**Data Structure and Algorithms**



Session: 2022 – 2026

**Submitted by:**

Gul-e-Zahra 2022-CS-75

**Submitted to:**

Sir Nazeef Ul Haq

Department of Computer Science

**University of Engineering and Technology**

**Lahore Pakistan**

**Question 1: Insertion Sort:**

For this array, two loops will be implemented one is outer loop that iterates on the entire length of array and second is inner loop that iterates one times less than the outer loop. The inner loop will iterate for 7 times because the length of array A is 8. So it will iterate for seven times because first element is supposed to be already sorted.

**For array A:**

In 1st iteration, key value is 5 and the Key is 43. As 43>5 so value of key will remain same and there will be no shifting. In 2nd iteration, key value is 43 and element to be compared is 76. As 75>45 and 75>5 so no shifting will occur. In 3rd iteration, key value is 76 and Key is 2. Since 2 is less than 75 so it will shift to the index of 2 in array. By comparing it to 45, it appears to be small and again it will be shift to the index of larger value until it is sorted. In 4th 5th 6th and 7th iteration same process will go on until entire array is sorted.

Input A: {5, 43, 76, 2, 98, 23, 12, 32}

Input B: {6, 7, 8, 9, 10}

**Iteration 1:**

Sorted List: [5]

Key: 43

Compare 43 with 5, it is greater, so insert 43 after 5.

Sorted List: [5, 43]

**Iteration 2:**

Sorted List: [5, 43]

Key: 76

Compare 76 with 43, it is greater, so insert 76 after 43.

Sorted List: [5, 43, 76]

**Iteration 3:**

Sorted List: [5, 43, 76]

Key: 2

Compare 2 with 76, it is smaller, so continue comparing with 43 and 5.

Compare 2 with 43, it is smaller, so continue comparing with 5.

Compare 2 with 5, it is smaller, so insert 2 at the beginning.

Sorted List: [2, 5, 43, 76]

**Iteration 4:**

Sorted List: [2, 5, 43, 76]

Key: 98

Compare 98 with 76, it is greater, so insert 98 after 76.

Sorted List: [2, 5, 43, 76, 98]

**Iteration 5:**

Sorted List: [2, 5, 43, 76, 98]

Key: 23

Compare 23 with 98, it is smaller, so continue comparing with 76, 43, and 5.

Compare 23 with 76, it is smaller, so continue comparing with 43 and 5.

Compare 23 with 43, it is smaller, so insert 23 after 5.

Sorted List: [2, 5, 23, 43, 76, 98]

**Iteration 6:**

Sorted List: [2, 5, 23, 43, 76, 98]

Key: 12

Compare 12 with 98, it is smaller, so continue comparing with 76, 43, 23, and 5.

Compare 12 with 76, it is smaller, so continue comparing with 43, 23, and 5.

Compare 12 with 43, it is smaller, so continue comparing with 23 and 5.

Compare 12 with 23, it is smaller, so insert 12 after 5.

Sorted List: [2, 5, 12, 23, 43, 76, 98]

**Iteration 7:**

Sorted List: [2, 5, 12, 23, 43, 76, 98]

Key: 32

Compare 32 with 98, it is smaller, so continue comparing with 76, 43, 23, 12, and 5.

Compare 32 with 76, it is smaller, so continue comparing with 43, 23, 12, and 5.

Compare 32 with 43, it is smaller, so continue comparing with 23, 12, and 5.

Compare 32 with 23, it is greater, so insert 32 after 23.

Sorted List: [2, 5, 12, 23, 32, 43, 76, 98]

Now, all elements from input A are sorted.

**For array B:**

B=[6,7,8,9,10] is already sorted but loop iterates on it whether no shifting is needed.

**Iteration 1:**

Sorted List B: [6]

Key: 7

Compare 7 with 6, it's greater, so insert 7 after 6. No shifts are needed

Sorted List B: [6, 7]

**Iteration 2:**

Sorted List B: [6, 7]

Key: 8

Compare 8 with 7, it's greater, so insert 8 after 7. No shifts are needed

Sorted List B: [6, 7, 8]

**Iteration 3:**

Sorted List B: [6, 7, 8]

Key: 9

Compare 9 with 8, it's greater, so insert 9 after 8. No shifts are needed

Sorted List B: [6, 7, 8, 9]

**Iteration 4:**

Sorted List B: [6, 7, 8, 9]

Key: 10

Compare 10 with 9, it's greater, so insert 10 after 9. No shifts are needed

Sorted List B: [6, 7, 8, 9, 10]

Now, all elements from input B are sorted.

**Question 2: Merge Sort:**

Detailed execution of the Merge Sort algorithm for the input array A = [1, 3, 54, 2, 72, 23, 12, 32, 76, 12]:

**Step 1:**

Initial Call

mergeSort(A)

**Step 2:**

Array mergeSort (Recursion)

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

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

**Step 3: Recursive Calls**

Recursive call for the left subarray: mergeSort([1, 3, 54, 2, 72])

Subarray Splitting:

Left Subarray: [1, 3]

Right Subarray: [54, 2, 72]

Recursive call for the left sub-subarray: mergeSort([1, 3])

Both sub-subarrays are considered sorted.

Merging: [1, 3]

Recursive call for the right sub-subarray: mergeSort([54, 2, 72])

Subarray Splitting:

Left Subarray: [54]

Right Subarray: [2, 72]

Recursive call for further spliting: mergeSort([54])

The it has only one element.

Merging: [54]

Recursive call for the right sub-sub-subarray: mergeSort([2, 72])

Both arrays are considered sorted.

Merging: [2, 72]

Merging: [2, 54, 72]

Merging: [1, 2, 3, 54, 72]

Merging: [1, 2, 3, 54, 72]

**Step 4:**

Recursive call for the right subarray: mergeSort([23, 12, 32, 76, 12])

Subarray Splitting:

Left Subarray: [23, 12]

Right Subarray: [32, 76, 12]

Recursive call for the left sub-subarray: mergeSort([23, 12])

Both sub-subarrays are considered sorted.

Merging: [12, 23]

Recursive call for the right sub-subarray: mergeSort([32, 76, 12])

Subarray Splitting:

Left Subarray: [32]

Right Subarray: [76, 12]

Recursive call for the left sub-sub-subarray: mergeSort([32])

The left sub-sub-subarray has only one element.

Merging: [32]

Recursive call for the right sub-sub-subarray: mergeSort([76, 12])

Both sub-sub-subarrays are considered sorted.

Merging: [12, 76]

Merging: [12, 32, 76]

Merging: [12, 12, 23, 32, 76]

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

**Step 5:** **Final Merge**

Final mergeArray: [1, 2, 3, 12, 12, 23, 32, 54, 72, 76]

The Merge Sort algorithm divides, sorts, and merges the subarrays to produce the final sorted array.

**Question 4:**

Merge Sort usually takes the same amount of time for both sorted and unsorted lists in terms of big O notation (O(n log n)). However, in practice, it can be faster for already sorted lists because it optimizes by avoiding some unnecessary operations like comparisons and data movement (swaps) that are not needed when sorting an already partially sorted or sorted array.

**Question 5:**

The modified pseudocode for Merge Sort in descending order:

**1. mergeSort(arr):**

If the length of the input array `arr` is 1 or less, return `arr` (it's already sorted).

Find the middle index of the array `arr` and split the array into two parts: `left` and `right`, using the middle index. mergerSort(arr) recursively call itself to divide the array upto single problem.

**2. merge(left, right):**

Create an empty array called `result` to store the merged elements.

Initialize two variables, leftIndex and rightIndex, to handle the indices in the left and right

subarrays.

**3. While loop (leftIndex < length(left) and rightIndex < length(right)):**

Compare elements at left[leftIndex] and right[rightIndex].

If the element in the left subarray is greater than the element in the right subarray, append the left element to the result, and increment leftIndex.

**4.** After the loop, check if there are any remaining elements in the left or right subarrays. If there are, append them to the `result`.

**5.** Return the result array, which contains the merged and sorted elements in descending order.

This modified pseudocode ensures that the Merge Sort algorithm arranges elements in descending order during the merge step.

**Question 6:**

Both Selection Sort and Insertion Sort are simple sorting algorithms with quadratic time complexities. Selection Sort repeatedly selects the smallest (or largest) element and moves it to its correct position, while Insertion Sort takes each element and inserts it into its correct position. Insertion Sort has an advantage when dealing with nearly sorted data, but neither algorithm is the most efficient choice for large lists or arrays.

**Question 7:**

**Loop invariant for selection sort algorithm**

**1. Initialization:**

At the start (before the first iteration), the sorted subarray is empty because no elements are sorted yet.

**2. Maintenance:**

After each iteration of the outer loop, the algorithm identifies the smallest element in the unsorted part and places it in its correct position in the sorted subarray.

This ensures that the sorted subarray contains the smallest “i” elements after the “i”th iteration.

**3. Termination:**

When the outer loop completes all iterations (when `i` reaches n, where `n` is the array length), the entire array is sorted because all elements are correctly placed.

In simple terms, the loop invariant guarantees that Selection Sort gradually builds a sorted subarray by repeatedly selecting the smallest remaining element and moving it to its proper place.

**Question 8:**

Bubble Sort is a basic sorting algorithm that works by repeatedly checking pairs of adjacent elements in a list and swapping them if they are in the wrong order. It continues this process until no more swaps are needed, signifying that the list is sorted. While Bubble Sort is easy to understand, it tends to be slow for large lists because it may require many comparisons and swaps, resulting in a time complexity of O(n^2) in the worst case. Consequently, it is not used for sorting large datasets efficiently.

**Question 9:**

1. Outer loop is executed form i=0 to i=end of the array.
2. Inner loop iterates form j=0 to length of array j=ni.
3. It compares adjacent elements arr[j] and arr[j+1] where j is the loop variable of the inner loop.
4. If arr[j] is greater than arr[j+1], it swaps the elements to move the larger element to the right.
5. This comparison and swapping process continues for each adjacent pair of elements within the unsorted portion of the list.

**Question 10:**

Dry run the Bubble Sort algorithm on the input array A = [9, 8, 7, 6, 4, 3, 2, 1]

A **=** [9, 8, 7, 6, 4, 3, 2, 1]

**Iteration no 1**

The outer loop starts with i = 0.

The inner loop iterates through the array, comparing and swapping adjacent elements:

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

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

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

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

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

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

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

After the first pass, the largest element (9) is in its correct position at the end of the array.

**Iteration from i=1 to the i=n1**

This process repeats for each pass until the entire array is sorted.

**Final Sorted Array:**

After all passes, the array is fully sorted: [1, 2, 3, 4, 6, 7, 8, 9]

Bubble Sort repeatedly compares and swaps adjacent elements until the largest elements move to the end of the array, and this process is repeated for each element in the array until the entire array is sorted.