Analysis of Algorithm (AoA)

SCHOOL OF COMPUTER ENGINEERING & TECHNOLOGY

AoA Objectives & Outcomes

Course Objectives:

- Study the performance analysis of algorithms
- Select algorithmic strategies to solve given problem.
- Explore solution space to solve the problems.
- Provide the knowledge about complexity theory

Course Outcomes:

1. Analyze the algorithm complexity using asymptotic notations and describe the divide-and-conquer paradigm and recite algorithms that employ this paradigm.
2. Describe greedy and dynamic programming algorithmic strategies and analysis algorithms that employ this paradigm.
3. Illustrate the solution space using backtracking and branch and bound algorithmic techniques.
4. Describe the concept of complexity theory.

Course prerequisite

Data Structure II

C, C++, Java or other programming languages

Module I - Fundamentals of Algorithm

Proof techniques : Contradiction, Mathematical induction, Direct proofs, Proof by counter example, proof by contradiction
Analysis of algorithm: Efficiency-Analysis framework, asymptotic notations - big oh, theta, omega

Analysis of non-recursive and recursive algorithms : Solving recurrence equations using Master's theorem and substitution method

Brute force method : Introduction to brute force method and exhaustive search, brute force solution to 8 queens' problem

Proof Terminology

Theorem: statement that can be shown to be true
Proof: a valid argument that establishes the truth of a theorem
Axioms: statements we assume to be true
Lemma: a less important theorem that is helpful in the proof of other results
Corollary: theorem that can be established directly from a theorem that has been proved
Conjecture: statement that is being proposed to be a true statement

What is a mathematical proof?

- A mathematical proof of the statement S is a sequence of logically valid statements that connect axioms, definitions, and other already validated statements into a demonstration of the correctness of S. The rules of logic and axioms are agreed upon ahead of time. At a minimum, the axioms should be independent and consistent. The amount of detail presented should be appropriate for the intended audience.

Structure of a Mathematical Proof

- Begin with a set of initial assumptions called **hypotheses**.

- Apply logical reasoning to derive the final result (the **conclusion**) from the hypotheses.

- Assuming that all intermediary steps are sound logical reasoning, the conclusion follows from the hypotheses.

Direct Proof

- A direct proof is the simplest type of proof.

- Starting with an initial set of hypotheses, apply simple logical steps to prove the conclusion.

    - Directly proving that the result is true.

Two Quick Definitions

- An integer n is even if there is some integer k such that n = 2k.
- This means that 0 is even.
- An integer n is odd if there is some integer k such that n = 2k + 1.
- We'll assume the following for now:
    - Every integer is either even or odd.
    - No integer is both even and odd.

Implications

- An implication is a statement of the form **If P, then Q**.
- We write “If P, then Q” as P → Q.
- Read: “P implies Q.”

When P → Q, we call **P the antecedent** and **Q the consequent.**

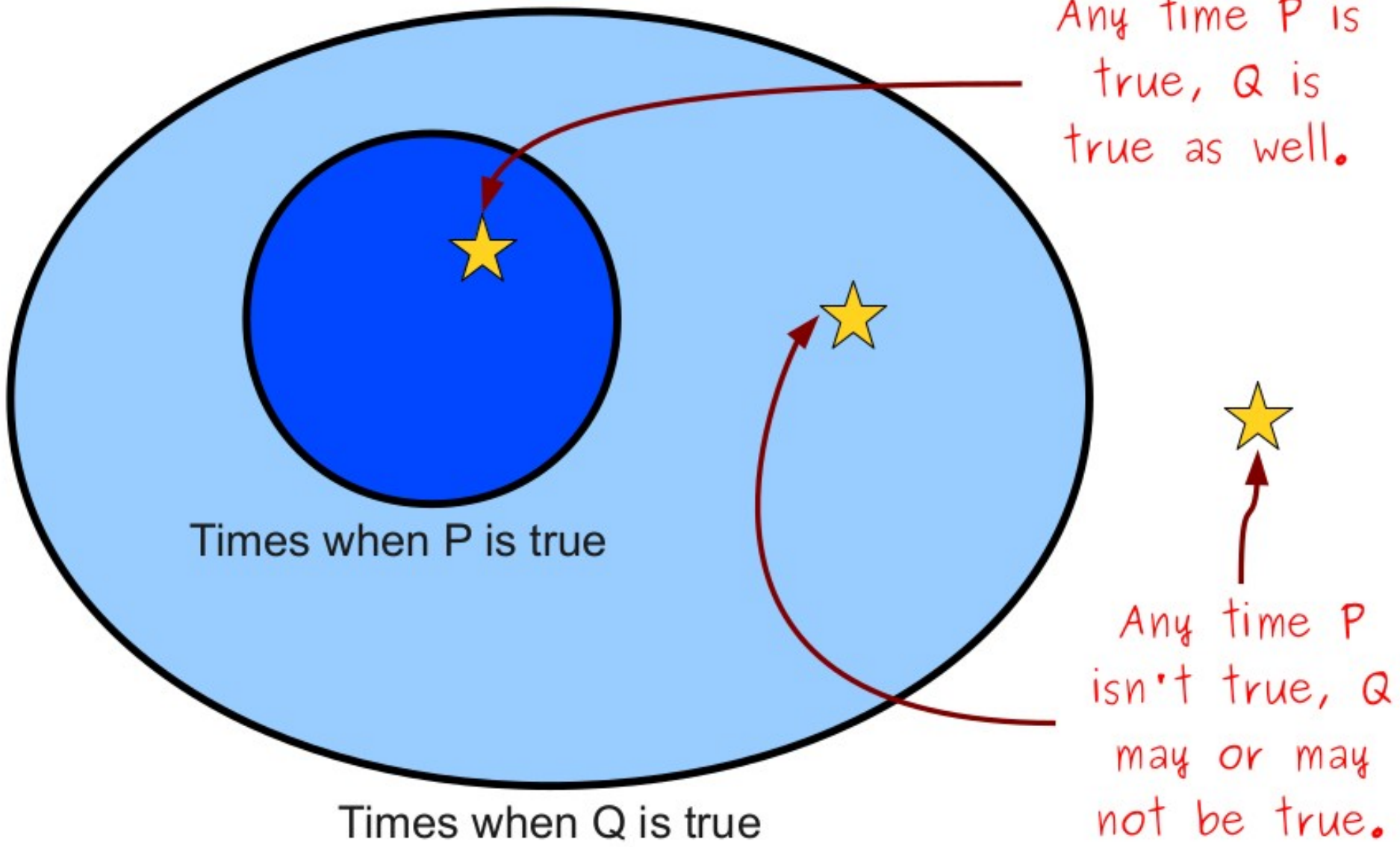
What does Implication Mean?

The statement $P \rightarrow Q$ means exactly the following:

Whenever P is true, Q must be true as well.

For example:

- n is even $\rightarrow$ n^2 is even.
- $(A \subseteq B \text{ and } B \subseteq C) \rightarrow A \subseteq C$


Any time P is true, Q is true as well.
Times when P is true
Any time P isn't true, Q may or may not be true.
Times when Q is true

When P Does Not Imply Q

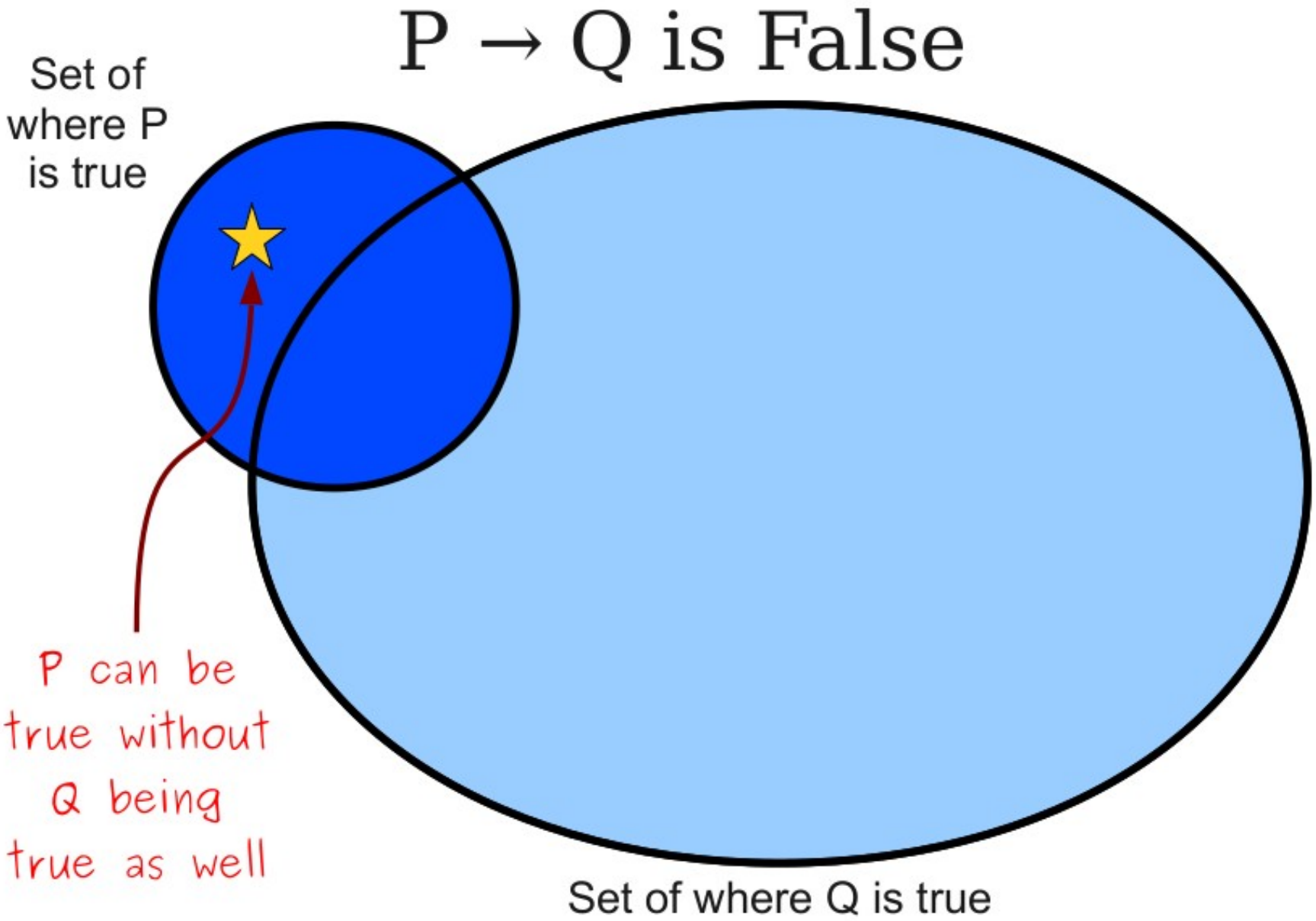
What would it mean for P → Q to be false?
Answer: There must be some way for P to be true and Q to be false.

P → Q means “any time P is true, Q is true.”
The only way to disprove this is to show that there is some way for P to be true and Q to be false**.**

To prove that P → Q is false, find an example where P is true and Q is false.

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

OK: THOMAS CORMEN



Direct Proof

To prove a statement of the form

"If *P*, then *Q*"

Assume that ***P*** is true, then show that ***Q*** must be true as well.

Direct Proof Example 1

Theorem: If n is even, then n^2 is even.
Proof: Let n be an even integer.
Since n is even, there is some integer k such that $n = 2k$.
This means that $n^2 = (2k)^2 = 4k^2 = 2(2k^2)$.
Since $2k^2$ is an integer, this means that there is some integer m (namely, $2k^2$) such that $n^2 = 2m$.

Thus n^2 is even.

Direct Proof Example 2

Give direct proof that : **Theorem** : "If n is an odd integer, then n^2 is an odd integer".

Proof

We assume that the hypothesis of this condition is true " n is odd"

$n = 2k+1$ for some integer k

We want to show that n^2 is odd ,

thus $n^2 = (2k+1)^2$

$n^2 = 4k^2 + 4k + 1$

$n^2 = 2(2k^2 + 2k) + 1$

Therefore n^2 is of the form $2j + 1$

(with j the integer $2k^2 + 2k$), **thus** n^2 is odd

Proof by Contradiction

"When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth."

- Sir Arthur Conan Doyle, The Adventure of the Blanched Soldier

Proof by Contradiction

A proof by contradiction is a proof that works as follows:

To prove that P is true, assume that P is not true.

Based on the assumption that P is not true, conclude something impossible.

Assuming the logic is sound, the only option is that the assumption that P is not true is incorrect.

Conclude, therefore, that P is true.

Contradictions and Implications

Suppose we want to prove that P → Q is true by contradiction.

The proof will look something like this:

Assume that P → Q is false (i.e., P is true and Q is false).
Using this assumption, derive a contradiction.
Conclude that P → Q must be true.

A Simple Proof by Contradiction

Theorem: If n^2 is even, then n is even.

Proof: By contradiction; assume n^2 is even but n is odd.

Since n is odd, $n = 2k + 1$ for some integer k.

Then $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$.

Now, let $m = 2k^2 + 2k$. Then $n^2 = 2m + 1$, so by definition n^2 is odd. But this is impossible, since n^2 is even.

We have reached a contradiction, so our assumption was false. Thus if n^2 is even, n is even as well.

Proof by Contradiction

Prove: If p then q.

Proof strategy:

- Assume p and the negation of q.
- In other words, assume that p ∧ ¬q is true.
- Then arrive at a contradiction p ∧ ¬p (or something that contradicts a known fact).
- Since this cannot happen, our assumption must be wrong, thus, ¬q is false. q is true.

Proof by Contradiction Example 2

Prove: If (3n+2) is odd, then n is odd.

Proof:

- Given: (3n+2) is odd.
- Assume that n is not odd, that is n is even.
- If n is even, there is some integer k such that n=2k.
- (3n+2) = (3(2k)+2)=6k+2 = 2(3k+1), which is 2 times a number.
- Thus 3n+2 turned out to be even, but we know it’s odd.
- This is a contradiction. Our assumption was wrong.
- Thus, n must be odd.

Proof by Contradiction Example 3

Prove theorem: There is no largest integer.
Proof: **Step 1. Contrary assumption**: Assume that there is a largest integer. Call it B (for “biggest”).
Step 2. Show this assumption leads to a contradiction: Consider C = B + 1. C is an integer because it is the sum of two integers. Also, C > B, which means that B is not the largest integer after all. Thus, we have reached a contradiction. The only flaw in our reasoning is the initial assumption that the theorem is false. Thus, we conclude that the theorem is correct.

Proof by Contrapositive

- The contrapositive of "**If P, then Q**" is the statement "If not Q, then not P."

Example 1:

- "If **I stored the cat food inside**, then the raccoons wouldn't have stolen my cat food."
- Contrapositive: "If the raccoons stole my cat food, then **I didn't store it inside.**"

Example 2:

- "If **I had been a good test subject**, then I would have received cake."
- Contrapositive: "If I didn't receive cake, then **I wasn't a good test subject**."

Notation

- Recall that we can write “If P, then Q” as $P \rightarrow Q$.

- **Notation**: We write “not P” as $\neg P$.

Examples:
- “If P is false, then Q is true:” $\neg P \rightarrow Q$
- “Q is false whenever P is false:” $\neg P \rightarrow \neg Q$

The contrapositive of $P \rightarrow Q$ is $\neg Q \rightarrow \neg P$.

An Important Result

Theorem: If $\neg Q \rightarrow \neg P$, then $P \rightarrow Q$.

Proof: By contradiction. Assume that $\neg Q \rightarrow \neg P$, but that $P \rightarrow Q$ is false. Since $P \rightarrow Q$ is false, it must be true that P is true and Q is false, i.e., $\neg Q$ is true. Since $\neg Q$ is true and $\neg Q \rightarrow \neg P$, we know that $\neg P$ is true.

But this means that we have shown P and $\neg P$, which is impossible.

We have reached a contradiction, so if $\neg Q \rightarrow \neg P$, then $P \rightarrow Q$.

An Important Proof Strategy

To show that P → Q, you may instead show that ¬Q → ¬P.

This is called a proof by contrapositive.

Proof by Contrapositive Example

Theorem: If **n^2 is even**, then n is even.

Proof: By contrapositive; we prove that if n is odd, then **n^2 is odd**.

Since n is odd, n = 2k + 1 for some integer k. Then

$n^2 = (2k + 1)^2$

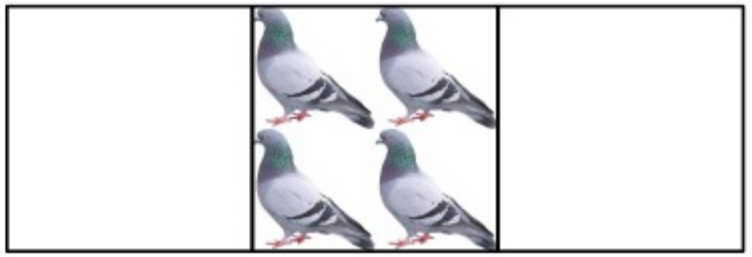
$n^2 = 4k^2 + 4k + 1$

$n^2 = 2(2k^2 + 2k) + 1.$

Since $(2k^2 + 2k)$ is an integer, n^2 is odd.

Proof by Contrapositive Example 2 : **The pigeonhole principle**

- Suppose that you have n pigeonholes.
- Suppose that you have $m > n$ pigeons.
- If you put the pigeons into the pigeonholes, some pigeonhole will have more than one pigeon in it.



Theorem: Let m objects be distributed into n bins. If $m > n$, then some bin contains at least two objects.

Proof: By contrapositive; we prove that if every bin contains at most one object, then $m \leq n$.

Let x_i be the number of objects in bin i. Since m is the number of total objects, we have that

$$m = \sum_{i=1}^{n} x_i$$

Since every bin has at most one object, $x_i \leq 1$ for all i. Thus

$$m = \sum_{i=1}^{n} x_i \leq \sum_{i=1}^{n} 1 = n$$

So $m \leq n$, as required. ■

Proof by counter example

A counter-example is an example that shows that a statement is not always true.

It is sufficient to just give one example.

This method is useful in **disproving** statements of the form "for all" (∀x, *P*(x)) or **proving** statements of the form "there exist" (∃x, *P*(x))

Proof by counter example

1. 4n+4 is always a multiple of 8 for all positive integer values of n.

Proof by counter example: When n=2, 4n+4=12,
12 is not a multiple of 8. Hence, proved.

2. $p-q <= p^2-q^2$ for all values of p and q.

Proof by counter example: when p=4, q=-5
$p-q=9$ and $p^2-q^2=-9$
So, p-q is not less than p^2-q^2 for all values of p and q. Hence proved.

Proof by Mathematical Induction

Induction, Intuitively

- It's true for 0.
- Since it's true for 0, it's true for 1.
- Since it's true for 1, it's true for 2.
- Since it's true for 2, it's true for 3.
- Since it's true for 3, it's true for 4.
- Since it's true for 4, it's true for 5.
- Since it's true for 5, it's true for 6.
- …

Proof by Induction

- Suppose that you want to prove that some property P(n) holds of all natural numbers.

To do so:

- Prove that P(0) is true. (This is called the **basis** or the **base case**.)
- Prove that for all n ∈ ℕ, that if P(n) is true, then P(n + 1) is true as well. (This is called the **inductive step**.)
- P(n) is called the **inductive hypothesis**.

Conclude by induction that P(n) holds for all n.

Mathematical Induction Example 1 : Some Sums

$$1 = 1 = 1(1 + 1) / 2$$
$$1 + 2 = 3 = 2(2 + 1) / 2$$
$$1 + 2 + 3 = 6 = 3(3 + 1) / 2$$
$$1 + 2 + 3 + 4 = 10 = 4(4 + 1) / 2$$
$$1 + 2 + 3 + 4 + 5 = 15 = 5(5 + 1) / 2$$

Mathematical Induction Example 1 : Some Sums

Theorem: The sum of the first n positive natural numbers is n(n + 1)/2.

Proof: By induction. Let P(n) be “the sum of the first n positive natural numbers is n(n + 1) / 2.” We show that P(n) is true for all n ∈ ℕ.

For our base case, we need to show P(0) is true, meaning that the sum of the first zero positive natural numbers is 0(0 + 1)/2. Since the sum of the first zero positive natural numbers is 0 = 0(0 + 1)/2, P(0) is true.

Mathematical Induction Example 1 : Some Sums

For the inductive step, assume that for some $n \in \mathbb{N}$ that $P(n)$ holds, meaning that $1 + 2 + \ldots + n = n(n + 1) / 2$. We need to show that $P(n + 1)$ holds, meaning that the sum of the first $n + 1$ natural numbers is $(n + 1)(n + 2)/2$.

Consider the sum of the first $n + 1$ positive natural numbers. This is the sum of the first n positive natural numbers, plus $n + 1$. By the inductive hypothesis, this is given by

$$1+\ldots+n+(n+1)=\frac{n(n+1)}{2}+n+1=\frac{n(n+1)+2(n+1)}{2}=\frac{(n+1)(n+2)}{2}$$

Thus $P(n + 1)$ is true, completing the induction.

The **principle of mathematical induction** states that if for some property $P(n)$, we have that

$P(0)$ is true

If it starts …

… and it keeps going …

and

For any $n \in \mathbb{N}$, we have $P(n) \to P(n+1)$

Then

… then it's always true.

For any $n \in \mathbb{N}$, $P(n)$ is true.

MIT-WPU
।। विश्वशान्तिर्ध्रुवं ध्रुवा ।।

Mathematical Induction: Example

- Show that any postage of ≥ 8¢ can be obtained using 3¢ and 5¢ stamps.
- First check for a few particular values:

  8¢ = 3¢ + 5¢

  9¢ = 3¢ + 3¢ + 3¢

  10¢ = 5¢ + 5¢

  11¢ = 5¢ + 3¢ + 3¢

  12¢ = 3¢ + 3¢ + 3¢ + 3¢
- How to generalize this?

Mathematical Induction: Example

- Let P(n) be the sentence "n cents postage can be obtained using 3¢ and 5¢ stamps".
- Want to show that

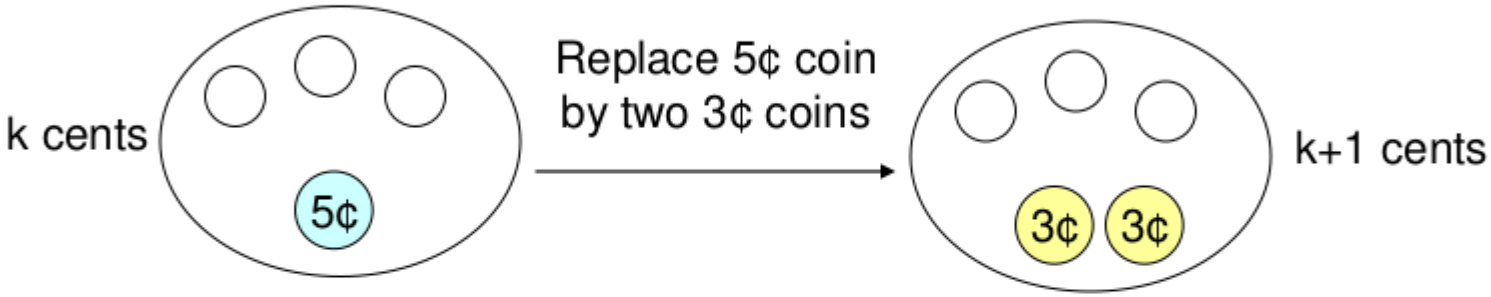
  "P(k) is true" *implies* "P(k+1) is true"

  for any k ≥ 8¢.
- 2 cases: 1) P(k) is true and the k cents contain at least one 5¢.

  2) P(k) is true and the k cents do not contain any 5¢.

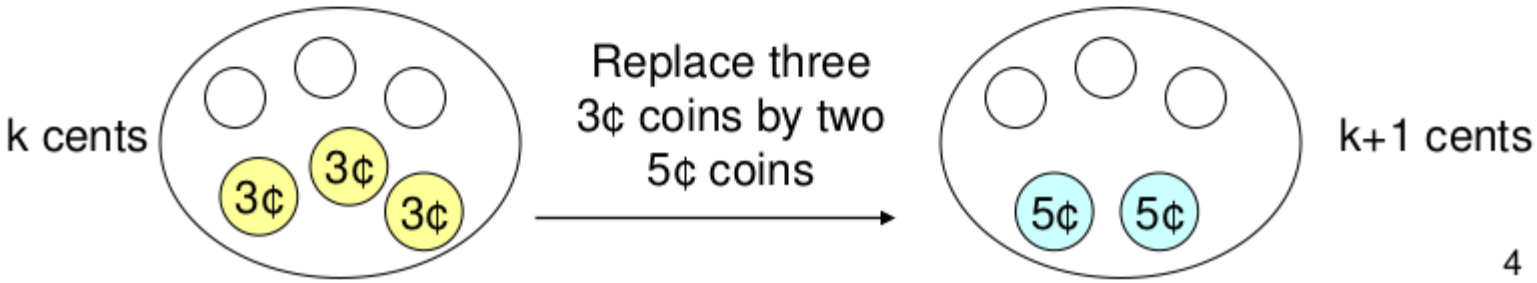
Mathematical Induction: Example

Case 1: k cents contain at least one 5¢ coin.



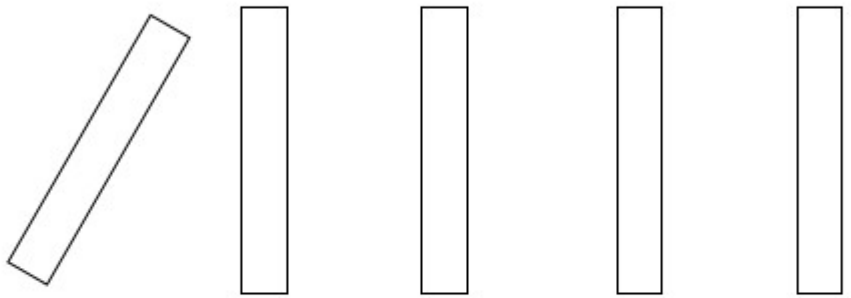
Case 2: k cents do not contain any 5¢ coin.

Then there are at least three 3¢ coins.



Domino Effect

- Mathematical induction works like domino effect:



- Let P(n) be “The n*th* domino falls backward”
- If (a) “P(1) is true”;
  (b) “P(k) is true” implies “P(k+1) is true”

  Then P(n) is true for every n

Principle of Mathematical Induction

Let P(n) be a predicate defined for integers n.

Suppose the following statements are true:

1. Basis step:

   P(a) is true for some fixed $a \in \mathbf{Z}$.

2. Inductive step: For all integers $k \geq a$,

   if P(k) is true then P(k+1) is true.

Then for all integers $n \geq a$, P(n) is true.

Example: Sum of Odd Integers

- **Proposition:** $1 + 3 + \ldots + (2n-1) = n^2$

  for all integers $n \geq 1$.

- **Proof (by induction):**

  1) Basis step:

  The statement is true for n=1: $1 = 1^2$.

  2) Inductive step:

  Assume the statement is true for some $k \geq 1$

  (inductive hypothesis),

  show that it is true for k+1 .

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Example: Sum of Odd Integers

EF BOOK: THOMAS CORMEN

➢ **Proof (cont.):**

The statement is true for k:

$$1+3+\ldots+(2k-1) = k^2 \qquad (1)$$

We need to show it for k+1:

$$1+3+\ldots+(2(k+1)-1) = (k+1)^2 \qquad (2)$$

Showing (2):

$$1+3+\ldots+(2(k+1)-1) = 1+3+\ldots+(2k+1) =$$

$$1+3+\ldots+(2k-1)+(2k+1) \overset{\text{by (1)}}{=}$$

$$k^2+(2k+1) = (k+1)^2 .$$

We proved the basis and inductive steps,

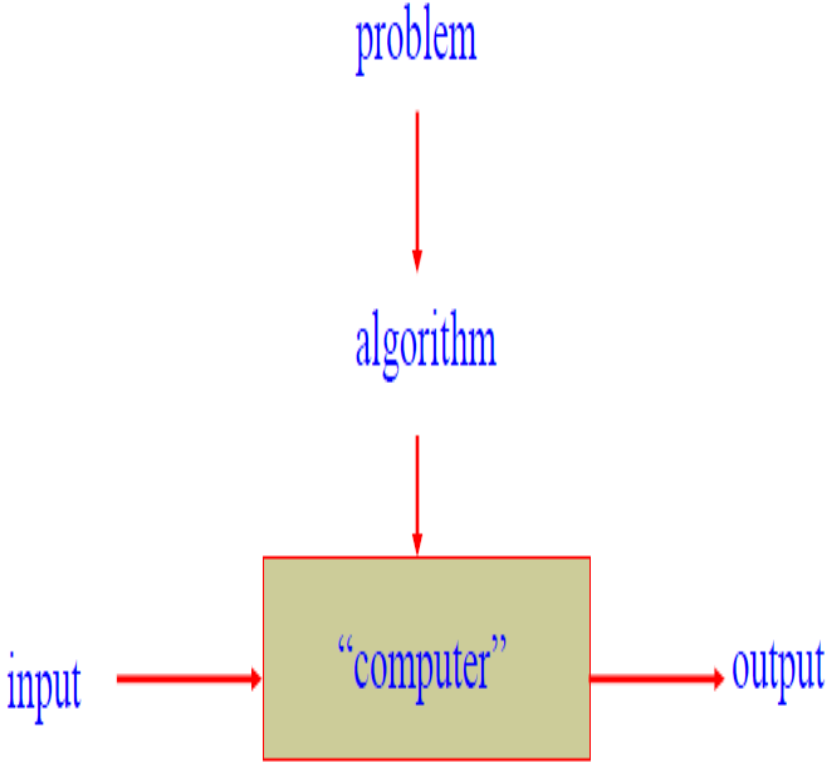
so we conclude that the given statement true. ■

REF BOOK: THOMAS CORMEN

What is an Algorithm?

An algorithm is a sequence of unambiguous instructions for solving a computational problem i.e., for obtaining a required output for any legitimate input in a finite amount of time.

It is any well defined computational procedure that takes some value or set of values as **input** and produces some value, or set of values as **output**



Algorithm & its Properties

More precisely, An *algorithm* is a finite set of instructions that accomplishes a particular task.

Algorithm must satisfy the following properties:

- Input : valid inputs must be clearly specified.
- Output : can be proved to produce the correct output given a valid input.
- Finiteness : terminates after a finite number of steps
- Definiteness : Each instruction must be clear and unambiguously specified
- Effectiveness : Every instruction must be sufficiently simple and basic.

Examples of Algorithms – Computing the Greatest Common Divisor of Two Integers

Problem: Find gcd (*m*,*n*), the greatest common divisor of two nonnegative, not both zero integers *m* and *n*

//First try –**School level algorithm:**

Step1: factorize m.($m = m_1 * m_2 * m_3 \ldots$)

Step2: factorize n. ($n = n_1 * n_2 * \ldots$)

Step 3: Identify common factors , multiply and return.

Pseudocode of Euclid's Algorithm

Algorithm : ***Euclid (m, n)***

//Computes gcd(m, n) by Euclid's algorithm

//Input: Two non negative, not-both-zero integers m and n

//Output: Greatest common divisor of m and n **gcd(m, n) = gcd(n, m mod n)**

while (m does not divide n) **do**

    r = n mod m

    n = m

    m = r

return m

Questions:

Finiteness: how do we know that Euclid's algorithm actually comes to a stop?

Definiteness: non-ambiguity

Effectiveness: effectively computable.

Which algorithm is faster, the Euclid's or previous one?

Example of Euclid's Algorithm

Simple Algorithm

m = 36, n = 48

m = 2 * 2 * 3 * 3

n = 2 * 2 * 2 * 2 * 3

Common factors

2, 2, 3

GCD = 12

9 divisions

Euclid Algorithm

36 divides 48

r = 48 mod 36

n = 36

m = 12

Return 12

2 divisions

Another example:

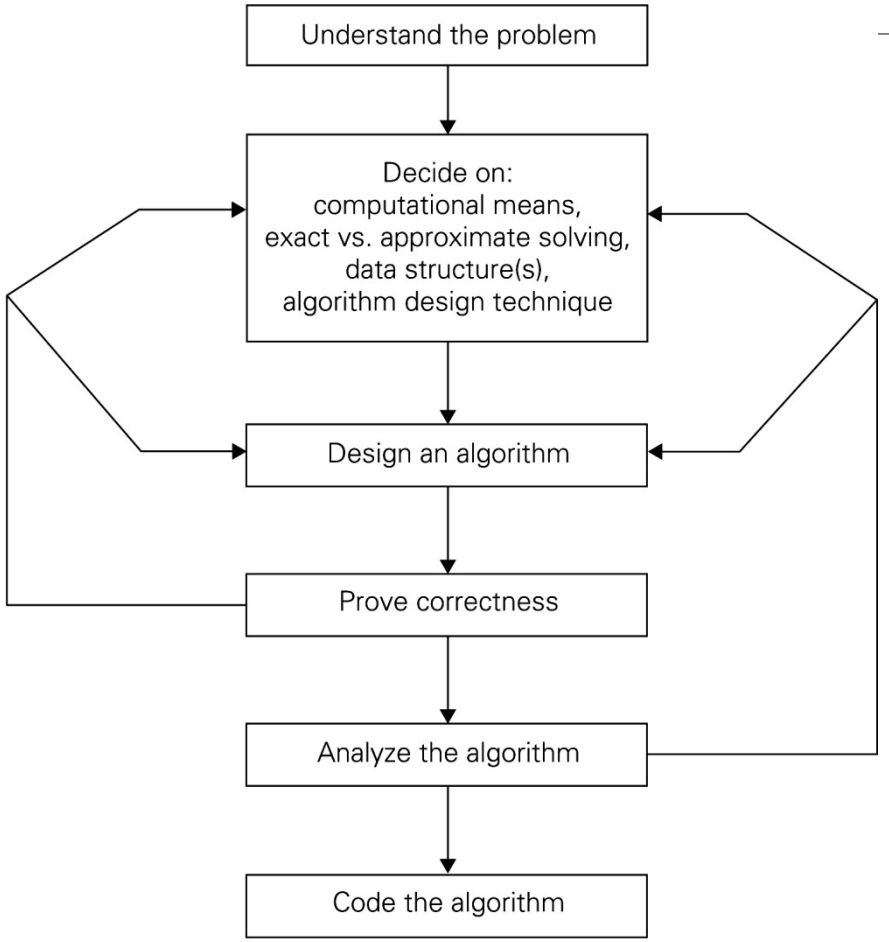
m = 434 and n = 966

Fundamentals of Algorithmic Problem Solving

- Understanding the problem : Asking questions, do a few examples by hand, think about special cases, etc.
- Deciding on : Exact vs. approximate problem solving
- Appropriate data structure
- Design an algorithm
- Proving correctness
- Analyzing an algorithm
  - Time efficiency : how fast the algorithm runs
  - Space efficiency: how much extra memory the algorithm needs.
- Coding an algorithm

REF BOOK: THOMAS CORMEN

Algorithm Design and Analysis Process



REF BOOK: THOMAS CORMEN

Algorithms as a Technology

Even if computers were infinitely fast and memory was plentiful and free

- Study of algorithms still important – still need to establish algorithm correctness
- Since time and space resources are infinitely fast, any correct algorithm for solving a problem would do.

Real-world computers may be fast but not infinitely fast.

Memory is cheap but not free

Hence Computing time is bounded resource and so need space in memory.

We should use resources wisely by efficient algorithm in terms of space and time

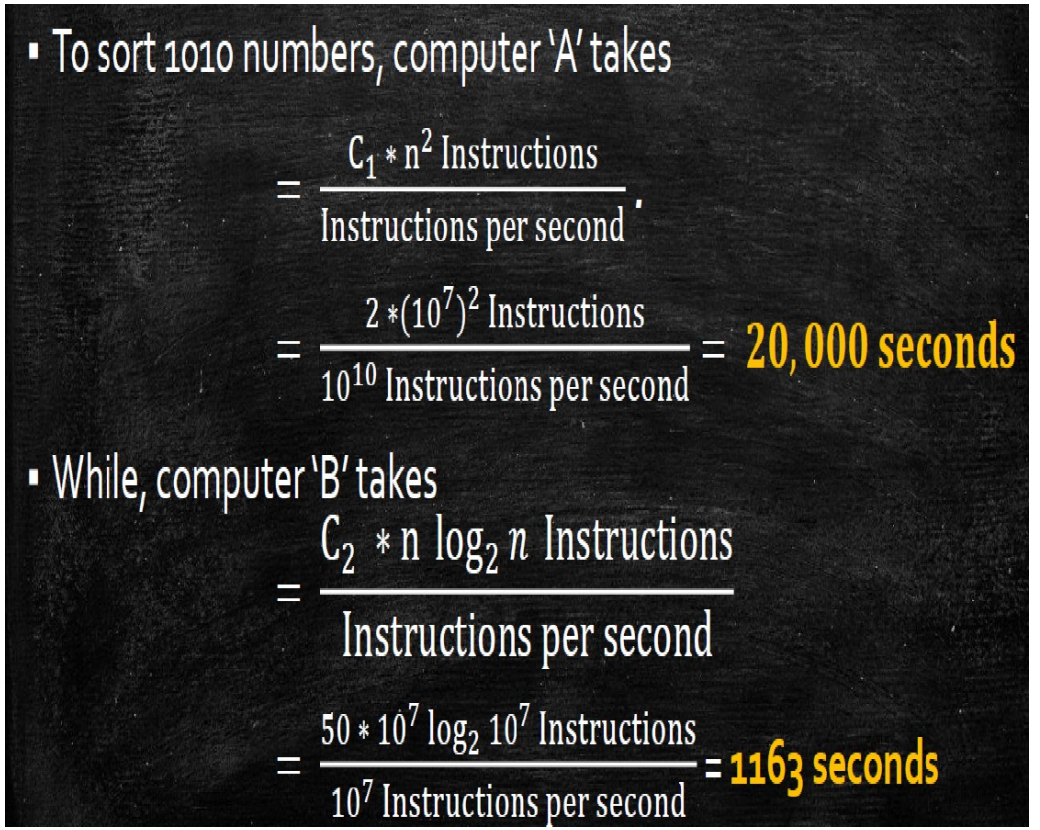
REF BOOK: THOMAS CORMEN

Algorithm Efficiency

Time and space efficiency are the goal

Algorithms often differ dramatically in their efficiency

- Example: Two sorting algorithms
  - **INSERTION-SORT** – time efficiency is $c_1 n^2$
  - **MERGE-SORT** – time efficiency is $c_2 n \log n$
- For which problem instances would one algorithm be preferable to the other?
- For example,
- A faster computer 'A' (10^{10} instructions/sec) running insertion sort against a slower computer 'B' (10^7 instructions/sec) running merge sort. Suppose that c1=2 , c2=50 and n= 10^7.



Efficiency

Problem Size	Machine A Insertion-Sort	Machine B Merge-Sort
n	$2n^2/10^9$	$50n\log n/10^7$
10,000	0.20	0.66
50,000	5.00	3.90
100,000	20.00	8.30
500,000	500.00	47.33
1,000,000	2,000.00	99.66
5,000,000	50,000.00	556.34
10,000,000	200,000.00	1,162.67
50,000,000	5,000,000.00	6,393.86

Methods of Specifying an Algorithm

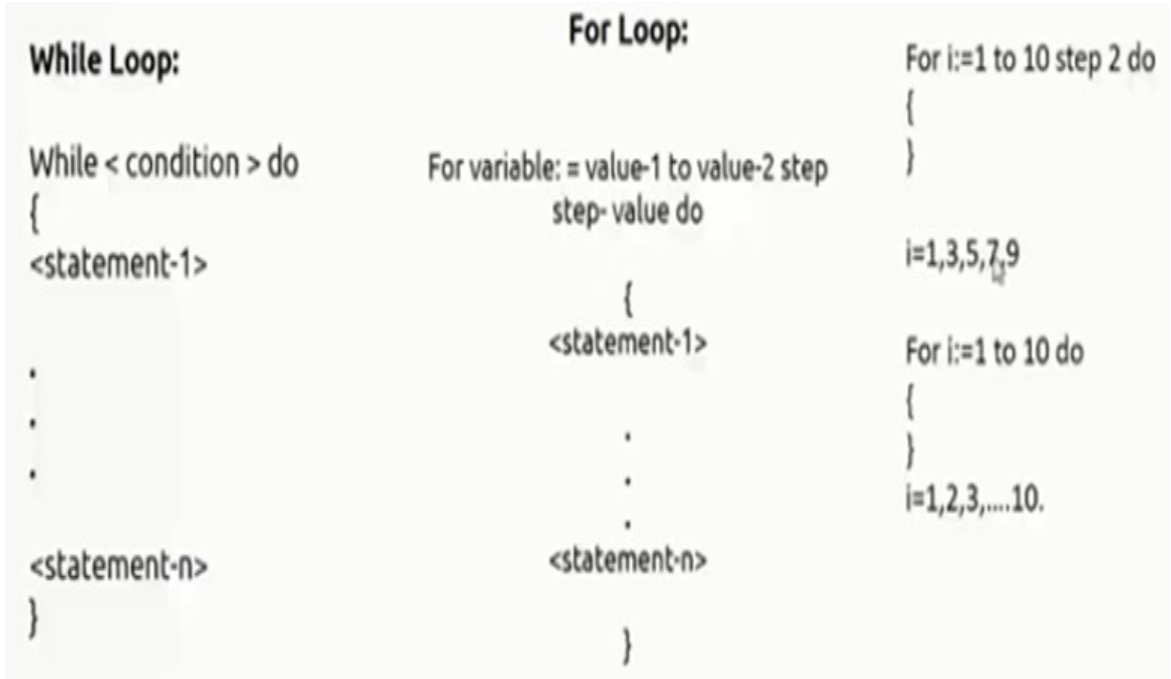
- **Natural language** : Ambiguous
- **Flowchart** : Graphic representations called flowcharts , only if the algorithm is small and simple.
- **Pseudo code**
  - A mixture of a natural language and programming language-like structures
  - Precise and concise.

Pseudo code Conventions

- Comments begin with // and continue until the end of line.
- Blocks are indicated with matching braces :{ and }. A compound statement (i.e., a collection of simple statements) can be represented as as a block. The body of a procedure also forms a block. Statements are delimited by;
- Assignment of values to variables is done using the assignment statement
- ⟨variable⟩:=⟨expression⟩;
- There are two Boolean values ***true*** and ***false***. In order to produce these values,
  - Logical operators : ***and***, ***or***, and ***not***
  - Relational operators<, ≤, =, ≠, ≥, and >
- Elements of multidimensional arrays are accessed using [and]. Ex A[i,j]

Pseudo code Conventions

- Looping statement can be employed as follows : (while, for , repeat-until)



While Loop:

```
While < condition > do
{
<statement-1>
.
.
.
<statement-n>
}
```

For Loop:

```
For variable: = value-1 to value-2 step
step- value do
        {
        <statement-1>
        .
        .
        .
        <statement-n>
        }
```

```
For i:=1 to 10 step 2 do
{
}

i=1,3,5,7,9

For i:=1 to 10 do
{
}
i=1,2,3,....10.
```

A conditional statement has the following forms:

- **If** ⟨condition⟩ **then** ⟨statement⟩
- **If** ⟨condition⟩ **then** ⟨statement 1⟩ **else** ⟨statement 2⟩

We also employ the following case statement:

case

{

⟨condition 1⟩: ⟨statement 1⟩

....

⟨condition n⟩: ⟨statement n⟩

else: ⟨statement n + 1⟩

}

Pseudo code Conventions

Input and output are done using the instructions ***read*** and ***write***. No format is used to specify the size of input or output quantities.

An algorithm consists of a heading and a body. The heading takes the form

Algorithm Name (⟨parameter list⟩)

{

// body

}

Example : Algorithm to find and return the maximum of n given numbers:

Algorithm Max (A, n)

// A is an array of size n.

{

Result :=A[1];

For i :=2 to n do

If A[i] > Result **then** Result :=A[i];

Return Result;

}

REF BOOK: THOMAS CORMEN

Analysis of Algorithms

Two main issues related to algorithms

- How to design algorithms
- How to analyze algorithm efficiency

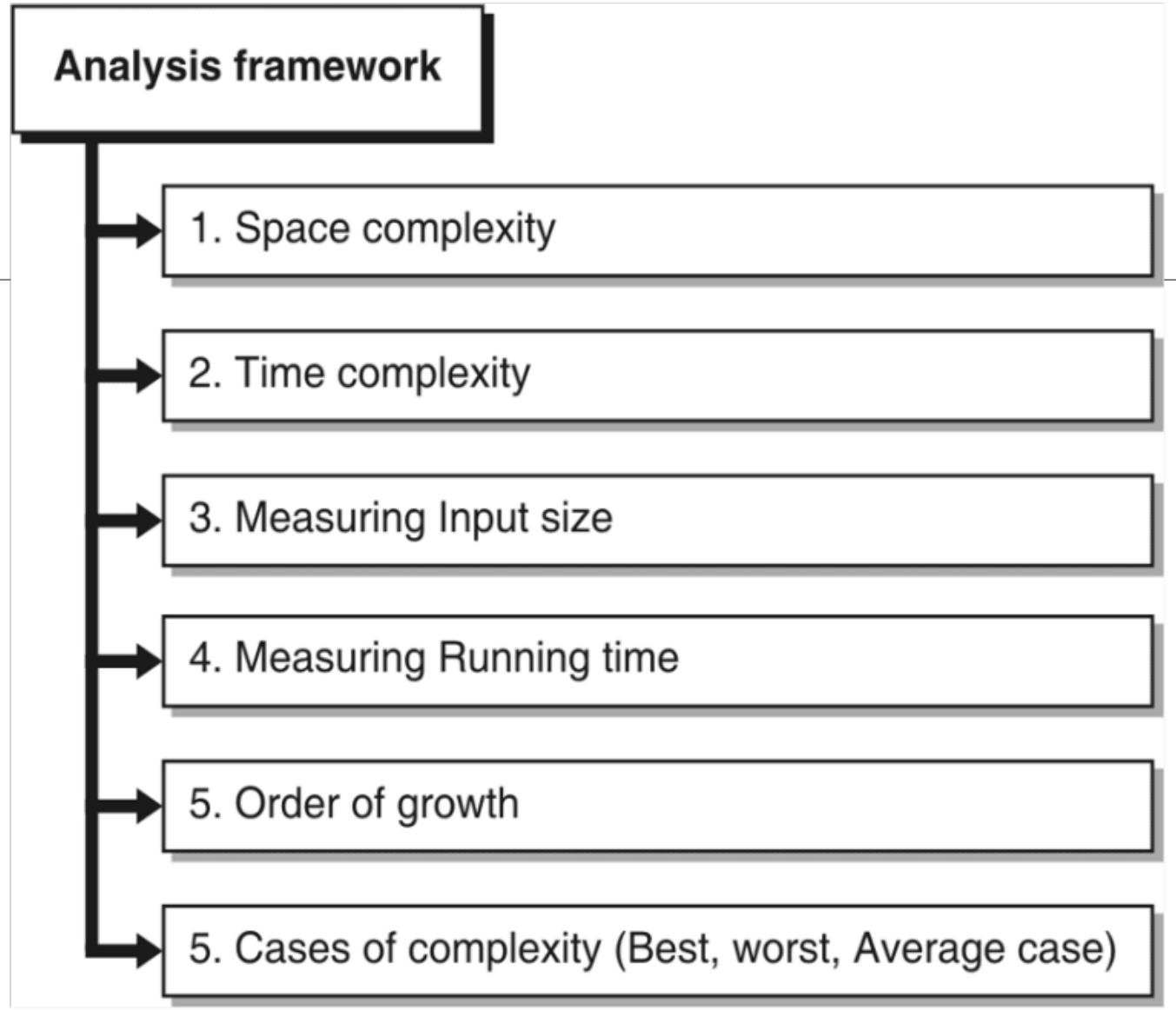
Analysis of Algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency
- Does there exist a better algorithm?
  - lower bounds
  - optimality

Efficiency Analysis Framework of Algorithm

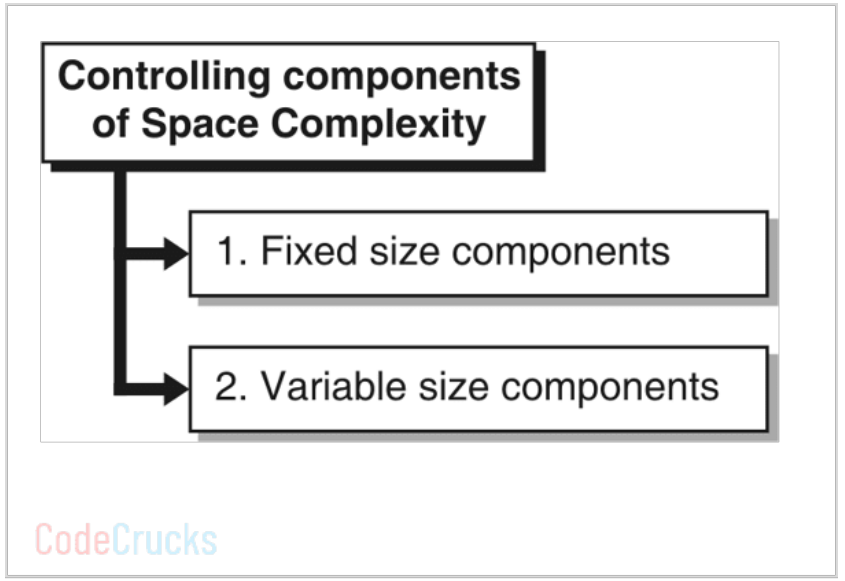
Efficiency Analysis of the algorithm is the way of finding the resources required by the algorithm. This helps to decide the best algorithm among the set of candidate algorithms. Resources may be time, space, power consumption, communication bandwidth, computer hardware etc.

REF BOOK: THOMAS CORMEN



Space Complexity

Problem-solving using a computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve a given problem is called the space complexity of the algorithm.



Time Complexity

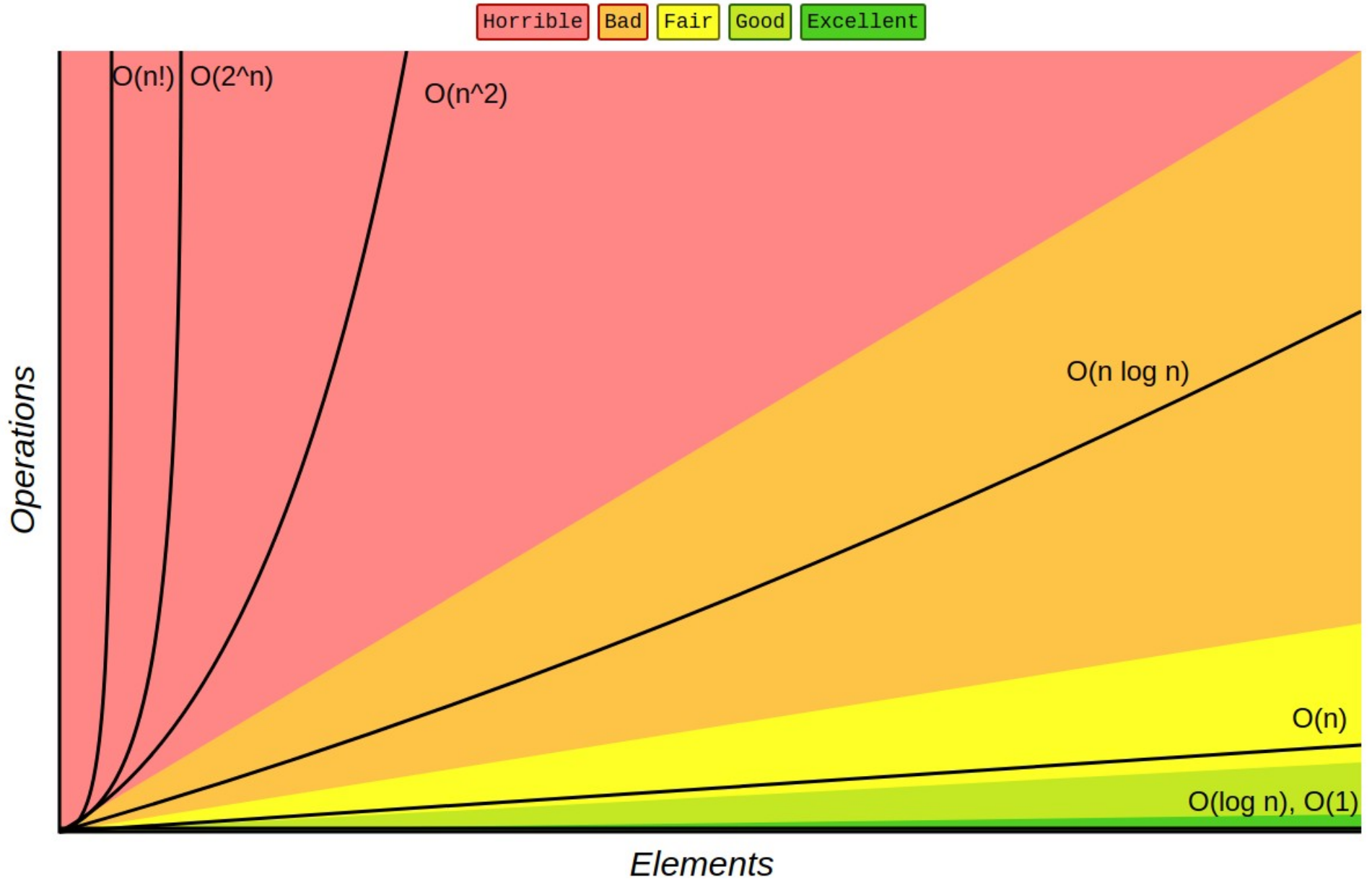
The valid algorithm executes in a finite period of time. The time complexity of the algorithm refers to how long it takes the algorithm to solve a given problem. In algorithm analysis, time complexity is very useful measure.

Common Rates of Growth

Let n be the size of input to an algorithm, and k some constant. The following are common rates of growth.

- Constant: $\Theta(k)$, for example $\Theta(1)$
- Linear: $\Theta(n)$
- Logarithmic: $\Theta(\log_k n)$
- Linear : n $\Theta(n)$ or n log n: $\Theta(n \log_k n)$
- Quadratic: $\Theta(n^2)$
- Polynomial: $\Theta(n^k)$
- Exponential: $\Theta(k^n)$

K: THOMAS CORMEN



REF BOOK: NPTEL

Why Analysis?

- Practical reasons:
  - Resources are scarce
  - Greed to do more with less
  - Avoid performance bugs
- Core Issues:
- Predict performance
  - How much time does binary search take?
- Compare algorithms
  - How quick is Quicksort?
- Provide guarantees
  - Size not with standing, Red-Black tree inserts in O(log n)
- Understand theoretical basis
  - Sorting by comparison cannot do better than (n log n)

What to analyze?

Core issue: Cannot control what we cannot measure

Time

- The ***time complexity, T(n),*** taken by a program P is the sum of the running times for each statement executed

Space

The ***space complexity*** of a program is the amount of memory that it needs to run to completion

Examples : Sum of Natural Numbers

```
// Sum of Natural Numbers
Algorithm sum (a, n)
{
s := 0 ;
For i := 1 to n do
s := s + a[i];
return s;
}
```

Time T(n) = n (additions)

Space S(n) = 2 (n, s)

Tabular Method

Iterative function to sum a list of numbers (steps/execution)

Statement	s/e	Frequency	Total steps
Algorithm sum (a, n)	0	-	0
{	0	-	0
s := 0 ;	1	1	1
For i := 1 to n do	1	n+1	n+1
s := s + a[i];	1	n	n
return s;	1	1	1
}			
Total T(n)			2n+3

Machine Model: Random Access Machine (RAM) Computing Model

- Input data & size
- Operations
- Intermediate Stages
- Output data & size

REF BOOK: THOMAS CORMEN

Asymptotic Analysis

Core Idea: Cannot compare actual times; hence compare Growth or how time increases with input

- O notation (“Big Oh”)
- Ω notation (Omega)
- Θ notation (Theta)
- o notation (Little oh)
- ω notation (Little omega)

Asymptotic Notations : O-notation **(Big Oh)**

Asymptotic Upper Bound

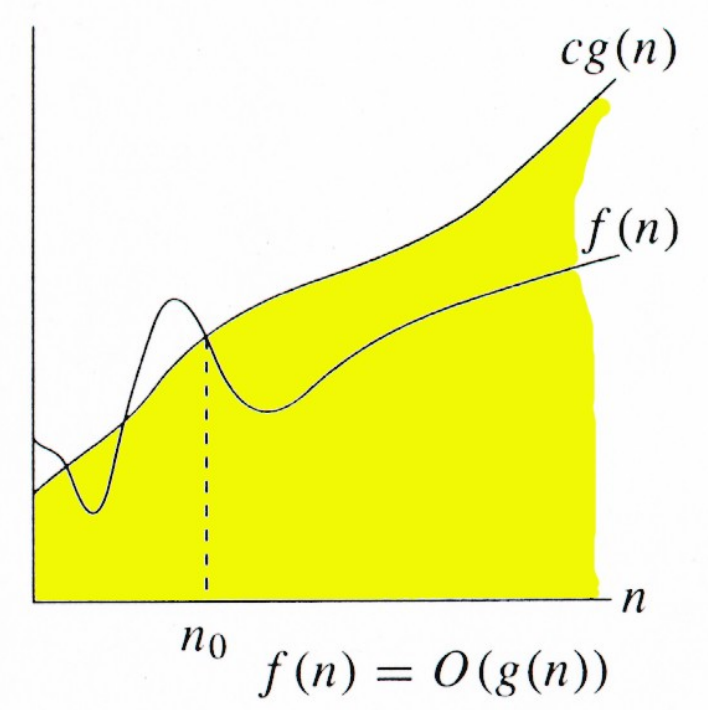
For a given function g(n), we denote O(g(n)) as the set of functions:

O(g(n)) = { f(n)| there exists positive constants c and n0

such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n0$ }

It is used to represent the worst case growth of an algorithm in time or a data structure in space when they are dependent on n, where n is big enough

Example :



Big Oh - Example

$f(n) = n^2 + 5n = O(n2)$

$g(n) = n^2$ ……. c = 2

n	$n^2 + 5n$	$2n^2$
1	6	2
2	14	8
5	50	50

$f(n) <= c\ g(n)$ for all $n> = n_0$ where $c=2$ & $n_0 =5$

Big Oh - Example

Let $f(N) = 2N^2$. Then

- $f(N) = O(N^4)$ (loose bound)
- $f(N) = O(N^3)$ (loose bound)
- $f(N) = O(N^2)$ (It is the best answer and the bound is asymptotically tight.)

$O(N^2)$: reads “order N-squared” or “Big-Oh N-squared”.

Big Oh - Example

Prove that if $T(n) = 15n^3 + n^2 + 4$, $T(n) = O(n^3)$.

Proof.

Let $c = 20$ and $n_0 = 1$.

Must show that $0 \leq f(n)$ and $f(n) \leq cg(n)$.

$0 \leq 15n^3 + n^2 + 4$ for all $n \geq n_0 = 1$.

$f(n) = 15n^3 + n^2 + 4 \leq 20n^3$ ($20g(n) = cg(n)$)

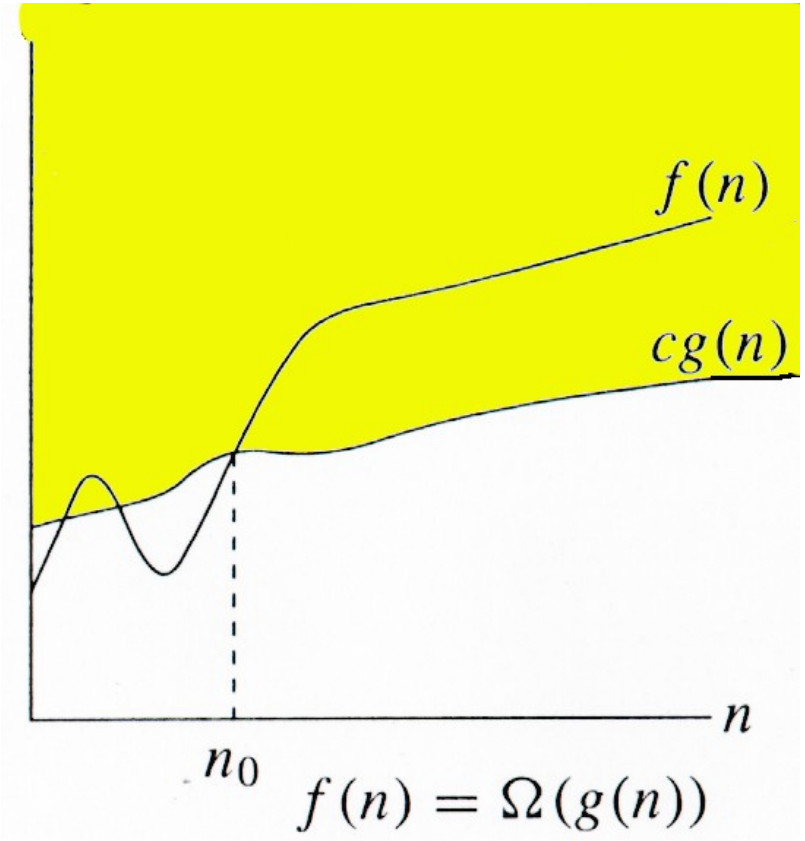
As per definition of Big O, hence $T(n) = \mathbf{O(n^3)}$

Asymptotic Notations Ω-notation

Asymptotic lower bound

Ω (g(n)) represents a set of functions such that:

Ω(g(n)) = {f(n): there exist positive
constants c and n0 such that
0 ≤ c g(n) ≤ f(n) for all n≥ n0}



Big Omega - Example

Example 1 :

$f(n) = n^2 + 5n$

$g(n) = n^2$ ……. c = 1

n	$n^2 + 5n$	$c*n^2$
1	6	1
2	14	4
5	50	25

$f(n) >= c\ g(n)$ for all $n> = n_0$ where c=1 & n_0 =1

Example 1 :

Prove that if $T(n) = 15n^3 + n^2 + 4$, $T(n) = \Omega(n^3)$.

Proof.

Let c = 15 and $n_0 = 1$.

Must show that $0 \leq cg(n)$ and $cg(n) \leq f(n)$.

$0 \leq 15n^3$ for all $n \geq n_0 = 1$.

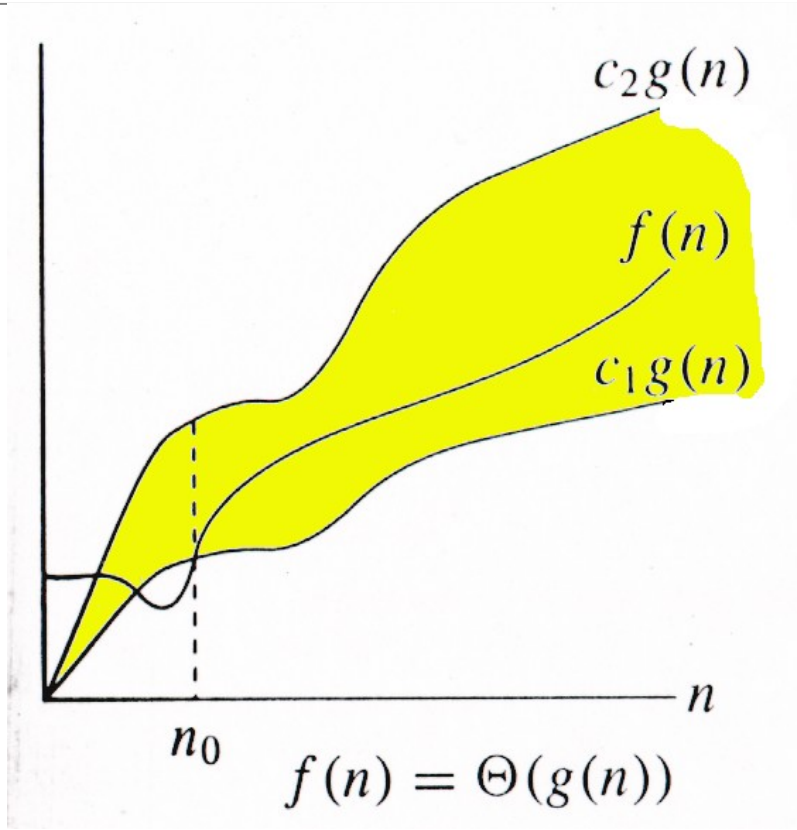
$cg(n) = 15n^3 \leq 15n^3 + n^2 + 4 = f(n)$

Asymptotic Notations Θ-notation

Asymptotic tight bound

Θ (g(n)) represents a set of functions such that:

Θ (g(n)) = {f(n): there exist positive
constants c1, c2, and n0such
that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n)
for all n≥ n0 }



Theta Example

$f(N) = \Theta(g(N))$ iff $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$

It can be read as “f(N) has order exactly g(N)”.

The growth rate of f(N) equals the growth rate of g(N). The growth rate of f(N) is the same as the growth rate of g(N) for large N.

Theta means the bound is the tightest possible.

If T(N) is a polynomial of degree k, $T(N) = \Theta(N^k)$.

For logarithmic functions, $T(\log_m N) = \Theta(\log N)$.

o notation (Little oh)

The function $f(n) = o(g(n))$ iff

$$\lim_{n->\infty} \frac{f(n)}{g(n)} = 0$$

$\boldsymbol{o(g(n))} = \{f(n)$: $\forall\ c > 0$, $\exists\ n_0 > 0$ such that
$\forall\ n \geq n_0$, we have $0 \leq f(n) < cg(n)\}$.

Example: The function $3n + 2 = o(n^2)$ as

$$\lim_{n->\infty} \frac{3n+2}{n^2} = 0$$

ω notation (Little omega)

The function $f(n) = \omega(g(n))$ iff

$$\lim_{n->\infty} \frac{g(n)}{f(n)} = 0$$

$\omega(g(n))$ = {$f(n)$: **∀** $c > 0$, **∃** $n_0 > 0$ such that
∀ $n \geq n_0$, we have $0 \leq cg(n) < f(n)$}.

REF BOOK: THOMAS CORMEN

Asymptotic Notations

It is a way to compare “sizes” of functions

O-notation ----------Less than equal to (“≤”)

Θ-notation ----------Equal to (“=“)

Ω-notation ----------Greater than equal to (“≥”)

$O \approx \leq$

$\Omega \approx \geq$

$\Theta \approx =$

$o \approx <$

$\omega \approx >$

REF BOOK: THOMAS CORMEN/SAHNI

Examples On Asymptotic Notations

Explain How is f(x) = 4n^2 – 5n + 3 is O(n^2)

Show that

1) $30n+8$ is $O(n)$
2) $100n + 5 \neq \Omega(n^2)$
3) $5n^2 = \Omega(n)$
4) $100n + 5 = O(n^2)$
5) $n^2/2 - n/2 = \Theta(n^2)$

Algorithm Design Techniques/Strategies

REF BOOK: THOMAS CORMEN

- Brute force
- Divide and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound

Divide & Conquer

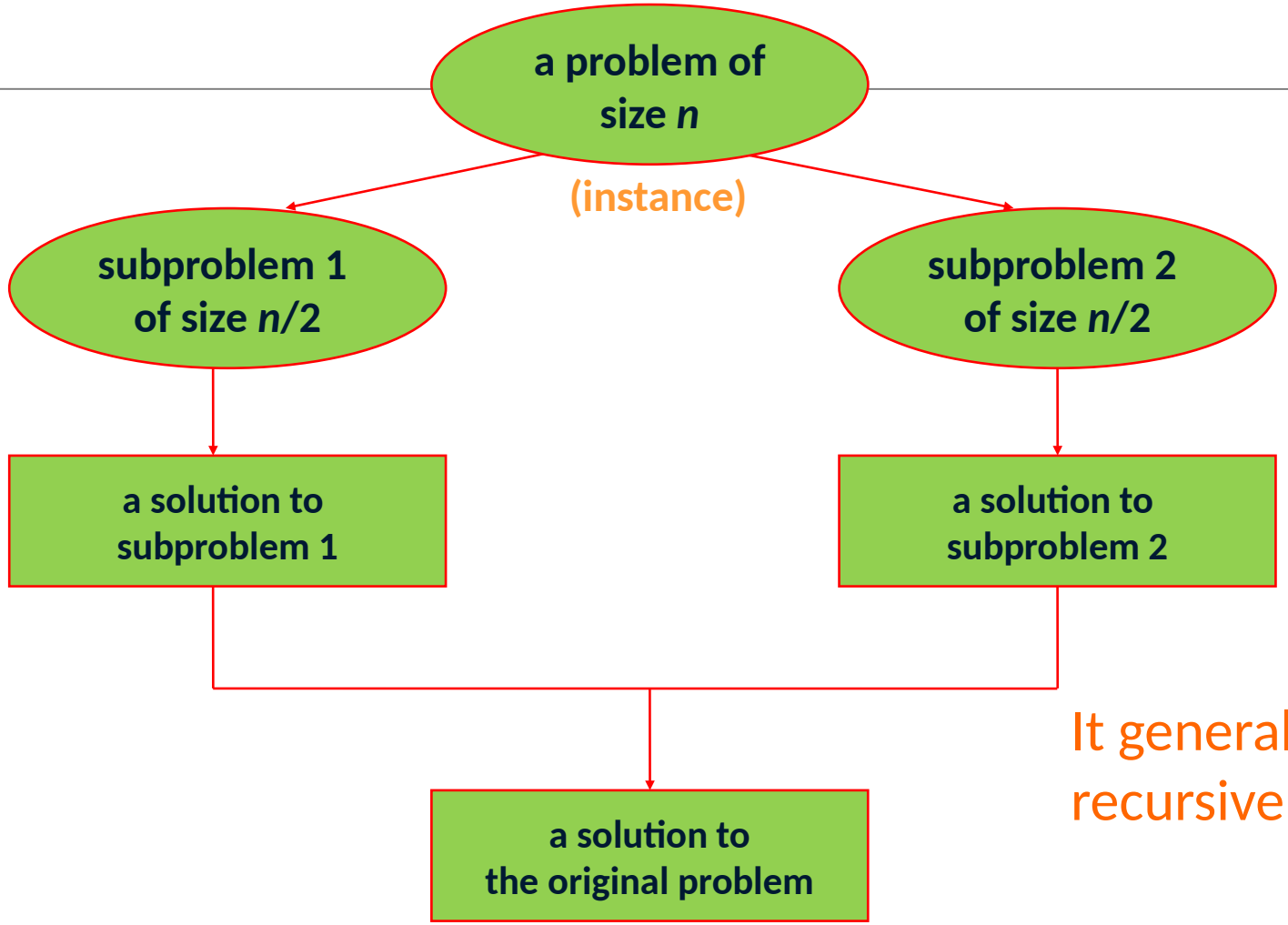
Control Abstraction

```
DANDC (P)
{
if SMALL (P) then return S (p);
else
{
divide p into smaller instances p1, p2, …. Pk, k >= 1;
apply DANDC to each of these sub problems;
return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
}
}
```

Divide the problem into smaller sub problems
Conquer the sub problems by solving them recursively.
Combine the solutions to the sub problems into the solution of the original problem.

Divide-and-Conquer Technique (cont.)

REF BOOK: INTERNET



It general leads to a recursive algorithm!

- An equation or inequality that describes a function in terms of its value on smaller inputs.

$$T(n) = T(n-1) + n$$

- Recurrences arise when an algorithm contains recursive calls to itself
- What is the actual running time of the algorithm?
- Need to solve the recurrence
  - Find an explicit formula of the expression
  - Bound the recurrence by an expression that involves n

Time complexity of the general algorithm

REF BOOK: THOMAS CORMEN

A Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs

Special techniques are required to analyze the space and time required

$$T(n)=\begin{cases} aT(n/b)+D(n)+C(n) & , n \geq c \\ O(1) & , n < c \end{cases}$$

Time complexity (recurrence relation):

where D(n) : time for splitting

C(n) : time for conquer

c : a constant

Example Recurrences

- $T(n) = T(n-1) + n$ $\Theta(n^2)$
  - Recursive algorithm that loops through the input to eliminate one item
- $T(n) = T(n/2) + c$ $\Theta(\lg n)$
  - Recursive algorithm that halves the input in one step
- $T(n) = T(n/2) + n$ $\Theta(n)$
  - Recursive algorithm that halves the input but must examine every item in the input
- $T(n) = 2T(n/2) + 1$ $\Theta(n)$
  - Recursive algorithm that splits the input into 2 halves and does a constant amount of other work

REF BOOK: THOMAS CORMEN

Methods for Solving recurrences

- Substitution Method
  - We guess a bound and then use mathematical induction to prove our guess correct.
- Recursion Tree
  - Convert recurrence into tree
- Master Method

REF BOOK: INTERNET

Math You need to Review

properties of logarithms:

$\log_b(xy) = \log_b x + \log_b y$

$\log_b (x/y) = \log_b x - \log_b y$

$\log_b xa = a\log_b x$

$\log_b a = \log_x a/\log_x b$

- **properties of exponentials**:

$a^{(b+c)} = a^b a^c$

$a^{bc} = (a^b)^c$

$a^b / a^c = a^{(b-c)}$

$b = a^{\log_a b}$

$b^c = a^{c*\log_a b}$

REF BOOK: THOMAS CORMEN

Substitution Method

- Sum of n Natural Numbers

```
int sum(int n) {
    int s = 0;
    for(; n > 0; --n)
        s = s + n;
    return s;
}
```

$$\begin{aligned} T(n) &= T(n-1)+1, & n>1 \\ &= 1, & n=1 \end{aligned}$$

Solve Recurrence in Long hand:

$$\begin{aligned} T(n) &= T(n-1)+1 \\ &= T(n-2)+(1+1) \\ &= \cdots \\ &= T(1)+(n-1) \\ &= 1+(n-1) \\ &= n \end{aligned}$$

- Time $T(n)$ (additions)
- Space $S(n) = 2$

- Factorial (Recursive)

```
int fact(int n) {
    if (0 != n) return n*fact(n-1);
    return 1;
}
```

- Time $T(n)$ (multiplication)

$$\begin{aligned} T(n) &= T(n-1)+1, & n>0 \\ &= 0, & n=0 \end{aligned}$$

$$T(n) = n$$

Quick Sort

$$T(n) = 2T(n/2) + O(n)$$

$$= 2(2(n/2^2) + (n/2)) + n$$

$$= 2^2\, T(n/2^2) + n + n$$

$$= 2^2\, (T(n/2^3) + (n/2^2)) + n + n$$

$$= 2^3\, T(n/2^3) + \underline{n + n + n}$$

$$= \boldsymbol{n \log n}$$

Analysis of BINARY-SEARCH

Alg.: BINARY-SEARCH (A, lo, hi, x)

```
if (lo > hi)                                  ← constant time: c1
    return FALSE
mid ← ⌊(lo+hi)/2⌋                             ← constant time: c2
if x = A[mid]                                 ← constant time: c3
    return TRUE
if ( x < A[mid] )
    BINARY-SEARCH (A, lo, mid-1, x)           ← same problem of size n/2
if ( x > A[mid] )
    BINARY-SEARCH (A, mid+1, hi, x)           ← same problem of size n/2
```

- $T(n) = c + T(n/2)$
  - T(n) – running time for an array of size n

Binary Search

EXAMPLE 2: BINARY SEARCH

$T(n)=O(1) + T(n/2)$

$T(1)=1$

Above is another example of recurrence relation and the way to solve it is by Substitution.

$T(n)=T(n/2) +1$

$= T(n/2^2)+1+1$

$= T(n/2^3)+1+1+1$

$= \log n$

T(n)= O(logn)

The recursion-tree method

Convert the recurrence into a tree:

- Each node represents the cost incurred at various levels of recursion
- Sum up the costs of all levels

Used to “guess” a solution for the recurrence

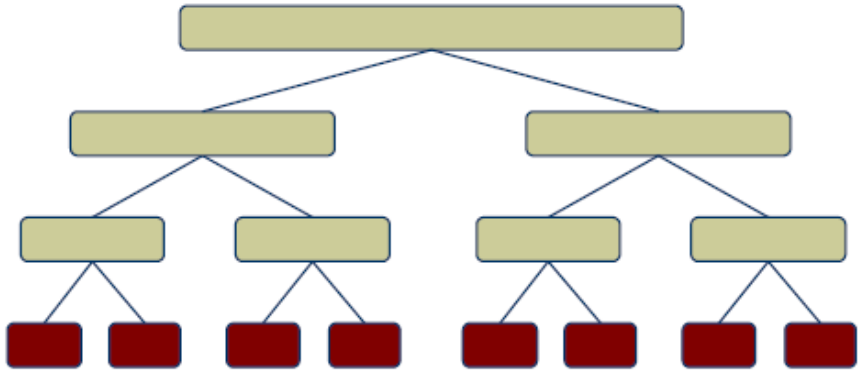
REF BOOK: THOMAS CORMEN

The Recursion Tree

Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
...	...	...	...



Total time = $bn + bn \log n$

(last level plus all previous levels)

REF BOOK: THOMAS CORMEN

Master Theorem

Master method provides a “cookbook” method for solving recurrences of the following form

$$T(n) = a\,T(n/b) + f(n)$$

where $a \geq 1$, $b > 1$. If f(n) is asymptotically positive function. T(n) has following asymptotic bounds:

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Example of Master Method

REF BOOK: THOMAS CORMEN

To use the master theorem, we simply plug the numbers into the formula

Example 1: $T(n) = 9T(n/3)+n$. Here $a = 9$, $b = 3$, $f(n) = n$, and $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\epsilon})$ for $\epsilon = 1$, case 1 of the master theorem applies, and the solution is $T(n) = \Theta(n^2)$.

Example 2: $T(n) = T(2n/3)+1$. Here $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^0 = 1$. Since $f(n) = \Theta(n^{\log_b a})$, case 2 of the master theorem applies, so the solution is $T(n) = \Theta(\log n)$.

Example 3: $T(n) = 3T(n/4) + n\log n$. Here $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. For $\epsilon = 0.2$, we have $f(n) = \Omega(n^{\log_4 3+\epsilon})$. So case 3 applies if we can show that $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n. This would mean $3\frac{n}{4}\log\frac{n}{4} \leq cn\log n$. Setting $c = 3/4$ would cause this condition to be satisfied.

Example 4: $T(n) = 2T(n/2)+n\log n$. Here the master method does not apply. $n^{\log_b a} = n$, and $f(n) = n\log n$. Case 3 does not apply because even though $n\log n$ is asymptotically larger than n, it is not polynomially larger. That is, the ratio $f(n)/n^{\log_b a} = \log n$ is asymptotically less than n^ϵ for all positive constants ϵ.

REF BOOK: SRIDHAR (OXFORD PUBLICATION)

Master Method (Simplified)

Let T(n) be a monotonically increasing function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$

$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then solution to recurrence relation is given as

Case 1: $T(n) \in \Theta(n^d)$ ----- if $a < b^d$

Case 2: $T(n) \in \Theta(n^d \log n)$ -------- if $a = b^d$

Case 3: $T(n) \in \Theta(n^{\log_b a})$ -------- if $a > b^d$

REF BOOK: THOMAS CORMEN/SAHNI

Divide-and-Conquer Examples

Sorting : Mergesort and Quicksort

Binary tree traversals

Binary search

Multiplication of large integers

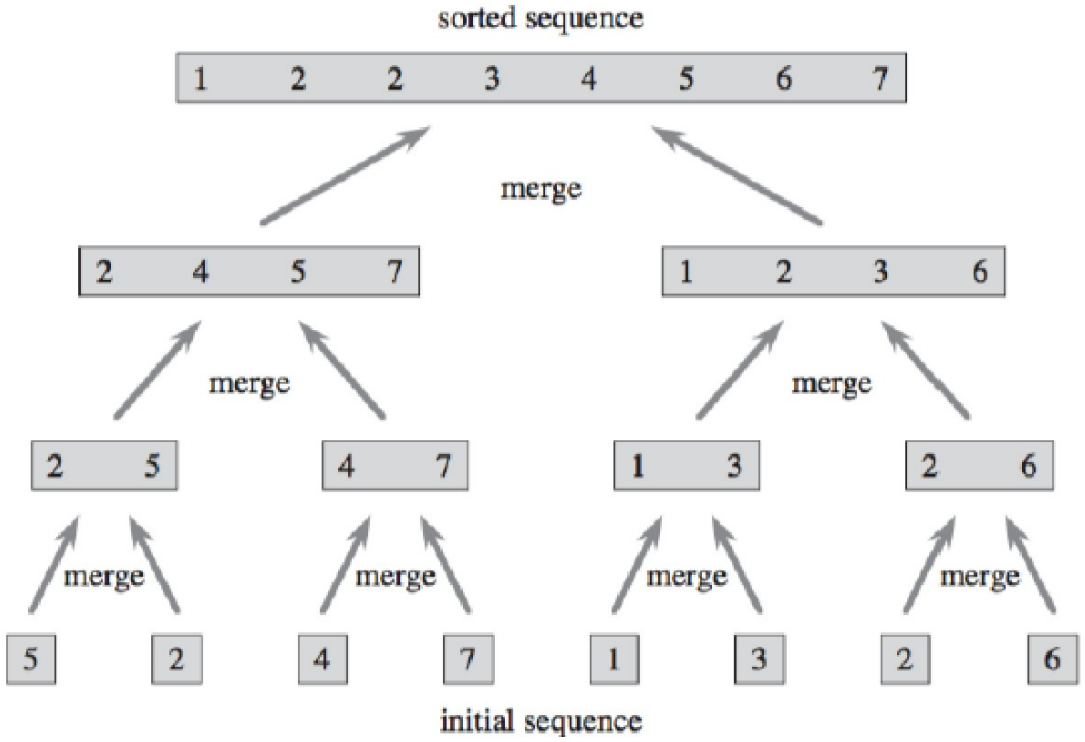
Matrix multiplication: Strassen’s algorithm

Closest-pair and convex-hull algorithms

REF BOOK: THOMAS CORMEN

Mergesort

Merge sort is a divide and conquer algorithm for sorting arrays. To sort an array, first you split it into two arrays of roughly equal size. Then sort each of those arrays using merge sort, and merge the two sorted arrays.



Pseudo code of Merge-Sort (A, p, r)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
1   if p < r
2       then q ← ⌊(p+r)/2⌋
3            MergeSort (A, p, q)
4            MergeSort (A, q+1, r)
5            Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(*A*, 1, *n*)

Procedure Merge

REF BOOK: THOMAS CORMEN

```
Merge(A, p, q, r)
1  n1 ← q – p + 1
2  n2 ← r – q
3      for i ← 1 to n1
4          do L[i] ← A[p + i – 1]
5      for j ← 1 to n2
6          do R[j] ← A[q + j]
7      L[n1+1] ← ∞
8      R[n2+1] ← ∞
9      i ← 1
10     j ← 1
11     for k ← p to r
12         do if L[i] ≤ R[j]
13             then A[k] ← L[i]
14                  i ← i + 1
15             else A[k] ← R[j]
16                  j ← j + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Analysis of Mergesort

Running time $T(n)$ of Merge Sort:

Divide: computing the middle takes $\Theta(1)$

Conquer: solving 2 sub problems takes $2T(n/2)$

Combine: merging n elements takes $\Theta(n)$

Total:

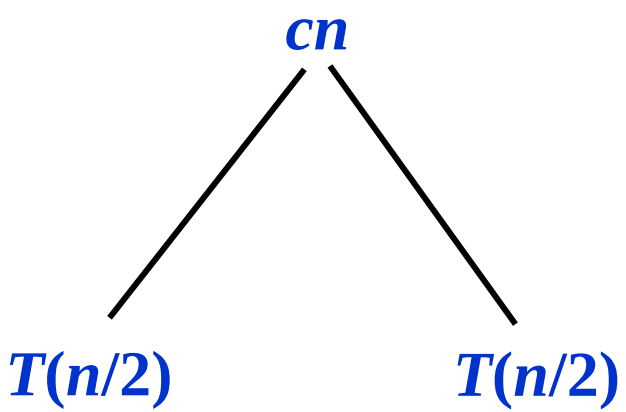
$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

$\Rightarrow T(n) = \Theta(n \lg n)$

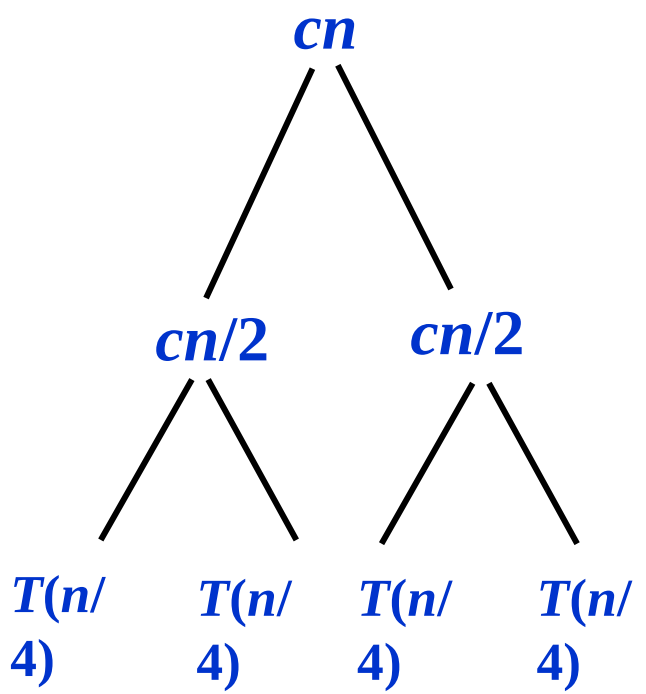
$\Rightarrow$Space requirement: $\Theta(n)$ (not in-place)

Recursion Tree for Merge Sort

For the original problem, we have a cost of *cn*, plus two sub-problems each of size ($n/2$) and running time $T(n/2)$.



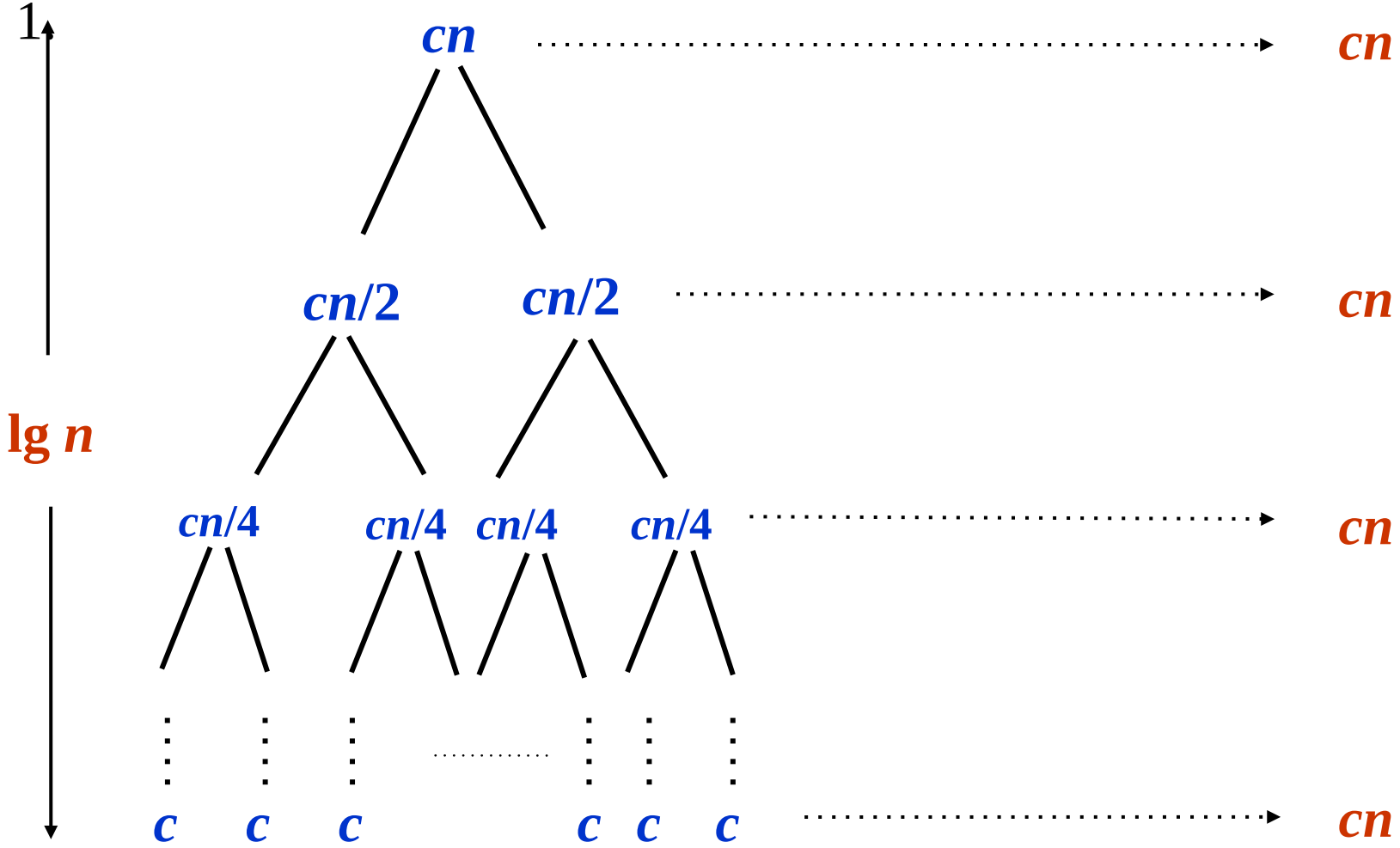
Each of the size $n/2$ problems has a cost of $cn/2$ plus two sub-problems, each costing $T(n/4)$.



Recursion Tree for Merge Sort

Ref Book: Thomas Cormen

Continue expanding until the problem size reduces to 1.



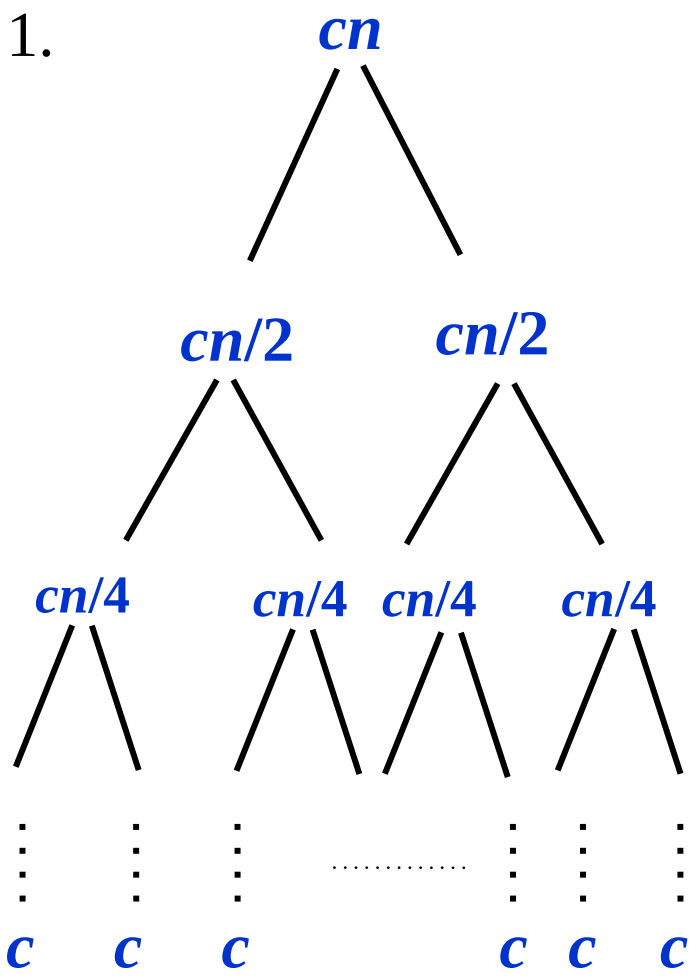
Total : $cn \lg n + cn$

Ref Book: Thomas Cormen

Recursion Tree for Merge Sort

Ref Book: Thomas Cormen

Continue expanding until the problem size reduces to 1.



- Each level has total cost ***cn***.
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves ⇒ *cost per level remains the same*.
- There are lg n + 1 levels, height is lg n. (Assuming n is a power of 2.)
  - Can be proved by induction.
- Total cost = sum of costs at each level = (lg n + 1)cn = cnlg$n$ + cn = $\Theta(n \lg n)$.

Ref Book: Thomas Cormen

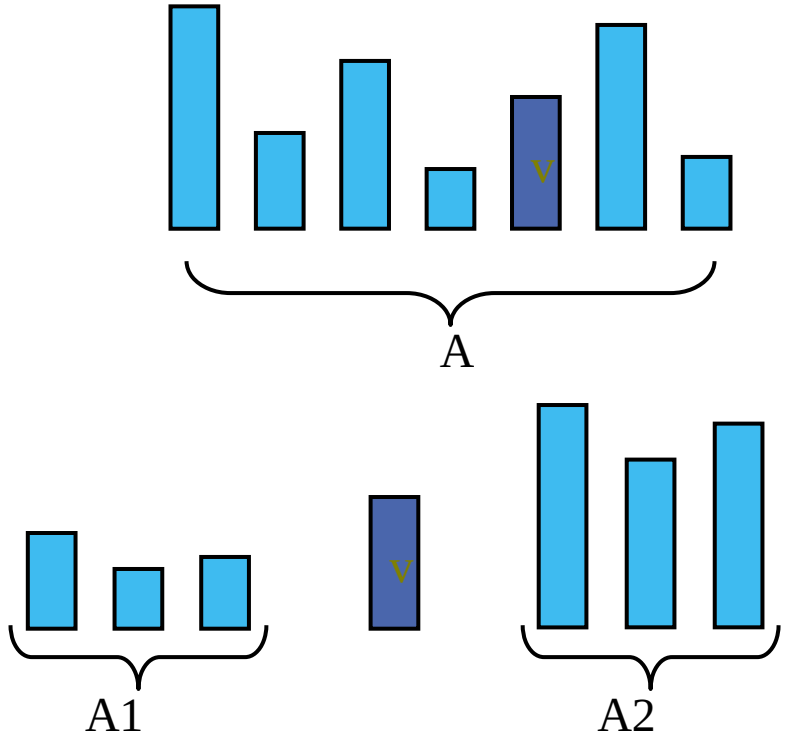
Quicksort

Ref Book: Thomas Cormen

- Divide :
  - Pick any element (***pivot***) v in array A[p…r]
  - Partition array A into two groups

    A1[p - - -q-1] , A2 [q+1----r] Compute the index q

  A1 element < A[q] < A2 element
- Conquer step: recursively sort A1 and A2
- Combine step: the sorted A1 (by the time returned from recursion), followed by A[q], followed by the sorted A2 (i.e., nothing extra needs to be done)



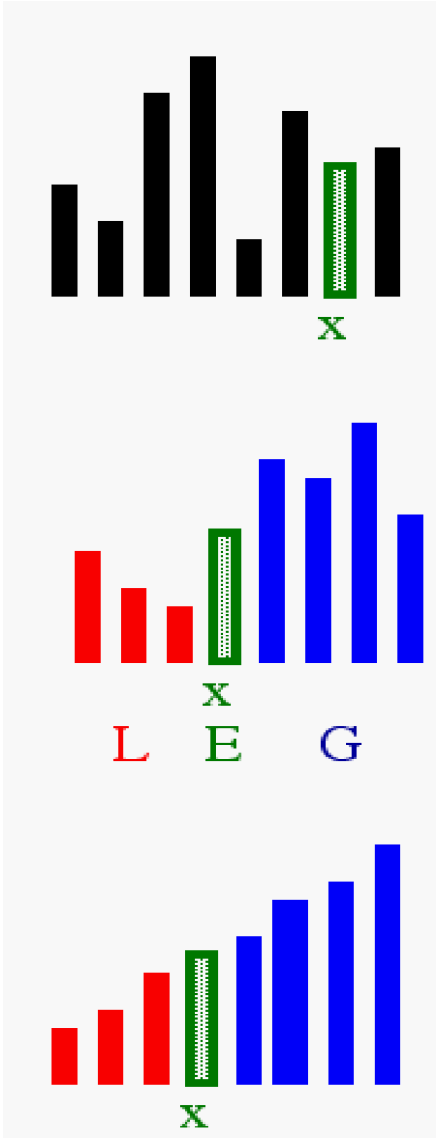
Idea of Quick Sort

Ref Book: Internet

1) **Select**: pick an element

2) **Divide**: rearrange elements so that x goes to its final position E

3) **Recurse and Conquer**: recursively sort



Ref Book: Thomas Cormen

Quicksort Pseudo-code

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
QuickSort (A, p, r)   // sort A[p..r] by divide & conquer
1    if p < r
2        then q = Partition (A, p, r)
3             QuickSort (A, p, q-1)
4             QuickSort (A, q+1, r)
```

Initial Call: QuickSort(*A*, 1, *n*)

Procedure Partitioning the array

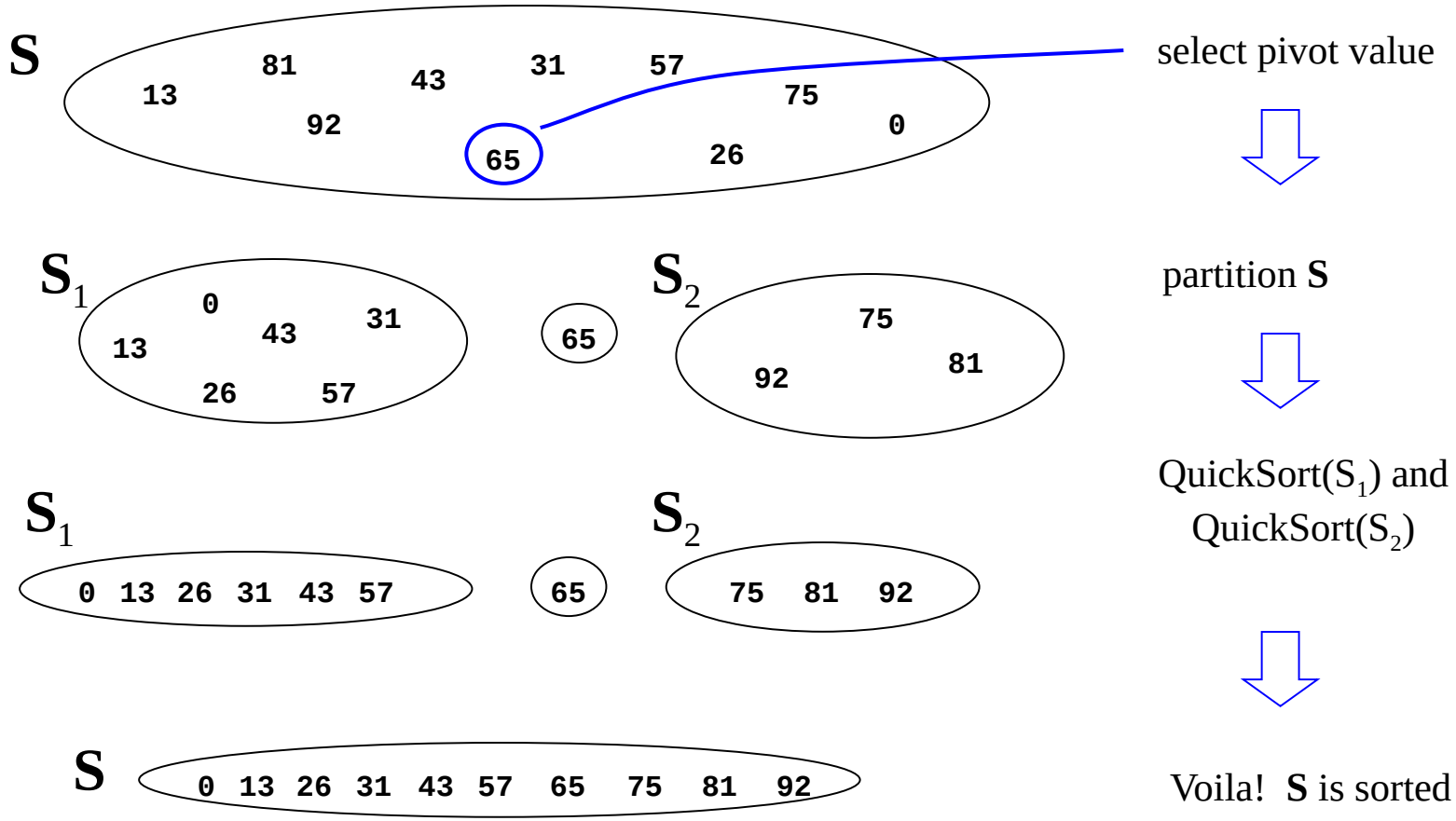
Ref Book: Thomas Cormen

```
Partition(A, p, r)
1  x = A[r]
2  i = p - 1
3      for j ← p to r-1
4          if A[j] ≤ x
5              i= i +1
6          exchange A[i] with A[j]
7      exchange A[i] with A[j]
8      Return i + 1
```

Input: Array containing sorted sub-arrays $A[p..q]$ and $A[q+1..r]$.

Output: sorted sub-array in $A[p..r]$.

The steps of QuickSort



Quicksort Analysis

Assumptions:

- A random pivot (no median-of-three partitioning)
- No cutoff for small arrays

Running time

- pivot selection: constant time, i.e. O(1)
- partitioning: linear time, i.e. O(N)
- running time of the two recursive calls

T(N)=T(i)+T(N-i-1)+cN where c is a constant

- i: number of elements in S1

Worst-Case Analysis

worst case Partition?

- The pivot is the smallest element, all the time
- Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Best-case Analysis

best case Partitioning?

- Partition is perfectly balanced.
- Pivot is always in the middle (median of the array)

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c \log N \\
T(N) &= cN \log N + N = O(N \log N)
\end{aligned}
$$

Average-Case Analysis

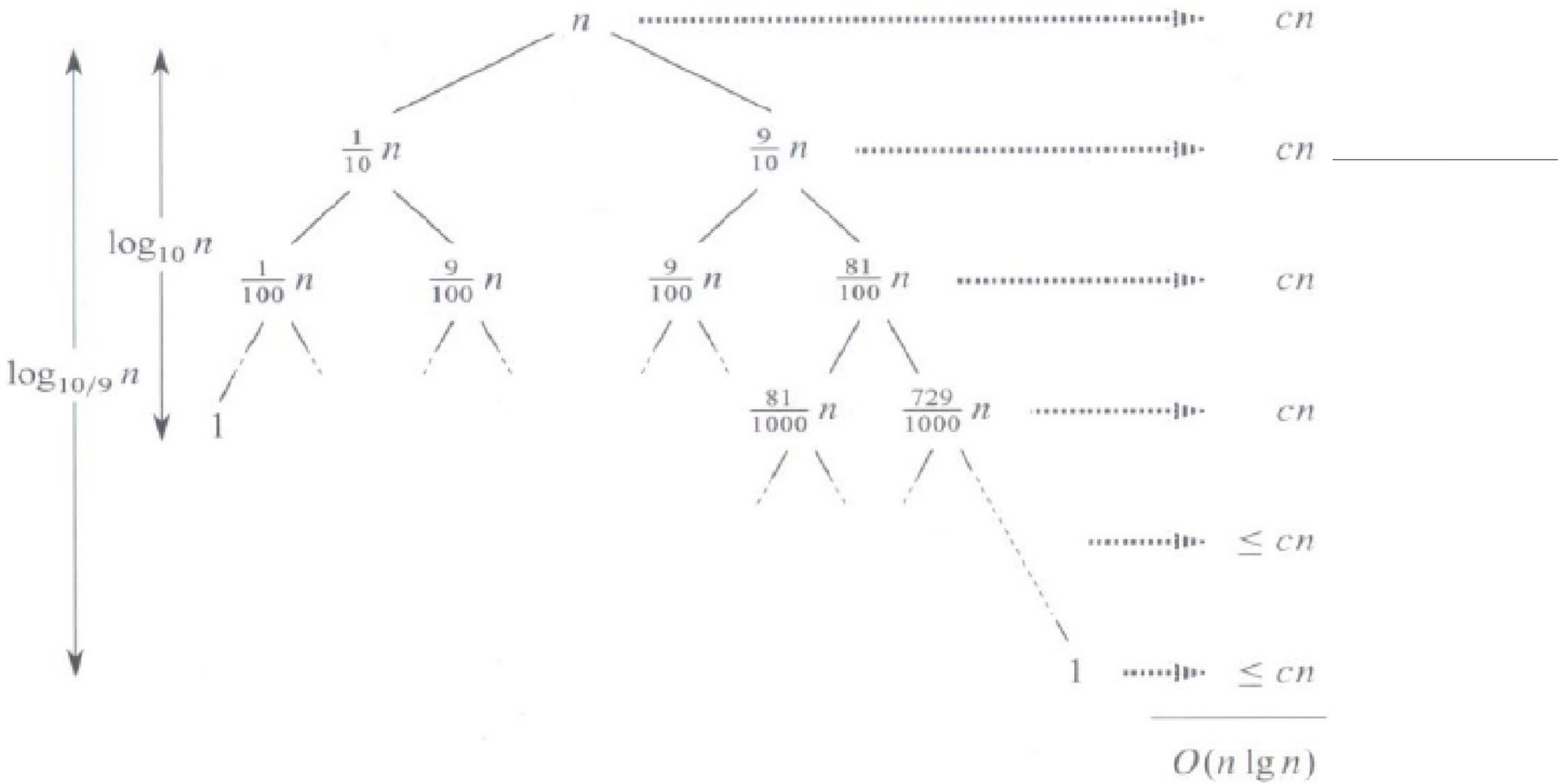
Intuition for Average Case : We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set.

Following is recurrence for this case.

$T(n) = T(n/9) + T(9n/10) + \Theta(n)$

Solution of above recurrence is also O(nlogn)

REF BOOK: THOMAS CORMEN



REF BOOK: THOMAS CORMEN

Summary Analysis of Quicksort

Best case: split in the middle — $\Theta(n \log n)$

Worst case: sorted array! — $\Theta(n^2)$

Average case: random arrays — $\Theta(n \log n)$

Improvements:

- better pivot selection: median of three partitioning
- switch to insertion sort on small subfiles

Multiplication of Large Integers

Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{r}
a_1\ a_2 \ldots\ a_n \\
\underline{b_1\ b_2 \ldots\ b_n} \\
(d_{10})\, d_{11} d_{12} \ldots d_{1n} \\
(d_{20})\, d_{21} d_{22} \ldots d_{2n} \\
\ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\
\hline
(d_{n0})\, d_{n1} d_{n2} \ldots d_{nn}
\end{array}
$$

Efficiency: $\Theta(n^2)$ single-digit multiplications

First Divide-and-Conquer Algorithm

A small example: A * B where A = 2135 and B = 4014

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are *n*-digit, A_1, A_2, B_1, B_2 are *n*/2-digit numbers),

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications M(*n*):

$M(n) = 4M(n/2), \quad M(1) = 1$

Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

The idea is to decrease the number of multiplications from 4 to 3:

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$$

I.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$
which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications M(*n*):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

What if we count both multiplications and additions?

Example of Large-Integer Multiplication

2135 * 4014

= (21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + 35*14

where c1 = (21+35)*(40+14) - 21*40 - 35*14, and

21*40 = (2*10 + 1) * (4*10 + 0)

= (2*4)*10^2 + c2*10 + 1*0

where c2 = (2+1)*(4+0) - 2*4 - 1*0, etc.

This process requires 9 digit multiplications as opposed to 16.

Introduction to Brute force method

Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. Generally it involved iterating through all possible solutions until a valid one is found.

Exhaustive Search

Exhaustive search refers to brute force search for combinatorial problems. We essentially generate each element of the problem domain and see if it satisfies the solution.

We do the following:

• Construct a way of listing all potential solutions to the problem in a systematic manner

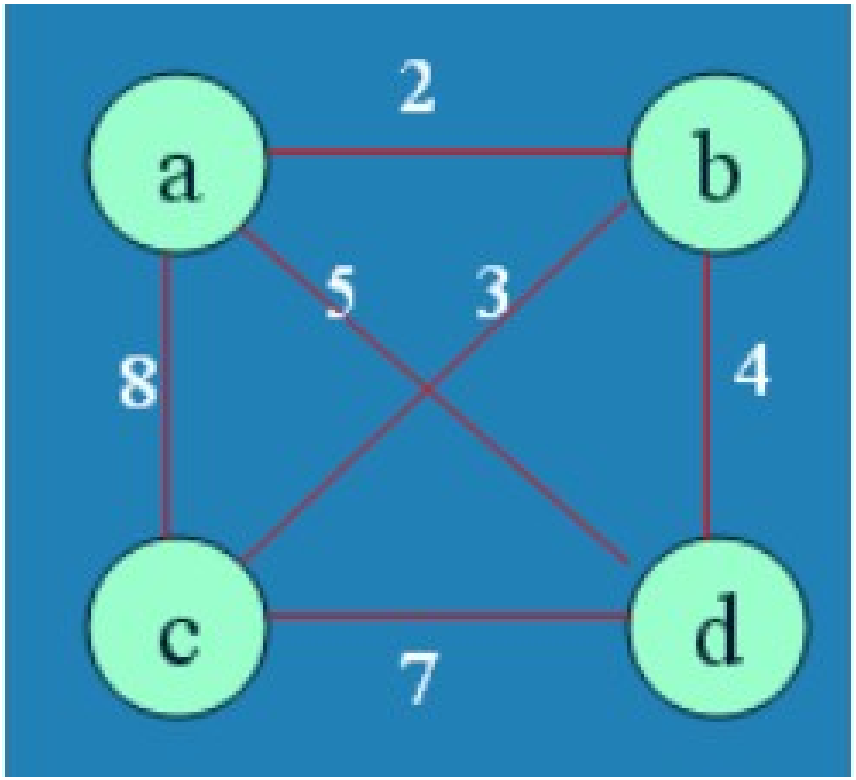
- all solutions are eventually listed
- no solution is repeated

• Evaluate solutions one by one, perhaps disqualifying infeasible ones and keeping track of the best one found so far

• When search ends, announce the winner

Exhaustive Search Example : Traveling salesman

Find shortest Hamiltonian circuit in a weighted connected graph.



Exhaustive Search Example : Traveling salesman

Tour	Cost
a→_x005F_x0001_b→_x005F_x0001_c→_x005F_x0001_d→_x005F_x0001_a	2+3+7+5 = 17
a_x005F_x0001_→b_x005F_x0001_→d→_x005F_x0001_c→_x005F_x0001_a	2+4+7+8 = 21
a_x005F_x0001_→c→_x005F_x0001_b→_x005F_x0001_d→_x005F_x0001_a	8+3+4+5 = 20
a_x005F_x0001_→c→_x005F_x0001_d→_x005F_x0001_b→_x005F_x0001_a	8+7+4+2 = 21
a_x005F_x0001_→d→_x005F_x0001_b→_x005F_x0001_c→_x005F_x0001_a	5+4+3+8 = 20
a_x005F_x0001_→d→_x005F_x0001_c→_x005F_x0001_b→_x005F_x0001_a	5+7+3+2 = 17

Exhaustive Search Example : Knapsack Problem

Given n items:

weights: w_1 w_2 ... w_n

values: v_1 v_2 ... v_n

A knapsack of capacity W

Find the most valuable subset of the items that fit into the knapsack (sum of Weights _x005F_x0002_<=W)

Exhaustive Search Example : Knapsack Problem

Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Exhaustive Search Example : Knapsack Problem

Subset	Total weight	Total value	Subset	Total weight	Total value
{1}	2	$20	{3,4}	15	$60
{2}	5	$30	{1,2,3}	17	not feasible
{3}	10	$50	{1,2,4}	12	$60
{4}	5	$10	{1,3,4}	17	not feasible
{1,2}	7	$50	{2,3,4}	20	not feasible
{1,3}	12	$70	{1,2,3,4}	22	not feasible
{1,4}	7	$30			
{2,3}	15	$80			
{2,4}	10	$40			

Exhaustive Search approach

Exhaustive search algorithms run in a realistic amount of time only on very small instances.

In many cases there are much better alternatives! In general though we end up with an approximation to the optimal solution instead of the guaranteed optimal solution.

- Euler circuits
- shortest paths
- minimum spanning tree
- various AI techniques

In some cases exhaustive search (or variation) is the only known solution.

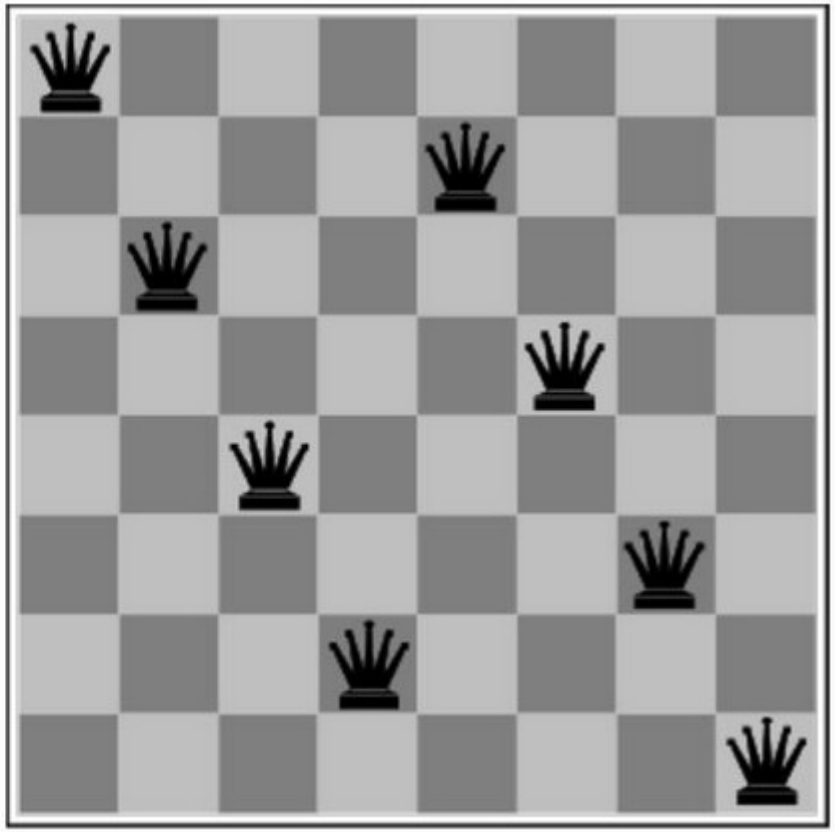
N-queen’s problem using brute force method

Basic idea: The basic idea of the brute force algorithm is to place the queens on all possible positions and check each time if the queens cannot capture each other. If not then it has found a solution. Because of the vast amount of possible positions (N^N for a table of size N while each row has 1 queen), this algorithm is not practical even for small table sizes (like N=12).

Advantages over other methods: Probably none. The brute force algorithm is only mentined to point out the superiority of the other algorithms, as a brute force approach is the last resort, when every other attempt failed.

8-queen’s problem using brute force method

Place eight queens on a chessboard such that no queen attacks any other!



8-queen’s problem using brute force method

A simple bruteforce solution would be to generate all possible chess boards with 8 queens. Accordingly, there would be N^2 positions to place the first queen, N^2 – 1 position to place the second queen and so on.

The total time complexity, in this case, would be O(N^(2N)), which is too high.

Brute force method strengths

- wide applicability
- simplicity
- yields reasonable algorithms for some important problems
  - searching
  - string matching
  - matrix multiplication
- yields standard algorithms for simple computational tasks
  - sum/product of n numbers
  - finding max/min in a list

Brute force method weaknesses

- rarely yields efficient algorithms
- some brute force algorithms unacceptably slow
- not as constructive/creative as some other design techniques

References

1. Thomas H Cormen and Charles E.L Leiserson, "Introduction to Algorithm" PHI Third Edition
2. Horowitz and Sahani, "Fundamentals of Computer Algorithms", 2ND Edition. University Press, ISBN: 978 81 7371 6126, 81 7371 61262.