics, and the Profession: (2 hours)

Prerequisites:

- CS2

Course objectives: A student who completes this course should be able to describe user-centered design principles and apply them in the context of a small project.

Introduction to Data Visualization to include the following:

- GIT-Visualization (30 hours)
- GIT-Rendering (10 hours)
- HCI-User: Understanding the User (3 hours)

- SEP-Privacy, SEP-Ethical-Analysis (4 hours)

Prerequisites:

- CS2
- MSF-Linear

Course objectives: Students should understand how to select a dataset; ensure the data are accurate and appropriate; and design, develop and test a visualization program that depicts the data and is usable.

Advanced Course: Usability Testing

- HCI-User (5 hours)
- HCI-Accountability (3 hours)
- HCI-Accessibility (4 hours)
- HCI-Evaluation (20 hours)
- HCI-Design (3 hours)
- HCI-SEP (5 hours)

Prerequisites:

- Introductory/Foundation courses in HCI
- Research methods, MSF-Statistics

Course objectives: Students should be able to formally evaluate products including the design and execution of usability test tasks, recruitment of appropriate users, design of test tasks, design of the test environment, test plan development and implementation, analysis and interpretation of the results, and documentation and presentation of results and recommendations. Students should be able to select appropriate techniques, procedures, and protocols to apply in various situations for usability testing with users. Students should also be able to design an appropriate evaluation plan, effectively conduct the usability test, collect data, and analyze results so that they can suggest improvements.

Committee

Chair: Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, NY, USA

Members:

- Sherif Aly, The American University of Cairo, Cairo, Egypt
- Jeremiah Blanchard, University of Florida, Gainesville, FL, USA
- Zoya Bylinskii, Adobe Research, Cambridge, MA, USA
- Paul Gestwicki, Ball State University, Muncie, IN, USA
- Susan Reiser, University of North Carolina at Asheville, Asheville, NC, USA
- Amanda M. Holland-Minkley, Washington and Jefferson College, Washington, PA, USA
- Ajit Narayanan, Google, Chennai, India
- Nathalie Riche, Microsoft, Redmond, WA, USA
- Kristen Shinohara, Rochester Institute of Technology, Rochester, NY, USA
- Olivier St-Cyr, University of Toronto, Toronto, Canada

Mathematical and Statistical Foundations (MSF)

Preamble

A strong mathematical foundation remains a bedrock of computer science education and infuses the practice of computing whether in developing algorithms, designing systems, modeling real-world phenomena, or computing with data. This Mathematical and Statistical Foundations (MSF) knowledge area – the successor to the ACM CS2013 [1] curriculum's "Discrete Structures" area – seeks to identify the mathematical and statistical material that undergirds modern computer science. The change of name corresponds to a realization both that the broader name better describes the combination of topics from the 2013 report and from those required for the recently growing areas of computer science, such as artificial intelligence, machine learning, data science, and quantum computing, many of which have continuous mathematics as their foundations.

The committee sought the following inputs to prepare their recommendations:

- A survey about mathematical preparation distributed to computer science faculty (nearly 600 respondents) across a variety of institutional types and in various countries;
- Math-related curricular views amongst data collected from ACM's survey of industry professionals (865 respondents);
- Mathematics requirements stated by all the knowledge areas in the report;
- Direct input from the computer science theory community; and
- Review of past curricular reports including recent ones on data science (e.g., Park City report [2]) and quantum computing education.

Changes since CS2013

The breadth of mathematics needed to address the mathematical needs of rapidly growing areas such as artificial intelligence, machine learning, robotics, data science, and quantum computing has grown beyond discrete structures. These areas call for a renewed focus on probability, statistics, and linear algebra, as supported by the faculty survey that asked respondents to rate various mathematical areas in their importance for both an industry career as well as for graduate school: the combined such ratings for probability, statistics, and linear algebra, for example, were 98%, 98% and 89% respectively, reflecting a strong consensus in the computer science academic community.

Core Hours

Acknowledging some tensions

Several challenges face computer science (CS) programs when weighing mathematics requirements: (1) many CS majors, perhaps aiming for a software career, are unenthusiastic about investing in mathematics; (2) institutions such as liberal-arts colleges often limit how many courses a major may require, while others may require common engineering courses that fill up the schedule; and (3) some programs adopt a more pre-professional curricular outlook while others emphasize a more foundational one. Thus, we are hesitant to recommend an all-encompassing set of mathematical topics that "every

CS degree must require." Instead, we outline two sets of *core* requirements, a CS Core set suited to hours-limited majors and a more expansive set of CS Core plus KA Core to align with technically focused programs. The principle here is that considering the additional foundational mathematics needed for AI, data science, and quantum computing, programs ought to consider as much as possible from the more expansive CS+KA version unless there are sound institutional reasons for alternative requirements.

Note: the hours in a row (example: linear algebra) add up to 40 (= 5 + 35), reflecting a standard course; shorter combined courses may be created, for example, by including probability in discrete mathematics (29 hours of discrete mathematics, 11 hours of probability).

Knowledge Unit	CS Core	KA Core
Discrete Mathematics	29	11
Probability	11	29
Statistics	10	30
Linear Algebra	5	35
Calculus	0	40
Total	**55**	**145**

Rationale for recommended hours

CS Core: While some discrete mathematics courses include probability, we highlight its importance with a minimum number of hours (11) to reflect the strong consensus in the academic community based on the survey. Taken together, the total CS Core across discrete mathematics and probability (40 hours) is typical of a one-term course. Fifteen hours are allotted to statistics and linear algebra for basic definitions so that, for example, students should at least be familiar with terms like *mean*, *standard deviation,* and *vector*. These could be covered in regular computer science courses. Many programs typically include a broader statistics requirement.

KA Core: The KA Core hours can be read as the remaining hours available to flesh out each topic into a standard 40-hour course. Note that the calculus hours roughly correspond to the typical Calculus-I course now standard across the world. Based on our survey, most programs already require Calculus-I. However, we have left out Calculus-II (an additional 40 hours) and left it to programs to decide whether Calculus-II should be added to program requirements. Programs could choose to require a more rigorous calculus-based probability or statistics sequence, or non-calculus-based versions. Similarly, linear algebra can be taught as an applied course without a calculus prerequisite or as a more advanced course.

Knowledge Units

MSF-Discrete: Discrete Mathematics

CS Core:

1. Sets, relations, functions, cardinality
2. Recursive mathematical definitions
3. Proof techniques (induction, proof by contradiction)
4. Permutations, combinations, counting, pigeonhole principle
5. Modular arithmetic
6. Logic: truth tables, connectives (operators), inference rules, formulas, normal forms, simple predicate logic
7. Graphs: basic definitions
8. Order notation

Illustrative Learning Outcomes:

CS Core:

1. Sets, Relations, and Functions, Cardinality
   a. Explain with examples the basic terminology of functions, relations, and sets.
   b. Perform the operations associated with sets, functions, and relations.
   c. Relate practical examples to the appropriate set, function, or relation model, and interpret the associated operations and terminology in context.
   d. Calculate the size of a finite set, including making use of the sum and product rules and inclusion-exclusion principle.
   e. Explain the difference between finite, countable, and uncountable sets.
2. Recursive mathematical definitions
   a. Apply recursive definitions of sequences or structures (e.g., Fibonacci numbers, linked lists, parse trees, fractals).
   b. Formulate inductive proofs of statements about recursive definitions.
   c. Solve a variety of basic recurrence relations.
   d. Analyze a problem to determine underlying recurrence relations.
   e. Given a recursive/iterative code snippet, describe its underlying recurrence relation, hypothesize a closed form for the recurrence relation, and prove the hypothesis correct (usually, using induction).
3. Proof Techniques
   a. Identify the proof technique used in a given proof.
   b. Outline the basic structure of each proof technique (direct proof, proof by contradiction, and induction) described in this unit.
   c. Apply each of the proof techniques (direct proof, proof by contradiction, and induction) correctly in the construction of a sound argument.
   d. Determine which type of proof is best for a given problem.
   e. Explain the parallels between ideas of mathematical and/or structural induction to recursion and recursively defined structures.

   f. Explain the relationship between weak and strong induction and give examples of the appropriate use of each.
4. Permutations, combinations, and counting
   a. Apply counting arguments, including sum and product rules, inclusion-exclusion principle, and arithmetic/geometric progressions.
   b. Apply the pigeonhole principle in the context of a formal proof.
   c. Compute permutations and combinations of a set, and interpret the meaning in the context of the specific application.
   d. Map real-world applications to appropriate counting formalisms, such as determining the number of ways to arrange people around a table, subject to constraints on the seating arrangement, or the number of ways to determine certain hands in cards (e.g., a full house).
5. Modular arithmetic
   a. Perform computations involving modular arithmetic.
   b. Explain the notion of the greatest common divisor and apply Euclid's algorithm to compute it.
6. Logic
   a. Convert logical statements from informal language to propositional and predicate logic expressions.
   b. Apply formal methods of symbolic propositional and predicate logic, such as calculating validity of formulae, computing normal forms, or negating a logical statement.
   c. Use the rules of inference to construct proofs in propositional and predicate logic.
   d. Describe how symbolic logic can be used to model real-life situations or applications, including those arising in computing contexts such as software analysis (e.g., program correctness), database queries, and algorithms.
   e. Apply formal logic proofs and/or informal, but rigorous, logical reasoning to real problems, such as predicting the behavior of software or solving problems such as puzzles.
   f. Describe the strengths and limitations of propositional and predicate logic.
   g. Explain what it means for a proof in propositional (or predicate) logic to be valid.
7. Graphs
   a. Illustrate by example the basic terminology of graph theory, and some of the properties and special cases of types of graphs, including trees.
   b. Demonstrate different traversal methods for trees and graphs, including pre-, post-, and in-order traversal of trees, along with breadth-first and depth-first search for graphs.
   c. Model a variety of real-world problems in computer science using appropriate forms of graphs and trees, such as representing a network topology, the organization of a hierarchical file system, or a social network.
   d. Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting.

KA Core:
The recommended topics are the same between CS core and KA-core, but with far more hours, the KA-core can cover these topics in depth and might include more computing-related applications.

MSF-Probability: Probability

CS Core:

1. Basic notions: sample spaces, events, probability, conditional probability, Bayes' rule
2. Discrete random variables and distributions
3. Continuous random variables and distributions
4. Expectation, variance, law of large numbers, central limit theorem
5. Conditional distributions and expectation
6. Applications to computing, the difference between probability and statistics (as subjects)

KA Core:
The recommended topics are the same between CS core and KA-core, but with far more hours, the KA-core can cover these topics in depth and might include more computing-related applications.

Illustrative Learning Outcomes:
CS Core:
1. Basic notions: sample spaces, events, probability, conditional probability, Bayes' rule
    a. Translate a prose description of a probabilistic process into a formal setting of sample spaces, outcome probabilities, and events.
    b. Calculate the probability of simple events.
    c. Determine whether two events are independent.
    d. Compute conditional probabilities, including through applying (and explaining) Bayes' Rule.
2. Discrete random variables and distributions
    a. Define the concept of a random variable and indicator random variable.
    b. Determine whether two random variables are independent.
    c. Identify common discrete distributions (e.g., uniform, Bernoulli, binomial, geometric).
3. Continuous random variables and distributions
    a. Identify common continuous distributions (e.g., uniform, normal, exponential).
    b. Calculate probabilities using cumulative density functions.
4. Expectation, variance, law of large numbers, central limit theorem
    a. Define the concept of expectation and variance of a random variable.
    b. Compute the expected value and variance of simple or common discrete/continuous random variables.
    c. Explain the relevance of the law of large numbers and central limit theorem to probability calculations.
5. Conditional distributions and expectation
    a. Explain the distinction between joint, marginal, and conditional distributions.
    b. Compute marginal and conditional distributions from a full distribution, for both discrete and continuous random variables.
    c. Compute conditional expectations for both discrete and continuous random variables.
6. Applications to computing
    a. Describe how probability can be used to model real-life situations or applications, such as predictive text, hash tables, and quantum computation.
    b. Apply probabilistic processes to solving computational problems, such as through randomized algorithms or in security contexts.

MSF-Statistics: Statistics

CS Core:

1. Basic definitions and concepts: populations, samples, measures of central tendency, variance
2. Univariate data: point estimation, confidence intervals

KA Core:

3. Multivariate data: estimation, correlation, regression
4. Data transformation: dimension reduction, smoothing
5. Statistical models and algorithms
6. Hypothesis testing

Illustrative Learning Outcomes:

CS Core:

1. Basic definitions and concepts: populations, samples, measures of central tendency, variance
    a. Create and interpret frequency tables.
    b. Display data graphically and interpret graphs (e.g., histograms).
    c. Recognize, describe, and calculate means, medians, quantiles (location of data).
    d. Recognize, describe, and calculate variances, interquartile ranges (spread of data).
2. Univariate data: point estimation, confidence intervals
    a. Formulate maximum likelihood estimation (in linear-Gaussian settings) as a least-squares problem.
    b. Calculate maximum likelihood estimates.
    c. Calculate maximum a posteriori estimates and make a connection with regularized least squares.
    d. Compute confidence intervals as a measure of uncertainty.

KA Core:

3. Sampling, bias, adequacy of samples, Bayesian vs frequentist interpretations
4. Multivariate data: estimation, correlation, regression
    a. Formulate the multivariate maximum likelihood estimation problem as a least-squares problem.
    b. Interpret the geometric properties of maximum likelihood estimates.
    c. Derive and calculate the maximum likelihood solution for linear regression.
    d. Derive and calculate the maximum a posteriori estimates for linear regression.
    e. Implement both maximum likelihood and maximum a posteriori estimates in the context of a polynomial regression problem.
    f. Formulate and understand the concept of data correlation (e.g., in 2D)
5. Data transformation: dimension reduction, smoothing
    a. Formulate and derive Principal Component Analysis (PCA) as a least-squares problem.
    b. Geometrically interpret PCA (when solved as a least-squares problem).
    c. Describe when PCA works well (one can relate back to correlated data).
    d. Geometrically interpret the linear regression solution (maximum likelihood).
6. Statistical models and algorithms
    a. Apply PCA to dimensionality reduction problems.
    b. Describe the tradeoff between compression and reconstruction power.

c. Apply linear regression to curve-fitting problems.
d. Explain the concept of overfitting.
e. Discuss and apply cross-validation in the context of overfitting and model selection (e.g., degree of polynomials in a regression context).

MSF-Linear: Linear Algebra

CS Core:

1. Vectors: definitions, vector operations, geometric interpretation, angles: Matrices: definition, matrix operations, meaning of Ax=b.

KA Core:

2. Matrices, matrix-vector equation, geometric interpretation, geometric transformations with matrices
3. Solving equations, row-reduction
4. Linear independence, span, basis
5. Orthogonality, projection, least-squares, orthogonal bases
6. Linear combinations of polynomials, Bezier curves
7. Eigenvectors and eigenvalues
8. Applications to computer science: Principal Components Analysis (PCA), Singular Value Decomposition (SVD), page-rank, graphics

Illustrative Learning Outcomes:

CS Core:

1. Vectors: definitions, vector operations, geometric interpretation, angles
   a. Describe algebraic and geometric representations of vectors in R^n and their operations, including addition, scalar multiplication, and dot product.
   b. List properties of vectors in $R^{n.}$
   c. Compute angles between vectors in $R^{n.}$

KA Core:

2. Matrices, matrix-vector equation, geometric interpretation, geometric transformations with matrices
   a. Perform common matrix operations, such as addition, scalar multiplication, multiplication, and transposition.
   b. Relate a matrix to a homogeneous system of linear equations.
   c. Recognize when two matrices can be multiplied.
   d. Relate various matrix transformations to geometric illustrations.
3. Solving equations, row-reduction
   a. Formulate, solve, apply, and interpret properties of linear systems.
   b. Perform row operations on a matrix.
   c. Relate an augmented matrix to a system of linear equations.
   d. Solve linear systems of equations using the language of matrices.
   e. Translate word problems into linear equations.
   f. Perform Gaussian elimination.
4. Linear independence, span, basis
   a. Define subspace of a vector space.

b. List examples of subspaces of a vector space.
c. Recognize and use basic properties of subspaces and vector spaces.
d. Determine if specific subsets of a vector space are subspaces.
e. Discuss the existence of a basis of an abstract vector space.
f. Describe coordinates of a vector relative to a given basis.
g. Determine a basis for and the dimension of a finite-dimensional space.
h. Discuss spanning sets for vectors in R^n.
i. Discuss linear independence for vectors in R^n.
j. Define the dimension of a vector space.

5. Orthogonality, projection, least-squares, orthogonal bases
   a. Explain the Gram-Schmidt orthogonalization process.
   b. Define orthogonal projections.
   c. Define orthogonal complements.
   d. Compute the orthogonal projection of a vector onto a subspace, given a basis for the subspace.
   e. Explain how orthogonal projections relate to least square approximations.
6. Linear combinations of polynomials, Bezier curves
   a. Identify polynomials as generalized vectors.
   b. Explain linear combinations of basic polynomials.
   c. Describe orthogonality for polynomials.
   d. Distinguish between basic polynomials and Bernstein polynomials.
   e. Apply Bernstein polynomials to Bezier curves.
7. Eigenvectors and eigenvalues
   a. Find the eigenvalues and eigenvectors of a matrix.
   b. Define eigenvalues and eigenvectors geometrically.
   c. Use characteristic polynomials to compute eigenvalues and eigenvectors.
   d. Use eigenspaces of matrices, when possible, to diagonalize a matrix.
   e. Perform diagonalization of matrices.
   f. Explain the significance of eigenvectors and eigenvalues.
   g. Find the characteristic polynomial of a matrix.
   h. Use eigenvectors to represent a linear transformation with respect to a particularly nice basis.
8. Applications to computer science: PCA, SVD, page-rank, graphics
   a. Explain the geometric properties of PCA.
   b. Relate PCA to dimensionality reduction.
   c. Relate PCA to solving least-squares problems.
   d. Relate PCA to solving eigenvector problems.
   e. Apply PCA to reducing the dimensionality of a high-dimensional dataset (e.g., images).
   f. Explain the page-rank algorithm and understand how it relates to eigenvector problems.
   g. Explain the geometric differences between SVD and PCA.
   h. Apply SVD to a concrete example (e.g., movie rankings).

MSF-Calculus

KA Core:

1. Sequences, series, limits

2. Single-variable derivatives: definition, computation rules (chain rule etc.), derivatives of important functions, applications
3. Single-variable integration: definition, computation rules, integrals of important functions, fundamental theorem of calculus, definite vs indefinite, applications (including in probability)
4. Parametric and polar representations
5. Taylor series
6. Multivariate calculus: partial derivatives, gradient, chain-rule, vector valued functions,
7. Optimization: convexity, global vs local minima, gradient descent, constrained optimization, and Lagrange multipliers.
8. Ordinary Differential Equations (ODEs): definition, Euler method, applications to simulation, Monte Carlo integration
9. CS applications: gradient descent for machine learning, forward and inverse kinematics, applications of calculus to probability

Note: the calculus topics listed above are aligned with computer science goals rather than with traditional calculus courses. For example, multivariate calculus is often a course by itself, but computer science undergraduates only need parts of it for machine learning.

Illustrative Learning Outcomes:

KA Core:

1. Sequences, series, limits
    a. Explain the difference between infinite sets and sequences.
    b. Explain the formal definition of a limit.
    c. Derive the limit for examples of sequences and series.
    d. Explain convergence and divergence.
    e. Apply L'Hospital's rule and other approaches to resolving limits.
2. Single-variable derivatives: definition, computation rules (chain rule etc.), derivatives of important functions, applications
    a. Explain a derivative in terms of limits.
    b. Explain derivatives as functions.
    c. Perform elementary derivative calculations from limits.
    d. Apply sum, product, and quotient rules.
    e. Work through examples with important functions.
3. Single-variable integration: definition, computation rules, integrals of important functions, fundamental theorem of calculus, definite vs indefinite, applications (including in probability)
    a. Explain the definitions of definite and indefinite integrals.
    b. Apply integration rules to examples with important functions.
    c. Explore the use of the fundamental theorem of calculus.
    d. Apply integration to problems.
4. Parametric and polar representations
    a. Apply parametric representations of important curves.
    b. Apply polar representations.
5. Taylor series
    a. Derive Taylor series for some important functions.
    b. Apply the Taylor series to approximations.

6. Multivariate calculus: partial derivatives, gradient, chain-rule, vector valued functions, applications to optimization, convexity, global vs local minima.
    a. Compute partial derivatives and gradients.
    b. Work through examples with vector-valued functions with gradient notation.
    c. Explain applications to optimization.
7. ODEs: definition, Euler method, applications to simulation
    a. Apply the Euler method to integration.
    b. Apply the Euler method to a single-variable differential equation.
    c. Apply the Euler method to multiple variables in an ODE.

Professional Dispositions

We focus on dispositions helpful to students learning mathematics.

- ***Growth mindset***. Perhaps the most important of the dispositions, students should be persuaded that anyone can learn mathematics, certainly the subset foundational to CS, and that success is not dependent on innate ability.
- ***Practice mindset***. Students should be educated about the nature of "doing" mathematics and learning through practice with problems as opposed to merely listening or observing demonstrations in the classroom.
- ***Deferred gratification***. Most students are likely to learn at least some mathematics from mathematics departments unfamiliar with computing applications; computing departments should acclimate the students to the notion of waiting to see computing applications. Many of the new growth areas such as AI or quantum computing can serve as motivation.
- ***Persistence***. Student perceptions are often driven by frustration with inability to solve hard problems that they see some peers tackle seemingly effortlessly; computing departments should help promote the notion that eventual success through persistence is what matters.
- ***Skepticism and inquiry***. Students often look for "given formulas" as handed down by experts only to be memorized and used. Yet, a theoretical mindset and, more broadly, a scientific one, should feature skepticism and a curiosity about how formulas are established.

Mathematics Requirements

The most important topics expected from students entering a computing program typically correspond to pre-calculus courses in high school.

Required:

- Algebra and numeracy
    - Numeracy: numbers, operations, types of numbers, fluency with arithmetic, exponent notation, rough orders of magnitude, fractions, and decimals.
    - Algebra: rules of exponents, solving linear or quadratic equations with one or two variables, factoring, algebraic manipulation of expressions with multiple variables.
- Precalculus
    - Coordinate geometry: distances between points, areas of common shapes.
    - Functions: function notation, drawing and interpreting graphs of functions.

- Exponentials and logarithms: a general familiarity with the functions and their graphs.
- Trigonometry: familiarity with basic trigonometric functions and the unit circle.

Course Packaging Suggestions

Every department faces constraints in delivering content, which precludes merely requiring a long list of courses covering every single desired topic. These constraints include content-area ownership, faculty size, student preparation, and limits on the number of departmental courses a curriculum can require. We list below some options for offering mathematical foundations, combinations of which might best fit any specific institution.

- ***Traditional course offerings***. With this approach, a computer science department can require students to take courses provided by mathematics departments in any of the five broad mathematical areas listed above.
- ***A "Continuous Structures" analog of Discrete Structures***. Many computer science departments now offer courses that prepare students mathematically for AI and machine learning. Such courses can combine just enough calculus, optimization, linear algebra, and probability; yet others may split linear algebra into its own course. These courses have the advantage of motivating students with computing applications and including programming as pedagogy for mathematical concepts.
- ***Integration into application courses***. An application course, such as machine learning, can be spread across two courses, with the course sequence including the needed mathematical preparation taught just-in-time, or a single machine learning course can balance preparatory material with new topics. This may have the advantage of mitigating turf issues and helping students see applications immediately after encountering mathematics.
- ***Specific course adaptations***. For nearly a century, physics and engineering needs have driven the structure of calculus, linear algebra, and probability. Computer science departments can collaborate with their colleagues in mathematics departments to restructure mathematics-offered sections in those areas that are driven by computer science applications. For example, calculus could be reorganized to fit the needs of computing programs into two calculus courses, leaving a later third calculus course for engineering and physics students.

Committee

Chair: Rahul Simha, The George Washington University, Washington DC, USA

Members:

- Richard Blumenthal, Regis University, Denver, CO, USA
- Marc Deisenroth, University College London, London, UK
- MIkey Goldweber, Denison University, Granville, OH, USA
- David Liben-Nowell, Carleton College, Northfield, MN, USA
- Jodi Tims, Northeastern University, Boston, MA, USA

References

1. ACM/IEEE-CS Joint Task Force on Computing Curricula. “Computing Science Curricula 2013.” (New York, USA: ACM Press and IEEE Computer Society Press, 2013).
2. Richard D. De Veaux, Mahesh Agarwal, Maia Averett, Benjamin S. Baumer, Andrew Bray, Thomas C. Bressoud, Lance Bryant, Lei Z. Cheng, Amanda Francis, Robert Gould, Albert Y. Kim, Matt Kretchmar, Qin Lu, Ann Moskol, Deborah Nolan, Roberto Pelayo, Sean Raleigh, Ricky J. Sethi, Mutiara Sondjaja, Neelesh Tiruviluamala, Paul X. Uhlig, Talitha M. Washington, Curtis L. Wesley, David White, Ping Ye, Curriculum Guidelines for Undergraduate Programs in Data Science, *Annual Review of Statistics and Its Application*, 4, 1 (2017): 15-30.

Networking and Communication (NC)

Preamble

Networking and communication play a central role in interconnected computer systems that are transforming the daily lives of billions of people. The public internet provides connectivity for networked applications that serve ever-increasing numbers of individuals and organizations around the world. Complementing the public sector, major proprietary networks leverage their global footprints to support cost-effective distributed computing, storage, and content delivery. Advances in satellite networks expand connectivity to rural areas. Device-to-device communication underlies the emerging Internet of Things.

This knowledge area deals with key concepts in networking and communication, as well as their representative instantiations in the internet and other computer networks. Besides the basic principles of switching and layering, the area at its core provides knowledge on naming, addressing, reliability, error control, flow control, congestion control, domain hierarchy, routing, forwarding, modulation, encoding, framing, and access control. The area also covers knowledge units in network security and mobility, such as security threats, countermeasures, device-to-device communication, and multi-hop wireless networking. In addition to the fundamental principles, the area includes their specific realization of the Internet as well as hands-on skills in the implementation of networking and communication concepts. Finally, the area comprises emerging topics such as network virtualization and quantum networking.

As the main learning outcome, learners develop a thorough understanding of the role and operation of networking and communication in networked computer systems. They learn how network structure and communication protocols affect the behavior of distributed applications. The area can be used to educate not only key principles but also their specific instantiations in the internet and equip the student with hands-on implementation skills. While computer-system, networking, and communication technologies are advancing at a fast pace, the gained fundamental knowledge enables the student to readily apply the concepts in new technological settings.

Changes since CS2013

Compared to the 2013 curricula, the knowledge area broadens its core focus to expand on reliability support, routing, forwarding, and single-hop communication. Due to the enhanced core, learners acquire a deeper understanding of the impact that networking and communication have on the behavior of distributed applications. Reflecting the increased importance of network security, the area adds a respective knowledge unit as a new elective. To track the advancing frontiers in networking and communication knowledge, the social networking knowledge unit was removed and an emerging knowledge unit on topics, such as middleboxes, software-defined networks, and quantum networking, was added. Other changes consist of redistributing all the topics from the old unit on resource allocation among other units to resolve overlap between knowledge units in the 2013 curricula.

Core Hours

Knowledge Unit	CS Core	KA Core
Fundamentals	2.5 + 0.25 (SEP) + 0.25 (SF)	
Networked Applications	3.5 + 0.25 (SEP) + 0.25 (PDC)	
Reliability Support		5.75 + 0.25 (SF)
Routing And Forwarding		4
Single-Hop Communication		3
Mobility Support		4
Network Security		2.25 + 0.5 (SEC) + 0.25 (SEP)
Emerging Topics		4
Total	**7**	**24**

The shared hours correspond to overlapping concepts that are covered from a networking perspective and are only counted here.

Knowledge Units

NC-Fundamentals: Fundamentals

CS Core:

1. Importance of networking in contemporary computing, and associated challenges. (See also: SEP-Context, SEP-Privacy)
2. Organization of the internet (e.g., users, Internet Service Providers, autonomous systems, content providers, content delivery networks)
3. Switching techniques (e.g., circuit and packet)
4. Layers and their roles (application, transport, network, datalink, and physical)
5. Layering principles (e.g., encapsulation and hourglass model) (See also: SF-Foundations)
6. Network elements (e.g., routers, switches, hubs, access points, and hosts)
7. Basic queueing concepts (e.g., relationship with latency, congestion, service levels, etc.)

Illustrative Learning Outcomes:

CS Core:

1. Articulate the organization of the internet.
2. List and define the appropriate network terminology.

3. Describe the layered structure of a typical networked architecture.
4. Identify the different types of complexity in a network (edges, core, etc.).

NC-Applications: Networked Applications

CS Core:

1. Naming and address schemes (e.g., DNS, and Uniform Resource Identifiers)
2. Distributed application paradigms (e.g., client/server, peer-to-peer, cloud, edge, and fog) (See also: PDC-Communication, PDC-Coordination)
3. Diversity of networked application demands (e.g., latency, bandwidth, and loss tolerance) (See also: PDC-Communication, SEP-Sustainability, SEP-Context)
4. Coverage of application-layer protocols (e.g., HTTP)
5. Interactions with TCP, UDP, and Socket APIs (See also: PDC-Programs)

Illustrative Learning Outcomes:

CS Core:

1. Define the principles of naming, addressing, resource location.
2. Analyze the needs of specific networked application demands.
3. Describe the details of one application layer protocol.
4. Implement a simple client-server socket-based application.

NC-Reliability: Reliability Support

KA Core:

1. Unreliable delivery (e.g., UDP)
2. Principles of reliability (e.g., delivery without loss, duplication, or out of order) (See also: SF-Reliability)
3. Error control (e.g., retransmission, error correction)
4. Flow control (e.g., stop and wait, window based)
5. Congestion control (e.g., implicit and explicit congestion notification)
6. TCP and performance issues (e.g., Tahoe, Reno, Vegas, Cubic)

Illustrative Learning Outcomes:

KA Core:

1. Describe the operation of reliable delivery protocols.
2. List the factors that affect the performance of reliable delivery protocols.
3. Describe some TCP reliability design issues.
4. Design and implement a simple reliable protocol.

NC-Routing: Routing and Forwarding

KA Core:

1. Routing paradigms and hierarchy (e.g., intra/inter domain, centralized and decentralized, source routing, virtual circuits, QoS)
2. Forwarding methods (e.g., forwarding tables and matching algorithms)
3. IP and Scalability issues (e.g., NAT, CIDR, BGP, different versions of IP)

Illustrative Learning Outcomes:

KA Core:

1. Describe various routing paradigms and hierarchies.
2. Describe how packets are forwarded in an IP network.
3. Describe how the Internet tackles scalability challenges. .

NC-SingleHop: Single Hop Communication

KA Core:

1. Introduction to modulation, bandwidth, and communication media
2. Encoding and Framing
3. Medium Access Control (MAC) (e.g., random access and scheduled access)
4. Ethernet and WiFi
5. Switching (e.g., spanning trees, VLANS).
6. Local Area Network Topologies (e.g., data center, campus networks).

Illustrative Learning Outcomes:

KA Core:

1. Describe some basic aspects of modulation, bandwidth, and communication media.
2. Describe in detail a MAC protocol.
3. Demonstrate understanding of encoding and framing solution tradeoffs.
4. Describe details of the implementation of Ethernet.
5. Describe how switching works.
6. Describe one kind of a LAN topology.

NC-Security: Network Security

KA Core:

1. General intro about security (Threats, vulnerabilities, and countermeasures) (See also: SEP-Security, SEC-Foundations, SEC-Engineering)
2. Network specific threats and attack types (e.g., denial of service, spoofing, sniffing and traffic redirection, attacker-in-the-middle, message integrity attacks, routing attacks, ransomware, and traffic analysis) (See also: SEC-Foundations, SEC-Engineering)
3. Countermeasures (: SEC-Foundations, SEC-Crypto, SEC-Engineering)
    a. Cryptography (e.g. SSL, TLS, symmetric/asymmetric)
    b. Architectures for secure networks (e.g., secure channels, secure routing protocols, secure DNS, VPNs, DMZ, Zero Trust Network Access, hyper network security, anonymous communication protocols, isolation)
    c. Network monitoring, intrusion detection, firewalls, spoofing and DoS protection, honeypots, tracebacks, BGP Sec, RPKI

Illustrative Learning Outcomes:

KA Core:

1. Describe some of the threat models of network security.

2. Describe specific network-based countermeasures.
3. Analyze various aspects of network security from a case study.

NC-Mobility: Mobility

KA Core:

1. Principles of cellular communication (e.g., 4G, 5G)
2. Principles of Wireless LANs (mainly 802.11)
3. Device to device communication (e.g., IoT communication)
4. Multi-hop wireless networks (e.g., ad hoc networks, opportunistic, delay tolerant)

Illustrative Learning Outcomes:

KA Core:

1. Describe some aspects of cellular communication such as registration
2. Describe how 802.11 supports mobile users
3. Describe practical uses of device-to-device communication, as well as multihop
4. Describe one type of mobile network such as ad hoc

NC-Emerging: Emerging Topics

KA Core:

1. Middleboxes (e.g., advances in usage of AI, intent-based networking, filtering, deep packet inspection, load balancing, NAT, CDN)
2. Network Virtualization (e.g., SDN, Data Center Networks)
3. Quantum Networking (e.g., Intro to the domain, teleportation, security, Quantum Internet)
4. Satellite, mmWave, Visible Light

Illustrative Learning Outcomes:

KA Core:

1. Describe the value of advances in middleboxes in networks.
2. Describe the importance of Software Defined Networks.
3. Describe some of the added value achieved by using Quantum Networking.

Professional Dispositions

- **Meticulous:** Students must be particular about the specifics of understanding and creating networking protocols.
- **Collaborative:** Students must work together to develop multiple components that interact together and to respond to failures and threats.
- **Proactive:** Students must be able to predict failures, threats, and how to deal with them while avoiding reactive modes of operation only.
- **Professional:** Students must comply with the needs of the community and their expectations from a networked environment, and the demands of regulatory bodies.

- **Responsive:** Students must act swiftly to changes in requirements in network configurations and changing user requirements.
- **Adaptive:** Students need to reconfigure systems under varying modes of operation.

Mathematics Requirements

Required:

- MSF-Probability.
- MSF-Statistics.
- MSF-Discrete.
- MSF-Linear Simple queuing theory concepts.

Course Packaging Suggestions

Coverage of the concepts of networking including but not limited to types of applications used by the network, reliability, routing and forwarding, single hop communication, security, and other emerging topics.

Note: both courses cover the same knowledge units but with different allocation of hours for each knowledge unit.

Course objectives: By the end of this course, learners should be able to understand many of the fundamental concepts associated with networking, learn about many types of networked applications, and develop at least one, understand basic routing and forwarding, single hop communications, and deal with some issues pertaining to mobility, security, and emerging areas, all with embedded social, ethical, and issues pertaining to the profession.

Introductory Course:

- NC-Fundamentals (8 hours)
- NC-Applications (12 hours)
- NC-Reliability (6 hours)
- NC-Routing (4 hours)
- NC-SingleHop (3 hours)
- NC-Mobility (3 hours)
- NC-Security (3 hours)
- SEP-Context (1 hour)
- NC-Emerging (2 hours)

Course objectives: By the end of this course, learners would have obtained a refresher about some of the fundamental issues of networking, networked applications, reliability, and routing and forwarding, and indulged in additional details of single hop communications, mobility, security, and emerging topics in the area, all while considering embedded social and ethical issues as well as issues pertaining to the profession.

Advanced Course:

- NC-Fundamentals (3 hours)
- NC-Applications (4 hours)
- NC-Reliability(7 hours)
- NC-Routing (6 hours)
- NC-SingleHop (5 hours)
- NC-Mobility (5 hours)
- NC-Security (5 hours)
- SEP-Privacy, SEP-Security, SEP-Sustainability (2 hours)
- NC-Emerging (5 hours)

Committee

Chair: Sherif G. Aly, The American University in Cairo, Cairo, Egypt
Members:

- Khaled Harras, Carnegie Mellon University, Pittsburgh, PA, USA
- Moustafa Youssef, The American University in Cairo, Cairo, Egypt
- Sergey Gorinsky, IMDEA Networks Institute, Madrid, Spain
- Qiao Xiang, Xiamen University, Xiamen, China

Contributors:

- Alex (Xi) Chen: Huawei, Montreal, Canada

Operating Systems (OS)

Preamble

The operating system is a collection of services needed to safely interface the hardware with applications. Core topics focus on the mechanisms and policies needed to virtualize computation, memory, and Input/Output (I/O). Overarching themes that are reused at many levels in computer systems are well illustrated in operating systems (e.g., polling vs interrupts, caching, flexibility vs costs, scheduling approaches to processes, page replacement, etc.). The Operating Systems knowledge area contains the key underlying concepts for other knowledge areas — trust boundaries, concurrency, persistence, and safe extensibility.

Changes since CS2013

Changes from CS2013 include:

- Renamed File Systems knowledge unit to File Systems API and Implementation knowledge unit,
- Moved topics from the previous Performance and Evaluation knowledge unit to the Systems Fundamentals (SF) knowledge area,
- Moved some topics from File Systems API and Implementation and Device Management to the Advanced File Systems knowledge unit, and
- Added topics on systems programming and the creation of platform-specific executables to the Foundations of Programming Languages (FPL) knowledge area.

Core Hours

Knowledge Unit	CS Core	KA Core
Role and Purpose of Operating Systems	2	
Principles of Operating System	2	
Concurrency	2	1
Protection and Safety	2	1
Scheduling		2
Process Model		2
Memory Management		0.5 +1.5 (AR)
Device Management		0.5+0.5 (AR)

File Systems API and Implementation		2
Advanced File Systems		1
Virtualization		1
Real-time and Embedded Systems		1
Fault Tolerance		1
Society, Ethics, and the Profession	Included in SEP hours	
Total	8	13 (+ 2 counted in AR)

Knowledge Units

OS-Purpose: Role and Purpose of Operating Systems

CS Core:

1. Operating systems mediate between general purpose hardware and application-specific software.
2. Universal operating system functions (e.g., process, user and device interfaces, persistence of data)
3. Extended and/or specialized operating system functions (e.g., embedded systems, server types such as file, web, multimedia, boot loaders and boot security)
4. Design issues (e.g., efficiency, robustness, flexibility, portability, security, compatibility, power, safety, tradeoffs between error checking and performance, flexibility and performance, and security and performance) (See also: SEC-Engineering)
5. Influences of security, networking, multimedia, parallel and distributed computing
6. Overarching concern of security/protection: Neglecting to consider security at every layer creates an opportunity to inappropriately access resources.
   Example concepts:
   a. Unauthorized access to files on an unencrypted drive can be achieved by moving the media to another computer.
   b. Operating systems enforced security can be defeated by infiltrating the boot layer before the operating system is loaded.
   c. Process isolation can be subverted by inadequate authorization checking at API boundaries.
   d. Vulnerabilities in system firmware can provide attack vectors that bypass the operating system entirely.
   e. Improper isolation of virtual machine memory, computing, and hardware can expose the host system to attacks from guest systems.
   f. The operating system may need to mitigate exploitation of hardware and firmware vulnerabilities, leading to potential performance reductions (e.g., Spectre and Meltdown mitigations).
7. Exposure of operating systems functions in shells and systems programming. (See also: FPL-Scripting)

Illustrative Learning Outcomes:

CS Core:

1. Understand the objectives and functions of modern operating systems.
2. Evaluate the design issues in different usage scenarios (e.g., real time OS, mobile, server).
3. Understand the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve.
4. Understand how evolution and stability are desirable and mutually antagonistic in operating systems function.

OS-Principles: Principles of Operating System

CS Core:

1. Operating system software design and approaches (e.g., monolithic, layered, modular, micro-kernel, unikernel)
2. Abstractions, processes, and resources
3. Concept of system calls and links to application program interfaces (e.g., Win32, Java, Posix). (See also: AR-Assembly)
4. The evolution of the link between hardware architecture and the operating system functions
5. Protection of resources means protecting some machine instructions/functions (See also: AR-Assembly)
   Example concepts:
   a. Applications cannot arbitrarily access memory locations or file storage device addresses.
   b. Protection of coprocessors and network devices
6. Leveraging interrupts from hardware level: service routines and implementations. (See also: AR-Assembly)
   Example concepts:
   a. Timer interrupts for implementing time slices
   b. I/O interrupts for putting blocking threads to sleep without polling
7. Concept of user/system state and protection, transition to kernel mode using system calls (See also: AR-Assembly)
8. Mechanism for invoking system calls, the corresponding mode and context switch and return from interrupt (See also: AR-Assembly)
9. Performance costs of context switches and associated cache flushes when performing process switches in Spectre-mitigated environments.

Illustrative Learning Outcomes:

CS Core:

1. Understand how the application of software design approaches to operating systems design/implementation (e.g., layered, modular, etc.) affects the robustness and maintainability of an operating system.
2. Categorize system calls by purpose.
3. Understand dynamics of invoking a system call (e.g., passing parameters, mode change).
4. Evaluate whether a function can be implemented in the application layer or can only be accomplished by system calls.

5. Apply OS techniques for isolation, protection, and throughput across OS functions (e.g., starvation similarities in process scheduling, disk request scheduling, semaphores, etc.) and beyond.
6. Understand how the separation into kernel and user mode affects safety and performance.
7. Understand the advantages and disadvantages of using interrupt processing in enabling multiprogramming.
8. Analyze potential vectors of attack via the operating systems and the security features designed to guard against them.

OS-Concurrency: Concurrency

CS Core:

1. Thread abstraction relative to concurrency
2. Race conditions, critical regions (role of interrupts, if needed) (See also: PDC-Programs)
3. Deadlocks and starvation (See also: PDC-Coordination)
4. Multiprocessor issues (spin-locks, reentrancy).
5. Multiprocess concurrency vs multithreading

KA Core:

6. Thread creation, states, structures (See also: SF-Foundations)
7. Thread APIs
8. Deadlocks and starvation (necessary conditions/mitigations) (See also: PDC-Coordination)
9. Implementing thread safe code (semaphores, mutex locks, condition variables). (See also: AR-Performance-Energy, SF-Evaluation, PDC-Evaluation)
10. Race conditions in shared memory (See also: PDC-Coordination)

Non-Core:

11. Managing atomic access to OS objects (e.g., big kernel lock vs many small locks vs lockless data structures like lists)

Illustrative Learning Outcomes:

CS Core:

1. Understand the advantages and disadvantages of concurrency as inseparable functions within the operating system framework.
2. Understand how architecture level implementation results in concurrency problems including race conditions.
3. Understand concurrency issues in multiprocessor systems.

KA Core:

4. Understand the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each.
5. Understand techniques for achieving synchronization in an operating system (e.g., describe how a semaphore can be implemented using OS primitives) including intra-concurrency control and use of hardware atomics.
6. Accurately analyze code to identify race conditions and appropriate solutions for addressing race conditions.

OS-Protection: Protection and Safety

CS Core:

1. Overview of operating system security mechanisms (See also: SEC-Foundations)
2. Attacks and antagonism (scheduling, etc.) (See also: SEC-Foundations)
3. Review of major vulnerabilities in real operating systems (See also: SEC-Foundations)
4. Operating systems mitigation strategies such as backups (See also: SF-Reliability)

KA Core:

5. Policy/mechanism separation (See also: SEC-Governance)
6. Security methods and devices (See also: SEC-Foundations)
   Example concepts:
   a. Rings of protection (history from Multics to virtualized x86)
   b. x86_64 rings -1 and -2 (hypervisor and ME/PSP)
7. Protection, access control, and authentication (See also: SEC-Foundations, SEC-Crypto)

Illustrative Learning Outcomes:

CS Core:

1. Understand the requirement for protection and security mechanisms in operating systems.
2. List and describe the attack vectors that leverage OS vulnerabilities.
3. Understand the mechanisms available in an OS to control access to resources.

KA Core:

4. Summarize the features and limitations of an operating system that impact protection and security.

OS-Scheduling: Scheduling

KA Core:

1. Preemptive and non-preemptive scheduling
2. Schedulers and policies (e.g., first come, first serve, shortest job first, priority, round robin, multilevel) (See also: SF-Resource)
3. Concepts of Symmetric Multi-Processor (SMP) scheduling and cache coherence (See also: AR-Memory)
4. Timers (e.g., building many timers out of finite hardware timers) (See also: AR-Assembly)
5. Fairness and starvation

Non-Core:

6. Subtopics of operating systems such as energy-aware scheduling and real-time scheduling (See also: AR-Performance-Energy, SPD-Embedded, SPD-Mobile)
7. Cooperative scheduling, such as Linux futexes and userland scheduling.

Illustrative Learning Outcomes:

KA Core:

1. Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes.
2. Explain the relationships between scheduling algorithms and application domains.
3. Explain the distinctions among types of processor scheduler such as short-term, medium-term, long-term, and I/O.
4. Evaluate a problem or solution to determine appropriateness for asymmetric and/or symmetric multiprocessing.
5. Evaluate a problem or solution to determine appropriateness as a process vs threads.
6. List some contexts benefitting from preemption and deadline scheduling.

Non-Core:

7. Explain the ways that the logic embodied in scheduling algorithms are applicable to other operating systems mechanisms, such as first come first serve or priority to disk I/O, network scheduling, project scheduling, and problems beyond computing.

OS-Process: Process Model

KA Core:

1. Processes and threads relative to virtualization protected memory, process state, memory isolation, etc.
2. Memory footprint/segmentation (e.g., stack, heap, etc.) (See also: AR-Assembly)
3. Creating and loading executables, shared libraries, and dynamic linking **(**See also: FPL-Translation)
4. Dispatching and context switching (See also: AR-Assembly)
5. Interprocess communication (e.g., shared memory, message passing, signals, environment variables) (See also: PDC-Communication)

Illustrative Learning Outcomes:

KA Core:

1. Understand how processes and threads use concurrency features to virtualize control.
2. Understand reasons for using interrupts, dispatching, and context switching to support concurrency and virtualization in an operating system.
3. Understand the different states that a task may pass through, and the data structures needed to support the management of many tasks.
4. Understand the different ways of allocating memory to tasks, citing the relative merits of each.
5. Apply the appropriate interprocess communication mechanism for a specific purpose in a programmed software artifact.

OS-Memory: Memory Management

KA Core:

1. Review of physical memory, address translation and memory management hardware (See also: AR-Memory, MSF-Discrete)
2. Impact of memory hierarchy including cache concept, cache lookup, and per-CPU caching on operating system mechanisms and policy (See also: AR-Memory, SF-Performance)

3. Logical and physical addressing, address space virtualization (See also: AR-Memory, MSF-Discrete)
4. Concepts of paging, page replacement, thrashing and allocation of pages and frames
5. Allocation/deallocation/storage techniques (algorithms and data structure) performance and flexibility
   Example concept: Arenas, slab allocators, free lists, size classes, heterogeneously sized pages (huge pages)
6. Memory caching and cache coherence and the effect of flushing the cache to avoid speculative execution vulnerabilities (See also: AR-Organization, AR-Memory, SF-Performance)
7. Security mechanisms and concepts in memory management including sandboxing, protection, isolation, and relevant vectors of attack (See also: SEC-Foundations)

Non-Core:

8. Virtual memory: leveraging virtual memory hardware for OS services and efficiency

Illustrative Learning Outcomes:

KA Core:

1. Explain memory hierarchy and cost-performance tradeoffs.
2. Summarize the principles of virtual memory as applied to caching and paging.
3. Evaluate the tradeoffs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed.
4. Describe the reason for and use of cache memory (performance and proximity, how caches complicate isolation and virtual machine abstraction).
5. Code/Develop efficient programs that consider the effects of page replacement and frame allocation on the performance of a process and the system in which it executes.

Non-Core:

6. Explain how hardware is utilized for efficient virtualization

OS-Devices: Device management

KA Core:

1. Buffering strategies (See also: AR-IO)
2. Direct Memory Access (DMA) and polled I/O, Memory-mapped I/O (See also: AR-IO)
   Example concept: DMA communication protocols (e.g., ring buffers etc.)
3. Historical and contextual - Persistent storage device management (e.g., magnetic, Solid State Device (SSD)) (See also: SEP-History)

Non-Core:

4. Device interface abstractions, hardware abstraction layer
5. Device driver purpose, abstraction, implementation, and testing challenges
6. High-level fault tolerance in device communication

Illustrative Learning Outcomes:

KA Core:

1. Explain architecture level device control implementation and link relevant operating system mechanisms and policy (e.g., buffering strategies, direct memory access).
2. Explain OS device management layers and the architecture (e.g., device controller, device driver, device abstraction).
3. Explain the relationship between the physical hardware and the virtual devices maintained by the operating system.
4. Explain I/O data buffering and describe strategies for implementing it.
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted.

Non-Core:

6. Describe the complexity and best practices for the creation of device drivers.

OS-Files: File Systems API and Implementation

KA Core:

1. Concept of a file including data, metadata, operations, and access-mode
2. File system mounting
3. File access control
4. File sharing
5. Basic file allocation methods, including linked allocation table
6. File system structures comprising file allocation including various directory structures and methods for uniquely identifying files (e.g., name, identified or metadata storage location)
7. Allocation/deallocation/storage techniques (algorithms and data structure) impact on performance and flexibility (i.e., internal and external fragmentation and compaction)
8. Free space management such as using bit tables vs linking
9. Implementation of directories to segment and track file location

Illustrative Learning Outcomes:

KA Core:

1. Explain the choices to be made in designing file systems.
2. Evaluate different approaches to file organization, recognizing the strengths and weaknesses of each.
3. Apply software constructs appropriately given knowledge of the file system implementation.

OS-Advanced-Files: Advanced File systems

KA Core:

1. File systems: partitioning, mount/unmount, virtual file systems
2. In-depth implementation techniques
3. Memory-mapped files (See also: AR-IO)
4. Special-purpose file systems
5. Naming, searching, access, backups
6. Journaling and log-structured file systems (See also: SF-Reliability)

Non-Core: (including emerging topics)

1. Distributed file systems
2. Encrypted file systems
3. Fault tolerance

Illustrative Learning Outcomes:

KA Core:

1. Explain how hardware developments have led to changes in the priorities for the design and the management of file systems.
2. Map file abstractions to a list of relevant devices and interfaces.
3. Identify and categorize different mount types.
4. Explain specific file systems requirements and the specialized file systems features that meet those requirements.
5. Explain the use of journaling and how log-structured file systems enhance fault tolerance.

Non-Core:

6. Explain purpose and complexity of distributed file systems.
7. List examples of distributed file systems protocols.
8. Explain mechanisms in file systems to improve fault tolerance.

OS-Virtualization: Virtualization

KA Core:

1. Using virtualization and isolation to achieve protection and predictable performance. (See also: SF-Performance)
2. Advanced paging and virtual memory. (See also: SF-Performance)
3. Virtual file systems and virtual devices.
4. Containers and their comparison to virtual machines.
5. Thrashing (e.g., Popek and Goldberg requirements for recursively virtualizable systems).

Non-core:

6. Types of virtualizations (including hardware/software, OS, server, service, network). (See also: SF-Performance)
7. Portable virtualization; emulation vs isolation. (See also: SF-Performance)
8. Cost of virtualization. (See also: SF-Performance)
9. Virtual machines and container escapes, dangers from a security perspective. (See also: SEC-Engineering)
10. Hypervisors- hardware virtual machine extensions, hosts with kernel support, QEMU KVM

Illustrative Learning Outcomes:

KA Core:

1. Explain how hardware architecture provides support and efficiencies for virtualization.
2. Explain the difference between emulation and isolation.
3. Evaluate virtualization tradeoffs.

Non-Core:

4. Explain hypervisors and the need for them in conjunction with different types of hypervisors.

OS-Real-time: Real-time and Embedded Systems

KA Core:

1. Process and task scheduling.
2. Deadlines and real-time issues. (See also: SPD-Embedded)
3. Low-latency vs ”soft real-time" vs "hard real time." (See also: SPD-Embedded, FPL-Event-Driven)

Non-Core:

4. Memory/disk management requirements in a real-time environment.
5. Failures, risks, and recovery.
6. Special concerns in real-time systems (safety).

Illustrative Learning Outcomes:

KA Core:

1. Explain what makes a system a real-time system.
2. Explain latency and its sources in software systems and its characteristics.
3. Explain special concerns that real-time systems present, including risk, and how these concerns are addressed.

Non-Core:

4. Explain specific real time operating systems features and mechanisms.

OS-Faults: Fault tolerance

KA Core:

1. Reliable and available systems. (See also: SF-Reliability)
2. Software and hardware approaches to address tolerance (RAID). (See also: SF-Reliability)

Non-Core:

3. Spatial and temporal redundancy. (See also: SF-Reliability)
4. Methods used to implement fault tolerance. (See also: SF-Reliability)
5. Error identification and correction mechanisms, checksums of volatile memory in RAM. (See also: AR-Memory)
6. File system consistency check and recovery.
7. Journaling and log-structured file systems. (See also: SF-Reliability)
8. Use-cases for fault-tolerance (databases, safety-critical). (See also: SF-Reliability)
9. Examples of OS mechanisms for detection, recovery, restart to implement fault tolerance, use of these techniques for the OS’s own services. (See also: SF-Reliability)

Illustrative Learning Outcomes:

KA Core:

1. Explain how operating systems can facilitate fault tolerance, reliability, and availability.

2. Explain the range of methods for implementing fault tolerance in an operating system.
3. Explain how an operating system can continue functioning after a fault occurs.
4. Explain the performance and flexibility tradeoffs that impact using fault tolerance.

Non-Core:

5. Describe operating systems fault tolerance issues and mechanisms in detail.

OS-SEP: Society, Ethics, and the Profession

KA Core:

1. Open source in operating systems. (See also: SEP-IP)
   Example concepts:
   a. Identification of vulnerabilities in open-source kernels,
   b. Open-source guest operating systems,
   c. Open-source host operating systems, and
   d. Changes in monetization (paid vs free upgrades).
2. End-of-life issues with sunsetting operating systems.
   Example concept: Privacy implications of using proprietary operating systems/operating environments, including telemetry, automated scanning of personal data, built-in advertising, and automatic cloud integration.

Illustrative Learning Outcomes:

KA Core:

1. Explain advantages and disadvantages of finding and addressing bugs in open-source kernels.
2. Contextualize history and positive and negative impact of Linux as an open-source product.
3. List complications with reliance on operating systems past end-of-life.
4. Understand differences in finding and addressing bugs for various operating systems payment models.

Professional Dispositions

- **Proactive**: Students must anticipate the security and performance implications of how operating systems components are used.
- **Meticulous**: Students must carefully analyze the implications of operating system mechanisms on any project.

Mathematics Requirements

Required:

- MSF-Discrete

Course Packaging Suggestions

Introductory Course to include the following:

- OS-Purpose (3 hours)
- OS-Principles (3 hours)
- OS-Concurrency (7 hours)
- OS-Scheduling (3 hours)
- OS-Process (3 hours)
- OS-Memory (4 hours)
- OS-Protection (4 hours)
- OS-Devices (2 hours)
- OS-Files (2 hours)
- OS-Virtualization (3 hours)
- OS-Advanced-Files (2 hours)
- OS-Real-time (1 hour)
- OS-Faults (1 hour)
- OS-SEP (4 hours)

Prerequisites:

- AR-Assembly
- AR-Memory
- AR-Reliability
- AR-IO
- AR-Organization
- MSF-Discrete

Course objectives: Students should understand the impact and implications of operating system resource management in terms of performance and security. They should understand and implement inter-process communication mechanisms safely. They should be able to differentiate between the use and evaluation of open-source and/or proprietary operating systems. They should understand virtualization as a feature of safe modern operating system implementation.

Committee

Chair: Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
Members:

- Qiao Xiang, Xiamen University, Xiamen, China
- Mikey Goldweber, Denison University, Granville, OH, USA
- Marcelo Pias, Federal University of Rio Grande (FURG), Rio Grande, RS, Brazil
- Avi Silberschatz, Yale University, New Haven, CT, USA
- Renzo Davoli, University of Bologna, Bologna, Italy

Parallel and Distributed Computing (PDC)

Preamble

Parallel and distributed programming arranges, coordinates, and controls multiple computations occurring at the same time across different places. The ubiquity of parallelism and distribution are inevitable consequences of increasing numbers of gates in processors, processors in computers, and computers everywhere that may be used to improve performance compared to sequential programs, while also coping with the intrinsic interconnectedness of the world, and the possibility that some components or connections fail or behave maliciously. Parallel and distributed programming removes the restrictions of sequential programming that require computational steps to occur in a serial order in a single place, revealing further distinctions, techniques, and analyses applying at each layer of computing systems.

In most conventional usage, “parallel” programming focuses on establishing and coordinating multiple activities that may occur at the same time, “distributed” programming focuses on establishing and coordinating activities that may occur in different places, and “concurrent” programming focuses on interactions of ongoing activities with each other and the environment. However, all three terms may apply in most contexts. Parallelism generally implies some form of distribution because multiple activities occurring without sequential ordering constraints happen in multiple physical places (unless they rely on context-switching or quantum effects). Conversely, actions in different places need not bear any specific sequential ordering with respect to each other in the absence of communication constraints.

Parallel, distributed, and concurrent programming techniques form the core of High Performance Computing (HPC), distributed systems, and increasingly, nearly every computing application. The PDC knowledge area has evolved from a diverse set of advanced topics into a central body of knowledge and practice, permeating almost every other aspect of computing. Growth of the field has occurred irregularly across different subfields of computing, sometimes with different goals, terminology, and practices, masking the considerable overlap of basic ideas and skills that are the primary focus of this knowledge area. Nearly every problem with a sequential solution also admits parallel and/or distributed solutions; additional problems and solutions arise only in the context of concurrency. Nearly every application domain of parallel and distributed computing is a well-developed area of study and/or engineering too large to enumerate.

Changes since CS2013

This knowledge area has been refactored to focus on commonalities across different forms of parallel and distributed computing, also enabling more flexibility in KA Core coverage, with more guidance on coverage options.

Overview

This knowledge area is divided into five knowledge units, each with CS Core and KA Core topics that extend but do not overlap CS Core coverage that appears in other knowledge areas. The five

knowledge units cover: The nature of parallel and distributed **Programs** and their execution; **Communication** (via channels, memory, or shared data stores), **Coordination** among parallel activities to achieve common outcomes; **Evaluation** with respect to specifications, and **Algorithms** across multiple application domains.

CS Core topics span approaches to parallel and distributed computing but restrict coverage to those that apply to nearly all of them. Learning outcomes include developing small programs (in a choice of several styles) with multiple activities and analyzing basic properties. The topics and hours do not include coverage of specific languages, tools, frameworks, systems, and platforms needed as a basis for implementing and evaluating concepts and skills. The topics also avoid reliance on specifics that may vary widely (for example GPU programming vs cloud container deployment scripts), Prerequisites for CS Core coverage include the following.

- SDF-Fundamentals: programs, executions, specifications, implementations, variables, arrays, sequential control flow, procedural abstraction and invocation, Input/Output.
- SF-Overview: Layered systems, state machines, reliability.
- AR-Assembly, AR-Memory: von Neumann architecture, memory hierarchy.
- MSF-Discrete: Discrete structures including directed graphs.

Additionally, Foundations of Programming Languages (FPL) may be treated as a prerequisite, depending on other curricular choices. CS Core requires familiarity with languages and platforms that enable construction of parallel and distributed programs. Also, PDC includes definitions of safety, liveness, and related concepts that are covered with respect to language properties and semantics in FPL. Similarly, PDC CS Core includes concepts that are further developed in the context of network protocols in Networking and Communication (NC), Operating Systems (OS), and Security (SEC), that could be covered in any order.

KA Core topics in each unit are of the form “one or more of the following” for *a la carte* topics extending associated core topics. Any selection of KA-core topics meeting the KA Core hour requirement constitutes fulfillment of the KA Core. This structure permits variation in coverage depending on the focus of any given course (see below for examples). Depth of coverage of any KA Core subtopic is expected to vary according to course goals. For example, shared-memory coordination is a central topic in multicore programming, but much less so in most heterogeneous systems, and conversely for bulk data transfer. Similarly, fault tolerance is central to the design of distributed information systems, but much less so in most data-parallel applications.

Core Hours

Knowledge Unit	CS Core hours	KA Core hours
Programs	2	2
Communication	2	6
Coordination	2	6

Evaluation	1	3
Algorithms	2	9
Society, Ethics, and the Profession	Included in SEP hours	
Total	**9**	**26**

Knowledge Units

PDC-Programs: Programs

CS Core:

1. Parallelism
    a. Declarative parallelism: Determining which actions may, or must not, be performed in parallel, at the level of instructions, functions, closures, composite actions, sessions, tasks, and services is the main idea underlying PDC algorithms; failing to do so is the main source of errors. (See also: PDC-Algorithms)
    b. Defining order: for example, using happens-before relations or series/parallel directed acyclic graphs representing programs.
    c. Independence: determining when ordering does not matter, in terms of commutativity, dependencies, preconditions.
    d. Ensuring ordering among otherwise parallel actions when necessary, including locking, safe publication; and imposing communication – sending a message happens before receiving it; conversely relaxing when unnecessary.
2. Distribution
    a. Defining places, as devices executing actions, including hardware components, remote hosts, may also include external, uncontrolled devices, hosts, and users. (See also: AR-IO)
    b. One device may time-slice or otherwise emulate multiple parallel actions by fewer processors by scheduling and virtualization. (See also: OS-Scheduling)
    c. Naming or identifying places (e.g., device IDs) and actions as parties (e.g., thread IDs).
    d. Activities across places may communicate across media. (See also: PDC-Communication)
3. Starting activities
    a. Options that enable actions to be performed (eventually) at places range from hardwiring to configuration scripts; also establishing communication and resource management; these are expressed differently across languages and contexts, usually relying on automated provisioning and management by platforms (See also: SF-Resources)
    b. Procedural: Enabling multiple actions to start at a given program point; for example, starting new threads, possibly scoping, or otherwise organizing them in hierarchical groups
    c. Reactive: Enabling upon an event by installing an event handler, with less control of when actions begin or end, and may apply even on uniprocessors
    d. Dependent: Enabling upon completion of others; for example, sequencing sets of parallel actions (See also: PDC-Coordination)
    e. Granularity: Execution cost of action bodies should outweigh the overhead of arranging them

4. Execution Properties
    a. Nondeterministic execution of unordered actions
    b. Consistency: Ensuring agreement among parties about values and predicates when necessary to avoid races, maintain safety and atomicity, or arrive at consensus
    c. Fault tolerance: Handling failures in parties or communication, including (Byzantine) misbehavior due to untrusted parties and protocols, when necessary to maintain progress or availability (See also: SF-Reliability)
    d. Tradeoffs are one focus of evaluation (See also: PDC-Evaluation)

KA Core:

5. One or more of the following mappings and mechanisms across layered systems:
    a. CPU data- and instruction-level-parallelism (See also: AR-Organization)
    b. SIMD and heterogeneous data parallelism (See also: AR-Heterogeneity)
    c. Multicore scheduled concurrency, tasks, actors (See also: OS-Scheduling)
    d. Clusters, clouds; elastic provisioning. (See also: SPD-Common)
    e. Networked distributed systems (See also: NC-Applications)
    f. Emerging technologies such as quantum computing and molecular computing

Illustrative Learning Outcomes;

CS Core:

1. Graphically show (as a Directed Acyclic Graph (DAG)) how to parallelize a compound numerical expression; for example, a = (b + c) * (d + e).
2. Explain why the concepts of consistency and fault tolerance do not arise in purely sequential programs.

KA Core:

3. Write a function that efficiently counts events such as networking packet receptions.
4. Write a filter/map/reduce program in multiple styles.
5. Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client.

PDC-Communication: Communication

CS Core:

1. Media
    a. Varieties: channels (message passing or I/O), shared memory, heterogeneous, data stores
    b. Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation (See also: AR-IO)
2. Channels
    a. Explicit (usually named) party-to-party communication media
    b. APIs: Sockets, architectural, language-based, and toolkit constructs, such as Message Passing Interface (MPI), and layered constructs such as Remote Procedure Call (RPC) (See also: NC-Fundamentals)
    c. I/O channel APIs
3. Memory

a. Shared memory architectures in which parties directly communicate only with memory at given addresses, with extensions to heterogeneous memory supporting multiple memory stores with explicit data transfer across them; for example, GPU local and shared memory, Direct Memory Access (DMA)
b. Memory hierarchies: Multiple layers of sharing domains, scopes, and caches; locality: latency, false-sharing
c. Consistency properties: Bitwise atomicity limits, coherence, local ordering

4. Data Stores
    a. Cooperatively maintained data structures implementing maps and related ADTs
    b. Varieties: Owned, shared, sharded, replicated, immutable, versioned

KA Core:

5. One or more of the following properties and extensions
    a. Topologies: Unicast, Multicast, Mailboxes, Switches; Routing via hardware and software interconnection networks
    b. Media concurrency properties: Ordering, consistency, idempotency, overlapping communication with computation
    c. Media performance: Latency, bandwidth (throughput) contention (congestion), responsiveness (liveness), reliability (error and drop rates), protocol-based progress (acks, timeouts, mediation)
    d. Media security properties: integrity, privacy, authentication, authorization (See also: SEC-Secure Coding)
    e. Data formats: Marshaling, validation, encryption, compression
    f. Channel policies: Endpoints, sessions, buffering, saturation response (waiting vs dropping), rate control
    g. Multiplexing and demultiplexing many relatively slow I/O devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs
    h. Formalization and analysis of channel communication; for example, CSP
    i. Applications of queuing theory to model and predict performance.
    j. Memory models: sequential and release/acquire consistency
    k. Memory management; including reclamation of shared data; reference counts and alternatives
    l. Bulk data placement and transfer; reducing message traffic and improving locality; overlapping data transfer and computation; impact of data layout such as array-of-structs vs struct-of-arrays
    m. Emulating shared memory: distributed shared memory, Remote Direct Memory Access (RDMA)
    n. Data store consistency: Atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains
    o. Faults, partitioning, and partial failures; voting; protocols such as Paxos and Raft.
    p. Design tradeoffs among consistency, availability, partition (fault) tolerance; impossibility of meeting all at once
    q. Security and trust: Byzantine failures, proof of work and alternatives

Illustrative Learning Outcomes:

CS Core:

1. Explain the similarities and differences among: (1) Party A sends a message on channel X with contents 1 received by party B (2) A sets shared variable X to 1, read by B (3) A sets “X=1’ in a distributed shared map accessed by B.

KA Core:

2. Write a program that distributes different segments of a data set to multiple workers, and collects results (for the simplest example, summing segments of an array).
3. Write a parallel program that requests data from multiple sites and summarizes them using some form of reduction.
4. Compare the performance of buffered versus unbuffered versions of a producer-consumer program.
5. Determine whether a given communication scheme provides sufficient security properties for a given usage.
6. Give an example of an ordering of accesses among concurrent activities (e.g., program with a data race) that is not sequentially consistent.
7. Give an example of a scenario in which blocking message sends can deadlock.
8. Describe at least one design technique for avoiding liveness failures in programs using multiple locks.
9. Write a program that illustrates memory-access or message reordering.
10. Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates.
11. Give an example of a scenario in which an attempted optimistic update may never complete.
12. Modify a concurrent system to use a more scalable, reliable, or available data store.
13. Using an existing platform supporting replicated data stores, write a program that maintains a key-value mapping even when one or more hosts fail.

PDC-Coordination: Coordination

CS Core:

1. Dependencies
    a. Initiation or progress of one activity may be dependent on other activities, so as to avoid race conditions, ensure termination, or meet other requirements
    b. Ensuring progress by avoiding dependency cycles, using monotonic conditions, removing inessential dependencies
2. Control constructs and design patterns
    a. Completion-based: Barriers, joins, including termination control
    b. Data-enabled: Queues, producer-consumer designs
    c. Condition-based: Polling, retrying, backoffs, helping, suspension, signaling, timeouts
    d. Reactive: Enabling and triggering continuations
3. Atomicity
    a. Atomic instructions, enforced local access orderings
    b. Locks and mutual exclusion; lock granularity
    c. Using locks in a specific language; maintaining liveness without introducing races

d. Deadlock avoidance: Ordering, coarsening, randomized retries; backoffs, encapsulation via lock managers
e. Common errors: Failing to lock or unlock when necessary, holding locks while invoking unknown operations
f. Avoiding locks: replication, read-only, ownership, and non-blocking constructions

KA Core:

4. One or more of the following properties and extensions
    a. Progress properties including lock-free, wait-free, fairness, priority scheduling, interactions with consistency, reliability
    b. Performance with respect to contention, granularity, convoying, scaling
    c. Non-blocking data structures and algorithms
    d. Ownership and resource control
    e. Lock variants and alternatives: sequence locks, read-write locks; Read-Copy-Update (RCU), reentrancy; tickets; controlling spinning versus blocking
    f. Transaction-based control: Optimistic and conservative
    g. Distributed locking: reliability
    h. Alternatives to barriers: Clocks; counters, virtual clocks; dataflow and continuations; futures and RPC; consensus-based, gathering results with reducers and collectors
    i. Speculation, selection, cancellation; observability and security consequences
    j. Resource control using semaphores and condition variables
    k. Control flow: Scheduling computations, series-parallel loops with (possibly elected) leaders, pipelines and streams, nested parallelism
    l. Exceptions and failures. Handlers, detection, timeouts, fault tolerance, voting

Illustrative Learning Outcomes:

CS Core:

1. Show how to avoid or repair a race error in a given program.
2. Show how to ensure that a program correctly terminates when all of a set of concurrent tasks have completed.

KA Core:

3. Write a function that efficiently counts events such as sensor inputs or networking packet receptions.
4. Write a filter/map/reduce program in multiple styles.
5. Write a program in which the termination of one set of parallel actions is followed by another.
6. Write a program that speculatively searches for a solution by multiple activities, terminating others when one is found.
7. Write a program in which a numerical exception (such as divide by zero) in one activity causes termination of others.
8. Write a program for multiple parties to agree upon the current time of day; discuss its limitations compared to protocols such as network transfer protocol (NTP).
9. Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client.

PDC-Evaluation: Evaluation

CS Core:

1. Safety and liveness requirements in terms of temporal logic constructs to express “always” and “eventually” (See also: FPL-Parallel)
2. Identifying, testing for, and repairing violations, including common forms of errors such as failure to ensure necessary ordering (race errors), atomicity (including check-then-act errors), and termination (livelock)
3. Performance requirements metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements (See also: SF-Performance)
4. Performance impact of design and implementation choices, including granularity, overhead, consensus costs, and energy consumption (See also: SEP-Sustainability)
5. Estimating scalability limitations, for example using Amdahl’s Law or Universal Scalability Law (See also: SF-Evaluation)

KA Core:

6. One or more of the following methods and tools:
    a. Extensions to formal sequential requirements such as linearizability
    b. Protocol, session, and transactional specifications
    c. Use of tools such as Unified Modelling Language (UML), Temporal Logic of Actions (TLA), program logics
    d. Security analysis: safety and liveness in the presence of hostile or buggy behaviors by other parties; required properties of communication mechanisms (for example lack of cross-layer leakage), input screening, rate limiting (See also: SEC-Foundations)
    e. Static analysis applied to correctness, throughput, latency, resources, energy (See also: SEP-Sustainability)
    f. Directed Acyclic Graph (DAG) model analysis of algorithmic efficiency (work, span, critical paths)
    g. Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, visualizations, continuous integration, continuous deployment, and test generators
    h. Measuring and comparing throughput, overhead, waiting, contention, communication, data movement, locality, resource usage, behavior in the presence of excessive numbers of events, clients, or threads (See also: SF-Evaluation)
    i. Application domain specific analyses and evaluation techniques

Illustrative Learning Outcomes:

CS Core:

1. Revise a specification to enable parallelism and distribution without violating other essential properties or features.
2. Explain how concurrent notions of safety and liveness extend their sequential counterparts.
3. Specify a set of invariants that must hold at each bulk-parallel step of a computation.

4. Write a test program that can reveal a data race error; for example, missing an update when two activities both try to increment a variable.
5. In a given context, explain the extent to which introducing parallelism in an otherwise sequential program would be expected to improve throughput and/or reduce latency, and how it may impact energy efficiency.
6. Show how scaling and efficiency change for sample problems without and with the assumption of problem size changing with the number of processors; further explain whether and how scalability would change under relaxations of sequential dependencies.

KA Core:

7. Specify and measure behavior when a service is requested by unexpectedly many clients.
8. Identify and repair a performance problem due to sequential bottlenecks.
9. Empirically compare throughput of two implementations of a common design (perhaps using an existing test harness framework).
10. Identify and repair a performance problem due to communication or data latency.
11. Identify and repair a performance problem due to communication or data latency.
12. Identify and repair a performance problem due to resource management overhead.
13. Identify and repair a reliability or availability problem.

PDC-Algorithms: Algorithms

CS Core:

1. Expressing and implementing algorithms in given languages and frameworks, to initiate activities (for example threads), use shared memory constructs, and channel, socket, and/or remote procedure call APIs. (See also: FPL-Parallel).
    a. Data parallel examples including map/reduce.
    b. Using channel, socket, and/or RPC APIs in a given language, with program control for sending (usually procedural) vs receiving. (usually reactive or RPC-based).
    c. Using locks, barriers, and/or synchronizers to maintain liveness without introducing races.
2. Survey of common application domains across multicore, reactive, data parallel, cluster, cloud, open distributed systems, and frameworks (with reference to the following table).

Category	**Typical Execution agents**	**Typical Communication mechanisms**	**Typical Algorithmic domains**	**Typical Engineering goals**
Multicore	Threads	Shared memory, Atomics, locks	Resource management, data processing	Throughput, latency, energy
Reactive	Handlers, threads	I/O Channels	Services, real-time	Latency
Data parallel	GPU, SIMD,	Heterogeneous	Linear algebra,	Throughput,

	accelerators, hybrid	memory	graphics, data analysis	energy
Cluster	Managed hosts	Sockets, channels	Simulation, data analysis	Throughput
Cloud	Provisioned hosts	Service APIs	Web applications	Scalability
Open distributed	Autonomous hosts	Sockets, Data stores	Fault tolerant data stores and services	Reliability

KA Core:

3. One of more of the following algorithmic domains. (See also: AL-Strategies):
    a. Linear algebra: Vector and matrix operations, numerical precision/stability, applications in data analytics and machine learning.
    b. Data processing: sorting, searching and retrieval, concurrent data structures.
    c. Graphs, search, and combinatorics: Marking, edge-parallelization, bounding, speculation, network-based analytics.
    d. Modeling and simulation: differential equations; randomization, N-body problems, genetic algorithms.
    e. Computational logic: satisfiability (SAT), concurrent logic programming.
    f. Graphics and computational geometry: Transforms, rendering, ray-tracing.
    g. Resource management: Allocating, placing, recycling and scheduling processors, memory, channels, and hosts; exclusive vs shared resources; static, dynamic and elastic algorithms; Real-time constraints; Batching, prioritization, partitioning; decentralization via work-stealing and related techniques.
    h. Services: Implementing web APIs, electronic currency, transaction systems, multiplayer games.

Illustrative Learning Outcomes:

CS Core:

1. Implement a parallel/distributed component based on a known algorithm.
2. Write a data-parallel program that for example computes the average of an array of numbers.
3. Write a producer-consumer program in which one component generates numbers, and another computes their average. Measure speedups when the numbers are small scalars versus large multi-precision values.
4. Extend an event-driven sequential program by establishing a new activity in an event handler (for example a new thread in a GUI action handler).
5. Improve the performance of a sequential component by introducing parallelism and/or distribution.
6. Choose among different parallel/distributed designs for components of a given system.

KA Core:

7. Design, implement, analyze, and evaluate a component or application for X operating in a given context, where X is in one of the listed domains, for example, a genetic algorithm for factory floor design.
8. Critique the design and implementation of an existing component or application, or one developed by classmates.
9. Compare the performance and energy efficiency of multiple implementations of a similar design, for example, multicore versus clustered versus GPU.

Professional Dispositions

- **Meticulous**: Students' attention to detail is essential when applying constructs with non-obvious correctness conditions.
- **Persistent**: Students must be prepared to try alternative approaches when solutions are not self-evident.

Mathematics Requirements

Required:
- MSF-Discrete – Logic, discrete structures including directed graphs.

Desired:
- MSF-Linear
- MSF-Calculus – Differential equations

Course Packaging Suggestions

The CS Core requirements need not be provided by a single course. They may be included across courses primarily devoted to software development, programming languages, systems, data management, networking, computer architecture, and/or algorithms.

Alternatively, the CS Core provides a basis for courses focusing on parallel and/or distributed computing. At one extreme, it is possible to offer a single broadly constructed course covering all PDC KA Core topics to varying depths. At the other extreme, it is possible to infuse PDC KA Core coverage across the curriculum with courses that cover parallel and distributed approaches alongside sequential ones for nearly every topic in computing. More conventional choices include courses that focus on one or a few categories (such as multicore or cluster), and algorithmic domains (such as linear algebra, or resource management). Such courses may go into further depth than listed in one or more KUs, and include additional software development experience, but include only CS-Core-level coverage of other topics.

As an example, a course mainly focusing on multicores could extend CS Core topics as follows.

1. Programs: KA Core on threads, tasks, instruction-level parallelism.
2. Communication: KA Core on multicore architectures, memory, concurrent data stores.

3. Coordination: KA Core on blocking and non-blocking synchronization, speculation, cancellation, futures, and divide-and-conquer data parallelism.
4. Evaluation: KA Core on performance analysis.
5. Algorithms: project-based KA Core coverage of data processing and resource management.

More extensive examples and guidance for courses focusing on HPC are provided by the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing [1].

Committee

Chair: Doug Lea, State University of New York at Oswego, Oswego, NY, USA

Members:

- Sherif Aly, American University of Cairo, Cairo, Egypt
- Michael Oudshoorn, High Point University, High Point, NC, USA
- Qiao Xiang, Xiamen University, Xiamen, China
- Dan Grossman, University of Washington, Seattle, WA, USA
- Sebastian Burckhardt, Microsoft Research, Redmond WA, USA
- Vivek Sarkar, Georgia Tech, Atlanta, GA, USA
- Maurice Herlihy, Brown University, Providence, RI, USA
- Sheikh Ghafoor, Tennessee Tech, Cookeville, TN, USA
- Chip Weems, University of Massachusetts, Amherst, MA, USA

Contributors:

- Paul McKenney, Meta, Beaverton, OR, USA
- Peter Buhr, University of Waterloo, Waterloo, Ontario, Canada

References

1. Prasad, S. K., Estrada, T., Ghafoor, S., Gupta, A., Kant, K., Stunkel, C., Sussman, A., Vaidyanathan, R., Weems, C., Agrawal, K., Barnas, M., Brown, D. W., Bryant, R., Bunde, D. P., Busch, C., Deb, D., Freudenthal, E., Jaja, J., Parashar, M., Phillips, C., Robey, B., Rosenberg, A., Saule, E., Shen, C. 2020. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version II-beta, Online: http://tcpp.cs.gsu.edu/curriculum/, 53 pages. Accessed March 2024.

Software Development Fundamentals (SDF)

Preamble

Fluency in the process of software development is fundamental to the study of computer science. To use computers to solve problems most effectively, students must be competent at reading and writing programs. Beyond programming skills, however, they must be able to select and use appropriate data structures and algorithms and use modern development and testing tools.

The SDF knowledge area brings together fundamental concepts and skills related to software development, focusing on concepts and skills that should be taught early in a computer science program, typically in the first year. This includes fundamental programming concepts and their effective use in writing programs, use of fundamental data structures which may be provided by the programming language, basics of programming practices for writing good quality programs, reading, and understanding programs, and some understanding of the impact of algorithms on the performance of the programs. The 43 hours of material in this knowledge area may be augmented with core material from other knowledge areas as students progress to mid- and upper-level courses.

This knowledge area assumes a contemporary programming language with built-in support for common data types including associative data types like dictionaries/maps as the vehicle for introducing students to programming (e.g., Python, Java). However, this is not to discourage the use of older or lower-level languages for SDF – the knowledge units below can be suitably adapted for the actual language used.

The emergence of generative AI and Large Language Models (LLMs), which can generate programs for many programming tasks, will undoubtedly affect the programming profession and consequently the teaching of many CS topics. However, to be able to effectively use generative AI in programming tasks, a programmer must have a good understanding of programs, and hence must still learn the foundations of programming and develop basic programming skills - which is the aim of SDF. Consequently, we feel that the desired outcomes for SDF should remain the same, though different instructors may now give more emphasis to program understanding, documenting, specifications, analysis, and testing. (This is like teaching students multiplication, addition, etc. even though calculators can be used to do them).

Changes since CS 2013

The main change from 2013 is a stronger emphasis on developing fundamental programming skills and effective use of in-built data structures (which many contemporary languages provide) for problem solving.

Overview

This Knowledge Area has five knowledge units which follow.

1. **SDF-Fundamentals**: Fundamental Programming Concepts and Practices – This knowledge unit aims to develop an understanding of basic concepts, and the ability to fluently use basic language

constructs as well as modularity constructs. It also aims to familiarize students with the concept of common libraries and frameworks, including those to facilitate API-based access to resources.

2. **SDF-Data-Structures**: Fundamental Data Structures – This knowledge unit aims to develop core concepts relating to Data Structures and associated operations. Students should understand the important data structures available in the programming language or as libraries, and how to use them effectively, including choosing appropriate data structures while designing solutions for a given problem.
3. **SDF-Algorithms**: Algorithms – This knowledge unit aims to develop the foundations of algorithms and their analysis. The KU should also empower students in selecting suitable algorithms for building modest-complexity applications.
4. **SDF-Practices**: Software Development Practices – This knowledge unit develops the core concepts relating to modern software development practices. It aims to develop student understanding and basic competencies in program testing, enhancing the readability of programs, and using modern methods and tools including some general-purpose IDE.
5. **SDF-SEP**: Society, Ethics, and the Profession – This knowledge unit aims to develop an initial understanding of some of the ethical issues related to programming, professional values programmers need to have, and the responsibility to society that programmers have. This knowledge unit is a part of the SEP Knowledge Area.

Core Hours

Knowledge Unit	CS Core	KA Core
Fundamental Programming Concepts and Practices	20	
Fundamental Data Structures	6 + 6 (AL)	
Algorithms	3 + 3 (AL)	
Software Development Practices	5	
Society, Ethics and the Profession	Included in SEP hours	
Total	**43**	

Note: The CS Core hours include 9 hours shared with AL, but counted here.

Knowledge Units

SDF-Fundamentals: Fundamental Programming Concepts and Practices

CS Core:

1. Basic concepts such as variables, primitive data types, expressions, and their evaluation

2. How imperative programs work: state and state transitions on execution of statements, flow of control
3. Basic constructs such as assignment statements, conditional and iterative statements, basic I/O
4. Key modularity constructs such as functions (and methods and classes, if supported in the language) and related concepts like parameter passing, scope, abstraction, data encapsulation (See also: FPL-OOP)
5. Input and output using files and APIs
6. Structured data types available in the chosen programming language like sequences (e.g., arrays, lists), associative containers (e.g., dictionaries, maps), others (e.g., sets, tuples) and when and how to use them (See also: AL-Foundational)
7. Libraries and frameworks provided by the language (when/where applicable)
8. Recursion
9. Dealing with runtime errors in programs (e.g., exception handling).
10. Basic concepts of programming errors, testing, and debugging (See also: SE-Construction, SEC-Coding)
11. Documenting/commenting code at the program and module level.(See also: SE-Construction)
12. Develop a security mindset. (See also: SEC-Foundations)

Illustrative Learning Outcomes:

CS Core:

In these learning outcomes, the term "develop" means "design, write, test, and debug."

1. Develop programs that use the fundamental programming constructs: assignment and expressions, basic I/O, conditional and iterative statements.
2. Develop programs using functions with parameter passing.
3. Develop programs that effectively use the different structured data types provided in the language like arrays/lists, dictionaries, and sets.
4. Develop programs that use file I/O to provide data persistence across multiple executions.
5. Develop programs that use language-provided libraries and frameworks (where applicable).
6. Develop programs that use APIs to access or update data (e.g., from the web).
7. Develop programs that create simple classes and instantiate objects of those classes (if supported by the language).
8. Explain the concept of recursion and identify when and how to use it effectively.
9. Develop recursive functions.
10. Develop programs that can handle runtime errors.
11. Read a given program and explain what it does.
12. Write comments for a program or a module specifying what it does.
13. Trace the flow of control during the execution of a program.
14. Use appropriate terminology to identify elements of a program (e.g., identifier, operator, operand).

SDF-Data-Structures: Fundamental Data Structures

CS Core: **(**See also: AL-Foundational**)**

1. Standard abstract data types such as lists, stacks, queues, sets, and maps/dictionaries, including operations on them.

2. Selecting and using appropriate data structures.
3. Performance implications of choice of data structure(s).
4. Strings and string processing.

Illustrative Learning Outcomes:

CS Core:

1. Write programs that use each of the key abstract data types provided in the language (e.g., arrays, tuples/records/structs, lists, stacks, queues, and associative data types like sets, dictionaries/maps).
2. Select the appropriate data structure for a given problem.
3. Explain how the performance of a program may change when using different data structures or operations.
4. Write programs that work with text by using string processing capabilities provided by the language.

SDF-Algorithms: Algorithms

CS Core: **(**See also: AL-Foundational**,** AL-Complexity**)**

1. Concept of algorithm and notion of algorithm efficiency
2. Some common algorithms (e.g., sorting, searching, tree traversal, graph traversal)
3. Impact of algorithms on time-space efficiency of programs

Illustrative Learning Outcomes:

CS Core:

1. Explain the role of algorithms for writing programs.
2. Demonstrate how a problem may be solved by different algorithms, each with different properties.
3. Explain some common algorithms (e.g., sorting, searching, tree traversal, graph traversal).
4. Explain the impact on space/time performance of some algorithms.

SDF-Practices: Software Development Practices

CS Core: (See also: SE-Construction)

1. Basic testing, including test case design
2. Use of a general-purpose IDE, including its debugger
3. Programming style that improves readability
4. Specifying functionality of a module in a natural language.

Illustrative Learning Outcomes:

CS Core:

1. Develop tests for modules and apply a variety of strategies to design test cases.
2. Explain some limitations of testing programs.
3. Build, execute, and debug programs using a modern IDE and associated tools such as visual debuggers.

4. Apply basic programming style guidelines to aid readability of programs such as comments, indentation, proper naming of variables, etc.
5. Write specifications of a module as module comment describing its functionality.

SDF-SEP: Society, Ethics, and the Profession

CS Core:

1. Intellectual property rights of programmers for programs they develop.
2. Plagiarism and academic integrity.
3. Responsibility and liability of programmers regarding code they develop for solutions. (See also: SEC-Foundations)
4. Basic professional work ethics of programmers.

Illustrative Learning Outcomes:

CS Core:

1. Explain/understand some of the intellectual property issues relating to programs.
2. Explain/understand when code developed by others can be used and proper ways of disclosing their use.
3. Explain/understand the responsibility of programmers when developing code for an overall solution (which may be developed by a team).
4. Explain/understand one or more codes of conduct applicable to programmers.

Professional Dispositions

- **Self-Directed:** Students must seek out solutions to issues on their own (e.g., using technical forums, FAQs, discussions). Resolving issues is an important part of becoming proficient in programming.
- **Experimental:** Students must experiment with language features to understand them and to quickly prototype solutions. This helps in learning about programming language features.
- **Technical curiosity:** Students must develop interest in understanding how programs are executed, how programs and data are stored in memory, etc. This will help build better mental models of the underlying execution system on which programs run.
- **Adaptable:** Students must be willing to learn and use different tools and technologies that facilitate software development. Tools are commonly used while programming and new tools often emerge – using tools effectively and learning the use of new tools will help.
- **Persistent:** Students must continue efforts until, for example, a bug is identified, a program is made robust and handles all situations, etc. This will help as programming requires effort and ability to persevere till a program works satisfactorily.
- **Meticulous:** Students must pay attention to detail and use orderly processes while programming. The underlying machine is unforgiving and there is no room for even small errors in the programs as they can cause major failures.

Mathematics Requirements

As SDF focuses on the first year and is foundational, it assumes only basic mathematical knowledge that students acquire in school, in particular Sets, Relations, Functions, and Logic. (See also: MSF-Discrete)

Course Packaging Suggestions

The SDF KA will generally be covered in introductory courses, often called CS1 and CS2. How much of the SDF KA can be covered in CS1 and how much is to be left for CS2 is likely to depend on the choice of programming language for CS1. For languages like Python or Java, CS1 can cover all the Programming Concepts and Development Methods KAs, and some of the Data Structures KA. It is desirable that they be further strengthened in CS2. The topics under algorithms KA and some topics under data structures KA can be covered in CS2. In case CS1 uses a language with fewer in-built data structures, then much of the Data Structures KA and some aspects of the programming KA may also need to be covered in CS2. With the former approach, the introductory course in programming can include the following:

1. SDF-Fundamentals (20 hours)
2. SDF-Data-Structures (12 hours)
3. SDF-Algorithms (6 hours)
4. SDF-Practices (5 hours)
5. SDF-SEP

Prerequisites: High school mathematics, specifically Sets, Relations, Functions, and Logic. (See also: MSF-Discrete)

Course objectives: At the end of the course, students should be able to:

- Design, code, test, and debug a modest sized program that effectively uses functional abstraction.
- Select and use the appropriate language-provided data structure for a given problem (like arrays, tuples/records/structs, lists, stacks, queues, and associative data types like sets, dictionaries/maps.)
- Design, code, test, and debug a modest-sized object-oriented program using classes and objects.
- Design, code, test, and debug a modest-sized program that uses language provided libraries and frameworks (including accessing data from the web through APIs).
- Read and explain given code including tracing the flow of control during execution.
- Write specifications of a program or a module in natural language explaining what it does.
- Build, execute and debug programs using a modern IDE and associated tools such as visual debuggers.
- Explain the key concepts relating to programming like parameter passing, recursion, runtime exceptions and exception handling.

Committee

Chair: Pankaj Jalote, Chair, IIIT-Delhi, Delhi, India

Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Titus Winters, Google, New York City, NY, USA
- Andrew Luxton-Reilly, University of Auckland, Auckland, New Zealand
- Christian Servin, El Paso Community College, El Paso, TX, USA
- Karen Reid, University of Toronto, Toronto, Canada
- Adrienne Decker, University at Buffalo, Buffalo, NY, USA

Software Engineering (SE)

Preamble

As far back as the early 1970s, British computer scientist Brian Randell allegedly said, “Software engineering is the multi-person construction of multi-version programs.” This is an essential insight: while programming is the skill that governs our ability to write a program, software engineering is distinct in two dimensions: time and people.

First, a software engineering project is a team endeavor; being a solitary programming expert is insufficient. Skilled software engineers must demonstrate expertise in communication and collaboration. Programming may be an individual activity, but software engineering is a collaborative one, deeply tied to issues of professionalism, teamwork, and communication.

Second, a software engineering project is usually “multi-version.” It has an expected lifespan; it needs to function properly for months, years, or decades. Features may be added or removed to meet product requirements. The engineering team itself will likely change. The technological context will shift, as our computing platforms evolve, programming languages change, dependencies upgrade, etc. This exposure to matters of time and change is novel when compared to a programming project: it isn’t enough to build a thing that works, instead it must work and stay working. Many of the most challenging topics in tech share “time will lead to change” as a root cause: backward compatibility, version skew, dependency management, schema changes, protocol evolution.

Software engineering presents a particularly difficult challenge for learning in an academic setting. Given that the major differences between programming and software engineering are time and teamwork, it is hard to generate lessons that *require* successful teamwork and that faithfully present the challenges of time. Additionally, some topics in software engineering will be more authentic and more relevant if our learners experience collaborative and long-term software engineering projects *in vivo* rather than in the classroom. Regardless of whether that happens as an internship, involvement in an open-source project, or full-time engineering role, a month of full-time hands-on experience has more available hours than the average software engineering course.

Thus, a software engineering curriculum must focus on concepts needed by most new-graduate hires, and that either are novel for those who are trained primarily as programmers, or that are abstract concepts that may not get explicitly stated/shared on the job. Such topics include, but are not limited to:

- Testing
- Teamwork, collaboration
- Communication
- Design
- Maintenance and evolution
- Software engineering tools

Some such material is reasonably suited to a standard lecture or lecture + lab course. Discussing theoretical underpinnings of version control systems, or branching strategies in such systems, can be an effective way to familiarize students with those ideas. Similarly, a theoretical discussion can highlight

the difference between static and dynamic analysis tools or may motivate discussion of diamond dependency problems in dependency networks.

On the other hand, many of the fundamental topics of software engineering are best experienced in a hands-on fashion. Historically, project-oriented courses have been a common vehicle for such learning. We believe that such experience is valuable but also bears some interesting risks: students may form erroneous notions about the difficulty/complexity of collaboration if their only exposure is a single project with teams formed of other novice software engineers. It falls to instructors to decide on the right balance between theoretical material and hands-on projects – neither is a perfect vehicle for this challenging material. We strongly encourage instructors of project courses to aim for iteration and fast feedback – a few simple tasks repeated, as in an Agile-structured project, is better than singular high-friction introductions to many types of tasks. Projects with real-world industry partners and clients are also especially encouraged. If long-running project courses are not an option, anything that can expose learners to the collaborative and long-term aspects of software engineering is valuable – adding features to an existing codebase, collaborating on distinct parts of a larger whole, pairing up to write an encoder and decoder, etc.

All evidence suggests that the role of software in our society will continue to grow for the foreseeable future. Additionally, the era of “two programmers in a garage” seems to have drawn to a close. Most important software these days is a team effort, building on existing code and leveraging existing functionality. The study of software engineering skills is a deeply important counterpoint to the everyday experience of computing students – we *must* impress on them the reality that few software projects are managed by writing from scratch as a solo endeavor. Communication, teamwork, planning, testing, and tooling are far more important as our students move on from the classroom and make their mark on the wider world.

Although most CS graduates will go on to an industry position that requires this material, the CS Core topics presented here are of value regardless of whether graduates go on to industry or academia.

Changes since CS 2013

This document shifts the focus of the Software Engineering knowledge area in a few ways compared to the goals of CS2013. The common reason behind most of these changes is to focus on material that learners would not pick up elsewhere in the curriculum, and that will be relevant *immediately* upon graduation, rather than at some future point in their careers.

- More explicit focus on the software workflow (version control, testing, code review, tooling).
- Less focus on team *leadership* and project management.
- More focus on team *participation*, communication, and collaboration.

Overview

1. **SE-Teamwork:** Because of the nature of learning programming, most students in introductory SE have little or no exposure to the collaborative nature of SE. Practice (for instance in project work) may help, but lecture and discussion time spent on the value of clear, effective, and efficient communication and collaboration is essential for Software Engineering.

2. **SE-Tools:** Industry reliance on SE tools has exploded in the past generation, with version control becoming ubiquitous, testing frameworks growing in popularity, increased reliance on static and dynamic analysis in practice, and the near-ubiquitous use of continuous integration systems. Increasingly powerful IDEs provide code searching and indexing capabilities, as well as small scale refactoring tools and integration with other SE tools. An understanding of the nature of these tools is broadly valuable - especially version control systems.
3. **SE-Requirements:** Knowing how to build something is of little help if we do not know what to build. Product Requirements (aka Requirements Engineering, Product Design, Product Requirements solicitation, Product Requirements Documents, etc.) introduces students to the processes surrounding the specification of the broad requirements governing development of a new product or feature.
4. **SE-Design:** While Product Requirements focus on the user-facing functionality of a software system, Software Design focuses on the engineer-facing design of internal software components. This encompasses large design concerns such as software architecture, as well as small-scale design choices like API design.
5. **SE-Construction:** Software Construction focuses on practices that influence the direct production of software: use of tests, test driven development, coding style. More advanced topics extend into secure coding, dependency injection, work prioritization, etc.
6. **SE-Validation:** Software Verification and Validation focuses on how to improve the value of testing – understand the role of testing, failure modes, and differences between good tests and poor ones.
7. **SE-Refactoring:** Refactoring and Code Evolution focuses on refactoring and maintenance strategies, incorporating code health, use of tools, and backwards compatibility considerations.
8. **SE-Reliability:** Software Reliability aims to improve understanding of and attention to error cases, failure modes, redundancy, and reasoning about fault tolerance.
9. **SE-Formal:** Formal Methods provides mathematically rigorous mechanisms to apply to software, from specification to verification. (Prerequisites: Substantial dependence on core material from the Discrete Structures area, particularly knowledge units DS/Basic Logic and DS/Proof Techniques.)

Core Hours

Knowledge Unit	CS Core	KA Core
Teamwork	2 + 3 (SEP)	2
Tools and Environments	1	3 + 1 (SDF)
Product Requirements	0 + 3 (SEP)	2
Software Design	1	4 + 2 (DM)
Software Construction	1 + 3 (SDF)	3 + 1 (SDF)
Software Verification and Validation	1	3

Refactoring and Code Evolution		2
Software Reliability		2
Formal Methods		
Total	**6**	**21**

Note: We have specifically highlighted Teamwork and Product Requirements as two knowledge units where SEP lessons are most directly obvious and applicable. Issues like impact on society, interaction with others, and social power disparities are pervasive in Software Engineering and should be woven into as many practical lessons as possible.

Knowledge Units

SE-Teamwork: Teamwork

CS Core:

1. Effective communication, including oral and written, as well as formal (email, docs, comments, presentations) and informal (team chat, meetings). (See also: SEP-Communication)
2. Common causes of team conflict, and approaches for conflict resolution.
3. Cooperative programming:
    a. Pair programming or Swarming
    b. Code review
    c. Collaboration through version control
4. Roles and responsibilities in a software team: (See also: SEP-Professional-Ethics)
    a. Advantages of teamwork
    b. Risks and complexity of such collaboration
5. Team processes – responsibilities for tasks, effort estimation, meeting structure, work schedule
6. Importance of team diversity and inclusivity. (See also: SEP-Communication)

KA Core:

7. Interfacing with stakeholders, as a team:
    a. Management & other non-technical teams
    b. Customers
    c. Users
8. Risks associated with physical, distributed, hybrid, and virtual teams – including communication, perception, structure, points of failure, mitigation, and recovery, etc.

Illustrative Learning Outcomes:

CS Core:

1. Follow effective team communication practices.
2. Articulate the sources of, hazards of, and potential benefits of team conflict – especially focusing on the value of disagreeing about ideas or proposals without insulting people.

3. Facilitate a conflict-resolution and problem-solving strategy in a team setting.
4. Collaborate effectively in cooperative development/programming.
5. Propose and delegate necessary roles and responsibilities in a software development team.
6. Compose and follow an agenda for a team meeting.
7. Facilitate through involvement in a team project, the central elements of team building, establishing healthy team culture, and team management including creating and executing a team work plan.
8. Promote the importance of and benefits that diversity and inclusivity brings to a software development team.

KA Core:

9. Reference, as a team, the importance of, and strategies to interface with stakeholders outside the team on both technical and non-technical levels.
10. Enumerate the risks associated with physical, distributed, hybrid, and virtual teams and possible points of failure and how to mitigate against and recover/learn from failures.

SE-Tools: Tools and Environments

CS Core:

1. Software configuration management and version control: (See also: SDF-Practices)
   a. Configuration in version control, reproducible builds/configuration.
   b. Version control branching strategies. Development branches vs release branches. Trunk-based development.
   c. Merging/rebasing strategies, when relevant.

KA Core:

2. Release management.
3. Testing tools including static and dynamic analysis tools. (See also: SDF-Practices, SEC-Coding)
4. Software process automation:
   a. Build systems – the value of fast, hermetic, reproducible builds, compare/contrast approaches to building a project.
   b. Continuous Integration (CI) – the use of automation and automated tests to do preliminary validation that the current head/trunk revision builds and passes (basic) tests.
   c. Dependency management – updating external/upstream dependencies, package management, SemVer.
5. Design and communication tools (docs, diagrams, common forms of design diagrams like UML).
6. Tool integration concepts and mechanisms. (See also: SDF-Practices)
7. Use of modern IDE facilities – debugging, refactoring, searching/indexing, ML-powered code assistants, etc. (See also: SDF-Practices)

Illustrative Learning Outcomes:

CS Core:

1. Describe the difference between centralized and distributed software configuration management.
2. Describe how version control can be used to help manage software release management.
3. Identify configuration items and use a source code control tool in a small team-based project.

KA Core:

4. Describe how available static and dynamic test tools can be integrated into the software development environment.
5. Understand the use of CI systems as a ground-truth for the state of the team's shared code (build and test success).
6. Describe the issues that are important in selecting a set of tools for the development of a specific software system, including tools for requirements tracking, design modeling, implementation, build automation, and testing.
7. Demonstrate the capability to use software tools in support of the development of a software product of medium size.

SE-Requirements: Product Requirements

KA Core:

1. Describe functional requirements using, for example, use cases or user stories.
    a. Using at least one method of documenting and structuring functional requirements.
    b. Understanding how the method supports design and implementation.
    c. Strengths and weaknesses of using a specific approach.
2. Properties of requirements including consistency, validity, completeness, and feasibility.
3. Requirements elicitation.
    a. Sources of requirements, for example, users, administrators, or support personnel.
    b. Methods of requirement gathering, for example, surveys, interviews, or behavioral analysis.
4. Non-functional requirements, for example, security, usability, or performance, also called as Quality Attributes. (See also: SEP-Sustainability)
5. Risk identification and management, including ethical considerations surrounding the proposed product. (See also: SEP-Professional-Ethics)
6. Communicating and/or formalizing requirement specifications.

Non-core:

7. Prototyping a tool for both eliciting and validating/confirming requirements.
8. Product evolution: when requirements change, how to understand what effect that has and what changes need to be made.
9. Effort estimation:
    a. Learning techniques for better estimating the effort required to complete a task;
    b. Practicing estimation and comparing it to how long tasks take;
    c. Effort estimation is quite difficult, so students are likely to be way off in many cases, but seeing the process play out with their own work is valuable.

Illustrative Learning Outcomes:

KA Core:

1. Compare different methods of eliciting requirements along multiple axes.
2. Identify differences between two methods of describing functional requirements (e.g., customer interviews, user studies) and the situations where each would be preferred.
3. Identify which behaviors are required, allowed, or barred from a given set of requirements and a list of candidate behaviors.

4. Collect a set of requirements for a simple software system.
5. Identify areas of a software system that need to be changed, given a description of the system and a set of new requirements to be implemented.
6. Identify the functional and non-functional requirements in a set of requirements.

Non-core:

7. Create a prototype of a software system to validate a set of requirements – building a mock-up, MVP, etc.
8. Estimate the time to complete a set of tasks, then compare estimates to the actual time taken.
9. Determine an implementation sequence for a set of tasks, adhering to dependencies between them, with a goal to retire risk as early as possible.
10. Write a requirement specification for a simple software system.

SE-Design: Software Design

CS Core:

1. System design principles. (See also: SF-Reliability)
    a. Levels of abstraction (e.g., architectural design and detailed design)
    b. Separation of concerns
    c. Information hiding
    d. Coupling and cohesion
2. Software architecture. (See also: SF-Reliability)
    a. Design paradigms
        i. Top-down functional decomposition/layered design
        ii. Data-oriented architecture
        iii. Object-oriented analysis and design
        iv. Event-driven design
    b. Standard architectures (e.g., client-server and microservice architectures including REST discussions, n-layer, pipes-and-filters, Model View Controller)
    c. Identifying component boundaries and dependencies
3. Programming in the large vs programming in the small. (See also: SF-Reliability)
4. Code smells and other indications of code quality, distinct from correctness. (See also: SEC-Engineering)

KA Core:

5. API design principles
    a. Consistency
        i. Consistent APIs are easier to learn and less error-prone
        ii. Consistency is both internal (between different portions of the API) and external (following common API patterns)
    b. Composability
    c. Documenting contracts
        i. API operations should describe their effect on the system, but not generally their implementation
        ii. Preconditions, postconditions, and invariants

    d. Expandability
    e. Error reporting
        i. Errors should be clear, predictable, and actionable
        ii. Input that does not match the contract should produce an error
        iii. Errors that can be reliably managed without reporting should be managed
6. Identifying and codifying data invariants and time invariants
7. Structural and behavioral models of software designs
8. Data design (See also: DM-Modeling)
    a. Data structures
    b. Storage systems
9. Requirement traceability
    a. Understanding which requirements are satisfied by a design

Non-Core:

10. Design modeling, for instance with class diagrams, entity relationship diagrams, or sequence diagrams
11. Measurement and analysis of design quality
12. Principles of secure design and coding (See also: SEC-Engineering)
    a. Principle of least privilege
    b. Principle of fail-safe defaults
    c. Principle of psychological acceptability
13. Evaluating design tradeoffs (e.g., efficiency vs reliability, security vs usability)

Illustrative Learning Outcomes:

CS Core:

1. Identify the standard software architecture of a given high-level design.
2. Select and use an appropriate design paradigm to design a simple software system and explain how system design principles have been applied in this design.
3. Adapt a flawed system design to better follow principles such as separation of concerns or information hiding.
4. Identify the dependencies among a set of software components in an architectural design.

KA Core:

5. Design an API for a single component of a large software system, including identifying and documenting each operation's invariants, contract, and error conditions.
6. Evaluate an API description in terms of consistency, composability, and expandability.
7. Expand an existing design to include a new piece of functionality.
8. Design a set of data structures to implement a provided API surface.
9. Identify which requirements are satisfied by a provided software design.

Non-Core:

10. Translate a natural language software design into class diagrams.
11. Adapt a flawed system design to better follow the principles of least privilege and fail-safe defaults.
12. Contrast two software designs across different qualities, such as efficiency or usability.

SE-Construction: Software Construction

CS Core:

1. Practical small-scale testing (See also: SDF-Practices)
    a. Unit testing
    b. Test-driven development – This is particularly valuable for students psychologically, as it is far easier to engage constructively with the challenge of identifying challenging inputs for a given API (edge cases, corner cases) a priori. If they implement first, the instinct is often to avoid trying to crash their new creation, while a test-first approach gives them the intellectual satisfaction of spotting the problem cases and then watching as more tests pass during the development process.
2. Documentation (See also: SDF-Practices)
    a. Interface documentation – describe interface requirements, potentially including (formal or informal) contracts, pre and post conditions, invariants.
    b. Implementation documentation should focus on tricky and non-obvious pieces of code, whether because the code is using advanced language features, or the behavior of the code is complex. (Do not add comments that re-state common/obvious operations and simple language features.)
        i. Clarify dataflow, computation, etc., focusing on what the code is.
        ii. Identify subtle/tricky pieces of code and refactor to be self-explanatory if possible or provide appropriate comments to clarify.

KA Core:

3. Coding style (See also: SDF-Practices)
    a. Style guides
    b. Commenting
    c. Naming
4. "Best Practices" for coding: techniques, idioms/patterns, mechanisms for building quality programs (See also: SEC-Coding, SDF-Practices)
    a. Defensive coding practices
    b. Secure coding practices and principles
    c. Using exception handling mechanisms to make programs more robust, fault-tolerant
5. Debugging (See also: SDF-Practices)
6. Logging
7. Use of libraries and frameworks developed by others (See also: SDF-Practices)

Non-Core:

8. Larger-scale testing
    a. Test doubles (stubs, mocks, fakes)
    b. Dependency injection
9. Work sequencing, including dependency identification, milestones, and risk retirement
    a. Dependency identification: Identifying the dependencies between different tasks
    b. Milestones: A collection of tasks that serve as a marker of progress when completed. Ideally, the milestone encompasses a useful unit of functionality.
    c. Risk retirement: Identifying what elements of a project are risky and prioritizing completing tasks that address those risks.
10. Potential security problems in programs (See also: SEC-Coding)

   a. Buffer and other types of overflows
   b. Race conditions
   c. Improper initialization, including choice of privileges
   d. Input validation
11. Documentation (autogenerated)
12. Development context: “green field” vs existing code base
   a. Change impact analysis
   b. Change actualization
13. Release management
14. DevOps practices

Illustrative Learning Outcomes:

CS Core:

1. Write appropriate unit tests for a small component (several functions, a single type, etc.).
2. Write appropriate interface and (if needed) implementation comments for a small component.

KA Core:

3. Describe techniques, coding idioms and mechanisms for implementing designs to achieve desired properties such as reliability, efficiency, and robustness.
4. Write robust code using exception handling mechanisms.
5. Describe secure coding and defensive coding practices.
6. Select and use a defined coding standard in a small software project.
7. Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration.
8. Describe the process of analyzing and implementing changes to code base developed for a specific project.
9. Describe the process of analyzing and implementing changes to a large existing code base.

Non-Core:

10. Rewrite a simple program to remove common vulnerabilities, such as buffer overflows, integer overflows and race conditions.
11. Write a software component that performs some non-trivial task and is resilient to input and run-time errors.

SE-Validation: Software Verification and Validation

CS Core:

1. Verification and validation concepts
   a. Verification: Are we building the thing right?
   b. Validation: Did we build the right thing?
2. Why testing matters: Does the component remain functional as the code evolves?
3. Testing objectives
   a. Usability
   b. Reliability
   c. Conformance to specification

    d. Performance
    e. Security
4. Test kinds
    a. Unit
    b. Integration
    c. Validation
    d. System
5. Stylistic differences between tests and production code: DAMP vs DRY – more duplication is warranted in test code.

KA Core:

6. Test planning and generation
    a. Test case generation, from formal models, specifications, etc.
    b. Test coverage
        i. Test matrices
        ii. Code coverage – how much of the code is tested?
        iii. Environment coverage – how many hardware architectures, operating systems, browsers, etc. are tested?
    c. Test data and inputs
7. Test development
    a. Test-driven development
    b. Object oriented testing, mocking, and dependency injection
    c. Opaque-box (previously, black-box) and transparent-box (previously, white-box) testing techniques
    d. Test tooling, including code coverage, static analysis, and fuzzing
8. Verification and validation in the development cycle
    a. Code reviews
    b. Test automation, including automation of tooling
    c. Pre-commit and post-commit testing
    d. Tradeoffs between test coverage and throughput/latency of testing
    e. Defect tracking and prioritization: reproducibility of reported defects
9. Domain specific verification and validation challenges
    a. Performance testing and benchmarking
    b. Asynchrony, parallelism, and concurrency
    c. Safety-critical
    d. Numeric

Non-Core:

10. Verification and validation tooling and automation
    a. Static analysis
    b. Code coverage
    c. Fuzzing
    d. Dynamic analysis and fault containment (sanitizers, etc.)
    e. Fault logging and fault tracking
11. Test planning and generation

a. Fault estimation and testing termination including defect seeding
b. Use of random and pseudo random numbers in testing

12. Performance testing and benchmarking
    a. Throughput and latency
    b. Degradation under load (stress testing, FIFO vs LIFO handling of requests)
    c. Speedup and scaling
        i. Amdahl's law
        ii. Gustafson's law
        iii. Soft and weak scaling
    d. Identifying and measuring figures of merits
    e. Common performance bottlenecks
        i. Compute-bound
        ii. Memory-bandwidth bound
        iii. Latency-bound
    f. Statistical methods and best practices for benchmarking
        i. Estimation of uncertainty
        ii. Confidence intervals
    g. Analysis and presentation (graphs, etc.)
    h. Timing techniques
13. Testing asynchronous, parallel, and concurrent systems
14. Verification and validation of non-code artifacts (documentation, training materials)

Illustrative Learning Outcomes:

CS Core:

1. Explain why testing is important.
2. Distinguish between program validation and verification.
3. Describe different objectives of testing.
4. Compare and contrast the different types and levels of testing (regression, unit, integration, systems, and acceptance).

KA Core:

5. Describe techniques for creating a test plan and generating test cases.
6. Create a test plan for a medium-size code segment which includes a test matrix and generation of test data and inputs.
7. Implement a test plan for a medium-size code segment.
8. Identify the fundamental principles of test-driven development methods and explain the role of automated testing in these methods.
9. Discuss issues involving the testing of object-oriented software.
10. Describe mocking and dependency injection and their application.
11. Undertake, as part of a team activity, a code review of a medium-size code segment.
12. Describe the role that tools can play in the validation of software.
13. Automate the testing in a small software project.
14. Explain the roles, pros, and cons of pre-commit and post-commit testing.
15. Discuss the tradeoffs between test coverage and test throughput/latency and how this can impact verification.

16. Use a defect tracking tool to manage software defects in a small software project.
17. Discuss the limitations of testing in certain domains.

Non-Core:

18. Describe and compare different tools for verification and validation.
19. Automate the use of different tools in a small software project.
20. Explain how and when random numbers should be used in testing.
21. Describe approaches for fault estimation.
22. Estimate the number of faults in a small software application based on fault density and fault seeding.
23. Describe throughput and latency and provide examples of each.
24. Explain speedup and the different forms of scaling and how they are computed.
25. Describe common performance bottlenecks.
26. Describe statistical methods and best practices for benchmarking software.
27. Explain techniques for and challenges with measuring time when constructing a benchmark.
28. Identify the figures of merit, construct and run a benchmark, and statistically analyze and visualize the results for a small software project.
29. Describe techniques and issues with testing asynchronous, concurrent, and parallel software.
30. Create a test plan for a medium-size code segment which contains asynchronous, concurrent, and/or parallel code, including a test matrix and generation of test data and inputs.
31. Describe techniques for the verification and validation of non-code artifacts.

SE-Refactoring: Refactoring and Code Evolution

KA Core:

1. Hyrum's Law/The Law of Implicit Interfaces
2. Backward compatibility
    a. Compatibility is not a property of a single entity, it's a property of a *relationship*.
    b. Backward compatibility needs to be evaluated in terms of provider + consumer(s) or with a well-specified model of what forms of compatibility a provider aspires to/promises.
3. Refactoring
    a. Standard refactoring patterns (rename, inline, outline, etc.)
    b. Use of refactoring tools in IDE
    c. Application of static-analysis tools (to identify code in need of refactoring, generate changes, etc.)
    d. Value of refactoring as a remedy for technical debt
4. Versioning
    a. Semantic Versioning (SemVer)
    b. Trunk-based development

Non-Core:

5. "Large Scale" Refactoring – techniques when a refactoring change is too large to commit safely (large projects), or when it is impossible to synchronize change between provider + all consumers (multiple repositories, consumers with private code).
    a. Express both old and new APIs so that they can co-exist.
    b. Minimize the size of *behavior* changes.

c. Why these techniques are required, (e.g., “API consumers I can see” vs “consumers I can’t see”).

Illustrative Learning Outcomes:

KA-Core:

1. Identify both explicit and implicit behavior of an interface and identify potential risks from Hyrum’s Law.
2. Consider inputs from static analysis tools and/or Software Design principles to identify code in need of refactoring.
3. Identify changes that can be broadly considered “backward compatible,” potentially with explicit statements about what usage is or is not supported.
4. Refactor the implementation of an interface to improve design, clarity, etc. with minimal/zero impact on existing users.
5. Evaluate whether a proposed change is sufficiently safe given the versioning methodology in use for a given project.

Non-Core:

6. Plan a complex multi-step refactoring to change default behavior of an API safely.

SE-Reliability: Software Reliability

KA Core:

1. Concept of reliability as probability of failure or mean time between failures, and faults as cause of failures
2. Identifying reliability requirements for different kinds of software
3. Software failures caused by defects/bugs, and so for high reliability the goal is to have minimum defects – by injecting fewer defects (better training, education, planning), and by removing most of the injected defects (testing, code review, etc.)
4. Software reliability, system reliability and failure behavior
5. Defect injection and removal cycle, and different approaches for defect removal
6. Compare the “error budget” approach to reliability with the “error-free” approach and identify domains where each is relevant.

Non-Core:

7. Software reliability models
8. Software fault tolerance techniques and models
    a. Contextual differences in fault tolerance (e.g., crashing a flight critical system is strongly avoided, crashing a data processing system before corrupt data is written to storage is highly valuable)
9. Software reliability engineering practices – including reviews, testing, practical model checking
10. Identification of dependent and independent failure domains, and their impact on system reliability
11. Measurement-based analysis of software reliability – telemetry, monitoring and alerting, dashboards, release qualification metrics, etc.

Illustrative Learning Outcomes:

KA Core:

1. Describe how to determine the level of reliability required by a software system.
2. Explain the problems that exist in achieving very high levels of reliability.
3. Understand approaches to minimizing faults that can be applied at each stage of the software lifecycle.

Non-Core:

4. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system.
5. Identify methods that will lead to the realization of a software architecture that achieves a specified level of reliability.
6. Identify ways to apply redundancy to achieve fault tolerance.
7. Identify single-point-of-failure (SPF) dependencies in a system design.

SE-Formal: Formal Methods

Non-Core:

1. Formal specification of interfaces
   a. Specification of pre- and post- conditions
   b. Formal languages for writing and analyzing pre- and post-conditions.
2. Problem areas well served by formal methods
   a. Lock-free programming, data races
   b. Asynchronous and distributed systems, deadlock, livelock, etc.
3. Comparison to other tools and techniques for defect detection
   a. Testing
   b. Fuzzing
4. Formal approaches to software modeling and analysis
   a. Model checkers
   b. Model finders

Illustrative Learning Outcomes:

1. Describe the role formal specification and analysis techniques can play in the development of complex software and compare their use as validation and verification techniques with testing.
2. Apply formal specification and analysis techniques to software designs and programs with low complexity.
3. Explain the potential benefits and drawbacks of using formal specification languages.

Professional Dispositions

- **Collaborative**: Software engineering is increasingly described as a “team sport” – successful software engineers are able to work with others effectively. Humility, respect, and trust underpin the collaborative relationships that are essential to success in this field.

- **Professional**: Software engineering produces technology that has the chance to influence literally billions of people. Awareness of our role in society, strong ethical behavior, and commitment to respectful day-to-day behavior outside of one's team are essential.
- **Communicative:** No single software engineer on a project is likely to know all the project details. Successful software projects depend on engineers communicating clearly and regularly to coordinate effectively.
- **Meticulous:** Software engineering requires attention to detail and consistent behavior from everyone on the team. Success in this field is clearly influenced by a meticulous approach - comprehensive understanding, proper procedures, and a solid avoidance of cutting corners.
- **Responsible:** The collaborative aspects of software engineering also highlight the value of being responsible. Failing to take responsibility, failing to follow through, and failing to keep others informed are all classic causes of team friction and bad project outcomes.

Mathematics Requirements

Desirable:

- Introductory statistics (performance comparisons, evaluating experiments, interpreting survey results, etc.). (See also CS-Core requirements for MSF-Statistics)

Course Packaging Suggestions

Advanced Course to include at least the following:

- SE-Teamwork (4 hours)
- SE-Tools (4 hours)
- SE-Requirements (2 hours)
- SE-Design (5 hours)
- SE-Construction (4 hours)
- SE-Validation (4 hours)
- SE-Refactoring (2 hours)
- SE-Reliability (2 hours)
- SEP-Professional-Ethics (7 hours)

Prerequisites:

- SDF-Fundamentals

Course objectives: Students should be able to perform good quality code review for colleagues (especially focusing on professional communication and teamwork needs), read and write unit tests, use basic software tools (IDEs, version control, static analysis tools) and perform basic activities expected of a new hire on a software team.

Committee

Chair: Titus Winters, Google, New York City, NY, USA

Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Adam Vartanian, Cord, London, UK
- Bryce Adelstein Lelbach, NVIDIA, New York City, NY, USA
- Patrick Servello, CIWRO, Norman, OK, USA
- Pankaj Jalote, IIIT-Delhi, Delhi, India
- Christian Servin, El Paso Community College, El Paso, TX, USA

Contributors:

- Hyrum Wright, Google, Pittsburgh, PA, USA
- Olivier Giroux, Apple, Cupertino, CA, USA
- Gennadiy Civil, Google, New York City, NY, USA

Security (SEC)

Preamble

Computing supports nearly every facet of modern critical infrastructure: transportation, communication, healthcare, education, energy generation and distribution, to name a few. With rampant attacks on and breaches of this infrastructure, computer science graduates have an important role in designing, implementing, and operating software systems that are robust, safe, and secure.

The Security (SEC) knowledge area focuses on developing a *security mindset* into the overall ethos of computer science graduates so that security is embedded in all their work products. Computer science students need to learn about system vulnerabilities and understand threats against computer systems. The *Security* title choice was intentional to serve as a one-word umbrella term for this knowledge area, which also includes concepts to support privacy, cryptography, secure systems, secure data, and secure code.

The SEC knowledge area relies on shared concepts pervasive in all the other areas of CS2023. It identifies seven crosscutting concepts of cybersecurity: *confidentiality*, *integrity*, *availability*, *risk assessment*, *systems thinking*, *adversarial thinking*, and *human-centered thinking*. The seventh concept, *human-centered thinking*, is additional to the six crosscutting concepts originally defined in the Cybersecurity Curricula 2017 (CSEC2017) [1]. This addition reinforces to students that humans are also a link in the overall chain of security, a theme that is also covered in knowledge areas such as HCI. Principles of protecting systems (also in the DM, OS, SDF, SE and SF knowledge areas) include security-by-design, privacy-by-design, defense-in-depth, and zero-trust.

Another concept is the notion of assurance, which is an attestation that security mechanisms need to comply with the security policies that have been defined for data, processes, and systems. Assurance is tied in with the concepts of verification and validation in the SE knowledge area. Considerations of data privacy and security are shared with the DM (technical aspects) and SEP knowledge areas.

The SEC knowledge area thus sits atop several of the other CS2023 knowledge areas, while including additional concepts that are not present in those knowledge areas. The specific dependence on other knowledge areas is stated below, starting with the Core Hours table. CS2023 treats security as a crucial component of the skillset of any CS graduate, and the hours needed for security preparation come from all the other 16 CS2023 knowledge areas.

Changes since CS2013

The Security knowledge area is an updated name for CS2013's Information Assurance and Security (IAS) knowledge area. Since 2013, Information Assurance and Security has been rebranded as Cybersecurity, which has become a new computing discipline, with its own curricular guidelines (CSEC 2017) developed by a Joint Task Force of the ACM, IEEE Computer Society, AIS and IFIP in 2017.

Moreover, since 2013, other curricular recommendations for cybersecurity beyond CS2013 and CSEC 2017 have been made. In the US, the National Security Agency recognizes institutions as Centers of

Academic Excellence (CAE) in Cyber Defense and/or Cyber Operations if their cybersecurity programs meet the respective CAE curriculum requirements. Additionally, the National Initiative for Cybersecurity Education (NICE) of the US National Institute for Standards and Technologies (NIST) has developed and revised the Workforce Framework for Cybersecurity (NICE Workforce Framework), which identifies competencies (knowledge and skills) needed to perform tasks relevant to cybersecurity work roles. The European Cybersecurity Skills Framework (ECSF) includes a standard ontology to describe cybersecurity tasks and roles, as well as addressing the cybersecurity personnel shortage in EU member countries. Similarities and differences of these cybersecurity guidelines, viewed from the CS perspective, also informed the SEC knowledge area.

Building on CS2013's recognition of the pervasiveness of security in computer science, the CS2023 SEC knowledge area focuses on ensuring that students develop a *security mindset* so that they are prepared for the continual changes occurring in computing. One useful addition is the knowledge unit for security analysis, design, and engineering to support the concepts of security-by-design and privacy-by-design.

The importance of computer science in ensuring the protection of future computing systems and societal critical infrastructure will continue to grow. Consequently, it is imperative that faculty teaching computer science incorporate the latest advances in security and privacy approaches to keep their curriculum current.

Differences between CS2023 Security knowledge area and Cybersecurity

CS2023's SEC knowledge area focuses on those aspects of security, privacy, and related concepts important for computer science students. In comparison, CSEC 2017 characterizes similarities and differences in the cybersecurity book of knowledge using the disciplinary lenses of computer science, computer engineering, software engineering, information systems, information technology, and other disciplines. In short, the major goal of the SEC knowledge area is to ensure that computer science graduates can design and develop more secure code, ensure data security and privacy, and apply a security mindset to their daily activities.

Protecting what happens *within* the perimeter of a networked computer system is a core competency of computer science graduates. Although the computer science and cybersecurity knowledge units overlap, the demands upon cybersecurity graduates typically are to protect the perimeter. CSEC 2017 defines cybersecurity as a highly interdisciplinary field of study that covers eight areas (data, software, component, connection, system, human, organizational, and societal security) and prepares its students for both technical and managerial roles in cybersecurity.

The first five CSEC 2017 areas are technical and have overlaps with the CS2023 SEC knowledge area, but the intent of coverage is substantively different as computer science students bring to bear the core competencies described in all the 17 CS2023 knowledge areas. For instance, consider the SEC knowledge area's Secure Coding knowledge unit. The computer science student will need to view this knowledge unit from a computer science lens, as an extension of the material covered in the SDF, SE, and PDC knowledge areas, while the Cybersecurity student will need to view software security in the overall context of diverse cybersecurity goals. These viewpoints are not totally distinct and have

overlaps, but the lenses used to examine and present the content are different. There are similar commonalities and differences among CS2023 SEC knowledge units and corresponding CSEC 2017 knowledge units.

Core Hours

Knowledge Unit	CS Core	KA Core
Foundational Security	1 + 7 (DM, FPL, PDC, SDF, SE, OS)	7
Society, Ethics, and the Profession	1 + 4 (SEP)	2
Secure Coding	2 + 6 (FPL, SDF, SE)	5
Cryptography	1 + 8 (MSF)	4
Security Analysis, Design, and Engineering	1 + 4 (MSF, SE)	8
Digital Forensics	0	6
Security Governance	0	3
Total hours	**6**	**35**

The SEC knowledge area requires approximately 28 hours of CS Core hours from the other knowledge areas, either to provide the basis or to complement its content. Of these, MSF-Discrete, MSF-Probability, and MSF-Statistics are likely to be relied upon extensively in all the SEC knowledge units, as are SDF-Fundamentals, SDF-Algorithms, and SDF-Practices. The others are mentioned within each of the SEC knowledge units described below.

Knowledge Units

SEC-Foundations: Foundational Security

CS Core:

1. Developing a security mindset incorporating crosscutting concepts: confidentiality, integrity, availability, risk assessment, systems thinking, adversarial thinking, human-centered thinking
2. Basic concepts of authentication and authorization/access control
3. Vulnerabilities, threats, attack surfaces, and attack vectors (See also: OS-Protection)
4. Denial of Service (DoS) and Distributed Denial of Service (DDoS) (See also: OS-Protection)
5. Principles and practices of protection, e.g., least privilege, open design, fail-safe defaults, defense in depth, and zero trust; and how they can be implemented (See also: OS-Principles, OS-Protection, SE-Construction, SEP-Security)

6. Optimization considerations between security, privacy, performance, and other design goals (See also: SDF-Practices, SE-Validation, HCI-Design)
7. Impact of AI on security and privacy: using AI to bolster defenses as well as address increased adversarial capabilities due to AI (See also: AI-SEP, HCI-Design, HCI-SEP)

KA Core:

8. Access control models (e.g., discretionary, mandatory, role-based, and attribute-based)
9. Security controls
10. Concepts of trust and trustworthiness
11. Applications of a security mindset: web, cloud, and mobile devices (See also: SF-System Design, SPD-Common)
12. Protecting embedded and cyber-physical systems (See also: SPD-Embedded)
13. Principles of usable security and human-centered computing (See also: HCI-Design, SEP-Security)
14. Security and trust in AI/machine learning systems, e.g., fit for purpose, ethical operating boundaries, authoritative knowledge sources, verified training data, repeatable system evaluation tests, system attestation, independent validation/certification; unintended consequences from: adverse effect (See also: AI-Introduction, AI-ML, AI-SEP, SEP-Security)
15. Security risks in building and operating AI/machine learning systems (e.g., algorithm bias, knowledge corpus bias, training corpus bias, copyright violation) (See also: AI-Introduction, AI-ML, AI-SEP)
16. Hardware considerations in security, e.g., principles of secure hardware, secure processor architectures, cryptographic acceleration, compartmentalization, software-hardware interaction (See also: AR-Assembly, AR-Representation, OS-Purpose)

Illustrative Learning Outcomes:

CS Core:

1. Evaluate a system for possible attacks that can be launched by an adversary.
2. Design and develop approaches to protect a system from a set of identified threats.

KA Core:

3. Describe how harm to user privacy can be avoided.
4. Develop a system that incorporates various principles of security and privacy.
5. Compare the different access control models in terms of functionality and performance.
6. Show how an adversary could use machine learning algorithms to reduce the security of a system.
7. Show how a developer could improve the security of a system using machine learning algorithms.
8. Describe hardware (especially CPU) vulnerabilities that can impact software.

SEC-SEP: Society, Ethics, and the Profession

CS Core:

1. Principles and practices of privacy (See also: SEP-Security)
2. Societal impacts on breakdowns in security and privacy (See also: SEP-Context, SEP-Privacy, SEP-Security)
3. Applicability of laws and regulations on security and privacy (See also: SEP-Security)
4. Professional ethical considerations when designing secure systems and maintaining privacy; ethical hacking (See also: SEP-Professional-Ethics, SEP-Privacy, SEP-Security)

***KA-Core*:**

5. Security by design (See also: SF-Security, SF-Design)
6. Privacy by design and privacy engineering (See also: SEP-Privacy, SEP-Security)
7. Security and privacy implications of malicious AI/machine learning actors, e.g., identifying deep fakes (See also: AI-Introduction, AI-ML, SEP-Privacy, SEP-Security)
8. Societal impacts of Internet of Things (IoT) devices and other emerging technologies on security and privacy (See also: SEP-Privacy, SEP-Security)

Illustrative Learning Outcomes:

CS Core:

1. Calculate the impact of a breakdown in security of a given system.
2. Construct a system that conforms to security laws.
3. Apply a set of privacy regulations to design a system that protects privacy.

KA Core:

4. Evaluate the legal ramifications of a system not corresponding to applicable laws and regulations.
5. Construct a system that is designed to avoid harm to user privacy.

SEC-Coding: Secure Coding

CS Core:

1. Common vulnerabilities and weaknesses
2. SQL injection and other injection attacks
3. Cross-site scripting techniques and mitigations
4. Input validation and data sanitization (See also: OS-Protection, SDF-Fundamentals, SE-Validation)
5. Type safety and type-safe languages (See also: FPL-Types, FPL-Systems, OS-Protection, SDF-Fundamentals, SE-Validation)
6. Buffer overflows, stack smashing, and integer overflows (See also: AR-Assembly, FPL-Systems, OS-Protection)
7. Security issues due to race conditions (See also: FPL-Parallel, PDC-Evaluation)

KA Core:

8. Principles of noninterference and nondeducibility
9. Preventing information flow attacks
10. Offensive security techniques as a defense
11. AI-assisted malware detection techniques
12. Ransomware: creation, prevention, and mitigation
13. Secure use of third-party components (See also: SE-Construction, SE-Validation)
14. Malware: varieties, creation, reverse engineering, and defense against them (See also: FPL-Systems, FPL-Translation)
15. Assurance: testing (including fuzzing and penetration testing), verification, and validation (See also: OS-Protection, SDF-Fundamentals, SE-Construction, SE-Validation)
16. Static and dynamic analyses (See also: FPL-Analysis, MSF-Protection, PDC-Evaluation, SE-Validation)
17. Secure compilers and secure code generation (See also: FPL-Runtime, FPL-Translation)

Illustrative Learning Outcomes:

CS Core:

1. Identify underlying problems in given examples of an enumeration of common weaknesses and explain how they can be circumvented.
2. Apply input validation and data sanitization techniques to enhance security of a program.
3. Describe how the selection of a programming language can impact the security of the system being constructed.
4. Rewrite a program in a type-safe language (e.g., Java or Rust) originally written in an unsafe programming language (e.g., C/C++).
5. Evaluate a program for possible buffer overflow attacks and rewrite to prevent such attacks.
6. Evaluate a set of related programs for possible race conditions and prevent an adversary from exploiting them.
7. Evaluate and prevent SQL injections attacks on a database application.
8. Evaluate and prevent cross-site scripting attacks against a website.

KA Core:

9. Describe different kinds of malicious software.
10. Construct a program that tests for all input handling errors.
11. Explain the risks of misusing interfaces with third-party code and how to correctly use third-party code.
12. Discuss the need to update software to fix security vulnerabilities and the lifecycle management of the fix.
13. Construct a system that is protected from unauthorized information flows.
14. Apply static and dynamic tools to identify programming faults.
15. Evaluate a system for the existence of malware and remove it.
16. Implement preventive techniques to reduce the occurrence of ransomware.

SEC-Crypto: Cryptography

CS Core:

1. Differences between algorithmic, applied, and mathematical views of cryptography
2. Mathematical preliminaries: modular arithmetic, Euclidean algorithm, probabilistic independence, linear algebra basics, number theory, finite fields, complexity, asymptotic analysis (See also: MSF-Discrete, MSF-Linear)
3. Basic cryptography: symmetric key and public key cryptography (See also: AL-Foundational, MSF-Discrete)
4. Basic cryptographic building blocks, including symmetric encryption, asymmetric encryption, hashing, and message authentication (See also: MSF-Discrete)
5. Classical cryptosystems, such as shift, substitution, transposition ciphers, code books, and machines (See also: MSF-Discrete)
6. Kerckhoff's principle and use of vetted libraries (See also: SE-Construction)
7. Usage of cryptography in real-world applications, e.g., electronic cash, secure channels between clients and servers, secure electronic mail, entity authentication, device pairing, steganography, and voting systems (See also: NC-Security, GIT-Image)

KA Core:

8. Additional mathematics: primality, factoring, and elliptic curve cryptography (See also: MSF-Discrete)

9. Private-key cryptosystems: substitution-permutation networks, linear cryptanalysis, differential cryptanalysis, DES, and AES (See also: MSF-Discrete, NC-Security)
10. Public-key cryptosystems: Diffie-Hellman and RSA (See also: MSF-Discrete)
11. Data integrity and authentication: hashing, and digital signatures (See also: MSF-Discrete, DM-Security)
12. Cryptographic protocols: challenge-response authentication, zero-knowledge protocols, commitment, oblivious transfer, secure two- or multi-party computation, hash functions, secret sharing, and applications (See also: MSF-Discrete)
13. Attacker capabilities: chosen-message attack (for signatures), birthday attacks, side channel attacks, and fault injection attacks (See also: NC-Security)
14. Quantum cryptography; Post Quantum/Quantum resistant cryptography (See also: AL-Foundational, MSF-Discrete)
15. Blockchain and cryptocurrencies (See also: MSF-Discrete, PDF-Communication)

Illustrative Learning Outcomes:

CS Core:

1. Explain the role of cryptography in supporting security and privacy.
2. Discuss the risks of inventing one's own cryptographic methods.
3. Discuss the importance of prime numbers in cryptography and explain their use in cryptographic algorithms.
4. Implement and cryptanalyze classical ciphers.

KA Core:

5. Describe how crypto keys can be managed securely.
6. Compare the space and time performance of a given set of cryptographic methods.
7. Discuss how modern private-key cryptosystems work and ways to cryptanalyze them.
8. Discuss how modern public-key cryptosystems work and ways to cryptanalyze them.
9. Compare different cryptographic algorithms in terms of security.
10. Explain key exchange protocols and show approaches to reduce their failure.
11. Describe real-world applications of cryptographic primitives and protocols.
12. Discuss how quantum cryptography works and the impact of quantum computing on cryptographic algorithms.

SEC-Engineering: Security Analysis, Design, and Engineering

CS Core:

1. Security engineering goals: building systems that remain dependable despite errors, accidents, or malicious adversaries (See also: SE-Construction, SE-Validation, SEP-Security)
2. Privacy engineering goals: building systems that design, implement, and deploy privacy features and controls (See also: SEP-Privacy)
3. Problem analysis and situational analysis to address system security (See also: SE-Validation)
4. Engineering tradeoff analysis based on time, cost, risk tolerance, risk acceptance, return on investment, and so on (See also: PDC-Evaluation, SE-Validation)

KA Core:

5. Security design and engineering, including functional requirements, security subsystems, information protection, security testing, security assessment, and evaluation (See also: PDC-Evaluation, SE-Requirements, SE-Validation)
6. Security analysis, covering security requirements analysis; security controls analysis; threat analysis; and vulnerability analysis (See also: FPL-Analysis, PDC-Evaluation)
7. Security attack domains and attack surfaces, e.g., communications and networking, hardware, physical, social engineering, software, and supply chain (See also: NC-Security)
8. Security attack modes, techniques, and tactics, e.g., authentication abuse; brute force; buffer manipulation; code injection; content insertion; denial of service; eavesdropping; function bypass; impersonation; integrity attack; interception; phishing; protocol analysis; privilege abuse; spoofing; and traffic injection (See also: NC-Security, OS-Protection, SE-Validation)
9. Attestation of software products with respect to their specification and adaptiveness (See also: SE-Requirements, SE-Validation)
10. Design and development of cyber-physical systems
11. Considerations for trustworthy computing, e.g., tamper resistant packaging, trusted boot, trusted kernel, hardware root of trust, software signing and verification, hardware-based cryptography, virtualization, and containers (See also: SE-Construction, SE-Validation)

Illustrative Learning Outcomes:

CS Core:

1. Create a threat model for a system or system design.
2. Apply situational analysis to develop secure solutions under a specified scenario.
3. Evaluate a given scenario for tradeoff analysis for system performance, risk assessment, and costs.

KA Core:

4. Design a set of technical security controls, countermeasures, and information protections to meet the security requirements and security objectives for a system.
5. Evaluate the effectiveness of security functions, technical controls, and componentry for a system.
6. Identify and mitigate security vulnerabilities and weaknesses in a system.
7. Evaluate and predict emergent behavior in areas such as Data Science, AI, and Machine Learning.

SEC-Forensics: Digital Forensics

KA Core:

1. Basic principles and methodologies for digital forensics
2. System design for forensics
3. Forensics in different situations: operating systems, file systems, application forensics, web forensics, network forensics, mobile device forensics, use of database auditing (See also: NC-Security)
4. Attacks on forensics and preventing such attacks
5. Incident handling processes
6. Rules of evidence – general concepts and differences between jurisdictions (See also: SEP-Security)
7. Legal issues: digital evidence protection and management, chains of custody, reporting, serving as an expert witness (See also: SEP-Security)

Illustrative Learning Outcomes:

KA Core:

1. Explain what a digital investigation is and how it can be implemented (See also: SEP-Security)
2. Design and implement software to support forensics.
3. Describe legal requirements for using seized data and its usage. (See also: SEP-Security)
4. Describe and implement an end-to-end chain of custody from initial digital evidence seizure to evidence disposal. (See also: SEP-Privacy, SEP-Security)
5. Extract data from a hard drive to comply with the law (See also: SEP-Security)
6. Discuss a person's professional responsibilities and liabilities when testifying as a forensics expert (See also: SEP-Professional-Ethics)
7. Recover data based on a given search term from an imaged system
8. Reconstruct data and events from an application history, or a web artifact, or a cloud database, or a mobile device. (See also: SPD-Mobile, SPD-Web)
9. Capture and analyze network traffic. (See also: NC-Security)
10. Develop approaches to address the challenges associated with mobile device forensics.
11. Apply forensics tools to investigate security breaches.
12. Identify and mitigate anti-forensic methods.

SEC-Governance: Security Governance

KA Core:

1. Protecting critical assets from threats
2. Security governance: organizational objectives and general risk assessment
3. Security management: achieve and maintain appropriate levels of confidentiality, integrity, availability, accountability, authenticity, and reliability (See also: SE-Validation)
4. Security policy: organizational policies, issue-specific policies, system-specific policies
5. Approaches to identifying and mitigating risks to computing infrastructure
6. Data lifecycle management policies: data collection, backups, and retention; cloud storage and services; breach disclosure (See also: DM-Security)

Illustrative Learning Outcomes:

KA Core:

1. Describe critical assets and how they can be protected.
2. Differentiate between security governance, management, and controls, giving examples of each.
3. Describe a technical control and implement it to mitigate specific threats.
4. Identify and assess risk of programs and database applications causing breaches.
5. Design and implement appropriate backup strategies conforming to a given policy.
6. Discuss a breach disclosure policy based on legal requirements and implement the policy.
7. Identify the risks and benefits of outsourcing to the cloud.

Professional Dispositions

- **Meticulous:** students need to pay careful attention to details to ensure the protection of real-world software systems.

- **Self-directed**: students must be ready to deal with the many novel and easily unforeseeable ways in which adversaries might launch attacks.
- **Collaborative**: students must be ready to collaborate with others, as collective knowledge and skills will be needed to prevent attacks, protect systems and data during attacks, and plan for the future after the immediate attack has been mitigated.
- **Responsible:** students need to show responsibility when designing, developing, deploying, and maintaining secure systems, as their enterprise and society is constantly at risk.
- **Accountable**: students need to know that as future professionals they will be held accountable if a system or data breach were to occur, which should strengthen their resolve to prevent such breaches from occurring in the first place.

Mathematics Requirements

Required:

- MSF-Discrete
- MSF-Probability
- MSF-Statistics

Desired:

- MSF-Linear

Course Packaging Suggestions

There are two suggestions for course packaging, along with an additional suggestion for a more advanced course.

The first suggestion for course packaging is to infuse the CS Core hours of the SEC KA into appropriate places in other coursework that covers related security topics in the following knowledge units. As the CS Core Hours of the SEC KA are only 6 hours, coursework covering one or more of the following knowledge units could accommodate them.

- AI-SEP
- AL-SEP
- AR-Assembly
- AR-Memory
- DM-Security
- FPL-Translation
- FPL-Run-Time
- FPL-Analysis
- FPL-Types
- HCI-Design
- HCI-Accountability
- HCI-SEP

- NC-Security
- OS-Protection
- PDC-Communication
- PDC-Coordination
- PDC-Evaluation
- SDF-Fundamentals
- SDF-Practices
- SE-Validation
- SEP-Privacy
- SEP-Security
- SF-Design
- SF-Security
- SPD-Common
- SPD-Mobile
- SPD-Web

The second approach for course packaging is to create an additional full course focused on security that packages the following, building on the topics already covered in other knowledge areas.

Fundamentals of Computer Security:

- SEC-Foundations (6 hours)
- SEC-SEP (4 hours)
- SEC-Coding (7 hours)
- SEC-Crypto (5 hours)
- SEC-Engineering (4 hours)
- SEC-Forensics (2 hours)
- SEC-Governance (1 hour)
- AI-SEP (1 hour)
- AR-Assembly (1 hour)
- AR-Memory (1 hour)
- DM-Security (3 hours)
- FPL-Translation (1 hour)
- FPL-Run-Time (1 hour)
- FPL-Analysis (1 hour)
- FPL-Types (2 hours)
- HCI-Design (1 hour)
- HCI-Accountability (1 hour)
- HCI-SEP (1 hour)
- NC-Security (2 hours)
- OS-Protection (1 hour)
- PDC-Communication (1 hour)
- PDC-Coordination (1 hour)
- PDC-Evaluation (1 hour)
- SDF-Fundamentals (1 hour)

- SDF-Practices (1 hour)
- SE-Validation: (2 hours)
- SEP-Privacy (1 hour)
- SEP-Security (2 hours)
- SF-Design (2 hours)
- SF-Security (2 hours)
- SPD-Common (2 hours)
- SPD-Mobile (2 hours)
- SPD-Web: Web Platforms (2 hours)

The coverage exceeds 45 lecture hours, and so, in a typical course, instructors would need to decide what topics to emphasize and what not to cover without losing the perspective that the course should help students develop a *security mindset*.

Prerequisites: Depends on the selected topics, but appropriate coursework covering MSF, SDF, and SE knowledge areas is needed.

Course objectives: Students should develop a security mindset and be ready to apply this mindset to securing data, software, systems, and applications.

A third suggested packaging is to create an advanced course that develops a security architect/engineer's view by including the following:

Security Engineering:

- SEC-Foundations (6 hours)
- SEC-SEP (4 hours)
- SEC-Coding (6 hours)
- SEC-Crypto (2 hours)
- SEC-Engineering (10 hours)
- SEC-Forensics (2 hours)
- SEC-Governance (1 hour)
- DM-Security (2 hours)
- NC-Security (3 hours)
- OS-Protection (2 hours)
- PDC-Evaluation (2 hours)
- SDF-Fundamentals (1 hour)
- SDF-Practices (1 hour)
- SE-Validation (2 hours)
- SEP-Privacy (1 hour)
- SEP-Security (1 hour)
- SF-Design (2 hours)
- SF-Security (2 hours)
- SPD-Mobile (2 hours)
- SPD-Web (2 hours)

The coverage for all topics is over 45 lecture hours, and so instructors would need to decide what topics to emphasize and what not to cover without losing the perspective that the course should help students develop the security engineer's mindset. Laboratory time related to data and network security, web platform, secure coding and validation would be valuable aspects of this course.

Prerequisites: Depends on the selected topics, either the first or second packaging suggested above would be recommended based on degree program needs.

Course objectives: Computer science students should develop the mindset of a security engineer and be ready to apply this mindset to problems in designing and evaluating the security of a range of computing systems and information services.

Committee

Chair: Rajendra K. Raj, Rochester Institute of Technology, Rochester, NY, USA

Members:

- Vijay Anand, University of Missouri – St. Louis, St. Louis, MO, USA
- Diana Burley, American University, Washington, DC, USA
- Sherif Hazem, Central Bank of Egypt, Cairo, Egypt
- Michele Maasberg, United States Naval Academy, Annapolis, MD, USA
- Bruce McMillin, Missouri University of Science and Technology, Rolla, MO, USA
- Sumita Mishra, Rochester Institute of Technology, Rochester, NY, USA
- Nicolas Sklavos, University of Patras, Patras, Greece
- Blair Taylor, Towson University, Towson, MD, USA
- Jim Whitmore, Dickinson College, Carlisle, PA, USA

Contributors:

- Markus Geissler, Cosumnes River College, Sacramento, CA, USA
- Michael Huang, Rider University, Lawrenceville, NJ, USA
- Tim Preuss, Minnesota State Community and Technical College, Moorhead, MN, USA
- Daniel Zappala, Brigham Young University, Provo, UT, USA

References

1. Joint Task Force on Cybersecurity Education. 2017. Cybersecurity Curricula 2017. ACM, IEEE-CS, AIS SIGSEC, and IFIP WG 11.8. https://doi.org/10.1145/3184594

Society, Ethics, and the Profession (SEP)

Preamble

The ACM Code of Ethics and Professional Conduct states: “Computing professionals' actions change the world. To act responsibly, they should reflect upon the wider impacts of their work, consistently supporting the public good.” The IEEE Code of Ethics starts by recognizing “the importance of our technologies in affecting the quality of life throughout the world.” The AAAI Code of Professional Ethics and Conduct begins with “Computing professionals, and in particular, AI professionals’ actions change the world. To act responsibly, they should reflect upon the wider impacts of their work, consistently supporting the public good.”

While technical issues dominate the computing curriculum, they do not constitute a complete educational program in the broader context. It is more evident today than ever that students must also be exposed to the larger societal context of computing to develop an understanding of the critical and relevant social, ethical, legal, and professional issues and responsibilities at hand. This need to incorporate the study of these non-technical issues into the ACM curriculum was formally recognized in 1991, as articulated in the following excerpt from CS1991 [1].

> *Undergraduates also need to understand the basic cultural, social, legal, and ethical issues inherent in the discipline of computing. They should understand where the discipline has been, where it is, and where it is heading. They should also understand their individual roles in this process, as well as appreciate the philosophical questions, technical problems, and aesthetic values that play an important part in the development of the discipline.*
>
> *Students also need to develop the ability to ask serious questions about the social impact of computing and to evaluate proposed answers to those questions. Future practitioners must be able to anticipate the impact of introducing a given product into a given environment. Will that product enhance or degrade the quality of life? What will the impact be upon individuals, groups, and institutions?*
>
> *Finally, students need to be aware of the basic legal rights of software and hardware vendors and users, and they also need to appreciate the ethical values that are the basis for those rights. Future practitioners must understand the responsibility that they will bear, and the possible consequences of failure. They must understand their own limitations as well as the limitations of their tools. All practitioners must make a long-term commitment to remaining current in their chosen specialties and in the discipline of computing as a whole.*

Nonetheless, in recent years myriad high-profile issues affecting society at large have occurred leading to the conclusion that computer science professionals are not as prepared as they should be.

As technological advances (more specifically, how these advances are used by humans) continue to significantly impact the way we live and work, the critical importance of social and ethical issues and professional practice continues to increase in magnitude and consequence. The ways we use

computing products and platforms, while hopefully providing opportunities, also introduce ever more challenging problems. A recent example is the emergence of generative AI, including large language models that generate code. A 2020 *Communications of the ACM* article [4] stated: "... because computing as a discipline is becoming progressively more entangled within the human and social lifeworld, computing as an academic discipline must move away from engineering-inspired curricular models and integrate the analytic lenses supplied by social science theories and methodologies."

In parallel to, and as part of, the heightened awareness of the social consequences computing has on the world, computing communities have become much more aware – and active – in areas of diversity, equity, inclusion, and accessibility. These feature in statements and initiatives at ACM [8], IEEE [9], and AAAI [10] and in their codes of conduct [1-3]. All students deserve an inclusive, diverse, equitable and accessible learning environment. Computing students also have a unique duty to ensure that when put to practice, their skills, knowledge, and competencies are applied in ways that work for, and not against, the principles of diversity, equity, inclusion, and accessibility. These principles are inherently a part of computing, and a new knowledge unit "Diversity, Equity, Inclusion and Accessibility" (SEP-DEIA) has been added to this knowledge area.

Computer science educators may opt to deliver the material in this knowledge area within the contexts of traditional technical and theoretical courses, in dedicated courses, and as part of capstone, project, and professional practice courses. The material in this knowledge area is best covered through a combination of all the above. It is too commonly held that many topics in this knowledge area may not readily lend themselves to being covered in other more traditional computer science courses. However, many of these topics naturally arise in traditional courses, or can be included with minimal effort. The benefits of exposing students to SEP topics within the context of those traditional courses are invaluable. Nonetheless institutional challenges will present barriers; for instance, some of these traditional courses may not be offered at a given institution and, in such cases, it is difficult to cover these topics appropriately without a dedicated SEP course. If social, ethical, and professional considerations are covered only in a dedicated course and not in the context of others, it could reinforce the false notion that technical processes are void of these important aspects, or that they are more isolated than they are. Because of the broad relevance of these knowledge units, it is important that as many traditional courses as possible include aspects such as case studies, that analyze ethical, legal, social, and professional considerations in the context of the technical subject matter of those courses. Courses in areas such as software engineering, databases, computer graphics, computer networks, information assurance & security, and introduction to computing, all provide obvious context for analysis of such issues. However, an ethics-related module could be developed for almost any program. It would be explicitly against the spirit of these recommendations to have *only* a dedicated course within a specific computer science curriculum without great practical reason. Further, these topics should be covered in courses starting from year 1. Presenting them as advanced topics in later courses only creates an artificial perception that SEP topics are only important at a certain level or complexity. While it is true that the importance and consequence of SEP topics *increases* with level and complexity, introductory topics are not devoid of SEP topics. Further, many SEP topics are *best* presented early to lay a foundation for more intricate topics later in the curriculum.

Running through all the topics in this knowledge area is the need to speak to the computing practitioner's responsibility to proactively address issues through both ethical and technical actions. Today it is important not only for the topics in this knowledge area, but for students' knowledge in general, that the ethical issues discussed in any course should be directly related to – and arise naturally from – the subject matter of that course. Examples include a discussion in a database course of the SEP aspects of data aggregation or data mining; or a discussion in a software engineering course of the potential conflicts between obligations to the customer and users as well as all others affected by their work. Computing faculty who are unfamiliar with the content and/or pedagogy of applied ethics are urged to take advantage of the considerable resources from ACM, IEEE-CS, AAAI, SIGCAS (ACM Special Interest Group on Computers and Society), and other organizations. Additionally, it is the educator's responsibility to impress upon students that this area is just as important – in ways more important – than technical areas. The societal, ethical, and professional knowledge gained in studying topics in this knowledge area will be used throughout one's career and are transferable between projects, jobs, and often even industries, particularly as one's career progresses into project leadership and management.

The ACM Code of Ethics and Professional Conduct [5], the IEEE Code of Ethics [6], and the AAAI Code of Professional Ethics and Conduct [7] provide guidance that serve as the basis for the conduct of all computing professionals in their work. The ACM Code emphasizes that ethical reasoning is not an algorithm to be followed, and computer professionals are expected to consider how their work impacts the public good as the primary consideration. It falls to computing educators to highlight the domain-specific role of these topics for our students, but computer science programs should certainly be willing to lean on complementary courses from the humanities and social sciences.

Most computing educators are not also moral philosophers. Yet CS2023, along with past CS curricular recommendations, indicate the need for ethical analysis. CS2023 and prior curricular recommendations are quite clear on the required mathematical foundations that students are expected to gain which are often delivered by mathematics departments. Yet, the same is not true of moral philosophy. No one would expect a student to be able to provide a proof by induction until after having successfully completed a course in discrete mathematics. Yet, the parallel with respect to ethical analyses is somehow absent. We seemingly do (often) expect our students to perform ethical analysis without having the appropriate prerequisite knowledge from philosophy. Further, the application of ethical analysis also underlies every other knowledge unit in this knowledge area. We acknowledge that the knowledge unit Methods for Ethical Analysis (SEP-Ethical-Analysis) is the only one in this knowledge area that does not readily lend itself to being taught in the context of other CS2023 knowledge areas. Suggestions in terms of addressing this appear in the Course Packaging Suggestions.

The lack of prerequisite training in social, ethical, and professional topics has facilitated graduates operating with a certain ethical egoism (e.g., 'Here is what I believe/think/feel is right'). Regardless of how well intentioned, one might conclude that this is what brought us to a point in history where there are frequent occurrences of unintended consequences of technology, serious data breaches, and software failures causing economic, emotional. and physical harm. Certainly, computing graduates who have learned how to apply the various ethical frameworks or lenses proposed through the ages would only serve to improve this situation. In retrospect, to ignore the lessons from moral philosophy, which

have been debated and refined for millennia – on what it means to act justly, or work for the common good – appears as hubris.

A computer science student must not graduate without understanding how society and ethics influence the computing profession. Nor should it be possible to complete a computer science degree without learning how computing professionals influence society, the ethical considerations involved in shaping that impact, and the student-turned-graduate's role in these relationships, as both computing professionals and members of society.

Changes Since CS2013

- The overall number of hours dedicated to this knowledge unit has doubled since CS2013 from 16 to 32 – CS2013 had 11 'Tier 1' hours and 5 'Tier 2' hours while CS2023 has 18 CS Core hours and 14 KA Core hours). However, many of these hours are best covered within the context of other knowledge areas as discussed in the introduction to CS2023. Additionally:
    - Several knowledge units that were 'elective only' in CS2013 now have CS Core and KA Core hours.
    - The number of hours in most knowledge units has increased.
- SEP has been re-titled from "Social Issues and Professional Practice" to "Society, Ethics, and the Profession." Our rationale follows.
    - Professional practice is an important part of being a professional. However, a solid understanding of the profession in which one is to become a professional is needed – indeed it is a prerequisite. Through this lens, professional conduct is part of a profession, along with its history, values, norms, etc. Additionally, "the profession" appears in the ACM tagline "Advancing Computing as a Science & Profession", and the ACM code [5]: "The ACM Code of Ethics and Professional Conduct …expresses the conscience of the profession." In a similar way the AAAI Code [7] states that it "...expresses the conscience of the AI profession", and the IEEE code [6] states: "in accepting a personal obligation to our profession, its members and the communities we serve, do hereby commit ourselves to the highest ethical and professional conduct." While studying computing, students should not only learn how to become professionals, but also learn about the profession in which they will become professionals.
- Inclusion of the Diversity, Equity, Inclusion and Accessibility (DEIA) knowledge unit.
- Changed titles of two knowledge units
    - Professional Communication -> Communication
    - Analytical Tools -> Methods for Ethical Analysis

Core Hours

Knowledge Unit	CS Core	KA Core
Social Context	3	2
Methods for Ethical Analysis	2	1

Professional Ethics	2	2
Intellectual Property	1	1
Privacy and Civil Liberties	2	1
Communication	2	1
Sustainability	1	1
History	1	1
Economies of Computing	0	1
Security Policies, Laws, and Computer Crimes	2	1
Diversity, Equity, Inclusion and Accessibility	2	2
Total	**18**	**14**

Knowledge Units

SEP-Context: Social Context

Computers, the internet, and artificial intelligence - perhaps more than any other technologies - have transformed society over the past several decades, with dramatic increases in human productivity; an explosion of options for news, entertainment, and communication; and fundamental breakthroughs in almost every branch of science and engineering. It is imperative to recognize that this is not a one-way street. Society also affects computing, resulting in a complex socio-technical context that is constantly changing, requiring the perspective of history to put the present as well as the possible future into appropriate perspective.

Social Context provides the foundation for all other knowledge units in SEP, particularly Professional Ethics.

CS Core:

1. Social implications (e.g., political and cultural ideologies) in a hyper-networked world where the capabilities and impact of social media, artificial intelligence, and computing in general are rapidly evolving.
2. Impact of computing applications (e.g., social media, artificial intelligence applications) on individual well-being, and safety of all kinds (e.g., physical, emotional, economic).
3. Consequences of involving computing technologies, particularly artificial intelligence, biometric technologies, and algorithmic decision-making systems, in civic life (e.g., facial recognition technology, biometric tags, resource distribution algorithms, policing software) and how human agency and oversight is crucial.

4. How deficits in diversity and accessibility in computing affect society and what steps can be taken to improve equity in computing.

KA Core:

5. Growth and control of the internet, data, computing, and artificial intelligence
6. Often referred to as the digital divide, differences in access to digital technology resources and its resulting ramifications for gender, class, ethnicity, geography, and/or developing countries, including consideration of responsibility to those who might be less wealthy, under threat, or who would struggle to have their voices heard.
7. Accessibility issues, including legal requirements such as Web Content Accessibility Guidelines (www.w3.org/TR/WCAG21)
8. Context-aware computing

Illustrative Learning Outcomes:

CS Core:

1. Describe the different ways that computer technology (networks, mobile computing, artificial intelligence) mediates social interaction at the personal and collective levels.
2. Identify developers' assumptions and values embedded in hardware and software design, especially as they pertain to usability for diverse populations including under-served and those with disabilities.
3. Interpret the social context of a given design and its implementation.
4. Analyze the efficacy of a given design and implementation using empirical data.
5. Understand the implications of technology use (e.g., social media) for different identities, cultures, and communities.

KA Core:

6. Describe the internet's role in facilitating communication between citizens, governments, and each other.
7. Analyze the effects of reliance on computing in the implementation of democracy (e.g., delivery of social services, electronic voting).
8. Describe the impact of a lack of appropriate representation of people from historically minoritized populations in the computing profession (e.g., industry culture, product diversity).
9. Discuss the implications of context awareness in ubiquitous computing systems.
10. Express how access to the internet and computing technologies affect different societies.
11. Identify why/how internet access can be viewed as a human right.

SEP-Ethical-Analysis: Methods for Ethical Analysis

Ethical theories and principles are the foundations of ethical analysis because they are the viewpoints which can provide guidance along the pathway to a decision. Each theory emphasizes different assumptions and methods for determining the ethicality of a given action. It is important for students to recognize that decisions in different contexts may require different ethical theories (including combinations) to arrive at ethically acceptable outcomes, and what constitutes 'acceptable' depends on

a variety of factors such as cultural context. Applying methods for ethical analysis requires both an understanding of the underlying principles and assumptions guiding a given tool and an awareness of the social context for that decision. Traditional ethical frameworks (e.g., [11]) as provided by western philosophy can be useful, but they are not all-inclusive. Effort must be taken to include decolonial, indigenous, and historically marginalized ethical perspectives whenever possible. No theory will be universally applicable to all contexts, nor is any single ethical framework the 'best.' Engagement across various ethical schools of thought is important for students to develop the critical thinking needed in judiciously applying methods for ethical analysis of a given situation.

CS Core:

1. Avoiding fallacies and misrepresentation in argumentation
2. Ethical theories and decision-making (philosophical and social frameworks, e.g. [1])
3. Recognition of the role culture plays in our understanding, adoption, design, and use of computing technology
4. Why ethics is important in computing, and how ethics is similar to, and different from, laws and social norms

KA Core:

5. Professional checklists
6. Evaluation rubrics
7. Stakeholder analysis
8. Standpoint theory
9. Introduction to ethical frameworks (e.g., consequentialism such as utilitarianism, non-consequentialism such as duty, rights, or justice, agent-centered such as virtue or feminism, contractarianism, ethics of care) and their use for analyzing an ethical dilemma

Illustrative Learning Outcomes:

CS Core:

1. Describe how a given cultural context impacts decision making.
2. Express the use of example and analogy in ethical argument.
3. Analyze (and avoid) basic logical fallacies in an argument.
4. Analyze an argument to identify premises and conclusion.
5. Evaluate how and why ethics is so important in computing and how it relates to cultural norms, values, and law.
6. Justify a decision made on ethical grounds.

KA Core:

7. Distinguish all stakeholder positions in relation to their cultural context in a given situation.
8. Analyze the potential for introducing or perpetuating ethical debt (deferred consideration of ethical impacts or implications) in technical decisions.
9. Discuss the advantages and disadvantages of traditional ethical frameworks.

10. Analyze ethical dilemmas related to the creation and use of technology from multiple perspectives using ethical frameworks.

SEP-Professional-Ethics: Professional Ethics

Computer ethics is a branch of practical philosophy that deals with how computing professionals should make decisions regarding professional and social conduct. There are three primary influences: 1) the individual's own personal ethical code, 2) any informal or formal regulation/decree/etc. of ethical behavior existing in the workplace, applicable licensures, certifications, or laws, and 3) exposure to formal codes of ethics and ethical frameworks.

CS Core:

1. Community values and the laws by which we live
2. The nature of being a professional including care, attention, discipline, fiduciary responsibility, and mentoring
3. Keeping up to date as a computing professional in terms of familiarity, tools, skills, legal and professional frameworks as well as the ability and responsibility to self-assess and progress in the computing field
4. Professional certification, codes of ethics, conduct, and practice, such as the ACM, IEEE, AAAI, and other international societies
5. Accountability, responsibility, and liability (e.g., software correctness, reliability and safety, warranty, negligence, strict liability, ethical approaches to security vulnerability disclosures) including whether a product/service should be built, not just doing so because it is technically possible.
6. Introduction to theories describing the human creation and use of technology including instrumentalism, sociology of technological systems, disability justice, neutrality thesis, pragmatism, and decolonial models, including developing and using technology to right wrongs and do good
7. Strategies for recognizing and reporting designs, systems, software, and professional conduct (or their outcomes) that may violate law or professional codes of ethics

KA Core:

8. The role of the computing professional and professional societies in public policy
9. Maintaining awareness of consequences
10. Ethical dissent and whistleblowing
11. The relationship between regional culture and ethical dilemmas
12. Dealing with harassment and discrimination
13. Forms of professional credentialing
14. Ergonomics and healthy computing environments
15. Time-to-market and cost considerations versus quality professional standards

Illustrative Learning Outcomes:

CS Core:

1. Identify ethical issues that arise in software design, development practices, and software deployment.
2. Discuss how to address ethical issues in specific situations.
3. Express the ethical responsibility of ensuring software correctness, reliability and safety including from where this responsibility arises (e.g., ACM/IEEE/AAAI Codes of Ethics, laws and regulations, organizational policies).
4. Describe the mechanisms that typically exist for a professional to keep up to date in ethical matters.
5. Describe the strengths and weaknesses of relevant professional codes as expressions of being a professional and guides to decision-making.
6. Analyze a global computing issue, observing the role of professionals and government officials in managing this problem.

KA Core:

7. Describe ways in which professionals and professional organizations may contribute to public policy.
8. Describe the consequences of inappropriate professional behavior.
9. Be familiar with whistleblowing and have access to knowledge to guide one through an incident.
10. Identify examples of how regional culture interplays with ethical dilemmas.
11. Describe forms of harassment and discrimination and avenues of assistance.
12. Assess various forms of professional credentialing.
13. State the relationship between ergonomics in computing environments and people's health.
14. Describe issues associated with industries' push to focus on time-to-market versus enforcing quality professional standards.

SEP-IP: Intellectual Property

Intellectual property refers to a range of intangible rights of ownership in any product of human intellect, such as a software program. Laws, which vary by locality, provide different methods for protecting these rights of ownership based on their type. Ideally, intellectual property laws balance the interests of creators and users of the property. There are four types of intellectual property rights relevant to software: patents, copyrights, trade secrets, and trademarks. Moreover, property rights are often protected by user licenses. Each affords a different type of legal protection.

CS Core:

1. Intellectual property rights
2. Intangible digital intellectual property (IDIP)
3. Legal foundations for intellectual property protection
4. Common software licenses (e.g., MIT, GPL and its variants, Apache, Mozilla, Creative Commons)
5. Plagiarism and authorship

KA Core:

6. Philosophical foundations of intellectual property

7. Forms of intellectual property (e.g., copyrights, patents, trade secrets, trademarks) and the rights they protect
8. Limitations on copyright protections, including fair use and the first sale doctrine
9. Intellectual property laws and treaties that impact the enforcement of copyrights
10. Software piracy and technical methods for enforcing intellectual property rights, such as digital rights management and closed source software as a trade secret
11. Moral and legal foundations of the open-source movement
12. Systems that use others' data (e.g., large language models)

Illustrative Learning Outcomes:

CS Core:

1. Describe and critique legislation and precedent aimed at digital copyright infringements.
2. Identify contemporary examples of intangible digital intellectual property.
3. Select an appropriate software license for a given project.
4. Defend legal and ethical uses of copyrighted materials.
5. Interpret the intent and implementation of software licensing.
6. Discuss whether a use of copyrighted material is likely to be fair use.
7. Analyze the ethical issues inherent in various plagiarism detection mechanisms.
8. Identify multiple forms of plagiarism beyond verbatim copying of text or software (e.g., intentional paraphrasing, authorship misrepresentation, and improper attribution).

KA Core:

9. Discuss the philosophical bases of intellectual property in an appropriate context (e.g., country).
10. Distinguish the conflicting issues involved in securing software patents.
11. Contrast the protections and obligations of copyright, patent, trade secret, and trademarks.
12. Describe the rationale for the legal protection of intellectual property in the appropriate context (e.g., country).
13. Analyze the use of copyrighted work under the concepts of fair use and the first sale doctrine.
14. Identify the goals of the open-source movement and its impact on fields beyond computing, such as the right-to-repair movement.
15. Summarize the global nature of software piracy.
16. Criticize the use of technical measures of digital rights management (e.g., encryption, watermarking, copy restrictions, and region lockouts) from multiple stakeholder perspectives.
17. Discuss the nature of anti-circumvention laws in the context of copyright protection.

SEP-Privacy: Privacy and Civil Liberties

Electronic information sharing highlights the need to balance privacy protections with information access. The ease of digital access to many types of data – in addition to copying and distributing these data – makes privacy rights and civil liberties more complex, especially given cultural and legal differences in these areas. Complicating matters further, privacy also has interpersonal, organizational, professional/business, and governance components. In addition, the interconnected nature of online communities raises challenges for managing expectations and protections for freedom of expression in

various cultures and nations. Technology companies that provide platforms for user-generated content are under increasing pressure to perform governance tasks, potentially facing liability for their decisions.

CS Core:

1. Privacy implications of widespread data collection including but not limited to transactional databases, data warehouses, surveillance systems, cloud computing, and artificial intelligence
2. Conceptions of anonymity, pseudonymity, and identity
3. Technology-based solutions for privacy protection (e.g., end-to-end encryption and differential privacy)
4. Civil liberties, privacy rights, and cultural differences

KA Core:

5. Philosophical and legal conceptions of the nature of privacy including the right to privacy
6. Legal foundations of privacy protection in relevant jurisdictions (e.g., GDPR in the EU)
7. Privacy legislation in areas of practice (e.g., HIPAA in the US, AI Act in the EU)
8. Basic Principles of human-subjects research and principles beyond what the law requires (e.g., Belmont Report, UN Universal Declaration on Human Rights and how this relates to technology)
9. Freedom of expression and its limitations
10. User-generated content, content moderation, and liability

Illustrative Learning Outcomes:

CS Core:

1. Evaluate solutions to privacy threats in transactional databases and data warehouses.
2. Describe the role of data collection in the implementation of pervasive surveillance systems (e.g., RFID, face recognition, toll collection, mobile computing).
3. Distinguish the concepts and goals of anonymity and pseudonymity.
4. Describe the ramifications of technology-based privacy protections, including differential privacy and end-to-end encryption.
5. Identify cultural differences regarding the nature and necessity of privacy and other civil liberties.

KA Core:

6. Discuss the philosophical basis for the legal protection of personal privacy in an appropriate context (e.g., country).
7. Critique the intent, potential value, and implementation of various forms of privacy legislation and principles beyond what the law requires.
8. Identify strategies to enable appropriate freedom of expression.

SEP-Communication: Communication

Computing is an inherently collaborative and social discipline making communication an essential aspect of the profession. Much but not all of this communication occurs in a professional setting where communication styles, expectations, and norms differ from other contexts where similar technology

might be used. Both professional and informal communication conveys information to various audiences who may have different goals and needs for that information. Good communication is also necessary for transparency and trustworthiness. It is also important to note that computing professionals are not just communicators but are also listeners who must be able to hear and thoughtfully make use of feedback received from various stakeholders. Effective communication skills are not something one 'just knows' – they are developed and can be learned. Communication skills are best taught in context throughout the undergraduate curriculum.

CS Core:

1. Oral, written, and electronic team and group communication
2. Technical communication materials (e.g., source code, and documentation, tutorials, reference materials, API documentation)
3. Communicating with different stakeholders such as customers, leadership, or the public
4. Team collaboration (including tools) and conflict resolution
5. Accessibility and inclusivity requirements for addressing professional audiences
6. Cultural competence in communication including considering the impact of difference in natural language

KA Core:

7. Tradeoffs in competing factors that affect communication channels and choices
8. Communicating to solve problems or make recommendations in the workplace, such as raising ethical concerns or addressing accessibility issues

Illustrative Learning Outcomes:

CS Core:

1. Understand the importance of writing concise and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references.
2. Analyze written technical documentation for technical accuracy, concision, lack of ambiguity, and awareness of audience.
3. Compose and deliver an audience-aware, accessible, and organized formal presentation.
4. Plan interactions (e.g., virtual, face-to-face, shared documents) with others in ways that invite inclusive participation, model respectful consideration of others' contributions, and explicitly value diversity of ideas.
5. Identify and describe qualities of effective communication (e.g., virtual, face-to-face, intragroup, shared documents).
6. Understand how to communicate effectively and appropriately as a member of a team including conflict resolution techniques.
7. Discuss ways to influence performance and results in diverse and cross-cultural teams.

KA Core:

8. Assess personal strengths and weaknesses to work remotely as part of a team drawing from diverse backgrounds and experiences.
9. Choose an appropriate way to communicate delicate ethical concerns.

SEP-Sustainability: Sustainability

Sustainability is defined by the United Nations as “development that meets the needs of the present without compromising the ability of future generations to meet their own needs.” [12] Alternatively, it is the “balance between the environment, equity and economy.” [13] As computing extends into more and more aspects of human existence, we are already seeing estimates that double-digit percentages of global electricity usage are consumed by computing activities, which unchecked will likely grow. Further, electronics contribute individually to demand for rare earth elements, mineral extraction, and countless e-waste concerns. Students should gain a background that recognizes these global and environmental costs and their potential long-term effects on the environment and local communities.

CS Core:

1. Environmental, social, and cultural impacts of implementation decisions (e.g., sustainability goals, algorithmic bias/outcomes, economic viability, and resource consumption)
2. Local/regional/global social and environmental impacts of computing systems and their use (e.g., carbon footprints, resource usage, e-waste) due to hardware (e.g., e-waste, data centers, rare element and resource utilization, recycling) and software (e.g., cloud-based services, blockchain, AI model training and use). This includes everyday use of hardware (cheap hardware replaced frequently) and software (web-browsing, email, and other services with hidden/remote computational demands).
3. Guidelines for sustainable design standards.

KA Core:

4. Systemic effects of complex computing technologies and phenomena (e.g., generative AI, data centers, social media, offshoring, remote work).
5. Pervasive computing: Information processing that has been integrated into everyday objects and activities, such as smart energy systems, social networking, and feedback systems to promote sustainable behavior, transportation, environmental monitoring, citizen science and activism.
6. How the sustainability of software systems is interdependent with social systems, including the knowledge and skills of its users, organizational processes and policies, and its societal context (e.g., market forces, government policies).

Illustrative Learning Outcomes:

CS Core:

1. Identify ways to be a sustainable practitioner in a specific area or with a specific project.
2. Assess the environmental impacts of a given project’s deployment (e.g., energy consumption, contribution to e-waste, impact of manufacturing).
3. Describe global social and environmental impacts of computer use and disposal.
4. List the sustainable effects of modern practices and activities (e.g., remote work, e-commerce, cryptocurrencies, AI models, data centers).

KA Core:

5. Describe the environmental impacts of design choices within the field of computing that relate to algorithm design, operating system design, networking design, database design, etc.
6. Analyze the social and environmental impacts of new system designs.
7. Design guidelines for sustainable IT design or deployment in areas such as smart energy systems, social networking, transportation, agriculture, supply-chain systems, environmental monitoring, and citizen activism.
8. Assess computing applications in respect to environmental issues (e.g., energy, pollution, resource usage, recycling and reuse, food management and production).

SEP-History: Computing History

History is important because it provides a mechanism for understanding why our computing systems operate the way they do, the societal contexts in which current approaches arose, and how those continue to echo through the discipline today. Not only does computing affect society but vice-versa, resulting in a complex socio-technical context that is constantly changing, requiring the perspective of history to put the present, as well as possible futures, into appropriate perspective. It also informs decisions based on successes and failures of the past including harm done and how to not repeat them. The history of computing is often taught in context with foundational concepts, such as system fundamentals and software development fundamentals. A focus should be placed on those who, due to marginalization, have not historically featured as prominently as they should.

CS Core:

1. The history of computing: hardware, software, and human/organizational.
2. The role of history in the present including within different social contexts, and the relevance of this history on the future.

KA Core:

3. Age I (Pre-digital): Ancient analog computing (Stonehenge, Antikythera mechanism, Salisbury Cathedral clock, etc.), human-calculated number tables, Euclid, Lovelace, Babbage, Gödel, Church, Turing, pre-electronic (electro-mechanical and mechanical) hardware
4. Age II (Early modern computing): ENIAC, UNIVAC, Bombes (Bletchley Park and codebreakers), computer companies (e.g., IBM), mainframes, etc.
5. Age III (PC era): PCs, modern computer hardware and software, Moore's Law
6. Age IV (Internet): Networking, internet architecture, browsers and their evolution, standards, born-on-the-internet companies, and services (e.g., Google, Amazon, Microsoft, etc.), distributed computing
7. Age V (Mobile & Cloud): Mobile computing and smartphones, cloud computing and models thereof (e.g., SaaS), remote servers, security and privacy, social media
8. Age VI (AI): Decision making systems, recommender systems, generative AI and other machine learning driven tools and technologies

Illustrative Learning Outcomes:

CS Core:

1. Understand the relevance and impact of computing history on recent events, present context, and possible future outcomes, from more than one cultural perspective.
2. Discuss how perspectives held today have been shaped by history, and that alternative perspectives exist (e.g., fears of AI replacing human workers vs AI augmenting human work, various views on Moore's Law).

KA Core:

3. Identify formative and consequential trends in the history of the computing field.
4. Identify the contributions of pioneering individuals or organizations (research labs, computer companies, government offices) in the computing field.
5. Discuss the historical context for important moments in history of computing, such as the move from vacuum tubes to transistors (TRADIC), early seminal operating systems (e.g., OS 360), Xerox PARC and the first Apple computer with a GUI, the creation of specific programming language paradigms, the first computer virus, the creation of the internet, the creation of the WWW, the dot com bubble, Y2K, the introduction of smartphones, etc.
6. Compare daily life before and after the advent of milestone developments (e.g., personal computers or the internet).

SEP-Economies: Economies of Computing

The economies of computing are important to those who develop and provide computing resources and services to others as well as society in general. They impact users of these resources and services, both professional and non-professional. Computing professionals have a duty to know the impact of these topics on their own roles and activities and how choices made will impact users and society.

KA Core:

1. Economic models: regulated and unregulated, monopolies, network effects, and open market; knowledge and attention economies
2. Pricing and deployment strategies: planned obsolescence, subscriptions, freemium, software licensing, open-source, free software, adware
3. Impacts of differences in access to computing resources, and the effect of skilled labor supply and demand on the quality of computing products
4. Automation, AI, and their effects on job markets, developers, and users
5. Ethical concerns surrounding the attention economy and other economies of computing (e.g. informed consent, data collection, use of verbose legalese in user agreements)

Illustrative Learning Outcomes:

KA Core:

1. Summarize the social effects of economic models (e.g., the knowledge and attention economies).
2. Describe the differences and similarities of competing strategies (e.g., subscription vs freemium vs free).
3. Discuss examples of digital divides.
4. Understand the effects of automation and AI on society.
5. Understand the ethical implications of computing economies that rely on attention and data.

SEP-Security: Security Policies, Laws and Computer Crimes

While security policies, laws and computer crimes are important topics, it is essential they are viewed with the foundation of other social and professional knowledge units, such as Intellectual Property, Privacy and Civil Liberties, Social Context, and Professional Ethics. Computers, the internet, and artificial intelligence, perhaps more than any other technologies, have transformed society over the past 75 years. At the same time, these technologies have contributed to unprecedented threats to privacy; new categories of computer crime and antisocial behavior; major disruptions to organizations; and the large-scale concentration of risk in information systems.

CS Core:

1. Computer crimes, legal redress for computer criminals and impact on victims and society
2. Social engineering, computing-enabled fraud, identity theft and recovery from these
3. Cyber terrorism, criminal hacking, and hacktivism
4. Malware, viruses, worms
5. Attacks on critical infrastructure such as electrical grids and pipelines
6. Non-technical fundamentals of security (e.g., human engineering, policy, confidentiality)

KA Core:

7. Benefits and challenges of existing and proposed computer crime laws
8. Security policies and the challenges of change and compliance
9. Responsibility for security throughout the computing life cycle
10. International and local laws and how they intersect

Illustrative Learning Outcomes:

CS Core:

1. List classic examples of computer crimes and social engineering incidents with societal impact.
2. Identify issues with laws that apply to computer crimes.
3. Describe the motivation and ramifications of cyber terrorism, data theft, hacktivism, ransomware, and other attacks.
4. Examine the ethical and legal issues surrounding the misuse of access and various breaches of security.
5. Discuss the professional's role in security and the tradeoffs and challenges involved.

KA Core:

6. Investigate measures that can be taken by both individuals and organizations including governments to prevent or mitigate the undesirable effects of computer crimes.
7. Design a company-wide security policy, which includes procedures for managing passwords and employee monitoring.
8. Understand how legislation from one region may affect activities in another (e.g., how EU GDPR applies globally, when EU persons are involved).

SEP-DEIA: Diversity, Equity, Inclusion, and Accessibility

Despite being a creative, often highly compensated field with myriad job (and other) opportunities, racial, gender, and many other inequities in representation are pervasive in many regions. For too many students, their first computer science course is their last. There are many factors including the legacy of systemic racism, ableism, sexism, classism, and other injustices that contribute to the lack of diverse identities within computer science, and there is no single or quick fix.

CS2023's sponsoring organizations are ACM, IEEE, and AAAI. Each of these places a high value on diversity, equity, inclusion, and accessibility; our computer science classrooms should promote and model those principles. We should welcome and seek diversity – the gamut of human differences including gender identity, ethnicity, race, politics, abilities and attributes, religion, nationality, etc.– in our classrooms, departments, and campuses. We should strive to make our classrooms, labs, and curricula accessible and to promote inclusion. We should aim for students and all community members to have a sense of belonging that comes with being respected, wanted, and valued. To achieve equity, we must allocate resources, promote fairness, and check our biases to ensure persons of all identities achieve success. Accessibility should be addressed and implemented in all computing activities and products.

Explicitly infusing diversity, equity, inclusion, and accessibility (DEIA) across the computer science curriculum demonstrates its importance for the department, institution, and our field – all of which are likely to have a DEIA statement and/or initiative(s). This emphasis on DEIA is important ethically and a bellwether issue of our time. Many professionals in computing already recognize attention to DEIA as integral to disciplinary practice. Regardless of the degree to which these values appear in any one computer science class, research suggests that a lack of attention to them results in inferior designs in addition to harm and its perpetuation. Not only does data support that diverse teams outperform homogeneous ones, but diverse teams may better prevent egregious technology failures in recent headlines such as facial recognition misuse, airbag injuries and deaths, and other well-known failures of science and computing.

CS Core:

1. How identity impacts and is impacted by computing technologies and environments (academic and professional)
2. The benefits of diverse development teams and the impacts of teams that are not diverse
3. Inclusive language and charged terminology, and why their use matters
4. Inclusive behaviors and why they matter
5. Designing and developing technology with accessibility in mind
6. How computing professionals can influence and impact diversity, equity, inclusion and accessibility, including but not only through the software they create

KA Core:

7. Experts and their practices that reflect the identities of the classroom and the world through practical DEIA principles
8. Historic marginalization due to systemic social mechanisms, technological supremacy and global infrastructure challenges to diversity, equity, inclusion, and accessibility
9. Cross-cultural differences in, and needs for, diversity, equity, inclusion, and accessibility

Illustrative Learning Outcomes:

CS Core:

1. Define and distinguish equity, equality, diversity, inclusion, and accessibility.
2. Identify language, practices, and behaviors that may make someone feel included in a workplace and/or a team, and why is it relevant. Avoid charged terminology - see *Words Matter* (www.acm.org/diversity-inclusion/words-matter) – this includes identifying and accommodating users who are often excluded without thought and not considered at all.
3. Evaluate the accessibility of your classroom or lab. Evaluate the accessibility of your webpage. (See www.w3.org/WAI and www.w3.org/TR/WCAG21).
4. Demonstrate collegiality and respect when working with team members who do not share your identity. *It is not enough to merely assign team projects. Faculty should prepare students for teamwork and monitor, mentor, and assess the effectiveness of their student teams throughout a project.*
5. Compare the demographics of your institution's computer science and STEM majors to the overall institutional demographics. If they differ, identify factors that contribute to inequitable access, engagement, and achievement in computer science among marginalized groups. If they do not, assess why not.
6. Identify developers' assumptions and values embedded in hardware and software design, especially as they pertain to usability by diverse populations.
7. Identify examples of the benefits that diverse teams can bring to software products, and how a lack of diversity has costs.

KA Core:

8. Analyze the work of experts who reflect the identities of the classroom and the world.
9. Assess the impact of power and privilege in the computing profession as it relates to culture, industry, products, and society.
10. Develop examples of systemic changes that could positively address diversity, equity, inclusion, and accessibility in a familiar context (i.e., in an introductory computing course) and an unfamiliar context and when these might be different, or the same.
11. Compare the demographics of your institution to the overall community demographics. If they differ, identify factors that contribute to inequitable access, engagement, and achievement among marginalized groups. If they do not, assess why not.

Professional Dispositions

- **Critically Self-reflective:** Students should be able to inspect their own actions, thoughts, biases, privileges, and motives to discover places where professional activity is not up to current standards. They must strive to understand both conscious and unconscious biases and continuously work to counteract them.
- **Responsive:** Students must quickly and accurately respond to changes in the field and adapt in a professional manner, such as shifting from in-person office work to remote work at home. These shifts require rethinking one's entire approach to what is considered "professional".

- **Proactive:** Students must be able to identify areas of importance (e.g., in accessibility and inclusion) and understand how to address them for a more professional working environment.
- **Culturally Competent:** Students must prioritize cultural competence—the ability to work with people from cultures different from one's own – by using inclusive language, watching for, and counteracting conscious and unconscious biases, and encouraging honest and open communication.
- **Advocative:** Students must think, speak, and act in ways that foster and promote diversity, equity, inclusion, and accessibility in all ways including but not limited to teamwork, communication, and product development (hardware and software).
- **Responsible:** Students must act responsibly in all areas of their work toward all users and stakeholders including the society at large, colleagues, and their profession in general.

Course Packaging Suggestions

In computing, societal and ethical considerations arise in all other knowledge areas and therefore should arise in the context of other computing courses, not just siloed in an "SEP course." These topics should be covered in courses starting from the first year (the only likely exception is SEP-Ethical-Analysis: Methods for Ethical Analysis) although this could be delivered as part of a first-year course or via a seminar or an online independent study.

Presenting SEP topics as advanced topics only covered in later courses could create the incorrect perception that SEP topics are only important at a certain level or complexity. While it is true that the importance and consequence of SEP topics *increases* with level and complexity, introductory topics are not devoid of SEP topics. Further, many SEP topics are *best* presented early to lay a foundation for more intricate topics later in the curriculum.

Instructor choice for some of these topics is complex. When SEP topics arise in other courses these are naturally often taught by the instructor teaching that course, although at times bringing in expert educators from other disciplines (e.g., law, ethics) could be advantageous. Stand-alone courses in SEP – should they be needed – are likely best delivered by an interdisciplinary team. However, this brings additional complexity. Regardless, who teaches SEP topics and/or courses warrants careful consideration.

At a minimum the SEP CS Core learning outcomes are best covered in the context of courses covering other knowledge areas – ideally the SEP KA Core hours are also, with the likely exception of SEP-Ethical-Analysis. This knowledge unit (KU) underlies every other KU in the SEP knowledge area (KA). However, this KU is the only one in the SEP KA that does not readily lend itself to being taught in the context of other KAs. Delivering these topics warrants even more careful consideration as to how/where they will be covered, and who will teach them. In conjunction with covering SEP topics as they occur naturally in other KAs, dedicated SEP courses can add value. However, a sole, stand-alone course in a program where SEP topics are not covered in other courses should be a last resort.

At some institutions, an **in-depth dedicated course** at the mid- or advanced-level may be offered covering all recommended topics in both the CS Core and KA Core KUs **in close coordination with learning outcomes best covered in the context of courses covering other KAs.** Such a course could include:

- SEP-Context (5 hours)
- SEP-Ethical-Analysis (3 hours)
- SEP-Professional-Ethics (4 hours)
- SEP-IP (2 hours)
- SEP-Privacy (3 hours)
- SEP-Communication (3 hours)
- SEP-Sustainability (2 hours)
- SEP-History (2 hours)
- SEP-Economies (1 hour)
- SEP-Security (3 hours)
- SEP-DEIA (4 hours)

Skill Statement

A student who completes this course should be able to contribute to systemic change by applying societal and ethical knowledge using relevant underpinnings and frameworks to their work in the computing profession in a culturally competent manner including contributing to positive developments in inclusion, equity, diversity, and accessibility in computing.

At some institutions, a **dedicated minimal course** may be offered covering the CS Core knowledge units **in close coordination with learning outcomes best covered in the context of courses covering other knowledge areas**. Such a course could include:

- SEP-Context (3 hours)
- SEP-Ethical-Analysis (2 hours)
- SEP-Professional-Ethics (2 hours)
- SEP-IP (1 hour)
- SEP-Privacy (2 hours)
- SEP-Communication (2 hours)
- SEP-Sustainability (1 hour)
- SEP-History (1 hour)
- SEP-Security (2 hours)
- SEP-DEIA (2 hours)

Skill Statement

A student who completes this course should be able to apply societal and ethical knowledge to their work in the computing profession while fostering and contributing to inclusion, equity, diversity, and accessibility in computing.

Some Exemplary Materials

- Emanuelle Burton, Judy Goldsmith, Nicholas Mattei, Cory Siler, and Sara-Jo Swiatek. 2023. Teaching Computer Science Ethics Using Science Fiction. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2 (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 1184. https://doi.org/10.1145/3545947.3569618
- Randy Connolly. 2020. Why computing belongs within the social sciences. *Commun. ACM* 63, 8 (August 2020), 54–59. https://doi.org/10.1145/3383444

- Casey Fiesler. Tech Ethics Curricula: A Collection of Syllabi Used to Teach Ethics in Technology Across Many Universities
    a. https://cfiesler.medium.com/tech-ethics-curricula-a-collection-of-syllabi-3eedfb76be18 ; accessed March 12, 2024.
    b. Tech Ethics Curricula; accessed March 12, 2024.
- Casey Fiesler. Tech Ethics Readings: A Spreadsheet of Readings Used to Teach Ethics in Technology Tech Ethics Class Readings; accessed March 12, 2024.
- Stanford Embedded EthiCS, Embedding Ethics in Computer Science. https://embeddedethics.stanford.edu/; accessed March 12, 2024.
- Jeremy, Weinstein, Rob Reich, and Mehran Sahami. *System Error: Where Big Tech Went Wrong and How We Can Reboot*. Hodder Paperbacks, 2023.
- Baecker, R. Computers in Society: Modern Perspectives, Oxford University Press. (2019).
- Embedded EthiCS @ Harvard: bringing ethical reasoning into the computer science curriculum. https://embeddedethics.seas.harvard.edu/about; accessed March 12, 2024.

Committee

Chair: Brett A. Becker, University College Dublin, Dublin, Ireland

Members:

- Richard L. Blumenthal, Regis University, Denver, CO, USA
- Mikey Goldweber, Denison University, Granville, OH, USA
- James Prather, Abilene Christian University, Abilene, TX, USA
- Susan Reiser, University of North Carolina Asheville, Asheville, NC, USA
- Michelle Trim, University of Massachusetts Amherst, Amherst, MA, USA
- Titus Winters, Google, Inc, New York, NY, USA

Contributors:

- Jake Baskin, Computer Science Teachers Association, Chicago, IL, USA
- Johanna Blumenthal, Regis University, Denver, CO, USA
- Chris Stephenson, Google, Portland, OR, USA
- MaryAnne Egan, Siena College, Loudonville, NY, USA
- Catherine Mooney, University College Dublin, Dublin, Ireland
- Fay Cobb Payton, North Carolina State University, Raleigh, NC, USA
- Keith Quille, Technological University of Dublin, Dublin, Ireland
- Mehran Sahami, Stanford University, Stanford, CA, USA
- Mark Scanlon, University College Dublin, Dublin, Ireland
- Karren Shorofsky, University of San Francisco School of Law, San Francisco, CA, USA
- Andreas Stefik, University of Nevada, Las Vegas, Las Vegas, NV, USA
- Ellen Walker, Hiram College, Cleveland, OH, USA

References

1. ACM/IEEE-CS Joint Curriculum Task Force. “Computing Curricula 1991.” (New York, USA: ACM Press and IEEE Computer Society Press, 1991).
2. ACM/IEEE-CS Joint Curriculum Task Force. “Computing Curricula 2001 Computer Science.” (New York, USA: ACM Press and IEEE Computer Society Press, 2001).
3. ACM/IEEE-CS Interim Review Task Force. “Computer Science Curriculum 2008: An interim revision of CS 2001.” (New York, USA: ACM Press and IEEE Computer Society Press, 2008).
4. Randy Connolly. 2020. Why computing belongs within the social sciences. *Commun. ACM* 63, 8 (August 2020), 54–59. https://doi.org/10.1145/3383444
5. ACM Code of Ethics and Professional Conduct. www.acm.org/about/code-of-ethics
6. IEEE Code of Ethics. https://www.ieee.org/about/corporate/governance/p7-8.html; accessed March 12, 2024.
7. AAAI Code of Professional Ethics and Conduct. https://aaai.org/Conferences/code-of-ethics-and-conduct.php; accessed March 12, 2024.
8. Diversity, Equity, and Inclusion - Welcoming All to Computing https://www.acm.org/diversity-inclusion; accessed March 12, 2024.
9. Diversity, Equity & Inclusion at IEEE. https://www.ieee.org/about/diversity-index.html; accessed March 12, 2024.
10. AAAI Diversity Statement. https://aaai.org/about-aaai/ethics-and-diversity/#diversity-statement; accessed March 12, 2024.
11. A Framework for Ethical Decision Making. https://www.scu.edu/ethics/ethics-resources/a-framework-for-ethical-decision-making/ accessed March 12, 2024.
12. Sustainability | United Nations. https://www.un.org/en/academic-impact/sustainability; accessed March 12, 2024.
13. What is Sustainability? https://www.sustain.ucla.edu/what-is-sustainability; accessed March 12, 2024.

Systems Fundamentals (SF)

Preamble

A computer system is a set of hardware and software infrastructures upon which applications are constructed. Computer systems have become a pillar of people's daily life. As such, it is essential for students to learn knowledge about computer systems, grasp the skills to use and design these systems, and understand the fundamental rationale and principles in computer systems. It could equip students with the necessary competence for a career related to computer science.

In the curriculum of computer science, the study of computer systems typically spans multiple knowledge areas, including, but not limited to, operating systems, parallel and distributed systems, communications networks, computer architecture and organization, and software engineering. The System Fundamentals knowledge area, as suggested by its name, focuses on the fundamental concepts and design principles in computer systems that are shared by these courses within their respective cores. The goal of this knowledge area is to present an integrative view of these fundamental concepts and design principles in a unified albeit simplified fashion, providing a common foundation for the different specialized mechanisms and policies appropriate to the specific domain area. The fundamental concepts in this knowledge area include an overview of computer systems, basic concepts such as state and state transition, resource allocation and scheduling, and so on. Moreover, this knowledge area introduces basic design principles to improve the reliability, availability, efficiency, and security of computer systems.

Changes since CS2013

Compared to CS2013, the SF knowledge area incorporated significant changes to the knowledge units.

- Added two new knowledge units: System Security and System Design.
- Added a new knowledge unit named System Performance, which includes the topics from the deprecated knowledge unit of Proximity and the deprecated knowledge unit of Virtualization and Isolation.
- Added a new knowledge unit named Performance Evaluation, that includes the topics from the deprecated unit of Evaluation and the deprecated unit of Quantitative Evaluation.
- Renamed the Computational Paradigms knowledge unit to Overview of Computer Systems, deprecated some topics in the unit, and added topics from the deprecated unit of Cross-Layer Communications.
- Renamed the State and State Machines knowledge unit to Basic Concepts and added topics such as finite state machines.
- Deprecated the Cross-Layer Communications knowledge unit and moved parts of its topics to the unit of Overview of Computer Systems.
- Deprecated the Evaluation and Quantitative Evaluation knowledge units and moved parts of their topics to the unit of Performance Evaluation.
- Deprecated the Proximity and Virtualization and Isolation knowledge units and moved parts of their topics to the knowledge unit of System Performance.

- Deprecated the Parallelism knowledge unit and moved parts of its topic to the Basic Concepts knowledge unit.
- Renamed the Reliability through Redundancy knowledge unit to System Reliability.
- Added the Society, Ethics, and the Profession knowledge unit.

Core Hours

Knowledge Unit	CS Core	KA Core
Overview of Computer Systems	3	0
Basic Concepts	4	0
Resource Management	1	1
System Performance	2	2
Performance Evaluation	2	2
System Reliability	2	1
System Security	2	1
System Design	2	1
Society, Ethics, and Profession	Included in SEP hours	
Total	**18**	**8**

Knowledge Units

SF-Overview: Overview of Computer Systems

***CS Core**:*

1. Basic building blocks and components of a computer (gates, flip-flops, registers, interconnections; datapath + control + memory)
2. Hardware as a computational paradigm: Fundamental logic building blocks; Logic expressions, minimization, sum of product forms (See also: AR-Logic)
3. Programming abstractions, interfaces, use of libraries (See also: PDC-Programs)
4. Distinction and interaction between application and OS services, remote procedure call (See also: OS-Purpose)
5. Basic concept of pipelining, overlapped processing stages (See also: AR-Organization)
6. Basic concept of scaling: performance vs problem size

Illustrative Learning Outcomes:
***CS Core**:*

1. Describe the basic building blocks of computers and their role in the historical development of computer architecture.
2. Design a simple logic circuit using the fundamental building blocks of logic design to solve a simple problem (e.g., adder).
3. Describe how computing systems are constructed of layers upon layers, based on separation of concerns, with well-defined interfaces, hiding details of low layers from the higher layers.
4. Describe that hardware, OS, VM, and application are additional layers of interpretation/processing.
5. Describe the mechanisms of how errors are detected, signaled back, and handled through the layers.
6. Construct a simple program (e.g., a TCP client/server) using methods of layering, error detection and recovery, and reflection of error status across layers.
7. Identify bugs in a layered program by using tools for program tracing, single stepping, and debugging.
8. Understand the concept of strong vs weak scaling, i.e., how performance is affected by the scale of the problem vs the scale of resources to solve the problem. This can be motivated by simple, real-world examples.

SF-Foundations: Basic Concepts

CS Core:

1. Digital vs Analog/Discrete vs Continuous Systems
2. Simple logic gates, logical expressions, Boolean logic simplification
3. Clocks, State, Sequencing
4. State and state transition (e.g., starting state, final state, life cycle of states) (See also: AL-Models)
5. Finite state machines (e.g., NFA, DFA) (See also: AL-Models)
6. Combinational Logic, Sequential Logic, Registers, Memories (See also: AR-Logic)
7. Computers and Network Protocols as examples of State Machines (See also: NC-Fundamentals)
8. Sequential vs parallel processing. (See also: PDC-Programs, OS-Concurrency)
9. Application-level sequential processing: single thread (See also: PDC-Programs, OS-Concurrency)
10. Simple application-level parallel processing: request level (web services/client-server/distributed), single thread per server, multiple threads with multiple servers, pipelining (See also: PDC-Programs, OS-Concurrency)

Illustrative Learning Outcomes:

CS Core*:*

1. Describe the differences between digital and analog systems, and between discrete and continuous systems. Can give real-world examples of these systems.
2. Describe computations as a system characterized by a known set of configurations with transitions from one unique configuration (state) to another (state).
3. Describe the distinction between systems whose output is only a function of their input (stateless) and those with memory/history (stateful).
4. Develop state machine descriptions for simple problem statement solutions (e.g., traffic light sequencing, pattern recognizers).
5. Describe a computer as a state machine that interprets machine instructions.

6. Explain how a program or network protocol can also be expressed as a state machine and that alternative representations for the same computation can exist.
7. Derive the time-series behavior of a state machine from its state machine representation (e.g., TCP connection management state machine).
8. Write a simple sequential problem and a simple parallel version of the same program.
9. Evaluate the performance of simple sequential and parallel versions of a program with different problem sizes and be able to describe the speed-ups achieved.
10. Describe on an execution timeline how parallelism events and operations can take place simultaneously (i.e., at the same time). Explain how work can be performed in less elapsed time if this can be exploited.

SF-Resource: Resource Management

***CS Core**:*

1. Different types of resources (e.g., processor share, memory, disk, net bandwidth) (See also: OS-Scheduling, OS-Memory, OS-Files, NC-Fundamentals)
2. Common resource allocation/scheduling algorithms (e.g., first-come-first-serve, priority-based scheduling, fair scheduling, and preemptive scheduling) (See also: OS-Scheduling)

KA Core:

3. Advantages and disadvantages of common scheduling algorithms (See also: OS-Scheduling)

***Illustrative Learning Outcomes**:*

CS Core:

1. Define how finite computer resources (e.g., processor share, memory, storage, and network bandwidth) are managed by their careful allocation to existing entities.
2. Describe how common resource allocation/scheduling algorithms work.
3. Develop common scheduling algorithms and evaluate their performances.

KA Core:

4. Describe the pros and cons of common scheduling algorithms.

SF-Performance: System Performance

***CS Core**:*

1. Latencies in computer systems
    a. Speed of light and computers (one foot per nanosecond vs one GHz clocks) (See also: AR-Organization)
    b. Memory vs disk latencies vs across-the-network memory (See also: AR-Memory, AR-Performance-Energy)
2. Caches and the effects of spatial and temporal locality on performance in processors and systems (See also: AR-Memory, AR-Performance-Energy, OS-Memory)
3. Caches and cache coherency in databases, operating systems, distributed systems, and computer architecture (See also: OS-Memory, AR-Memory, DM-Internals)
4. Introduction to the processor memory hierarchy (See also: AR-Memory, AR-Performance-Energy)

KA Core:

5. The formula for average memory access time (See also: AR-Memory)
6. Rationale of virtualization and isolation: protection and predictable performance (See also: OS-Virtualization)
7. Levels of indirection, illustrated by virtual memory for managing physical memory resources (See also: OS-Virtualization)
8. Methods for implementing virtual memory and virtual machines (See also: OS-Virtualization)

Illustrative Learning Outcomes*:*

CS Core:

1. Describe the breakdown of the latency of computer systems in terms of memory, disk, and network.
2. Explain the importance of locality in determining system performance.
3. Calculate average memory access time and describe the tradeoffs in memory hierarchy performance in terms of capacity, miss/hit rate, and access time.

KA Core:

4. Explain why it is important to isolate and protect the execution of individual programs and environments that share common underlying resources.
5. Describe how the concept of indirection can create the illusion of a dedicated machine and its resources even when physically shared among multiple programs and environments.
6. Evaluate the performance of two application instances running on separate virtual machines and determine the effect of performance isolation.

SF-Evaluation: Performance Evaluation

CS Core*:*

1. Performance figures of merit (See also: AR-Performance-Energy, PDC-Evaluation)
2. Workloads and representative benchmarks, and methods of collecting and analyzing performance figures of merit (See also: AR-Performance-Energy, PDC-Evaluation)
3. CPI (Cycles per Instruction) equation as a tool for understanding tradeoffs in the design of instruction sets, processor pipelines, and memory system organizations (See also: AR-Performance-Energy, PDC-Evaluation)
4. Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can (See also: AR-Performance-Energy, PDC-Evaluation)
5. Order of magnitude analysis (Big O notation) (See also: AL-Complexity)
6. Analysis of slow and fast paths of a system (See also: AR-Organization)
7. Events on their effect on performance (e.g., instruction stalls, cache misses, page faults) (See also: OS-Memory, AR-Organization)

KA Core:

7. Analytical tools to guide quantitative evaluation
8. Understanding layered systems, workloads, and platforms, their implications for performance, and the challenges they represent for evaluation
9. Microbenchmark pitfalls

***Illustrative Learning Outcomes**:*

CS Core:

1. Explain how the components of system architecture contribute to improving its performance.
2. Explain the circumstances in which a given figure of a system performance metric is useful.
3. Explain the usage and inadequacies of benchmarks as a measure of system performance.
4. Describe Amdahl's law and discuss its limitations.
5. Apply limit studies or simple calculations to produce order-of-magnitude estimates for a given performance metric in a given context.
6. Apply software tools to profile and measure program performance.

KA Core:

7. Design and conduct a performance-oriented experiment of a common system (e.g., an OS and Spark).
8. Design a performance experiment on a layered system to determine the effect of a system parameter on system performance.

SF-Reliability: System Reliability

***CS Core**:*

1. Distinction between bugs, faults, and failures (See also: PDC-Coordination, SE-Reliability)
2. Reliability vs availability
3. Reliability through redundancy
   a. check and retry (See also: OS-Faults, NC-Reliability)
   b. redundant encoding (error correction codes, CRC, FEC, RAID) (See also: AR-Memory, NC-Reliability, DM-Distributed)
   c. duplication/mirroring/replicas (See also: DM-Distributed)

KA Core:

4. Other approaches to reliability (e.g., journaling) (See also: OS-Faults, NC-Reliability, SE-Reliability)

***Illustrative Learning Outcomes**:*

CS Core:

1. Explain the distinction between program errors, system errors, and hardware faults (e.g., corrupted memory) and exceptions (e.g., attempt to divide by zero).
2. Articulate the distinction between detecting, handling, and recovering from faults and the methods for their implementation.
3. Describe the role of error correction codes in providing error checking and correction techniques in memories, storage, and networks.
4. Apply simple algorithms for exploiting redundant information for the purposes of data correction.

KA Core:

5. Compare different error detection and correction methods for their data overhead, implementation complexity, and relative execution time for encoding, detecting, and correcting errors.

SF-Security: System Security

***CS Core**:*

1. Common system security issues (e.g., viruses, denial-of-service attacks, and eavesdropping) (See also: OS-Protection, NC-Security, SEC-Foundations, SEC-Engineering)
2. Countermeasures (See also: OS-Principles, OS-Protection, NC-Security)
   a. Cryptography (See also: SEC-Crypto)
   b. Security architecture (See also: SEC-Engineering)

KA Core:

3. Representative countermeasure systems
   a. Intrusion detection systems, firewalls (See also: NC-Security)
   b. Antivirus systems

***Illustrative Learning Outcomes**:*

***CS Core**:*

1. Describe some common system security issues and give examples
2. Describe some countermeasures against system security issues

***KA Core**:*

3. Describe representative countermeasure systems

SF-Design: System Design

***CS Core**:*

1. Common criteria of system design (e.g., liveness, safety, robustness, scalability, and security) (See also: PDC-Evaluation)

KA Core:

2. Designs of representative systems (e.g., Apache web server, Spark, and Linux)

***Illustrative Learning Outcomes**:*

CS Core:

1. Describe common criteria of system design.
2. Given the functionality requirements of a system and its key design criteria, provide a high-level design of this system.

KA Core:

3. Describe the design of some representative systems.

SF-SEP: Society, Ethics, and the Profession

***KA Core**:*

1. Intellectual property rights of computer systems (See also: SEP-IP)
2. Common software licenses (See also: SEP-IP)
3. Computer crimes (See also: SEP-Security)

***Illustrative Learning Outcomes**:*
***KA Core**:*

1. Describe the intellectual property rights of computer systems.
2. List representative software licenses and compare their differences.
3. List representative computer crimes.

Professional Dispositions

- **Meticulous**: Students must pay attention to details of different perspectives when learning about and evaluating systems.
- **Adaptive**: Students must be flexible and adaptive when designing systems. Different systems have different requirements, constraints and working scenarios. As such, they require different designs. Students must be able to make appropriate design decisions correspondingly.

Mathematics Requirements

Required:

- Discrete Mathematics (See also: MSF-Discrete)
  - Sets and relations
  - Basic graph theory
  - Basic logic
- Linear Algebra (See also: MSF-Linear)
  - Basic matrix operations
- Probability and Statistics (See also: MSF-Probability, MSF-Statistics)
  - Random variable
  - Bayes theorem
  - Expectation and Variation
  - Cumulative distribution function and probability density function

Desirable:

- Basic queueing theory
- Basic stochastic process

Course Packaging Suggestions

Introductory Course to include the following:

- SF-Overview (2 hours)
- SF-Foundations (6 hours)
- SF-Resource (4 hours)
- SF-Performance (6 hours)
- SF-Evaluation (6 hours)
- SF-Reliability (4 hours)

- SF-Security (5 hours)
- SF-SEP (1 hour)
- SF-Design (6 hours)

Prerequisites:

- Sets and relations, basic graph theory and basic logic from Discrete Mathematics (See also: MSF-Discrete)
- Basic matrix operations from Linear Algebra (See also: MSF-Linear)
- Random variable, Bayes theorem, expectation and variation, cumulative distribution function and probability density function from Probability and Statistics (See also: MSF-Probability, MSF-Statistics)

Course objectives: Students should be able to (1) understand the fundamental concepts in computer systems; (2) understand the key design principles, in terms of performance, reliability and security, when designing computer systems; (3) deploy and evaluate representative complex systems (e.g., MySQL and Spark) based on their documentations, and (4) design and implement simple computer systems (e.g., an interactive program, a simple web server, and a simple data storage system).

Advanced Course to include the following:

- SF-Overview: Overview of Computer Systems (2 hours)
- SF-Design: System Design (8 hours)
- OS-Purpose, OS-Principles (2 hours)
- NC-Fundamentals, NC-Networked-Applications (2 hours)
- PDC-Programs (2 hours)
- AR-IO, AR-Performance-Energy (2 hours)
- SF-Reliability (8 hours)
- SF-Performance (6 hours)
- SF-Security (6 hours)
- SF-SEP: (2 hours)

Prerequisites:

- Basic queueing theory and stochastic process (See also: MSF-Probability, MSF-Statistics)
- Introductory course of the SF knowledge area

Course objectives: Students should be able to (1) have a deeper understanding in the key design principles of computer system design, (2) map such key principles to the designs of classic systems (e.g., Linux, SQL and TCP/IP network stack) as well as that of more recent systems (e.g., Hadoop, Spark and distributed storage systems), and (3) design and implement more complex computer systems (e.g., a file system and a high-performance web server).

Committee

Chair: Qiao Xiang, Xiamen University, Xiamen, China

Members:

- Doug Lea, State University of New York at Oswego, Oswego, NY, USA
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Matthias Hauswirth, University of Lugano, Lugano, Switzerland
- Ennan Zhai, Alibaba Group, Hangzhou, China
- Yutong Liu, Shanghai JiaoTong University, Shanghai, China

Contributors:

- Michael S. Kirkpatrick, James Madison University, Harrisonburg, VA, USA
- Linghe Kong, Shanghai JiaoTong University, Shanghai, China

Specialized Platform Development (SPD)

Preamble

The Specialized Platform Development (SPD) knowledge area refers to attributes involving the creation of software targeting non-traditional hardware platforms. Developing for each specialized platform, for example, robots, mobile systems, web-based systems, and embedded systems, typically involves unique considerations.

Societal and industry needs have created a high demand for developers on specialized platforms, such as mobile applications, web platforms, robotic platforms, and embedded systems. Some unique professional abilities relevant to this knowledge area include the following.

- Creating applications that provide a consistent user experience across various devices, screen sizes, and operating systems.
- Developing application programming interfaces (APIs) to support the functionality of each specialized platform.
- Managing challenges related to resource constraints such as computation, memory, storage, and networking and communication.
- Applying cross-cutting concerns such as optimization, security, better development practices, etc.

Changes since CS2013

The assessment of these factors has led to significant modifications from the CS2013 version, including the following.

- The knowledge area name has been changed to reflect the specialized development platforms which serve as the target for software development.
- Reflecting the increased deployment of specialized hardware platforms, the number of CS Core hours has been increased.
- Reflecting modern computing systems, knowledge units in Robotics, Embedded Systems, and Society, Ethics, and the Profession (SEP) have been introduced.
- Other changes include: 1) renamed Introduction knowledge unit to Common Aspects/Shared Concerns and 2) renamed Industrial Platforms to Robot Platforms.

Core Hours

Knowledge Unit	CS Core	KA Core
Common Aspects	3 + (1 SE)	2
Web Platforms		5 + (1 HCI)

Mobile Platforms		3 + (2 DM) + (1 GIT) + (1 HCI) + (1 NC)
Robot Platforms		4 + (3 GIT) + (3 AI)
Embedded Platforms		4 + (4 AR) + (1 FPL) + (1 GIT) + (3 OS) + (1 SF)
Game Platforms		4 + (1 AL) + (1 AI) + (4 AR) + (5 GIT) + (2 HCI) + (1 SDF) + (1 SE) + (1 MSF)
Interactive Computing Platforms		3 + (1 DM) + (2 GIT) + (1 AR) + (1 FPL)
SPD-SEP	Included in SEP hours	
Total	**4**	

Note: The CS Core hours total includes 1 hour shared with SE.

Knowledge Units

SPD-Common: Common Aspects/Shared Concerns

CS Core:

1. Overview of development platforms (i.e., web, mobile, game, robotics, embedded, and interactive).
    a. Input/sensors/control devices/haptic devices
    b. Resource constraints
        i. Computational
        ii. Data storage
        iii. Memory
        iv. Communication
    c. Requirements – security, uptime availability, fault tolerance (See also: SE-Reliability, SEC-Engineering)
    d. Output/actuators/haptic devices
2. Programming via platform-specific Application Programming Interface (API) vs traditional application construction
3. Overview of platform Languages (e.g., Python, Swift, Lua, Kotlin)
4. Programming under platform constraints and requirements (e.g., available development tools, development, security considerations) (See also: SEC-Foundations)
5. Techniques for learning and mastering a platform-specific programming language

Illustrative Learning Outcomes:

CS Core:

1. List the constraints of mobile programming.
2. List the characteristics of scripting languages.
3. Describe the three-tier model of web programming.

4. Describe how the state is maintained in web programming.

SPD-Web: Web Platforms

KA Core:

1. Web programming languages (e.g., HTML5, JavaScript, PHP, CSS)
2. Web platforms, frameworks, or meta-frameworks
    a. Cloud services
    b. API, Web Components
3. Software as a Service (SaaS)
4. Web standards such as document object model, accessibility (See also: HCI-Accessibility)
5. Security and Privacy Considerations (See also: SEP-Security)

Non-core:

6. Analyzing requirements for web applications
7. Computing services (See also: DM-NoSQL)
    a. Cloud Hosting
    b. Scalability (e.g., Autoscaling, Clusters)
    c. Cost estimation for services
8. Data management (See also: DM-Core)
    a. Data residency: where the data is located and what paths can be taken to access it
    b. Data integrity: guaranteeing data is accessible and that data is deleted when required
9. Architecture
    a. Monoliths vs Microservices
    b. Micro-frontends
    c. Event-Driven vs RESTful architectures: advantages and disadvantages
    d. Serverless, cloud computing on demand
10. Storage solutions (See also: DM-Relational, DM-NoSQL)
    a. Relational Databases
    b. NoSQL databases

Illustrative Learning Outcomes:

KA Core:

1. Design and implement a web-based application using a microservice architecture design.
2. Describe the constraints, such as hosting, services, and scalability, related to web platforms.
3. Compare and contrast web programming with general-purpose programming.
4. Describe the differences between Software-as-a-Service (SaaS) and traditional software products.
5. Discuss how web standards impact software development.
6. Evaluate an existing web application against current web standards.

SPD-Mobile: Mobile Platforms

KA Core:

1. Development with
    a. Mobile programming languages

   b. Mobile programming environments
2. Mobile platform constraints
   a. User interface design (See also: HCI-User)
   b. Security
3. Access
   a. Accessing data through APIs (See also: DM-Querying)
   b. Designing API endpoints for mobile apps: pitfalls and design considerations
   c. Network and web interfaces (See also: NC-Fundamentals, DM-Modeling)

Non-core:

4. Development
   a. Native versus cross-platform development
   b. Software design/architecture patterns for mobile applications (See also: SE-Design)
5. Mobile platform constraints
   a. Responsive user interface design (See also: HCI-Accessibility)
   b. Heterogeneity and mobility of devices
   c. Differences in user experiences (e.g., between mobile and web-based applications)
   d. Power and performance tradeoff
6. Mobile computing affordances
   a. Location-aware applications
   b. Sensor-driven computing (e.g., gyroscope, accelerometer, health data from a watch)
   c. Telephony and instant messaging
   d. Augmented reality (See also: GIT-Immersion)
7. Specification and testing (See also: SDF-Practices, SE-Validation)
8. Asynchronous computing (See also: PDC-Algorithms)
   a. Difference from traditional synchronous programming
   b. Handling success via callbacks
   c. Handling errors asynchronously
   d. Testing asynchronous code and typical problems in testing

Illustrative Learning Outcomes:

KA Core:

1. Compare mobile programming with general-purpose programming.
2. Develop a location-aware mobile application with data API integration.
3. Build a sensor-driven mobile application capable of logging data on a remote server.
4. Create a communication app incorporating telephony and instant messaging.
5. Evaluate the pros and cons of native and cross-platform mobile application development.

SPD-Robot: Robot Platforms

KA Core:

1. Types of robotic platforms and devices (See also: AI-Robotics)
2. Sensors, embedded computation, and effectors (actuators) (See also: GIT-Physical)
3. Robot-specific languages and libraries (See also: AI-Robotics)
4. Robotic software architecture (e.g., using the Robot Operating System (ROS))

5. Robotic platform constraints and design considerations (See also: AI-Robotics)
6. Interconnections with physical or simulated systems (See also: GIT-Physical, GIT-Simulation)
7. Robotic Algorithms (See also: AI-Robotics, GIT-Animation)
    a. Forward kinematics
    b. Inverse kinematics
    c. Dynamics
    d. Navigation and path planning
    e. Grasping and manipulation
8. Safety and interaction considerations (See also: SEP-Professional-Ethics, SEP-Context)

Illustrative Learning Outcomes:

KA Core:

1. Design and implement an application on a given robotic platform.
2. Integrate an Arduino-based robot kit and program it to navigate a maze.
3. Compare robot-specific languages and techniques with those used for general-purpose software development.
4. Explain the rationale behind the design of the robotic platform and its interconnections with physical or simulated systems.
5. Given a high-level application, design a robot software architecture using ROS specifying all components and interconnections (ROS topics) to accomplish that application.
6. Discuss the constraints a given robotic platform imposes on developers.

SPD-Embedded: Embedded Platforms

KA Core:

1. Introduction to the unique characteristics of embedded systems
    a. Real-time vs soft real-time and non-real-time systems
    b. Resource constraints, such as memory profiles and deadlines (See also: AR-Memory)
2. API for custom architectures
    a. GPU technology (See also: AR-Heterogeneity, GIT-Shading)
    b. Field Programmable Gate Arrays (FPGA) (See also: AR-Logic)
    c. Cross-platform systems
3. Embedded Systems
    a. Microcontrollers
    b. Interrupts and feedback
    c. Interrupt handlers in high-level languages (See also: SF-Overview)
    d. Hard and soft interrupts and trap-exits (See also: OS-Principles)
    e. Interacting with hardware, actuators, and sensors
    f. Energy efficiency
    g. Loosely timed coding and synchronization
    h. Software adapters
4. Embedded programming
5. Hard real-time systems vs soft real-time systems (See also: OS-Real-time)
    a. Timeliness
    b. Time synchronization/scheduling

    c. Prioritization
    d. Latency
    e. Compute jitter
6. Real-time resource management
7. Memory management
    a. Mapping programming construct (variable) to a memory location (See also: AR-Memory)
    b. Shared memory (See also: OS-Memory)
    c. Manual memory management.
    d. Garbage collection (See also: FPL-Translation)
8. Safety considerations and safety analysis (See also: SEP-Context, SEP-Professional-Ethics)
9. Sensors and actuators
10. Analysis and verification
11. Application design

Illustrative Learning Outcomes:

KA Core:

1. Design and implement a small embedded system for a given platform (e.g., a smart alarm clock or a drone).
2. Describe the unique characteristics of embedded systems versus other systems.
3. Interface with sensors/actuators.
4. Debug a problem with an existing embedded platform.
5. Identify different types of embedded architectures.
6. Evaluate which architecture is best for a given set of requirements.
7. Design and develop software to interact with and control hardware.
8. Design methods for real-time systems.
9. Evaluate real-time scheduling and schedulability analysis.
10. Evaluate formal specification and verification of timing constraints and properties.

SPD-Game: Game Platforms

KA Core:

1. Historical and contemporary platforms for games (See also: AR-Logic)
    a. Evolution of Game Platforms (e.g., Brown Box to Metaverse and beyond; Improvement in Computing Architectures (CPU and GPU); Platform Convergence and Mobility)
    b. Typical Game Platforms (e.g., Personal Computer; Home Console; Handheld Console; Arcade Machine; Interactive Television; Mobile Phone; Tablet; Integrated Head-Mounted Display; Immersive Installations and Simulators; Internet of Things enabled Devices; CAVE Systems; Web Browsers; Cloud-based Streaming Systems)
    c. Characteristics and Constraints of Different Game Platforms (e.g., Features (local storage, internetworking, peripherals); Run-time performance (GPU/CPU frequency, number of cores); Chipsets (physics processing units, vector co-processors); Expansion Bandwidth (PCIe); Network throughput (Ethernet); Memory types and capacities (DDR/GDDR); Maximum stack depth; Power consumption; Thermal design; Endian)
    d. Typical Sensors, Controllers, and Actuators (e.g., distinctive control system designs – peripherals (mouse, keypad, joystick), game controllers, wearables, interactive surfaces;

electronics and bespoke hardware; computer vision, inside-out tracking, and outside-in tracking; IoT-enabled electronics and I/O (See also: GIT-Interaction)

e. eSports Ecosystems (e.g., evolution of gameplay across platforms; games and eSports; game events such as LAN/arcade tournaments and international events such as the Olympic eSports Series; streamed media and spectatorship; multimedia technologies and broadcast management; professional play; data and machine learning for coaching and training)

2. Real-time Simulation and Rendering Systems
    a. CPU and GPU architectures (e.g., Flynn's taxonomy; parallelization; instruction sets; standard components – graphics compute array, graphics memory controller, video graphics array basic input/output system; bus interface; power management unit; video processing unit; display interface) (See also: AR-Heterogeneity)
    b. Pipelines for physical simulations and graphical rendering: (e.g., tile-based, immediate-mode). (See also: GIT-Rendering)
    c. Common Contexts for Algorithms, Data Structures, and Mathematical Functions (e.g., game loops; spatial partitioning, viewport culling, and level of detail; collision detection and resolution; physical simulation; behavior for intelligent agents; procedural content generation) (See also: MSF-Discrete, AL-Foundational)
    d. Media representations (e.g., I/O, and computation techniques for virtual worlds: audio; music; sprites; models and textures; text; dialogue; multimedia (e.g., olfaction, tactile) (See also: GIT-Fundamentals)
3. Game Development Tools and Techniques
    a. Programming Languages (e.g., C++; C#; Lua; Python; JavaScript)
    b. Shader Languages (e.g., HLSL, GLSL; Shader Graph)
    c. Graphics Libraries and APIs (e.g., DirectX; SDL; OpenGL; Metal; Vulkan; WebGL). (See also: GIT-Rendering, HCI-Design)
    d. Common Development Tools and Environments (e.g., IDEs; Debuggers; Profilers; Version Control Systems including those handling binary assets; Development Kits and Production/Consumer Kits; Emulators) (See also: SDF-Practices, SE-Tools)
4. Game Engines
    a. Open Game Engines (e.g., Unreal; Unity; Godot; CryEngine; Phyre; Source 2; Pygame and Ren'Py; Phaser; Twine; Spring RTS)
    b. Techniques (e.g., Ideation, Prototyping, Iterative Design and Implementation, Compiling Executable Builds, Development Operations and Quality Assurance – Play Testing and Technical Testing, Profiling; Optimization, Porting; Internationalization and Localization, Networking) (See also: AR-Performance-Energy, SE-Requirements)
5. Game Design
    a. Vocabulary (e.g., game definitions; mechanics-dynamics-aesthetics model; industry terminology; experience design; models of experience and emotion)
    b. Design Thinking and User-Centered Experience Design (e.g., methods of designing games; iteration, incrementing, and the double-diamond; phases of pre- and post-production; quality assurance, including alpha and beta testing; stakeholder and customer involvement; community management) (See also: SE-Design)

c. Genres (e.g., adventure; walking simulator; first-person shooter; real-time strategy; multiplayer online battle arena (MOBA); role-playing game (rpg))
d. Audiences and Player Taxonomies (e.g., people who play games; diversity and broadening participation; pleasures, player types, and preferences; Bartle, yee) (See also: HCI-User)
e. Proliferation of digital game technologies to domains beyond entertainment (e.g., Education and Training; Serious Games; Virtual Production; eSports; Gamification; Immersive Experience Design; Creative Industry Practice; Artistic Practice; Procedural Rhetoric) (See also: AI-SEP)

Illustrative Learning Outcomes:

KA Core:

1. Recall the characteristics of common general-purpose graphics processing architectures.
2. Identify the key stages of the immediate-mode rendering pipeline.
3. Describe the key constraints a specific game platform will likely impose on developers.
4. Explain how eSports are streamed to large audiences over the internet.
5. Translate complex mathematical functions into performant source code.
6. Use an industry-standard graphics API to render a 3D model in a virtual scene.
7. Modify a shader to change a visual effect according to stated requirements.
8. Implement a game for a particular platform according to the specification.
9. Optimize a function for processing collision detection in a simulated environment.
10. Assess a game's run-time and memory performance using an industry-standard tool and development environment.
11. Compare the interfaces of different game platforms, highlighting their respective implications for human-computer interaction.
12. Recommend an appropriate set of development tools and techniques for implementing a game of a particular genre for a given platform.
13. Discuss the key challenges in making a digital game that is cross-platform compatible.
14. Express how game developers can enhance the accessibility of a game interface.
15. Create novel forms of gameplay using frontier game platforms.

SPD-Interactive: Interactive Computing Platforms

Non-core:

1. Data Analysis Platforms
    a. Jupyter notebooks; Google Colab; R; SPSS; Observable.
    b. Cloud SQL/data analysis platforms (e.g., BigQuery) (See also: DM-Querying)
        i. Apache Spark
        ii. Data Visualizations (See also: GIT-Visualization)
    c. Interactive presentations backed by data
    d. Design tools requiring low-latency feedback loops
        i. Rendering tools
        ii. Graphic design tools
2. Prompt programming
    a. Generative AI (e.g., OpenAI's ChatGPT, OpenAI's Codex, GitHub's Copilot) and LLMs are accessed/interacted

3. Quantum Platforms (See also: AR-Quantum)
    a. Program quantum logic operators in quantum machines.
    b. Use API for available quantum services
    c. Signal analysis/Fourier analysis/Signal processing (for music composition, audio/RF analysis) (See also: GIT-Image)

Illustrative Learning Outcomes:
Non-core:
1. Analyze large datasets interactively.
2. Create a backing track for a musical performance, such as live coding.
3. Create compelling computational notebooks that construct a narrative for a given journalistic goal/story.
4. Implement interactive code that uses a dataset and generates exploratory graphics.
5. Create a program that performs a task using LLM systems.
6. Contrast a program developed by an AI platform and by a human.
7. Implement a system that interacts with a human without using a screen.
8. Contextualize the attributes of different data analysis styles, such as interactive vs engineered pipeline.
9. Write a program using a notebook computing platform (e.g., searching, sorting, or graph manipulation).
10. Demonstrate a quantum gate outcome using a quantum platform.

SPD-SEP/Mobile

Non-core:
1. Privacy and data protection
2. Accessibility in mobile design
3. Security and cybersecurity
4. Social impacts of mobile technology
5. Ethical use of AI and algorithms

Illustrative Learning Outcomes:
Non-core:
1. Understand and uphold ethical responsibilities for safeguarding user privacy and data protection in mobile applications.
2. Design mobile applications with accessibility in mind, ensuring effective use by people with disabilities.
3. Demonstrate proficiency in secure coding practices to mitigate risks associated with various security threats in mobile development.
4. Analyze the broader social impacts of mobile technology, including its influence on communication patterns, relationships, and mental health.
5. Comprehend the ethical considerations of using AI in mobile applications, ensuring unbiased and fair algorithms.

SPD-SEP/Web

Non-core:

1. Privacy concerns with mobile apps
2. Designing for inclusivity and accessibility
3. Ethical use of AI in mobile apps
4. Sustainable app development and server hosting
5. Avoiding spam or intrusive notifications
6. Addressing cyberbullying and harassment
7. Promoting positive online communities
8. Monetization and advertising
9. Ethical use of gamification

Illustrative Learning Outcomes:

Non-core:

1. Understand how mobile computing impacts communications and the flow of information within society.
2. Design mobile apps that have made daily tasks easier/faster.
3. Recognize how the ubiquity of mobile computing has affected work-life balance.
4. Understand how mobile computing impacts health monitoring and healthcare services.
5. Define how mobile apps are used to educate about and help achieve UN sustainability goals.

SPD-SEP/Game

Non-core:

1. Intellectual Property Rights in Creative Industries
    a. Intellectual Property Ownership: copyright, trademark; design right, patent, trade secret, civil versus criminal law; international agreements; procedural content generation and the implications of generative artificial intelligence
    b. Licensing: Usage and fair usage exceptions; open-source license agreements; proprietary and bespoke licensing; enforcement
2. Fair Access to Play
    a. Game Interface Usability: user requirements, affordances, ergonomic design, user research, experience measurement, and heuristic evaluation methods for games
    b. Game Interface Accessibility: forms of impairment and disability; means to facilitate game access; universal design; legislated requirements for game platforms; compliance evaluation; challenging game mechanics and access
3. Game-Related Health and Safety
    a. Injuries in Play: ways of mitigating common upper body injuries, such as repetitive strain injury; exercise psychology and physiotherapy in eSports
    b. Risk Assessment for Events and Manufacturing: control of substances hazardous to health (COSHH); fire safety; electrical and electronics safety; risk assessment for games and game events; risk assessment for manufacturing
    c. Mental Health: motivation to play; gamification and gameful design; game psychology – internet gaming disorder

4. Platform Hardware Supply Chain and Sustainability
    a. Platform Lifecycle: platform composition – materials, assembly; mineral excavation and processing; power usage; recycling; planned obsolescence.
    b. Modern Slavery: supply chains; forced labor and civil rights; working conditions; detection and remission; certification bodies and charitable endeavors.
5. Representation in the Media and Industry
    a. Inclusion: identity and identification; inclusion of a broad range of characters for diverse audiences; media representation and its effects; media literacy; content analysis; stereotyping; sexualization
    b. Equality: histories and controversies, such as gamergate, quality of life in the industry, professional discourse and conduct in business contexts, pathways to game development careers, social mobility, the experience of developers from different backgrounds and identities, gender, and technology

Illustrative Learning Outcomes:

Non-core:

1. Discuss how creators can protect their intellectual property.
2. Identify common pitfalls in game interfaces that exclude players with impaired or non-functional vision.
3. Describe how heuristic evaluation can be used to identify usability problems in game interfaces.
4. Explain why upper body injuries are common in eSports.
5. Discuss how to reform characters and dialogues in a scene to reduce stereotype threat.
6. Illustrate how the portrayal of race in a game can influence the risk of social exclusion in the associated online community around the game.
7. Modify a policy for a LAN party event to include mitigations that lower the risk of fire.
8. Design a gamification strategy to motivate serious play for an awareness-raising game.
9. Analyze the role of company hiring policies and advocacy on social mobility.
10. Assess the appropriateness of two manufacturers for producing a new game console.
11. Compare options for open-source licensing of a game development tool.
12. Recommend changes to a specific game interface to improve access to players who are deaf or whose hearing is otherwise impaired.
13. Discuss whether games are addictive.
14. Suggest how the portrayal of women in video games influences how players perceive members of those groups.
15. Create a video game that successfully advocates for climate science.

SPD-SEP/Robotics

Non-core:

1. Fairness, transparency, and accountability in robotic algorithms
2. Mitigating biases in robot decision-making
3. Public safety in shared spaces with robots
4. Compliance with data protection laws
5. Patient consent and trust in medical robots

Illustrative Learning Outcomes:
Non-core:

1. Identify instances of bias in robotic algorithms and propose strategies to mitigate them.
2. Evaluate and critique robotic systems for ethical and fairness considerations, suggesting improvements where necessary.
3. Analyze real-world examples of biases in robot decision-making and develop strategies to reduce bias in robotic systems.
4. Assess the potential risks associated with robots in public spaces and propose safety measures to mitigate those risks.
5. Evaluate the impact of patient consent and trust on the effectiveness of medical robot deployments in healthcare contexts.

SPD-SEP/Interactive

Non-core:

1. Ethical guidelines when using AI models to assist in journalism and content creation
2. Accountability for AI-generated outputs
3. Behavior among prompt programmers and AI developers
4. Trust with the public when using AI models

Illustrative Learning Outcomes:
Non-core:

1. Indicate a framework for accountability in AI model deployment, including clear documentation and attribution.
2. Discuss ethical codes of conduct and professional standards relevant to prompt programming and AI development.
3. Create communication plans and materials to educate the public about AI capabilities, limitations, and ethical safeguards.

Professional Dispositions

- **Self-Directed**: Students should be able to learn new platforms and languages with a growth-oriented mindset and thrive in dynamic environments, while continually enhancing skills.
- **Inventive:** Students should demonstrate excellence in designing software architecture within unconventional constraints, emphasizing adaptability and creative problem-solving for innovative solutions.
- **Adaptable**: Students should adapt to diverse challenges, showing resilience, open-mindedness, and a proactive approach to changing requirements and constraints.

Mathematics Requirements

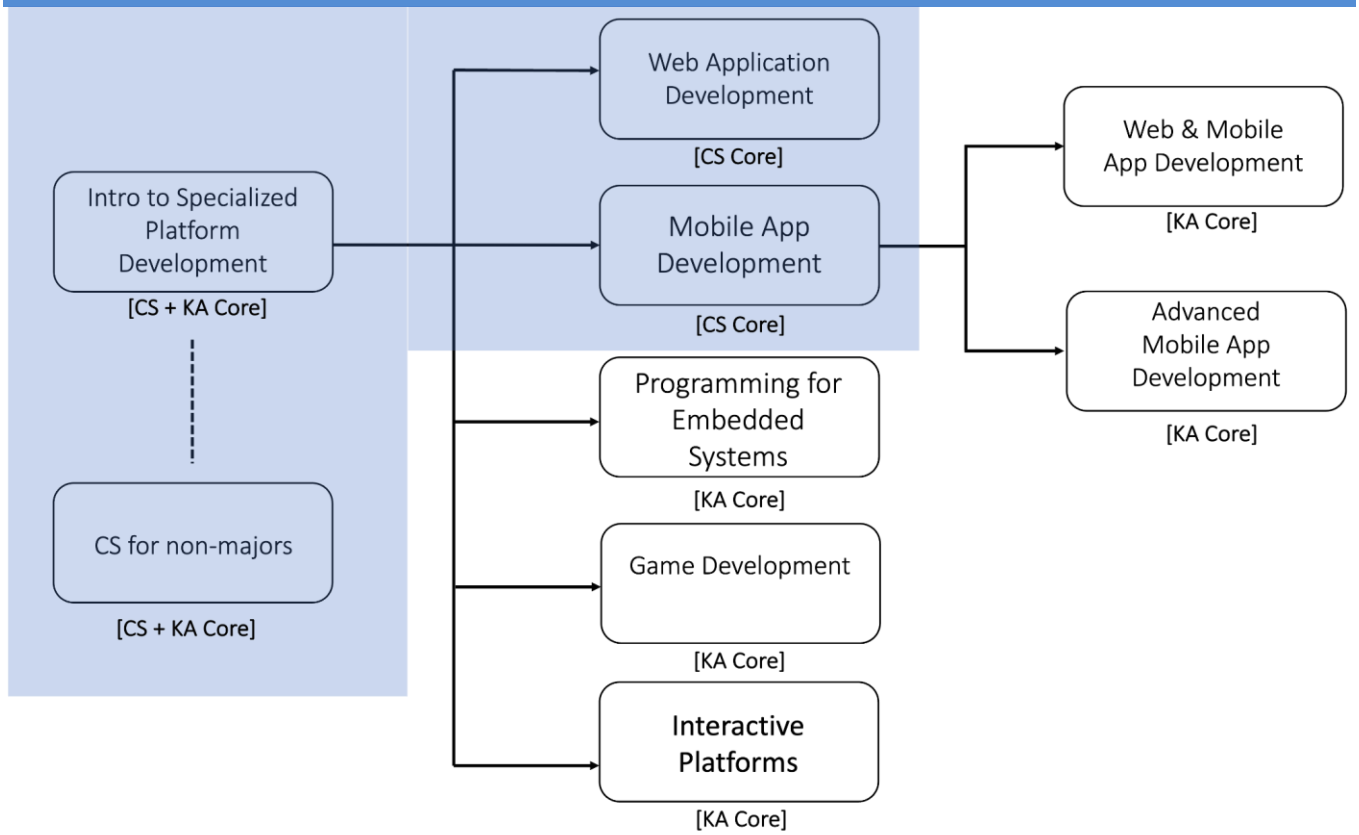
Required:

- MSF-Discrete

Desired:

- MSF-Calculus
- MSF-Linear
- MSF-Statistics

Course Packaging Suggestions



Introduction to Special Platform Development Course to include the following:

- SPD-Common (10 hours)
- SPD-Web (4 hours)
- SPD-Mobile (4 hours)
- SPD-SEP/Robotics (3 hours)
- SPD-Embedded (3 hours)
- SPD-Game (4 hours)
- SPD-SEP/Interactive (2 hours)
- SEP-Context (2 hours)
- SDF-Practices (4 hours)
- SE-Design (2 hours)
- FPL-Scripting (2 hours)

Course objectives: Students should be able to grasp common aspects of platform development, acquire foundational knowledge in web development, and attain proficiency in web techniques. They will apply comprehensive mobile development skills and explore challenges in robotics platforms.

Expertise in developing platforms for embedded systems, along with skills in game development and creating interactive platforms, will be developed. Students will analyze societal, ethical, and professional implications of platform development, fostering a well-rounded understanding of this field within a concise curriculum.

Mobile Development Course to include the following:

- SPD-Common: Common Aspects (3 hours)
- SPD-Mobile (25 hours)
- SDF-Practices: Software Development Practices (2 hours)
- SE-Design: Software Design (3 hours)
- SE-Construction: Software Construction (2 hours)
- SPD-SEP/Mobile (3 hours)

Course objectives: Students should be able to design, develop, and deploy cross-platform mobile applications using languages like Java, Kotlin, Swift, or React Native. Proficiency in implementing user experience best practices, exploring cross-platform development tools, and utilizing platform-specific APIs for seamless integration is emphasized. The course covers security vulnerability identification, testing methodologies, and distribution/versioning of mobile applications. Students gain insights into user behavior and application performance through analytics tools. Additionally, they learn version control, release management, and ethical considerations relevant to mobile development, providing a well-rounded skill set for successful and responsible mobile application development across diverse platforms.

Web Development Course to include the following:

- SPD-Web (19 hours)
- FPL-OOP (3 hours)
- SE-Construction (2 hours)
- DM-Querying (2 hours)
- SE-Tools (4 hours)
- SDF-Practices (4 hours)
- SE-Design (2 hours)
- FPL-Scripting (2 hours)
- SPD-SEP/Web (2 hours)

Course objectives: Students should be able to gain expertise in designing, developing, and deploying modern web applications. The curriculum covers key concepts, ensuring proficiency in HTML, CSS, and JavaScript for responsive and visually appealing pages. Students explore and implement frontend frameworks (e.g., React, Angular) for efficient development, understand server-side languages (e.g., Node.js, Python) for dynamic applications, and design effective architectures prioritizing scalability and security. They learn version control (e.g., GIT), integrate APIs for enhanced functionality, implement responsive design, optimize for performance, and ensure security through best practices. Testing, debugging, accessibility, deployment, and staying current with industry trends are also emphasized.

Game Development Course to include the following:

- SPD-Game (16 hours)
- SPD-SEP/Game (4 hours)
- SDF-Practices (4 hours)
- GIT-Interaction (1 hour)
- HCI-Design (3 hours)
- HCI-User (1 hour)
- SE-Tools (1 hour)
- AL-Foundational (2 hours)
- GIT-Rendering (4 hours)
- SE-Design (4 hours)

Course objectives: Students should be able to master designing, developing, and deploying interactive games. The curriculum covers fundamental game design principles, proficiency in languages like C++, C#, or Python, and utilization of popular engines such as Unity or Unreal. Students gain 3D modeling and animation skills, implement physics and simulations for realism, and create AI algorithms for intelligent non-player characters. They design multiplatform games, optimize UI/UX for engagement, apply game-specific testing and debugging techniques, integrate audio effectively, and explore industry monetization models. The course emphasizes ethical considerations, ensuring students analyze and address content, diversity, and inclusivity in game development.

Committee

Chair: Christian Servin, El Paso Community College, El Paso, TX, USA
Members:

- Sherif G. Aly, The American University in Cairo, Cairo, Egypt
- Yoonsik Cheon, The University of Texas at El Paso, El Paso, TX, USA
- Eric Eaton, University of Pennsylvania, Philadelphia, PA, USA
- Claudia L. Guevara, Jochen Schweizer mydays Holding GmbH, Munich, Germany
- Larry Heimann, Carnegie Mellon University, Pittsburgh, PA, USA
- Amruth N. Kumar, Ramapo College of New Jersey, Mahwah, NJ, USA
- R. Tyler Pirtle, Google, USA
- Michael James Scott, Falmouth University, Falmouth, Cornwall, UK

Contributors:

- Sean R. Piotrowski, Rider University, Lawrenceville, NJ, USA
- Mark O'Neil, Blackboard Inc., Newport, NH, USA
- John DiGennaro, Qwickly, Cleveland, OH, USA
- Rory K. Summerley, London South Bank University, London, England, UK

Core Topics Table

In the following seventeen tables, CS and KA core topics have been listed, one table per knowledge area. For each topic, desired skill levels have been identified and used to estimate the time needed for the instruction of CS Core and KA Core topics. The skill levels should be treated as recommended, not prescriptive. The time needed to cover CS Core and KA Core topics is expressed in terms of **instructional hours**. Instructional hours are hours spent in the classroom imparting knowledge regardless of the pedagogy used. Students are expected to spend additional time after class practicing related skills and exercising professional dispositions.

For convenience, the tables have been listed under three competency areas: Software, Systems, and Applications. The tables on Society, Ethics, and the Profession (SEP) and Mathematical and Statistical Foundations (MSF) are listed last as crosscutting topics that apply to all the competency areas.

Software Competency Area

The core topics in Software Development Fundamentals (SDF) and Algorithmic Foundations (AL) typically constitute the introductory course sequence in computer science and have been listed first.

	Knowledge Area	Knowledge Units	CS Core	KA Core
SDF	Software Development Fundamentals	5	43	
AL	Algorithmic Foundations	5	32	32
FPL	Foundations of Programming Languages	22	21	19
SE	Software Engineering	9	6	21
	Total		102	72

SDF: Software Development Fundamentals

KU	Topic	Skill Level	Core	Hours
SDF-Fundamentals	1. Basic concepts such as variables, primitive data types, and expression evaluation 2. How imperative programs work: state and state transitions on execution of statements, flow of control 3. Basic constructs such as assignment statements, conditional and iterative	Develop	CS	18

	statements, basic I/O 4. Key modularity constructs such as functions and related concepts like parameter passing, scope, abstraction, data encapsulation, etc. 5. Input and output using files and APIs 6. Structured data types available in the chosen programming language like sequences, associative containers, others and when and how to use them 7. Libraries and frameworks provided by the language (when/where applicable) 8. Recursion			
SDF-Fundamentals	9. Dealing with runtime errors in programs 10. Basic concepts of programming errors, testing, and debugging 11. Documenting/commenting code at the program and module level	Evaluate Apply	CS	2
SDF-Data-Structures	Standard abstract data types such as lists, stacks, queues, sets, and maps/dictionaries and operations on the data types 4. Strings and string processing	Develop	CS	10
SDF-Data-Structures	Selecting and using appropriate data structures Performance implications of choice of data structure(s)	Evaluate	CS	2
SDF-Algorithms	Concept of algorithm and notion of algorithm efficiency Some common algorithms (e.g., sorting, searching, tree traversal, graph traversal) Impact of algorithms on time/space efficiency of programs	Explain	CS	6
SDF-Practices	Basic testing including test case design Specifying functionality of a module in a natural language	Develop	CS	3
SDF-Practices	3. Programming style that improves readability	Evaluate	CS	1
SDF-Practices	Use of a general-purpose IDE, including its debugger	Apply	CS	1

AL: Algorithmic Foundations

KU	Topic	Skill Level	Core	Hours
AL-Foundational AL-Complexity	2. Arrays 1. Abstract Data Types and Operations 2b i. Foundational Complexity Classes: Constant		CS	1
AL-Foundational AL-Complexity AL-Strategies	11a. Search Algorithms O(n) (e.g., linear array search) 2b iii. Foundational Complexity Classes: Linear 1a. Brute Force	Apply Evaluate Explain	CS	1
AL-Foundational AL-Complexity AL-Strategies	12a. Sorting O(n2), (e.g., selection sort of an array) 2b v. Foundational Complexity classes: Quadratic 1a. Brute Force	Apply Evaluate Explain	CS	1
AL-Foundational AL-Complexity AL-Strategies	11b. Search $O(\log_2 n)$, (e.g., Binary search of an array) 2b ii. Foundational Complexity Classes: *Logarithmic* 1b ii. Decrease-and-Conquer	Apply Evaluate Explain	CS	1
AL-Foundational AL-Complexity AL-Strategies	12b. Sorting $O(n \log n)$, (e.g., Quick, Merge, Tim) 2b iv. Foundational Complexity Classes: *Log Linear* 1c. Divide-and-Conquer	Apply Evaluate Explain	CS	1
AL-Foundational AL-Complexity AL-Strategies	4. Linked Lists 1. Abstract Data Types and Operations 11a. Search $O(n)$, (e.g., linear linked list search) 2b iii. Foundational Complexity Classes: *Linear* 1a. Brute Force	Explain Apply Apply Evaluate Explain	CS	1
AL-Foundational AL-Complexity AL-	5. Stacks 1. Abstract Data Types and Operations 2b i. Foundational complexity classes: *Constant* 6. Queues and Deques 1. Abstract Data Types and Operations	Explain, Apply Explain, Apply	CS	1

Foundational				
AL-Foundational AL-Complexity AL-Strategies	7. Hash Tables/Maps 7a. Collision resolution and complexity 1. Abstract Data Types and Operations 2b i. Foundational complexity classes: *Constant* 1f. Time vs Space tradeoff	Explain Explain Apply Explain Explain	CS	1 1
AL-Foundational AL-Strategies AL-Foundational AL-Strategies	9. Trees 1. Abstract Data Types and Operations 11c. Search Algorithms DFS/BFS 2b. Decrease-and-Conquer 9b. Balanced Trees (e.g., AVL, 2-3, Red-Black, Heap) 1e ii. Transform-and-Conquer: Representation Change (e.g., heapsort)	Explain Apply Apply Explain Apply Explain	CS	1 3
AL-Foundational AL-Foundational AL-Strategies	8. Graphs (e.g., [un]directed, [a]cyclic, [un]connected, [un]weighted) 8a. Representation: Adjacency List vs Matrix 13. Graph Algorithms 13a. Shortest Path (e.g., Dijkstra's, Floyd's) 13b. Minimal, spanning tree (e.g., Prim's, Kruskal's) 1d. Greedy 1e. iv. Dynamic Programming	Explain Apply Apply Explain	CS	1 4
AL-Foundational	1. Abstract Data Types and Operations 3. Records/Structures/Tuples and Objects 10. Sets	Explain	CS	1
AL-Strategies AL-Strategies AL-Strategies AL-Strategies	1. Paradigms (demonstrated in AL-Foundational) 1a. Brute-Force 1b. Decrease-and-Conquer 1c. Divide-and-Conquer 1f. Time-Space Tradeoff 3. Iteration vs Recursion 1e. Transform-and-Conquer 1e i. Instance Simplification (e.g., pre-sorting) 1e iii. Problem Reduction (e.g., least-common-multiple) 2. Handling Exponential Growth (e.g., heuristic A*, backtracking, ranch-and-bound 1e iv. Dynamic Programming	Explain Explain Explain Explain	CS	3 1 1 1

	(e.g., Bellman-Ford, Knapsack, Floyd, Warshall)			
AL-Complexity	1. Complexity Analysis Framework	Explain	CS	1
	2. Asymptotic Complexity Analysis 2a. Big O, Big Omega, and Big Theta 2b. Foundational complexity classes	Explain		
	demonstrated by AL-Foundational algorithms (with complexity): *Constant, Logarithmic, Linear, Log Linear, Quadratic*, and *Cubic*	Evaluate		1
AL-Complexity	4. Tractability and Intractability 4a. P, NP, and NP-C complexity classes 4b. NP-Complete Problems (e.g., SAT, Knapsack, TSP) 4c. Reductions	Explain		2
AL-Strategies	1a. Paradigms: Exhaustive brute force 1e. iv. Dynamic Programming	Explain		1
AL-Complexity	2 vii. Foundational Complexity Classes: *Exponential* 2b viii. Factorial complexity classes: Factorial *O*(n!) (e.g. All Permutations, Hamiltonian Circuit)	Explain		1
AL-Models	1. Formal Automata 1a. Finite State Automata 2a. Regular language, grammar, and expressions	Explain, Apply	CS	1
	1b. Pushdown Automata 2b. Context-Free language and grammar	Explain, Apply		1
	1d. Turing Machine	Explain		3
	2d. Recursively Enumerable language and grammar 1c. Linear-Bounded	Explain		
	2c. Context-Sensitive language and grammar			1
	2. Formal Languages and Grammars	Explain		2
	4. Decidability, Computability, Halting problem	Explain		
	5. The Church-Turing Thesis	Explain		1
	6. Algorithmic Correctness Invariants (e.g., in iteration, recursion, sorting, heaps)	Explain		
AL-SEP	1. Social, Ethical, and Secure Algorithms 2. Algorithmic Fairness (e.g., differential privacy) 3. Accountability/Transparency 4. Responsible algorithms 5. Economic and other impacts of algorithms 6. Sustainability	Explain	CS	In SEP Hours

FPL: Foundations of Programming Languages

KU	Topic	Skill Level	Core	Hours
FPL-OOP	1. Imperative programming as a subset of object-oriented programming 2. Object-oriented design a. Decomposition into objects carrying state and having behavior b. Class-hierarchy design for modeling 3. Definition of classes: fields, methods, and constructors 4. Subclasses, inheritance (including multiple inheritance), and method overriding 5. Dynamic dispatch – definition of method-call 6. Exception handling 7. Object-oriented idioms for encapsulation a. Privacy, data hiding, and visibility of class members b. Interfaces revealing only method signatures c. Abstract base classes, traits and mixins 8. Dynamic vs static properties 9. Composition vs inheritance 10. Subtyping a. Subtype polymorphism; implicit upcasts in typed languages b. Notion of behavioral replacement – subtypes acting like supertype c. Relationship between subtyping and inheritance	Develop	CS	1. 5
	11. Collection classes, iterators, and other common library components 12. Metaprogramming and reflection	Develop	KA	1
FPL-Functional	1. Lambda expressions and evaluation a. Variable binding and scope rules b. Parameter-passing c. Nested lambda expressions and reduction order 2. Effect-free programming a. Function calls have no side effects, facilitating compositional reasoning. b. Immutable variables and data copying vs reduction	Develop	CS	4

	c. Use of recursion vs loops vs pipelining (map/reduce) 3. Processing structured data (e.g., trees) via functions with cases for each data variant a. Functions defined over compound data in terms of functions applied to the constituent pieces b. Persistent data structures 4. Using higher-order functions (taking, returning, and storing functions)			
	5. Metaprogramming and reflection 6. Function closures (functions using variables in the enclosing lexical environment) a. Basic meaning and definition – creating closures at run-time by capturing the environment b. Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments c. Using a closure to encapsulate data in its environment d. Delayed vs eager evaluation	Explain	KA	3
FPL-Logic	1. Universal vs existential quantifiers 2. First order predicate logic vs higher order logic 3. Expressing complex relations using logical connectives and simpler relations 4. Definitions of Horn clause, facts, goals and subgoals 5. Unification and unification algorithm; unification vs assertion vs expression evaluation 6. Mixing relations with functions 7. Cuts, backtracking, and non-determinism 8. Closed-world vs open-world assumptions	Explain	KA	3
FPL-Scripting	1. Error/exception handling 2. Piping 3. System commands a. Interface with operating systems 4. Environment variables 5. File abstraction and operators 6. Data structures, such as arrays and lists 7. Regular expressions 8. Programs and processes 9. Workflow	Develop	CS	2
FPL-Event-Driven	1. Procedural programming vs reactive programming – advantages of reactive programming in capturing	Develop	CS	2

	events 2. Components of reactive programming – event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers 3. Stateless and state-transition models of event-based programming 4. Canonical uses such as GUIs, mobile devices, robots, servers			
	5. Using a reactive framework a. Defining event handlers/listeners b. Parameterization of event senders and event arguments c. Externally generated events and program-generated events 6. Separation of model, view, and controller 7. Event-driven and reactive programs as state-transition systems	Develop	KA	2
FPL-Parallel	1. Safety and liveness a. Race conditions. b. Dependencies/preconditions c. Fault models d. Termination 2. Programming models: One or more of the following a. Actor models b. Procedural and reactive models c. Synchronous/asynchronous programming models d. Data parallelism 3. Properties a. Order-based properties i. Commutativity ii. Independence b. Consistency-based properties i. Atomicity ii. Consensus 4. Execution control: a. Async await b. Promises c. Threads 5. Communication and coordination a. Mutexes b. Message-passing	Develop	CS	3

	c. Shared memory d. Cobegin-coend e. Monitors f. Channels g. Threads h. Guards			
	6. Futures 7. Language support for data parallelism such as forall, loop unrolling, map/reduce 8. Effect of memory-consistency models on language semantics and correct code generation 9. Representational State Transfer Application Programming Interfaces (REST APIs) 10. Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing 11. Overheads of message-passing 12. Granularity of program for efficient exploitation of concurrency 13. Concurrency and other programming paradigms (e.g., functional)	Explain	KA	2
FPL-Types	1. A type as a set of values together with a set of operations a. Primitive types (e.g., numbers, Booleans) b. Compound types built from other types (e.g., records/structs, unions, arrays, lists, functions, references using set operations) 2. Association of types to variables, arguments, results, and fields 3. Type safety as an aspect of program correctness 4. Type safety and errors caused by using values inconsistently given their intended types 5. Goals and limitations of static and dynamic typing a. Detecting and eliminating errors as early as possible 6. Generic types (parametric polymorphism) a. Definition and advantages of polymorphism – parametric, subtyping, overloading, and coercion b. Comparison of monomorphic and polymorphic types c. Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism	Develop	CS	3

	d. Generic parameters and typing e. Use of generic libraries such as collections f. Comparison with ad hoc polymorphism (overloading) and subtype polymorphism g. Prescriptive vs descriptive polymorphism h. Implementation models of polymorphic types i. Subtyping			
	7. Type equivalence – structural vs name equivalence 8. Complementary benefits of static and dynamic typing a. Errors early vs errors late/avoided b. Enforce invariants during code development and code maintenance vs postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections c. Typing rules i. Rules for function, product, and sum types d. Avoid misuse of code vs allow more code reuse e. Detect incomplete programs vs allow incomplete programs to run f. Relationship to static analysis g. Decidability	Develop	KA	4
FPL-Systems	1. Data structures for translation, execution, translation, and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string 2. Direct, indirect, and indexed access to memory location 3. Run-time representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects 4. Abstract low-level machine with simple instruction, stack, and heap to explain translation and execution 5. Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap a. Translating selection and iterative constructs to control-flow diagrams b. Translating control-flow diagrams to low level abstract code c. Implementing loops, recursion, and tail calls d. Translating function/procedure calls and return	Develop	CS	3

	from calls, including different parameter-passing mechanisms using an abstract machine 6. Memory management a. Low level allocation and accessing of high-level data structures such as basic data types, n-dimensional array, vector, record, and objects b. Return from procedure as automatic deallocation mechanism for local data elements in the stack c. Manual memory management: allocating, de-allocating, and reusing heap memory d. Automated memory management – garbage collection as an automated technique using the notion of reachability 7. Green computing			
FPL-Translation	1. Execution models for JIT (Just-In-Time), compiler, interpreter 2. Use of intermediate code, e.g., bytecode. 3. Limitations and benefits of JIT, compiler, and interpreter 4. Cross compilers/transpilers 5. BNF and extended BNF representation of context-free grammar 6. Parse tree using a simple sentence such as arithmetic expression or if-then-else statement 7. Execution as native code or within a virtual machine 8. Language translation pipeline – syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution	Explain	CS	2
	9. Run-time representation of core language constructs such as objects (method tables) and functions that can be passed as parameters to and returned from functions (closures) 10. Secure compiler development	Explain	KA	2
FPL-Abstraction	1. BNF and regular expressions 2. Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators 3. Components of a language a. Definitions of alphabets, delimiters, sentences,	Explain	KA	3

	syntax, and semantics b. Syntax vs semantics 4. Program as a set of non-ambiguous meaningful sentences 5. Basic programming abstractions – constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling 6. Mutable vs immutable variables: advantages and disadvantages of reusing existing memory location vs advantages of copying and keeping old values; storing partial computation vs recomputation 7. Types of variables – static, local, nonlocal, global; need and issues with nonlocal and global variables 8. Scope rules – static vs dynamic; visibility of variables; side-effects 9. Side-effects induced by nonlocal variables, global variables, and aliased variables			

SE: Software Engineering

KU	Topic	Skill Level	Core	Hours
SE-Teamwork	1. Effective communication 2. Common causes of team conflict, and approaches for conflict resolution 3. Cooperative programming 4. Roles and responsibilities in a software team 5. Team processes 6. Importance of team diversity and inclusion	Evaluate	CS	2
SE-Teamwork	7. Interfacing with stakeholders, as a team 8. Risks associated with physical, distributed, hybrid and virtual teams	Explain	KA	2
SE-Tools	1. Software configuration management and version control	Evaluate	CS	1
SE-Tools	2. Release management 3. Testing tools including static and dynamic analysis tools			

	4. Software process automation 5. Design and communication tools (docs, diagrams, common forms of design diagrams like UML) 6. Tool integration concepts and mechanisms 7. Use of modern IDE facilities – debugging, refactoring, searching/indexing, etc.	Explain	KA	3
SE-Requirements	1. Describe functional requirements using, for example, use cases or user stories 2. Properties of requirements including consistency, validity, completeness, and feasibility 3. Requirements elicitation 4. Non-functional requirements, for example, security, usability, or performance (aka Quality Attributes) 5. Risk identification and management 6. Communicating and/or formalizing requirement specifications	Apply	KA	2
SE-Design	1. System design principles 2. Software architecture 3. Programming in the large vs programming in the small 4. Code smells and other indications of code quality, distinct from correctness.	Explain	CS	1
SE-Design	5. API design principles 6. Identifying and codifying data invariants and time invariants 7. Structural and behavioral models of software designs 8. Data design 9. Requirement traceability	Apply	KA	4
SE-Construction	1. Practical small-scale testing 2. Documentation	Apply	CS	1
SE-Construction	3. Coding Style 4. “Best Practices” for coding 5. Debugging 6. Logging 7. Use of libraries and frameworks developed by others	Apply	KA	3
SE-Validation	1. Verification and validation concepts 2. Why testing matters	Explain	CS	1

	3. Testing objectives 4. Test kinds 5. Stylistic differences between tests and production code			
SE-Validation	6. Test planning and generation 7. Test development (see SDF) 8. Verification and validation in the development cycle 9. Domain specific verification and validation challenges	Explain	KA	4
SE-Refactoring	1. Hyrum's Law 2. Backward Compatibility 3. Refactoring 4. Versioning	Explain	KA	1
SE-Reliability	1. Concept of reliability 2. Identifying reliability requirements (see SEP) 3. Software failures vs defect injection/detection 4. Software reliability, system reliability and failure behavior (cross-reference SF/Reliability Through Redundancy) 5. Defect injection and removal cycle, and different approaches for defect removal 6. Compare the "error budget" approach to reliability with the "error-free" approach, and identify domains where each is relevant	Explain	KA	4

Systems Competency Area

The core topics in Architecture and Organization (AR) and Operating Systems (OS) are typically covered early in the curriculum and have been listed first. Data Management (DM) and Security (SEC) topics listed in this section can be applied to all three competency areas.

	Knowledge Area	Knowledge Units	CS Core	KA Core
AR	Architecture and Organization	9	9	16
OS	Operating Systems	14	8	13
NC	Networking and Communication	8	7	24

PDC	Parallel and Distributed Computing	5	9	26
SF	Systems Fundamentals	8	18	8
DM	Data Management	12	10	26
SEC	Security	6	6	35
	Total		67	148

AR: Architecture and Organization

KU	Topic	Skill Level	Core	Hours
AR-Logic	1. Combinational vs sequential logic/field programmable gate arrays (FPGAs) a. Fundamental combinational b. Sequential logic building block	Explain	KA	3
	2. Computer-aided design tools that process hardware and architectural representations 3. High-level synthesis a. Register transfer notation b. Hardware description language (e.g., Verilog/VHDL/Chisel) 4. System-on-chip (SoC) design flow 5. Physical constraints a. Gate delays b. Fan-in and fan-out c. Energy/power d. Speed of light	Evaluate		
AR-Representation	1. Overview and history of computer architecture 2. Bits, bytes, and words 3. Unsigned, signed and two's complement representations 4. Numeric data representation and number bases a. Fixed-point b. Floating-point 5. Representation of non-numeric data 6. Representation of records, arrays and UTF data types	Apply	CS	1

AR-Assembly	1. von Neumann machine architecture 2. Control unit: instruction fetch, decode, and execution 3. Introduction to SIMD vs MIMD and the Flynn taxonomy 4. Shared memory multiprocessors/multicore organization	Explain	CS	1
AR-Assembly	5. Instruction set architecture (ISA) (e.g., x86, ARM, and RISC-V) a. Instruction formats b. Data manipulation, control, I/O c. Addressing modes d. Machine language programming e. Assembly language programming 6. Subroutine call and return mechanisms 7. I/O and interrupts 8. Heap, static, stack, and code segments	Develop	KA	2
AR-Memory	1. Memory hierarchy: the importance of temporal and spatial locality 2. Main memory organization and operations 3. Persistent memory (e.g., SSD, standard disks) 4. Latency, cycle time, bandwidth, and interleaving 7. Virtual memory (hardware support) 8. Fault handling and reliability 9. Reliability a. Error coding b. Data compression c. Data integrity 10. In-Memory Processing (PIM)	Explain	CS	6
	5. Cache memories a. Address mapping b. Block size c. Replacement and store policy 6. Multiprocessor cache coherence	Evaluate		
	1. I/O fundamentals a. Handshaking and buffering b. Programmed I/O c. Interrupt-driven I/O 2. Interrupt structures: vectored and prioritized, interrupt acknowledgment	Explain	CS	1

AR-IO	3. I/O devices (e.g., mouse, keyboard, display, camera, sensors, actuators) 4. External storage, physical organization, and drives 5. Bus fundamentals a. Bus protocols b. Arbitration c. Direct-memory access (DMA)			
AR-Organization	1. Implementation of simple datapaths, including instruction pipelining, hazard detection, and resolution 2. Control unit a. Hardwired implementation b. Microprogrammed realization	Develop	KA	2
	3. Instruction pipelining 4. Introduction to instruction-level parallelism (ILP)	Explain		
AR-Performance-Energy	1. Performance-energy evaluation (introduction): performance, power consumption, memory, and communication costs 2. Branch prediction, speculative execution, out-of-order execution, Tomasulo's algorithm	Evaluate	KA	2
AR-Performance-Energy	3. Enhancements for vector processors and GPUs 4. Hardware support for multithreading a. Race conditions b. Lock implementations c. Point-to-point synchronization d. Barrier implementation 5. Scalability 6. Alternative architectures, such as VLIW/EPIC, accelerators, and other special-purpose processors 7. Dynamic voltage and frequency scaling (DVFS) 8. Dark Silicon	Explain	KA	1
AR-Heterogeneity	1. SIMD and MIMD architectures (e.g., General-Purpose GPUs, TPUs, and NPUs) 2. Heterogeneous memory system a. Shared memory versus distributed memory b. Volatile vs non-volatile memory c. Coherence protocols 3. Domain-Specific Architectures (DSAs) a. Machine Learning Accelerator	Explain	KA	2

	b. In-networking computing c. Embedded systems for emerging applications d. Neuromorphic computing e. Edge computing devices 4. Packaging and integration solutions such as 3DIC and chiplets 5. Machine learning in architecture design a. AI algorithms for workload analysis b. Optimization of architecture configurations for performance and power efficiency			
AR-Security	1. Principles of Secure Hardware a. Security Risk Analysis, Asset Protection, and Threat Model b. Cryptographic Acceleration with Hardware c. Support for virtualization (e.g., OS isolation) 2. Roots of trust in hardware, Physically Unclonable Functions (PUF) 3. Hardware Random Number Generators 4. Memory protection extensions a. Runtime pointer bounds checking (e.g., buffer overflow) b. Protection at the microarchitectural level c. Protection at the ISA level 5. Trusted Execution Environment (TEE) a. Trusted Computer Base Protections b. Protecting virtual machines c. Protecting containers d. Trusted software modules (Enclaves)	Explain	KA	2
AR-Quantum	1. Principles a. The wave-particle duality principle b. The uncertainty principle in the double-slit experiment c. What is a Qubit? Superposition and measurement; Photons as qubits d. Systems of two qubits; Entanglement; Bell states; The No-Signaling theorem 2. Axioms of QM: superposition principle, measurement axiom, unitary evolution 3. Single qubit gates for the circuit model of quantum computation: X, Z, H.	Explain	KA	2

	4. Two qubit gates and tensor products; Working with matrices 5. The No-Cloning Theorem; The Quantum Teleportation protocol 6. Algorithms a. Simple quantum algorithms (Bernstein-Vazirani, Simon's algorithm) b. Implementing Deutsch-Josza with Mach-Zehnder Interferometers c. Quantum factoring (Shor's Algorithm) d. Quantum search (Grover's Algorithm) 7. Implementation aspects a. The physical implementation of qubits b. Classical control of a Quantum Processing Unit (QPU) c. Error mitigation and control. NISQ and beyond. 8. Emerging Applications a. Post-quantum encryption b. The Quantum Internet c. Adiabatic quantum computation (AQC) and quantum annealing			

OS: Operating Systems

KU	Topic	Skill Level	Core	Hours
OS-Purpose	1. Operating system as mediator between general purpose hardware and application-specific software 2. Universal operating system functions 3. Extended and/or specialized operating system functions 4. Design issues 5. Influences of security, networking, multimedia, parallel and distributed computing 6. Overarching concern of security/protection: Neglecting to consider security at every layer creates an opportunity to inappropriately access resources.	Explain	CS	2
OS-Principles	1. Operating system software design and approaches 2. Abstractions, processes, and resources	Explain	CS	2

	3. Concept of system calls and links to application program interfaces 4. The evolution of the link between hardware architecture and the operating system functions 5. Protection of resources means protecting some machine instructions/functions 6. Leveraging interrupts from hardware level: service routines and implementations 7. Concept of user/system state and protection, transition to kernel mode using system calls 8. Mechanism for invoking of system calls, the corresponding mode and context switch and return from interrupt 9. Performance costs of context switches and associated cache flushes when performing process switches in Spectre-mitigated environments			
OS-Concurrency	1. Thread abstraction relative to concurrency 2. Race conditions, critical regions (role of interrupts if needed) 3. Deadlocks and starvation 4. Multiprocessor issues (spin-locks, reentrancy) 5. Multiprocess concurrency vs multithreading	Explain	CS	2
	6. Thread creation, states, structures 7. Thread APIs 8. Deadlocks and starvation (necessary conditions/mitigations) 9. Implementing thread safe code (semaphores, mutex locks, cond vars) 10. Race conditions in shared memory	Apply	KA	1
OS-Protection	1. Overview of operating system security mechanisms 2. Attacks and antagonism (scheduling, etc.) 3. Review of major vulnerabilities in real operating systems 4. Operating systems mitigation strategies such as backups	Apply	CS	2
	5. Policy/mechanism separation 6. Security methods and devices 7. Protection, access control, and authentication	Apply	KA	1
OS-Scheduling	1. Preemptive and non-preemptive scheduling 2. Schedulers and policies.	Explain	KA	2

	3. Concepts of Symmetric Multi-Processor (SMP) multiprocessor scheduling and cache coherence 4. Timers (e.g., building many timers out of finite hardware timers) 5. Fairness and starvation			
OS-Process	1. Processes and threads relative to virtualization – Protected memory, process state, memory isolation, etc. 2. Memory footprint/segmentation (stack, heap, etc.) 3. Creating and loading executables, shared libraries, and dynamic linking 4. Dispatching and context switching 5. Interprocess communication	Explain	KA	2
OS-Memory	1. Review of physical memory, address translation and memory management hardware 2. Impact of memory hierarchy including cache concept, cache lookup, etc. on operating system mechanisms and policy 3. Logical and physical addressing, address space virtualization 4. Concepts of paging, page replacement, thrashing and allocation of pages and frames 5. Allocation/deallocation/storage techniques (algorithms and data structure) performance and flexibility 6. Memory caching and cache coherence and the effect of flushing the cache to avoid speculative execution vulnerabilities 7. Security mechanisms and concepts in memory management including sandboxing, protection, isolation, and relevant vectors of attack	Explain	KA	2
OS-Devices	1. Buffering strategies 2. Direct Memory Access and Polled I/O, Memory-mapped I/O 3. Historical and contextual - Persistent storage device management (magnetic, SSD, etc.)	Explain	KA	1
OS-Files	1. Concept of a file including Data, Metadata, Operations and Access-mode 2. File system mounting 3. File access control 4. File sharing	Explain	KA	2

	5. Basic file allocation methods including linked, allocation table, etc. 6. File system structures comprising file allocation including various directory structures and methods for uniquely identifying files (name, identified or metadata storage location) 7. Allocation/deallocation/storage techniques (algorithms and data structure) impact on performance and flexibility (i.e. Internal and external fragmentation and compaction) 8. Free space management such as using bit tables vs linking 9. Implementation of directories to segment and track file location			
OS-Advanced-Files	1. File systems: partitioning, mount/unmount, virtual file systems 2. In-depth implementation techniques 3. Memory-mapped files 4. Special-purpose file systems 5. Naming, searching, access, backups 6. Journaling and log-structured file systems	Explain	KA	1
OS-Virtualization	1. Using virtualization and isolation to achieve protection and predictable performance 2. Advanced paging and virtual memory 3. Virtual file systems and virtual devices 4. Containers and their comparison to virtual machine 5. Thrashing	Explain	KA	1
OS-Real-time	1. Process and task scheduling 2. Deadlines and real-time issues 3. Low-latency vs ”soft real-time" vs "hard real time"	Explain	KA	1
OS-Faults	1. Reliable and available systems 2. Software and hardware approaches to address tolerance (RAID)	Explain	KA	1
OS-SEP	1. Open source in operating systems 2. End-of-life issues with sunsetting operating systems	Explain	KA	-

NC: Networking and Communication

KU	Topic	Skill Level	Core	Hours
NC-Fundamentals	1. Importance of networking in contemporary computing, and associated challenges	Explain	CS	3
	2. Organization of the Internet a. Users b. Internet Service Providers c. Autonomous systems d. Content providers e. Content delivery networks	Explain		
	3. Switching techniques a. Circuit Switching b. Packet Switching	Evaluate		
	4. Layers and their roles a. Application b. Transport c. Network d. Datalink e. Physical	Explain		
	5. Layering principles a. Encapsulation b. Hourglass model	Explain		
	6. Network elements a. Routers b. Switches c. Hubs d. Access points e. Hosts	Explain		
	7. Basic queueing concepts a. Relationship with latency b. Relationship with Congestion c. Relationship with Service levels	Explain		
NC-Applications	1. Naming and address schemes. a. DNS b. IP addresses	Explain	CS	4

	c. Uniform Resource Identifiers			
	2. Distributed application paradigms a. Client/server b. Peer-to-peer c. Cloud d. Edge e. Fog	Evaluate		
	3. Diversity of networked application demands a. Latency b. Bandwidth c. Loss tolerance	Explain		
	4. Application-layer development using one or more protocols: a. HTTP b. SMTP c. POP3	Develop		
	5. Interactions with TCP, UDP, and Socket APIs.	Explain		
NC-Reliability	1. Unreliable delivery a. UDP b. Other	Explain	KA	6
	2. Principles of reliability a. Delivery without loss b. Duplication c. Out of order	Develop		
	3. Error control a. Retransmission b. Error correction	Evaluate		
	4. Flow control a. Stop and wait b. Window based	Develop		
	5. Congestion control a. Implicit congestion notification b. Explicit congestion notification	Explain		
	6. TCP and performance issues a. Tahoe b. Reno	Evaluate		

	c. Vegas d. Cubic e. QUIC			
NC-Routing	1. Routing paradigms and hierarchy a. Intra/inter domain b. Centralized and decentralized c. Source routing d. Virtual circuits e. QoS	Evaluate	KA	4
	2. Forwarding methods a. Forwarding tables b. Matching algorithms	Apply		
	3. IP and Scalability issues a. NAT b. CIDR c. BGP d. Different versions of IP	Explain		
NC-SingleHop	1. Introduction to modulation, bandwidth, and communication media	Explain	KA	3
	2. Encoding and Framing.	Evaluate		
	3. Medium Access Control (MAC) a. Random access b. Scheduled access	Evaluate		
	4. Ethernet	Explain		
	5. Switching	Apply		
	6. Local Area Network Topologies (e.g. data center networks)	Explain		
NC-Security	1. General intro about security a. Threats b. Vulnerabilities c. Countermeasures	Explain	KA	4
	2. Network specific threats and attack types a. Denial of service b. Spoofing c. Sniffing	Explain		

	d. Traffic redirection e. Attacker-in-the-middle f. Message integrity attacks g. Routing attacks h. Traffic analysis			
	3. Countermeasures [Shared with Security] a. Cryptography (e.g., SSL, symmetric/asymmetric). b. Architectures for secure networks (e.g., secure channels, secure routing protocols, secure DNS, VPNs, DMZ, Zero Trust Network Access, hyper network security, anonymous communication protocols, isolation) c. Network monitoring, intrusion detection, firewalls, spoofing and DoS protection, honeypots, tracebacks, BGP Sec.	Explain		
NC-Mobility	1. Principles of cellular communication (e.g., 4G, 5G)	Explain	KA	3
	2. Principles of Wireless LANs (mainly 802.11)	Explain		
	3. Device to device communication	Explain		
	4. Multihop wireless networks	Explain		
	5. Examples (e.g., ad hoc networks, opportunistic, delay tolerant)	Explain		
NC-Emerging	1. Middleboxes (e.g., filtering, deep packet inspection, load balancing, NAT, CDN)	Explain	KA	4
	2. Virtualization (e.g. SDN, Data Center Networks)	Explain		
	3. Quantum Networking (e.g. Intro to the domain, teleportation, security, Quantum Internet)	Explain		

PDC: Parallel and Distributed Computing

KU	Topic	Skill Level	Core	Hours

PDC-Programs	1. Parallelism a. Declarative parallelism – determining which actions may, or must not, be performed in parallel, at the level of instructions, functions, closures, composite actions, sessions, tasks, and services is the main idea underlying PDC algorithms; failing to do so is the main source of errors. b. Defining order – for example, using happens-before relations or series/parallel directed acyclic graphs representing programs c. Independence – determining when ordering doesn't matter, in terms of commutativity, dependencies, preconditions d. Ensuring ordering among otherwise parallel actions when necessary, including locking, safe publication; and imposing communication – sending a message happens before receiving it; conversely relaxing when unnecessary 2. Distribution a. Defining places as devices executing actions, including hardware components, remote hosts, may also include external, uncontrolled devices, hosts, and users b. One device may time-slice or otherwise emulate multiple parallel actions by fewer processors by scheduling and virtualization. c. Naming or identifying places (e.g., device IDs) and actions as parties (e.g., thread IDs) d. Activities across places may communicate across media 3. Starting activities a. Options that enable actions to be performed (eventually) at places range from hardwiring to configuration scripts; also establishing communication and resource management; these are expressed differently across languages and contexts, usually relying on automated provisioning and management by platforms b. Procedural: Enabling multiple actions to start at a given program point; for example, starting	Explain	CS	2

	new threads, possibly scoping, or otherwise organizing them in hierarchical groups c. Reactive: Enabling upon an event by installing an event handler, with less control of when actions begin or end d. Dependent: Enabling upon completion of others; for example, sequencing sets of parallel actions e. Granularity: Execution cost of action bodies should outweigh the overhead of arranging. 4. Execution Properties a. Nondeterministic execution of unordered actions b. Consistency – ensuring agreement among parties about values and predicates when necessary to avoid races, maintain safety and atomicity, or arrive at consensus. c. Fault tolerance – handling failures in parties or communication, including (Byzantine) misbehavior due to untrusted parties and protocols, when necessary to maintain progress or availability. d. Tradeoffs are one focus of evaluation			
	5. One or more of the following mappings and mechanisms across layered systems a. CPU data- and instruction-level parallelism. b. SIMD and heterogeneous data parallelism c. Multicore scheduled concurrency, tasks, actors d. Clusters, clouds; elastic provisioning e. Networked distributed systems f. Emerging technologies such as quantum computing and molecular computing	Explain, Develop	KA	2
PDC-Communication	1. Media a. Varieties – channels (message passing or IO), shared memory, heterogeneous, data stores b. Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation 2. Channels a. Explicit (usually named) party-to-party communication media	Explain	CS	2

	b. APIs: sockets, architectural and language-based constructs, and layered constructs such as RPC (remote procedure call) c. IO channel APIs 3. Memory a. Shared memory architectures in which parties directly communicate only with memory at given addresses, with extensions to heterogeneous memory supporting multiple memory stores with explicit data transfer across them; for example, GPU local and shared memory, Direct Memory Access (DMA) b. Memory hierarchies – multiple layers of sharing domains, scopes, and caches; locality: latency, false-sharing c. Consistency properties – bitwise atomicity limits, coherence, local ordering 4. Data Stores a. Cooperatively maintained data structures implementing maps and related ADTs. b. Varieties – owned, shared, sharded, replicated, immutable, versioned			
	5. One or more of the following properties and extensions a. Topologies – unicast, multicast, mailboxes, switches; routing via hardware and software interconnection networks b. Media concurrency properties – ordering, consistency, idempotency, overlapping communication with computation c. Media performance – latency, bandwidth (throughput) contention (congestion), responsiveness (liveness), reliability (error and drop rates), protocol-based progress (acks, timeouts, mediation) d. Media security properties – integrity, privacy, authentication, authorization e. Data formats – marshaling, validation, encryption, compression f. Channel policies: Endpoints, Sessions, buffering, saturation response (waiting vs dropping), Rate control	Explain, Develop	KA	6

	g. Multiplexing and demultiplexing many relatively slow I/O devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs h. Formalization and analysis of channel communication; for example, CSP i. Applications of queuing theory to model and predict performance j. Memory models – sequential and release/acquire consistency k. Memory management, including reclamation of shared data; reference counts and alternatives l. Bulk data placement and transfer; reducing message traffic and improving locality; overlapping data transfer and computation; impact of data layout such as array-of-structs vs struct-of-arrays m. Emulating shared memory: distributed shared memory, Remote Direct Memory Access (RDMA) n. Data store consistency – atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains o. Faults, partitioning, and partial failures; voting; protocols such as Paxos and Raft p. Design tradeoffs among consistency, availability, partition (fault) tolerance; impossibility of meeting all at once q. Security and trust: Byzantine failures, proof of work and alternatives			
PDC-Coordination	1. Dependencies a. Initiation or progress of one activity may be dependent on other activities, so as to avoid race conditions, ensure termination, or meet other requirements. b. Ensuring progress by avoiding dependency cycles, using monotonic conditions, removing inessential dependencies 2. Control constructs and design patterns a. Completion-based – barriers, joins, including termination control	Explain	CS	2

	b. Data-enabled – queues, producer-consumer designs c. Condition-based – polling, retrying, backoffs, helping, suspension, signaling, timeouts d. Reactive: enabling and triggering continuations 3. Atomicity a. Atomic instructions enforced local access orderings b. Locks and mutual exclusion; lock granularity c. Deadlock avoidance – ordering, coarsening, randomized retries; encapsulation via lock managers d. Common errors: Failing to lock or unlock when necessary, holding locks while invoking unknown operations. e. Avoiding locks – replication, read-only, ownership, and non-blocking constructions			
	4. One or more of the following properties and extensions a. Progress properties including lock-free, wait-free, fairness, priority scheduling, interactions with consistency, reliability b. Performance with respect to contention, granularity, convoying, scaling c. Non-blocking data structures and algorithms d. Ownership and resource control e. Lock variants and alternatives: sequence locks, read-write locks; Read-Copy-Update (RCU), reentrancy; tickets; controlling spinning versus blocking f. Transaction-based control – optimistic and conservative g. Distributed locking: reliability h. Alternatives to barriers: clocks; counters, virtual clocks; dataflow and continuations; futures and RPC; consensus-based, gathering results with reducers and collectors i. Speculation, selection, cancellation; observability and security consequences j. Resource control using semaphores and condition variables	Explain, Develop	KA	6

	k. Control flow – scheduling computations, series-parallel loops with (possibly elected) leaders, pipelines and streams, nested parallelism. l. Exceptions and failures, handlers, detection, timeouts, fault tolerance, voting			
PDC-Evaluation:	1. Safety and liveness requirements in terms of temporal logic constructs to express “always” and “eventually” 2. Identifying, testing for, and repairing violations, including common forms of errors such as failure to ensure necessary ordering (race errors), atomicity (including check-then-act errors), and termination (livelock) 3. Performance requirements metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements 4. Performance impact of design and implementation choices, including granularity, overhead, and energy consumption 5. Estimating scalability limitations, for example, using Amdahl’s Law or Universal Scalability Law	Explain, Evaluate	CS	1
	6. One or more of the following methods and tools a. Extensions to formal sequential requirements such as linearizability b. Protocol, session, and transactional specifications c. Use of tools such as Unified Modelling Language (UML), Temporal Logic of Actions (TLA), program logics d. Security analysis: safety and liveness in the presence of hostile or buggy behaviors by other parties; required properties of communication mechanisms (for example lack of cross-layer leakage), input screening, rate limiting e. Static analysis applied to correctness, throughput, latency, resources, energy f. Directed Acyclic Graph (DAG) model analysis of algorithmic efficiency (work, span, critical paths)	Explain, Evaluate	KA	3

	g. Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, visualizations, continuous integration, continuous deployment, and test generators h. Measuring and comparing throughput, overhead, waiting, contention, communication, data movement, locality, resource usage, behavior in the presence of excessive numbers of events, clients, or threads i. Application domain specific analyses and evaluation techniques			
PDC-Algorithms	1. Expressing and implementing algorithms in given languages and frameworks, to initiate activities (for example threads), use shared memory constructs, and channel, socket, and/or remote procedure call APIs a. Data parallel examples including map/reduce b. Using channel, socket, and/or RPC APIs in a specified language, with program control for sending (usually procedural) vs receiving (usually reactive or RPC-based) c. Using locks, barriers, and/or synchronizers to maintain liveness without introducing races 2. Survey of common application domains across multicore, reactive, data parallel, cluster, cloud, open distributed systems, and frameworks	Explain, Develop	CS	2
	3. One or more of the following algorithmic domains a. Linear algebra – vector and matrix operations, numerical precision/stability, applications in data analytics and machine learning b. Data processing – sorting, searching and retrieval, concurrent data structures c. Graphs, search, and combinatorics – marking, edge-parallelization, bounding, speculation, network-based analytics d. Modeling and simulation – differential equations; randomization, N-body problems, genetic algorithms e. Computational Logic – satisfiability (SAT), concurrent logic programming f. Graphics and computational geometry – transforms, rendering, ray-tracing	Explain, Develop, Evaluate	KA	9

	g. Resource management – allocating, placing, recycling and scheduling processors, memory, channels, and hosts; exclusive vs shared resources; static, dynamic, and elastic algorithms; real-time constraints; batching, prioritization, partitioning; decentralization via work-stealing and related techniques h. Services – implementing web APIs, electronic currency, transaction systems, multiplayer games			

SF: Systems Fundamentals

KU	Topic	Skill Level	Core	Hours
SF-Overview	1. Basic building blocks and components of a computer (gates, flip-flops, registers, interconnections; datapath + control + memory) 2. Hardware as a computational paradigm – fundamental logic building blocks; logic expressions, minimization, sum of product forms 3. Programming abstractions, interfaces, and use of libraries 4. Distinction and interaction between application and OS services, remote procedure call 5. Basic concept of pipelining, overlapped processing stages 6. Basic concept of scaling: going faster vs handling larger problems	Explain	CS	3
SF-Foundations	1. Digital vs Analog/Discrete vs Continuous Systems 2. Simple logic gates, logical expressions, Boolean logic simplification 3. Clocks, State, Sequencing 4. State and state transition (e.g., starting state, final state, life cycle of states) 5. Finite state machines (e.g., NFA, DFA) 6. Combinational Logic, Sequential Logic, Registers, Memories 7. Computers and Network Protocols as examples of State Machines	Apply	CS	4

	8. Sequential vs parallel processing 9. Application-level sequential processing – single thread 10. Simple application-level parallel processing – request level (web services/client-server/distributed), single thread per server, multiple threads with multiple servers, pipelining			
SF-Resource	1. Different types of resources (e.g., processor share, memory, disk, net bandwidth) 2. Common resource allocation/scheduling algorithms (e.g., first-come-first-serve, priority-based scheduling, fair scheduling, and preemptive scheduling)	Explain	CS	1
	3. Advantages and disadvantages of common scheduling algorithms	Explain	KA	1
SF-Performance	1. Latencies in computer systems a. Speed of light and computers (one foot per nanosecond vs one GHz clocks) b. Memory vs disk latencies vs across-the-network memory 2. Caches and the effects of spatial and temporal locality on performance in processors and systems 3. Caches and cache coherency in databases, operating systems, distributed systems, and computer architecture 4. Introduction to the processor memory hierarchy	Apply	CS	2
	5. The formula for average memory access time 6. Rationale of virtualization and isolation – protection and predictable performance 7. Levels of indirection, illustrated by virtual memory for managing physical memory resources 8. Methods for implementing virtual memory and virtual machines	Apply	KA	2
SF-Evaluation	1. Performance figures of merit 2. Workloads and representative benchmarks, and methods of collecting and analyzing performance figures of merit 3. CPI (Cycles per Instruction) equation as a tool for understanding tradeoffs in the design of instruction	Evaluate	CS	2

	sets, processor pipelines, and memory system organizations 4. Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can be 5. Order of magnitude analysis (Big O notation) 6. Analysis of slow and fast paths of a system 7. Events on their effect on performance (e.g., instruction stalls, cache misses, page faults)			
	8. Analytical tools to guide quantitative evaluation 9. Understanding layered systems, workloads, and platforms, their implications for performance, and the challenges they represent for evaluation 10. Microbenchmarking pitfalls	Evaluate	KA	2
SF-Reliability	1. Distinction between bugs and faults 2. Reliability vs availability 3. Reliability through redundancy a. check and retry b. redundant encoding (error correction codes, CRC, FEC) c. duplication/mirroring/replicas	Evaluate	CS	2
	4. Other approaches to reliability (e.g., journaling)	Evaluate	KA	1
SF-Security	1. Common system security issues (e.g., viruses, denial-of-service attacks, and eavesdropping) 2. Countermeasures a. Cryptography b. Security architecture	Evaluate	CS	2
	3. Representative countermeasure systems a. Intrusion detection systems, firewalls	Evaluate	KA	1
SF-Design	1. Common criteria of system design (e.g., liveness, safety, robustness, scalability, and security)	Design	CS	2
	2. Designs of representative systems (e.g., Apache web server, Spark, and Linux)	Design	KA	1

DM: Data Management

KU	Topic	Skill Level	Core	Hours

DM-Data	1. The Data Life Cycle	Evaluate	CS	2
DM-Core	1. Purpose and advantages of database systems 2. Components of database systems 3. Design of core DBMS functions (e.g., query mechanisms, transaction management, buffer management, access methods) 4. Database architecture, data independence, and data abstraction 5. Transaction management 6. Normalization 7. Approaches for managing large volumes of data (e.g., NoSQL database systems, use of MapReduce) 8. How to support CRUD-only applications 9. Distributed databases/cloud-based systems 10. Structured, semi-structured, and unstructured databases	Explain	CS	2
	11. Use of a declarative query language	Develop		
DM-Core	12. Systems supporting structured and/or stream content	Explain	KA	1
DM-Modeling	1. Data modeling 2. Relational data models	Develop	CS	2
DM-Modeling	3. Conceptual models (e.g., entity-relationship, UML diagrams) 4. Semi-structured data model (expressed using DTD, XML, or JSON Schema, for example)	Explain	KA	3
DM-Relational	1. Entity and referential integrity a. Candidate key, superkeys 2. Relational database design	Explain	CS	1
DM-Relational	3. Mapping conceptual schema to a relational schema 4. Physical database design: file and storage structures 5. Introduction to Functional dependency theory 6. Normalization theory a. Decomposition of a schema; lossless-join and dependency-preservation properties of a decomposition b. Normal forms (BCNF) c. Denormalization (for efficiency)	Develop	KA	3
DM-	1. SQL Query Formation	Develop	CS	2

Querying	a. Interactive SQL execution b. Programmatic execution of an SQL query			
DM-Querying	2. Relational Algebra 3. SQL a. Data definition including integrity and other constraints specification b. Update sublanguage	Develop	KA	4
DM-Processing	1. Page structures 2. Index structures a. B+ trees b. Hash indices: static and dynamic c. Index creation in SQL 3. File Structures a. Heap files b. Hash files 4. Algorithms for query operators a. External Sorting b. Selection c. Projection; with and without duplicate elimination d. Natural Joins: Nested loop, Sort-merge, Hash join e. Analysis of algorithm efficiency 5. Query transformations 6. Query optimization a. Access paths b. Query plan construction c. Selectivity estimation d. Index-only plans 7. Parallel Query Processing (e.g., parallel scan, parallel join, parallel aggregation)	Explain	KA	4
	8. Database tuning/performance a. Index selection b. Impact of indices on query performance c. Denormalization	Develop		
DM-Internals	1. DB Buffer Management 2. Transaction Processing a. Isolation Levels b. ACID	Explain	KA	4

	c. Serializability d. Distributed transactions 3. Concurrency Control: a. 2-Phase Locking b. Deadlocks handling strategies c. Quorum-based consistency models 4. Recovery Manager a. Relation with Buffer Manager			
DM-NoSQL	1. Why NoSQL? (e.g., Impedance mismatch between Application [CRUD] and RDBMS) 2. Key-Value and Document data model	Explain	KA	2
DM-Analytics	1. Exploratory data techniques (motivation, representation, descriptive statistics, visualizations) 2. Data science lifecycle – business understanding, data understanding, data preparation, modeling, evaluation, deployment, and user acceptance 3. Data mining and machine learning algorithms: e.g., classification, clustering, association, regression 4. Data acquisition and governance 5. Data security and privacy considerations 6. Data fairness and bias 7. Data visualization techniques and their use in data analytics 8. Entity Resolution	Explain	KA	3
DM-Security	1. Differences between data security and data privacy 2. Protecting data and database systems from attacks, including injection attacks such as SQL injection 3. Personally identifying information (PII) and its protection 4. Ethical considerations in ensuring the security and privacy of data	Explain	CS	1
DM-Security	5. Need for, and different approaches to securing data at rest, in transit, and during processing 6. Database auditing and its role in digital forensics 7. Data inferencing and preventing attacks	Explain	KA	2

	8. Laws and regulations governing data security and data privacy			
DM-SEP	1. Issues related to scale 2. Data privacy overall a. Privacy compliance by design 3. Data anonymity 4. Data ownership/custodianship 5. Intended and unintended applications of stored data	Explain	CS	
DM-SEP	6. Reliability of data 7. Provenance, data lineage, and metadata management 8. Data security	Explain	KA	

SEC: Security

KU	Topic	Skill Level	Core	Hours
SEC-Foundations	1. Developing a security mindset incorporating crosscutting concepts: confidentiality, integrity, availability, risk assessment, systems thinking, adversarial thinking, human-centered thinking 2. Basic concepts of authentication and authorization/access control 3. Vulnerabilities, threats, attack surfaces, and attack vectors 4. Denial of Service (DoS) and Distributed Denial of Service (DDoS) 5. Principles and practices of protection, e.g., least privilege, open design, fail-safe defaults, defense in depth, and zero trust; and how they can be implemented 6. Optimization considerations between security, privacy, performance, and other design goals 7. Impact of AI on security and privacy: using AI to bolster defenses as well as address increased adversarial capabilities due to AI	Develop	CS	1
SEC-Foundations	8. Access control models (e.g., discretionary, mandatory, role-based, and attribute-based) 9. Security controls	Develop	KA	5

	10. Concepts of trust and trustworthiness 11. Applications of a security mindset: web, cloud, and mobile devices 12. Protecting embedded and cyber-physical systems 13. Principles of usable security and human-centered computing 14. Security and trust in AI/machine learning systems, e.g., fit for purpose, ethical operating boundaries, authoritative knowledge sources, verified training data, repeatable system evaluation tests, system attestation, independent validation/certification; unintended consequences from: adverse effect 15. Security risks in building and operating AI/machine learning systems, e.g., algorithm bias, knowledge corpus bias, training corpus bias, copyright violation 16. Hardware considerations in security, e.g., principles of secure hardware, secure processor architectures, cryptographic acceleration, compartmentalization, software-hardware interaction			
SEC-SEP	1. Principles and practices of privacy 2. Societal impacts on breakdowns in security and privacy. 3. Applicability of laws and regulations on security and privacy 4. Professional ethical considerations when designing secure systems and maintaining privacy; ethical hacking	Develop	CS	1
SEC-SEP	5. Security by design 6. Privacy by design and privacy engineering 7. Security and privacy implications of malicious AI/machine learning actors, e.g., identifying deep fakes 8. Societal impacts of Internet of Things (IoT) devices and other emerging technologies on security and privacy	Develop	KA	2
SEC-Coding	1. Common vulnerabilities and weaknesses 2. SQL injection and other injection attacks 3. Cross-site scripting techniques and mitigations 4. Input validation and data sanitization 5. Type safety and type-safe languages	Develop	CS	2

	6. Buffer overflows, stack smashing, and integer overflows 7. Security issues due to race conditions			
SEC-Coding	8. Principles of noninterference and non-deducibility 9. Preventing information flow attacks 10. Offensive security techniques as a defense 11. AI-assisted malware detection techniques 12. Ransomware: creation, prevention, and mitigation. 13. Secure use of third-party components 14. Malware: varieties, creation, reverse engineering, and defense against them 15. Assurance: testing (including fuzzing and penetration testing), verification and validation 16. Static and dynamic analyses 17. Secure compilers and secure code generation	Develop	KA	5
SEC-Crypto	1. Differences between algorithmic, applied, and mathematical views of cryptography 2. Mathematical preliminaries: modular arithmetic, Euclidean algorithm, probabilistic independence, linear algebra basics, number theory, finite fields, complexity, asymptotic analysis. 3. Basic cryptography – symmetric key and public key cryptography 4. Basic cryptographic building blocks, including symmetric encryption, asymmetric encryption, hashing, and message authentication 5. Classical cryptosystems, such as shift, substitution, transposition ciphers, code books, machines 6. Kerckhoff's principle and use of vetted libraries 7. History and real-world applications, e.g., electronic cash, secure channels between clients and servers, secure electronic mail, entity authentication, device pairing, voting systems	Evaluate	CS	1
SEC-Crypto	8. Additional mathematics – primality and factoring; elliptic curve cryptography 9. Private-key cryptosystems – substitution-permutation networks, linear cryptanalysis, differential cryptanalysis, DES, AES 10. Public-key cryptosystems – Diffie-Hellman, RSA 11. Data integrity and authentication – hashing, digital signatures	Develop	KA	4

	12. Cryptographic protocols – challenge-response authentication, zero-knowledge protocols, commitment, oblivious transfer, secure two- or multi-party computation, hash functions, secret sharing, and applications 13. Attacker capabilities – chosen-message attack (for signatures), birthday attacks, side channel attacks, fault injection attacks 14. Quantum cryptography – Post Quantum/Quantum resistant cryptography 15. Blockchain and cryptocurrencies			
SEC-Engineering	1. Security engineering goals – building systems that remain dependable despite errors, accidents, or malicious adversaries 2. Privacy engineering goals – building systems that design, implement, and deploy privacy features and controls 3. Problem analysis and situational analysis to address system security 4. Engineering tradeoff analysis based on time, cost, risk tolerance, risk acceptance, return on investment, and so on	Develop	CS	1
SEC-Engineering	5. Security design and engineering, including functional requirements, security subsystems, information protection, security testing, security assessment, and evaluation 6. Security analysis, covering security requirements analysis; security controls analysis; threat analysis; and vulnerability analysis 7. Security attack domains and attack surfaces, e.g., communications and networking, hardware, physical, social engineering, software, and supply chain 8. Security attack modes, techniques, and tactics, e.g., authentication abuse; brute force; buffer manipulation; code injection; content insertion; denial of service; eavesdropping; function bypass; impersonation; integrity attack; interception; phishing; protocol analysis; privilege abuse; spoofing; and traffic injection 9. Attestation of software products with respect to their specification and adaptiveness 10. Design and development of cyber-physical systems	Develop	KA	8

	11. Considerations for trustworthy computing, e.g., tamper resistant packaging, trusted boot, trusted kernel, hardware root of trust, software signing and verification, hardware-based cryptography, virtualization, and containers			
SEC-Forensics	1. Basic principles and methodologies for digital forensics 2. System design for forensics 3. Forensics in different situations – operating systems, file systems, application forensics, web forensics, network forensics, mobile device forensics, use of database auditing 4. Attacks on forensics and preventing such attacks 5. Incident handling processes 6. Rules of evidence – general concepts and differences between jurisdictions 7. Legal issues – digital evidence protection and management, chains of custody, reporting, serving as an expert witness.	Develop	KA	6
SEC-Governance	1. Protecting critical assets from threats 2. Security governance – organizational objectives and general risk assessment 3. Security management – achieve and maintain appropriate levels of confidentiality, integrity, availability, accountability, authenticity, and reliability 4. Security policy – Organizational policies, issue-specific policies, system-specific policies 5. Approaches to identifying and mitigating risks to computing infrastructure 6. Data lifecycle management policies: data collection, backups, and retention; cloud storage and services; breach disclosure	Develop	KA	3

Applications Competency Area

The core topics listed in this section are typically covered in advanced, often elective courses. Effort should be made to include the CS Core topics in earlier required courses in the curriculum.

	Knowledge Area	Knowledge Units	CS Core	KA Core

AI	Artificial Intelligence	12	12	18
GIT	Graphics and Interactive Techniques	11	4	70
HCI	Human-Computer Interaction	6	8	16
SPD	Specialized Platform Development	8	4	68
	Total		28	N/A

AI: Artificial Intelligence

KU	Topic	Skill Level	Core	Hours
AI-Introduction	1. Overview of AI problems, examples of successful recent AI applications 2. Definitions of agents with examples (e.g., reactive, deliberative) 3. What is intelligent behavior? a. The Turing test and its flaws b. Multimodal input and output c. Simulation of intelligent behavior d. Rational vs non-rational reasoning	Explain	CS	2
	4. Problem characteristics a. Fully versus partially observable b. Single vs multi-agent c. Deterministic versus stochastic d. Static vs dynamic e. Discrete versus continuous 5. Nature of agents a. Autonomous, semi-autonomous, mixed-initiative autonomy b. Reflexive, goal-based, and utility-based c. Decision making under uncertainty and with incomplete information d. The importance of perception and environmental interactions e. Learning-based agents f. Embodied agents i. sensors, dynamics, effectors	Evaluate		

	6. AI Applications, growth, and Impact (economic, societal, ethics)	Explain		
AI-Introduction	*7.* Practice identifying problem characteristics in example environments 8. Additional depth on nature of agents with examples *9.* Additional depth on AI Applications, growth, and Impact (economic, societal, ethics)	Evaluate	KA	1
AI-Search AL-Foundational	1. State space representation of a problem a. Specifying states, goals, and operators b. Factoring states into representations (hypothesis spaces) c. Problem solving by graph search i. e.g., Graphs as a space, and tree traversals as exploration of that space ii. Dynamic construction of the graph (not given upfront)	Explain	CS	5
	2. Uninformed graph search for problem solving a. Breadth-first search b. Depth-first search i. With iterative deepening c. Uniform cost search	Develop, Apply		
	3. Heuristic graph search for problem solving a. Heuristic construction and admissibility b. Hill-climbing c. Local minima and the search landscape i. Local vs global solutions d. Greedy best-first search e. A* search	Develop, Apply		
	4. Space and time complexities of graph search algorithms	Evaluate		
AI-Search	5. Bidirectional search 6. Beam search 7. Two-player adversarial games a. Minimax search b. Alpha-beta pruning i. Ply cutoff 8. Implementation of A* search 9. Constraint Satisfaction	Develop, Apply	KA	6

AI-KRR	1. Types of representations a. Symbolic, logical i. Creating a representation from a natural language problem statement b. Learned subsymbolic representations c. Graphical models (e.g., naive Bayes, Bayes net) 2. Review of probabilistic reasoning, Bayes theorem	Explain	CS	2
	3. Bayesian reasoning a. Bayesian inference	Apply		
AI-KRR	4. Random variables and probability distributions a. Axioms of probability b. Probabilistic inference c. Bayes' Rule (derivation) d. Bayesian inference (more complex examples) 5. Independence 6. Conditional Independence 7. Markov chains and Markov models 8. Utility and decision making	Apply	KA	2
AI-ML	1. Definition and examples of a broad variety of machine learning tasks a. Supervised learning i. Classification ii. Regression b. Reinforcement learning c. Unsupervised learning i. Clustering 2. Fundamental ideas: a. No free lunch theorem: no one learner can solve all problems; representational design decisions have consequences b. sources of error and undecidability in machine learning		CS	4
	3. A simple statistical-based supervised learning such as linear regression or decision trees a. Focus on how they work without going into mathematical or optimization details; enough to understand and use existing implementations correctly.	Apply, Develop, Evaluate		

	4. The overfitting problem / controlling solution complexity (regularization, pruning – intuition only)
	  a. The bias (underfitting) – variance (overfitting) tradeoff
	5. Working with Data
	  a. Data preprocessing
	  i. Importance and pitfalls of preprocessing choices
	  b. Handling missing values (imputing, flag-as-missing)
	  i. Implications of imputing vs flag-as-missing
	  c. Encoding categorical variables, encoding real-valued data
	  d. Normalization/standardization
	  e. Emphasis on real data, not textbook examples
	6. Representations
	  a. Hypothesis spaces and complexity
	  b. Simple basis feature expansion, such as squaring univariate features
	  c. Learned feature representations
	7. Machine learning evaluation
	  a. Separation of train, validation, and test sets
	  b. Performance metrics for classifiers
	  c. Estimation of test performance on held-out data
	  d. Tuning the parameters of a machine learning model with a validation set
	  e. Importance of understanding what a model is doing, where its pitfalls/shortcomings are, and the implications of its decisions
	8. Basic neural networks
	  a. Fundamentals of understanding how neural networks work and their training process, without details of the calculations
	  b. Basic introduction to generative neural networks (large language models, etc.)			

	9. Ethics for Machine Learning
	  a. Focus on real data, real scenarios, and case studies
	  b. Dataset/algorithmic/evaluation bias and unintended consequences
		Explain, Evaluate		

AI-ML	10. Formulation of simple machine learning as an optimization problem, such as least squares linear regression or logistic regression a. Objective function b. Gradient descent c. Regularization to avoid overfitting (mathematical formulation) 11. Ensembles of models a. Simple weighted majority combination 12. Deep learning a. Deep feed-forward networks (intuition only, no mathematics) b. Convolutional neural networks (intuition only, no mathematics) c. Visualization of learned feature representations from deep nets d. Other architectures (generative NN, recurrent NN, transformers, etc.) 13. Performance evaluation a. Other metrics for classification (e.g., error, precision, recall) b. Performance metrics for regressors c. Confusion matrix d. Cross-validation i. Parameter tuning (grid/random search, via cross-validation) 14. Overview of reinforcement learning methods 15. Two or more applications of machine learning algorithms a. e.g., medicine and health, economics, vision, natural language, robotics, game play	Apply, Develop, Evaluate	KA	6
	16. Ethics for Machine Learning a. Continued focus on real data, real scenarios, and case studies b. Privacy c. Fairness d. Intellectual property e. Explainability	Explain, Evaluate		
AI-SEP	1. At least one application of AI to a specific problem and field, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (at least one for the CS Core)	Explain, Evaluate	CS	3

	a. Formulating and evaluating a specific application as an AI problem i. How to deal with underspecified or ill-posed problems b. Data availability/scarcity and cleanliness i. Basic data cleaning and preprocessing ii. Data set bias c. Algorithmic bias d. Evaluation bias e. Assessment of societal implications of the application 2. Deployed deep generative models a. High-level overview of deep image generative models (e.g., as of 2023, DALL-E, Midjourney, Stable Diffusion), their uses, and their shortcomings/pitfalls. b. High-level overview of large language models (e.g., as of 2023, ChatGPT, Bard), their uses, and their shortcomings/pitfalls. 3. Overview of societal impact of AI a. Ethics b. Fairness c. Trust/explainability d. Privacy and usage of training data e. Human autonomy and oversight/regulations/legal requirements f. Sustainability			
AI-SEP	4. One or more additional applications of AI to a broad set of problems and diverse fields, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose a different area from that chosen for the CS Core). a. Formulating and evaluating a specific application as an AI problem i. How to deal with underspecified or ill-posed problems b. Data availability/scarcity and cleanliness i. Basic data cleaning and preprocessing ii. Data set bias c. Algorithmic bias d. Evaluation bias	Explain, Evaluate	KA	3

	e. Assessment of societal implications of the application 5. Additional depth on deployed deep generative models a. Introduction to how deep image generative models work, (e.g., as of 2023, DALL-E, Midjourney, Stable Diffusion) including discussion of attention b. Introduction to how large language models work, (e.g., as of 2023, ChatGPT, Bard) including discussion of attention c. Idea of foundational models, how to use them, and the benefits/issues with training them from big data 6. Analysis and discussion of the societal impact of AI a. Ethics b. Fairness c. Trust/explainability d. Privacy and usage of training data e. Human autonomy and oversight/regulations/legal requirements f. Sustainability			

GIT: Graphics and Interactive Techniques

KU	Topic	Skill Level	Core	Hours
GIT-Fundamentals	1. Uses and potential risks 2. Graphic output 3. Human vision system 4. Standard image formats 5. Digitization of analog data 6. Color Models 7. Tradeoffs between storing image data and recomputing image data 8. Spatialization 9. Animation	Explain	CS	4
	10. Applied computer graphics 11. Display characteristics	Develop	KA	3
	1. Data visualization and information visualization	Explain	KA	6

GIT-Visualization	2. Visualization of: a. 2D/3D scalar fields b. Vector fields and flow data c. Time-varying data d. High-dimensional data e. Non-spatial data 3. Visualization techniques (e.g., color mapping, isosurfaces, dimension reduction, parallel coordinates, multi-variate, tree/graph-structured, text) 4. Direct volume data rendering (e.g., ray-casting, transfer functions, segmentation) 5. Common data formats (e.g., HDF, netCDF, geotiff, GeoJSON, shape files, raw binary, CSV, ASCII to parse) 6. Common visualization software and libraries (e.g., R, Processing, D3.js, GIS, Matlab, IDL, Python) 7. Perceptual and cognitive foundations that drive visual abstractions 8. Visualization design 9. Evaluation of visualization methods and applications 10. Visualization bias 11. Applications of visualization			
GIT-Rendering	1. Object and scene modeling 2. Camera and projection modeling 3. Light models and radiometry 4. Rendering	Explain	KA	6
GIT-Modeling	1. Basic geometric operations 2. Surface representation/model 3. Volumetric representation/model 4. Procedural representation/model 5. Multi-resolution modeling 6. Reconstruction	Explain	KA	6
GIT-Shading	1. Solutions and approximations to the rendering equation 2. Time (motion blur), lens position (focus), and continuous frequency (color) and their impact on rendering 3. Shadow mapping 4. Occlusion culling 5. Bidirectional Scattering Distribution function (BSDF) theory and microfacets	Explain	KA	6

	6. Subsurface scattering 7. Area light sources 8. Hierarchical depth buffering 9. Image-based rendering 10. Non-photorealistic rendering 11. GPU architecture 12. Human visual systems including adaptation to light, sensitivity to noise, and flicker fusion			
GIT-Animation	1. Principles of Animation 2. Types of animation 3. Key-frame animation 4. Forward and inverse kinematics 5. Skinning algorithms 6. Motion capture	Explain	KA	6
GIT-Simulation	1. Collision detection and response 2. Procedural animation using noise 3. Particle systems 4. Grid-based fluids 5. Heightfields 6. Rule-based systems (e.g., L-systems, space-colonizing systems, Game of Life)	Explain	KA	6
GIT-Immersion	1. Immersion levels (i.e., Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR)) 2. The definition and distinction of immersion and presence 3. 360 Video 4. Stereoscopic display 5. Viewer tracking 6. Time-critical rendering to achieve optimal motion to photon (MTP) latency a. Branching movies 7. Distributed VR, collaboration over computer network 8. Presence and factors that impact level of immersion 9. 3D interaction 10. Applications in medicine, simulation, training, and visualization 11. Safety in immersive applications	Explain	KA	6
GIT-Interaction	1. Event Driven Programming 2. Graphical User Interface (Single Channel) 3. Accessibility	Apply	KA	4

GIT-Image	1. Morphological operations 2. Color histograms 3. Image enhancement 4. Image restoration 5. Image coding 6. Connections to deep learning	Explain	KA	6
GIT-Physical	1. Interaction with the physical world 2. Connection to physical artifacts 3. Internet of Things	Explain	KA	6
GIT-SEP	1. Accessibility 2. Ethics/privacy 3. Intellectual Property law as it relates to computer graphics and interactive techniques 4. DEIA, current and past contributors to the field	Evaluate	CS KA	1 3

HCI: Human-Computer Interaction

KU	Topic	Skill Level	Core	Hours
HCI-User	1. User-centered design and evaluation methods a. "You are not the users" b. user needs-finding c. formative studies d. interviews e. surveys f. usability tests	Explain, Evaluate, Develop	CS	2
HCI-User	2. User-centered design and evaluation methods a. personas/persona spectrum b. user stories/storytelling and techniques for gathering stories c. empathy maps d. needs assessment (techniques for uncovering needs and gathering requirements – e.g., interviews, surveys, ethnographic and contextual enquiry) e. journey maps f. evaluating the design g. interfacing with stakeholders, as a team	Explain, Evaluate, Develop	KA	5

	h. risks associated with physical, distributed, hybrid and virtual teams 3. Physical and cognitive characteristics of the user a. physical capabilities that inform interaction design (e.g., color perception, ergonomics) b. cognitive models that inform interaction design (e.g., attention, perception and recognition, movement, memory) c. topics in social/behavioral psychology (e.g., cognitive biases, change blindness) 4. Designing for diverse user populations a. how differences (e.g., in race, ability, age, gender, culture, experience, and education) impact user experiences and needs b. internationalization, other cultures, and cross-cultural design c. designing for users from other cultures d. cross-cultural design e. challenges to effective design evaluation (e.g., sampling, generalization; disability and disabled experiences) f. universal design 5. Collaboration and communication a. understanding the user in a multi-user context b. synchronous group communication (e.g., chat rooms, conferencing, online games) c. asynchronous group communication (e.g., email, forums, social networks) d. social media, social computing, and social network analysis e. online collaboration f. social coordination and online communities g. avatars, characters, and virtual worlds			
HCI-Accountability	1. Design impact a. Sustainability b. Inclusivity c. Safety, security, and privacy d. Harm and disparate impact 2. Ethics in design methods and solutions a. the role of artificial intelligence b. responsibilities for considering stakeholder impact and human factors c. the role of design to meet user needs	Explain, Apply, Evaluate Develop	CS	2

	3. Requirements in design a. ownership responsibility b. legal frameworks and compliance requirements c. consideration beyond immediate user needs including via iterative reconstruction of problem analysis and “digital well-being” features			
HCI-Accountability	4. Value-sensitive design a. identify stakeholders b. determine and include diverse stakeholder values and value systems 5. Persuasion through design a. assessing persuasive content of a design b. persuasion as a design goal	Explain, Apply, Evaluate Develop	KA	2
HCI-Accessibility	1. Background a. societal and legal support for and obligations to people with disabilities b. accessible design benefits everyone 2. Techniques a. accessibility standards (e.g., Web Content Accessibility Guidelines) 3. Technologies a. features and products that enable accessibility and support inclusive development by designers and engineers 4. Inclusive Design Frameworks (IDFs) a. recognizing differences 5. Universal design	Explain, Apply, Evaluate Develop	CS	2
HCI-Accessibility	6. Background a. Demographics and populations b. International perspectives c. Attitudes towards people with disabilities 7. Techniques a. UX (user experience) design and research b. software engineering practices that enable inclusion and accessibility. 8. Technologies a. examples of accessibility-enabling features, such as conformance to screen readers 9. Inclusive Design Frameworks a. creating inclusive processes, such as participatory design; designing for larger impact	Explain, Apply, Evaluate Develop	KA	2

	b. designing for larger impact			
HCI-Evaluation	1. Methods for evaluation with users a. formative (e.g., needs-finding and exploratory analysis) and summative assessment (e.g., functionality and usability testing) b. elements to evaluate (e.g., utility, efficiency, learnability, user satisfaction, affective elements such as pleasure and engagement) c. understanding ethical approval requirements before engaging in user research	Explain, Apply, Evaluate, Develop	CS	1
HCI-Evaluation	2. Methods for evaluation with users a. qualitative methods (qualitative coding and thematic analysis) b. quantitative methods (statistical tests) c. mixed methods (e.g., observation, think-aloud, interview, survey, experiment) d. presentation requirements (e.g., reports, personas) e. user-centered testing f. heuristic evaluation g. challenges and shortcomings to effective evaluation (e.g., sampling, generalization) 3. Study planning a. how to set study goals b. hypothesis design c. approvals from Institutional Research Boards and ethics committees d. how to pre-register a study e. within-subjects vs between-subjects design 4. Implications and impacts of design with respect to a. the environment b. material c. society d. security e. privacy f. ethics g. broader impacts	Explain, Apply, Evaluate, Develop	KA	2
HCI-Design	1. Prototyping techniques and tools a. low-fidelity prototyping b. rapid prototyping c. throw-away prototyping	Explain, Apply, Evaluate, Develop	CS	1

	d. granularity of prototyping 2. Design patterns a. iterative design b. universal design c. interaction design (e.g., data-driven design, event-driven design 3. Design constraints a. platforms b. devices c. resources d. balance among usability, security and privacy			
HCI-Design	4. Design patterns and guidelines a. software architecture patterns b. cross-platform design c. synchronization 5. Design processes a. participatory design b. co-design c. double-diamond d. convergence and divergence 6. Interaction techniques a. input and output vectors (e.g., gesture, pose, touch, voice, force) b. graphical user interfaces c. controllers d. haptics e. hardware design f. error handling 7. Visual UI design a. color b. layout c. gestalt principles	Explain, Apply, Evaluate Develop	KA	5
HCI-SEP	1. Universal and user-centered design 2. Accountability 3. Accessibility and inclusive design 4. Evaluating the design 5. System design	Explain, Apply, Evaluate, Develop	CS	Shared with SEP
HCI-SEP	6. Participatory and inclusive design processes	Explain, Apply, Evaluate, Develop	KA	Shared with SEP

SPD: Specialized Platform Development

KU	Topic	Skill Level	Core	Hours
SPD-Common	1. Overview of development platforms (i.e., web, mobile, game, robotics, embedded, and interactive) a. Input/sensors/control devices/haptic devices b. Resource constraints i. Computational ii. Data storage iii. Memory iv. Communication c. Requirements – security, uptime availability, fault tolerance d. Output/actuators/haptic devices 2. Programming via platform-specific Application Programming Interface (API) vs traditional application construction 3. Overview of platform Languages (e.g., Python, Swift, Lua, Kotlin) 4. Programming under platform constraints and requirements (e.g., available development tools, development, security considerations) 5. Techniques for learning and mastering a platform-specific programming language	Apply	CS	3
SPD-Web	1. Web programming languages (e.g., HTML5, JavaScript, PHP, CSS) 2. Web platforms, frameworks, or meta-frameworks a. Cloud services b. API, Web Components 3. Software as a Service (SaaS). 4. Web standards such as document object model, accessibility 5. Security and Privacy Considerations	Apply	KA	5
SPD-Mobile	1. Development with a. Mobile programming languages b. Mobile programming environments	Apply	KA	3

	2. Mobile platform constraints a. User interface design b. Security 3. Access a. Accessing data through API b. Designing API endpoints for mobile apps – pitfalls and design considerations c. Network and the web interfaces			
SPD-Robot	1. Types of robotic platforms and devices 2. Sensors, embedded computation, and effectors (actuators) 3. Robot-specific languages and libraries 4. Robotic software architecture (e.g., using the Robot Operating System (ROS)) 5. Robotic platform constraints and design considerations 6. Interconnections with physical or simulated systems 7. Robotic Algorithms 8. Forward kinematics a. Inverse kinematics b. Dynamics c. Navigation and path planning d. Grasping and manipulation 9. Safety and interaction considerations	Apply	KA	4
SPD-Embedded	1. Introduction to the unique characteristics of embedded systems a. Real-time vs soft real-time and non-real-time systems b. Resource constraints, such as memory profiles and deadlines c. API for custom architectures d. GPU technology. e. Field Programmable Gate Arrays (FPGA). f. Cross-platform systems 2. Embedded Systems a. Microcontrollers b. Interrupts and feedback c. Interrupt handlers in high-level languages d. Hard and soft interrupts and trap-exits e. Interacting with hardware, actuators, and sensors	Apply	KA	4

	f. Energy efficiency g. Loosely timed coding and synchronization h. Software adapters 3. Embedded programming 4. Hard real-time systems vs soft real-time systems a. Timeliness b. Time synchronization/scheduling c. Prioritization d. Latency e. Compute jitter 5. Real-time resource management. 6. Memory management: a. Mapping programming construct (variable) to a memory location b. Shared memory c. Manual memory management d. Garbage collection e. Safety considerations and safety analysis. 7. Sensors and actuators 8. Analysis and verification 9. Application design			
SPD-Game	1. Historical and contemporary platforms for games a. Evolution of Game Platforms (e.g., Brown Box to Metaverse and beyond) Improvement in Computing Architectures (CPU and GPU); Platform Convergence and Mobility) b. Typical Game Platforms (e.g., Personal Computer; Home Console; Handheld Console; Arcade Machine; Interactive Television; Mobile Phone; Tablet; Integrated Head-Mounted Display; Immersive Installations and Simulators; Internet of Things enabled Devices; CAVE Systems; Web Browsers; Cloud-based Streaming Systems) c. Characteristics and Constraints of Different Game Platforms (e.g., Features (local storage, internetworking, peripherals); Run-time performance (GPU/CPU frequency, number of cores); Chipsets (physics processing units, vector co-processors);	Apply	KA	4

	Expansion Bandwidth (PCIe); Network throughput (Ethernet); Memory types and capacities (DDR/GDDR); Maximum stack depth; Power consumption; Thermal design; Endian) d. Typical Sensors, Controllers, and Actuators (e.g., distinctive control system designs – peripherals (mouse, keypad, joystick), game controllers, wearables, interactive surfaces; electronics and bespoke hardware; computer vision, inside-out tracking, and outside-in tracking; IoT-enabled electronics and I/O e. eSports Ecosystems (e.g., evolution of gameplay across platforms; games and eSports; game events such as LAN/arcade tournaments and international events such as the Olympic eSports Series; streamed media and spectatorship; multimedia technologies and broadcast management; professional play; data and machine learning for coaching and training) 2. Real-time Simulation and Rendering Systems a. CPU and GPU architectures: (e.g., Flynn's taxonomy; parallelization; instruction sets; standard components—graphics compute array, graphics memory controller, video graphics array basic input/output system; bus interface; power management unit; video processing unit; display interface) b. Pipelines for physical simulations and graphical rendering (e.g., tile-based, immediate-mode) c. Common Contexts for Algorithms, Data Structures, and Mathematical Functions (e.g., game loops; spatial partitioning, viewport culling, and level of detail; collision detection and resolution; physical simulation; behavior for intelligent agents; procedural content generation) d. Media representations (e.g., I/O, and computation techniques for virtual worlds: audio; music; sprites; models and textures;			

	text; dialogue; multimedia (e.g., olfaction, tactile) 3. Game Development Tools and Techniques: a. Programming Languages (e.g., C++; C#; Lua; Python; JavaScript). b. Shader Languages (e.g., HLSL, GLSL; ShaderGraph) c. Graphics Libraries and APIs (e.g., DirectX; SDL; OpenGL; Metal; Vulkan; WebGL) d. Common Development Tools and Environments (e.g., IDEs; Debuggers; Profilers; Version Control Systems including those handling binary assets; Development Kits and Production/Consumer Kits; Emulators) 4. Game Engines a. Open Game Engines (e.g., Unreal; Unity; Godot; CryEngine; Phyre; Source 2; Pygame and Ren'Py; Phaser; Twine; SpringRTS) b. Techniques (e.g., Ideation, Prototyping, Iterative Design and Implementation, Compiling Executable Builds, Development Operations and Quality Assurance – Play Testing and Technical Testing, Profiling; Optimization, Porting; Internationalization and Localization, Networking) 5. Game Design a. Vocabulary (e.g., game definitions; mechanics-dynamics-aesthetics model; industry terminology; experience design; models of experience and emotion) b. Design Thinking and User-Centered Experience Design (e.g., methods of designing games; iteration, incrementing, and the double-diamond; phases of pre- and post-production; quality assurance, including alpha and beta testing; stakeholder and customer involvement; community management) c. Genres (e.g., adventure; walking simulator; first-person shooter; real-time strategy;			

	multiplayer online battle arena (MOBA); role-playing game (rpg)) d. Audiences and Player Taxonomies (e.g., people who play games; diversity and broadening participation; pleasures, player types, and preferences; Bartle, yee). e. Proliferation of digital game technologies to domains beyond entertainment (e.g., Education and Training; Serious Games; Virtual Production; eSports; Gamification; Immersive Experience Design; Creative Industry Practice; Artistic Practice; Procedural Rhetoric)			

Crosscutting Core Topics

The core topics in Society, Ethics, and the Profession (SEP) and Mathematical and Statistical Foundations (MSF) may be covered across the curriculum or in dedicated courses and benefit all the competency areas.

	Knowledge Area	Knowledge Units	CS Core	KA Core
SEP	Society, Ethics, and the Profession	11	18	14
MSF	Mathematical and Statistical Foundations	5	55	145
	Total		73	159

SEP: Society, Ethics, and the Profession

KU	Topic	Skill Level	Core	Hours
SEP-Context	1. Social implications (e.g., political and cultural ideologies) in a hyper-networked world where the capabilities and impact of social media, artificial intelligence and computing in general are rapidly evolving 2. Impact of computing applications (e.g., social media, artificial intelligence applications) on	Evaluate Evaluate	CS	3

	individual well-being, and safety of all kinds (e.g., physical, emotional, economic) 3. Consequences of involving computing technologies, particularly artificial intelligence, biometric technologies, and algorithmic decision-making systems, in civic life (e.g., facial recognition technology, biometric tags, resource distribution algorithms, policing software) and how human agency and oversight is crucial 4. How deficits in diversity and accessibility in computing affect society and what steps can be taken to improve equity in computing	Evaluate Evaluate		
SEP-Context	5. Growth and control of the internet, data, computing, and artificial intelligence 6. Often referred to as the digital divide, differences in access to digital technology resources and its resulting ramifications for gender, class, ethnicity, geography, and/or developing countries, including consideration of responsibility to those who might be less wealthy, under threat, or who would struggle to have their voices heard. 7. Accessibility issues, including legal requirements such as Web Content Accessibility Guidelines (www.w3.org/TR/WCAG21) 8. Context-aware computing	Explain Evaluate Explain Explain	KA	2
SEP-Ethical-Analysis	1. Avoiding fallacies and misrepresentation in argumentation 2. Ethical theories and decision-making (philosophical and social frameworks) 3. Recognition of the role culture plays in our understanding, adoption, design, and use of computing technology 4. Why ethics is important in computing, and how ethics is similar to, and different from, laws and social norms	Apply Apply Evaluate Explain	CS	2
SEP-Ethical-Analysis	5. Professional checklists 6. Evaluation rubrics 7. Stakeholder analysis 8. Standpoint theory 9. Introduction to ethical frameworks (e.g., consequentialism such as utilitarianism, non-	Develop Develop Develop Apply Explain	KA	1

	consequentialism such as duty, rights, or justice, agent-centered such as virtue or feminism, contractarianism, ethics of care) and their use for analyzing an ethical dilemma			
SEP-Professional-Ethics	1. Community values and the laws by which we live 2. The nature of being a professional including care, attention, discipline, fiduciary responsibility, and mentoring 3. Keeping up to date as a computing professional in terms of familiarity, tools, skills, legal and professional frameworks as well as the ability and responsibility to self-assess and progress in the computing field 4. Professional certification, codes of ethics, conduct, and practice, such as the ACM, IEEE, AAAI, and other international societies 5. Accountability, responsibility, and liability (e.g., software correctness, reliability and safety, warranty, negligence, strict liability, ethical approaches to security vulnerability disclosures) including whether a product/service should be built, not just doing so because it is technically possible. 6. Introduction to theories describing the human creation and use of technology including instrumentalism, sociology of technological systems, disability justice, neutrality thesis, pragmatism, and decolonial models, including developing and using technology to right wrongs and do good 7. Strategies for recognizing and reporting designs, systems, software, and professional conduct (or their outcomes) that may violate law or professional codes of ethics	Evaluate Apply Evaluate Evaluate Apply Explain Apply	CS	2
SEP-Professional-Ethics	8. The role of the computing professional and professional societies in public policy 9. Maintaining awareness of consequences 10. Ethical dissent and whistleblowing 11. The relationship between regional culture and ethical dilemmas 12. Dealing with harassment and discrimination 13. Forms of professional credentialing	Explain Apply Explain Evaluate Explain Explain	KA	2

	14. Ergonomics and healthy computing environments 15. Time-to-market and cost considerations versus quality professional standards	Explain Explain		
SEP-IP	1. Intellectual property rights 2. Intangible digital intellectual property (IDIP) 3. Legal foundations for intellectual property protection 4. Common software licenses (e.g., MIT, GPL and its variants, Apache, Mozilla, Creative Commons) 5. Plagiarism and authorship	Explain Explain Evaluate Evaluate Explain	CS	1
SEP-IP	6. Philosophical foundations of intellectual property 7. Forms of intellectual property (e.g., copyrights, patents, trade secrets, trademarks) and the rights they protect 8. Limitations on copyright protections, including fair use and the first sale doctrine 9. Intellectual property laws and treaties that impact the enforcement of copyrights 10. Software piracy and technical methods for enforcing intellectual property rights, such as digital rights management and closed source software as a trade secret 11. Moral and legal foundations of the open-source movement 12. Systems that use others' data (e.g., large language models)	Explain Explain Explain Explain Explain Explain	KA	1
SEP-Privacy	1. Privacy implications of widespread data collection including but not limited to transactional databases, data warehouses, surveillance systems, cloud computing, and artificial intelligence 2. Conceptions of anonymity, pseudonymity, and identity 3. Technology-based solutions for privacy protection (e.g., end-to-end encryption and differential privacy) 4. Civil liberties, privacy rights, and cultural differences	Explain Evaluate Evaluate Explain	CS	2
SEP-Privacy	5. Philosophical and legal conceptions of the nature of privacy including the right to privacy	Explain	KA	1

	6. Legal foundations of privacy protection in relevant jurisdictions (e.g., GDPR in the EU) 7. Privacy legislation in areas of practice (e.g., HIPAA in the US, AI Act in the EU) 8. Basic Principles of human-subjects research and principles beyond what the law requires (e.g., Belmont Report, UN Universal Declaration on Human Rights and how this relates to technology) 9. Freedom of expression and its limitations 10. User-generated content, content moderation, and liability	Explain Evaluate Explain Evaluate Explain		
SEP-Communication	1. Oral, written, and electronic team and group communication 2. Technical communication materials (e.g., source code and documentation, tutorials, reference materials, API documentation) 3. Communicating with different stakeholders such as customers, leadership, or the public 4. Team collaboration (including tools) and conflict resolution 5. Accessibility and inclusivity requirements for addressing professional audiences 6. Cultural competence in communication including considering the impact of difference in natural language	Apply Develop Apply Apply Apply Develop	CS	2
SEP-Communication	7. Tradeoffs in competing factors that affect communication channels and choices 8. Communicating to solve problems or make recommendations in the workplace, such as raising ethical concerns or addressing accessibility issues	Evaluate Apply	KA	1
SEP-Sustainability	1. Environmental, social, and cultural impacts of implementation decisions (e.g., sustainability goals, algorithmic bias/outcomes, economic viability, and resource consumption) 2. Local/regional/global social and environmental impacts of computing systems and their use (e.g., carbon footprints, resource usage, e-waste) in hardware (e.g., e-waste, data centers, rare element and resource utilization, recycling) and software (e.g., cloud-based services, blockchain,	Evaluate Evaluate	CS	1

	AI model training and use), not neglecting the impact of everyday use such as hardware (cheap hardware replaced frequently) and software (web-browsing, email, and other services with hidden/remote computational demands) 3. Guidelines for sustainable design standards	 Develop		
SEP-Sustainability	4. Systemic effects of complex computing technologies and phenomena (e.g., generative AI, data centers, social media, offshoring, remote work) 5. Pervasive computing – information processing that has been integrated into everyday objects and activities, such as smart energy systems, social networking, and feedback systems to promote sustainable behavior, transportation, environmental monitoring, citizen science and activism 6. How the sustainability of software systems is interdependent with social systems, including the knowledge and skills of its users, organizational processes and policies, and its societal context (e.g., market forces, government policies)	Evaluate Evaluate Explain	KA	1
SEP-History	1. The history of computing: hardware, software, and human/organizational 2. The role of history in the present including within different social contexts, and the relevance of this history on the future	Explain Evaluate	CS	1
SEP-History	3. Age I (Pre-digital) – Ancient analog computing (Stonehenge, Antikythera mechanism, Salisbury Cathedral clock, etc.), human-calculated number tables, Euclid, Lovelace, Babbage, Gödel, Church, Turing, pre-electronic (electro-mechanical and mechanical) hardware 4. Age II (Early modern computing) – ENIAC, UNIVAC, Bombes (Bletchley Park and codebreakers), computer companies (e.g., IBM), mainframes, etc. 5. Age III (PC era) – PCs, modern computer hardware and software, Moore's Law 6. Age IV (Internet) – Networking, internet architecture, browsers and their evolution,	Explain Explain Explain Explain	KA	1

	standards, born-on-the-internet companies, and services (e.g., Google, Amazon, Microsoft, etc.), distributed computing 7. Age V (Mobile & Cloud) – Mobile computing and smartphones, cloud computing and models thereof (e.g., SaaS), remote servers, security and privacy, social media 8. Age VI (AI) – Decision making systems, recommender systems, generative AI and other machine learning driven tools and technologies	Explain Explain		
SEP-Economies	1. Economic models – regulated and unregulated, monopolies, network effects, and open market; knowledge and attention economies 2. Pricing and deployment strategies – planned obsolescence, subscriptions, freemium, software licensing, open-source, free software, adware 3. Impacts of differences in access to computing resources, and the effect of skilled labor supply and demand on the quality of computing products 4. Automation, AI, and their effects on job markets, developers, and users 5. Ethical concerns surrounding the attention economy and other economies of computing (e.g., informed consent, data collection, use of verbose legalese in user agreements)	Explain Explain Evaluate Explain Evaluate	KA	1
SEP-Security	1. Computer crimes, legal redress for computer criminals and impact on victims and society 2. Social engineering, computing-enabled fraud, identity theft and recovery from these 3. Cyber terrorism, criminal hacking, and hacktivism 4. Malware, viruses, worms 5. Attacks on critical infrastructure such as electrical grids and pipelines 6. Non-technical fundamentals of security (e.g., human engineering, policy, confidentiality)	Explain Explain Explain Explain Explain	CS	2
SEP-Security	7. Benefits and challenges of existing and proposed computer crime laws 8. Security policies and the challenges of change and compliance 9. Responsibility for security throughout the computing life cycle	Evaluate Explain Explain	KA	1

	10. International and local laws and how they intersect	Explain		
SEP-DEIA	1. How identity impacts and is impacted by computing technologies and environments (academic and professional) 2. The benefits of diverse development teams and the impacts of teams that are not diverse 3. Inclusive language and charged terminology, and why their use matters 4. Inclusive behaviors and why they matter 5. Designing and developing technology with accessibility in mind 6. How computing professionals can influence and impact diversity, equity, inclusion, and accessibility, including but not only through the software they create	Explain Explain Apply Explain Explain Explain	CS	2
SEP-DEIA	7. Experts and their practices that reflect the identities of the classroom and the world through practical DEIA principles 8. Historic marginalization due to systemic social mechanisms, technological supremacy and global infrastructure challenges to diversity, equity, inclusion, and accessibility 9. Cross-cultural differences in, and needs for, diversity, equity, inclusion, and accessibility	Evaluate Explain Explain	KA	2

MSF: Mathematical and Statistical Foundations

KU	Topic	Skill Level	Core	Hours
MSF-Discrete	1. Sets, relations, functions, cardinality 2. Recursive mathematical definitions 3. Proof techniques (induction, proof by contradiction) 4. Permutations, combinations, counting, pigeonhole principle 5. Modular arithmetic 6. Logic: truth tables, connectives (operators), inference rules, formulas, normal forms, simple predicate logic 7. Graphs: basic definitions 8. Order notation	Apply, Develop, Explain	CS/KA	29-40

MSF-Probability	1. Basic notions: sample spaces, events, probability, conditional probability, Bayes' rule 2. Discrete random variables and distributions 3. Continuous random variables and distributions 4. Expectation, variance, law of large numbers, central limit theorem 5. Conditional distributions and expectation 6. Applications to computing, the difference between probability and statistics (as subjects)	CS-Core: Apply KA-Core: Apply, Develop, Explain	CS/KA	11-40
MSF-Statistics	1. Basic definitions and concepts: populations, samples, measures of central tendency, variance 2. Univariate data: point estimation, confidence intervals	Develop	CS	10
MSF-Statistics	3. Multivariate data – estimation, correlation, regression 4. Data transformation – dimension reduction, smoothing 5. Statistical models and algorithms 6. Hypothesis testing	Apply, Explain	KA	30
MSF-Linear	1. Vectors – definitions, vector operations, geometric interpretation, angles; Matrices – definition, matrix operations, meaning of Ax=b	Develop	CS	5
MSF-Linear	2. Matrices, matrix-vector equation, geometric interpretation, geometric transformations with matrices 3. Solving equations, row-reduction 4. Linear independence, span, basis 5. Orthogonality, projection, least-squares, orthogonal bases 6. Linear combinations of polynomials, Bezier curves 7. Eigenvectors and eigenvalues 8. Applications to computer science – PCA, SVD, page-rank, graphics	Apply, Explain	KA	35
MSF-Calculus	1. Sequences, series, limits 2. Single-variable derivatives – definition, computation rules (chain rule, etc.), derivatives of important functions, applications 3. Single-variable integration – definition, computation rules, integrals of important functions, fundamental theorem of calculus, definite vs indefinite, applications (including in probability) 4. Parametric and polar representations	Apply, Develop	KA	40

	5. Taylor series 6. Multivariate calculus – partial derivatives, gradient, chain-rule, vector valued functions, 7. Optimization – convexity, global vs local minima, gradient descent, constrained optimization, and Lagrange multipliers 8. ODEs – definition, Euler method, applications to simulation, Monte Carlo integration 9. CS applications – gradient descent for machine learning, forward and inverse kinematics, applications of calculus to probability			

Curricular Packaging

A few curricular packaging options of various sizes are presented here. These can be adapted to local strengths and needs to create a customized computer science curriculum. In each case, an effort should be made to include all the CS Core topics in required courses in the curriculum. The more KA Core topics covered, the greater the breadth of the curriculum. The more hours dedicated to KA Core topics, the greater the depth of the curriculum. Non-core topics add to the richness of the curriculum. In each curricular model, a capstone course is included to emphasize the importance of an integrative hands-on experience. It may also serve as the course where CS Core topics not covered elsewhere in the curriculum can be incorporated.

8 Course Model

This is a minimal course configuration that covers all the CS Core topics. However, it does not leave much room for exploration:

1. CS I (AL-2, FPL-1, SDF-34, SEP-3)
2. CS II (AL-18, FPL-4, MSF-4, SDF-9, SEC-1, SEP-3)
3. Mathematical and Statistical Foundations (MSF-40)
4. Algorithms (AL-12, MSF-11, PDC-2, SEP-3)
5. Introduction to Computing Systems (SF-18, OS-8, AR-9, NC-7, SEP-2)
6. Programming Language Concepts (FPL-16, PDC-7, SEP-2)
7. Introduction to Computing Applications (SEC-5, AI-12, GIT-4, DM-10, SEP-3)
8. Capstone (SE-6, HCI-8, SPD-3, SEP-2)

10 Course Model

1. CS I (SDF, SEP)
2. CS II (SDF, FPL-4, AL-12, SEP)
3. Mathematical and Statistical Foundations (MSF)
4. Data Structures and Algorithms (AL-20, AI, MSF, SEP)
5. Introduction to Computing Systems (SF, OS, AR, NC)
6. Programming Languages (FPL-17, AL, PDC, SEP)
7. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
8. One Systems elective:
    a. Operating Systems (OS, PDC)
    b. Computer Architecture (AR)
    c. Parallel and Distributed Computing (PDC)
    d. Networking (NC, SEC, SEP)
    e. Databases (DM, SEP)
9. One elective from Applications:
    a. Artificial Intelligence (AI, MSF, SPD, SEP)
    b. Graphics (GIT, HCI, MSF, SEP)

    c. Application Security (SEC, SEP)
    d. Human-Centered Design (HCI, GIT, SEP)
10. Capstone (SE, SEP)

12 Course Model

1. CS I (SDF, SEP)
2. CS II (SDF, AL-12, DM, SEP)
3. Mathematical and Statistical Foundations (MSF)
4. Algorithms (AL-20, AI, MSF, SEC, SEP)
5. Introduction to Computing Systems (SF, OS, AR, NC)
6. Programming Languages (FPL, AL, PDC, SEP)
7. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
8. Two from Systems electives:
    a. Operating Systems (OS, PDC)
    b. Computer Architecture (AR)
    c. Parallel and Distributed Computing (PDC)
    d. Networking (NC, SEC, SEP)
    e. Databases (DM, SEP)
9. Two electives from Applications:
    a. Artificial Intelligence (AI, MSF, SPD, SEP)
    b. Graphics (GIT, HCI, MSF, SEP)
    c. Application Security (SEC, SEP)
    d. Human-Centered Design (HCI, GIT, SEP)
10. Capstone (SE, SEP)

16 Course Model

Three different models are presented here, each with its own benefits.

Model 1:

1. CS I (SDF, SEP)
2. CS II (SDF, AL-12, DM, SEP)
3. Mathematical and Statistical Foundations (MSF)
4. Algorithms (AL-20, SEP)
5. Introduction to Computing Systems (SF, SEP)
6. Programming Languages (FPL, AL, PDC, SEP)
7. Theory of Computation (AL-32, SEP)
8. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
9. Operating Systems (OS, PDC, SEP)
10. Computer Architecture (AR, SEP)
11. Parallel and Distributed Computing (PDC, SEP)

12. Networking (NC, SEP)
13. Pick one of:
    a. Introduction to Artificial Intelligence (AI, MSF, SEP)
    b. Machine Learning (AI, MSF, SEP)
    c. Robotics (AI, SPD, SEP)
14. Pick one of:
    a. Graphics (GIT, MSF, SEP)
    b. Human-Centered Design (GIT, SEP)
    c. Animation (GIT, SEP)
    d. Virtual Reality (GIT, SEP)
15. Security (SEC, SEP)
16. Capstone (SE, SEP)

Model 2:

1. CS I (SDF, SEP)
2. CS II (SDF, AL, DM, SEP)
3. Mathematical and Statistical Foundations (MSF, AI, DM)
4. Algorithms (AL, MSF, SEP)
5. Introduction to Computing Systems (SF, SEP)
6. Programming Languages (FPL, AL, PDC, SEP)
7. Theory of Computation (AL, SEP)
8. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
9. Operating Systems (OS, PDC, SEP)
10. Two electives from:
    a. Computer Architecture (AR, SEP)
    b. Parallel and Distributed Computing (PDC, SEP)
    c. Networking (NC, SEP)
    d. Network Security (NC, SEC, SEP)
    e. Security (SEC, SEP)
11. Pick three of:
    a. Introduction to Artificial Intelligence (AI, MSF, SEP)
    b. Machine Learning (AI, MSF, SEP)
    c. Deep Learning (AI, MSF, SEP)
    d. Robotics (AI, SPD, SEP)
    e. Data Science (AI, DM, GIT, MSF)
    f. Graphics (GIT, MSF, SEP)
    g. Human-Computer interaction (HCI, SEP)
    h. Human-Centered Design (GIT, HCI, SEP)
    i. Animation (GIT, SEP)
    j. Virtual Reality (GIT, SEP)
    k. Physical Computing (GIT, SPD, SEP)
12. Society, Ethics, and the Profession (SEP)
13. Capstone (SE, SEP)

Model 3:

1. CS I (SDF, SEP)
2. CS II (SDF, AL, DM, SEP)
3. Mathematical and Statistical Foundations (MSF)
4. Algorithms (AL, AI, MSF, SEC, SEP)
5. Introduction to Computing Systems (SF, OS, AR, NC)
6. Programming Languages (FPL, AL, PDC, SEP)
7. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
8. Two from Systems electives:
    a. Operating Systems (OS, PDC)
    b. Computer Architecture (AR)
    c. Parallel and Distributed Computing (PDC)
    d. Networking (NC, SEC, SEP)
    e. Databases (DM, SEP)
9. Two electives from Applications:
    a. Artificial Intelligence (AI, MSF, SPD, SEP)
    b. Graphics (GIT, HCI, MSF, SEP)
    c. Application Security (SEC, SEP)
    d. Human-Centered Design (HCI, GIT, SEP)
10. Three open CS electives
11. Society, Ethics, and the Profession (SEP) course
12. Capstone (SE, SEP)

Competency Framework Examples

Sample Tasks

Sample tasks have been listed here for various combinations of component, activity and constraint for the three representative competency areas: software, systems and applications. Many, but not all the tasks are atomic. The tasks are not restricted to those that require CS or KA Core topics only.

Software Competency Area

Component – Activity	**Tasks with constraint(s) *italicized***
Program – Design	● Design *efficient* data structures for a problem ● Design test cases to determine if a program is *functionally correct* ● Design an API for a *service*
Program – Develop	● Write a program that *meets a given specification* ● *Automate* testing of new code under development ● Develop a program that *leverages libraries and APIs* ● *Work in a team effectively* to solve a problem
Program – Document	● Document a program ● *Consistently* format source code.
Program – Evaluate	● Evaluate an existing application (open source or proprietary) as a whole or partial solution for *meeting a defined requirement*
Program – Maintain	● Refactor a program. ● Perform code review to evaluate the quality of code
Program – Humanize	● Ensure *fair and equitable access* in a program ● Document the *accountability, responsibility, and liability* an individual/company assumes when releasing a given service/software/product ● Incorporate *legal and ethical privacy requirements* into a given service/software/product's development cycle ● Convey the *benefits of diverse development teams and user bases* on the services/software/products the company provides, as well as the impacts that a lack of diversity can have on them
Program – Improve	● Debug a program
Program – Research	● Compute the running time of a program ● Formally prove the *correctness* of code
Algorithm – Design	● Design an *efficient* algorithm for a problem
Algorithm – Document	● Explain how an algorithm *satisfies a set of requirements*
Algorithm – Evaluate	● Evaluate the *efficiency* of an algorithm

Algorithm – Maintain	• Redesign an algorithm to *improve a non-functional requirement*
Algorithm – Humanize	• Justify that an algorithm provides *fair and equitable access* to data
Algorithm – Research	• Prove the *correctness* of an algorithm • Compute the *run time efficiency* of an algorithm
Language/Paradigm – Design	• Select an *appropriate* language/paradigm for an application
Language/Paradigm – Document	• Justify the choice of a language/paradigm for a program • Write a white paper to describe how a program is translated into machine code and executed • White a white paper explaining how a program executes in an *efficient manner* with respect to memory and CPU utilization
Language/Paradigm – Evaluate	• Evaluate the *appropriateness* of a language/paradigm for an application • Explain the benefits and challenges of converting an application into *parallel/distributed versions* • Write a white paper explaining how a program effectively utilizes language features to make it *safe and secure*

Systems Competency Area

Component – Activity	**Tasks with constraint(s) *italicized***
Processor – Design	• Revise a specification to enable *parallel processing* without violating other essential properties or features
Processor – Develop	• Develop a version of CPU-based application to *run on a hardware accelerator* (GPU, TPU, NPU) • Implement a *parallel/distributed* version of a known algorithm
Processor – Evaluate	• Evaluate the *performance-watt* of a machine learning model deployed on an *embedded device*
Processor – Improve	• Identify and repair a *performance* problem due to sequential bottlenecks
Storage – Evaluate	• Assess the *performance implications* of cache memories in an application • Apply knowledge of operating systems to assess page faults in CPU-GPU memory management and their *performance impact* on an accelerated application
I/O – Design	• Design software modules for *sensor hardware integration*
I/O – Develop	• Develop a sensing-actuator robotics arm for an automated manufacturing cell

	• Develop a benchmarking software tool to assess the *performance gain in removing I/O bottlenecks* in code
Communication – Design	• Design a networking protocol. • Design software that enables *safe* communication between processes
Communication – Develop	• Develop a networked application • Deploy and securely operate a network of wireless sensors. • Develop software that enables *safe communication* between processes
Communication – Evaluate	• Evaluate the performance of a network, in specific *latency, throughput, congestion, and various service levels*
Communication – Maintain	• Defend a network from an *ongoing distributed* denial-of-service attack
Communication – Humanize	• Write a white paper to explain *social, ethical, and professional issues* governing the design and deployment of a networked system
Communication – Improve	• Identify failures in a datacenter network • Identify and repair a *performance problem* due to communication or data latency
Architecture – Develop	• Deploy a system in a *cloud environment* • Deploy an application component on a *virtualized container*
Architecture – Evaluate	• Find the *performance bottleneck* of a given system architecture
Data – Design	• Design how a new application's data will be stored
Data – Develop	• Create a database for a new application
Data – Maintain	• Get data back online after a disruption (e.g., power outage)
Data – Humanize	• Produce a white paper assessing the *social and ethical implications* of collecting and storing the data from a new (or existing) application • Assess the *legal and ethical implications* of collecting and using customer/user data
Data – Improve	• Improve a database application's *performance (speed)* • Modify a concurrent system to use a more *scalable, reliable or available* data store

Applications Competency Area

Component – Activity	**Tasks with constraint(s) *italicized***
Input – Design	• Design an *intuitive* user interface for an application
Input – Develop	• Implement the user interface of an application
Input – Humanize	• Write a paper on the *accessibility* of a user interface

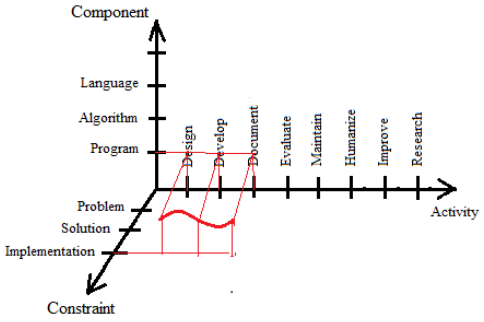
Computation – Design	• Specify the operators and partial-order planning graph to solve a logistics problem, showing all ordering constraints
Computation – Develop	• Implement an agent to play a *two-player complete information* board game • Write a program that uses Bayes rule to predict the probability of disease given the conditional probability table and a set of observations • Train a neural network to play a video game (e.g., Mario, Atari)
Computation – Evaluate	• Compare the *performance* of three supervised learning models on a dataset • Explain some of the *pitfalls* of deep generative models for image or text and how this can affect their use in an application
Computation – Humanize	• Write an essay on the effects of data set bias and how to mitigate them
Platform – Design	• Determine whether to develop an app as a *native app or as a cross-platform app*
Platform – Develop	• Create a mobile app that provides a *consistent user experience* across various devices, screen sizes, and operating systems • Develop a *secure* web interface for a business application
Platform – Evaluate	• Evaluate the *usability and accessibility* of an immersive system
Platform – Improve	• Optimize a *dynamic* web site for *evolving* business needs

Sample Competency Specifications

The following are some sample competency specifications for tasks that require various knowledge areas and skills. They have been listed under the three competency areas, that is, Software, Systems, and Applications.

Software Competency Area

- **Task Software1:** Develop test cases to determine if a program is functionally correct.
- **Competency statement:** Develop test cases and test a given program.
- **Required knowledge:**
  - SDF-Practices
- **Required skills:** Develop
- **Desirable professional dispositions:** Meticulous, Persistent, Responsible



This is a compound task: the component is a set of test cases for a program; the activities are design, develop, and document; and the constraint is verifying the correctness of implementation of the program.

- **Task Software2:** Perform code review for a teammate.
- **Competency statement:** Communicate clearly and collaboratively to provide feedback to a teammate about a piece of code.
- **Required knowledge:**
  - SDF-Practices
  - SE-Teamwork
  - SE-Validation
- **Required skills:** Apply
- **Desirable professional dispositions:** Collaborative, Communicative, Meticulous

- **Task Software3:** Work on a team effectively.
- **Competency statement:** Focus on long-term team dynamics and communicate effectively.
- **Required knowledge:**
  - SE-Teamwork
- **Required skills:** Apply
- **Desirable professional dispositions:** Collaborative, Communicative, Proactive, Responsive

- **Task Software4:** Make an informed decision regarding which programming language/paradigm to select and use for a specific application.
- **Competency statement:** Apply knowledge of multiple programming paradigms, including their strengths and weaknesses relative to the application to be developed, and select an appropriate paradigm and programming language.
- **Required knowledge:**
  - FPL-OOP
  - FPL-Functional
  - FPL-Logic
  - FPL-Event-Driven
  - FPL-Types
  - FPL-Translation
  - FPL-Pragmatics

    - SPD-Embedded
    - FPL- Constructs
- **Required skills:** Explain, Evaluate
- **Desirable professional dispositions:** Inventive

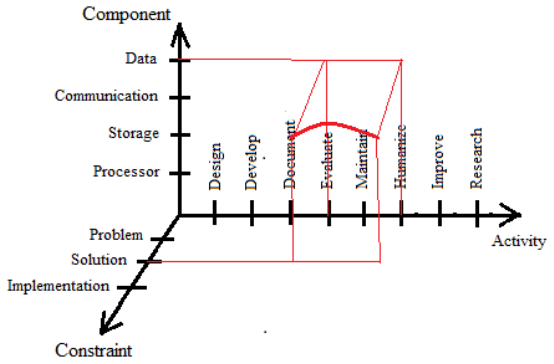
- **Task Software5:** Effectively use a programming language's type system to develop safe and secure software.
- **Competency statement:** Apply knowledge of static type rules for a language to ensure an application is safe, secure, and correct.
- **Required knowledge:**
    - FPL-Types
- **Required skills:** Develop
  Desirable professional dispositions: Inventive, Meticulous

Systems Competency Area

- **Task Systems1:** Develop a version of a CPU-based application to run on a hardware accelerator (GPU, TPU, NPU).
- **Competency statement:** Apply knowledge from systems design to accelerate an application code and evaluate the code speed-up.
- **Required knowledge:**
    - AR-Heterogeneity
    - PDC-Programs
    - SF-Design
- **Required skills:** Evaluate, Develop
- **Desirable professional dispositions:** Meticulous, Inventive

- **Task Systems2:** Produce a white paper assessing the social and ethical implications of collecting and storing data from a new (or existing) application.
- **Competency statement:** Identify the stakeholders and evaluate the potential long-term consequences of the collection and retention of data objects. Consider both potential harm from unintended data use and from data breaches.
- **Required knowledge:**
    - SEP-Context
    - SEP-Ethical-Analysis
    - SEP-Privacy
    - SEP-Professional-Ethics
    - SEP-Security
    - SEP- DEIA
    - DM-Data
    - SEC-Foundations
- **Required skills:** Evaluate, Explain

- **Desirable professional dispositions:** Meticulous, Responsible, Proactive



This is a compound task: the component is data; the activities are *evaluate* and *humanize*; the constraint is the *design* of the solution vis-à-vis social and ethical implications.

- **Task Systems3:** Secure data from unauthorized access.
- **Competency statement:** Create database views to ensure data access is appropriately limited.
- **Required knowledge:**
  - DM-Data
  - DM-Relational
  - DM-Processing
  - SEP-Security
  - SEP-Professional-Ethics
  - SEP-Privacy
  - SEC-Foundations
- **Required skills:** Develop
- **Desirable professional dispositions:** Meticulous, Proactive

- **Task Systems4:** Create a database for a new application.
- **Competency statement:** Design the data storage needs (data modeling), assess the social and ethical implications for collecting and storing the data, determine how to store a new application's data (RDBMS vs NoSQL), and create the database, including appropriate indices.
- **Required knowledge:**
  - DM-Data
  - DM-Core
  - DM-Modeling
  - DM-Relational
  - DM-NoSQL
  - DM-Internals
  - SEP-Context
  - SEP-Ethical-Analysis
  - SEP-Privacy

    - SEP-Professional-Ethics
    - SEP-Security
    - SEP- DEIA
    - SEC-Foundations
- **Required skills:** Develop
- **Desirable professional dispositions:** Inventive, Meticulous, Responsible

- **Task Systems5:** Evaluate the performance of a network.
- **Competency statement:** Evaluate the latency, throughput, congestion, and various service levels of a network.
- **Required knowledge:**
    - NC-Applications
    - NC-Routing
- **Required skills:** Evaluate
- **Desirable professional dispositions:** Meticulous, Proactive

- **Task Systems6:** Deploy an application component on an operating system/ runtime/virtualized operating system/container.
- **Competency statement:** Identify and mitigate potential problems with deployment; automate setup of deployment environment; set up monitoring of component execution.
- **Required knowledge:**
    - OS-Purpose
    - OS-Principles
    - OS-Concurrency
    - OS-Scheduling
    - OS-Process
    - OS-Memory
    - OS-Protection
    - AR-Assembly
    - FPL-Scripting
- **Required skills:** Apply
- **Desirable professional dispositions:** Meticulous, Persistent, Proactive

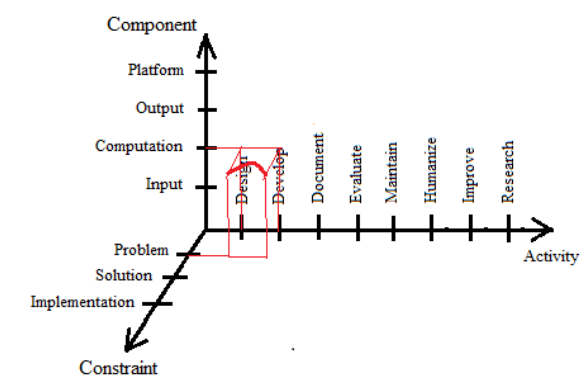
- **Task Systems7:** Improve the performance of a sequential application or component by introducing parallelism.
- **Competency statement:** Evaluate how and when parallelism can improve (or not improve) performance well enough to identify opportunities, as well as implement them and measure the results.
- **Required knowledge:**
    - PDC-Evaluation

    - FPL-Parallel
- **Required skills:** Evaluate, Develop
- **Desirable professional dispositions:** Meticulous, Persistent, Proactive

- **Task Systems8:** Find the performance bottleneck of a given system.
- **Competency statement:** Given a system and its target deployment environment, find its performance bottleneck (e.g., memory, CPU, networking) through analytical derivation or experimental study.
- **Required knowledge:**
    - SF-Performance
    - SF-Evaluation
    - SF-Design
    - SF-Overview
- **Required skills:** Apply, Evaluate, Develop
- **Desirable professional dispositions:** Meticulous, Persistent

Applications Competency Area

- **Task Applications1:** Implement an agent to make strategic decisions in a two-player adversarial game with uncertain actions (e.g., a board game, strategic stock purchasing).
- **Competency statement:** Use minimax with alpha-beta pruning, and possible chance nodes (expectiminimax), and heuristic move evaluation (at a particular depth) to solve a two-player zero-sum game.
- **Required knowledge:**
    - AI-Search
    - AI-KRR
- **Required skills:** Apply, Develop
- **Desirable professional dispositions:** Inventive, Persistent



This is a compound task: the component is computation, i.e., an algorithm designed for the specific problem of two-player game; the activities are design and develop; the constraints are imposed by the problem – adversarial game with uncertain actions.

- **Task Applications2:** Analyze tabular data (e.g., customer purchases) to identify trends and predict variables of interest.
- **Competency statement:** Use machine learning libraries, data preprocessing, training infrastructures, and evaluation methodologies to create a basic supervised learning pipeline.
- **Required knowledge:**
    - AI-ML
    - AI-SEP
- **Required skills:** Apply, Develop
- **Desirable professional dispositions:** Meticulous, Persistent, Responsible

- **Task Applications3:** Critique a deployed machine learning model in terms of potential bias and correct the issues.
- **Competency statement:** Understand, recognize, and evaluate issues of data set bias in AI, the types of bias, and algorithmic strategies for mitigation.
- **Required knowledge:**
    - AI-ML
    - AI-SEP
- **Required skills:** Explain
- **Desirable professional dispositions:** Responsible

- **Task Applications4:** Visualize a region's temperature record.
- **Competency statement:** Given weather data for a region, design and implement an animation depicting temperature changes over time.
- **Required knowledge:**
    - GIT-Fundamentals
    - GIT-Rendering
    - GIT-Visualization
    - HCI-Design
    - HCI-User
- **Required skills:** Apply, Evaluate, Develop
- **Desirable professional dispositions:** Inventive, Persistent

- **Task Applications5:** Evaluate and provide recommendations to improve a user-facing system.
- **Competency statement:** Apply knowledge of usability, accessibility, and inclusivity to evaluate a user-facing system.
- **Required knowledge:**
    - HCI-User
    - HCI-Accessibility
    - HCI-Evaluation
    - HCI-Design

    - HCI-SEP
- **Required skills:** Evaluate
- **Desirable professional dispositions:** Meticulous, Responsible

- **Task Applications6:** Determine the aspects of an implementation that require revision to support internationalization.
- **Competency statement:** Evaluate a system to identify culturally-relevant or language-relevant text, symbols, and patterns that may vary by locale.
- **Required knowledge:**
    - HCI-User
    - HCI-Accountability
    - HCI-Accessibility
    - HCI-Evaluation
- **Required skills:** Evaluate
- **Desirable professional dispositions:** Meticulous, Proactive, Responsible

- **Task Applications7:** Document the professional implications of a service/software/product for the company that produced it.
- **Competency statement:** Gather information regarding accountability, responsibility and liability assumed by a company when releasing a service/software/product and present it in a coherent and actionable manner.
- **Required knowledge:**
    - SEP-Professional-Ethics
    - SEP-IP
    - SEP-Privacy
    - SEP-Communication
    - SEP-DEIA
- **Required skills:** Explain
- **Desirable professional dispositions:** Meticulous, Proactive, Responsible

- **Task Applications8:** Determine whether to develop an app as a native app or as a cross-platform app.
- **Competency statement:** Understand performance and scalability issues, and evaluate different approaches and tools by carefully considering factors such as app requirements, target audience, time-to-market, and costs.
- **Required knowledge:**
    - SE-Tools
    - SPD-Common
    - SPD-Mobile
- **Required skills:** Explain
- **Desirable professional dispositions:** Inventive, Meticulous, Responsible

- **Task Applications9:** Build and optimize a secure web page for evolving business needs using a variety of *appropriate* programming languages.
- **Competency statement:** Evaluate potential security hazards and apply optimization techniques.
- **Required knowledge:**
    - AR-Performance-Energy
    - NC-Security
    - OS-Protection
    - SF-Security
    - SE-Design
    - SE-Tools
    - SPD-Common
    - SPD-Mobile
    - SEP-Privacy
- **Required skills:** Develop
- **Desirable professional dispositions:** Adaptable, Meticulous, Proactive, Responsible

Pedagogy and Practices

1. Pedagogical Considerations
2. Curricular Practices in Computer Science
3. Generative AI and the Curriculum

Pedagogical Considerations

Introduction

What are some current trends in the teaching and learning of computer science? What are the controversies of the day in terms of the pedagogy of computer science education? In this section, a top few trends, controversies, and challenges have been listed for each knowledge area as well as the curriculum as a whole. These issues are expected to influence the future evolution of computer science curricula.

Curriculum-Wide Considerations

The adoption of CS2023's recommendations presents numerous challenges at both the macro and the micro level. Some of the macro considerations include the following.

- Active learning is an important component of any computer science course – doing helps learn computer science. Courses that use interactive electronic resources (e.g., eBooks, python notebooks) are a significant improvement over the traditional lecture-based courses that do not involve any active learning component – they provide ample opportunities to learn by solving problems. In this regard, it is important to emphasize that ideally, active learning should cover the entire gamut of skill levels – not just *apply*, but also *evaluate* and *develop*.
- The success of generative AI systems is causing faculty to reconsider their approach to assessment. Student grades are increasingly being based on in-class, closed-book assessments. Care should be taken not to lose sight of the objective of assessment when using expedient assessment techniques.
- The landscape of computer science textbooks is in flux. The number of traditional publishing houses is shrinking, the cost of their offerings is increasing, and the vast majority of these resources are primarily only in English – though generative AI systems are making important inroads in providing accurate translations. Digital rentals, while cheaper, usually disappear at term's end and are unavailable as future references. Open Educational Resources (OER) present a cost-effective alternative, but often lack formal review. Compounding this are students who, for financial and generational reasons, forgo acquiring the selected text(s) in favor of self-selected YouTube videos.
- Broadening participation in computing is an ongoing concern in many contexts. Adopters of CS2023 should remain cognizant of how their curricular and pedagogical choices affect this important issue.
- Studying abroad is a quintessential high impact learning experience. One potential impediment to its wide-spread adoption is curriculum design. This can include long prerequisite chains and dense curricula. Care should be taken to leave room for, if not encouraged, penalty-free study abroad.

Considerations by Knowledge Area

Artificial Intelligence (AI)

- A balance must be struck between 1) the need to study fundamental issues of search and other approaches that are still in widespread use and 2) the desire to focus on cutting-edge AI and machine learning methods.
- Given AI's current and future potential for societal impact, educators should ensure that all students are well-versed in the ethical and societal considerations and implications of applying AI methods.
- Since AI is rapidly evolving as a field, it is challenging to create a curriculum that will remain current for long. Consequently, the onus is on the instructors to keep up to date with current methods and use their judgment in determining what to teach in order to keep their courses current.

Algorithmic Foundations (AL)

- Should computer science graduates be able to explain, at some level, algorithmic approaches making headlines in the popular press, such as Blockchain, SHA-246, and Quantum Computing algorithms? Topics focused on these, and other algorithms, received less than 50% support in the community survey of Algorithmic Foundations area. Consequently, they were not listed as CS Core topics. However, if computer science graduates are expected to explain these algorithms to non-technical members of the society, they should be exposed to these topics somewhere in the curriculum.
- Students are unlikely to implement a lot of historic algorithms in industry such as Bubble sort. A question that merits case-by-case consideration is whether these algorithms should be covered in the curriculum and, if so, the level of skill at which they should be covered.

Architecture and Organization (AR)

- Software tools for open hardware have lowered the cost and complexity of understanding the design of new processors. The community-driven RISC-V open instruction set architecture "democratizes" the design and evaluation of processors and presents a cutting-edge opportunity in this regard.
- Quantum Computing does not yet have a fully standardized interface between software and hardware. What should be the minimum set of Quantum Computing topics a computer science graduate should be able to explain?

Data Management (DM)

- For the most part, students write code that either reads/writes to a file or is interactive. Yet, in industry, most data are obtained programmatically from a database. This is a source of mismatch between academic preparation and industry expectation.
- Even though most databases are relational, NoSQL databases are enjoying a significant degree of popularity. Balancing the coverage of relational vs NoSQL databases is a concern for curricula.

Foundations of Programming Languages (FPL)

- Shell scripting is a skill that students should master to automate laborious tasks. It is also a helpful tool to coordinate the work of multiple applications. An interesting curricular exercise is determining how and when to teach it as a paradigm.
- There is an increasing need to develop large, complex software systems that have the potential for catastrophic failure (e.g., software driving medical devices such as robotic surgery). Such software needs to have its behavior validated and potentially formally proved correct. As a result, formal methods may be more important in the future. This would require greater mathematical skills and ability in graduates.

Mathematical and Statistical Foundations (MSF)

Faculty and students alike have strong opinions about how much and what mathematics should be included in the CS curriculum. Generally, faculty, who themselves have strong theoretical training, are typically concerned about poor student preparation and motivation to learn mathematics, while students complain about not seeing applications and wonder what any of the mathematics has to do with the software jobs they seek. Even amongst faculty, there is recurring debate on whether calculus should be required of computer science students, especially in light of the impact calculus failure rates have on broadening participation. Yet, at the same time, the discipline has itself undergone a significant mathematical change – machine learning, robotics, data science, and quantum computing all demand a different kind of mathematics than is typically covered in a standard discrete structures course. The combination of changing mathematical demands and inadequate student preparation or motivation, in an environment of enrollment-driven strain on resources, has become a key challenge for CS departments. Some recommendations that have been presented for the treatment of mathematics in computer science programs follow.

- Requiring *PreCalculus* as a prerequisite for discrete mathematics will ensure that students enter computer science with some degree of comfort with symbolic mathematics and functions.
- Studies show that students are motivated when they see applications of mathematics. It is recommended that minor programming assignments and demonstrations of applications of mathematics be included in computer science courses.
- Institutions should adopt preparatory options to ensure sufficient background without lowering standards in mathematics. Theory courses can be moved further back in the curriculum to accommodate first-year preparation. Where possible, institutions can offer online self-paced tutoring systems alongside regular coursework.
- What is clear, when looking forward to the next decade, is that exciting high-growth areas of computer science will require a strong background in linear algebra, probability, and statistics (preferably calculus-based). And as much of this material as possible should be included in the standard curriculum.
- Educators and institutions are often under pressure to help every student succeed, many of whom struggle with mathematics. While pathways, including computer science-adjacent degrees or tracks, are sometimes created to steer students around mathematics requirements towards software-focused careers, educators should be equally direct in explaining the importance of sufficient mathematical preparation for graduate school and for the very topical areas that excite students.

The better approach is to invest in remediation courses to sufficiently prepare as many students as possible.

Networking and Communication (NC)

- To what extent should students learn the very low-level details associated with networking or should higher levels of abstraction be the norm in teaching?
- Cutting-edge technologies promise to significantly affect networking. Generative AI might benefit the generation of networked configurations, security assessments, and capacity planning. Quantum computing may significantly affect the teaching and practice of networking. And the same goes for emerging communication technologies like 5G.

Operating Systems (OS)

- How do we teach operating systems as a cohesive set of functions when OS functions are increasingly embedded in architectures or distributed within software development frameworks?
- Educators should continue to emphasize the importance of operating systems knowledge in distributed, parallel, and secure applications.
- Is there value in having students recreate operating system functions as a pedagogical approach or should the focus be on a student's ability to reason about the performance of off-the-shelf library modules that compose applications?

Parallel and Distributed Computing (PDC)

- Should parallel and distributed programming be infused across a curriculum?

Software Development Fundamentals (SDF)

- Students will still need to be able to produce code. How they go about doing so may change rapidly and dramatically with improvements in the capabilities of generative AI. Students should also be able to read, critique, and verify the correctness of code — abilities that students will need if generative AI is used to write code.

- Assessment practices in introductory programming courses will need to be adapted to take into account the availability of generative AI. How remains to be seen. It is foreseeable that currently popular assessment approaches such as drill-and-practice, many-small-problems, and written Explain in Plain English (EiPE) assessments, will need to change or be utilized differently. A focus on what today would be considered 'alternative' means of assessment may rise in prominence—for instance, oral examination, code modification, and other difficult-to-accomplish (with generative AI) assessment schemes.
- High-level approaches to teaching and learning introductory programming may need to be dramatically rethought. Approaches similar to studio models of learning from the fine arts where students design, show, explain, and critique work made in the studio are a way to engage students with topics that let them express their own ideas and vision while learning about fundamental topics.

- The order in which Software Development Fundamentals (SDF) topics are discussed may need to be reconsidered. For instance, starting with syntax and creating increasingly complex programs from scratch may be replaced by concepts-first approaches where modularity is considered from the beginning, and syntax and other more basic constructs such as conditionals and repetition are learned during the process.
- New skills such as prompting (prompt engineering) and the use of other generative AI tool requirements/features may in the near future be considered to be basic programming skills. How the industry adopts generative AI may be a leading driver in such arenas.

Software Engineering (SE)

- Are undergraduate programs in computing that rely on a *single* team project beneficial? Teamwork is ubiquitous in industry, but no teams are formed entirely out of people with similar backgrounds who have no prior experience working on a team. The heterogeneous background and experience level of real teams is fundamentally different from the comparative homogeneity of students in a class.
- Do students have sufficient opportunity to practice with open-ended problems, where the choice of tools and approach are critical? In a software engineering context, most work involves evaluating tradeoffs – between space and time, between speed-of-implementation and runtime optimization, between Do-it-Yourself (DIY) and Commercial-off-the-shelf (COTS) or Open-Source Software (OSS) approaches. Are students given enough opportunities to practice the critical decision-making skills necessary to succeed in a professional environment?
- Software developed in a team setting (software engineering rather than programming) is more likely to have an impact on society for good or ill. At the same time, teamwork means no single person may be responsible for the impact of the software—the larger the project, the greater the potential impact, but the less responsible any team member feels about the impact. How do we instill among students a proper sense of responsibility for the whole solution regardless of the size of one's contribution to it?
- Formal methods for software validation will pay off (substantially) later in students' careers, but it is a perfect-but-infrequent solution. The current approach to validation focuses on attempts to get high-fidelity evidence (unit tests) over proofs and other formal methods, which is more immediately useful but fails to expose students to interesting long-term ideas. Given the finite resources provided for software engineering education, educators must strike the right balance between these two approaches.

Security (SEC)

- An ongoing pedagogical consideration is inculcating a security mindset among students, so that security is a goal from the start and not an afterthought.

Society, Ethics, and the Profession (SEP)

- Is an introduction to ethical thinking and an awareness of social issues and their emerging professional responsibilities sufficient for our graduates to act ethically and responsibly? If not, does that put the burden on instructors to not only lead discussion about the pressing questions of the

day but also weigh in on those matters? How should that be done? Will we just be imparting our own biases upon our students? Should this be avoided? If so, how?
- CS curricula regularly require prerequisite courses in mathematics. Should there be parallel prerequisite requirements in moral philosophy to prepare students for future course work in performing ethical analysis?
- How could we weave SEP throughout the curriculum in practice? Is this realistic? How much coordination would it take? How less optimal is it to have a standalone ethics course only? Is there another model in between these two extremes (neglecting the extreme of not having any coordinated or targeted SEP content in our courses)?
- Educators naturally possess real or perceived authority when it comes to technical issues. Many educators believe they lack this authority when it comes to SEP topics. How can CS instructors be empowered to effectively incorporate SEP topics into their courses?
- How can we effectively impart the core values of diversity, equity, inclusion, and accessibility? How is this best done in a computer science context? How can we effectively impart the core values and skills of being a computing professional into our students' education? Are engineered "toy" projects a suitable context for these? Are study abroad opportunities/work-placements/internships better? Are they worth the cost? Should we put more focus on efforts to have more programs/degrees contain study abroad opportunities/placements/internships?
- Should software developers be licensed like engineers, architects, and medical practitioners? This is an older debate but given the impact of software systems (akin to safe bridges, buildings, etc.), maybe it is time to revisit this question.
- What would a set of current SEP case studies look like and how could they be employed effectively?
- How can collateral learning be leveraged to improve the learning and appreciation of societal, ethical, and professional topics?

Specialized Platform Development (SPD)

- Computing is no longer limited to traditional desktop applications. Students need to learn how to develop software solutions for various platforms, including web, mobile, IoT devices, and emerging platforms like virtual reality (VR) and augmented reality (AR). A well-rounded curriculum should provide opportunities for learning on multiple platforms.
- With the increasing demand for cross-platform apps, educators should teach students how to develop applications that work seamlessly across different operating systems and devices using technologies such as React Native, Flutter, or Progressive Web Apps (PWAs).
- Cloud platforms and services, such as AWS, Azure, and Google Cloud, have become integral to modern platform development. Students need to learn how to deploy, scale, and manage applications in the cloud.

Systems Fundamentals (SF)

- How deeply should instructors elaborate on the design principles of computer systems in undergraduate courses?

- What role should generative AI play in system-related knowledge areas, not only systems fundamentals, but also architecture and organization, network and communication, operating systems, and parallel and distributed computing?
- Should instructors link knowledge units from systems-related knowledge areas with those from applications-related knowledge areas? And if so, how?

Curricular Practices in Computer Science

Introduction

Prior curricular guidelines enumerated issues in the design and delivery of computer science curriculum. Given the increased importance of these issues, in CS2023, peer-reviewed, well-researched, in-depth articles were solicited from recognized experts on how computer science educators could address these issues in their teaching practices. These articles complement the CS2023 curricular guidelines. Whereas curricular guidelines list what should be covered in the curriculum, these articles describe how and why they could best be covered, including challenges, state of the art practices, etc.

The articles may be categorized as covering the following.

- **Social aspects,** including teaching about accessibility, computer science for social good, responsible computing, and ethics in the global souths.
- **Pedagogical considerations,** including CS + X, the role of formal methods in computer science, quantum computing education, and the impact of generative AI on programming instruction.
- **Educational practices,** in varied settings such as liberal arts institutions, community colleges, and polytechnic institutes.

The articles provide a “lay of the land,” a snapshot of the current state of the art of computer science education. They are not meant to advocate specific approaches or viewpoints, but rather help computer science educators weigh their options and make informed decisions about the appropriate option for their degree program.

The computer science education community was invited to provide feedback and suggestions on the first drafts of most of these articles. Several of the articles have been or are in the process of being published in peer-reviewed conferences and journals. In this section, self-contained summaries of most of the articles have been included. The full articles themselves will be accessible at the csed.acm.org website.

In addition, to globalize computer science education, articles were also invited on educational practices in various parts of the world. (See *ACM Inroads*, Special Issue, 15, 1 (March 2024)). It is hoped that these articles will foster mutual understanding and exchange of ideas, engender transnational collaboration and student exchange, and serve to integrate computer science education at the global level through shared understanding of its challenges and opportunities.

Social Aspects

Accessibility is about making computing systems accessible to people with disabilities and designing technical solutions for accessibility problems faced by people with disabilities. The article “**Teaching**

about Accessibility in Computer Science Education" explains the practical, intellectual, and social reasons for integrating accessibility into the computer science curriculum.

The article "**Computing for Social Good in Education**" highlights how computing education can be used to improve society and address societal needs while also providing authentic computing environments in education. The authors discuss approaches, challenges, and benefits of incorporating computing for social good into computer science curriculum.

Given the pervasive use of computing in society, educators would be remiss not to teach their students about the principles of responsible computing. How they should go about doing so is explored in the article "**Multiple Approaches for Teaching Responsible Computing**." It uses research in the social sciences and humanities to transform responsible computing into an integrated consideration of values throughout the lifecycle of computing products.

In a globalized world, applications of computing transcend national borders. In this context, making ethics at home in global computer science education is about helping students relate to values within and beyond their own contexts. The article "**Making Ethics at Home in Global CS Education: Provoking Stories from the Souths**" presents storytelling as a mechanism that educators can use to engage students with "ethos building."

Pedagogical Considerations

"**CS + X: Approaches, Challenges, and Opportunities in Developing Interdisciplinary Computing Curricula**" states how interdisciplinary majors that apply computational methods in natural sciences, social sciences, humanities, and the arts can broaden participation in computing and reach a larger group of students.

"**The Role of Formal Methods in Computer Science Education**" makes the case for incorporating formal methods in computer science education. It lists the multiple ways in which formal methods can be incorporated into the undergraduate computer science curriculum and buttresses its advocacy of formal methods with testimonials from the industry.

"**Quantum Computing Education: A Curricular Perspective**" presents the current state of art in quantum computing and uses the results of a pedagogic experiment to illustrate that quantum computing education is within reach of even school children. It presents three curricular approaches for incorporating quantum computing in undergraduate computer science curriculum.

"**Generative AI in Introductory Programming**" explores how generative AI tools based on Large Language Models (LLMs) such as ChatGPT might affect programming education including how these tools can be used to assess student work, provide feedback, and to act as always-available virtual teaching assistants in introductory programming courses.

One issue with the study of databases/data management is that the number of possible topics far exceeds the bandwidth of a single undergraduate computer science course. "**The 2022 Undergraduate**

Database Course in Computer Science: What to Teach?" presents multiple viewpoints on what a single undergraduate course in Databases/Data Management should cover.

Educational Practices

No curricular guidelines are complete by themselves. They must be adapted to local strengths, constraints, and needs. In this regard, "**Computer science Curriculum Guidelines: A New Liberal Arts Perspective**" provides a process to adapt CS2023 to the needs of liberal arts colleges that constrain the size of the computer science coursework in order to expose students to a broad range of liberal arts subjects.

Community and polytechnic colleges across the world offer specialized programs that help students focus on specific educational pathways. They award academic degrees that enable students to transfer to four-year colleges and are attuned to the needs of the local workforce. "**Computer Science Education in Community Colleges**" presents the context and perspective of community college computer science education.

Teaching about Accessibility in Computer Science Education

Richard E. Ladner, University of Washington, Seattle, WA, USA
Stephanie Ludi, University of North Texas, Denton, TX, USA
Robert J. Domanski, Hunter College (CUNY), New York, NY, USA

Accessibility, in the context of computer science, is about making computing products accessible to people with disabilities. This means designing hardware and software products that can be used effectively by people who have difficulty reading a computer screen, hearing computer prompts, or controlling the keyboard, mouse, or touchscreen. Thus, accessibility topics should be woven into any course about human-facing applications or websites, such as app and web design/development, software engineering, and human-computer interaction. In addition, accessibility is about creating technical solutions to accessibility problems that people with disabilities encounter in everyday living. These technical solutions may include the use of artificial intelligence, computer vision, natural language processing, or other CS topics. Thus, accessibility topics can be included in technical courses, particularly those that incorporate projects where students attempt to solve accessibility problems using techniques taught in the course. There are practical, intellectual, and social reasons to integrate accessibility into computer science curriculum. From a practical standpoint, employers increasingly include accessibility knowledge in job descriptions because they want their products and services to be accessible to more customers and for legal compliance. From an intellectual standpoint, technical solutions to many accessibility problems often require creativity and a multi-disciplinary approach that includes understanding user needs integrated with technical knowledge. From a social standpoint, accessibility is an important topic in addressing inclusivity and an attractive topic for those students who enter the field to do social good, leading to a broader mix of students in terms of gender, race, ethnicity, and ability.

Helpful Resources:

[1] Catherine Caldwell-Harris, & Chloe Jordan. 2014. Systemizing and special interests: Characterizing the continuum from neurotypical to autism spectrum disorder. *Learning and Individual Differences*. Volume 29, Issue 2014, 98-105. https://doi.org/10.1016/j.lindif.2013.10.005.

[2] CAIR: RIT Center for Accessibility and Inclusion Research; http://cair.rit.edu/projects.html**.** accessed September 7, 2022.

[3] Robert F. Cohen, Alexander V. Fairley, David Gerry, and Gustavo R. Lima. 2005. Accessibility in introductory computer science. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education (SIGCSE '05)*. Association for Computing Machinery, New York, NY, USA, 17–21. https://doi.org/10.1145/1047344.1047367.

[4] Robert F. Dugan Jr (2011) A survey of computer science capstone course literature, *Computer Science Education*, 21:3, 201-267, https://www.tandfonline.com/doi/abs/10.1080/08993408.2011.606118. Accessed March 2024.

[5] Kristen Shinohara, Saba Kawas, Amy J. Ko, and Richard E. Ladner. 2018. Who Teaches Accessibility? A Survey of U.S. Computing Faculty. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18*). Association for Computing Machinery, New York, NY, USA, 197–202. https://doi.org/10.1145/3159450.3159484.

[6] Stephanie Ludi, Matt Huenerfauth, Vicki Hanson, Nidhi Rajendra Palan, and Paula Conn. 2018. Teaching Inclusive Thinking to Undergraduate Students in Computing Programs. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 717–722. DOI: https://doi.org/10.1145/3159450.3159512.

[7] Alannah Oleson, Amy J. Ko, Richard Ladner (Eds.) (2023). Teaching Accessible Computing. Self-Published. https://bookish.press/tac. Accessed November 28, 2023.

[8] PEAT; https://www.peatworks.org/. Accessed January 5, 2023.

[9] Teach Access website, http://www.teachaccess.org. Accessed September 10, 2022.

[10] Kendra Walther and Richard E. Ladner. 2021. Broadening participation by teaching accessibility. *Communications of the ACM* 64, 10 (October 2021), 19–21. https://doi.org/10.1145/3481356.

[11] WCAG https://www.w3.org/WAI/standards-guidelines/wcag/ Accessed November 6, 2022.

[12] Jacob O. Wobbrock, Shaun K. Kane, Krzysztof Z. Gajos, Susumu Harada, and Jon Froehlich. 2011. Ability-Based Design: Concept, Principles and Examples. *ACM Transactions on Accessible Computing* 3, 3, Article 9 (April 2011), 27 pages. https://dl.acm.org/doi/10.1145/1952383.1952384.

Computing for Social Good in Education

Heidi J. C. Ellis, Western New England University, Springfield, MA, USA
Gregory W. Hislop, Drexel University, Philadelphia, PA, USA
Mikey Goldweber, Denison University, Granville, OH, USA
Sam Rebelsky, Grinnell College, Grinnell, IA, USA
Janice L Pearce, Berea College, Berea, KY, USA
Patti Ordonez, University of Maryland Baltimore County, Baltimore, MD, USA
Marcelo Pias, Universidade Federal do Rio Grande, Rio Grande, Brazil
Neil Gordon, University of Hull, Hull, UK

Computing for Social Good (CSG) encompasses the potential of computing to have a positive impact on individuals, communities, and society, both locally and globally. Incorporating CSG into education (CSG-Ed) is especially relevant as computing has more and more impact across all areas of society and daily life. Educators can address CSG-Ed through a variety of means [2]. A simple way to start is by modifying a single assignment within a single course by updating the domain of the assignment to be one with social impact. The use of this domain can then be expanded across several assignments within the same course or across several courses. The domain could also provide the opportunity for collaborations across related departments. Indeed, some countries, such as England, integrate CSG throughout the curriculum starting before higher education studies [6].

Another way that educators may support CSG-Ed is the adoption or creation of a classroom project that solves a social problem either for a campus organization or from the larger community [1]. This approach allows students to see the impact of their work within their own community. On a larger scale, participation in established projects with national or global scope allows students to understand the breadth of influence that computing can have. Such efforts align well with institutions that have a service-learning requirement [4]. In addition, hackathons, code-days, clubs, and other extracurricular activities allow students to understand the social impact of computing outside of the classroom.

There are several challenges to integrating computing for social good into higher education [3]. One challenge is that instructors may not be inclined to incorporate new topics fearing that it could disrupt the curriculum or require course rework. Instructor time is a second barrier where it may take time to understand CSG domains and create new assignments. The interdisciplinary nature of many CSG topics may also require collaborating with other departments, disciplines, or community partners resulting in additional course preparation time. CSG-Ed assignments may result in the discussion of social issues within the classroom that could require instructors to prepare to discuss these issues with students. In addition, there appears to be a shortage of coverage of CSG in textbooks.

While barriers to CSG-Ed adoption exist, this focus of computing education provides multiple opportunities. CSG-Ed provides the possibility for students to connect with real-world problems to understand the complexity of computing while also apprehending the social impact of computing [5]. Students can be motivated by engaging in solving local problems that directly impact themselves or their community. They can also gain a better understanding of global citizenship and responsibility by participating in social projects that have a global scale.

There are several areas of future investigation including creation of a repository of CSG-Ed materials, addressing project-related challenges, exploring open source in CSG-Ed, and approaches for creating and growing an inclusive community to support CSG-Ed.

References

[1] Grant Braught, Steven Huss-Lederman, Stoney Jackson, Wes Turner, and Karl R. Wurst. 2023. Engagement Models in Education-Oriented H/FOSS Projects. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V*. (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 409–415. https://doi.org/10.1145/3545945.3569835

[2] Michael Goldweber, John Barr, Tony Clear, Renzo Davoli, Samuel Mann, Elizabeth Patitsas, and Scott Portnoff. 2012. A framework for enhancing the social good in computing education: a values approach. In *Proceedings of the final reports on Innovation and technology in computer science education 2012 working groups (ITiCSE-WGR '12).* Association for Computing Machinery, New York, NY, USA, 16–38. https://doi.org/10.1145/2426636.2426639.

[3] Mikey Goldweber, Lisa Kaczmarczyk, and Richard Blumenthal. Computing for the social good in education. *ACM Inroads*, 10, 4 (Dec 2019): 24–29.

[4] Janice L. Pearce. Requiring outreach from a CS0-level robotics course. *J. Comput. Sci. Coll.* 26, 5 (May 2011), 205–212.

[5] Lori Postner, Darci Burdge, Stoney Jackson, Heidi Ellis, George W. Hislop, and Sean Goggins. Using humanitarian free and open source software (HFOSS) to introduce computing for the social good. *SIGCAS Comput. Soc.* 45, 2 (June 2015), 35. https://doi.org/10.1145/2809957.2809967.

[6] Computer Science GCSE Subject Content. https://assets.publishing.service.gov.uk/media/5a7e3cb440f0b62305b81b02/Computer_Science_GCSE_-_subject_content_-_final.pdf. Accessed 26 Nov. 2023.

Multiple Approaches for Teaching Responsible Computing

Stacy A. Doore, Colby College, Waterville, ME, USA
Atri Rudra, University at Buffalo, Buffalo, NY, USA
Michelle Trim, University of Massachusetts, Amherst, MA, USA
Joycelyn Streator, Prairie View A&M University, Prairie View, TX, USA & the Mozilla Foundation
Richard Blumenthal, Regis University, Colorado, CO, USA
Bobby Schnabel, University of Colorado, Boulder, CO, USA

Teaching applied ethics in computer science (and computing in general) has shifted from a perspective of teaching about professional codes of conduct and an emphasis on risk management towards a broader understanding of the impacts of computing on humanity and the environment. This shift has produced a diversity of approaches for integrating responsible computing instruction into core computer science knowledge areas and for an expansion of dedicated courses focused on computing ethics. There is an increased recognition that students need intentional and consistent opportunities throughout their computer science education to develop the critical thinking, analytical reasoning, and cultural competency skills to understand their roles and professional duties in the responsible design, implementation, and management of complex computing systems. Therefore, computing education programs are re-evaluating the ways in which students learn to identify and assess the impact of computing on individuals, communities, and societies along with other critical professional skills such as effective communication, workplace conduct, and regulatory responsibilities. One of the primary shifts in the new approach comes from interdisciplinary collaborations, combining computing, social sciences and humanities researchers who work together to help students identify potential biases, blind spots, impacts, and harms in applications or systems and examine underlying assumptions and competing values driving design decisions.

There are examples of how topics within the CS2023 Society, Ethics, and the Profession (SEP) knowledge area can be implemented and assessed with numerous links to current module repositories [1-6], lessons [7-11], and resources [12-21] to embed responsible computing teaching across the CS curriculum. There are specific recommendations and resources that will help address current barriers for moving forward with the integration of responsible computing practices in the classroom [22]. These include ways of being open and confident in honoring all students' prior knowledge and lived experiences in sometimes difficult conversations [23-24] and overcoming student apathy or resistance to embedding responsible computing content [25-26]. These strategies require a willingness to work within an interdisciplinary community to incorporate social science and humanities domain expertise within these classroom interactions [27-29]. There are also recommendations on how to bring undergraduate students into curriculum planning as many of the earliest responsible computing teaching models were co-developed with undergraduate CS students [30-32]. Finally, there are recommendations about distinguishing between often conflated concepts, associated with responsible computing such as social justice [33-35], trust and safety [36-38], and value-sensitive design and co-design [39-40]. The understanding and use of these principles and practices in the classroom communicate the importance of stakeholder groups and impacted community inclusion from the beginning of the technology development lifecycle and affirms the agentive role of that community in development decisions. We hope this contribution will assist instructors as they develop their learning

objectives, activities, and assessments while adding to the growing body of knowledge on the best practices for weaving responsible computing principles and content throughout the evolving ACM/IEEE/AAAI computing curricula.

References

[1] ACM Engage CSEdu Ethics Repository. https://www.engage-csedu.org/ethics-and-computing/repository. Accessed Feb 28, 2024.

[2] Embedded EthiCS @ Harvard University - Modules Repository. https://embeddedethics.seas.harvard.edu/. Accessed Feb 28, 2024.

[3] Computing Ethics Narratives and Modules Repository at Bowdoin College and Colby College. https://computingnarratives.com. Accessed Feb 28, 2024.

[4] Embedded Ethics in Computer Science at Stanford University - Modules Repository. https://embeddedethics.stanford.edu/. Accessed Feb 28, 2024.

[5] Embedded EthiCS Modules Repository at University of Toronto. https://www.cs.toronto.edu/embedded-ethics/modules/index.html. Accessed Feb 28, 2024.

[6] Responsible Computer Science Repository at Bemidji State University. https://www.bemidjistate.edu/academics/departments/mathematics-computer-science/rcs/. Accessed Feb 28, 2024.

[7] Integrating Social Responsibility into Core CS. https://evanpeck.github.io/projects/responsibleCS. Accessed Feb 28, 2024.

[8] Internet Rules Lab University of Colorado Boulder. https://www.internetruleslab.com/responsible-computing. Accessed Feb 28, 2024.

[9] Responsible Computer Science at Washington University at St. Louis. https://www.cse.wustl.edu/~cytron/RCS/. Accessed Feb 28, 2024.

[10] University of Miami Dade Responsible Computing Role Playing Lesson. https://news.mdc.edu/role-playing-scenario-developed-at-entec/. Accessed Feb 28, 2024.

[11] Georgia Tech Responsible Computing Science. https://sites.gatech.edu/responsiblecomputerscience/. Accessed Feb 28, 2024.

[12] Mozilla Responsible Computing Playbook. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/. Accessed Feb 28, 2024.

[13] Teaching Responsible Computing at University of Buffalo. https://foundation.mozilla.org/en/responsible-computing-challenge-

playbook/https://c4sg.cse.buffalo.edu/projects/Teaching%20Responsible%20Computing.html. Accessed Feb 28, 2024.

[14] Human Context and Ethics at UC Berkeley. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/ https://data.berkeley.edu/academics/human-contexts-and-ethics. Accessed Feb 28, 2024.

[15] Social & Ethical Responsibilities of Computing at MIT. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/ https://computing.mit.edu/cross-cutting/social-and-ethical-responsibilities-of-computing. Accessed Feb 28, 2024.

[16] Socially Responsible Computing @ Brown University. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/ http://ethics.cs.brown.edu/. Accessed Feb 28, 2024.

[17] Embedded Ethics Program at Georgetown University. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/ https://ethicslab.georgetown.edu/embedded-ethics. Accessed Feb 28, 2024.

[18] Ethical Computer Science at Allegheny College. Accessed Feb 28, 2024. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/ https://csethics.allegheny.edu/.

[19] Ethics 4 EU - Educational Resources. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/ https://ascnet.ie/ethics4eu-website/welcome-to-the-bricks/. Accessed Feb 28, 2024.

[20] Human Context and Ethics at UC Berkeley. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/ https://data.berkeley.edu/academics/human-contexts-and-ethics. Accessed Feb 28, 2024.

[21] Markkula Center for Applied Ethics at Santa Clara University- Technology Ethics. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/ https://www.scu.edu/ethics/focus-areas/technology-ethics/. Accessed Feb 28, 2024.

[22] Colleen Greer & Marty J. Wolf. 2020. Overcoming barriers to including ethics and social responsibility in computing courses. In *Societal Challenges in the Smart Society*, 131-144. Universidad de La Rioja.

[23] Rodrigo Ferreira & Moshe Y. Vardi. 2021. Deep tech ethics: An approach to teaching social justice in computer science. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 1041-1047.

[24] Michelle Trim & Paige Gulley. 2023. Imagining, generating, and creating: Communication as feminist pedagogical method for teaching computing ethics. In *Proceedings of the 41st ACM International Conference on Design of Communication*, 206-209.

[25] Nina Zuber, Jan Gogoll, Severin Kacianka, Alexander Pretschner, & Julian Nida-Rümelin. Empowered and embedded: ethics and agile processes. *Humanities and Social Sciences Communications*. 9, 1 (2022): 1-13.

[26] Shamika Klassen & Casey Fiesler. Run Wild a Little with Your Imagination: Ethical Speculation in Computing Education with Black Mirror." *In Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. 1, (2022): 836-842.

[27] Barbara J. Grosz, David Gray Grant, Kate Vredenburgh, Jeff Behrends, Lily Hu, Alison Simmons, & Jim Waldo. 2019. Embedded EthiCS: Integrating ethics across CS education. *Communications of the ACM*, 62, 8 (2019): 54–61.

[28] National Academies of Sciences, Engineering, and Medicine. 2022. Fostering Responsible Computing Research: Foundations and Practices.

[29] Trystan S. Goetze. 2023. Integrating ethics into computer science education: Multi-, inter-, and transdisciplinary approaches. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education*, 1, (2023): 645-651.

[30] Beleicia B. Bullock, Fernando L. Nascimento, & Stacy A. Doore. Computing ethics narratives: Teaching computing ethics and the impact of predictive algorithms. *In Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 2021: 1020-1026.

[31] Nora McDonald, Adegboyega Akinsiku, Jonathan Hunter-Cevera, Maria Sanchez, Kerrie Kephart, Mark Berczynski, and Helena M. Mentis. Responsible computing: A longitudinal study of a peer-led ethics learning framework. *ACM Transactions on Computing Education (TOCE)* 22,4 (2022): 1-21.

[32] Alexandra Gillespie. 2023. Designing an ethical tech developer. *Communications of the ACM,* 66, 3 (2023): 38-40.

[33] Ruha Benjamin. Race after technology, *Social Theory Re-Wired*, 405-415. Routledge (2023).

[34] Rachel Charlotte Smith, Heike Winschiers-Theophilus, Daria Loi, Rogério Abreu de Paula, Asnath Paula Kambunga, Marly Muudeni Samuel, & Tariq Zaman. Decolonizing design practices: towards pluriversality. In *Extended abstracts of the 2021 CHI conference on human factors in computing systems*. 1-5.

[35] Sasha Costanza-Chock. *Design Justice: Community-led Practices to Build the Worlds We Need.* The MIT Press, 2020.

[36] Ben Shneiderman. Bridging the gap between ethics and practice: guidelines for reliable, safe, and trustworthy human-centered AI systems. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 10,4 (2020): 1-31.

[37] Cansu Canca. 2020. Operationalizing AI ethics principles. *Communications of the ACM*, 63, 12 (2020): 18-21.

[38] International Organization for Standardization. Information Technology–Artificial Intelligence – Management Systems (ISO/IEC Standard 42001-2023). https://www.iso.org/standard/81230.html Accessed Feb 28, 2024.

[39] John M. Carroll. Encountering others: Reciprocal openings in participatory design and user-centered design. Human-computer Interaction 11,3 (1996), 285-290. https://foundation.mozilla.org/en/responsible-computing-challenge-playbook/. Accessed Feb 28, 2024.

[40] David G. Hendry, Batya Friedman, & Stephanie Ballard. Value sensitive design as a formative framework. *Ethics and Information Technology* 23, 23 (2021): 1-6.

Making ethics at home in Global CS Education: Provoking stories from the Souths

Marisol Wong-Villacres, Escuela Superior Politecnica del Litoral, Guayaquil, Ecuador
Cat Kutay, Charles Darwin University, Northern Territory, Australia
Shaimaa Lazem, City of Scientific Research and Technological applications, Alexandria, Egypt
Nova Ahmed, North South University, Dhaka, Bangladesh
Cristina Abad, Escuela Superior Politecnica del Litoral, Guayaquil, Ecuador
Cesar Collazos, Universidad del Cauca, Popayan, Colombia
Shady Elbassuoni, American University of Beirut, Beirut, Lebanon
Farzana Islam, North South University, Dhaka, Bangladesh
Deepa Singh, University of Delhi, New Delhi, India,
Tasmiah Tahsin Mayeesha, North South University, Dhaka, Bangladesh
Martin Mabeifam Ujakpa, Ghana Communication Technology University, Accra, Ghana
Tariq Zaman, University of Technology, Sibu, Malaysia
Nicola J. Bidwell, Charles Darwin University and University of Melbourne, Australia, International University of Management, Namibia

We, a group of thirteen educators in computing programs and researchers in universities, retell the stories of 46 university educators and practitioners in Latin America, South-Asia, Africa, the Middle East, and Australian First Nations who participated in surveys and interviews with us [1]. We use the plural of Global Souths to indicate the multiple and overlapping geographic and conceptual spaces that are negatively impacted by contemporary capitalist globalization and the US–European norms and values exported in computing products, processes, and education. The stories illustrate frictions between local practices, values, and impacts of technologies and the static, anticipatory approaches to ethics that computer science (CS) curricula often promote through codes of ethics. The stories show diverse perspectives on privacy and institutional approaches to confidentiality; compliance with regulations to attain various goals and difficulties when regulations are absent or ambiguously relate to practices; discrimination based on their gender or technical ability and minoritized positions; and, finally, that relational, rather than transactional, approaches to ethics may better suit local ethical challenges.

CS codes of ethics can assist educators by listing factors for consideration and mitigating situations when regulations, laws or policies are not fully developed. Yet the gap between codes of ethics and local realities can also cause harm. Further prevalent codes of ethics are instruments of power that enable actors in the Global North to determine what legitimate CS practice comprises and the position of the Global Souths relative to this. Thus, we advocate for ethical guidance that speaks to and comes “from within” people’s messy realities in the Global Souths not only because connecting ethics to students’ and educators’ values, knowledge, and experiences is vital for learning but also to assert greater recognition and respect for localized ethical judgements.

Making ethics at home in global CS education is about fostering students’ ethical sensibilities and orienting them to engage reflexively with different values and positionalities within and beyond their own contexts. Ethical considerations are always updating as new technologies, new socio-technical situations and new sensitivities emerge and, thus, we suggest that educators use storytelling about

ongoing, real-world events to engage students with “ethos building.” [2] In the epilogue that extends this piece [3], we share two stories that arose when researching and presenting this article to show how ethics is embedded in every action and how as educators we must continuously refine our sensitivity to the varied ways our lives are implicated in technical and socio-technical systems, from local to global scales, and develop confidence to discuss their implications with our students.

Our modest study significantly extends existing research [1] on how CS educators account for the diverse ways ethical dilemmas and approaches to ethics are situated in cultural, philosophical, and governance systems, religions, and languages [1].

References

[1] Janet Hughes, Ethan Plaut, Feng Wang, Elizabeth von Briesen, Cheryl Brown, Gerry Cross, Viraj Kumar, and Paul Myers. Global and local agendas of computing ethics education. In *Proceedings of the Conference on Innovation and Technology in Computer Science* Education, 2020; (ACM, New York, NY, 2020) 239-245.

[2] Christopher Frauenberger and Peter Purgathofer. 2019. Responsible thinking educating future technologists. In *Proceedings of CHI Conference on Human Factors in Computing Systems (CHI’19).*

[3] Wong-Villacres, M., Kutay, C., Lazem, S., Ahmed, N., Abad, C., Collazos, C., ... & Bidwell, N. J. Making ethics at home in Global CS Education: Provoking stories from the Souths. *ACM Journal on Computing and Sustainable Societies*. (2023).

CS + X: Approaches, Challenges, and Opportunities in Developing Interdisciplinary Computing Curricula

Valerie Barr, Bard College, Annandale-on-Hudson, NY, USA
Carla E. Brodley, Northeastern University, Boston, MA, USA
Elsa L. Gunter, UIUC, Urbana-Champaign, IL, USA
Mark Guzdial, University of Michigan, Ann Arbor, MI, USA
Ran Libeskind-Hadas, Claremont McKenna College, Claremont, CA, USA
Bill Manaris, College of Charleston, Charleston, SC, USA

Interdisciplinary computing curricula and majors (often called CS+X) interweave foundational computing concepts with those of specific disciplines in the natural sciences, social sciences, humanities, and the arts. Well-designed CS+X programs have substantially increased diversity and inclusion in computing. They address a rapidly growing need for a computationally sophisticated workforce across many domains that are critical to society. Virtually every discipline has significant challenges and opportunities that require computational methods. Increasingly, many researchers and practitioners in those fields are using computational methods, yet undergraduates in those fields often get little or no computational training deeper than using existing software tools.

Interdisciplinary computing can be implemented in individual courses (e.g., a course that combines both the art and computing concepts for visualization); as a major+minor; or as its own major, where students take some courses from computing, a similar number from another discipline, and one or more integrative courses.

Interdisciplinary courses and majors have several additional benefits. There is ample evidence that such innovative programs significantly broaden participation in computing. For example, interdisciplinary programs can substantially improve gender diversity and, generally, engage diverse populations of students who are unlikely to pursue a within-discipline computing degree [1,2,3,4]. The gender diversity likely depends in part on the X in CS+X. For example, at some institutions with CS+X programs where X is related to the arts, the CS+X major has approximately equal numbers of women and men, which is more than twice the national statistic for CS programs (22% women).

A second benefit of interdisciplinary computing majors is the ability to reach a larger set of students – because of enrollment pressures and course caps in computer science departments, non-majors are often unable to access the computing courses that they seek. CS+X majors can help computing departments (and universities) better manage enrollments. A CS+X major typically will require fewer computing classes than a within-discipline CS major, reducing enrollment pressure on higher-level electives that are often harder to staff.

References

[1] William Bares, Bill Manaris, and Renée McCauley. Gender equity in computer science through Computing in the Arts – A six-year longitudinal study. *Computer Science Education Journal* 28, 3 (September 2018), 191–210. https://doi.org/10.1080/08993408.2018.1519322.

[2] William H. Bares, Bill Manaris, Renée McCauley, and Christine Moore. Achieving gender balance through creative expression. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (Minneapolis, MN, USA) (SIGCSE 2019)*. Association for Computing Machinery, New York, NY, USA, 293-299. https://doi.org/10.1145/3287324.3287435

[3] Carla E. Brodley, Benjamin J. Hescott, Jessica Biron, Ali Ressing, Melissa Peiken, Sarah Maravetz, and Alan Mislove. Broadening participation in computing via ubiquitous combined majors (CS+X). In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (Providence, RI, USA) (SIGCSE 2022).* Association for Computing Machinery, New York, NY, USA, 544–550. https://doi.org/10.1145/3478431.3499352

[4] Zachary Dodds, Malia Morgan, Lindsay Popowski, Henry Coxe, Caroline Coxe, Kewei Zhou, Eliot Bush, and Ran Libeskind-Hadas. A Biology-based CS1: Results and reflections, ten years in. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE 2021)*. Association for Computing Machinery, New York, NY, USA, 796-801.

The Role of Formal Methods in Computer Science Education

Maurice H. ter Beek, CNR–ISTI, Pisa, Italy
Manfred Broy, Technische Universität München, München, Germany
Brijesh Dongol, University of Surrey, Guilford, UK
Emil Sekerinski, McMaster University, Hamilton, Canada

Formal Methods (FM) are available in various forms, spanning from lightweight static analysis to interactive theorem proving. These methods provide a systematic demonstration to students of the application of formal foundations in Computer Science within engineering tasks. The core skill of abstraction, fundamental to computer science, is effectively addressed through FM [1]. Even students specializing in 'Formal Methods Thinking'—the application of ideas from FM in informal, lightweight, practical, and accessible ways—experience notable improvement in their programming skills [2]. Exposure to these ideas also positions students well for further study on why techniques work, how they can be automated, and the development of new approaches.

FM can contribute significantly to teaching programming to novices, complementing informal reasoning and testing methods. They elucidate algorithmic problem-solving, design patterns, model-driven engineering, software architecture, software product lines, requirements engineering, and security, thereby supporting various fields within computer science [3]. Formalisms provide a concise and precise means of expressing underlying design principles, equipping programmers with tools to address related problems.

In industry, FM find widespread application, from eliciting requirements and early design to deployment, configuration, and runtime monitoring [4]. A recent survey [5] involving 130 FM experts, including three Turing Award winners, all four FME Fellowship Award winners, and 16 CAV Award winners, indicates that the most suitable place for FM in a teaching curriculum is in bachelor courses at the university level, as reported by 79.2% of respondents. Furthermore, 71.5% of respondents identify the lack of proper training in FM among engineers as the key limiting factor for a broader adoption of FM by the industry.

The survey highlights the uneven nature of FM education across universities, with many experts advocating for the standardization of university curricula. A recent white paper [6] supports this view, proposing the inclusion of a compulsory FM course in Computer Science and Software Engineering curricula. This recommendation is based on the observation that there is a shortage of Computer Science graduates qualified to apply Formal Methods in industry.

The challenge is twofold: (1) the lack of definitive educational sources that support FM-based courses in Computer Science; and (2) the training of academic staff to teach FM. Help is, however, becoming available (https://fmeurope.org/teaching/), and the future is bright, as more and more educators contribute to the effort of creating and sharing teaching resources.

References

[1] Manfred Broy, Achim D. Brucker, Alessandro Fantechi, Mario Gleirscher, Klaus Havelund, Cliff Jones, Markus Kuppe, Alexandra Mendes, André Platzer, Jan Oliver Ringert, and Allison Sullivan. Does Every Computer Scientist Need to Know Formal Methods? Submitted to *Form. Asp. Comput.* (2023).

[2] Brijesh Dongol, Catherine Dubois, Stefan Hallerstede, Eric Hehner, Daniel Jackson, Carroll Morgan, Peter Müller, Leila Ribeiro, Alexandra Silva, Graeme Smith, and Erik de Vink. On Formal Methods Thinking in Computer Science Education. Submitted to *Form. Asp. Comput.* (2023).

[3] Emil Sekerinski, Marsha Chechik, João F. Ferreira, John Hatcliff, Michael Hicks, and Kevin Lano.. Should We Teach Formal Methods or Algorithmic Problem Solving, Design Patterns, Model-Driven Engineering, Software Architecture, Software Product Lines, Requirements Engineering, and Security? In preparation 2023.

[4] Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Y. Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. 2023. Formal Methods in Industry. Submitted to *Form. Asp. Comput.* (2023).

[5] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. 2020. The 2020 Expert Survey on Formal Methods. In Proceedings of the 25th International Conference on Formal Methods for Industrial Critical Systems (FMICS'20) (LNCS, Vol. 12327), Maurice H. ter Beek and Dejan Ničković (Eds.). Springer, Germany, 3–69. https://doi.org/10.1007/978-3-030-58298-2_1

[6] Antonio Cerone, Markus Roggenbach, James Davenport, Casey Denner, Marie Farrell, Magne Haveraaen, Faron Moller, Philipp Körner, Sebastian Krings, Peter Csaba Ölveczky, Bernd-Holger Schlingloff, Nikolay Shilov, and Rustam Zhumagambetov. 2021. Rooting Formal Methods Within Higher Education Curricula for Computer Science and Software Engineering – A White Paper. In Revised Selected Papers of the 1st International Workshop on Formal Methods – Fun for Everybody (FMFun'19) (CCIS, Vol. 1301), Antonio Cerone and Markus Roggenbach (Eds.). Springer, Germany, 1–26. https://doi.org/10.1007/978-3-030-71374-4_1

Quantum Computing Education: A Curricular Perspective

Dan-Adrian German, Indiana University, Bloomington, IN, USA
Marcelo Pias, Federal University of Rio Grande, Rio Grande, RS, Brazil
Qiao Xiang, Xiamen University, Xiamen, Fujian, China

At the end of 2023 we are still in the NISQ era [4,5]. The term (Noisy Intermediate-Scale Quantum) was introduced by John Preskill at Q2B in December 2017. Atom Computing first reached 1,000 qubits in 2013 [9], soon thereafter followed by IBM [10]. The milestone marks just how far the industry has come: only 6 years ago, typically, under 10 qubits were available for developers on the IBM Quantum Experience. Long-time quantum pioneer D-Wave remains an outlier in that it has a 5,000-qubit system (Advantage) but it is an analog, not a gate-based system; it is an open question whether gate-based approaches are necessary to get the full power of fault-tolerant quantum computing and D-Wave has recently started developing gate-based technology. On the other hand, adiabatic quantum computing (AQC) and quantum annealing (QA) remain legitimate (and promising) avenues of research in quantum computation. Also, this year, a Harvard-led team developed the first-ever quantum circuit with logical quantum bits [1]. Arrays of “noisy” physical Rydberg qubits were used to create quantum circuits with 48 error-correcting logical qubits, the largest number to date, a crucial step towards realizing fault-tolerant quantum computing. Meanwhile, PsiQuantum continues to pursue unabated the 1,000,000 (physical) qubits mark [7,8]. The competition between the various qubit implementation modalities intensified: superconducting qubits, trapped atoms/ions, spin qubits (Intel has a 12-qubit chip) and photonics are currently in the lead. Debates [6] now abound about the potential (or impending) demise of the NISQ era. The industry remains engaged in a sustained effort of both short-term (upskilling and reskilling workers, and HS teachers) and long-term workforce development. This past summer, researchers at Quantinuum and Oxford University [2,11] established the foundations and methodology for an ongoing educational experiment to investigate the question: ‘From what age can students learn quantum theory if taught using a diagrammatic approach?' The math-free framework in [3] was used to teach the pictorial method to UK schoolchildren, who then beat the average exam scores of Oxford University’s postgraduate physics students. The experiment involved 54 schoolchildren, aged 15-17, randomly selected from around 1,000 applicants, from 36 UK schools (mostly state schools). Teenagers spent two hours a week in online classes and after eight weeks were given a test using questions taken from past Oxford postgraduate quantum exams: more than 80% of the pupils passed and around half earned a distinction. Interest in incorporating quantum architecture topics in the traditional CS curriculum remains high for the next 10-year horizon. A growing consensus is that the CS undergraduate must have a proper appreciation for the quantum mechanical nature of our world. The main prerequisite to such a knowledge unit remains a certain intellectual versatility, manifested in the willingness to be exposed to information from more than one domain/discipline. In quantum computing, labs will be quintessential and will rely on (1) computer-assisted mathematics (e.g., Wolfram Alpha, NumPy, Qiskit, Matplotlib, etc.) as well as CAD/CAM and advanced software emulation (Qiskit Metal), (2) access to actual quantum computers via various cloud platforms (Amazon Braket, IBM Q, Xanadu Borealis, etc.) and (3) occasionally access to a physics lab, fab or foundry. A genuinely interdisciplinary program can only be built if faculty have wide general support towards such a goal. Three curricular approaches have emerged: one is entirely without math but leading into math and lasts about eight weeks. The second is a full semester, 14-week long, and entirely based on linear algebra. The last one

is two semesters long and includes weekly, messy but critical, quantum hardware labs supporting a quantum engineering degree. Incorporating material about all qubit modalities in the curriculum will ensure the material will remain relevant over a reasonably long period of time, if it includes such topics as the design and implementation of qubits (e.g., via Qiskit Metal) and error mitigation and (classical) control.

References

[1] D. Bluvstein, S. J. Evered, A. A. Geim, et al. Logical quantum processor based on reconfigurable atom arrays. In *Nature*. https://doi.org/10.1038/s41586-023-06927-3 (6 Dec. 2023).

[2] Bob Coecke. https://medium.com/quantinuum/everyone-can-learn-quantum-now-even-at-a-cutting-edge-level-and-we-have-the-test-scores-to-prove-49e7fdc5c509 (21 Dec. 2023). Accessed March 2024.

[3] Bob Coecke and Stefano Gogioso. *Quantum in Pictures: A New Way to Understand the Quantum World. Cambridge Quantum*, 1st edition (3 Feb. 2023).

[4] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum* 2, 79 (2018). https://quantum-journal.org/papers/q-2018-08-06-79/. Accessed March 2024; Preprint: https://arxiv.org/abs/1801.00862. Accessed March 2024.

[5] John Preskill. Quantum technology in the short term and long term: the search for applications. https://www.youtube.com/watch?v=TSzpz8N7Xw4 (Q2B 2018 Keynote Address). Accessed March 2024.

[6] John Preskill. Crossing the Quantum Chasm: From NISQ to Fault Tolerance. Q2B 2023 (6 Dec 2023) http://theory.caltech.edu/~preskill/talks/Preskill-Q2B-2023 (slides, video not yet available).

[7] Terry Rudolph. What is the logical gate speed of a photonic quantum computer? (June 21, 2023, via John Preskill's Twitter account and the Quantum Frontiers blog at the Institute for Quantum Information and Matter at Caltech) https://quantumfrontiers.com/2023/06/21/what-is-the-logical-gate-speed-of-a-photonic-quantum-computer/. Accessed March 2024.

[8] John Russell. PsiQuantum's Path to 1 Million Qubits.(21 April 2022, in hpcwire.com) https://www.hpcwire.com/2022/04/21/psiquantums-path-to-1-million-qubits-by-the-middle-of-the-decade/.

[9] John Russell. Atom Computing Wins the Race to 1000 Qubits. (24 Oct. 2023 in hpcwire.com) https://www.hpcwire.com/2023/10/24/atom-computing-wins-the-race-to-1000-qubits/ Accessed March 2024.

[10] The Quantum Mechanic. IBM and UC Berkeley Usher in New Era of Quantum Computing with 1,121 Qubit Machine. Hello IBM Condor. (4 Dec. 2023) https://quantumzeitgeist.com/ibm-and-uc-berkeley-usher-in-new-era-of-quantum-computing-with-1121-qubit-machine-hello-ibm-condor/. Accessed March 2024.

[11] Aleks Kissinger. Research unveils new picture-based approach to teaching physics. (20 Dec. 2023) https://www.cs.ox.ac.uk/news/2280-full.html. Accessed March 2024.

Generative AI in Introductory Programming

Brett A. Becker, University College Dublin, Dublin, Ireland
Michelle Craig, University of Toronto, Toronto, Canada
Paul Denny, The University of Auckland, Auckland, New Zealand
Hieke Keuning, Utrecht University, Utrecht, The Netherlands
Natalie Kiesler, DIPF Leibniz Institute for Research and Information in Education, Frankfurt, Germany
Juho Leinonen, Aalto University, Aalto, Finland
Andrew Luxton-Reilly, The University of Auckland, Auckland, New Zealand
Lauri Malmi, Aalto University, Aalto, Finland
James Prather, Abilene Christian University, Abilene, TX, USA
Keith Quille, TU Dublin, Dublin, Ireland

Generative AI tools based on Large Language Models (LLMs) such as OpenAI's ChatGPT, and IDEs powered by them such as GitHub Copilot, have demonstrated impressive performance in myriad types of programming tasks including impressive performance on CS1 and CS2 problems. They can often produce syntactically and logically correct code from natural language prompts that rival the performance of high-performing introductory programming students—an ability that has already been shown to extend beyond introductory programming [2]. However, their impact in the classroom goes beyond producing code. For example, they could help level the playing field between students with and without prior experience. Generative AI tools have been shown to be proficient in not only explaining programming error messages but in repairing broken code [6], and pair programming might evolve from two students working together into “me and my AI.” On the other hand they could have negative effects. Students could become over-reliant on them, and they may open up new divides due to different backgrounds, experience levels and access issues [9]. Generative AI has been successful in generating novel exercises and examples including providing correct solutions and functioning test cases [11]. Instructional materials are already being produced including a textbook that uses Generative AI from the first day of CS1 [8] that has already been used [4]. Given their ability to provide code explanations [7] they have the potential to assess student work, provide feedback, and to act as always-available virtual teaching assistants, easing the burden not only on the educator but on their human assistants and the broader educational systems where learning takes place [9]. Generative AI could even affect student intakes given its prominence in the media and the effect that such forces can have on who chooses to—and who chooses not to—study computing.

Given that Generative AI has the potential to reshape introductory programming, it is possible that it will impact the entire computing curriculum, affecting what is taught, when it is taught, how it is taught, and to whom it is taught. However, the dust is far from settled on these matters with some educators embracing Generative AI and others very fearful that the challenges could outweigh the opportunities [5]. The computing education community needs to understand more about how students interact with Generative AI [10] and provide tooling and strategies to effectively achieve that interaction [3]. Indeed, during the transformation from pre- to post-Generative AI introductory programming, several issues need to be mitigated including but certainly not limited to those of ethics, bias, academic integrity, and broadening participation in computing [1]. Further study is warranted to explore the long-term effects of

Generative AI on pedagogy, curriculum, student demographics, and the broader educational ecosystem.

References

[1] Brett A. Becker, Paul Denny, James Finnie-Ansley, et al. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V*. 1, 2023. (Toronto, ON, Canada) (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 500–506. https://doi.org/10.1145/3545945.3569759.

[2] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing Education in the Era of Generative AI. Commun. ACM 67, 2 (Feb. 2024). https://doi.org/10.1145/3624720. Preprint available: https://arxiv.org/abs/2306.02608. Accessed March 2024).

[3] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett Becker, and Brent Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In Proceedings of the 55th SIGCSE Technical Symposium on Computer Science Education (Portland, OR USA) (SIGCSE '24). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3626252.3630909. Preprint available: https://arxiv.org/abs/2311.05943. Accessed March 2024).

[4] Katie E. Ismael, Ioana Patringenaru, and Kimberley Clementi. In This Era of AI, Will Everyone Be a Programmer? UC San Diego Today (Dec 2023). https://today.ucsd.edu/story/in-this-era-of-ai-will-everyone-be-a-programmer. Accessed March 2024.

[5] Sam Lau and Philip Guo. 2023. From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools Such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) (ICER '23). Association for Computing Machinery, New York, NY, USA, 106–121. https://doi.org/10.1145/3568813.3600138.

[6] Juho Leinonen, Arto Hellas, Sami Sarsa, et al. Using Large Language Models to Enhance Programming Error Messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto, ON, Canada) (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 563–569. https://doi.org/10.1145/3545945.3569770.

[7] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. Generating Diverse Code Explanations using the GPT-3 Large Language Model. In Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2 (ICER '22), Vol. 2. Association for Computing Machinery, New York, NY, USA, 37–39. https://doi.org/10.1145/3501709.3544280.

[8] Leo Porter and Daniel Zingaro. 2023. Learn AI-Assisted Python Programming with GitHub Copilot and ChatGPT. Manning, Shelter Island, NY, USA. https://www.manning.com/books/learn-ai-assisted-python-programming. Accessed March 2024.

[9] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. The Robots Are Here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education (Turku, Finland) (ITiCSE-WGR '23)*. Association for Computing Machinery, New York, NY, USA, 108–159. https://doi.org/10.1145/3623762.3633499.

[10] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That It Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. ACM Trans. Comput.-Hum. Interact. 31, 1, Article 4 (Nov 2023), 31 pages. https://doi.org/10.1145/3617367.

[11] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1 (Lugano and Virtual Event, Switzerland) (ICER '22)*. Association for Computing Machinery, New York, NY, USA, 27–43. https://doi.org/10.1145/3501385.3543957.

The 2022 Undergraduate Database Course in Computer Science: What to Teach?

Mikey Goldweber, Denison University, Granville, OH, USA
Min Wei, Microsoft, Seatle, WA, USA
Sherif Aly, The American University in Cairo, Cairo, Egypt
Rajendra K. Raj, Rochester Institute of Technology, Rochester, NY, USA
Mohamed Mokbel, University of Minnesota, St. Paul, MN, USA

One issue with the study of databases, though maybe it should be labeled data management, or maybe even more precisely, the study of persistent data, is that the number of possible topics far exceeds the bandwidth of a single undergraduate CS course. Yes, there are several institutions with two course sequences. However, most undergraduate curricula, based on CS2013 [2] recommendations or ABET [1] criteria, have at most one database course, or just an elective. So, the question arises as to what to include and what to exclude.

Contributing to this phenomenon are the emergence of new topics (e.g., NoSQL, distributed and cloud-based databases) and the current renewed (and hopefully continuing) emphasis on both security and privacy, as well as societal and ethical issues associated with persistent data.

Another complicating factor is the institutional context. Every institution's curricular viewpoint sits somewhere on the spectrum between computer science as a pure science and computer science as a profession. Institutions are now preparing graduates for careers as Data Engineers, Data Infrastructure Engineers, and Data Scientists, in addition to Computer Scientists.

There are four primary perspectives with which to approach databases.
1. Database designers/modelers: those who model the data from an enterprise and organize it according to the principles of a given data model.
2. Database users: (SQL?) query writers.
3. Database administrators: those involved with tuning database performance through the building of index structures and the setting of various parameters.
4. Database engine developers: those who write the code for database engines.

Four different viewpoints for what an undergraduate CS course in Databases/Data Management should cover are described in [3].

References

[1] ABET (2022). ABET Computing Accreditation Commission: Criteria for Accrediting Computing Programs. https://www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-computing-programs-2022-2023/. Accessed March 2024.

[2] ACM (2013). Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science, Association for Computing Machinery and IEEE Computer Society. https://doi.org/10.1145/2534860.

[3] Mikey Goldweber, Min Wei, Sherif Aly, Rajendra K. Raj, and Mohamed Mokbel. The 2022 undergraduate database course in computer science: what to teach? *ACM Inroads* 13, 3 (September 2022), 16–21. https://doi.org/10.1145/3549545.

Computer Science Curriculum Guidelines: A New Liberal Arts Perspective

Jakob Barnard; University of Jamestown; Jamestown, MD, USA
Valerie Barr; Bard College; Annandale-on-Hudson, NY, USA
Grant Braught; Dickinson College; Carlisle, PA, USA
Janet Davis; Whitman College; Walla Walla, WA, USA
Amanda Holland-Minkley; Washington & Jefferson College; Washington, PA, USA
David Reed; Creighton University; Omaha, NE, USA
Karl Schmitt; Trinity Christian College; Palos Heights, IL, USA
Andrea Tartaro; Furman University; Greenville, SC, USA
James Teresco; Siena College; Loudonville, NY, USA

ACM/IEEE curriculum guidelines for computer science, such as CS2023, provide well-researched and detailed guidance regarding the content and skills that make up an undergraduate computer science (CS) program. Liberal arts CS programs often struggle to apply these guidelines within their institutional and departmental contexts [6]. Historically, this has been addressed through the development of model CS curricula tailored for the liberal arts context [1,2,3,4,7]. We take a different position: that no single model curriculum can apply across the wide range of liberal arts institutions. Instead, we argue that liberal arts CS educators need best practices for using guidelines such as CS2023 to inform curriculum design. These practices must acknowledge the opportunities and priorities of a liberal arts philosophy as well as institutional and program missions, priorities, and identities [5].

The history, context, and data about liberal arts CS curriculum design support the position that the liberal arts computing community is best supported by a process for working with curricular guidelines rather than a curriculum model or set of exemplars [5]. Previous work with ACM/IEEE curriculum guidelines over the decades has trended towards acknowledging that liberal arts CS curricula may take a variety of forms and away from presenting a unified “liberal arts” model [6]. A review of liberal arts CS programs demonstrates how institutional context, including institutional mission and structural factors, shape their curricula [5]. Survey data indicates that liberal arts programs have distinct identities or missions, and this directly impacts curriculum and course design decisions. Programs prioritize flexible pathways through their programs coupled with careful limits on required courses and lengths of prerequisite chains [6]. This can drive innovative course design where content from Knowledge Areas is blended rather than compartmentalized into distinct courses [7,8]. The CS curriculum is viewed as part of the larger institutional curriculum and the audience for CS courses is broader than just students in the major, at both the introductory level and beyond.

To support the unique needs of CS liberal arts programs, we propose a process that guides programs to work with CS2023 through the lens of institutional and program missions and identities, goals, priorities, and situational factors. The Process Workbook we have developed comprises six major steps:

1. articulate institutional and program mission and identity;
2. develop curricular design principles driven by program mission and identity, structural factors, and attention to diversity, equity, and inclusion;

3. identify aspirational learning outcomes in response to design principles and mission and identity;
4. engage with CS2023 to select curriculum and course content based on design principles to achieve learning outcomes and support mission and identity;
5. evaluate the current program, with attention to current strengths, unmet goals, and opportunities for improvement;
6. design, implement, and assess changes to the curriculum.

An initial version of the Process Workbook, based on our research and feedback from workshops [9, e.g., 10,11] and pilot usage within individual departments, is available as a supplement to this article [12]. The authors will continue this iterative design process and release additional updates as we gather more feedback. Future work includes development of a repository of examples of how programs have made use of the Workbook to review and redesign their curricula in the light of CS2023.

References

[1] Kim B. Bruce, Robert D. Cupper, and Robert L. Scot Drysdale. A History of the Liberal Arts Computer Science Consortium and Its Model Curricula. *ACM Trans. Comput. Educ.* 10,1, Article 3 (March 2010), 12 pages. https://doi.org/10.1145/1731041.1731044.

[2] Liberal Arts Computer Science Consortium. A 2007 Model Curriculum for a Liberal Arts Degree in Computer Science. *J. Educ. Resour. Comput.* 7,2 (June 2007), 2-es. https://doi.org/10.1145/1240200.1240202.

[3] Henry M. Walker and G. Michael Schneider. A Revised Model Curriculum for a Liberal Arts Degree in Computer Science. *Commun. ACM* 39,12 (Dec. 1996), 85–95. https://doi.org/10.1145/240483.240502.

[4] Norman E. Gibbs and Allen B. Tucker. A Model Curriculum for a Liberal Arts Degree in Computer Science. *Commun. ACM* 29, 3 (March 1986), 202-210. https://doi.org/10.1145/5666.5667.

[5] Amanda Holland-Minkley, Jakob Barnard, Valerie Barr, Grant Braught, Janet Davis, David Reed, Karl Schmitt, Andrea Tartaro, and James D. Teresco. Computer Science Curriculum Guidelines: A New Liberal Arts Perspective. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 617-623. https://doi.org/10.1145/3545945.3569793.

[6] James D. Teresco, Andrea Tartaro, Amanda Holland-Minkley, Grant Braught, Jakob Barnard, and Douglas Baldwin. CS Curricular Innovations with a Liberal Arts Philosophy. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (Providence, RI, USA) (SIGCSE 2022).* Association for Computing Machinery, New York, NY, USA, 537-543. https://doi.org/10.1145/3478431.3499329.

[7] Henry M. Walker and Samuel A. Rebelsky. Using CS2013 for a Department's Curriculum Review: A Case Study. *J. Comput. Sci. Coll.* 29,5 (May 2014), 138-144.

[8] David Reed. Spiraling CS2013 Knowledge Units across a Small CS Curriculum. *J. Comput. Sci. Coll.* 32,5 (May 2017), 125-131.

[9] Amanda Holland-Minkley, Andrea Tartaro, and Jakob Barnard. Innovations and Opportunities in Liberal Arts Computing Education, https://computing-in-the-liberal-arts.github.io/SIGCSE2023-Affiliated-Event/. URL. SIGCSE 2023 Affiliated Event by the SIGCSE Committee on Computing Education in Liberal Arts Colleges.

[10] Jakob Barnard, Grant Braught, Janet Davis, Amanda Holland-Minkley, David Reed, Karl Schmitt, Andrea Tartaro, and James Teresco. Developing Identity-Focused Program-Level Learning Outcomes for Liberal Arts Computing Programs. *J. Comput. Sci. Coll.* 39,4 (October 2023), 97-98.

[11] Jakob Barnard, Grant Braught, Janet Davis, Amanda Holland-Minkley, David Reed, Karl Schmitt, Andrea Tartaro, and James Teresco. Reflective Curriculum Review for Liberal Arts Computing Programs. *J. Comput. Sci. Coll.* 38, 3 (November 2022), 178–179.

[12] SIGCSE Committee on Computing Education in Liberal Arts Colleges. 2023. CS2023 Activity: The Curricular Practices Workbook. https://computing-in-the-liberal-arts.github.io/CS2023/. Accessed March 2024.

Computer Science Education in Community Colleges

Elizabeth Hawthorne, Rider University, Lawrenceville, NJ, USA
Lori Postner, Nassau Community College, Garden City, NY, USA
Christian Servin, El Paso Community College, El Paso, TX, USA
Cara Tang, Portland Community College, Portland, OR, USA
Cindy Tucker, Bluegrass Community and Technical College, Lexington, KY, USA

Community and Technical Colleges serve as two-year educational institutions, providing diverse academic degrees like associate's degrees in academic and applied sciences, certificates of completion, and remedial degrees. These colleges play a crucial role in fostering collaboration between students, workers, and institutions through educational and workforce initiatives. Over the past 50+ years, Community Colleges have served as a hub for various educational initiatives and partnerships involving K-12 schools, four-year colleges, and workforce/industry collaborations.

These colleges offer specialized programs that help students focus on specific educational pathways. Among the programs available, computing-related courses are prominent, including Computer Science degrees, particularly the Associate in Arts (AA) and Sciences (AS) degrees, known as academic transfer degrees. These transfer degrees are designed to align with the ACM/IEEE curricular guidelines, primarily focusing on creating two-year programs that facilitate smooth transferability to four-year colleges.

Furthermore, the computing programs offered by Community Colleges are influenced by the specific needs and aspirations of the regional workforce and industry. Advisory boards and committees play a significant role in shaping these programs by providing recommendations based on the demands of the job market. While the ACM Committee for Computing in Community Colleges (CCECC) and similar entities help address inquiries related to these transfer degrees, there is a desire to capture the challenges, requirements, and recommendations from the Community College perspective in developing general curricular guidelines.

This work presents the context and perspective of the community college education. It emphasizes the importance of understanding the unique challenges faced by Community Colleges and their specific needs while formulating curricular guidelines. Additionally, the work envisions considerations for the next decade regarding curricular development and administrative efforts, considering the evolving educational landscape and industry demands. By doing so, the vision is to enhance the effectiveness and relevance of computing programs offered by Community Colleges and foster better alignment with the needs of students and the job market.

References

[1] "ABET Accredits 54 Additional Programs in 2021, Including First Associate Cybersecurity programs." https://www.abet.org/abet-accredits-54-new-programs-in-2021-including-first-associate-cybersecurity-programs/. Accessed Feb 29, 2024.

[2] William F. Atchison, Samuel D. Conte, John W. Hamblen, Thomas E. Hull, Thomas A. Keenan, William B. Kehl, Edward J. McCluskey, Silvio O. Navarro, Werner C. Rheinboldt, Earl J. Schweppe, William Viavant, and David M. Young. "Curriculum 68: Recommendations for Academic Programs in Computer Science: A Report of the ACM Curriculum Committee on Computer Science." *Communications of the ACM* 11,3 (1968), 151-197. https://doi.org/10.1145/362929.362976.

[3] Jill Denner, Paul Tymann, and Huihui Wang. "Community College Pathways." In *Proceedings of the 2023 CISE EWF PI Meeting.* Georgia Tech Conference Center.

[4] Dennis Foley, Leslie Milan, and Karen Hamrick. 2020. "The Increasing Role of Community Colleges among Bachelor's Degree Recipients: Findings from the 2019 National Survey of College Graduates." Technical Report NSF 21-309. National Center for Science and Engineering Statistics (NCSES), Alexandria, VA. https://ncses.nsf.gov/pubs/nsf21309/. Accessed March 2024.

[5] ACM Committee for Computing Education in Community Colleges (CCECC). 2017. "ACM Computer Science Curricular Guidance for Associate-Degree Transfer Programs with Infused Cybersecurity." 2017. Association for Computing Machinery, New York, NY, USA.

[6] The Community College Presidents Initiative in STEM. 2023. "Community college presidents initiative – STEM – achieving excellence in workforce education." https://www.ccpi-stem.org/. Accessed March 2024.

[7] Robin G Isserles. "The Costs of Completion: Student Success in Community College." JHU Press, 2021

[8] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. "Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science." ACM, New York, NY, USA, 2013.

[9] A. Kahlon, D. Boisvert, L.A. Lyon, M. Williamson, and C. Calhoun. "The Authentic Inclusion and Role of Community Colleges in National Efforts to Broaden Participation in Computing." In Proceedings of the 2018 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '18). ACM, New York. https://doi.org/10.1145/3159450.3159627.

[10] Amruth N. Kumar and Rajendra K. Raj. "Computer Science Curricula 2023 (CS2023): Community Engagement by the ACM/IEEE-CS/AAAI Joint Task Force." In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2 (Toronto, ON, Canada) (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1212-1213. https://doi.org/10.1145/3545947.3569591.

[11] Joyce Currie Little, Richard H. Austing, Harice Seeds, John Maniotes, and Gerald L. Engel.. "Curriculum recommendations and guidelines for the community and junior college career program in computer programming: a working paper of the ACM committee on curriculum in computer sciences by the subcommittee on community and junior college curriculum." ACM SIGCSE Bulletin - Special issue on computer science curricula 9, 2 (1977), 1-16. https://doi.org/10.1145/988948.988951.

[12] B. Morrison and A. Settle. "Celebrating SIGCSE's 50th Anniversary!" SIGCSE Bulletin 50,1 (2018), 2-3.

[13] American Association of Community Colleges. 2022. "The Economic Value of America's Community Colleges." https://www.aacc.nche.edu/2022/11/29/the-economic-value-of-americas-community-colleges-report/. Accessed March 2024.

[14] Christian Servín. "Fuzzy Information Processing Computing Curricula: A Perspective from the First Two-Years in Computing Education." In *Explainable AI and Other Applications of Fuzzy Techniques: Proceedings of the 2021 Annual Conference of the North American Fuzzy Information Processing Society, NAFIPS 2021*. Springer, 453-460.

[15] Christian Servin, Elizabeth K. Hawthorne, Lori Postner, Cara Tang, and Cindy Tucker. "Community Colleges Perspectives: From Challenges to Considerations in Curricula Development (SIGCSE 2023)." Association for Computing Machinery, New York, NY, USA, 1244. https://doi.org/10.1145/3545947.3573335.

[16] Christian Servin, Elizabeth K. Hawthorne, Lori Postner, Cara Tang, and Cindy S. Tucker. "Mathematical Considerations in Two-Year Computing Degrees: The Evolution of Math in Curricular Guidelines." In *The 24th Annual Conference on Information Technology Education (SIGITE '23) (Marietta, GA, USA).* ACM. https://doi.org/10.1145/3585059.3611441.

[17] Cara Tang. 2017. "Community College Corner Community colleges in the United States and around the world." *ACM Inroads* 8,1 (2017), 21-23.

[18] Cara Tang. 2018. "Community Colleges and SIGCSE: A Legacy Fueling the Future." *ACM Inroads* 9,4 (2018), 49-52. https://doi.org/10.1145/3230699.

[19] Cara Tang, Elizabeth K Hawthorne, Cindy S Tucker, Ernesto Cuadros-Vargas, Diana Cukierman, and Ming Zhang. "Global Perspectives on the Role of Two-Year/Technical/Junior Colleges in Computing Education." In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 204-205.

[20] "Celebrating 40++ years of service to computing education communities." [n.d.]. ACM CCECC. https://ccecc.acm.org/correlations/all. Accessed March 2024.

[21] Stuart Zweben, Jodi L.Tims, Cindy Tucker, and Yan Timanovsky. "ACM-NDC Study 2021–2022: Tenth Annual Study of Non-Doctoral-Granting Departments in Computing." *ACM Inroads* 13,3 (2022), 38-54. https://doi.org/10.1145/3544304.

[22] Stuart Zweben and Cindy Tucker. "How Well Did We Keep Students in Computing Programs, Pre-COVID and COVID?" *ACM Inroads* 13,4 (2022), 32-52. https://doi.org/10.1145/3571094.

Generative AI and the Curriculum

Introduction

Generative AI technologies have begun to revolutionize the ways students learn. While it is too early to confidently predict how they will change computer science education, it is instructive to consider some of the ramifications already apparent. In this section, a few specific implications of generative AI are explored by competency and knowledge areas. The overall observations follow.

- Educators can use generative AI to create course syllabi, design projects and assignments, and automate grading.
- Students can use generative AI for individualized tutoring, undergraduate research, enhanced productivity, etc. The skill set expected of students is changing, but the foundational knowledge they need to know is not.
- Tasks that are intellectually challenging, yet mechanical in nature will be automated. So, students must learn to work at higher levels of abstraction.
- Issues of society, ethics, and the profession are taking on much more importance and significance and must not only be actively addressed in the curriculum, but also periodically revisited and revised.

Implications by Competency Area - Software

Algorithmic Foundations (AL)

- The immediate impact of generative AI on Algorithmic Foundations is expected to be less than what is found in other knowledge areas since AL topics are primarily focused on students understanding the foundations of algorithmic computation. As an example, while generative AI may be prompted to produce an algorithmic solution utilizing less memory or improved runtime performance, students are expected to be able to evaluate the results of generative AI from an AL perspective of time-space tradeoff and complexity analysis.

Foundations of Programming Languages (FPL)

- The next paradigm in programming might be conversational programming that involves providing appropriate prompts to help generative AI produce and test desired programs. Generative AI is also likely to be able to translate code from one programming language to another, and from one paradigm to another. Program correctness proofs and validating program behavior against a specification may also be areas in which generative AI makes advances because of the intellectually challenging, yet mechanical, nature of the process.

Software Development Fundamentals (SDF)

- While generative AI can write simple programs, the responsibility to verify the correctness of the programs still falls on the user. So, even for verification purposes only, computer science students still need to learn how to write programs. In order to provide appropriate prompts to generative AI, students need to be able to design and plan larger programs. How students design and write

programs will evolve as generative AI technologies improve. How computer science educators must adapt to teaching programming with the help of generative AI is currently an open question. It seems likely that reading programs will see a sharpened focus and the idea of learning to program by writing, and reliance on learning syntax, will change towards learning to program by prompting, comprehending, verifying, editing, modifying, adapting, and testing code. It is also likely that other aspects of the curriculum such as problem decomposition will see increased focus.

Software Engineering (SE)

- Generative AI is expected to have substantial impact on several aspects of the software process, including (but not limited to) development of new code, comprehension of complex logging and debugging artifacts, static analysis, and code reviewing. The most understandable and visible change is likely to be in the development of (rote) new code—assistance technologies like GitHub's Copilot and other advanced auto-complete mechanisms can meaningfully improve development time, to a point. Effective use of such tools requires a deeper investment in design and code comprehension, while potentially decreasing the need for hands-on programming time. Similar advances in static analysis and code review are anticipated to have a meaningful impact on code quality and clarity, ideally also reducing the impact of implicit bias by increasing consistency and quality of comments and diagnostics.

Implications by Competency Area - Systems

Architecture and Organization (AR)

- The evolution needed in computer architecture to better support generative AI technologies may itself become a critical area of study in the future. Another might be the impact of using generative AI for hardware design.

Data Management (DM)

- Generative AI systems appear to excel at accomplishing intellectually challenging, yet mechanical tasks. It seems evident that given the expression of a query in pseudocode (or relational algebra/calculus), the translation of a description of the desired query result into SQL will become a routine generative AI task. In one sense, generative AI allows database querying at a higher (natural language, pseudocode) level. However, regardless of the level at which students query a database, they will still need the skills to validate the results returned by an (AI-generated) SQL query.

Networking and Communication (NC)

- Generative AI technologies may be increasingly used in computer science education of networking and communications. For example, it can generate code for new or existing networking protocols, suggestions for improvement, generate network traffic data for experimentation purposes, and generate scenarios for which a network architecture needs to be designed. Furthermore, it can verify whether networking requirements are ambiguous or complete. It can be used to automatically

generate configuration scripts, perform security assessments of existing networked configurations such as identifying vulnerabilities, and suggesting countermeasures, and perform reliability assessments and capacity planning.

Operating Systems (OS)

- Generative AI will be helpful with deployment scripts and system optimization. In the short term, generative AI will not invalidate the need for students to understand the layer between applications and the architecture. With that said, AI eventually will provide insight into the expected performance or security implications of software that is developer- or AI-generated. Students will need to continue to innovate and reason at higher-levels of abstraction.

Systems Fundamentals (SF)

- Providing system support for generative AI applications is expected to turn into a robust area of research and education.

Security (SEC)

- Generative AI can be used both by an adversary to launch more sophisticated attacks or develop more dangerous malware and by the defender to protect against such attacks. Other issues in the impact of generative AI on security include interaction between an AI attacker and defender, security risks of data and reverse engineering AI models, AI-based data aggregation / phishing, and security of AI systems themselves. Students need to develop a nuanced understanding of the power and drawbacks of generative AI applied to security concerns.

Implications by Competency Area - Applications

Artificial Intelligence (AI)

- The success of generative AI is already attracting more interest in Artificial Intelligence as a field, both increasing the number of students interested in studying it and increasing its applicability to other subfields of computer science. Generative models provide opportunities to incorporate multimodal analysis into larger pipelines and provide alternatives to techniques such as classical planning (e.g., using an LLM to suggest next steps to solving a problem). However, they currently lack foundational guarantees of correctness, grounded perception, and explanation that are critical to many applications. Emerging techniques for combining subsymbolic and symbolic methods using generative models look promising for resolving these issues.

Graphics and Interactive Techniques (GIT)

- Intellectual property and ethical issues regarding the use of generative AI for 2D and 3D graphics, images, video, and animation are important and evolving. These apply both to the media content and/or creative work used to train AI as well as the media created using generative AI.

Human-Computer Interaction (HCI)

- Generative AI will have a broad impact on the ways that people regard, and therefore interact, with computers and their output. Programs that employ generative AI can offer a simple interface that responds to user prompts and rapidly produce acceptable output. Such products, however, require user education to set reasonable expectations and acknowledge generative AI's ability to err and to hallucinate. Users must also know how to engineer a prompt to achieve the desired effect. Ideally, a product that employs generative AI should have safeguards that evolve with their use and be adaptive for accuracy and sensitivity to the user's cultural norms. Finally, users should become aware of issues surrounding intellectual property, transparency, and the extensive human labor required to train these models. The speed at which this technology continues to evolve requires careful attention to both current and future potential pitfalls, failure modes, and human consequences.

Specialized Platform Development (SPD)

- From an educational standpoint, generative AI tools offer valuable assistance to both students and developers. They enable the rapid creation of prototypes and the generation of code templates for common mobile app elements. Moreover, these tools produce code snippets for routine tasks, reducing the necessity for manual coding. This functionality is particularly advantageous for novices and those in the initial stages of learning mobile development. Generative AI Integrated Development Environment (IDE) plugins also deliver real-time feedback and suggestions to students as they write code.

  Furthermore, generative AI enriches teaching by crafting interactive simulations and virtual environments tailored to mobile app development. These environments allow students to experiment and hone their skills effectively. In the educational context, generative AI is crucial in raising awareness of security and ethical considerations within mobile app development. Additionally, AI serves as an invaluable resource for developers by providing continuous updates on the latest trends and technologies in specialized platform development, ensuring developers maintain up-to-date skills and knowledge in this rapidly evolving field.

Implications for Crosscutting Areas

Mathematical and Statistical Foundations (MSF)

- Mathematics is learned best by "doing" with sustained practice in working through problems. Although generative AI tools may help in explaining, they are just as likely to present complete solutions to students, robbing them of the learning gains that occur in struggling through problems. In contrast, generative AI tools could be adapted to help instructors automate grading of proofs that would alleviate what is now a laborious task. In the longer term, a balanced approach could offer educator support (automated grading) and student support through customized tutoring to address skill gaps either before starting a new course or while taking a course as long as the during-course usage is calibrated carefully. Perhaps the best option at the moment is to encourage rigorous on-going experimentation with generative AI so that the academic computing community can identify best practices for the future.

Society, Ethics, and the Profession (SEP)

- The education computer science students receive needs to incorporate all dimensions and roles of the computing profession: technical, philosophical, and ethical. Generative AI raises unique risks for SEP, for instance deepfakes and misinformation will become more pervasive and harder to identify. Being technically capable of ethically questionable action and being technically able to identify and prevent such actions puts greater burden on SEP lessons to make students aware of these responsibilities and ensure that as they enter the workforce as graduates, they work to keep society on a just path.

Implications for the Curriculum

- It is clear that students must learn how to correctly use generative AI technologies for coursework. The boundary between using generative AI as a resource and using it to plagiarize must be clarified. The limitations (e.g., hallucinations) of the technology must be discussed, as should the inherent biases that may have been baked into the technology by virtue of the data used to train them.
- Many new technologies have already redefined the boundary between tasks that can be mechanized and those that will need human participation. Generative AI is no different. Correctly identifying the boundary will be the challenge for computer science educators going forward.
- Generative AI may be used to facilitate undergraduate research – more innovative research by more students at all levels of technical ability. Generative AI may be used by students for various research-related tasks: to fill the gaps in their understanding of prior research, build systems with which to test hypotheses, help interpret the results, etc.
- Students may use generative AI to summarize assigned readings, help explain gaps in their understanding of course material, fill in gaps in the presentations of the classes they missed, and interactively quiz themselves to assist in their studying.

Acknowledgments

Organization

The following assisted the task force in its work:

- ACM Staff:
  - Yan Timanovsky, ACM Education & Professional Development Manager
  - Lisa Kline, ACM Education and Professional Development Assistant
  - John Otero, Site Selection
- Jens Palsberg, University of California, Los Angeles, CA, USA, Chair of SIG Chairs
- ACM Education Board
  - Elizabeth Hawthorne, Rider University, Lawrenceville, NJ, USA
  - Alison Derbenwick Miller, Consultant, formerly Oracle Inc.
  - Christine Stephenson, Google Inc. (retired)
- Bruce McMillin, IEEE Computer Society, Professional & Educational Activities Board – Curriculum and Accreditation Committee (CA), Chair
- IEEE Computer Society Staff:
  - Eric Berkowitz, Director of Membership, USA
  - Michelle Phon, Certification and Professional Education, USA

Reviewers

The following reviewed various knowledge area drafts:

- Ginger Alford, Southern Methodist University, Dallas, TX, USA
- Jeannie Albrecht, Williams College, Williamstown, MA, USA
- Mostafa Ammar, Georgia Institute of Technology, Atlanta, GA, USA
- Tom Anderson, University of Washington, Seattle, WA, USA
- Christopher Andrews, Middlebury College, Middlebury, VT, USA
- Elisa Baniassad, The University of British Columbia, Vancouver, BC, Canada
- Arvind Bansal, Kent State University, Kent, OH, USA
- Phillip Barry, University of Minnesota, Minneapolis, MN, USA
- Brett A. Becker, University College Dublin, Dublin, Ireland
- Judith Bishop, Stellenbosch University, Stellenbosch, South Africa
- Alan Blackwell, University of Cambridge, Cambridge, UK
- Olivier Bonaventure, Université Catholique de Louvain, Louvain-la-Neuve, Belgium
- Kim Bruce, Pomona College, Claremont, CA, USA
- John Carroll, Penn State, University Park, PA, USA
- Gennadiy Civil, Google Inc., New York, NY, USA
- Thomas Clemen, Hamburg University of Applied Sciences, Hamburg, Germany
- Jon Crowcroft, University of Cambridge, Cambridge, UK
- Melissa Dark, Dark Enterprises, Inc., Lafayette, IN, USA
- Arindam Das, Eastern Washington University, Cheney, WA, USA

- Karen C. Davis, Miami University, Oxford, OH, USA
- Henry Duwe, Iowa State University, Ames, IA, USA
- Roger D. Eastman, University of Maryland, College Park, MD, USA
- Yasmine Elglaly, Western Washington University, Bellingham WA, USA
- Trilce Estrada, University of New Mexico, Albuquerque, NM, USA
- David Flanagan, Text book Author
- Akshay Gadre, University of Washington, Seattle, WA, USA
- Ed Gehringer, North Carolina State University, Raleigh, NC, USA
- Sheikh Ghafoor, Tennessee Tech University, Cookville, TN, USA
- Tirthankar Ghosh, University of New Haven, West Haven, CT, USA
- Michael Goldwasser, Saint Louis University, St. Louis, MO, USA
- Martin Goodfellow, University of Strathclyde, Glasgow, UK
- Vikram Goyal, IIIT, Delhi, India
- Dan Grossman, University of Washington, Seattle, WA, USA
- Xinfei Guo, Shanghai Jiao Tong University, Shanghai, China
- Anshul Gupta, IBM Research, Yorktown Heights, NY, USA
- Sally Hamouda, Virginia Tech, Blacksburg, VA, USA
- Matthew Hertz, University at Buffalo, Buffalo, NY, USA
- Michael Hilton, Carnegie Mellon University, Pittsburgh, PA, USA
- Bijendra Nath Jain, IIIT, Delhi, India
- Kenneth Johnson, Auckland University of Technology, Auckland, New Zealand
- Krishna Kant, Temple University, Philadelphia, PA, USA
- Hakan Kantas, Halkbank, Istanbul, Turkiye
- Amey Karkare, Indian Institute of Technology, Kanpur, India
- Kamalakar Karlapalem, International Institute of Information Technology, Hyderabad, India
- Theodore Kim, Yale University, New Haven, CT, USA
- Michael S. Kirkpatrick, James Madison University, Harrisonburg, VA, USA
- Tobias Kohn, Vienna University of Technology, Vienna, Austria
- Eleandro Maschio Krynski, Universidade Tecnológica Federal do Paraná, Guarapuava, Paraná, Brazil
- Ludek Kucera, Charles University, Prague, Czechia
- Fernando Kuipers, Delft University of Technology, Delft, The Netherlands
- Matthew Fowles Kulukundis, Google, Inc., New York, NY, USA
- Zachary Kurmas, Grand Valley State University, Allendale, MI, USA
- Rosa Lanzilotti, Università di Bari, Bari, Italy
- Alexey Lastovetsky, University College Dublin, Dublin, Ireland
- Gary T. Leavens, University of Central Florida, Orlando, FL, USA
- Kent D. Lee, Luther College, Decorah, IA, USA
- Bonnie Mackellar, St. John’s University, Queens, NY, USA
- Mary Lou Maher, University of North Carolina, Charlotte, NC, USA
- Alessio Malizia, Università di Pisa, Pisa, Italy
- Sathiamoorthy Manoharan, University of Auckland, Auckland, New Zealand
- Maristella Matera, Politecnico di Milano, Milano, Italy
- Stephanos Matsumoto, Olin College of Engineering, Needham, MA, USA

- Paul McKenney, Facebook, Inc.
- Mia Minnes, University of California San Diego, San Diego, CA, USA
- Michael A. Murphy, Coastal Carolina University, Conway, SC, USA
- Raghava Mutharaju, IIIT, Delhi, India
- V. Lakshmi Narasimhan, Georgia Southern University, Statesboro, GA, USA
- Marion Neumann, Washington University in St Louis, St. Louis, MO, USA
- Cheng Soon Ong, Data61 | CSIRO and Australian National University, Canberra Australia
- Peter Pacheco, University of San Francisco, San Francisco, CA, USA
- Andrew Petersen, University of Toronto, Mississauga, Canada
- Cynthia A Phillips, Sandia National Lab, Albuquerque, NM, USA
- Benjamin C. Pierce, University of Pennsylvania, Philadelphia, PA, USA
- Sushil K. Prasad, University of Texas, San Antonio, TX, USA
- Rafael Prikladnicki, Pontificia Universidade Catolica do Rio Grande do Sul, Porto Alegre, Brazil
- Keith Quille, Technological University Dublin, Dublin, Ireland
- Catherine Ricardo, Iona University, New Rochelle, NY, USA
- Luigi De Russis, Politecnico di Torino, Torino, Italy
- Beatriz Sousa Santos, University of Aveiro, Aveiro, Portugal
- Michael Shindler, University of California, Irvine, CA, USA
- Ben Shneiderman, University of Maryland, College Park, MD, USA
- Anna Spagnolli, Università di Padova, Padova, Italy
- Davide Spano, Università di Cagliari, Cagliari, Italy
- Andreas Stathopoulos, William & Mary, Williamsburg, VA, USA
- Anthony Steed, University College London, London, UK
- Michael Stein, Metro State University, Saint Paul, MN, USA
- Alan Sussman, University of Maryland, College Park, MD, USA
- Andrea Tartaro, Furman University, Greenville, SC, USA
- Tim Teitelbaum, Cornell University, Ithaca, NY, USA
- Joseph Temple, Coastal Carolina University, Conway, SC, USA
- Ramachandran Vaidyanathan, Louisiana State University, Baton Rouge, LA, USA
- Salim Virji, Google Inc., New York, NY, USA
- Guiliana Vitiello, Università di Salerno, Salerno, Italy
- Philip Wadler, The University of Edinburgh, Edinburgh, UK
- Charles Weems, University of Massachusetts, Amherst, MA, USA
- Xiaofeng Wang, Free University of Bozen-Bolzano, Bolzano, Italy
- Miguel Young de la Sota, Google Inc., USA
- Massimo Zancanaro, Università di Trento, Trento, Italy
- Ming Zhang, Peking University, Beijing, China

Contributors

The following contributed to various aspects of the report or to the design of the instruments used to collect data for the report:

- Tom Crick, Swansea University, Swansea, UK
- Steven Gordon, Ohio State University, Columbus, OH, USA (retired)
- Amanda Holland-Minkley, Washington & Jefferson College, Washington, PA, USA
- Mihaela Sabin, University of New Hampshire, Manchester, NH, USA
- Mehran Sahami, Stanford University, Stanford, CA, USA, ACM Co-Chair, CS 2013
- Karl Schmitt, Trinity Christian College, Palos Heights, IL, USA
- Andrea Tartaro, Furman University, Greenville, SC, USA
- Osei Tweneboah, Ramapo College of New Jersey, Mahwah, NJ, USA
- Patrick Van Metre, MITRE Corporation, McLean, VA, USA

In addition, numerous educators provided comments and suggestions through feedback forms posted for individual knowledge areas as well as for Version Beta and Version Gamma of the curricular report. Two hundred and twelve US educators and 215 international educators filled out the initial survey of their use of CS2013. Eight hundred and sixty-five industry practitioners filled out the initial survey of the importance of various curricular components for success after graduation. One hundred and eighty-two educators volunteered and filled out 70 surveys of CS Core topics. One hundred and ten educators filled out a survey of the characteristics of computer science graduates and 65 educators filled out a survey of institutional challenges for computer science programs. Their participation and input greatly contributed to the quality of the report.

Partial support was provided by the National Science Foundation under grant DUE-2231333.

Association for
Computing Machinery

IEEE
COMPUTER
SOCIETY

Association for the
Advancement of
Artificial Intelligence