ASSIGNMENT #3

Design and Analysis Of Algorithms

PRIORITY QUEUE AND LINEAR SORTING

Section: P

Muhammad Kashif	I21-0851
Usman Nazeer	I21-0556
Bilal Saleem	I21-0464
Muhammad Huzaifa	I21-2464

Work Breakdown

For Assignment 03, the group of four members divided their tasks as follows: Muhammad Kashif (I) and Muhammad Huzaifa collaborated on implementing a priority queue using a max-heap, while Usman Nazeer and Bilal Saleem worked on the min-heap implementation. We modified the dataset by selecting 20 entity instances and assigning priorities to them and used this data as input for both implementations.

Additionally, each member individually conducted dry runs for both Insertion Sort and Bucket Sort. They demonstrated whether Insertion Sort could achieve O(n) time complexity for specific input scenarios and highlighted the linear time efficiency of Bucket Sort compared to Insertion Sort. These dry runs were documented with clear pictures and explanations. Finally, Muhammad Kashif compiled all the work into a single document, including the work breakdown and other required components for submission. The points distribution for the assignment was also considered.

Individual Work

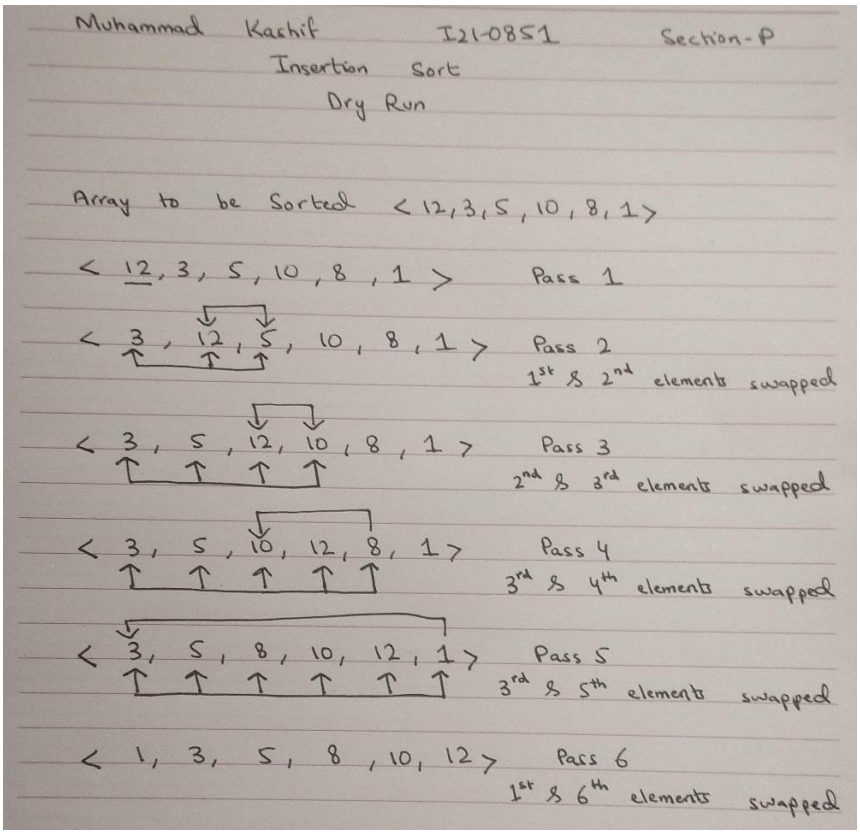
Muhammad Kashif (I21-0851)

- Insertion Sort
  - **Can we reduce it to O(n) for certain other inputs (besides the best-case input)?**
    Insertion Sort has a time complexity of O(n^2) in the worst case, but it can run in linear time (O(n)) for specific input conditions. This typically occurs when the input is nearly sorted or already sorted. The key factor that determines the efficiency of Insertion Sort is the number of inversions in the input list.

    An inversion is a pair of elements (i, j) where i < j, but a[i] > a[j]. In a sorted list, there are no inversions, and Insertion Sort performs optimally. However, if there are very few inversions, Insertion Sort can still be quite efficient.

    For instance, if the input array is already sorted, there are no inversions, and Insertion Sort performs optimally. It takes O(n) time as it only needs to compare each element once. But if the order is reversed and there are many inversions, the time complexity becomes O(n^2). If there are few inversions however, or to say an average case, Insertion Sort takes less than O(n^2) time but more than O(n). Its performance is better than when the list is fully reversed.

  - **Demonstrate this by dry running the Insertion sort on four such input lists, one by each group member.**

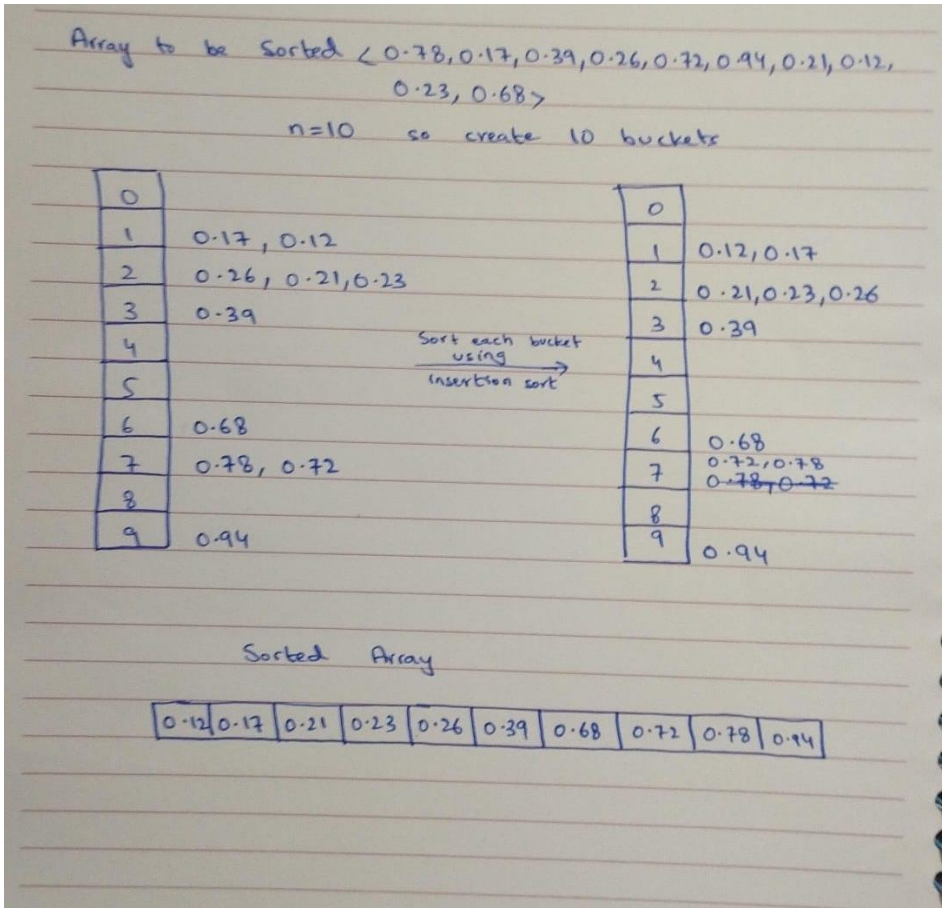


- **Describe the nature of the input and behavior of Insertion sort.**
  The nature of the input greatly affects Insertion Sort's performance. It is most efficient when the input is either already sorted (best-case scenario) or has very few inversions. In these cases, it approaches linear time complexity (O(n)). On the other hand, when the input has a high number of inversions, Insertion Sort becomes less efficient, taking closer to its worst-case time complexity of O(n^2).

  Insertion Sort is an in-place and stable sorting algorithm. Its simplicity makes it a good choice for small input sizes or when the input is almost sorted. However, for larger or more disordered lists, more efficient sorting algorithms like Merge Sort or Quick Sort are usually preferred due to their better average and worst-case time complexities.

Bucket Sort

- **Dry run the Bucket sort algorithm on four input lists, one by each group member.**



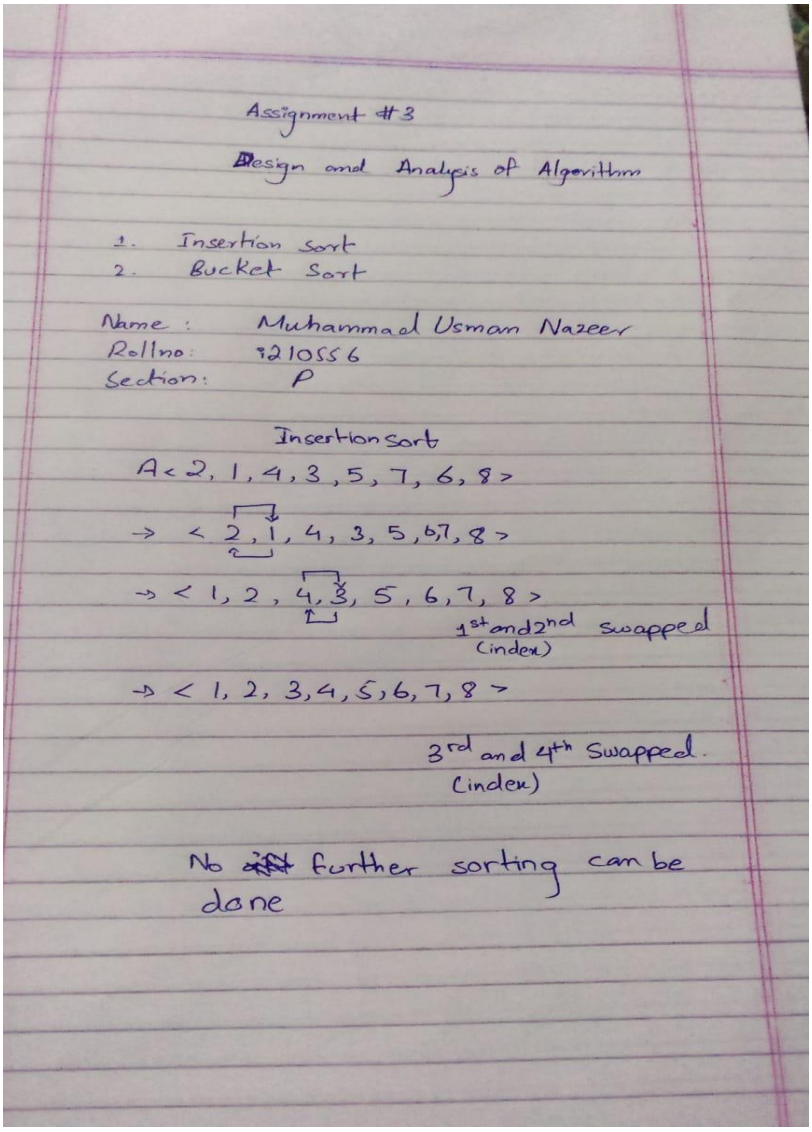
- **Explain why the Bucket sort has linear time complexity while using Insertion sort also.**
  Bucket Sort achieves linear time complexity under specific conditions. It excels when the input is uniformly distributed among buckets, reducing the need for complex comparisons and swaps. Moreover, the efficiency of sorting within small buckets is ensured, typically employing simpler sorting algorithms like Insertion Sort. However, it distinguishes itself from Insertion Sort by distributing elements into buckets based on their values, which can significantly reduce the time complexity when the input is evenly spread. The final concatenation step, while O(n), remains less influential compared to sorting within the

buckets when the input adheres to these criteria. Consequently, Bucket Sort provides a linear time complexity for well-distributed data, offering a valuable alternative to traditional comparison-based sorting algorithms.

Usman Nazeer (I21-0556)

- Insertion Sort
  - **Can we reduce it to O(n) for certain other inputs (besides the best-case input)?**
    Reducing the time complexity to O(n) for certain inputs means that, for those specific input scenarios, the algorithm's running time scales linearly with the size of the input. While this optimization is beneficial for those particular cases, it doesn't necessarily change the algorithm's time complexity for all possible inputs, as the algorithm's performance can vary based on the specific characteristics of the input data. It's essential to consider best-case, worst-case, and average-case scenarios when analyzing algorithm performance.

  - **Demonstrate this by dry running the Insertion sort on four such input lists, one by each group member.**

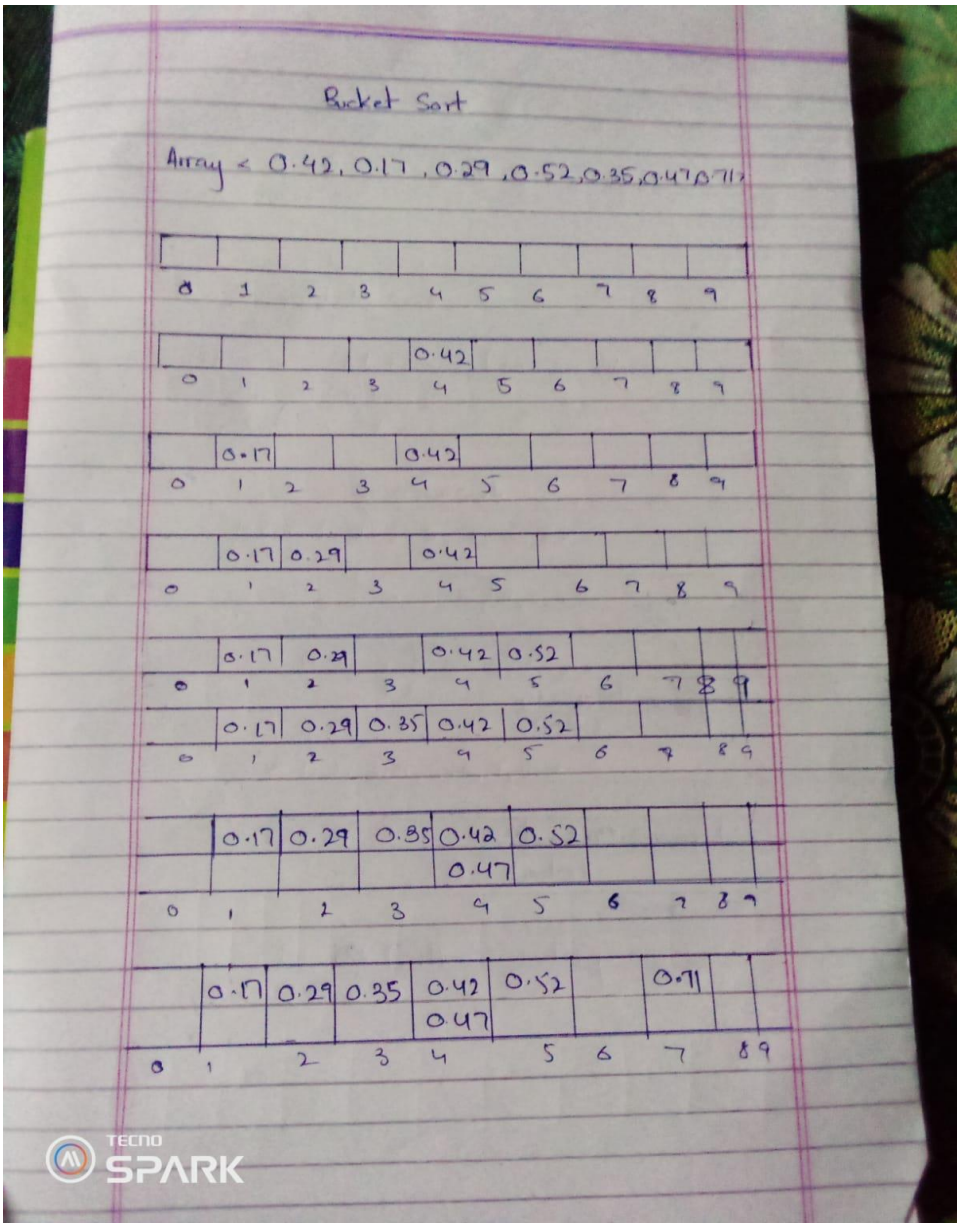


  - **Describe the nature of the input and behavior of Insertion sort.**

Insertion sort is best suited for small to moderately sized lists and performs well when the input is partially sorted. It can have a quadratic time complexity in the worst-case scenario, particularly when the input is in reverse order, but can be linear in the best-case scenario when the input is nearly sorted. It is often used for small datasets or as a part of larger sorting algorithms.

Insertion sort builds the sorted portion of the list incrementally by taking each element from the unsorted portion and inserting it into its correct position within the sorted portion. It operates in-place, meaning it doesn't require additional memory, and is a stable sorting algorithm, preserving the relative order of equal elements. While not the most efficient for large datasets, it is simple to implement and can be useful for small lists and improving partially sorted lists.

- Bucket Sort
  - **Dry run the Bucket sort algorithm on four input lists, one by each group member.**

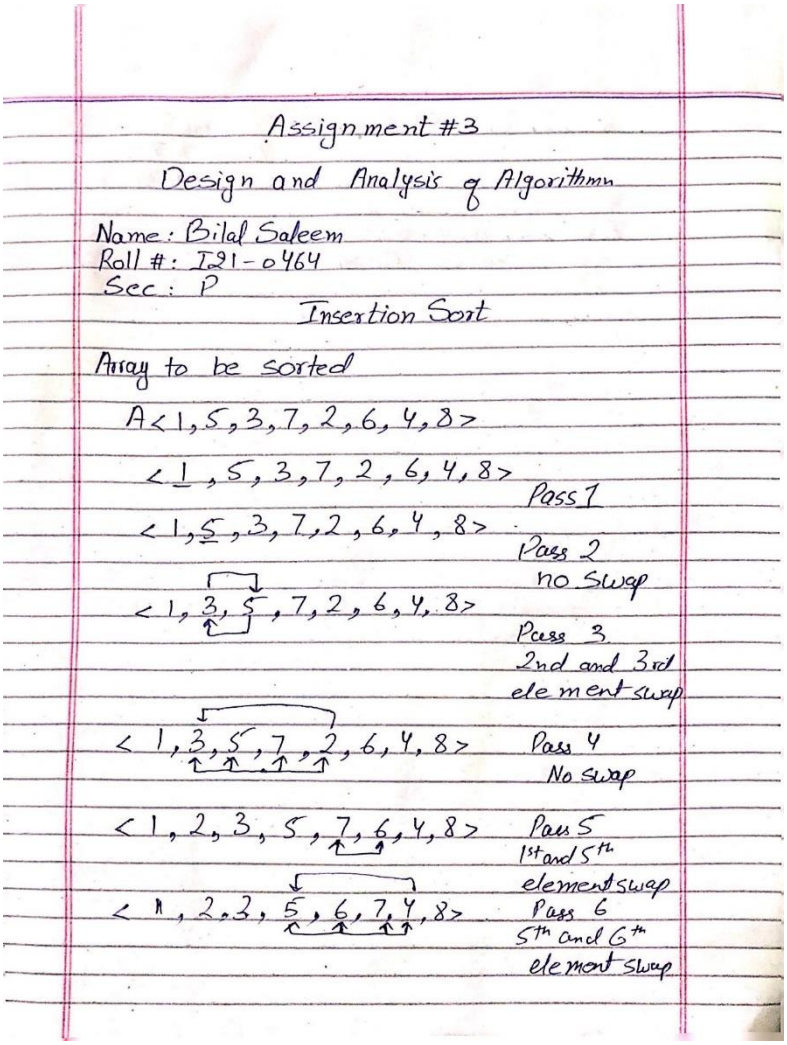


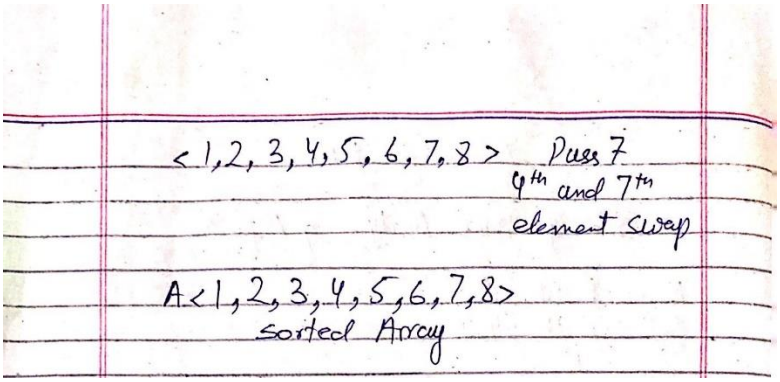
  - **Explain why the Bucket sort has linear time complexity while using Insertion sort also.**

Bucket sort achieves linear time complexity by distributing elements into buckets and then sorting each bucket using an efficient sorting algorithm, like insertion sort. The key to its efficiency is that when elements are evenly distributed among the buckets, and each bucket is relatively small, the sorting operation is fast. After sorting the buckets, you concatenate them, which is a linear-time operation. This combination of factors allows bucket sort to have a linear time complexity, making it efficient for specific scenarios where the input data satisfies these conditions.

Bilal Saleem (I21-0464)

- Insertion Sort
  - **Can we reduce it to O(n) for certain other inputs (besides the best-case input)?** Insertion Sort has a worst-case time complexity of O(n^2), but it can run in O(n) time when the input is nearly sorted or already sorted. The number of inversions in the input list is the key factor affecting its efficiency. Inversions are pairs of elements (i, j) where i < j, but a[i] > a[j]. When there are very few inversions or none in a sorted list, Insertion Sort performs optimally in O(n) time. However, if there are many inversions, it degrades to O(n^2) time. In average cases with some inversions, it's more efficient than fully reversed lists.
  - **Demonstrate this by dry running the Insertion sort on four such input lists, one by each group member.**



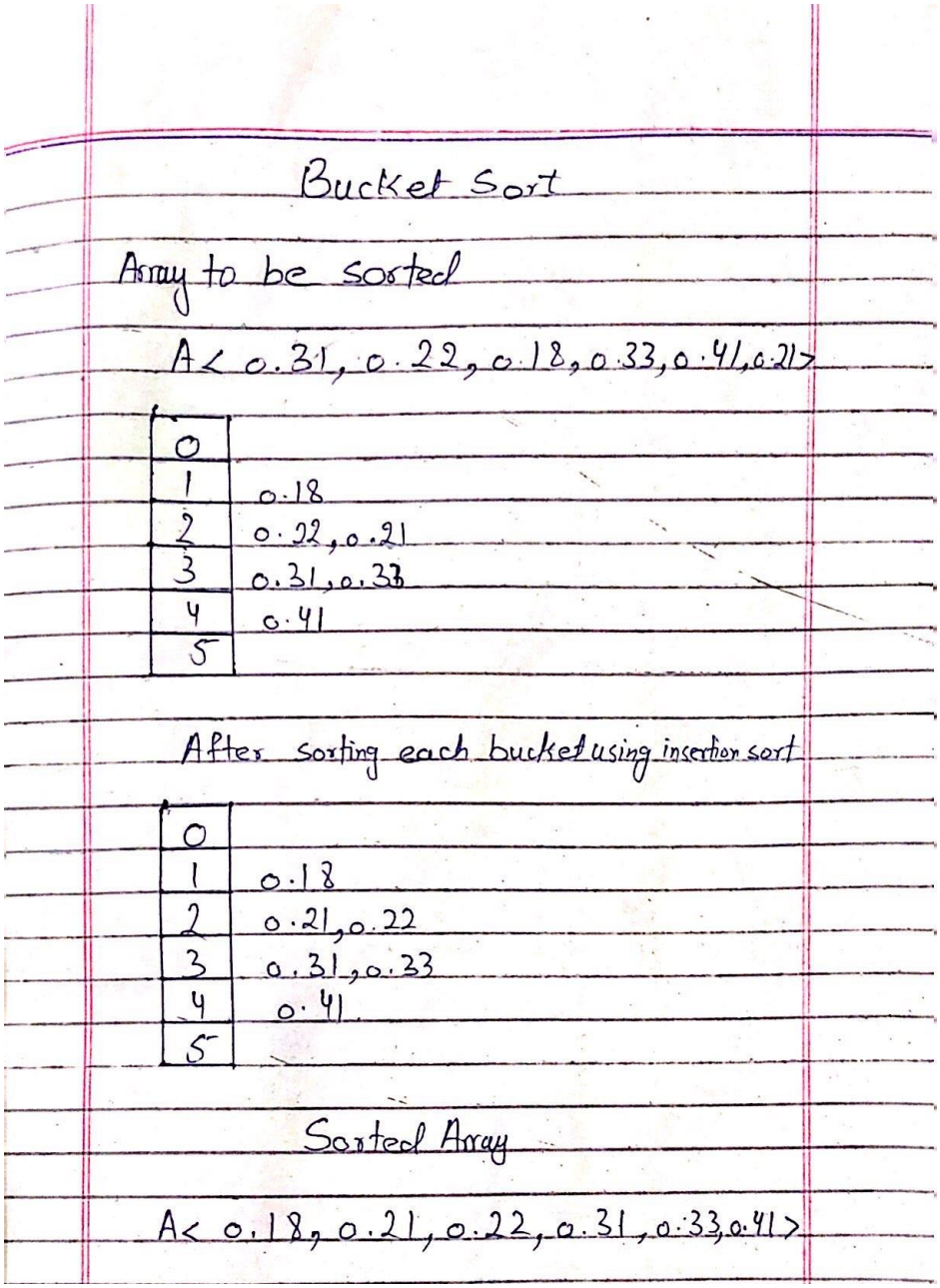


- **Describe the nature of the input and behavior of Insertion sort.**

Insertion Sort's performance is heavily influenced by the input nature. It works best when the input is already sorted or has very few inversions, approaching O(n) time complexity. However, with a high number of inversions, it becomes less efficient, nearing its O(n^2) worst-case scenario.

Insertion Sort is an in-place and stable sorting algorithm, suitable for small or nearly sorted inputs. For larger or more disordered lists, more efficient sorting algorithms like Merge Sort or Quick Sort are typically preferred due to their better average and worst-case time complexities.

- Bucket Sort
  - **Dry run the Bucket sort algorithm on four input lists, one by each group member.**



- **Explain why the Bucket sort has linear time complexity while using Insertion sort also.**

Bucket Sort achieves linear time complexity when the input is evenly distributed among buckets. This reduces the need for complex comparisons and swaps. Sorting within small buckets is efficient, typically using simpler algorithms like Insertion Sort. It distinguishes itself from Insertion Sort by distributing elements into buckets based on their values, which can significantly reduce time complexity. The final concatenation step, while O(n), remains less influential when the input adheres to these criteria. Bucket Sort offers a valuable alternative to traditional comparison-based sorting algorithms for well-distributed data.

Muhammad Huzaifa (I21-2460)

- Insertion Sort
  - **Can we reduce it to O(n) for certain other inputs (besides the best-case input)?**

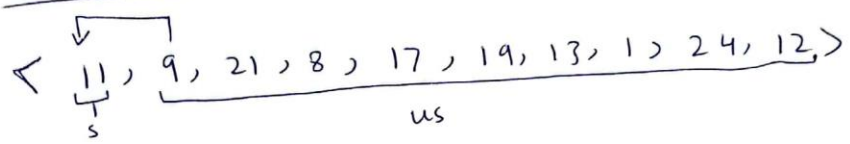
Insertion Sort is an algorithm with a worst-case time complexity of O(n^2), but under certain conditions, it can operate in linear time (O(n)). These conditions usually arise when the input is nearly sorted or completely sorted. The efficiency of Insertion Sort is largely determined by the number of inversions present in the input list. An inversion is defined as a pair of elements (i, j) where i < j, but a[i] > a[j]. In a sorted list, there are no inversions, allowing Insertion Sort to operate at its best. Even with a small number of inversions, Insertion Sort can still perform efficiently. For example, if the input array is already sorted, there are no inversions, and Insertion Sort operates at its best. It takes O(n) time as it only needs to compare each element once. However, if the order is reversed and there are many inversions, the time complexity escalates to O(n^2). In an average case scenario, where there are a moderate number of inversions, Insertion Sort operates in less than O(n^2) time but more than O(n). This performance is superior to the scenario where the list is fully reversed.

  - **Demonstrate this by dry running the Insertion sort on four such input lists, one by each group member.**

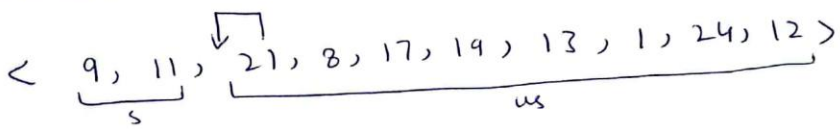
Insertion Sort
Dry - Run

Array = < 11, 9, 21, 8, 17, 19, 13, 1, 24, 12 >

0th pass

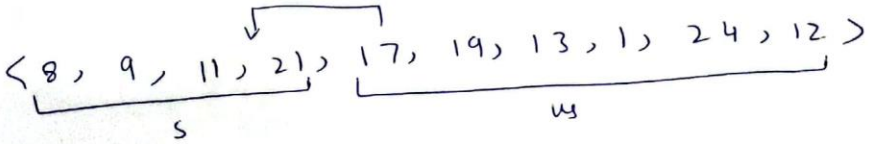


1st pass

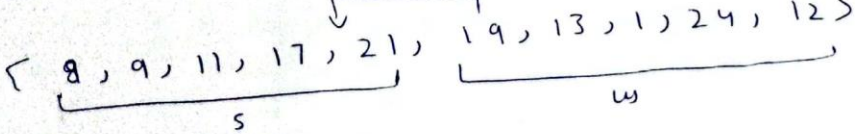


2nd pass

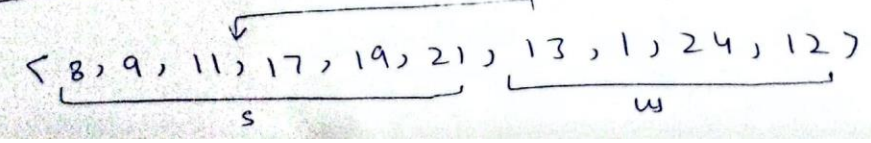
3rd pass

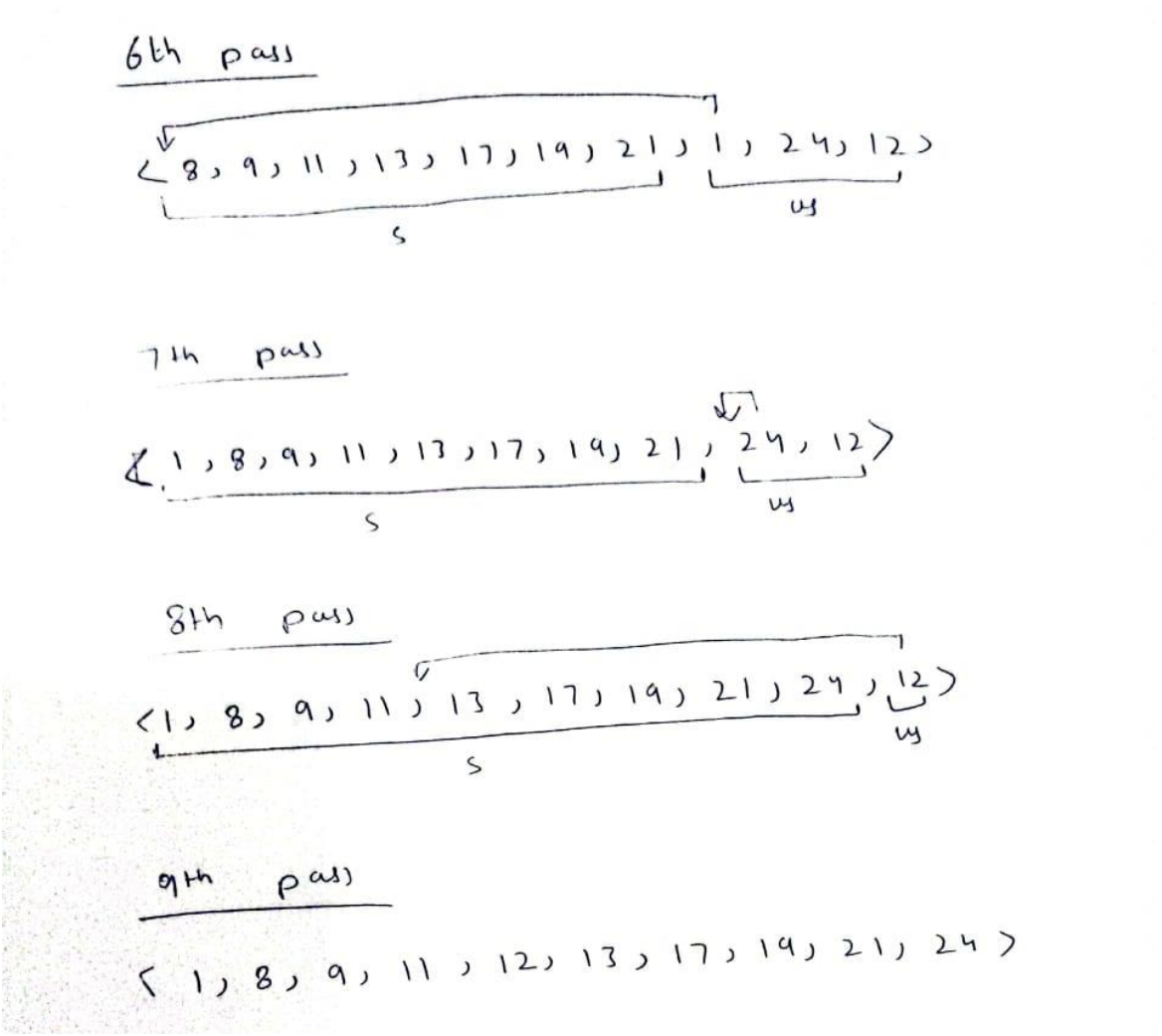


4th pass



5th pass

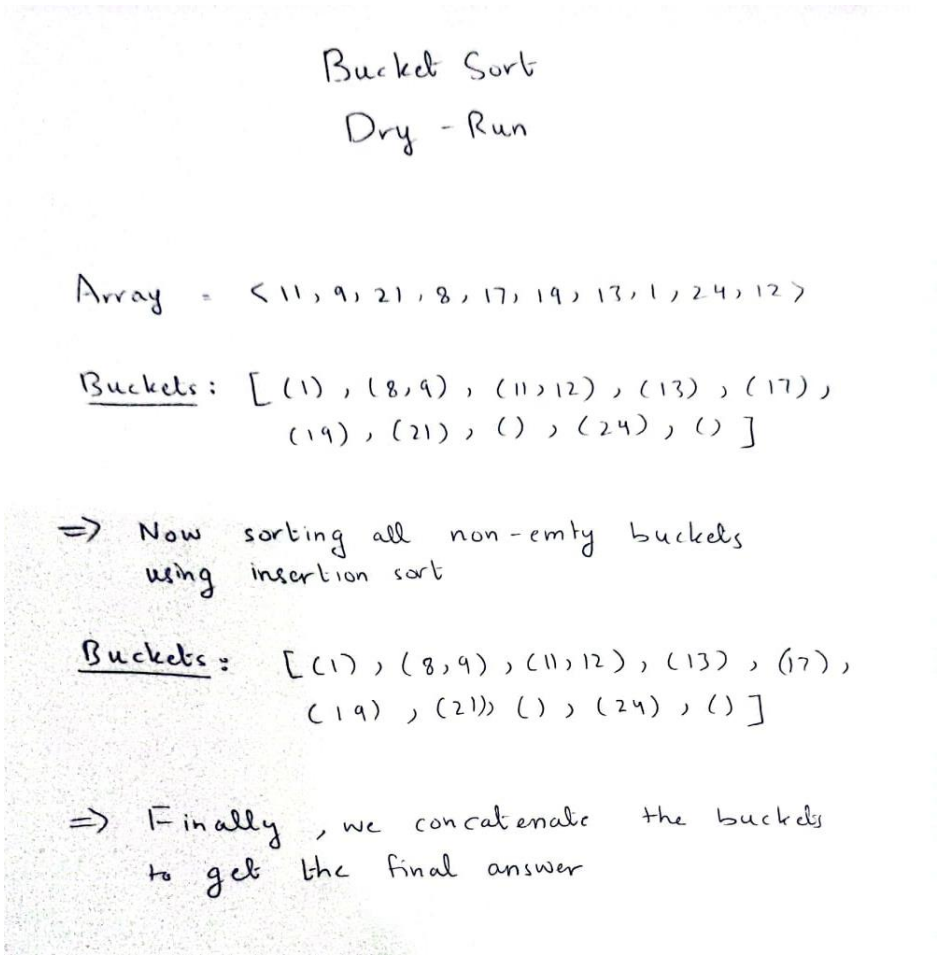




- **Describe the nature of the input and behavior of Insertion sort.**

The performance of Insertion Sort is heavily influenced by the characteristics of the input. It performs optimally when the input is already sorted or has minimal inversions, approaching a linear time complexity (O(n)). Conversely, when the input has a large number of inversions, the efficiency of Insertion Sort decreases, nearing its worst-case time complexity of O(n^2). Insertion Sort is a stable and in-place sorting algorithm. Its straightforward nature makes it an ideal choice for small inputs or nearly sorted inputs. However, for larger or more disordered lists, sorting algorithms with superior average and worst-case time complexities, such as Merge Sort or Quick Sort, are typically favored.

- Bucket Sort
  - **Dry run the Bucket sort algorithm on four input lists, one by each group member.**



- **Explain why the Bucket sort has linear time complexity while using Insertion sort also.**

Bucket Sort operates optimally under certain conditions, achieving linear time complexity. It performs best when the input is uniformly distributed across buckets, minimizing the need for complex comparisons and swaps. The efficiency of sorting within small buckets is ensured, typically using simpler sorting algorithms like Insertion Sort. However, Bucket Sort differentiates itself from Insertion Sort by distributing elements into buckets based on their values, which can significantly reduce time complexity when the input is evenly spread. The final step of concatenating the sorted buckets, while O(n), is less significant compared to the sorting within the buckets when the input meets these criteria. Therefore, for well-distributed data, Bucket Sort provides a linear time complexity, offering a valuable alternative to traditional comparison-based sorting algorithms.