**Data Structure and Algorithm**



Session: 2022 – 2026

**Submitted by:**

Syed Muhammad Faisal Ilyas 2022-CS-63

**Submitted to:**

Sir Nazeef ul Haq

Department of Computer Science

**University of Engineering and Technology**

**Lahore Pakistan**

**Question 1**

Dry running the Insertion Sort algorithm on both input arrays A and B showing the detailed iterations:

Array A = [5, 43, 76, 2, 98, 23, 12, 32]:

Iteration 1

At i = 1

Initial State: [5, 43, 76, 2, 98, 23, 12, 32]

Key: 43

Compare 43 with 5. No change.

Current State: [5, 43, 76, 2, 98, 23, 12, 32]

Iteration 2:

At i=2

Initial State: [5, 43, 76, 2, 98, 23, 12, 32]

Key: 76

Compare 76 with 43. No change.

Current State: [5, 43, 76, 2, 98, 23, 12, 32]

Iteration 3:

At i = 3

Initial State: [5, 43, 76, 2, 98, 23, 12, 32]

Key: 2

Compare 2 with 76. Move 76 to the right.

Current State: [5, 43, 2, 76, 98, 23, 12, 32]

Compare 2 with 43. Move 43 to the right.

Current State: [5, 2, 43, 76, 98, 23, 12, 32]

Compare 2 with 5. Move 5 to the right.

Current State: [2, 5, 43, 76, 98, 23, 12, 32]

Iteration 4:

At i=4

Initial State: [2, 5, 43, 76, 98, 23, 12, 32]

Key: 98

Compare 98 with 76. No change.

Current State: [2, 5, 43, 76, 98, 23, 12, 32]

Iteration 5:

At i=5

Initial State: [2, 5, 43, 76, 98, 23, 12, 32]

Key: 23

Compare 23 with 98. Move 98 to the right.

Current State: [2, 5, 43, 76, 23, 98, 12, 32]

Compare 23 with 76. Move 76 to the right.

Current State: [2, 5, 43, 23, 76, 98, 12, 32]

Compare 23 with 43. Move 43 to the right.

Current State: [2, 5, 23, 43, 76, 98, 12, 32]

Iteration 6:

At i = 6

Initial State: [2, 5, 23, 43, 76, 98, 12, 32]

Key: 12

Compare 12 with 98. Move 98 to the right.

Current State: [2, 5, 23, 43, 76, 12, 98, 32]

Compare 12 with 76. Move 76 to the right.

Current State: [2, 5, 23, 43, 12, 76, 98, 32]

Compare 12 with 43. Move 43 to the right.

Current State: [2, 5, 23, 12, 43, 76, 98, 32]

Compare 12 with 23. Move 23 to the right.

Current State: [2, 5, 12, 23, 43, 76, 98, 32]

Iteration 7

At i = 7

Initial State: [2, 5, 12, 23, 43, 76, 98, 32]

Key: 32

Compare 32 with 98. Move 98 to the right.

Current State: [2, 5, 12, 23, 43, 76, 32, 98]

Compare 32 with 76. Move 76 to the right.

Current State: [2, 5, 12, 23, 43, 32, 76, 98]

Compare 32 with 43. Move 43 to the right.

Current State: [2, 5, 12, 23, 32, 43, 76, 98]

Array A is now fully sorted: A= [2, 5, 12, 23, 32, 43, 76, 98]

Array B = [6, 7, 8, 9, 10]:

Since B is already sorted so there will be no swapping but program will execute.

Iteration 1 (i = 1): Initial state: [6, 7, 8, 9, 10]

Iteration 2 (i = 2): Initial state: [6, 7, 8, 9, 10]

Iteration 3 (i = 3): Initial state: [6, 7, 8, 9, 10]

Iteration 4 (i = 4): Initial state: [6, 7, 8, 9, 10]

Array B remains fully sorted: [6, 7, 8, 9, 10].

**Question 2:**

Initial Array: [1, 3, 54, 2, 72, 23, 12, 32, 76, 12]

**Step 1: Splitting into Subarrays**

Initial call to MergeSort: MergeSort([1, 3, 54, 2, 72, 23, 12, 32, 76, 12])

Splitting the array into two halves:

Left half: [1, 3, 54, 2, 72]

Right half: [23, 12, 32, 76, 12]

**Step 2: Recursive Sorting**

Recursive call on the left half:

MergeSort([1, 3, 54, 2, 72])

Splitting into two halves:

Left half: [1, 3]

Right half: [54, 2, 72]

Recursive call on the left half:

MergeSort([1, 3])

This is a sorted array; no further splitting is needed.

Recursive call on the right half:

MergeSort([54, 2, 72])

Splitting into two halves:

Left half: [54]

Right half: [2, 72]

Recursive call on the left half:

MergeSort([54])

This is a sorted array; no further splitting is needed.

Recursive call on the right half:

MergeSort([2, 72])

Splitting into two halves:

Left half: [2]

Right half: [72]

Recursive call on the left half:

MergeSort([2])

This is a sorted array; no further splitting is needed.

Recursive call on the right half:

MergeSort([72])

This is a sorted array; no further splitting is needed.

Merging the left and right halves: [2, 54, 72]

Merging the left and right halves: [1, 2, 3, 54, 72]

Recursive call on the right half:

MergeSort([23, 12, 32, 76, 12])

Splitting into two halves:

Left half: [23, 12]

Right half: [32, 76, 12]

Recursive call on the left half:

MergeSort([23, 12])

Splitting into two halves:

Left half: [23]

Right half: [12]

Recursive call on the left half:

MergeSort([23])

This is a sorted array; no further splitting is needed.

Recursive call on the right half:

MergeSort([12])

This is a sorted array; no further splitting is needed.

Merging the left and right halves: [12, 23]

Recursive call on the right half:

MergeSort([32, 76, 12])

Splitting into two halves:

Left half: [32]

Right half: [76, 12]

Recursive call on the left half:

MergeSort([32])

This is a sorted array; no further splitting is needed.

Recursive call on the right half:

MergeSort([76, 12])

Splitting into two halves:

Left half: [76]

Right half: [12]

Recursive call on the left half:

MergeSort([76])

This is a sorted array; no further splitting is needed.

Recursive call on the right half:

MergeSort([12])

This is a sorted array; no further splitting is needed.

Merging the left and right halves: [12, 76]

Merging the left and right halves: [12, 32, 76]

Merging the left and right halves: [12, 23, 32, 76]

**Step 3: Merging All Subarrays**

Merging the left and right halves of the original array:

Left half: [1, 2, 3, 54, 72]

Right half: [12, 23, 32, 76, 12]

Merging the left and right halves: [1, 2, 3, 12, 12, 23, 32, 54, 72, 76]

Array A is now fully sorted: [1, 2, 3, 12, 12, 23, 32, 54, 72, 76]

**Question 4:**

MergeSort usually takes about the same time for both sorted and unsorted lists. While it can be a bit faster in a sorted list, the main advantage of MergeSort is that it is a dependable sorting method that works well no matter how the list starts out, ensuring a consistent and predictable performance. MergeSort's main advantage lies in its reliability and steady performance, making it a practical choice for sorting lists, no matter how they're arranged initially but time complexity remains same.

**Question 5:**

In Merge Sort, loops are used for the merging step, where two sorted subarrays are combined into a single sorted subarray. Let's explore how loops work in the context of Merge Sort:

**1. Comparison and Merging Loop:**

In the merging step, you have two sorted subarrays: the left subarray and the right subarray by dividing the array into half and recursively call this function until one element is left.

**2. Element Comparison:**

Within the loop, you compare the elements at the front of both subarrays.

If the element from the left subarray is greater (for descending order), add it to the merged subarray and remove it from the left subarray.

If the element from the right subarray is greater, add it to the merged subarray and remove it from the right subarray.

This process continues until one of the subarrays becomes empty.

**3. Remaining Elements:**

After one of the subarrays becomes empty, check if there are any remaining elements in the other subarray.

If there are, simply add all the remaining elements from that subarray to the merged subarray.

**4. Completion:**

Once the loop finishes, arrays have been successfully merged a single sorted subarray in descending order.

**5. Recursion:**

The merge operation is performed recursively for all levels of the recursion stack until the entire array is sorted.

**Question 6:**

Comparison Selection Sort and Insertion Sort:

**Selection Sort:**

* Selection Approach: It repeatedly selects the smallest (or largest) element from the unsorted part and places it at the beginning of the sorted part.
* Stability: It is not a stable sorting algorithm, which means the relative order of equal elements may change after sorting.
* Time Complexity: In all cases (worst, average, and best), it has a time complexity of O(n^2), making it less efficient for large datasets.
* Use Cases: It is suitable for small datasets or when simplicity is more important than sorting speed.

**Insertion Sort:**

* Insertion Approach: It iterates through the array, taking each element from the unsorted part and inserting it into its correct position in the sorted part.
* Stability: It is a stable sorting algorithm, ensuring that the relative order of equal elements remains unchanged.
* Time Complexity: In the worst and average cases, it has a time complexity of O(n^2). However, in the best case scenario (when the array is nearly sorted), it can be as efficient as O(n).
* Use Cases: It is suitable for small datasets, and it performs well when dealing with data that is already partially sorted.

**Question 7:**

Loop Invariant in Selection Sort:

**Initialization:** Before the sorting process begins, the loop invariant holds true because at the start, considering the entire array as unsorted.

**Maintenance:** During each iteration of the Selection Sort algorithm, the loop invariant is maintained. This means that after each iteration, the smallest element from the unsorted part is moved to the sorted part.

**Termination:** When the loop finishes its iterations, at this point, the sorted part contains all the elements, and they are in their correct sorted order, while the unsorted part is empty.

**Question 8:**

Bubble Sort is a simple sorting method that works by repeatedly comparing adjacent elements in a list and swapping them if they are in the wrong order. This process is repeated until no more swaps are needed, indicating that the list is fully sorted. While Bubble Sort is easy to understand and implement, it tends to be inefficient for large lists, as it has a time complexity of O(n^2), making it less suitable for larger datasets compared to more efficient sorting algorithms like Merge Sort or Quick Sort.

**Question 9:**

**1. Initialization:** Start by considering the entire array as unsorted.

**2. Iterating Through the Array:** Begin a loop that goes through the array repeatedly. In each pass through the loop, we compare adjacent elements in the array and swap them if they are in the wrong order.

**3. Comparing Elements:** During each iteration, compare each element with the one next to it. If the current element is greater than the next one, we swap them.

**4. Repeat:** Continue this process for each element in the array, making multiple swaping if necessary.

**5. Completion:** After all passes, the largest element is at the end of the array. Repeat the process for the

**Question 10**

Dry run the Bubble Sort algorithm on the input array A = {9, 8, 7, 6, 4, 3, 2, 1} step by step:

Initial Array: [9, 8, 7, 6, 4, 3, 2, 1]

**Iteration 1:**

Compare and swap 9 and 8: [8, 9, 7, 6, 4, 3, 2, 1]

Compare and swap 9 and 7: [8, 7, 9, 6, 4, 3, 2, 1]

Compare and swap 9 and 6: [8, 7, 6, 9, 4, 3, 2, 1]

Compare and swap 9 and 4: [8, 7, 6, 4, 9, 3, 2, 1]

Compare and swap 9 and 3: [8, 7, 6, 4, 3, 9, 2, 1]

Compare and swap 9 and 2: [8, 7, 6, 4, 3, 2, 9, 1]

Compare and swap 9 and 1: [8, 7, 6, 4, 3, 2, 1, 9]

**Iteration 2:**

Compare and swap 8 and 7: [7, 8, 6, 4, 3, 2, 1, 9]

Compare and swap 9 and 6: [7, 8, 6, 4, 3, 2, 1, 9]

Compare and swap 9 and 4: [7, 8, 6, 4, 3, 2, 1, 9]

Compare and swap 9 and 3: [7, 8, 6, 4, 3, 2, 1, 9]

Compare and swap 9 and 2: [7, 8, 6, 4, 3, 2, 1, 9]

Compare and swap 9 and 1: [7, 8, 6, 4, 3, 2, 1, 9]

**Iteration 3:**

Compare and swap 8 and 7: [7, 6, 8, 4, 3, 2, 1, 9]

Compare and swap 8 and 6: [7, 6, 8, 4, 3, 2, 1, 9]

Compare and swap 9 and 4: [7, 6, 8, 4, 3, 2, 1, 9]

Compare and swap 9 and 3: [7, 6, 8, 4, 3, 2, 1, 9]

Compare and swap 9 and 2: [7, 6, 8, 4, 3, 2, 1, 9]

**Iteration 4:**

Compare and swap 7 and 6: [6, 7, 8, 4, 3, 2, 1, 9]

Compare and swap 8 and 4: [6, 7, 4, 8, 3, 2, 1, 9]

Compare and swap 9 and 3: [6, 7, 4, 8, 3, 2, 1, 9]

Compare and swap 9 and 2: [6, 7, 4, 8, 3, 2, 1, 9]

**Iteration 5:**

Compare and swap 7 and 6: [6, 7, 4, 8, 3, 2, 1, 9]

Compare and swap 8 and 4: [6, 7, 4, 8, 3, 2, 1, 9]

Compare and swap 9 and 3: [6, 7, 4, 8, 3, 2, 1, 9]

**Iteration 6:**

Compare and swap 7 and 6: [6, 4, 7, 8, 3, 2, 1, 9]

Compare and swap 8 and 4: [6, 4, 7, 8, 3, 2, 1, 9]

**Iteration 7:**

Compare and swap 7 and 4: [6, 4, 7, 8, 3, 2, 1, 9]

**Iteration 8:**

Compare and swap 6 and 4: [4, 6, 7, 8, 3, 2, 1, 9]

Final Result: After eight iterations, the array is fully sorted in ascending order:

[1, 2, 3, 4, 6, 7, 8, 9]