

NAME

MARYAM ZAMEER

ROLL NO

14659

SUBMITTED TO

SIR JAMAL ABDUL AHAD

SUBJECT DATA STRUCTURE ALGORITHMS (DSA)

DATE

31/10/202

FOUNDATION

UNITNO1

THE ROLE OF ALGORITHMS IN COMPUTING

❖ PART 1

ALGORITHMS

EXERCISE

1.1-1

Question:

Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

Explanation:

➤ Sorting Example:

Suppose you have a list of exam scores from students, and you want to rank them from the highest to lowest score to award the top performers. Sorting is the operation that arranges these scores in a particular order. In Python, this can be done with the `sorted()` function, which returns a new sorted list.

Example in Python:

```
grades = [85, 92, 78, 90, 89]
```

```
sorted_grades = sorted(grades, reverse=True) # Sort in descending order
```

```
print("Sorted Grades:", sorted_grades)
```

This would output the grades in descending order, helping to identify the highest scorers.

➤ Shortest Distance Example:

Another common problem is finding the shortest distance between two points, such as two cities on a map. This is often solved with algorithms like Dijkstra's or A* for finding the shortest path in a graph. For instance, if a city map is represented as nodes (cities) connected by edges (roads with distances), Dijkstra's algorithm can efficiently find the shortest route between two cities.

Example in Python:

```
import heapq

def dijkstra(graph, start):

    distances = {node: float('infinity') for node in graph}

    distances[start] = 0

    priority_queue = [(0, start)]

    while priority_queue:

        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:

            continue

        for neighbor, weight in graph[current_node].items():

            distance = current_distance + weight

            if distance < distances[neighbor]:

                distances[neighbor] = distance

                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```

Example graph

```
graph = {
```

```
'A': {'B': 1, 'C': 4},  
  
'B': {'A': 1, 'C': 2, 'D': 5},  
  
'C': {'A': 4, 'B': 2, 'D': 1},  
  
'D': {'B': 5, 'C': 1}  
  
}  
  
print("Shortest distances from A:", dijkstra(graph, 'A'))
```

1.1-2

Question:

Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

Explanation:

When evaluating algorithms or solutions, speed (time complexity) is often a key metric, but it's not the only one. Here are some additional measures of efficiency:

➤ **Memory Usage:**

How much memory (RAM) the algorithm requires to process data. For instance, large datasets might not fit into memory, requiring memory-efficient approaches like generators or streaming data.

➤ **Scalability:**

How well the algorithm performs as the input size increases. A solution might work well on a small dataset but could slow down significantly with larger inputs if it isn't scalable.

➤ **Power Consumption:**

Important in embedded systems, mobile devices, and other battery-powered devices. An algorithm that consumes less power is preferable in such cases.

➤ **Network Bandwidth:**

If the application relies on data transferred over a network, the amount of data sent and received is crucial. For example, an online streaming service should minimize the amount of data sent to save bandwidth.

Each of these factors can influence the choice of algorithm or data structure in a real-world setting.

1.1-3

Question:

Select a data structure that you have seen, and discuss its strengths and limitations.

Explanation:

➤ **Data Structure:**

Dictionary (Hash Map)

In Python, dictionaries are implemented as hash maps, where each element is a key-value pair. Dictionaries are used for quick lookups, making them useful for scenarios where we need to associate unique keys with values.

➤ **Strengths:**

Fast Lookups:

Accessing, inserting, and deleting elements in a dictionary is on average $O(1)$.

Flexible:

They can store complex data structures (e.g., dictionaries within dictionaries).

Limitations:

Memory Intensive:

Dictionaries consume more memory due to hashing.

Unordered (before Python 3.7):

They did not maintain insertion order until Python 3.7, where dictionaries began preserving order by default.

Example in Python:

```
data = {"apple": 5, "banana": 2, "orange": 10}

print(data["apple"]) # Fast lookup for the value associated with "apple"
```

1.1-4

Question:

- How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

Explanation:

- **Similarities:**

Both problems deal with finding optimal paths in a graph, aiming to minimize a certain cost (such as distance or travel time). They often require similar graph traversal techniques.

- **Differences:**

Shortest Path Problem: This problem finds the minimum path between two specific points. For example, you may want the shortest route from one city to another.

- **Traveling Salesperson Problem (TSP):**

TSP requires visiting all nodes (or cities) exactly once and returning to the starting point. It's an NP-hard problem, meaning it's computationally difficult to find the optimal solution as the number of cities grows.

The shortest-path problem can often be solved with efficient algorithms like Dijkstra's, while TSP usually requires approximation algorithms (e.g., genetic algorithms) for larger datasets.

1.1-5

Question:

- Suggest a real-world problem in which only the best solution will do. Then come up with one in which "approximately" the best solution is good enough.

Explanation:

Exact Solution Needed: Surgery Scheduling in a Hospital. Here, each surgery has a strict set of requirements such as required resources (equipment, operating room) and time slots. Finding an optimal schedule that satisfies all constraints is critical, as conflicts or delays can have serious consequences.

Approximate Solution is Good Enough:

Restaurant Recommendations Based on Location. For a food delivery app, recommending nearby restaurants doesn't require an exact measure of distance. As long as the restaurants are close enough, users are likely to be satisfied with approximate suggestions. Approximation techniques (e.g., clustering) are often used in such recommendation systems to save time and resources.

1.1-6

Question:

- Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

Explanation:

Example:

- **Stock Market Analysis.**

Historical Data Analysis:

When analyzing past stock data to identify trends, you have all the input data available in advance. This allows for more complex, data-intensive analysis since there are no constraints on waiting for new data to arrive.

Real-Time Trading:

In contrast, when executing trades in real time, data (such as stock prices) arrives continuously. Traders need to make decisions based on the latest information, without knowing what future prices will be. Algorithms like moving averages or real-time data streaming are often used in these cases.

This type of problem, where data is only partially available and must be handled as it arrives, is common in fields like finance, weather forecasting, and IoT systems.

❖ **PART 2**

ALGORITHMS AS A TECHNOLOGY

EXERCISES

1.2-1

Question

- This question asks for an example of an application that requires algorithmic content at the application level and to discuss the function of the algorithms involved.

Explanation

An application that requires significant algorithmic content at the application level is a search engine, like Google.

What does a search engine do?

1. A search engine crawls the web to gather information from web pages.
2. It then indexes these pages for quick retrieval.
3. When a user enters a query, the search engine processes it and retrieves the most relevant pages.
4. Finally, it ranks these pages based on relevance to the user's query.

Key Algorithms Used

1. Page Ranking (e.g., PageRank):

Determines the importance of web pages based on the number and quality of links.

2. Inverted Indexing:

This data structure is used to map terms to their locations in documents, making it faster to retrieve relevant pages based on keywords.

3. Machine Learning:

Used to understand the user's intent and improve the relevance of search results.

Each of these algorithms helps improve the speed, relevance, and efficiency of the search engine.

1.2-2

Question

- This question involves comparing the performance of two sorting algorithms—Insertion Sort and Merge Sort—for a given input size .

Given Information

1. Insertion Sort takes $8n^2$ steps.
2. Merge Sort takes $64n \log n$ steps.

We need to find the largest value of n where Insertion Sort is faster than Merge Sort. This can be expressed mathematically as:

$$8n^2 < 64n \log n$$

Let's break down the steps to solve this problem.

Steps to Solve

1. Rewrite the Inequality:

Divide both sides by 8 to simplify:

$$n^2 < 8n \log n$$

2. Check for Values of :

- For small values of , manually checking might work, but for larger values, we need a systematic way.
- We incrementally increase to check at which point $8n^2$ becomes larger than $64n * \log(n)$.

3. Python Code Explanation:

```
import math
```

```
def find_n_for_insertion_vs_merge():
```

```
    n = 1
```

```
    while True:
```

```
        insertion_sort_steps = 8 * n ** 2
```

```
        merge_sort_steps = 64 * n * math.log2(n)
```

```
        if insertion_sort_steps >= merge_sort_steps:
```

```
            return n - 1 # Return the last value where insertion sort is faster
```

```
        n += 1
```



```
1 import math
2 def find_n_for_insertion_vs_merge():
3     n = 1
4     while True:
5         insertion_sort_steps = 8 * n ** 2
6         mer (variable) insertion_sort_steps: int
7         if insertion_sort_steps >= merge_sort_steps:
8             return n - 1 # Return the last value where insertion sort is faster
9         n += 1
```

Loop through values of n:

We start from $n = 1$ and incrementally increase.

Calculate Steps:

- $\text{insertion_sort_steps} = 8 * n ** 2$: Calculates the number of steps for Insertion Sort.
- $\text{merge_sort_steps} = 64 * n * \text{math.log2}(n)$: Calculates the number of steps for Merge Sort.

Check Condition:

When `insertion_sort_steps` is no longer less than `merge_sort_steps`, the loop breaks, and we return the last value of `n` where Insertion Sort is faster.

1.2-3**Question**

- This question involves finding the smallest `n` for which an algorithm with running time $100n^2$ runs faster than an algorithm with running time 2^n

Given Information

1. Algorithm 1 has a runtime of $100n^2$
2. Algorithm 2 has a runtime of 2^n

We need to find the smallest `n` for which:

$$100n^2 < 2^n$$

Steps to Solve**1. Understanding Growth Rates:**

$100n^2$ (quadratic) grows much slower than 2^n (exponential) as `n` increases.

At smaller `n`, $100n^2$ will be larger, but as `n` grows, 2^n will quickly overtake $100n^2$.

2. Python Code Explanation:

```
def find_n_for_quadratic_vs_exponential():
```

```
    n = 1
```

```
    while True:
```

```
        quadratic_time = 100 * n ** 2
```

```
exponential_time = 2 ** n
```

```
if quadratic_time < exponential_time:
```

```
    return n
```

```
n += 1
```

```
10 def find_n_for_quadratic_vs_exponential():
11     n = 1
12     while True:
13         quadratic_time = 100 * n ** 2
14         exponential_time = 2 ** n
15         if quadratic_time < exponential_time:
16             return n
17         n += 1
18
```

- **Loop through values of n:**

Starting from $n=1$, increment n by 1 each time.

- **Calculate Times:**
 1. $\text{quadratic_time} = 100 * n ** 2$: Calculates the time for the quadratic algorithm.
 2. $\text{exponential_time} = 2 ** n$: Calculates the time for the exponential algorithm.
- **Check Condition:**

When $100n^2$ becomes less than 2^n , we return n as the smallest value where this holds true.

Complete Code and Output

Combining both functions, the full code is:

```
import math
```

```
def find_n_for_insertion_vs_merge():
```

```
    n = 1
```

```
    while True:
```

```
        insertion_sort_steps = 8 * n ** 2
```

```

merge_sort_steps = 64 * n * math.log2(n)

if insertion_sort_steps >= merge_sort_steps:

    return n - 1 # Return the last value where insertion sort is faster

n += 1

def find_n_for_quadratic_vs_exponential():

    n = 1

    while True:

        quadratic_time = 100 * n ** 2

        exponential_time = 2 ** n

        if quadratic_time < exponential_time:

            return n

        n += 1

n_insertion_vs_merge = find_n_for_insertion_vs_merge()

n_quadratic_vs_exponential = find_n_for_quadratic_vs_exponential()

print("1.2-2: Insertion sort beats merge sort up to n =", n_insertion_vs_merge)

print("1.2-3: Smallest n where  $100n^2 < 2^n$  is n =", n_quadratic_vs_exponential)

```

```
import math.py •
import math.py > find_n_for_insertion_vs_merge
1  import math
2  def find_n_for_insertion_vs_merge():
3      n = 1
4      while True:
5          insertion_sort_steps = 8 * n ** 2
6          merge_sort_steps = 64 * n * math.log2(n)
7          if insertion_sort_steps >= merge_sort_steps:
8              return n - 1 # Return the last value where insertion sort is faster
9          n += 1
10 def find_n_for_quadratic_vs_exponential():
11     n = 1
12     while True:
13         quadratic_time = 100 * n ** 2
14         exponential_time = 2 ** n
15         if quadratic_time < exponential_time:
16             return n
17         n += 1
18
19 # Results
20 n_insertion_vs_merge = find_n_for_insertion_vs_merge()
21 n_quadratic_vs_exponential = find_n_for_quadratic_vs_exponential()
22
23 print("1.2-2: Insertion sort beats merge sort up to n =", n_insertion_vs_merge)
24 print("1.2-3: Smallest n where 100n^2 < 2^n is n =", n_quadratic_vs_exponential)
```

When you run this code, it will print the following:

1.2-2:

The largest n where Insertion Sort is faster than Merge Sort.

1.2-3:

The smallest n where the quadratic algorithm becomes faster than the exponential algorithm.

- This way, we programmatically find the values of n rather than relying on manual calculations. Let me know if you'd like to run the code to see the actual values of n .

UNIT NO 2

GETTING STARTED

❖ PART 1

INSERTATION SORT

EXERCISES

2.1-1

Question:

- Given a sequence [31, 41, 59, 26, 41, 58], use Insertion Sort to sort the array.

Explanation of Insertion Sort:

Insertion Sort works similarly to how you might sort cards in your hands. You pick each element one by one and place it in its correct position among the previously sorted elements.

1. Start with the second element (index 1) and compare it to the elements before it.
2. Shift all elements greater than the current element one position to the right.
3. Insert the current element into the correct position.
4. Repeat the process for each element in the array until the array is sorted.

Code:

```
def insertion_sort(arr):
```

```
    for i in range(1, len(arr)):
```

```
        key = arr[i] # The current element to be sorted
```

```
j = i - 1
```

```
# Shift elements of arr[0..i-1] that are greater than key to one position ahead
```

```
while j >= 0 and arr[j] > key:
```

```
    arr[j + 1] = arr[j]
```

```
    j -= 1
```

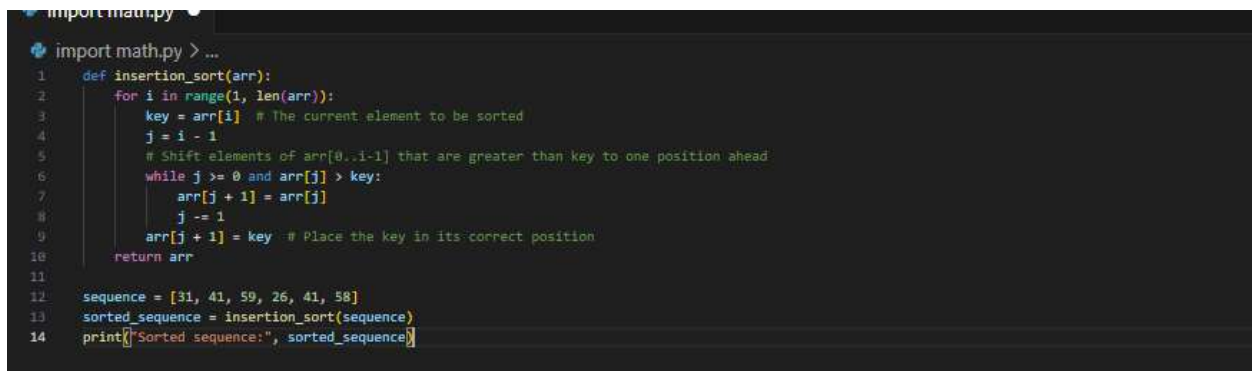
```
arr[j + 1] = key # Place the key in its correct position
```

```
return arr
```

```
sequence = [31, 41, 59, 26, 41, 58]
```

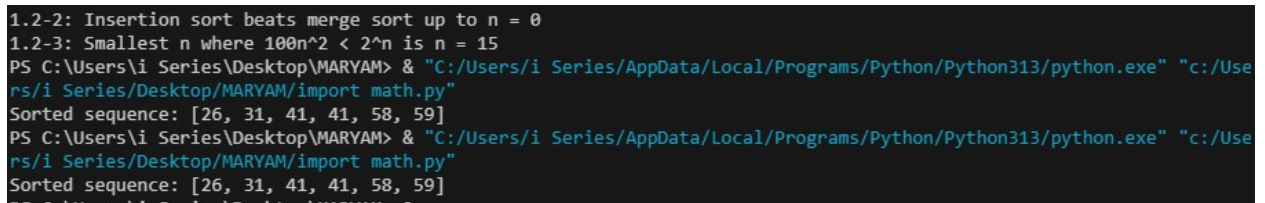
```
sorted_sequence = insertion_sort(sequence)
```

```
print("Sorted sequence:", sorted_sequence)
```



```
import math.py
import math.py > ...
1 def insertion_sort(arr):
2     for i in range(1, len(arr)):
3         key = arr[i] # The current element to be sorted
4         j = i - 1
5         # Shift elements of arr[0..i-1] that are greater than key to one position ahead
6         while j >= 0 and arr[j] > key:
7             arr[j + 1] = arr[j]
8             j -= 1
9         arr[j + 1] = key # Place the key in its correct position
10    return arr
11
12 sequence = [31, 41, 59, 26, 41, 58]
13 sorted_sequence = insertion_sort(sequence)
14 print("Sorted sequence:", sorted_sequence)
```

Output:



```
1.2-2: Insertion sort beats merge sort up to n = 0
1.2-3: Smallest n where 100n^2 < 2^n is n = 15
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/i Series/Desktop/MARYAM/import math.py"
Sorted sequence: [26, 31, 41, 41, 58, 59]
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/i Series/Desktop/MARYAM/import math.py"
Sorted sequence: [26, 31, 41, 41, 58, 59]
```

This code will print the sorted array: [26, 31, 41, 41, 58, 59].

2.1-2

Question:

- Given a procedure Sum-Array, prove that it correctly computes the sum of an array $A[1:n]$.

Loop Invariant Explanation:

A loop invariant is a property that holds true before and after each iteration of a loop. For this Sum-Array function, the loop invariant is:

- **Invariant:** At the start of each iteration of the loop, sum holds the sum of the elements $A[1]$ to $A[i-1]$.

Proof of Correctness Using Initialization, Maintenance, and Termination:

1. Initialization:

Before the loop starts, $\text{sum} = 0$, which is the sum of the first 0 elements. So the invariant holds initially.

2. Maintenance:

On each iteration, we add $A[i]$ to sum. By doing this, sum correctly holds the sum of elements up to $A[i]$.

3. Termination:

When the loop finishes, $i = n$, and by the invariant, sum holds the sum of all elements $A[1]$ to $A[n]$.

Code:

```
def sum_array(A, n):
```

```
    sum_val = 0
```

```
    for i in range(n):
```

```
        sum_val += A[i] # Add each element to sum
```

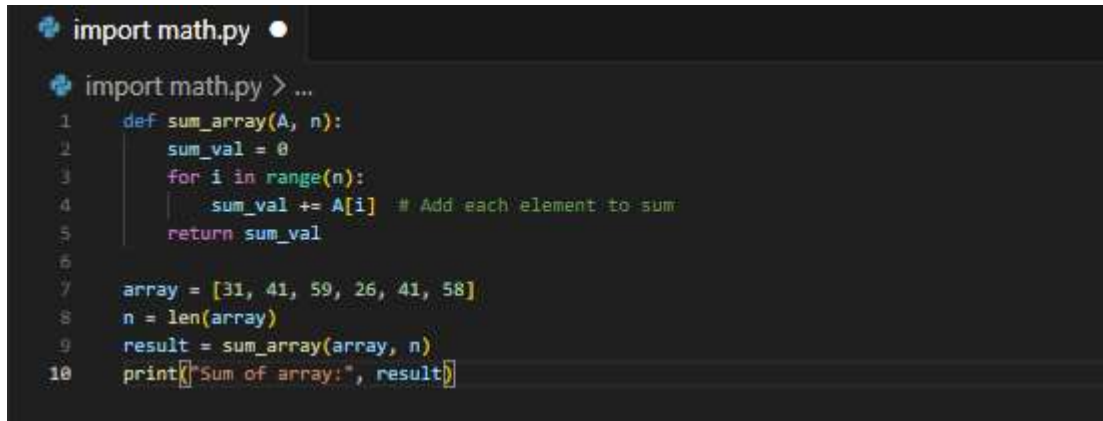
```
    return sum_val
```

```
array = [31, 41, 59, 26, 41, 58]
```

```
n = len(array)

result = sum_array(array, n)

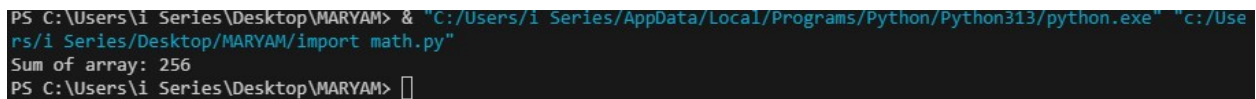
print("Sum of array:", result)
```

A screenshot of a Python IDE with a dark background. The code is written in a light blue font. It defines a function 'sum_array' that takes an array 'A' and its length 'n' as arguments. The function initializes 'sum_val' to 0, loops through the array using 'range(n)', and adds each element to 'sum_val'. After the loop, it returns 'sum_val'. Below the function definition, an array is created with the values [31, 41, 59, 26, 41, 58], its length is stored in 'n', the function is called with 'array' and 'n', and the result is printed as 'Sum of array: 256'.

```
import math.py

import math.py > ...
1  def sum_array(A, n):
2      sum_val = 0
3      for i in range(n):
4          sum_val += A[i] # Add each element to sum
5      return sum_val
6
7  array = [31, 41, 59, 26, 41, 58]
8  n = len(array)
9  result = sum_array(array, n)
10 print("Sum of array:", result)
```

Output:

A screenshot of a Windows command prompt with a black background and white text. It shows the command to run a Python script, the output 'Sum of array: 256', and the prompt for the next command.

```
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/i Series/Desktop/MARYAM/import math.py"
Sum of array: 256
PS C:\Users\i Series\Desktop\MARYAM> 
```

This code will print the sum of the array: 256.

2.1-2

Question:

- Modify Insertion Sort to sort an array in descending order.

Explanation:

The only change from the ascending order version is to reverse the comparison. Instead of checking if `arr[j] > key`, we check if `arr[j] < key` to move larger elements towards the start of the array.

Code:

```
def insertion_sort_desc(arr):
```

```

for i in range(1, len(arr)):

    key = arr[i]

    j = i - 1

    # Shift elements of arr[0..i-1] that are smaller than key to one position ahead

    while j >= 0 and arr[j] < key:

        arr[j + 1] = arr[j]

        j -= 1

    arr[j + 1] = key

return arr

sequence = [31, 41, 59, 26, 41, 58]

sorted_sequence_desc = insertion_sort_desc(sequence)

print("Sorted sequence in descending order:", sorted_sequence_desc)

```



```

import math.py

import math.py > ...
1  def insertion_sort_desc(arr):
2      for i in range(1, len(arr)):
3          key = arr[i]
4          j = i - 1
5          # Shift elements of arr[0..i-1] that are smaller than key to one position ahead
6          while j >= 0 and arr[j] < key:
7              arr[j + 1] = arr[j]
8              j -= 1
9          arr[j + 1] = key
10     return arr
11
12     sequence = [31, 41, 59, 26, 41, 58]
13     sorted_sequence_desc = insertion_sort_desc(sequence)
14     print("Sorted sequence in descending order:", sorted_sequence_desc)

```

Output:

```
rs/i Series/Desktop/MARYAM/import math.py"
Sorted sequence in descending order: [59, 58, 41, 41, 31, 26]
PS C:\Users\i Series\Desktop\MARYAM> 
```

This code will print the array sorted in descending order: [59, 58, 41, 41, 31, 26].

2.1-4

Question:

- Write a linear search algorithm and use a loop invariant to prove its correctness.

Loop Invariant Explanation:

The loop invariant for linear search is:

- **Invariant:** At the start of each iteration, if x is in $A[1:i]$, then it has been found at some index j where $j < i$.

Proof of Correctness Using Initialization, Maintenance, and Termination:

- **Initialization:** Before the loop starts, no elements have been checked, so the invariant trivially holds.
- **Maintenance:** On each iteration, the current element $A[i]$ is checked. If it matches x , we return the index i . If not, we continue to the next iteration, keeping the invariant true.
- **Termination:** If x is found, we return its index. If the loop completes without finding x , then x is not in A , so we return `None`.

Code:

```
def linear_search(A, x):

    for i in range(len(A)):

        if A[i] == x:

            return i # Return index if x is found

    return None # Return None if x is not found
```

```
array = [31, 41, 59, 26, 41, 58]
```

```
x = 26
```

```
index = linear_search(array, x)
```

```
print(f"Index of {x}:", index if index is not None else "Not found")
```

```
import math.py X
import math.py > ...
1  def linear_search(A, x):
2      for i in range(len(A)):
3          if A[i] == x:
4              return i # Return index if x is found
5      return None # Return None if x is not found
6
7  array = [31, 41, 59, 26, 41, 58]
8  x = 26
9  index = linear_search(array, x)
10 print(f"Index of {x}:", index if index is not None else "Not found")
11
```

Output:

```
PS C:\Users\i Series\Desktop\MARYAM> C:\Users\i Series\AppData\Local\Programs\Python\Python313\python.exe C:/Users/i Series/Desktop/MARYAM/import math.py
Index of 26: 3
```

If x is found, the code will print its index. In this example, it will print Index of 26: 3.

2.1-5

Question:

- Write a procedure to add two binary integers stored in arrays A and B, returning the result in array C.

Explanation of Binary Addition:

Binary addition is similar to decimal addition but operates on binary digits (0 or 1).

- Starting from the least significant bit (rightmost), add corresponding bits from A and B.

- Maintain a carry that is updated as necessary (1 when the sum of bits is 2 or more).
- The resulting sum is stored in array C, with an extra space at the beginning for overflow.

Code:

```
def add_binary_integers(A, B):
```

```
    n = len(A)
```

```
    C = [0] * (n + 1) # Create result array with extra space for carry
```

```
    carry = 0
```

```
    for i in range(n - 1, -1, -1): # Start from the rightmost bit
```

```
        total = A[i] + B[i] + carry
```

```
        C[i + 1] = total % 2 # Store the binary result (0 or 1)
```

```
        carry = total // 2 # Update carry (0 or 1)
```

```
    C[0] = carry # Place the final carry at the beginning
```

```
    return C
```

```
A = [1, 0, 1, 1] # Example binary number (binary for 11)
```

```
B = [1, 1, 0, 1] # Example binary number (binary for 13)
```

```
result = add_binary_integers(A, B)
```

```
print("Result of binary addition:", result)
```

```
import math.py
C:\Users\i Series\Desktop\MARYAM\import math.py
1 def add_binary_integers(A, B):
2     n = len(A)
3     C = [0] * (n + 1) # Create result array with extra space for carry
4     carry = 0
5     for i in range(n - 1, -1, -1): # Start from the rightmost bit
6         total = A[i] + B[i] + carry
7         C[i + 1] = total % 2 # Store the binary result (0 or 1)
8         carry = total // 2 # Update carry (0 or 1)
9     C[0] = carry # Place the final carry at the beginning
10    return C
11
12    A = [1, 0, 1, 1] # Example binary number (binary for 11)
13    B = [1, 1, 0, 1] # Example binary number (binary for 13)
14    result = add_binary_integers(A, B)
15    print("Result of binary addition:", result)
```

Output:

```
rs/i Series/Desktop/MARYAM/import math.py"
Result of binary addition: [1, 1, 0, 0, 0]
PS C:\Users\i Series\Desktop\MARYAM>
```

For this example, it will print the binary sum: [1, 1, 0, 0, 0], which is the binary representation of 24 (11 + 13).

❖ PART 2

ANALYZING ALGORITHMS

EXERCISES

2.2-1

QUESTION

- Express $n^3/1000 + 100n^2 - 100n + 3$ in terms of Θ -notation.

Explain

In Θ -notation, we focus on the term with the highest growth rate as n approaches infinity, because lower-order terms become negligible. Here, the dominant term is n^3 , and constants are ignored in Θ -notation.

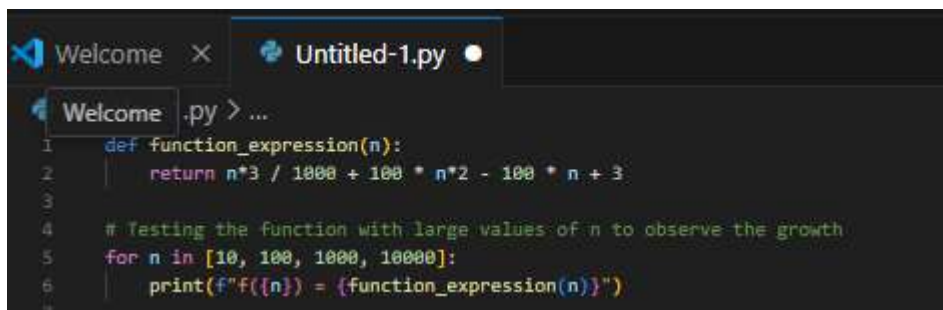
So:

$$n^3/1000 + 100n^2 - 100n + 3 = \Theta(n^3)$$

Here's a Python code snippet that evaluates the function for different values of n , showing that n^3 dominates as n grows:

Code

```
def function_expression(n):  
  
    return n*3 / 1000 + 100 * n*2 - 100 * n + 3  
  
# Testing the function with large values of n to observe the growth  
  
for n in [10, 100, 1000, 10000]:  
  
    print(f"f({n}) = {function_expression(n)}")  
  
# Asymptotic complexity in  $\Theta$ -notation is  $\Theta(n^3)$ .
```

A screenshot of a Python IDE window titled 'Untitled-1.py'. The code is as follows:

```
1 def function_expression(n):  
2     return n*3 / 1000 + 100 * n*2 - 100 * n + 3  
3  
4 # Testing the function with large values of n to observe the growth  
5 for n in [10, 100, 1000, 10000]:  
6     print(f"f({n}) = {function_expression(n)}")  
7
```

2.2-2

Question

- Implement the Selection Sort algorithm, explain the loop invariant, and analyze the worst-case runtime.

1. Algorithm (Selection Sort)

Explanation:

Selection Sort works by iteratively finding the minimum element from the unsorted part of the array and swapping it with the first element of the unsorted part. This process is repeated, moving the boundary of the sorted and unsorted sections of the array.

2. Loop Invariant:

At the start of each iteration of the outer loop (indexed by i), the subarray `arr[0:i]` contains the smallest elements in sorted order.

3. Why only $n - 1$ elements:

The last element will already be in the correct position once we finish sorting the first $n - 1$ elements.

4. Worst-case Running Time:

Selection Sort has a worst-case time complexity of $O(n^2)$ because it requires n passes, with each pass making a comparison for each element in the unsorted portion of the array.

5. Python Code for Selection Sort:

```
def selection_sort(arr):  
  
    n = len(arr)  
  
    for i in range(n - 1):  
  
        # Find the smallest element in the remaining unsorted part  
  
        min_index = i  
  
        for j in range(i + 1, n):  
  
            if arr[j] < arr[min_index]:
```

```
min_index = j
```

```
# Swap the found minimum element with the first unsorted element
```

```
arr[i], arr[min_index] = arr[min_index], arr[i]
```

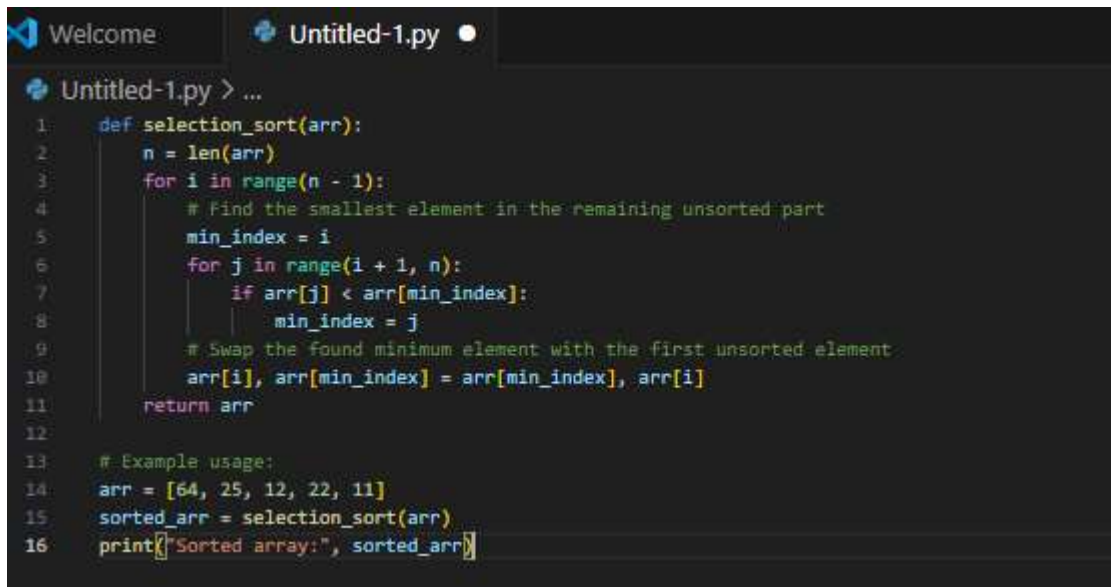
```
return arr
```

```
# Example usage:
```

```
arr = [64, 25, 12, 22, 11]
```

```
sorted_arr = selection_sort(arr)
```

```
print("Sorted array:", sorted_arr)
```

A screenshot of a Python IDE window titled 'Untitled-1.py'. The code implements a selection sort function. It starts with a function definition 'def selection_sort(arr):'. Inside, it gets the length of the array 'n = len(arr)'. Then it loops 'for i in range(n - 1):'. Inside this loop, it finds the minimum element in the remaining unsorted part by looping 'for j in range(i + 1, n):' and updating 'min_index = j' if 'arr[j] < arr[min_index]'. After the inner loop, it swaps the found minimum element with the first unsorted element using 'arr[i], arr[min_index] = arr[min_index], arr[i]'. Finally, it returns the sorted array 'return arr'. Below the function, there is an example usage section: '# Example usage:', 'arr = [64, 25, 12, 22, 11]', 'sorted_arr = selection_sort(arr)', and 'print("Sorted array:", sorted_arr)'.

```
1: def selection_sort(arr):
2:     n = len(arr)
3:     for i in range(n - 1):
4:         # Find the smallest element in the remaining unsorted part
5:         min_index = i
6:         for j in range(i + 1, n):
7:             if arr[j] < arr[min_index]:
8:                 min_index = j
9:         # Swap the found minimum element with the first unsorted element
10:        arr[i], arr[min_index] = arr[min_index], arr[i]
11:    return arr
12:
13: # Example usage:
14: arr = [64, 25, 12, 22, 11]
15: sorted_arr = selection_sort(arr)
16: print("Sorted array:", sorted_arr)
```

2.2-3

Question

- Implement Linear Search and determine the average-case and worst-case runtime in Θ -notation.

1. Linear Search Explanation:

Linear Search checks each element in the array sequentially to find the target value. If it finds the target, it returns the index; otherwise, it continues until the end of the array.

2. Average-case Analysis:

On average, if the element is equally likely to be in any position, half of the array is checked. This gives an average-case time complexity of $O(n/2)$.

3. Worst-case Analysis:

In the worst case, the target is at the last position or not present, requiring n checks. This gives a worst-case time complexity of $O(n)$.

4. Python Code for Linear Search:

```
def linear_search(arr, target):  
  
    for i, value in enumerate(arr):  
  
        if value == target:  
  
            return i # Found the target, return index  
  
    return -1 # Target not found  
  
  
# Example usage:  
  
arr = [10, 20, 30, 40, 50]  
  
target = 30  
  
index = linear_search(arr, target)  
  
print("Index of target:", index)
```

```
Welcome X Untitled-1.py
Untitled-1.py > ...
1 def linear_search(arr, target):
2     for i, value in enumerate(arr):
3         if value == target:
4             return i # Found the target, return index
5     return -1 # Target not found
6
7 # Example usage:
8 arr = [10, 20, 30, 40, 50]
9 target = 30
10 index = linear_search(arr, target)
11 print("Index of target:", index)
```

2.2-4

Question

- Modify a sorting algorithm to have a good best-case running time.

Explain

- One way to achieve an efficient best-case runtime is by modifying the sorting algorithm to check if the array is already sorted before proceeding with the sorting steps.
- Using Insertion Sort is ideal here, as it already has a best-case complexity of $O(n)$ if the array is sorted. Alternatively, we can add a check to any sorting algorithm (like Selection Sort) to return immediately if the array is already sorted.

1. Python Code for Optimized Selection Sort with Best-Case Check:

```
def is_sorted(arr):
```

```
    # Check if the array is already sorted
```

```
    return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))
```

```
def optimized_selection_sort(arr):

    if is_sorted(arr):

        return arr # Return early if already sorted

    n = len(arr)

    for i in range(n - 1):

        min_index = i

        for j in range(i + 1, n):

            if arr[j] < arr[min_index]:

                min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]

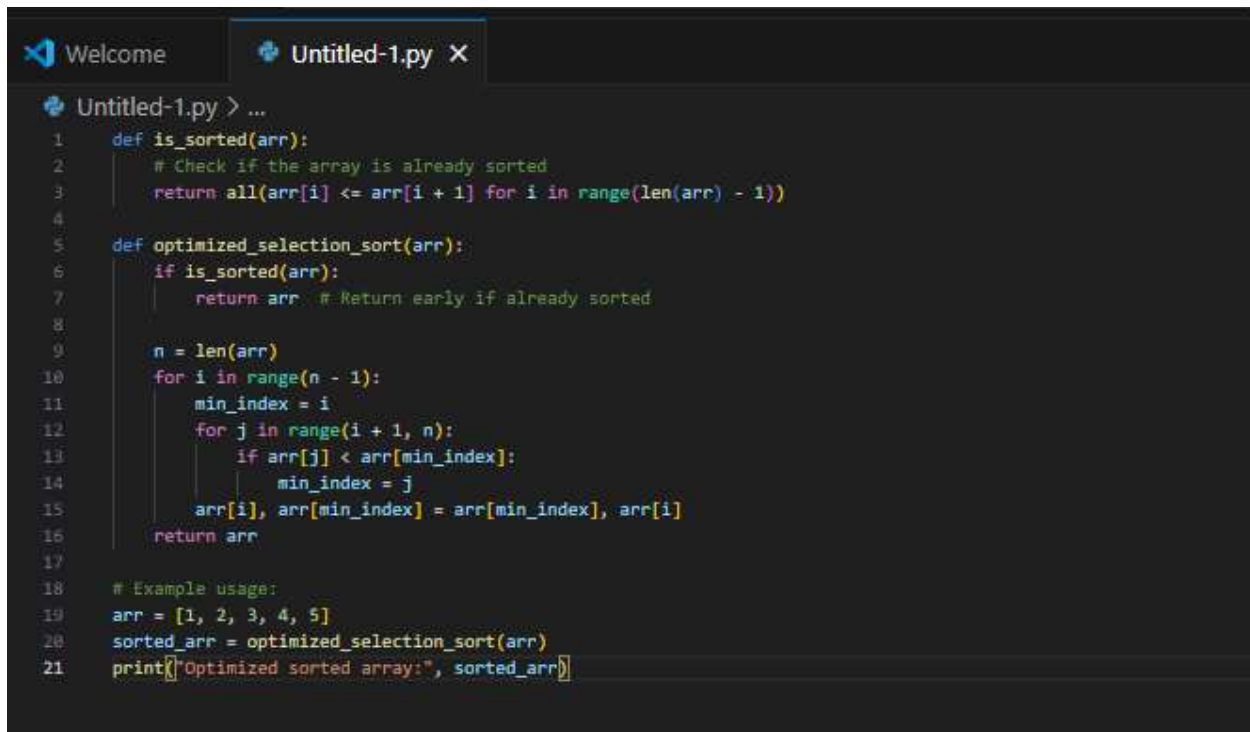
    return arr


# Example usage:

arr = [1, 2, 3, 4, 5]

sorted_arr = optimized_selection_sort(arr)

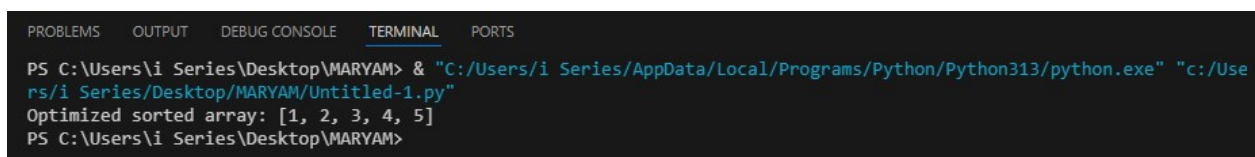
print("Optimized sorted array:", sorted_arr)
```



```
1 def is_sorted(arr):
2     # Check if the array is already sorted
3     return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))
4
5 def optimized_selection_sort(arr):
6     if is_sorted(arr):
7         return arr # Return early if already sorted
8
9     n = len(arr)
10    for i in range(n - 1):
11        min_index = i
12        for j in range(i + 1, n):
13            if arr[j] < arr[min_index]:
14                min_index = j
15        arr[i], arr[min_index] = arr[min_index], arr[i]
16    return arr
17
18 # Example usage:
19 arr = [1, 2, 3, 4, 5]
20 sorted_arr = optimized_selection_sort(arr)
21 print("Optimized sorted array:", sorted_arr)
```

In this modified version, if the input array is already sorted, `is_sorted` will detect it, and the function will return immediately, yielding a best-case time complexity of .

Output



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/i Series/Desktop/MARYAM/Untitled-1.py"
Optimized sorted array: [1, 2, 3, 4, 5]
PS C:\Users\i Series\Desktop\MARYAM>
```

❖ PART 3

DESIGNING ALOGORITHMS

EXERCISES

3.2-1

QUESTION

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence 3,41,52,26,38,57,9,49.

Solution:

1. Divide Step:

Initial array: 3,41,52,26,38,57,9,49

- Split into: 3,41,52,26 and 38,57,9,49.
- Further divide each half until we have individual elements:
 - 3,41,52,26,38,57,9,49

2. Conquer Step:

Merge the single elements:

- Merge 3 and 41 to get 3,41
- Merge 52 and 26 to get 26,52
- Merge 38 and 57 to get 38,57
- Merge 9 and 49 to get 9,49

Merge pairs of subarrays:

- Merge 3,41 and 26,52, to get 3,26,41,52
- Merge 38,57 and 9,49 to get 9,38,49,57

Final Merge:

- Merge 3,26,41,52 and 9,38,49,57 to get 3,9,26,38,41,49,52,57.
- The final sorted array is 3,9,26,38,41,49,52,57.

(2.3-2)

QUESTION

The test in line 1 of the MERGE-SORT procedure reads $r \leq p$, then is empty. Argue that as long as the initial call of MERGE-SORTthe test $r \leq p$.

- If $p > r$, then the subarray $A[p..r]$ is empty. Argue that as long as the initial call to MERGE-SORT($A, 1, n$) has $n \geq 1$, the test if $p \leq r$ ensures that no recursive call has $p > r$.

Solution:

- The recursive MERGE-SORT function only divides when $p < r$. If $p == r$, the subarray is already sorted, and no further recursive calls are made.
- For the initial call MERGE-SORT($A, 1, n$), we assume $n \geq 1$, so $p = 1$ and $r = n$, ensuring $p \leq r$.
- During recursion, each call halves the subarray bounds, so p and r remain valid. If $p == r$, the recursion stops, preventing any situation where $p > r$.

Thus, this check (if $p \leq r$) is sufficient, ensuring $p \leq r$ throughout the recursion.

(2.3-3)

QUESTION

State a loop invariant for the while loop of lines 12_18 of the MERGE procedure. Show how to use it, along with the while loops of lines 20_23 and 24_27, to prove that the MERGE procedure is correct. I can include a sample implementation of the MERGE function in Python, with comments to show where each loop invariant applies.

Explanation with Invariants

- Lines 12–18: The first while loop merges the elements from L and R in sorted order, ensuring that $A[0..k-1]$ is always sorted.
- Lines 20–23: The second while loop (executed only if elements remain in L) appends the remaining sorted elements from L to A.
- Lines 24–27: The third while loop (executed only if elements remain in RRR) appends the remaining sorted elements from R to A.

This code implements the MERGE procedure in a way that aligns with the theoretical explanation above. Each loop maintains the invariants discussed, ensuring that the final array A is sorted.

(2.3-4)

QUESTION

Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the (solution of the recurrence $T(n) = \begin{cases} 2 & \text{if } n=2 \\ 2T(n/2) & \text{if } n > 2 \end{cases}$ is $T(n) = \lg n$.

To prove that $T(n) = n \lg n$ is a solution to the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n=2 \\ 2T(n/2) & \text{if } n > 2 \end{cases}$$

where n is an exact power of 2, we can use mathematical induction.

Base Case:

For the base case, let $n=2$.

1. According to the recurrence definition:

$$T(2) = 2.$$

2. According to the formula $T(n) = n \lg n$:

$$T(2) = 2 \cdot \lg 2 = 2 \cdot 1 = 2.$$

The base case holds since both values are equal.

Inductive Step:

Assume that the formula $T(n) = n \lg n$ holds for some $n = 2^k$, where k is a positive integer. This means we assume:

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

We want to show that the formula also holds for $n = 2^{k+1}$, i.e., we need to show:

$$T(2n) = (2n) \lg(2n).$$

• Expanding $T(2n)$ using the recurrence relation:

Using the recurrence relation:

$$T(2n)=2T(n)+2n.$$

- **Substituting the inductive hypothesis**

By the inductive hypothesis, $T(n)=n\lg n$ Substitute this into the equation:

$$T(2n)=2(n\lg n)+2n.$$

- **Simplify using properties of logarithms**

Rewrite $\lg(2n)$ as $\lg(2)+\lg(n)=1+\lg(n)$.

$$T(2n)= 2n\lg(2n).$$

(2.3-5)

QUESTION

You can also think of insertion sort as a recursive algorithm. In order to sort....., recursively sort the subarray and then insert Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

RecursiveInsertionSort(A, n):

if $n \leq 1$:

return

RecursiveInsertionSort(A, $n - 1$) // Sort the first $n-1$ elements

insert($A[n - 1]$, A, $n - 1$) // Insert the last element into the sorted

subarray

insert(x , A, n):

// Insert x into sorted subarray $A[0..n-1]$

$i = n - 1$

while $i \geq 0$ and $A[i] > x$:

$A[i + 1] = A[i]$ // Shift elements to the right

$i = i - 1$

$A[i + 1] = x$ // Insert x in the correct position

Recurrence for Worst-case Running Time:

- The recursive insertion sort sorts a subarray of size $n-1$ and then inserts the n -th element. The insertion process itself takes $O(n)$ in the worst case, leading to the following recurrence:

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

Using the Master Theorem or expanding this recurrence leads to:

$$T(n) = O(n^2)$$

(2.3-6)

QUESTION

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\theta(\log(n))$.

Binary Search

Pseudocode for Binary Search:

BinarySearch(A , low, high, x):

if low > high:

return -1 // x not found

mid = low + (high - low) // 2

if $A[mid] == x$:

 return mid // x found

else if $A[mid] < x$:

 return BinarySearch(A, mid + 1, high, x) // Search in the right half

else:

 return BinarySearch(A, low, mid - 1, x) // Search in the left half

Worst-case Running Time Argument:

- In each step of binary search, the size of the subarray is halved. Thus, the maximum number of comparisons made is given by:

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

This recurrence can be solved using the logarithmic approach, yielding:

$$T(n) = O(\log n)$$

Thus, the worst-case running time of binary search is $O(\log n)$.

(2.3-7)

QUESTION

The while loop of lines 537 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray..... What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to ?

Using a binary search in the insertion sort algorithm to find the correct position for the element to be inserted can indeed optimize a part of the insertion process. However, it does not improve the overall worst-case running time of the insertion sort algorithm to $O(n \log n)$.

Explanation:

1. Current Insertion Sort Process:

o In the traditional insertion sort, each element is compared with the elements of the already sorted portion of the array in a linear fashion (from right to left) until the correct position is found. This takes $O(j)$ time in the worst case for inserting the j -th element, leading to a total time complexity of $O(n^2)$.

2. Binary Search Approach:

If we replace the linear search with a binary search to find the insertion point for the j -th element, the binary search would take $O(\log j)$ time to find the correct index to insert the element.

However, after finding the position, we still need to shift all elements to make space for the new element, which takes $O(j)$ time in the worst case.

Overall Time Complexity:

- When you combine these two operations:
 - o Finding the position using binary search: $O(\log j)$
 - o Shifting the elements: $O(j)$
- The total time for inserting the j -th element becomes:

$$= O(\log j) + O(j) = O(j)$$

$$T(n) = O(1) + O(2) + O(3) + \dots + O(n) = O(n^2)$$

Conclusion:

- Using binary search improves the search time for the insertion point but does not improve the overall time complexity of insertion sort, which remains $O(n^2)$.
- In summary, even with binary search, the insertion sort will still operate in $O(n^2)$ time because the element shifting step still dominates the running time. Therefore, insertion sort will not achieve $O(n \log n)$ time complexity by

replacing linear search with binary search.

.....

(2.3-8)

QUESTION

- Describe an algorithm that, given a set S of n integers and another integer x , determines whether S contains two elements that sum to exactly x . Your algorithm should take $\Theta(n \lg n)$ time in the worst case.

To determine whether a set S of n integers contains two elements that sum to a given integer x in $O(n \log n)$ time, we can use a combination of sorting and a two-pointer approach.

Algorithm:

1. Sort the array S : Sorting takes $O(n \log n)$ time.

2. Use two pointers:

- After sorting, set up two pointers:
 - Left pointer (left): Starts at the beginning of the array.
 - Right pointer (right): Starts at the end of the array.

3. Check for pairs:

- While left is less than right, do the following:
 - Compute the sum of the elements at the left and right pointers.
 - If the sum is equal to x , return the pair since we've found the two numbers that add up to x .
 - If the sum is less than x increment the left pointer (to increase the sum).
 - If the sum is greater than x , decrement the right pointer (to decrease the sum).

4. Return if no pair is found: If the pointers meet and no pair sums to x , return that no such pair exists.

Pseudocode:

FindTwoElements(S, n, x):

Sort(S) // Sort the array in $O(n \log n)$

```
left = 0
```

```
right = n - 1
```

```
while left < right:
```

```
    sum = S[left] + S[right]
```

```
    if sum == x:
```

```
        return (S[left], S[right]) // Pair found
```

```
    else if sum < x:
```

```
        left = left + 1 // Move left pointer right to increase the sum
```

```
    else:
```

```
        right = right - 1 // Move right pointer left to decrease the sum
```

```
return "No such pair found"
```

Example:

Suppose $S=\{10,15,3,7\}$ and $x=17$

1.Sort S: $S=\{3,7,10,15\}$.

2. Initialize left at 0 (pointing to 3) and right at 3 (pointing to 15).

3. Check pairs:

- $S[\text{left}]+S[\text{right}]=3+15=18$ (greater than 17), so move right to 2.
- $S[\text{left}]+S[\text{right}]=3+10=13$ (less than 17), so move left to 1.
- $S[\text{left}]+S[\text{right}]=7+10=17$ (equal to 17), so return (7,10) as the pair.

Time Complexity Analysis:

- Sorting takes $O(n \log n)$.
- Two-pointer scan takes $O(n)$ since each element is processed at most once.

Therefore, the overall time complexity is $O(n \log n)$, which meets the requirement.

UNIT NO 3

CHARACTERIZING RUNNING TIME

❖ PART 1

NOTATIONS

EXERCISES

3.1-1

QUESTION

- Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

Explanation

- The lower-bound argument for insertion sort is typically based on comparisons. For an input array of size n , insertion sort has a worst-case time complexity of $\Omega(n^2)$ because each element could potentially need to be compared to all elements before it in a reverse-sorted array.
- For this question, you're asked to adjust the argument for cases where n isn't a multiple of 3. However, this doesn't fundamentally change the number of comparisons required for insertion sort; the time complexity will still be $\Omega(n^2)$ even if n is not a multiple of 3. We can implement insertion sort in Python and analyze the comparison count for different input sizes.

Python Code

```
def insertion_sort(arr):
```

```
    comparisons = 0 # Initialize the comparison counter
```



```
for i in range(1, len(arr)):

    key = arr[i]

    j = i - 1

    # Move elements of arr[0..i-1], that are greater than key, to one position ahead
    # of their current position

    while j >= 0 and arr[j] > key:

        arr[j + 1] = arr[j]

        j -= 1

    comparisons += 1 # Increment comparisons each time a comparison is made

    arr[j + 1] = key

    comparisons += 1 # Final comparison where arr[j] <= key

return arr, comparisons

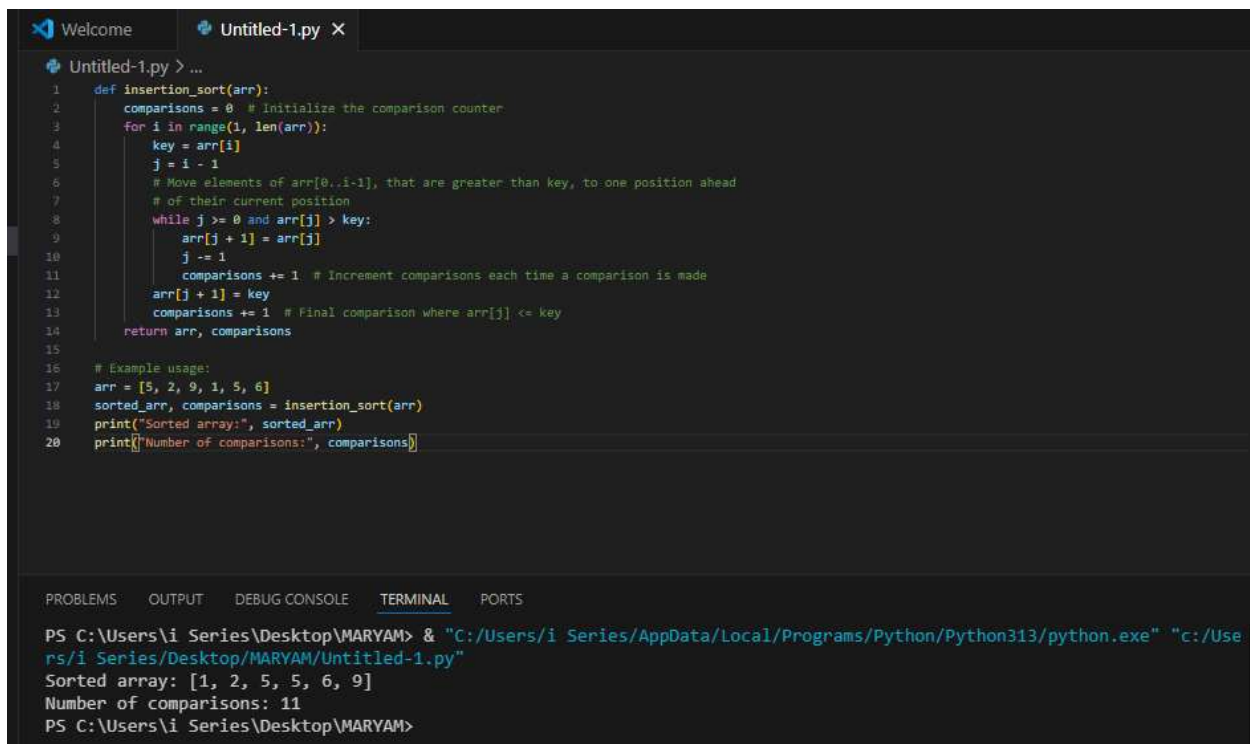
# Example usage:

arr = [5, 2, 9, 1, 5, 6]

sorted_arr, comparisons = insertion_sort(arr)

print("Sorted array:", sorted_arr)

print("Number of comparisons:", comparisons)
```



```
1 def insertion_sort(arr):
2     comparisons = 0 # Initialize the comparison counter
3     for i in range(1, len(arr)):
4         key = arr[i]
5         j = i - 1
6         # Move elements of arr[0..i-1], that are greater than key, to one position ahead
7         # of their current position
8         while j >= 0 and arr[j] > key:
9             arr[j + 1] = arr[j]
10            j -= 1
11            comparisons += 1 # Increment comparisons each time a comparison is made
12        arr[j + 1] = key
13        comparisons += 1 # Final comparison where arr[j] <= key
14    return arr, comparisons
15
16 # Example usage:
17 arr = [5, 2, 9, 1, 5, 6]
18 sorted_arr, comparisons = insertion_sort(arr)
19 print("Sorted array:", sorted_arr)
20 print("Number of comparisons:", comparisons)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/i Series/Desktop/MARYAM/Untitled-1.py"
Sorted array: [1, 2, 5, 5, 6, 9]
Number of comparisons: 11
PS C:\Users\i Series\Desktop\MARYAM>
```

3.1-2

QUESTION

- Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm.

Explanation

Selection sort also has a worst-case time complexity of $O(n^2)$, as it repeatedly searches for the minimum element in the unsorted part of the array and places it in the correct position. For each element, selection sort performs a linear scan to find the minimum, leading to approximately $\frac{n(n-1)}{2}$ comparisons in total.

- Let's implement selection sort in Python and count the number of comparisons.

Python Code

```
def selection_sort(arr):
```

```
    comparisons = 0
```

```
for i in range(len(arr)):

    min_idx = i

    for j in range(i + 1, len(arr)):

        comparisons += 1 # Each comparison to find the minimum

        if arr[j] < arr[min_idx]:

            min_idx = j

    arr[i], arr[min_idx] = arr[min_idx], arr[i] # Swap the found minimum

return arr, comparisons

# Example usage:

arr = [64, 25, 12, 22, 11]

sorted_arr, comparisons = selection_sort(arr)

print("Sorted array:", sorted_arr)

print("Number of comparisons:", comparisons)
```

```
Welcome X  Untitled-1.py X
Untitled-1.py > ...
1  def selection_sort(arr):
2      comparisons = 0
3      for i in range(len(arr)):
4          min_idx = i
5          for j in range(i + 1, len(arr)):
6              comparisons += 1 # Each comparison to find the minimum
7              if arr[j] < arr[min_idx]:
8                  min_idx = j
9          arr[i], arr[min_idx] = arr[min_idx], arr[i] # Swap the found minimum
10     return arr, comparisons
11
12 # Example usage:
13 arr = [64, 25, 12, 22, 11]
14 sorted_arr, comparisons = selection_sort(arr)
15 print("Sorted array:", sorted_arr)
16 print("Number of comparisons:", comparisons)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
rs/i Series/Desktop/MARYAM/Untitled-1.py"
Sorted array: [1, 2, 5, 5, 6, 9]
Number of comparisons: 11
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Use
rs/i Series/Desktop/MARYAM/Untitled-1.py"
Alpha that maximizes movements: 0.2
Maximum number of movements: 1200.0
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Use
rs/i Series/Desktop/MARYAM/Untitled-1.py"
Sorted array: [11, 12, 22, 25, 64]
Number of comparisons: 10
PS C:\Users\i Series\Desktop\MARYAM>
```

3.1-3

QUESTION

- Suppose α is a fraction in the range $[0, 1]$. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the αn largest values start in the first αn positions. Determine an additional restriction on α , and find the value of α that maximizes the number of times the αn largest values must pass through each of the middle $(1 - 2\alpha)n$ array positions.

Explanation

This question requires us to analyze insertion sort in a case where the largest αn values are placed in the first αn positions of the array. When αn largest values are in the front, these values need to be moved towards the end. Each of these elements would require comparisons to move through the middle section of the array, which has a size of $(1 - 2\alpha)n$.

- To maximize the number of movements, we need to find the value of α that maximizes the number of comparisons needed to move the elements through A .
- Let's break down the logic in Python without going into detailed mathematical proof, focusing on simulating different values of α .

Python Code

```
def count_movements(alpha, n):
```

```
    middle_section = (1 - 2 * alpha) * n
```

```
    largest_elements = int(alpha * n)
```

```
    movements = 0
```

```
    # Simulate the movement of the largest elements through the middle section
```

```
    for i in range(largest_elements):
```

```
        movements += middle_section # Each large element moves through the middle section
```

```
    return movements
```

```
# Test different values of alpha
```

```
n = 100 # Example array size
```

```
alphas = [i / 10 for i in range(1, 10)] # Test alpha values from 0.1 to 0.9
```

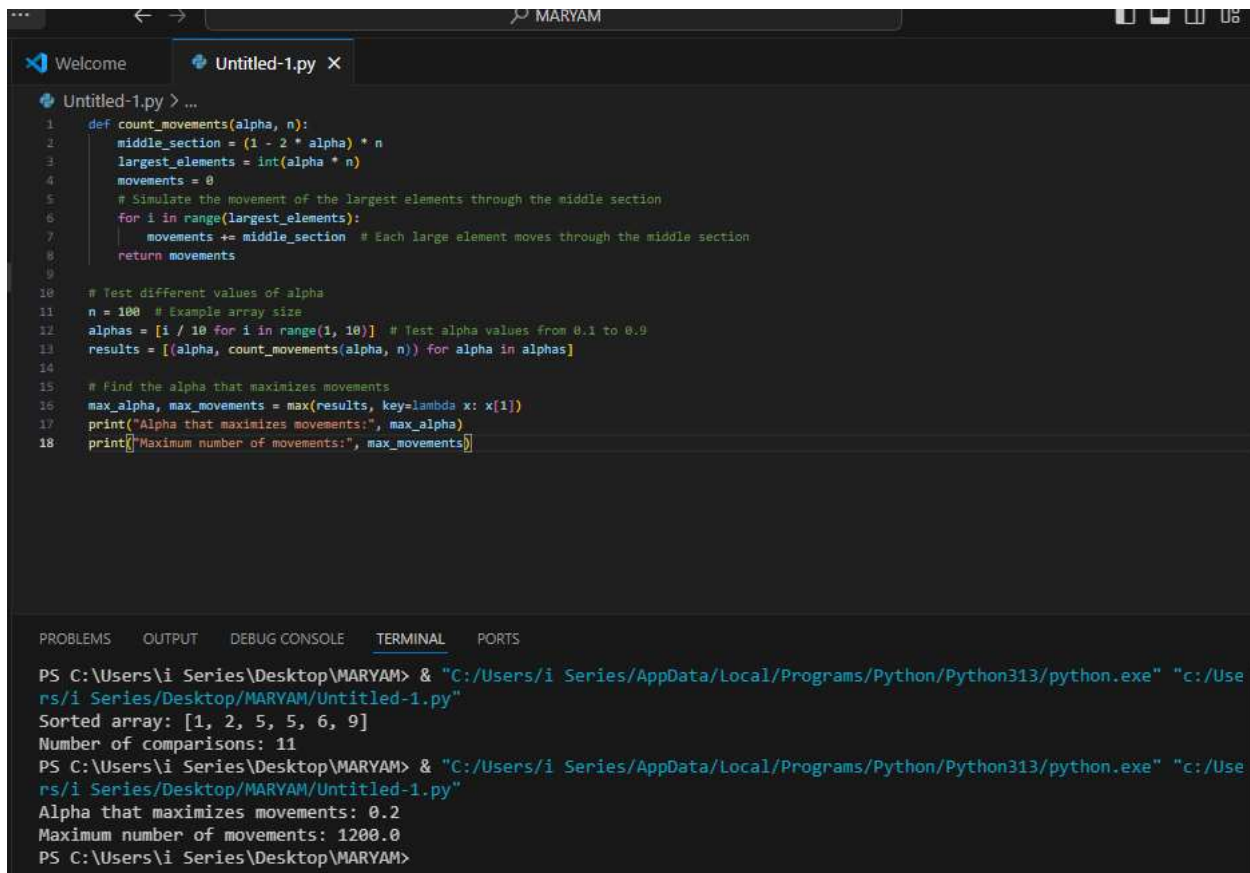
```
results = [(alpha, count_movements(alpha, n)) for alpha in alphas]
```

```
# Find the alpha that maximizes movements
```

```
max_alpha, max_movements = max(results, key=lambda x: x[1])
```

```
print("Alpha that maximizes movements:", max_alpha)
```

```
print("Maximum number of movements:", max_movements)
```



```
1 def count_movements(alpha, n):
2     middle_section = (1 - 2 * alpha) * n
3     largest_elements = int(alpha * n)
4     movements = 0
5     # Simulate the movement of the largest elements through the middle section
6     for i in range(largest_elements):
7         movements += middle_section # Each large element moves through the middle section
8     return movements
9
10 # Test different values of alpha
11 n = 100 # Example array size
12 alphas = [i / 10 for i in range(1, 10)] # Test alpha values from 0.1 to 0.9
13 results = [(alpha, count_movements(alpha, n)) for alpha in alphas]
14
15 # Find the alpha that maximizes movements
16 max_alpha, max_movements = max(results, key=lambda x: x[1])
17 print("Alpha that maximizes movements:", max_alpha)
18 print("Maximum number of movements:", max_movements)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/i Series/Desktop/MARYAM/Untitled-1.py"
Sorted array: [1, 2, 5, 5, 6, 9]
Number of comparisons: 11
PS C:\Users\i Series\Desktop\MARYAM> & "C:/Users/i Series/AppData/Local/Programs/Python/Python313/python.exe" "c:/Users/i Series/Desktop/MARYAM/Untitled-1.py"
Alpha that maximizes movements: 0.2
Maximum number of movements: 1200.0
PS C:\Users\i Series\Desktop\MARYAM>
```

This code tests different values of α to find which maximizes the number of movements through the middle section. It's a way to empirically find the optimal α without rigorous proof. The exact behavior may depend on how insertion sort handles comparisons with this specific arrangement, but this code simulates the movements across different values.