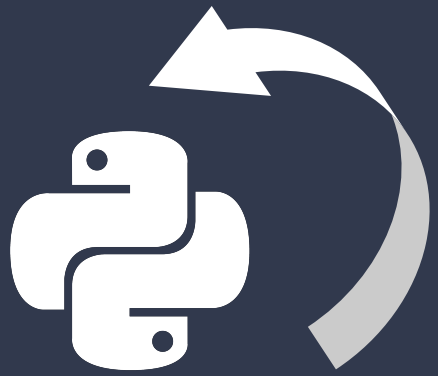
CSCA08 Final Review Seminar



Dec 7th, 2025 | 13:00 – 15:00

Course Reps:
Kevin Yu, Zohair Syed, Krisha Patel,
Nathanael Kevin Sitepu, Nihar Samant

PRESENTED BY

AMACSS
The Association of Mathematical and Computer Science Students

Content Overview

Agenda / Topics for Today

Agenda / Topics for Today:

- Lists
- Review of tuples/dictionaries
- Files
- Testing
- Sorting and Sorting Analysis
- Past Exams and Questions

PRESENTED BY

AMACSS

The Association of Mathematical and Computer Science Students

Part 1: Quick Review - Lists, Tuples, & Dictionaries

Lists

A data type that can store a **collection of elements** under one variable

```
# A list of integers
a = [1, 2, 3, 4]

# Elements of a list can be of different types
b = ["one", 2, "three", 4]

# An example of list-nesting
c = [[1], [1, 2], [1, 2, 3]]
```

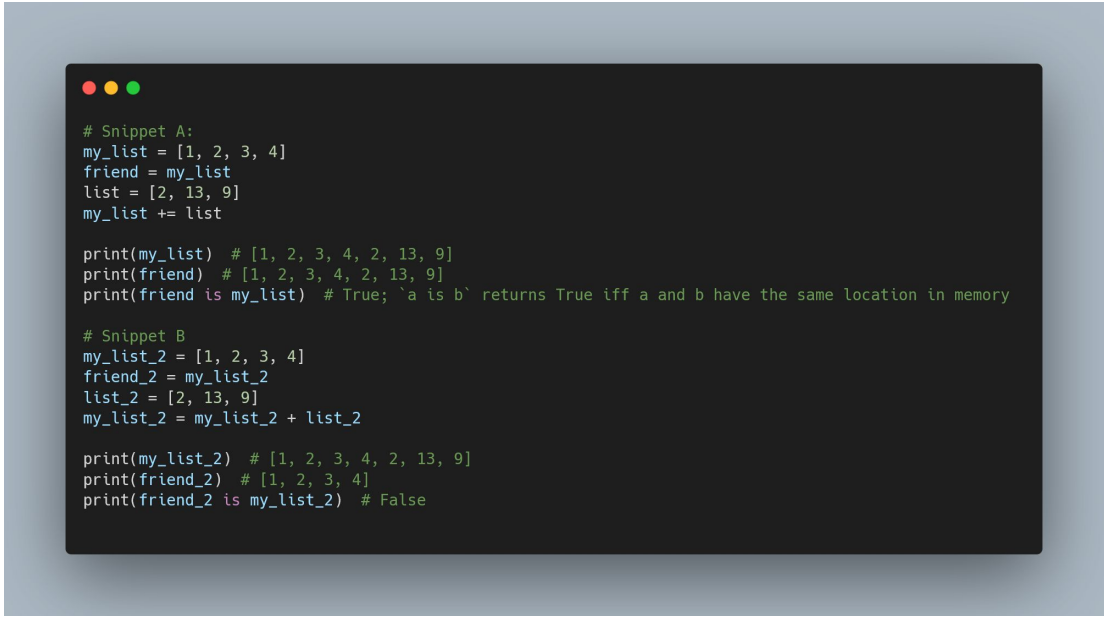
```
# Modifying a list:
a[1] = 99
print(a) # [1, 99, 3, 4]

# An example of aliasing
fake_b = b
fake_b[0] = "one million"
print(b) # ["one million", 2, "three", 4]

# Accessing elements in a nested list
print(c[0][0]) # 1
print(c[2][2]) # 3
print(c[0][3]) # IndexError; c[0] is a list with only one element
```

Lists - Continued

Some common stuff that trips a lot of people up:



```python
# Snippet A:
my_list = [1, 2, 3, 4]
friend = my_list
list = [2, 13, 9]
my_list += list

print(my_list)  # [1, 2, 3, 4, 2, 13, 9]
print(friend)  # [1, 2, 3, 4, 2, 13, 9]
print(friend is my_list)  # True; `a is b` returns True iff a and b have the same location in memory

# Snippet B
my_list_2 = [1, 2, 3, 4]
friend_2 = my_list_2
list_2 = [2, 13, 9]
my_list_2 = my_list_2 + list_2

print(my_list_2)  # [1, 2, 3, 4, 2, 13, 9]
print(friend_2)  # [1, 2, 3, 4]
print(friend_2 is my_list_2)  # False
```

Snippet A:

+= **modifies** the list. Since friend is an alias, we can see the change being reflected.

Snippet B:

my_list_2 + list_2 **creates a new list in memory**. Then, the assignment operator assigns this new list to my_list_2. friend_2 still points to the old object in memory.

Lists (Example From Last Slide)

```
# Snippet A:
my_list = [1, 2, 3, 4]
friend = my_list
list = [2, 13, 9]
my_list += list

print(my_list)  # [1, 2, 3, 4, 2, 13, 9]
print(friend)  # [1, 2, 3, 4, 2, 13, 9]
print(friend is my_list)  # True; `a is b` returns True iff a and b have the same location in memory

# Snippet B
my_list_2 = [1, 2, 3, 4]
friend_2 = my_list_2
list_2 = [2, 13, 9]
my_list_2 = my_list_2 + list_2

print(my_list_2)  # [1, 2, 3, 4, 2, 13, 9]
print(friend_2)  # [1, 2, 3, 4]
print(friend_2 is my_list_2)  # False
```

Tuples

An n-tuple is a grouped collection of n items, similar to a list, but:

- They are **immutable**
- Use `my_tuple = (1, 2, 3, 4)` syntax

Quick question: How do you sort the elements of a tuple?

A: Convert to a list first, then you can sort using the aforementioned list methods!

```
my_tuple = (4, 1, 3, 5, 2)




print(my_tuple)

# >>> (1, 2, 3, 4, 5)
```

Dictionaries

A dictionary is a mapping of **key-value pairs**, with many helpful built-in methods

```
d = {
    'a': ["apple", "apricot", "avocado"],
    'b': 1,
    'c': (5, 3, 4, 1, 2),
    'd': 2,
}

print(d['a'])


print(d.keys())
```

```
d["c"] = "New changed value"
print(d["c"])


d["e"] = "New added value"
print(d["e"])
```

Common Dictionary Techniques

- Iterate through values with .values()

```
groceries_to_price = {
    'apple': 0.5,
    'grapes': 2.0,
    'milk': 2.0,
    'bread': 2.5
}


for prices in groceries_to_price.values():
    print(prices)

# Output:
# 0.5
# 2.0
# 2.0
# 2.5
```

- Counting frequencies

```
groceries_to_price = {
    'apple': 0.5,
    'grapes': 2.0,
    'milk': 2.0,
    'bread': 2.5
}

price_freq = {}

for price in groceries_to_price.values():
    if price not in price_freq:
        price_freq[price] = 1
    else:
        price_freq[price] += 1
```

```
price_freq = {
    0.5: 1,
    2.5: 1,
    2.0: 2
}
```

Reminders

- Dictionaries are unordered collections. **Do not try to index them**
- '+=' and '+' methods aren't defined for Dictionaries(i.e you cannot add two dictionaries)
- There are only two methods defined on tuples:
  count() : Counts the number of occurrences of a particular element.
  index() : Looks for the first occurence of an element. Throws an error if the element is not in the tuple.
- A single element tuple is defined a little differently:
  >>> a = (1,)
  >>> b = (1)
  'a' is a tuple and 'b' is **not a tuple.** Then what is the type of 'b' here?

```python
data = [[('Rogers', 4), ('Bell', 3), ('Telus', 5)],
        [('Bell', 2), ('Rogers', 4), ('Virgin', 3)],
        [('Telus', 1), ('Fido', 2), ('Rogers', 3)],
        [('Public', 5), ('Telus', 4), ('Fido', 3)]]


def list_company_ratings(data: list) -> dict[str, list[int]]:
    """Given lists of tuples containing company names and their ratings, return a sorted dictionary
    mapping each company to a list of its ratings.

    >>> list_company_ratings(data)
    {'Rogers': [3, 4, 4], 'Bell': [2, 3], 'Telus': [1, 4, 5], 'Virgin': [3], 'Fido': [2, 3], 'Public': [5]}
    """

print(list_company_ratings(data))
```

Part 2: Files

File Operations

```
file1 = open("sample_txt.txt")
#Currently the file is open for reading
file1.close()
#The file is closed
```

```
with open ("sample_txt.txt") as file2:
    #Currently the file is open for reading
    print("Opened sample txt file")
#The file is closed
```

open(file_location , mode)

Both file_location and mode are of type string

file_location can be specified by : Absolute path or Relative path

The default value for mode is set to “r”

Which method would you prefer?

modes

r	read-only. Error if the file does not exist
r+	read and write. Error if the file does not exist
w	Write-only. New file created if the file does not exist
w+	read and write. New file created if the file does not exist
a	Append-only. New file created if the file does not exist
a+	read and append. New created file if the file does not exist

If the file does not exist the error raised is **FileNotFoundError**

If a new file has to be created, it is created in the same folder/directory as your python script

rb, rb+, wb, wb+, ab, ab+ perform the same tasks expect the file is opened for tasks in binary.

We will mostly work with r, w and a

Reading a file : Readline

Returns the *current line* of the file as a string and file pointer is updated.

If at EOF it returns an empty string and leaves the file pointer unaffected.

```
with open("sample_txt.txt") as file1:
    file1.readline()
    file1.readline()
    file1.readline()
    file1.readline()
    file1.readline()
    file1.readline()
```

```
Hello Everyone
Welcome to the review seminar
We are practicing files section
Good luck on the finals

```

Readline continued

The readline also *reads the '\n' character at the end of each line*.
The print line by default also adds a '\n' to the string to be printed in the console.

```
with open ("sample_txt.txt", "r") as file2:
    #Currently the file is open for reading
    print(file2.readline())
    print(file2.readline())
    print(file2.readline())
    print(file2.readline())
    print(len(file2.readline()))
```

```
Hello Everyone

Welcome to the review seminar

We are practicing files section

Good luck on the finals

0
```

Fixes:

```
print(file2.readline(), end = "")
print(file2.readline()[:-1])
```

Which is better and why?

Other read operations

file_object.read()
Returns the entire file content *starting from the current line* as a single string.

```
with open("sample_txt.txt", "r") as sample_file:
    sample_file.readline()
    print(sample_file.read())
```

```
Welcome to the review seminar
We are practicing files section
Good luck on the finals
```

file_object.readlines()
Returns a list of strings which are all the lines *starting from the current line.*

```
with open("sample_txt.txt", "r") as sample_file:
    sample_file.readline()
    print(sample_file.readlines())
```

```
['Welcome to the review seminar\n', 'We are
 practicing files section\n', 'Good luck on the
 finals\n']
```

File Reading

Another way of reading a file :

```
with open ("sample_txt.txt", "r") as file2:
    #Currently the file is open for reading
    for line in file2:
        print(line, end="")
```

```
Hello Everyone
Welcome to the review seminar
We are practicing files section
Good luck on the finals
```

Fun fact

readline, readlines and read
takes an optional int argument ‘size’ to specify how many character/bytes you want to read. By default it is set to ‘-1’

```
with open ("sample_txt.txt", "r") as file2:
    #Currently the file is open for reading
    print(file2.readline(11))
    print(file2.readline())
```

Output

```
Hello Every
one
```



Careful about how the file pointer is affected in your read operations

Writing to a file

file_obj.write("str") : Writes a string to the file opened by file_obj

```
with open ("sample_txt.txt", "w") as file2:
    #Currently the file is open for reading
    file2.write("--------------\n")
    file2.write("Happy Holidays\n")
```

```
with open ("sample_txt.txt", "a") as file2:
    #Currently the file is open for reading
    file2.write("--------------\n")
    file2.write("Happy Holidays\n")
```

```
--------------
Happy Holidays
```

You need to manually add the '\n' to go to the next line

```
Hello Everyone
Welcome to the review seminar
We are practicing files section
Good luck on the finals
--------------
Happy Holidays
```

Writing to a file

file_obj.writelines(iterable) : Writes multiple lines by iterating over the iterable object specified.

```
with open ("sample_txt.txt", "w") as file2:
    #Currently the file is open for reading
    lst = ["**********\n" , "this line was in a list\n"]
    tup = ("----------\n", "this line was in a tuple\n")
    diction = {"dict_key_1\n" : 1, "dict_key_2\n" : 2, "dict_key_3\n" : 3}
    file2.writelines(lst)
    file2.writelines(tup)
    file2.write("__________________\n")
    file2.writelines(diction)
```

```
**********
this line was in a list
----------
this line was in a tuple

__________________
dict_key_1
dict_key_2
dict_key_3
```

Practice Problem : Mystery Function

```python
def mystery_senior(mystery_string : str) -> None:
    """
    Precondition: mystery_string does not break this function
    """
    j = 1
    with open(mystery_string, "r") as mystery_file:
        for line in mystery_file:
            if mystery_junior(j):
                print(line)
            j += 1
```

Prints all lines in the file ‘mystery_string’ for which the line number is not divisible by any integer smaller than ‘num’ and greater than 2.

```python
def mystery_junior(num : int) -> bool:
    """
    Precondition: num > 0
    """
    i = 2
    while i < num:
        if num % i == 0:
            return False
        i += 1
    return True
```

Returns True iff ‘num’ is not divisible by any integer smaller than ‘num’ and greater than 2.

Returns True iff ‘num’ is a prime number or 1

Part 3: Algorithm Analysis

Useful Prereq: Summation Notation

You'll need it in the future if you haven't learnt it already … Not really needed for this course but makes things A LOT easier

$$a_1 + a_2 + \cdots + a_n = \sum_{i=1}^{n} a_i$$

Start at i = 1, stop **exactly** at i = n

Useful Prereq: Arithmetic Series

- **Arithmetic sequence:** a sequence of numbers where the difference between each term is constant
- **Arithmetic series:** the total sum of all numbers in an arithmetic sequence
- Very useful for counting iterations for programs
- We give you the following formulas

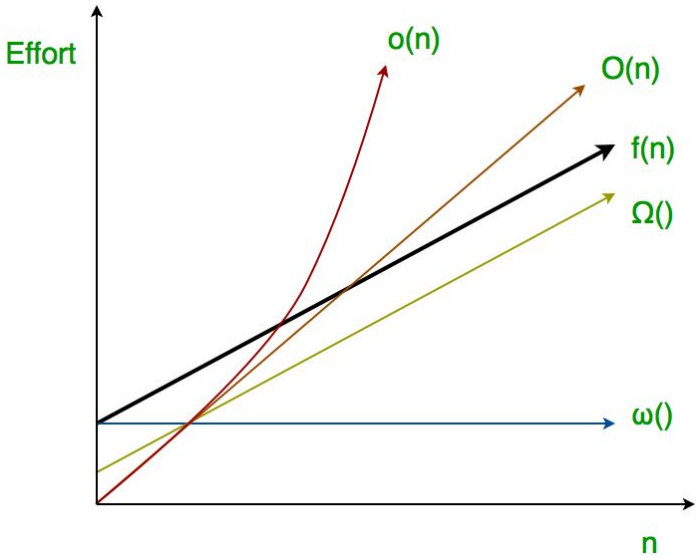
$$1 + 2 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\underbrace{c + \cdots + c}_{n \text{ times}} = \sum_{i=1}^{n} c = c \cdot n$$

Analyzing Algorithms

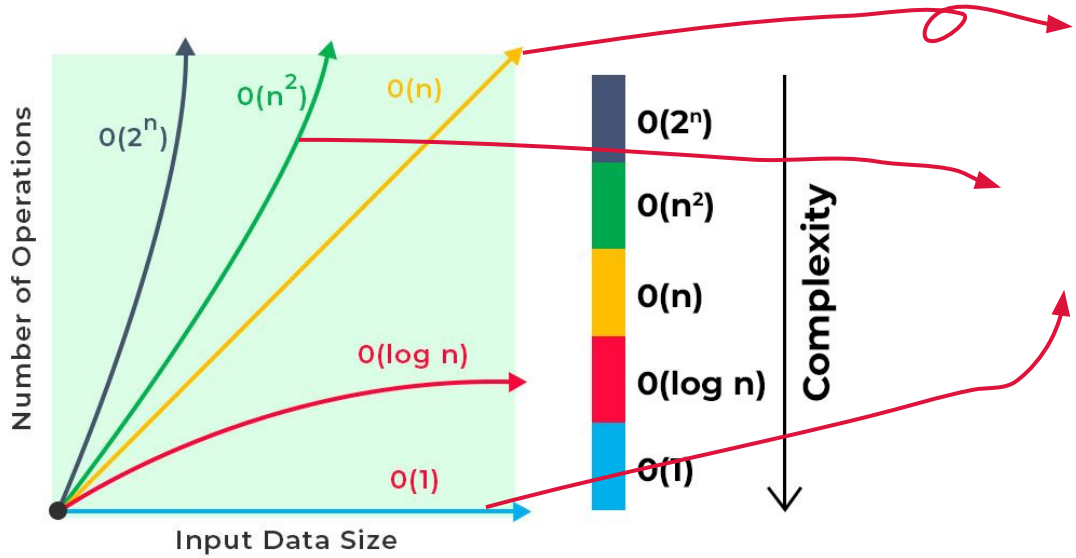
When analyzing algorithms, you were taught to look at the ***runtime complexity*** of some functions.

- How do we measure how ***efficient our program is***?
  - We count the # of ***operations or steps*** for a given **input size**
- What are our ***operations or steps***?
  - Depends on the context
  - Is specified during the exam
  - I.e. *#of arithmetic operations, #of iterations of a loop*
- What is the ***input size?***
  - Can be the size of a list, string, tuple, etc.



Common types of ‘growth’

NOTE: YOU ARE NOT REQUIRED TO KNOW ALL OF THESE! WE WILL POINT OUT THE ONES YOU SHOULD KNOW!



Linear O(n): The code ‘grows’ directly with input size.

Quadratic O(n^2): The code ‘grows’ with the square of the input.

Constant O(1): When the code doesn’t depend on input size.

- Is ‘constant’ with respect to the size.

Try it out...

How many iterations of each loop is roughly preformed?

```
def func1(n):
    for i in range(n):
        for j in range(4):
            print(i + j)
```

(A) 4n (B) n (C) 4 + n (D) none of above

```
def func2(n):
    i = 0
    while i < n:
        i += 3
```

(A) n/3 (B) 3n (C) log(n) (D) none of above

Some terms to know...

On your test, exam or worksheets, some common terms will pop up that you should know...

- **BEST RUNTIME CASE:** The case with the ‘least’ #of steps.
- **WORST RUNTIME CASE:** The case with the ‘greatest #of steps.

Try it out...

What is the best and worst case scenario of both functions, in terms of #conditional statements checked? [A IS NON EMPTY, len(A) >= 1]

```python
def first_local_min(A: list[int]) -> int:
    """
    Returns the index of the first element which is smaller than both neighbors.
    If no local minimum exists, returns -1.
    """
    n = len(A)
    if n == 0:
        return -1
    if n == 1 or A[0] < A[1]:
        return 0
    for i in range(1, n-1):
        if A[i] < A[i-1] and A[i] < A[i+1]:
            return i
    if A[n-1] < A[n-2]:
        return n-1
    return -1
```

Try it out...

What is the best and worst case scenario of both functions, in terms of #conditional statements checked? [A IS NON EMPTY, len(A) >= 1]

```
n = len(A)
if n == 0:
    return -1
if n == 1 or A[0] < A[1]:
    return 0
for i in range(1, n-1):
    if A[i] < A[i-1] and A[i] < A[i+1]:
        return i
if A[n-1] < A[n-2]:
    return n-1
return -1
```

We consider in both cases, array of the same size, n.

BREAK DOWN THE ALGORITHM...

- If the length of the array is zero, we return -1
  - ***One check***
- If the zeroth element in the array is less than its neighbour, we return ***zero*** **OR** the length of the array is 1
  - ***One to two checks***
- Else, we iterate and find the neighbour
  - ***# checks varies per amount of iterations done***

Try it out...

What is the best and worst case scenario of both functions, in terms of #conditional statements checked? [SUPPOSE A IS NOT EMPTY]

```
n = len(A)
if n == 0:
    return -1
if n == 1 or A[0] < A[1]:
    return 0
for i in range(1, n-1):
    if A[i] < A[i-1] and A[i] < A[i+1]:
        return i
if A[n-1] < A[n-2]:
    return n-1
return -1
```

We consider in both cases, array of the same size, n.

- BEST CASE...
  - If n == 1
    - We return early, giving **2 checks**
  - If n > 1...
    - If we let A[0] < A[1], we return early, giving **3 checks**

Try it out...

What is the best and worst case scenario of both functions, in terms of #conditional statements checked? [SUPPOSE A IS NOT EMPTY]

```
n = len(A)
if n == 0:
    return -1
if n == 1 or A[0] < A[1]:
    return 0
for i in range(1, n-1):
    if A[i] < A[i-1] and A[i] < A[i+1]:
        return i
if A[n-1] < A[n-2]:
    return n-1
return -1
```

We consider in both cases, array of the same size, n.

- WORST CASE... (consider n > 1)
  - We let **A = [100, 99 .... 1]**
    - Let A be a strictly decreasing sequence of numbers
  - First two if statements are **FALSE** = **3 checks**
  - We iterate through the loop **FULLY**
    - **A total of [(n-1) - 1] = n - 2 times**
    - **We have two checks in each loop, giving 2(n-2) checks**
  - Our final if statement runs = **1 CHECK**

TOTAL # CHECKS = 3 + 2(n-2) + 1 = 2n

Linear Search

```python
def linear_search(arr: list[int], num: int) -> bool:
    for c in arr:
        if c == num:
            return True
    return False
```

Linear search involves iterating through the entire array, ***arr*** until you find the element you want, ***num***.
In terms of the # of iterations...

- **BEST CASE:** When arr[0] == num, or arr is empty!
- **WORST CASE:** When the last element in arr is num, or num is **NOT** in arr.

Binary Search

```python
def binary_search(L: list[int], v: int):
    b = 0
    e = len(L) - 1

    while b <= e:
        m = (b + e) // 2
        if L[m] < v:
            b = m + 1
        else:
            e = m - 1

    if b == len(L) or L[b] != v:
        return -1
    else:
        return b
```

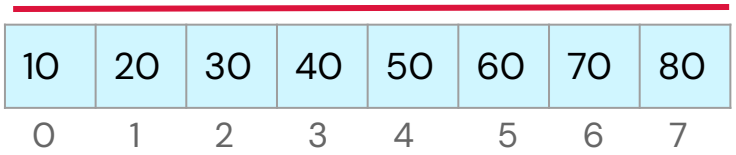
Binary search also searches for the element you want, but differently. **REQUIREMENT: LIST MUST BE SORTED!!**

- We initialize b and e to be the ***starting and ending indexes of*** our list
  - This is our search interval!
- Compute the middle index of the interval **m**
  - If L[m] < v...
    - V must be to the right as the array is sorted
  - Else...
    - V could be at m, or to the left (so we move e down)
- At the end, b is the smallest index which v will be found, or is ***invalid*** (out of bounds OR not v)

This version searches for the **FIRST OCCURENCE** of v.

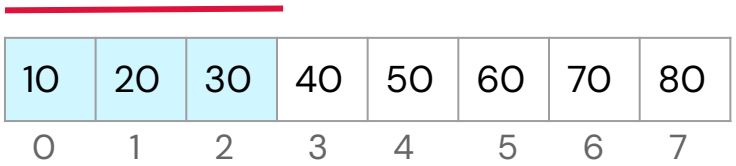
Binary Search

I.e. Searching for v = 30



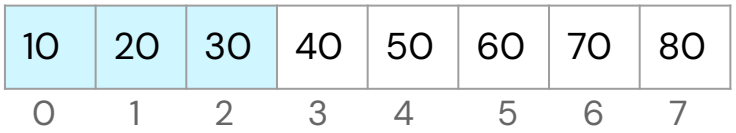
m = (0+7)//2 = 3

L[m] > v so e = m−1



m = (0+2)//2 = 1

L[m] < v so b = m+1



m = (2+2)//2 = 2

L[m] == v so e = m−1

We break out of the loop, and return b as L[b] == 30!

Selection Sort (REFRESHER)

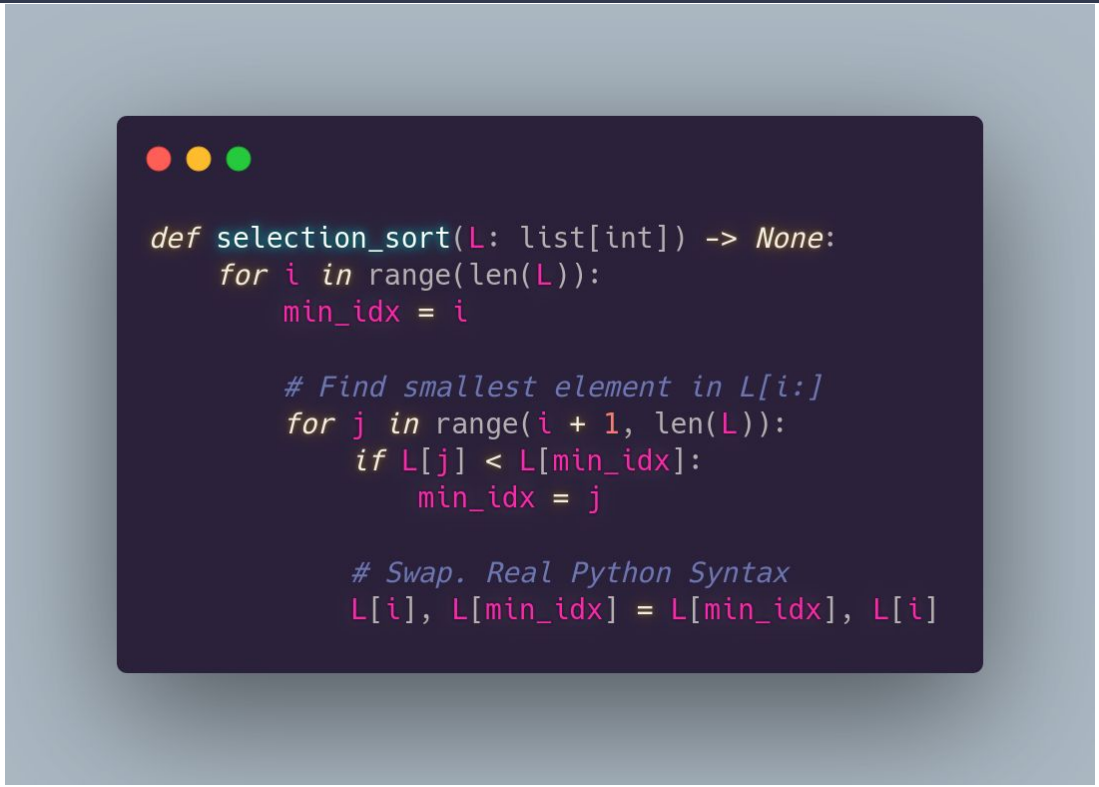
Idea:

- Divide list into sorted and unsorted portions – *How?*
- Loop over each item in the list:
  - For the item at index i, find `min(L[i:])` – the smallest item of `L[i:]`
  - Swap `min(L[i:])` and `L[i]`
  - After each iteration, `L[:i + 1]` will be unsorted and `L[i+1:]` will be sorted

Please don't use the min function on the exam :)

Update after seminar: fixed unsorted vs. sorted portion

Selection Sort – Python Code



```python
def selection_sort(L: list[int]) -> None:
    for i in range(len(L)):
        min_idx = i

        # Find smallest element in L[i:]
        for j in range(i + 1, len(L)):
            if L[j] < L[min_idx]:
                min_idx = j

            # Swap. Real Python Syntax
            L[i], L[min_idx] = L[min_idx], L[i]
```

Selection Sort – Analysis

Suppose that len(L) = n. We would like to count the number iterations.

Observe that there are no early exits in the algorithm – **Best Case, Worst Case, and Average Case are all the same.**

Outer loop: one iteration for each element => **runs n times**

Inner loop: On the ^{i}th iteration of the outer loop, the inner loop runs n−1−(i+1)+1 = **n−i−1 times**. *How? Calculate: "end index" − "start index" + 1*

The end index is **inclusive**. e.g., if the range ends at n, the end index will be n−1

Selection Sort – Analysis (CONT.)

Note: Indices of outer loop start at 0 and end at n-1

Total iterations:

$$(n-0-1)+(n-1-1)+\cdots+(n-(n-1)-1)=\sum_{i=0}^{n-1}(n-i-1)$$

$$=\sum_{m=1}^{n}(n-m),\ \text{let}\ \ m=i+1$$

$$=\sum_{m=1}^{n}n-\sum_{m=1}^{n}m$$

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

$$=\frac{n(n-1)}{2}$$

If m = i + 1, then

- i = 0 => m = 0 + 1 = 1
- i = n - 1 => m = (n - 1) + 1 = n

PLEASE ask if you are confused

Notice the n^2 term. Therefore, we have **quadratic** running time

Fix after seminar: note about indices only applies to outer loop

Past Exam Question (F23)

```
def zero_sum(lst: list[int]) -> bool:
    """Return True if and only if there are two non-zero elements in lst that sum to zero.
    >>> zero_sum([-1, 2, 0])
    False
    >>> zero_sum([1, -2, 0, 1, 2])
    True
    """
    for i in range(len(lst)):
        for j in range(i + 1, len(lst)):
            if lst[i] != 0 and lst[i] + lst[j] == 0:
                return True
    return False
```

Count comparisons: != and ==

We want: # of comparisons in worst case given len(lst) = n

ANSWER

Worst case: no early exit, all code runs. No two nonzero elements sum to zero, or only last two elements are nonzero and sum to zero.

Outer for loop: will run with indices from 0 to n−1 (i.e., n times)

Inner loop: for ^{i}th iteration of the outer loop, inner loop runs (n−1)−(i+1)+1=n−1 times. The inner loop has 2 comparisons, namely != and ==.

Number of comparisons:

$$2\sum_{i=0}^{n-1}(n-i-1) = 2\cdot\left(\frac{n(n-1)}{2}\right), \text{ see slide on selection sort}$$
$$= n(n-1)$$

Quadratic run time!

Fix after seminar: added additional scenario for worst case

Past Exam Question (AGAIN) (F23)

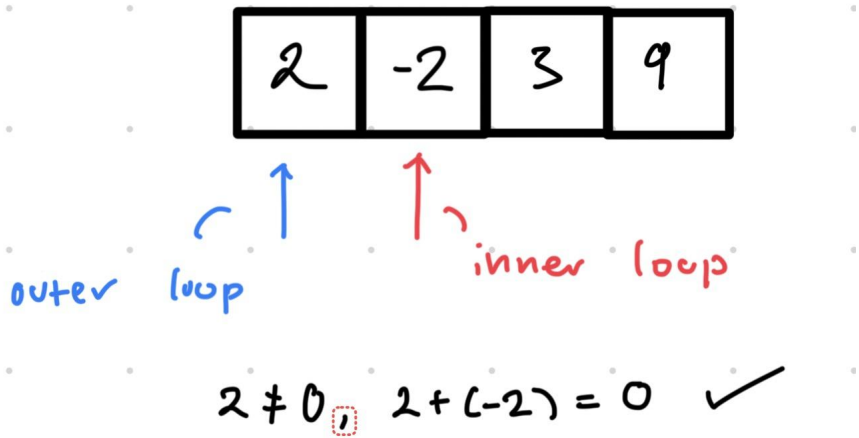
```python
def zero_sum(lst: list[int]) -> bool:
    """Return True if and only if there are two non-zero elements in lst that sum to zero.
    >>> zero_sum([-1, 2, 0])
    False
    >>> zero_sum([1, -2, 0, 1, 2])
    True
    """
    for i in range(len(lst)):
        for j in range(i + 1, len(lst)):
            if lst[i] != 0 and lst[i] + lst[j] == 0:
                return True
    return False
```

Count comparisons: != and ==

We want: # of comparisons in **best** case given len(lst) = n

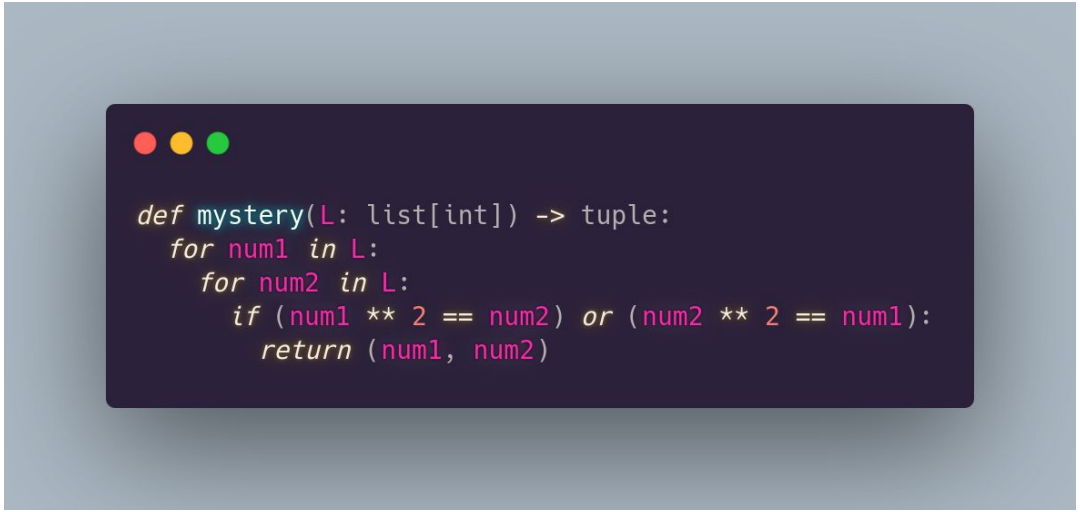
ANSWER

First two elements are nonzero and sum to zero. e.g.,



If-statement only needs to run once => 2 comparisons. Constant run time

Past Exam Question (W 2024)



```python
def mystery(L: list[int]) -> tuple:
  for num1 in L:
    for num2 in L:
      if (num1 ** 2 == num2) or (num2 ** 2 == num1):
        return (num1, num2)
```

Note: **1 if statement is 1 comparison**

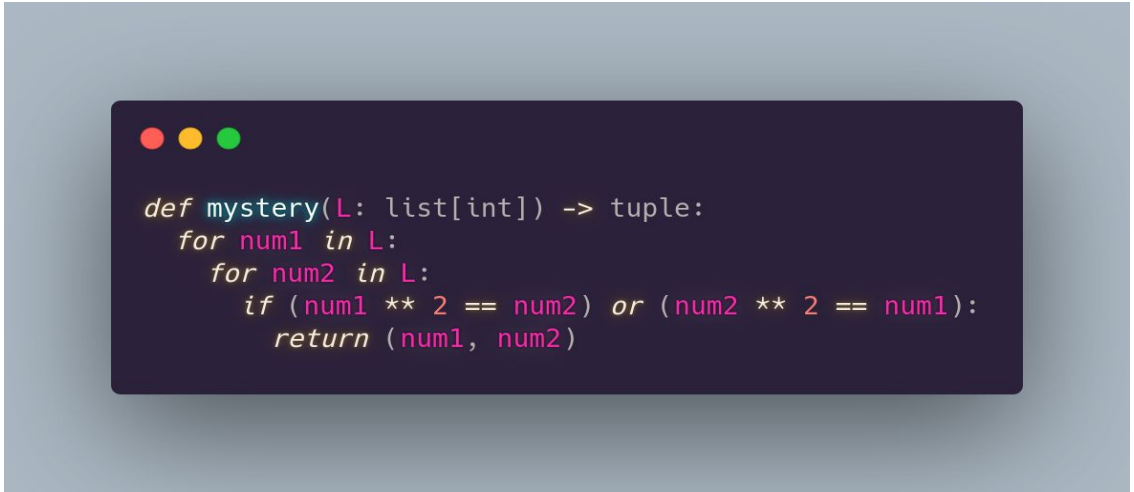
Fix after seminar: clarified what a comparison was

*How many **comparisons** would be executed on one call to this function with an argument of length* n *that produces the **worst-case**? Explain and show all your work. **Is the worst-case constant, linear, or quadratic?***

Answer

Worst case: We are assuming the worst case scenario, meaning that there is no early exit. Hence, every line of code is executed.

Worst case happens when none of the elements is another elements square. (or last and only last element is the square of itself e.g., 0 or 1). – Early exit will happen at end

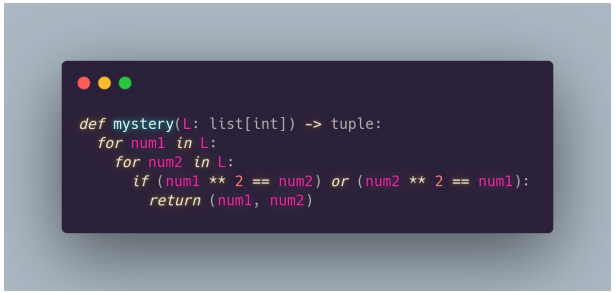


```
def mystery(L: list[int]) -> tuple:
  for num1 in L:
    for num2 in L:
      if (num1 ** 2 == num2) or (num2 ** 2 == num1):
        return (num1, num2)
```

Answer

Outer loop: one iteration for each element in the list – since `len(L) = n` (as given in the question) we have `n` iterations of the outer loop

Inner loop: similarly, there are `n` iterations of the inner loop.

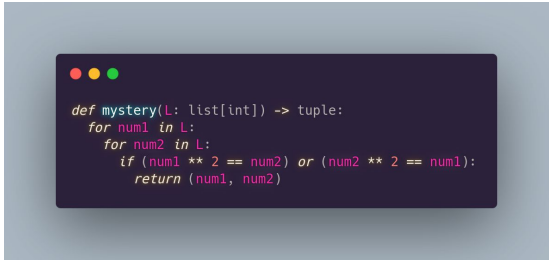


```
def mystery(L: list[int]) -> tuple:
  for num1 in L:
    for num2 in L:
      if (num1 ** 2 == num2) or (num2 ** 2 == num1):
        return (num1, num2)
```

Answer

Total Comparisons: There is **one** comparison for each iteration of the inner loop. The inner loop runs **n** times for each iteration of the outer loop. The outer loop runs **n** times. Therefore, there are $n \times n \times 1 = n^2$ comparisons in the worst case.

Linear, Constant, or Quadratic? Quadratic – note the n^2 term in the n.o. Comparisons



```
def mystery(L: list[int]) -> tuple:
  for num1 in L:
    for num2 in L:
      if (num1 ** 2 == num2) or (num2 ** 2 == num1):
        return (num1, num2)
```

Note: If we counted a == as a comparison, there would be $2n^2$ comparisons. False values do not cause short-circuit evaluations with the or operator

Fix after seminar: clarified n^2 vs $2n^2$ confusion. Removed redundant repetition

Part 4: Test Cases & Unit Testing

Intro to Testing

Testing is used to confirm the correctness of an application. There are many types of testing but we will focus on **unit tests**.

Unit Tests - used for testing in isolation

- Doctest
  - Mainly as an example
  - Around 3-4 examples

- Unit Testing (**Preferred**)
  - Better Maintainability
  - As many as needed/wanted
  - Can test more conditions

Unit Tests - UnitTest Module

UnitTest Summary

- Import the unittest module
- Test file in same folder (directory) as the function
- Each function tested is its own TestCase class, with each function being a different test case
- Modules vs Libraries vs Packages
- if __name__ == “__main__”

```python
# file: rotate_test.py

import unittest
from rotate import rotate_list

class TestRotateList(unittest.TestCase):
    """Unit tests for the rotate_list function."""

    def test_rotate_list_basic_rotation(self):
        "Test rotate_list given a valid list of length 5 and a non-negative rotation value."

        arg_list = [1, 2, 3, 4, 5]
        arg_rotation = 2

        actual = rotate_list(arg_list, arg_rotation)
        expected = [4, 5, 1, 2, 3]

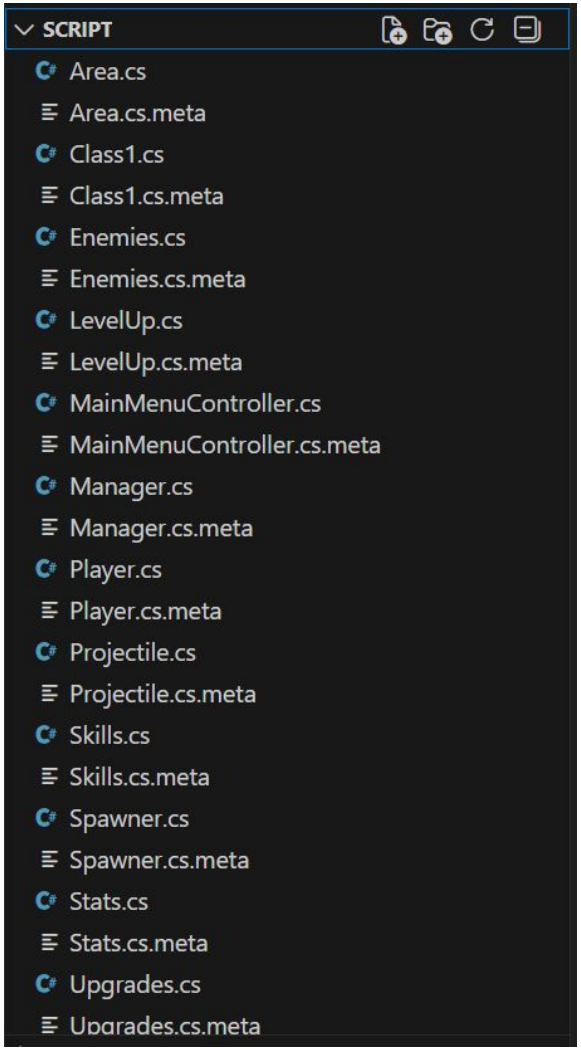
        msg = message(f"rotate_list({arg_list}, {arg_rotation})", str(expected), str(actual))
        self.assertEqual(rotate_list(arg_list, arg_rotation), expected, msg)


> def message(test_case: str, expected: str, actual: str) -> str:···

if __name__ == "__main__":
    unittest.main()
```

File Structure





Choosing Test Cases

- Know what the function does
  - Strings? Numbers? Floats? Lists?
  - Mutation?
  - Loops?
- Forget how the function works
- Test with valid inputs
- One test for each test case
- Write as many as you can think of

Input and Output of Functions

- Strings
  - Empty
  - Single/Many
  - Upper/lowercase/special characters
- Numbers
  - Zero
  - Positive/negative/even/odd
  - Decimals (if allowed)

Input and Output of Functions

- Mutation
  - Check if original list is mutated
- Lists/Dictionaries
  - Empty
  - Single/Many
  - Combination of all allowed inputs
- Loops
  - Off by one errors

```
def numZero(list):
    i = 1
    count = 0
    while i < len(list):
        if list[i] == 0:
            count += 1
        i += 1
    return count
```

Test Case Examples

def containsElement(lst: List[int], item: int) -> boolean:
>>> returns whether the item appears in lst

Description	lst	item	Return

Test Case Examples

def containsElement(lst: List[int], item: int) -> boolean:
>>> returns whether the item appears in lst

Description	lst	item	Return
Empty list, any integer	[]	5	False
Element appears as the only item in the list	[5]	5	True
List has one element, but item does not appear	[5]	4	False
List has multiple elements, and item appears	[1, 2, 3, 4, 5]	4	True
List has multiple elements, but item doesn't appear	[1, 2, 3, 4, 6, 7]	5	False

Test Case Examples

def mergeSort(lst: List[int]) -> None:
>>> sorts the list ‘lst’ according to merge sort

Description	lst	Return

Test Case Examples

def mergeSort(lst: List[int]) -> None:
>>> sorts the list 'lst' according to merge sort

Note: Merge sort divides the list by two and arranges each half

Description	lst	Modified lst
Empty list	[]	[]
Single element list	[2]	[2]
Many element list, all the same number, even length	[3, 3, 3, 3]	[3, 3, 3, 3]
Many element list, all negative, odd length	[-3, -5, -7, -1 -2]	[-7, -5, -3, -2, -1]
Many element list, mixed	[1, 4, 0, -5, 9, -2]	[-5. -2, 0, 1, 4, 9]
Many element list, already sorted	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]

Acknowledgments



We would like to thank, *Ben (AMACCS Director of CS)* and *Professor Purva Gawde* for providing us lecture notes to use as a basis for the review seminar.

Feedback Form!

Please fill out the feedback form below:

