

NAME

HINA SAJID

ROLL NO

14650

SUBMITTED TO

SIR JAMAL ABDUL AHAD

SUBJECT

DATA STRUCTURE ALGORITHMS (DSA)

DATE

31/10/2024

ASSIGNMENT NO 1

CHAP 1:

➤ **EXERCISE QUESTIONS:**

(1.1-1)

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

➤ **Sorting Example:**

Library Book Sorting:

Imagine a library with 10,000 books that need to be sorted alphabetically by title.

Approach:

To sort these books, a sorting algorithm like Merge Sort or Quick Sort can be applied. These algorithms generally operate at $O(n \log n)$ complexity.

Example Calculation:

- With 10,000 books, the estimated time complexity for sorting them would be $O(10,000 \log(10,000))$.
- Calculating $10,000 \log(10,000)$ (assuming base 2):

$$10,000 \times \log_2(10,000) = 10,000 \times 13.29 = 132,900 \text{ operations}$$

Thus, sorting the books using a fast $O(n \log n)$ algorithm requires around 132,900 operations.

➤ **Shortest Distance Example:**

Navigating in a City

Imagine you're trying to get from Location A to Location B in a city. Let's say the distance between locations is measured on a grid, and you're allowed to move up, down, left, or right.

If Location A is at (1, 2) and Location B is at (8, 9), we calculate the Manhattan distance.

Calculation:

The Manhattan distance formula between points (x_1, y_1) and (x_2, y_2) .

$$\text{➤ Distance} = |x_2 - x_1| + |y_2 - y_1|$$

$$\text{➤ } |x_2 - x_1| + |y_2 - y_1|$$

$$\text{➤ Distance} = |x_2 - x_1| + |y_2 - y_1|$$

So,

$$\text{Distance} = |8 - 1| + |9 - 2| = 7 + 7 = 14$$

$$= |8 - 1| + |9 - 2|$$

$$= 7 + 7 = 14 \text{ units}$$

Thus, the shortest path on a grid would require 14 moves to get from Location A to Location B.

.....

(1.1-2)

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

Efficiency Measures Beyond Speed:

Let's take an algorithm that identifies spam emails. Beyond speed, here's how we'd consider other efficiency measures:

1. Memory Usage:

Suppose the algorithm uses 500 MB of memory. If it runs on a device with only 2 GB of RAM, it's efficient. But on a device with only 512 MB of RAM, it would be too large.

2. Energy Efficiency:

In a smartphone app, the spam filter should minimize battery drain. An energy-intensive algorithm might shorten battery life quickly, so we'd favor an efficient algorithm that conserves power.

3. Scalability:

Suppose the algorithm performs well on 1,000 emails, but what about 1,000,000? We'd need to measure how it scales and performs under increased load to ensure it doesn't crash or slow down significantly.

.....

(1.1-3)

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

➤ **Data Structure Example:**

(Array)

Consider an array storing integers.

1. Strengths:

- Accessing any element by index, such as `arr[4]`, is an $O(1)$ operation.
- Memory-efficient as elements are stored contiguously.

2. Limitations:

- If you want to insert an element at the beginning (e.g., inserting 5 at index 0), all elements must shift one position to the right.
- For an array of size n , inserting an element at the start has a time complexity of $O(n)$ because of shifting elements.

.....

(1.1-4)

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

➤ Similarities and Differences between Shortest-Path and Traveling Salesperson Problems:

- **Similarities:**

Both involve finding optimal routes and use graphs to represent possible paths between nodes.

- **Differences:**

- The shortest-path problem focuses on the minimum distance between two specific points, while the traveling salesperson problem (TSP) aims to find the shortest possible route that visits all points and returns to the starting point. TSP is more complex and often requires approximation.

.....

(1.1-5)

Q: 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.

➤ **Real-World Problem Requiring the Best Solution vs an Approximate Solution:**

1. **Best Solution Needed:** In medical diagnosis, exact identification of diseases is critical to provide effective treatment.
2. **Approximate Solution is Good Enough:**
 - Example: Movie recommendations. While the algorithm tries to suggest the best options based on viewing history, an approximate match (like showing similar genres) can satisfy the user.

.....

(1.1-6)

Q: 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.

❖ **Problem Description**

Imagine a logistics company that delivers packages daily to various destinations. The goal is to determine an **optimal route** that minimizes total travel time or distance. However, there are two possible scenarios:

Scenario 1:

➤ **Entire Input Available:**

When all destinations are known in advance, we can use the **Traveling Salesperson Problem (TSP)** approach to find the optimal route that minimizes distance by visiting each destination exactly once and returning to the starting location.

- **we** are trying to plan a route for a salesperson (or delivery driver) who needs to visit four locations: **A, B, C, and D**. The goal is to minimize the total distance traveled while visiting each location exactly once and returning to the starting point.

Distance Matrix

We have a distance matrix that provides the distances (in kilometers) between each pair of locations:

From/To	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

- The diagonal (0) represents the distance from a location to itself (which is zero).
- The other values represent the distance between different locations. For example, the distance from **A to B** is 10 km, and from **B to C** is 35 km.

TSP Calculation Steps:

To find the optimal route using the TSP, we must consider all possible routes.

With four locations, the possible routes (or permutations) can be calculated as follows:

1. **List All Possible Routes:** Since there are 4 locations, we need to calculate routes starting and ending at the same location. For example, if we start at **A**, we can visit the other locations in different orders. Some possible routes are:

- $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$
- $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$
- $A \rightarrow C \rightarrow B \rightarrow D \rightarrow A$
- $A \rightarrow C \rightarrow D \rightarrow B \rightarrow A$
- $A \rightarrow D \rightarrow B \rightarrow C \rightarrow A$
- $A \rightarrow D \rightarrow C \rightarrow B \rightarrow A$

2. **Calculate Total Distance for Each Route:** For each route, sum the distances based on the distance matrix.

Let's calculate a few routes:

- **Route 1: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$**
 - Distance: $10+35+30+20=95$ km
- **Route 2: $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$**
 - Distance: $10+25+30+15=80$ km
- **Route 3: $A \rightarrow C \rightarrow B \rightarrow D \rightarrow A$**
 - Distance: $15+35+25+20=95$ km

- **Route 4: $A \rightarrow C \rightarrow D \rightarrow B \rightarrow A$**

- Distance: $15+30+25+10=80$ km $15 + 30 + 25 + 10 = 80$ km

- **Route 5: $A \rightarrow D \rightarrow B \rightarrow C \rightarrow A$**

- Distance: $20+25+35+15=95$ km

- **Route 6: $A \rightarrow D \rightarrow C \rightarrow B \rightarrow A$**

- Distance: $20+30+35+10=95$ km

3. **Identify the Shortest Route:** After calculating the total distances for each route, we compare them to find the minimum:

- **Route 2 and Route 4** both have a total distance of **80 km**, which is the shortest distance possible for this set of locations.

Conclusion:

In this TSP scenario, we systematically evaluated all possible routes starting from point **A** and returning to **A** after visiting **B**, **C**, and **D**. By summing the distances from the distance matrix for each route and finding the minimum, we determined that the optimal route is either $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$ or $A \rightarrow C \rightarrow D \rightarrow B \rightarrow A$, both totaling **80 km**.

This example illustrates how TSP can be used in practical scenarios such as delivery route planning, ensuring efficiency in travel.

Scenario 2:

Partial Input Available (Dynamic Routing)

➤ When new delivery requests are received throughout the day, we need a **dynamic routing approach**. A commonly used algorithm for this is the **Nearest Neighbor Algorithm**, where the driver always goes to the closest unvisited location. Alternatively, we can use **Dijkstra's algorithm** each time a new location is added to recalibrate the route dynamically.

- **Dynamic Calculation Example:**

- Assume the driver has visited destinations A and B and is at C when a new delivery request for D arrives.
- We calculate the nearest route from C to D dynamically.
- Suppose the distance from C to D is 30 km, and we need to calculate the new total route, including D
- The recalculated distance becomes the previous route's distance plus the new 30 km to D ensuring that the route remains efficient as new destinations are added.

Summary:

This example highlights how the delivery route can be calculated either in advance (using TSP) or dynamically (using Nearest Neighbor or Dijkstra's algorithms), depending on whether the entire input is available upfront or

arrives over time. Calculations vary based on the algorithm, with TSP requiring factorial calculations for exact solutions and dynamic routing typically handled by recalculating with simpler heuristic approaches.

(Algorithm as technology)

(1.2-1)

Q: Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

- **Example of an Application Requiring Algorithmic Content:**
- **Application:** Recommendation Systems (e.g., Netflix or Amazon)

Function of Algorithms Involved:

1. **Collaborative Filtering Algorithms:** These algorithms analyze user behavior and preferences to recommend items based on what similar users liked. They involve matrix factorization techniques like Singular Value Decomposition (SVD) or alternatives like k-Nearest Neighbors (k-NN).
2. **Content-Based Filtering Algorithms:** These algorithms recommend items based on the characteristics of items the user has liked in the past.

They utilize algorithms like TF-IDF (Term Frequency-Inverse Document Frequency) and cosine similarity to find similar items.

3. **Hybrid Algorithms:** Many modern systems use a combination of collaborative and content-based filtering to improve recommendations. Algorithms like stacking, where predictions from different models are combined, are often used to enhance the accuracy of recommendations.

Overall, these algorithms analyze vast datasets of user interactions and item attributes to provide personalized recommendations, significantly impacting user engagement and satisfaction.

.....

(1.2-2)

Q: To find the values of n for which insertion sort beats merge sort, we need to determine when the time complexity of insertion sort is less than that of merge sort.

Given:

Insertion Sort: $8n^2$

Merge Sort: $64n \lg n$

We are looking for values of n such that:

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

Dividing both sides by $8n$ (assuming $n > 0$), we get:

$$n < 8 \lg n$$

Now we need to solve this inequality, n :

Approach:

We can approximate the values of n for which $n > 8 \lg n$ by testing different values of n and checking the inequality. However, let's also approach it theoretically to get a rough sense of where the inequality might hold:

- For small values of n (like 1 through 10), we can check explicitly.
- For larger values of n , since n grows faster than $\lg n$, there will be a point where n exceeds $8 \lg n$, so we'll search for that threshold.

Let's calculate and verify a few values of n to pinpoint this threshold. The inequality $n < 43$ holds for value of n up to **43**. Therefore, insertion sort outperforms merge sort for input sizes $n \leq 43$ on this machine. For $n > 43$, merge sort becomes more efficient.

.....

(1.2-3)

Q: What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

➤ To find the smallest value of n such that an algorithm with running time $100n^2$ runs faster than an algorithm with running time 2^n , we need to find the smallest integer n for which:

➤ $100n^2 < 2^n$

Lets break it down into simple steps:

STEP 1:

SET UP INEQUALITY:

We have given:

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

STEP 2:

CHECK SMALL VALUES OF (n) :

Since 2^n grows exponentially, it will eventually surpass $100n^2$, which grows quadratically. To find when this happens, we can substitute small integer values for n and calculate each side of the inequality until we find the smallest n that satisfies it.

1. For $n = 10$:

- $100n^2 = 100 \times 10^2 = 10000$
- $2^n = 2^{\{10\}} = 1024$
- Result: $10000 > 1024$ (does not satisfy the inequality)

2. For $n = 15$:

- $100n^2 = 100 \times 15^2 = 22500$
- $2^n = 2^{\{15\}} = 32768$
- Result: $22500 < 32768$ (satisfies the inequality)

3. For $n = 14$:

- $100n^2 = 100 \times 14^2 = 19600$
- $2^n = 2^{\{14\}} = 16384$
- Result: $19600 > 16384$ (does not satisfy the inequality)

Step 3:

Conclusion The smallest value of n that satisfies $100n^2 < 2^n$ is

$$\text{➤ } n = 15$$

So, $(n = 15)$ is the answer.

.....

CHAP 2:

GETTING STARTED

(2.1-1)

Q: Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on an array initially containing the sequence [31;41;59;26;41;58] .

For an array initially containing the sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, we want to illustrate the operation of **Insertion Sort** on this array.

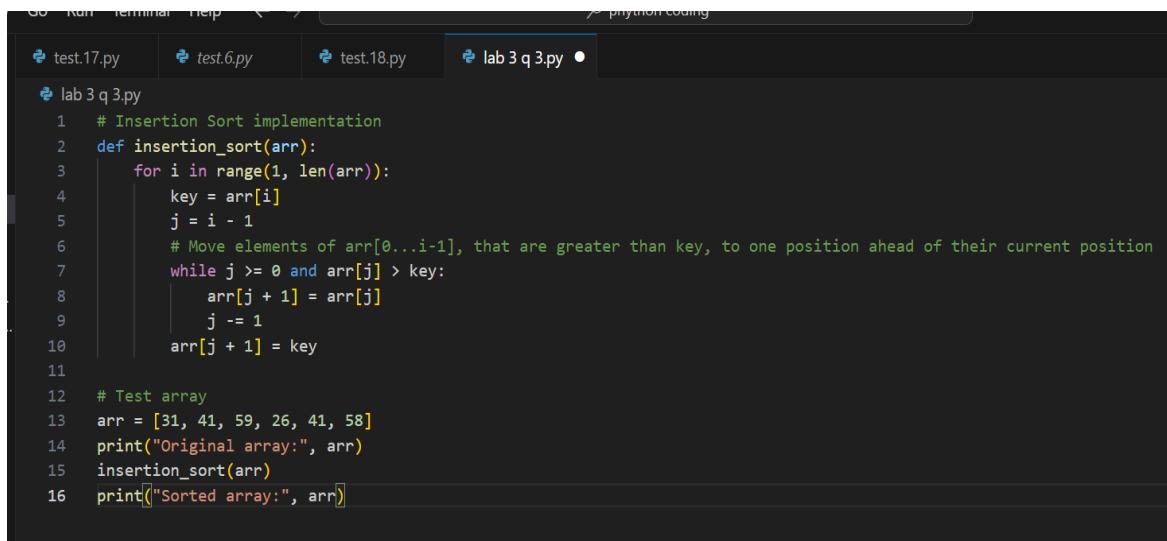
Here's a step-by-step demonstration of the Insertion Sort process for this sequence:

Initial Array: $\langle 31, 41, 59, 26, 41, 58 \rangle$.

- **Step 1:** The first element (31) is already sorted.
- **Step 2:** The second element (41) is already greater than 31, so it stays in place. Array: $\langle 31, 41, 59, 26, 41, 58 \rangle$.
- **Step 3:** The third element (59) is greater than 41, so it stays in place. Array: $\langle 31, 41, 59, 26, 41, 58 \rangle$.
- **Step 4:** The fourth element (26) needs to be inserted in the correct position. It's less than 59, 41, and 31. After shifting these elements, 26 is placed at the beginning:
 - Array after insertion: $\langle 26, 31, 41, 59, 41, 58 \rangle$
- **Step 5:** The fifth element (41) needs to be placed in order. It is less than 59 but greater than 31, so it's inserted in the fourth position:
 - Array after insertion: $\langle 26, 31, 41, 41, 59, 58 \rangle$

- **Step 6:** The sixth element (58) needs to be placed in order. It is less than 59 but greater than 41, so it's inserted in the fifth position:
 - Array after insertion: {26,31,41,41,58,59}

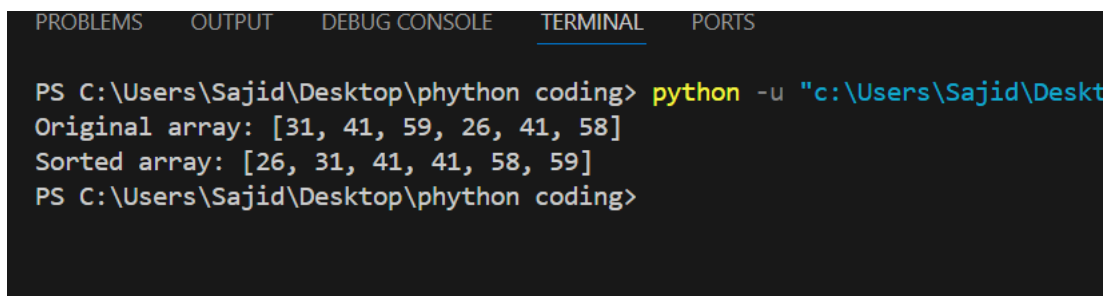
After these steps, the array is fully sorted: {26,31,41,41,58,59}



```

1  # Insertion Sort implementation
2  def insertion_sort(arr):
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 of their current position
7          while j >= 0 and arr[j] > key:
8              arr[j + 1] = arr[j]
9              j -= 1
10             arr[j + 1] = key
11
12 # Test array
13 arr = [31, 41, 59, 26, 41, 58]
14 print("Original array:", arr)
15 insertion_sort(arr)
16 print("Sorted array:", arr)
  
```

OUTPUT:



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Sajid\Desktop\python coding> python -u "c:\Users\Sajid\Desktop\python coding\lab 3 q 3.py"
Original array: [31, 41, 59, 26, 41, 58]
Sorted array: [26, 31, 41, 41, 58, 59]
PS C:\Users\Sajid\Desktop\python coding>
  
```

.....

(2.1-2)

Q: Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array $A[1:n]$. State a loop invariant for this

procedure, and use its initialization, maintenance, and termination properties to show that the SUM- ARRAY procedure returns the sum of the numbers in $A[1:n]$.

Loop Invariant for SUM-ARRAY

Procedure:

SUM-ARRAY computes the sum of n numbers in an array $A[1 \dots n]$.

To prove this, let's define a **loop invariant** and verify it through the steps of **initialization, maintenance, and termination**.

Loop Invariant

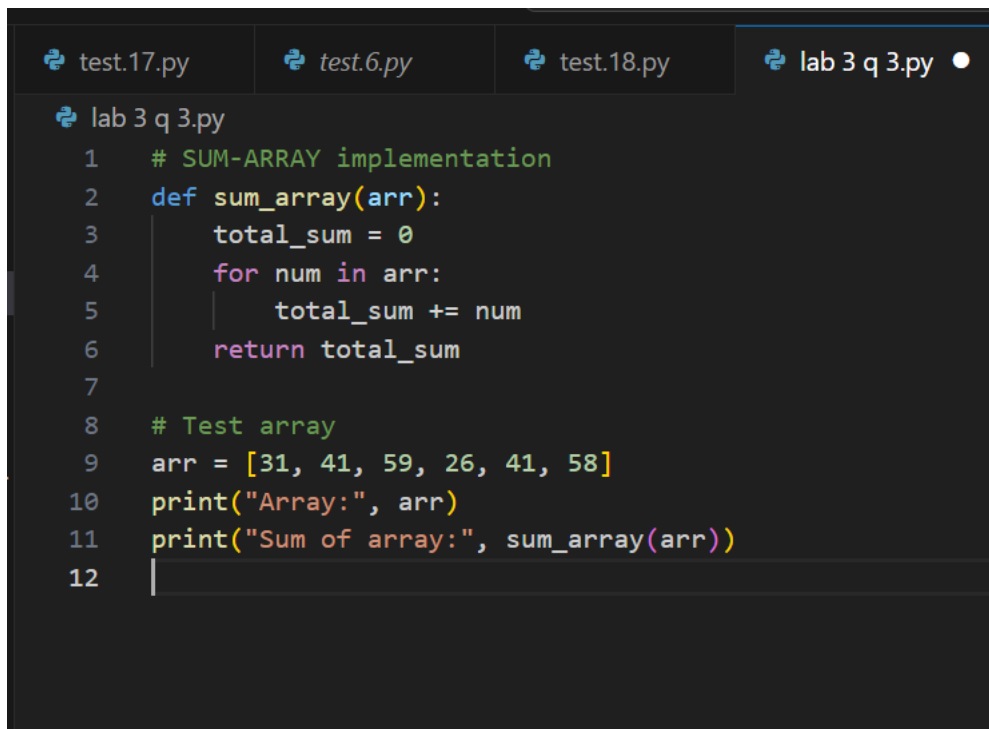
At the start of each iteration of the loop, the variable `sum` contains the sum of the elements from $A[1]$ to $A[I - 1]$.

Proof by Invariant:

1. **Initialization:** Before the loop starts, `sum` is initialized to 0, which is the sum of an empty subarray. So, the invariant holds at initialization.
2. **Maintenance:** On each iteration, the element $A[i]$ is added to `sum`. After adding $A[i]$ `sum` now holds the sum of $A[1]$ through $A[i]$, maintaining the invariant.

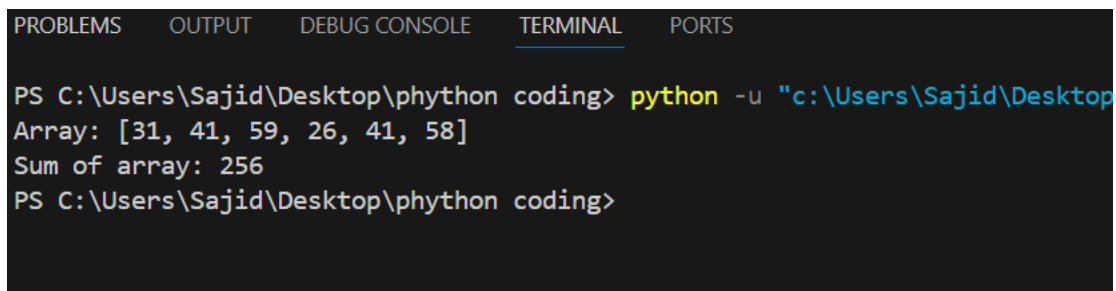
3. **Termination:** The loop terminates when $i=n+1$. By the loop invariant, at this point, sum contains the sum of all elements from $A[1]$ through $A[n]$. Thus, sum now holds the total sum of the array, which is returned by the procedure.

Therefore, the SUM-ARRAY procedure correctly computes and returns the sum of the numbers in $A[1 \dots n]$.



```
test.17.py test.6.py test.18.py lab 3 q 3.py ●
lab 3 q 3.py
1  # SUM-ARRAY implementation
2  def sum_array(arr):
3      total_sum = 0
4      for num in arr:
5          total_sum += num
6      return total_sum
7
8  # Test array
9  arr = [31, 41, 59, 26, 41, 58]
10 print("Array:", arr)
11 print("Sum of array:", sum_array(arr))
12
```

OUTPUT:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Sajid\Desktop\python coding> python -u "c:\Users\Sajid\Desktop\python coding\lab 3 q 3.py"
Array: [31, 41, 59, 26, 41, 58]
Sum of array: 256
PS C:\Users\Sajid\Desktop\python coding>
```

.....

(2.1-3)

Q: Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing in- stead of monotonically increasing order.

➤ Insertion Sort in Decreasing Order

To modify the **Insertion Sort** to sort in decreasing order instead of increasing order, we need to adjust the condition in the inner while loop. Instead of moving elements that are **greater than** the key, we now want to move elements that are **less than** the key.

```
test.17.py × test.6.py test.18.py lab 3 q 3.py •
lab 3 q 3.py
1 def insertion_sort_descending(arr):
2     for i in range(1, len(arr)):
3         key = arr[i]
4         j = i - 1
5         # Move elements of arr[0...i-1] that are less than key to one position ahead of their current position
6         while j >= 0 and arr[j] < key:
7             arr[j + 1] = arr[j]
8             j -= 1
9         arr[j + 1] = key
10
11 # Test array
12 arr = [31, 41, 59, 26, 41, 58]
13 print("Original array:", arr)
14 insertion_sort_descending(arr)
15 print("Sorted array in decreasing order:", arr)
16 |
```

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL FORKS
PS C:\Users\Sajid\Desktop\python coding> python -u "c:\Users\Sajid\Desktop\python coding\lab 3 q 3.py"
Original array: [31, 41, 59, 26, 41, 58]
Sorted array in decreasing order: [59, 58, 41, 41, 31, 26]
PS C:\Users\Sajid\Desktop\python coding>
```

.....

(2.1-4)

Consider the searching problem:

Input: A sequence of n numbers having $(a_1 ; a_2 ; \dots ; a_n)$ stored in array $A[1:n]$ and a value x .

Output: An index i such that x equals $A[i]$ or the special value NIL if x does not appear in A .

Write pseudocode for linear search, which scans through the array from beginning to end, looking for x . Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

ANSWER:

➤ **Linear Search**

The goal here is to write a linear search algorithm that finds the index of a specified value x in an array A . If x is not found, it should return NIL.

the pseudocode and Python implementation:

Pseudocode:

1. Set $i = 1$.

2. While $i \leq n$, do:
 - If $A[i]=x$, return i .
 - Otherwise, increment i .
3. If no match is found, return NIL.

```
test.17.py test.6.py test.18.py lab 3 q 3.py ●
lab 3 q 3.py
1 def linear_search(arr, x):
2     for i in range(len(arr)):
3         if arr[i] == x:
4             return i # Return index i if x is found
5     return None # Return None (NIL) if x is not found
6
7 # Test array and value to search
8 arr = [31, 41, 59, 26, 41, 58]
9 x = 41
10 result = linear_search(arr, x)
11 if result is not None:
12     print(f"Value {x} found at index {result}")
13 else:
14     print("Value not found")
15
```

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Sajid\Desktop\python coding> python -u "c:\Users\Sajid\Desktop\python coding\lab 3 q 3.py"
Value 41 found at index 1
PS C:\Users\Sajid\Desktop\python coding>
```

Proof by Loop Invariant:

Define the loop invariant as: "At the start of each iteration, none of the elements in the array from $A[1]$ to $A[i-1]$ is equal to x unless x has already been found."

1. **Initialization:** At the beginning ($i = 1$), no elements have been checked, so the invariant holds.
 2. **Maintenance:** On each iteration, if $A[i] \neq x$, the loop proceeds to the next index, preserving the invariant.
 3. **Termination:** The loop terminates either because x was found (in which case the function returns i), or all elements were checked, confirming that x is not in the array.
-

(2.1-5)

Q: Consider the problem of adding two n -bit binary integers a and b . Write a procedure **ADD-BINARY-INT that takes as input arrays A and B , along with the length n , and returns array C holding the sum.**

- Create a procedure that adds two n -bit binary numbers, stored in arrays A and B , and returns the result in a $(n+1)$ -element array.

➤ Adding Two n-bit Binary Integers

```
test.17.py test.6.py test.18.py lab 3 q 3.py •
lab 3 q 3.py
1 def add_binary_integers(A, B, n):
2     C = [0] * (n + 1) # Result array with n+1 elements
3     carry = 0
4
5     # Traverse from the least significant bit to the most significant
6     for i in range(n-1, -1, -1):
7         # Sum current bits of A and B and the carry
8         sum_bits = A[i] + B[i] + carry
9         C[i+1] = sum_bits % 2 # Current bit
10        carry = sum_bits // 2 # Update carry for the next bit
11
12    # The most significant bit is the final carry
13    C[0] = carry
14    return C
15
16    # Test with two 4-bit numbers
17    A = [1, 0, 1, 1] # Binary for 11
18    B = [1, 1, 0, 1] # Binary for 13
19    n = 4
20    C = add_binary_integers(A, B, n)
21    print("Binary sum:", C)
22
```

➤ OUTPUT:

```
7 # Sum current bits of A and B and the carry
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Sajid\Desktop\pyhton coding> python -u "c:\Users\Sajid\Desktop\pyhton
Binary sum: [1, 1, 0, 0, 0]
PS C:\Users\Sajid\Desktop\pyhton coding>
```

.....

(2.2-1)

Q: Express the function $F(n)=n^3+100n^2+100n+3$ in terms of , -notation.

➤ **Express the Function in Terms of Big-O Notation:**

The function is :

$$F(n)=n^3+100n^2+100n+3$$

- TO express this in big-O Notation (O) we focus on the highest order term because it dominates the growth rates as n to infinity.
- The highest order term is n^3 .
- Thus $f(n)=O(n^3)$.

So the answer is :

$$F(n)=O(n^3).$$

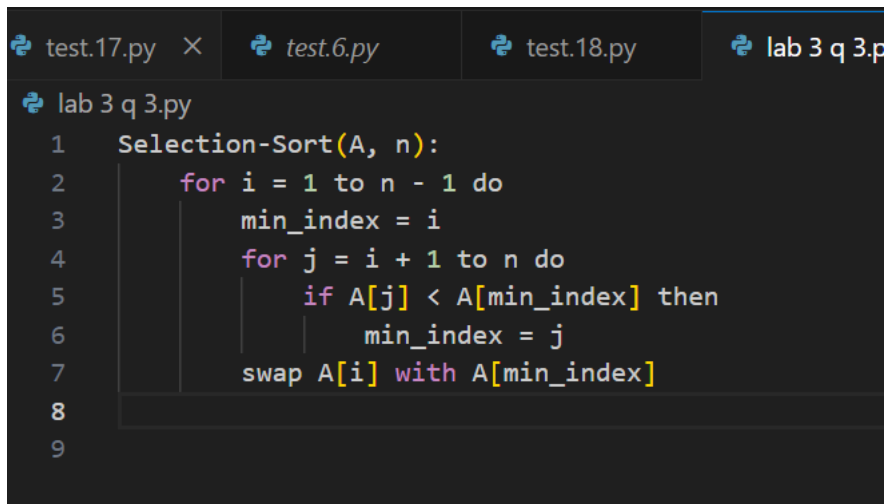
.....

(2.2-2)

Q:Consider sorting n numbers stored in array..... Write pseudocode for this algorithm, which is known as selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only $n-1$ elements, rather than for all n elements? Give the worst-case running time of selection sort in , -notation. Is the best-case running time any better?

➤ Selection Sort Algorithm

Selection Sort sorts an array $A[1 \dots n]$ by finding the smallest element in the unsorted portion of the array and swapping it with the first unsorted element.



```
test.17.py × test.6.py test.18.py lab 3 q 3.p
lab 3 q 3.py
1  Selection-Sort(A, n):
2      for i = 1 to n - 1 do
3          min_index = i
4          for j = i + 1 to n do
5              if A[j] < A[min_index] then
6                  min_index = j
7          swap A[i] with A[min_index]
8
9
```

Loop Invariant:

The loop invariant for the outer loop (for $i = 1$ to $n - 1$) is:

At the start of each iteration of the outer loop, the subarray $A[1 \dots i-1]$ is sorted, and every element in $A[1 \dots i-1]$ is smaller than or equal to every element in $A[i \dots n]$.

Why Only $n-1$ Elements?

We only need to iterate through the first $n-1$ elements because by the time we reach the last element, the array is already sorted. The last element will automatically be the largest remaining unsorted element.

Worst-Case Time Complexity

In the worst case, Selection Sort requires comparing each element with every other element in the array. This results in $O(n^2)$ comparisons. So the worst-case time complexity is $O(n^2)$.

Best-Case Time Complexity

Since Selection Sort always goes through the same number of comparisons regardless of the initial order of elements, the best-case time complexity is also $O(n^2)$.

.....

(2.2-3)

Q: Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case?

➤ Average-Case and Worst-Case for Linear Search

In linear search, we are searching for a target x in an array $A[1 \dots n]$ and returning the index if found, or NIL if it is not present.

Average-Case Analysis:

If the element being searched for is equally likely to be any element in the array, on average, we expect to search half the array before finding the target. So the average number of comparisons is $n/2$. Thus, the average-case time complexity is:

$$O(n)$$

Worst-Case Analysis:

In the worst case, the element we are looking for is either the last element or is not present at all. This requires checking all n elements, giving a worst-case time complexity of:

$$O(n)$$

.....

(2.2-4)

Q:How can you modify any sorting algorithm to have a good best-case running time?

➤ Improving Best-Case Running Time for Sorting Algorithms

To improve the best-case running time of any sorting algorithm, we can add a preliminary check to see if the array is already sorted. If the array is sorted, we can return immediately, achieving a best-case time complexity of $O(n)$ for algorithms that would otherwise have a higher best-case time.

Modification

1. Add a check at the beginning of the sorting algorithm to iterate through the array once to see if it is already sorted.
2. If the array is sorted, terminate and return the array immediately.
3. If the array is not sorted, proceed with the sorting algorithm.

This optimization allows any sorting algorithm to have a best-case running time of $O(n)$. However, the average-case and worst-case running times remain unaffected.

.....

(2.3-1)

Q: 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)

Q: The test in line 1 of the MERGE-SORT procedure reads

**$r \dots\dots\dots$, then is empty. Argue that as long as the initial call of
MERGE-SORT $\dots\dots\dots$ the test $r \leq p$.**

ANSWER:

- 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 $p \neq r$ 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 $p \neq r$) is sufficient, ensuring $p \leq r$ throughout the recursion.

.....

(2.3-3)

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.

```
test.17.py test.6.py test.18.py lab 3 q 3.py •
lab 3 q 3.py
1 def merge(L, R):
2     # Initialize indices for L, R, and A
3     i, j, k = 0, 0, 0
4     A = [0] * (len(L) + len(R)) # Output array A with enough space for both L and R
5
6     # Invariant for lines 12-18:
7     # A[0...k-1] is sorted and contains the smallest elements from L and R up to that point.
8     while i < len(L) and j < len(R):
9         if L[i] <= R[j]:
10             A[k] = L[i]
11             i += 1
12         else:
13             A[k] = R[j]
14             j += 1
15             k += 1
16
17     # Invariant for lines 20-23:
18     # All remaining elements of L are greater than or equal to A[0...k-1]
19     while i < len(L):
20         A[k] = L[i]
21         i += 1
22         k += 1
23
24     # Invariant for lines 24-27:
25     # All remaining elements of R are greater than or equal to A[0...k-1]
26     while j < len(R):
27         A[k] = R[j]
28         j += 1
29         k += 1
30
31     return A
```

Explanation with Invariants

- **Lines 12–18:** The first while loop merges the elements from L and R in sorted order, ensuring that $A[0 \dots k-1]$ 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)

Use mathematical induction to show that when $n \geq 2$ is an exact power of 2 , the (solution of the recurrence $T(n) = \begin{cases} 2T(n/2) & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases}$ is $T(n) = n \lg n$.

ANSWER:

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

$$T(n) = \begin{cases} 2T(n/2) & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases}$$

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)

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.

ANSWER:

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)

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)

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:

- 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:
 - Finding the position using binary search: $O(\log j)$
 - 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)

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.

ANSWER:

- 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 SSS:** Sorting take $O(n \log n)$ 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 = $S[\text{left}] + S[\text{right}]$

 if sum == x :

 return ($S[\text{left}], S[\text{right}]$) // 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.

CHAP 3:

CHARACTERIZING RUNNING TIMES

(Exercises 3.1-1)

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

Solution Outline:

The lower-bound argument for insertion sort uses comparisons to sort the list, and the argument involves grouping elements into sections (originally in multiples of 3) to calculate the number of comparisons required to sort the entire list.

1. Let's assume the array has n elements, which is not necessarily a multiple of 3. We can divide it into groups of 3 as much as possible, with a smaller group of fewer than 3 elements if n is not divisible by 3.
2. If $n=3k+r$ for integer k and remainder r (where r can be 0, 1, or 2), we have:
 - K groups of size 3 and
 - one additional group of size r if $r \neq 0$.

3. The number of comparisons in insertion sort would still involve comparing each element to the preceding elements to find its correct position. The lower-bound argument should thus take into account the number of comparisons needed for each group.

The modification in the argument should address the final, smaller group if $r \neq 0$, but this adjustment doesn't fundamentally change the $\Omega(n^2)$ lower bound for insertion sort.

.....

(3.1-2)

Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.

Solution Outline:

Selection sort works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the beginning of the sorted portion.

1. **Inner Loop:** The inner loop finds the minimum element in the unsorted part of the list. For an array of n elements:
 - On the first pass, it makes $n-1$ comparisons.

- On the second pass, it makes $n-2$ comparisons, and so on.
- On the last pass, it makes 1 comparison.

2. **Total Comparisons:** The total number of comparisons for selection sort is given by:

$$(n-1)+(n-2)+\dots+1=n(n-1)/2=\Theta(n^2)$$

Swaps: Selection sort performs exactly $n-1$ swaps, as each element is moved to its correct position exactly once.

Therefore, the running time of selection sort is $\Theta(n^2)$ due to the quadratic number of comparisons required.

.....

(3.1-3)

Suppose that α is a fraction in the range $0 < \alpha < 1$. 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. What additional restriction do you need to put on α ? What value of α maximizes the number of times that the αn largest values must pass through each of the middle $(1-2\alpha)n$ array positions?

Solution Outline:

1. **Setup:** Suppose that the αn largest elements are in the first αn positions of the array.
2. **Condition on α :** For insertion sort to maintain its comparisons, we need that α does not cover the entire array, so $0 < \alpha < 1$.
3. **Movement of Elements:** As each of the αn largest elements moves to its correct position, it must pass through the remaining $(1 - \alpha)n$ positions.
4. **Maximizing Crossings:** The value of α that maximizes the movement is typically when α is around 0.5, as this will create the most movement between the two halves of the list.

.....