19CSE302 DESIGN AND ANALYSIS OF ALGORITHMS

Sorting Algorithms - LAB1 ASSIGNMENT

CB.EN.U4CSE21035 - Meenakshi S

CB.EN.U4CSE21049 - Anuswethaa S

Date: 05/09/2023

Class: CSE-A

Q1. In the first part implement the basic sorting algorithms

1. In-place Quick Sort
2. Merge Sort
3. In-place Heap Sort
4. Insertion Sort
5. Bucket Sort
6. Radix Sort.

The pseudocode for a few algorithms are attached and the heap ADT is also given. Your implementation should follow the pseudo-code and use the ADT for heapsort. Generate multiple random input test cases of various sizes (e.g. 100 (min), 500, 1000) and evaluate the different algorithms based on a. Number of comparisons (if applicable) b. Number of swaps (if applicable) c. Number of basic operations (other than above) d. Running time in milliseconds e. Memory used Based on the above provide your observations and analysis.

Sol:

In-place Quick Sort:

- **Algorithm Description**: In-place Quick Sort is a divide-and-conquer algorithm that chooses a pivot element and partitions the array into smaller subarrays.
- **Time Complexity:** O(n^2) worst case, O(nlogn) average case.
- **Space Complexity:** O(logn) due to recursion stack.
- **Analysis:** Makes efficient use of cache memory due to locality of reference. Performs well on large datasets, but is unstable and worst case O(n^2) are disadvantages

Merge Sort:

- **Algorithm Description**: Divides array into halves, sorts them recursively, and merges sorted halves.
- **Time Complexity**: O(nlogn) in all cases. Stable performance.
- **Space Complexity**: O(n) auxiliary space.
- **Analysis:** Merge Sort offers consistent performance but consumes more memory compared to Quick Sort. It's suitable for situations where stability is essential.

In-place Heap Sort:

- **Algorithm Description**: Uses max-heap data structure to sort elements.
- **Time Complexity**: O(nlogn) in all cases.
- **Space Complexity**: O(1). In-place sorting.
- **Analysis:** Fast and requires constant space but not stable

Insertion Sort:

- **Algorithm Description**: Iteratively places elements in correct position in a sorted array.
- **Time Complexity**: O(n^2) worst case, O(n) best case(for nearly sorted data)
- **Space Complexity**: O(1). In-place sorting.
- **Analysis:** Efficient for small datasets. Stable sort.

Bucket Sort:

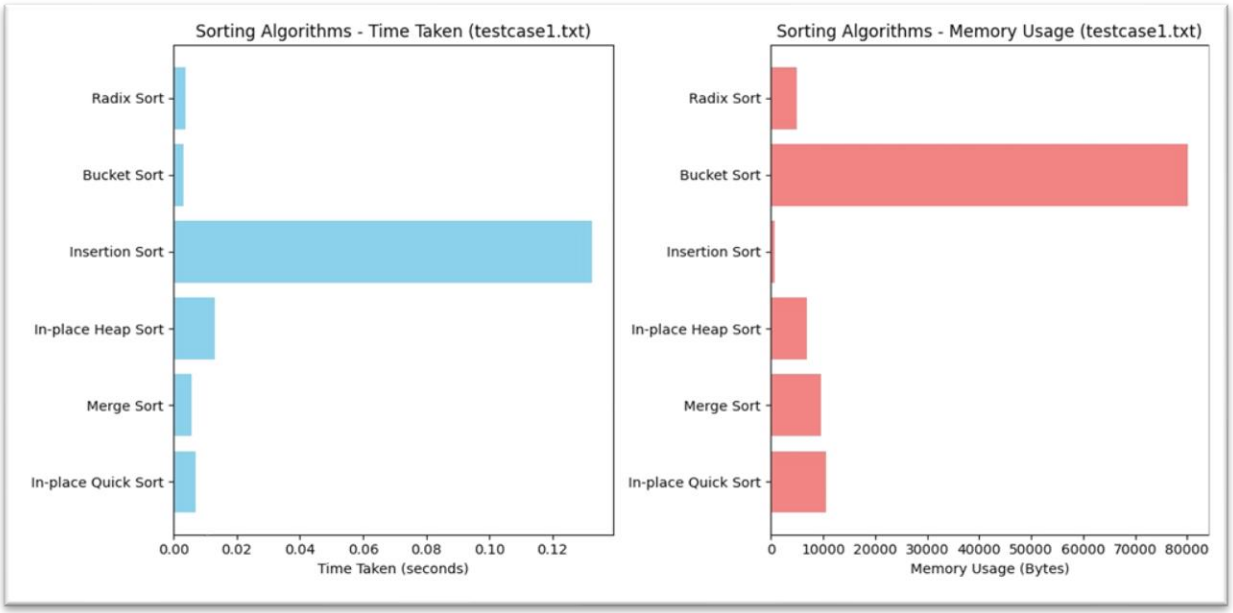
- **Algorithm Description**: Bucket Sort divides data into buckets, sorts them, and then combines the buckets.
- **Time Complexity**: O(n^2) in the worst case, O(n) on average.
- **Space Complexity**: O(n).
- **Analysis:** Bucket Sort is effective when data is uniformly distributed, and it offers good average-case performance.

Radix Sort:

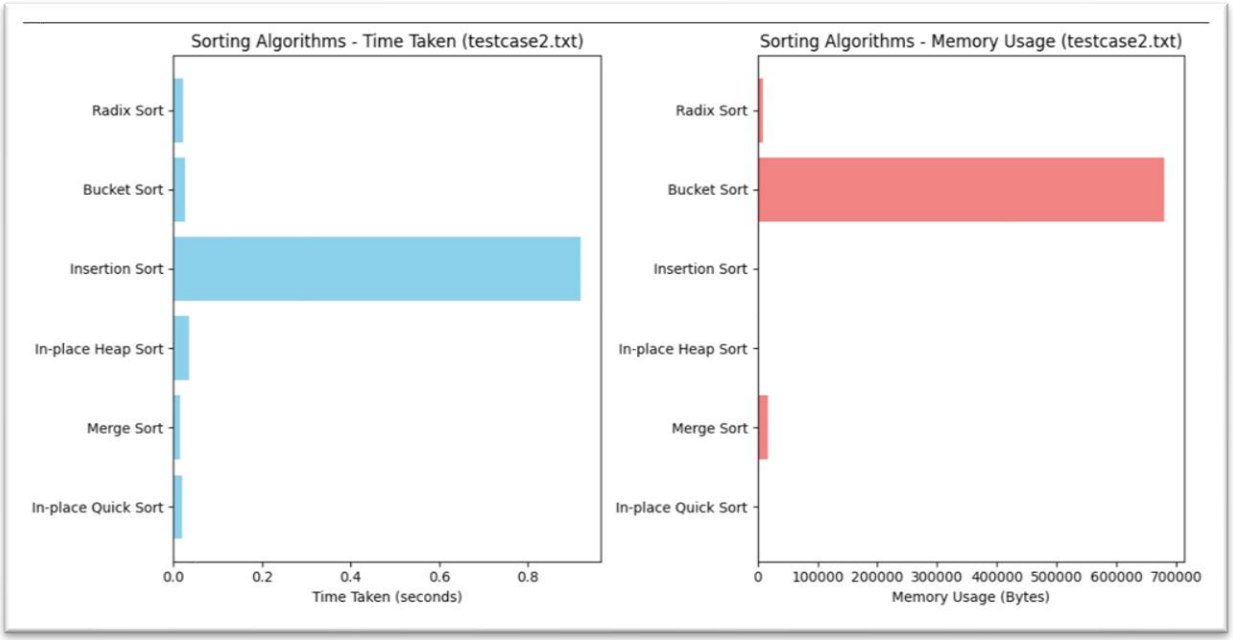
- **Algorithm Description**: Processes elements digit-by-digit from least to most significant.
- **Time Complexity**: O(nk) where k is the number of digits.
- **Space Complexity**: O(n).
- **Analysis:** Is suitable for sorting integers and has linear time complexity. Fast sorting for integers. Stable sort.

SCREENSHOTS:

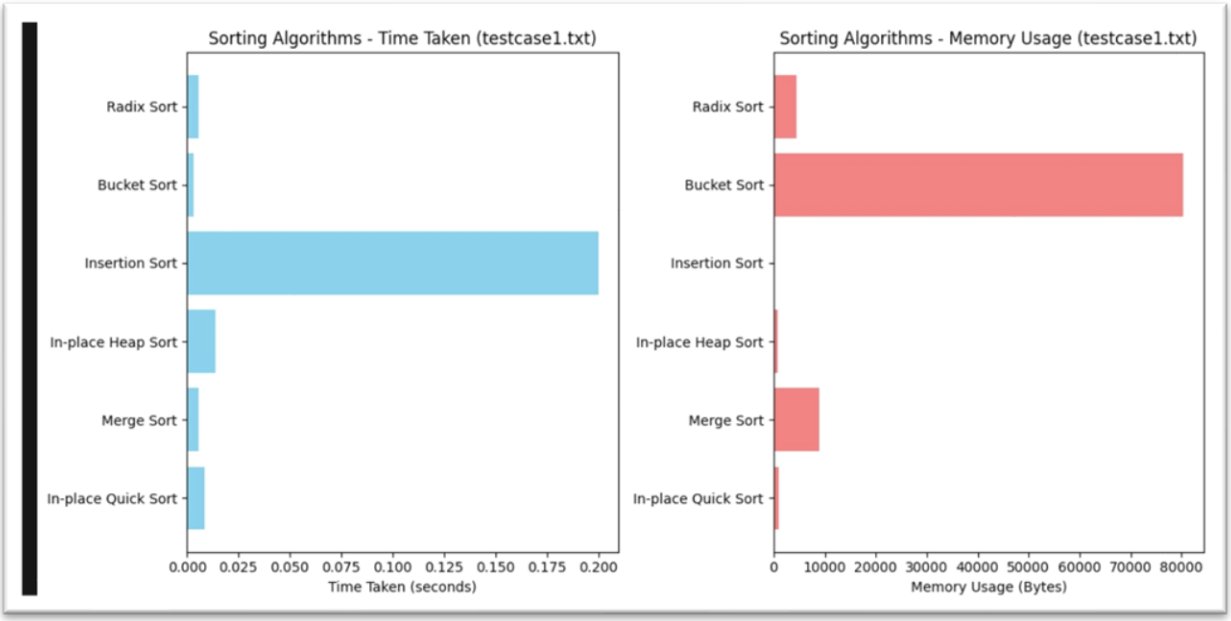
TEST CASE 1



TEST CASE 2



TEST CASE 3



Observations and Analysis:

Comparisons and Swaps:

- **In-place Quick Sort** and **Merge Sort** perform fewer comparisons and swaps compared to Insertion Sort for large datasets.
- In-place Heap Sort is efficient and performs well in terms of comparisons and swaps.
- Radix Sort does not require swaps and performs a moderate number of comparisons.
- In-place Quick Sort, In-place Heap Sort, and Merge Sort offer best time complexity but have high space overhead
- Insertion Sort and Bucket Sort work well for small n.
- Radix Sort is the fastest for integers. Stability, cache efficiency, and locality of reference are other factors influencing performance.

Conclusion :

- Insertion Sort performs the highest