**Prob Set 1:**

**Khuram iqbal (2022-CS-48)**

**Insertion Sort:**

**Question 1a:**

**Step 1:** Begin with the second element (43) and compare it to the first element (5). As 43 is greater than 5, no swapping is required, and the array remains unchanged.

Array after the 1st pass: {5, 43, 76, 2, 98, 23, 12, 32}

**Step 2:** Move on to the third element (76) and compare it with the second element (43). Since 76 is greater than 43, there's no need to swap, and the array remains the same.

Array after the 2nd pass: {5, 43, 76, 2, 98, 23, 12, 32}

**Step 3:** Progress to the fourth element (2) and begin comparing it to the elements on its left (76, 43, 5). Recognizing that 2 is less than 5, a swap occurs between 5 and 2.

Array after the 3rd pass: {2, 43, 76, 5, 98, 23, 12, 32}

**Step 4:** At this point, the focus is on the element 5. It's compared with the elements to its left (76, 43). Given that 5 is less than 43, a swap takes place between 43 and 5.

Array after the 4th pass: {2, 5, 76, 43, 98, 23, 12, 32}

**Step 5:** Continue this process for each element in the array, gradually sorting the elements from left to right. During this process, 23, 12, and 32 are all compared with 76, and no swaps are required, leaving the array unchanged.

Array after the 7th pass: {2, 5, 12, 23, 32, 43, 76, 98}

**Step 6:** Finally, after these passes, the entire array is sorted. No further swaps are necessary, resulting in the fully sorted array using the Insertion Sort algorithm: {2, 5, 12, 23, 32, 43, 76, 98}.

**Question 2:**

**First Recursive Call:**

Initial Array: {1, 3, 54, 2, 72, 23, 12, 32, 76, 12}

MergeSort([1, 3, 54, 2, 72], 0, 4):

This call splits the array into two halves: [1, 3, 54] and [2, 72].

MergeSort([1, 3, 54], 0, 2):

This call further splits the left half into [1], [3], and [54].

MergeSort([2, 72], 0, 1):

This call further splits the right half into [2] and [72].

**Merging Phase:**

Merge([1], [3], [54]): Merges [1, 3] and [54] into [1, 3, 54].

Merge([2], [72]): Merges [2] and [72] into [2, 72].

After the merging is complete, we have [1, 3, 54] and [2, 72] as sorted sub-arrays.

**Second Recursive Call:**

MergeSort([1, 3, 54, 2, 72], 0, 4) now continues to merge the sorted sub-arrays.

Merge([1, 3, 54], [2, 72]): Merges [1, 3, 54] and [2, 72] into [1, 2, 3, 54, 72].

**Third Recursive Call:**

MergeSort([12, 32, 76, 12], 0, 3): Splits the right sub-array into [12, 32] and [76, 12].

Merging Phase for Right Sub-array:

Merge([12], [32]): Merges [12] and [32] into [12, 32].

Merge([12], [76, 12]): Merges [12] and [12, 76] into [12, 12, 76].

After the merging, we have [12, 32] and [12, 12, 76] as sorted sub-arrays.

**Fourth Recursive Call:**

MergeSort([12, 32, 12, 12, 76], 0, 4): Merges the sorted sub-arrays.

Merge([12, 32], [12, 12, 76]): Merges [12, 32] and [12, 12, 76] into [12, 12, 12, 32, 76].

**Final Merging Phase:**

Merge([1, 2, 3, 54, 72], [12, 12, 12, 32, 76]): Merges the two sorted sub-arrays into the final sorted array. The result is a fully sorted array: [1, 2, 3, 12, 12, 12, 32, 54, 72, 76].

**Question 3:**

. Initial Call: merge\_sort(A)

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

Recursive Call on Left Half: merge\_sort(left\_half)

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

Recursive Call on Left Half: merge\_sort(left\_half)

Left Half: [1, 3]

Recursive Call on Left Half: merge\_sort(left\_half)

Left Half: [1]

Sorted: [1]

Right Half: [3]

Sorted: [3]

Merge: [1, 3]

Recursive Call on Right Half: merge\_sort(right\_half)

Left Half: [54]

Sorted: [54]

Right Half: [2, 72]

Recursive Call on Left Half: merge\_sort(left\_half)

Left Half: [2]

Sorted: [2]

Right Half: [72]

Sorted: [72]

Merge: [2, 72]

Recursive Call on Right Half: merge\_sort(right\_half)

Left Half: [54]

Sorted: [54]

Right Half: [12]

Sorted: [12]

Merge: [12, 54]

Merge: [2, 12, 54, 72]

Merge: [2, 12, 54, 72]

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

Recursive Call on Right Half: merge\_sort(right\_half)

Left Half: [23, 12, 32, 76, 12]

Recursive Call on Left Half: merge\_sort(left\_half)

Left Half: [23, 12]

Recursive Call on Left Half: merge\_sort(left\_half)

Left Half: [23]

Sorted: [23]

Right Half: [12]

Sorted: [12]

Merge: [12, 23]

Recursive Call on Right Half: merge\_sort(right\_half)

Left Half: [32]

Sorted: [32]

Right Half: [76, 12]

Recursive Call on Left Half: merge\_sort(left\_half)

Left Half: [76]

Sorted: [76]

Right Half: [12]

Sorted: [12]

Merge: [12, 76]

Recursive Call on Right Half: merge\_sort(right\_half)

Left Half: [32]

Sorted: [32]

Right Half: [12, 76]

Sorted: [12, 32, 76]

Merge: [12, 32, 76]

Merge: [12, 12, 32, 76]

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

Recursive Call on Right Half: merge\_sort(right\_half)

Left Half: [23, 12, 32, 76]

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

Right Half: [12]

Sorted: [12]

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

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

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

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

**Question 4:**

In the best-case scenario where the array is already sorted, Merge Sort still performs the splitting and merging steps but does so more efficiently due to fewer comparisons. The time complexity remains O(n log n), but execution may be faster.

**Question 5:**

To perform Merge Sort in descending order, you can modify the merge step to arrange elements in reverse order (largest to smallest).

**Selection Sort:**

**Question 7:**

This explains the Selection Sort algorithm using an example, showing how it iterates through the array, selecting the smallest element and placing it in its correct position, then repeating the process for the remaining elements until the array is sorted.

**Bubble Sort:**

**Question 8:**

Bubble Sort compares adjacent elements and swaps them if they are in the wrong order. It repeats this process until no more swaps are needed, resulting in a sorted array.

**Question 9:**

This code snippet outlines the Bubble Sort algorithm, which iterates through the array, compares adjacent elements, and swaps them if necessary. It continues until no more swaps are made in a pass.

**Question 10:**

An example demonstrates how Bubble Sort works by repeatedly comparing and swapping elements until the entire array is sorted.