Project 1: Comparison-based Sorting Algorithms

ITCS-6114, Summer 2024
July 12, 2024

Data Structures Used

Arrays: The main data structure used in all sorting algorithms is the array.

Sorting Algorithms and Complexity Analysis

For this project, I used a total of 5 different sorting algorithms.

1. Insertion Sort:

- Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a stable sorting algorithm, meaning that elements with equal values maintain their relative order in the sorted output.
- **Time Complexity:**
  - Best Case: $O(n)$
  - Average Case: $O(n^2)$
  - Worst Case: $O(n^2)$
- **Space Complexity:** $O(1)$

2. Merge Sort:

- Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.
- **Time Complexity:**
  - Best Case: $O(n \log n)$
  - Average Case: $O(n \log n)$
  - Worst Case: $O(n \log n)$

- **Space Complexity:** $O(n)$

3. Heap Sort:

- Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.
  - Best Case: $O(n \log n)$
  - Average Case: $O(n \log n)$
  - Worst Case: $O(n \log n)$
- **Space Complexity:** $O(1)$

4. Quick Sort:

- Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- **Time Complexity:**
  - Best Case: $O(n \log n)$
  - Average Case: $O(n \log n)$
  - Worst Case: $O(n^2)$
- **Space Complexity:** $O(\log n)$

5. Modified Quick Sort:

- This Quick Sort algorithm uses the median of three methods by selecting the first, middle, and last elements, finding the median of the three for the pivot, and switching to insertion sort for small subarrays with a size less than or equal to 10.
- **Time Complexity:** Same as Quick Sort
- **Space Complexity:** $O(\log n)$

Setting Recursion Limit

In Python, the default recursion limit is set to a 1000 to prevent a stack overflow caused by infinite recursion. I encountered a maximum recursion depth error while sorting large arrays because algorithms like Quick Sort use recursion heavily. To handle larger input sizes I used `sys.setrecursionlimit(10**6)` to increase the recursion limit to 1,000,000.

Results

I performed tests with multiple different input sizes, generating **random**, **sorted**, and **reversed** arrays for each size. The tables below summarize the time taken (in seconds) by each sorting algorithm for different input sizes and array conditions (**random**, **sorted**, **reversed**).

Random Arrays:

Table 1: Time taken by each sorting algorithm on random arrays (in seconds)

Input Size	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Modified Quick Sort
1000	0.03271	0.00200	0.00371	0.00120	0.00121
5000	0.68625	0.01779	0.01893	0.00673	0.00951
10000	3.10595	0.02592	0.03653	0.01560	0.01270

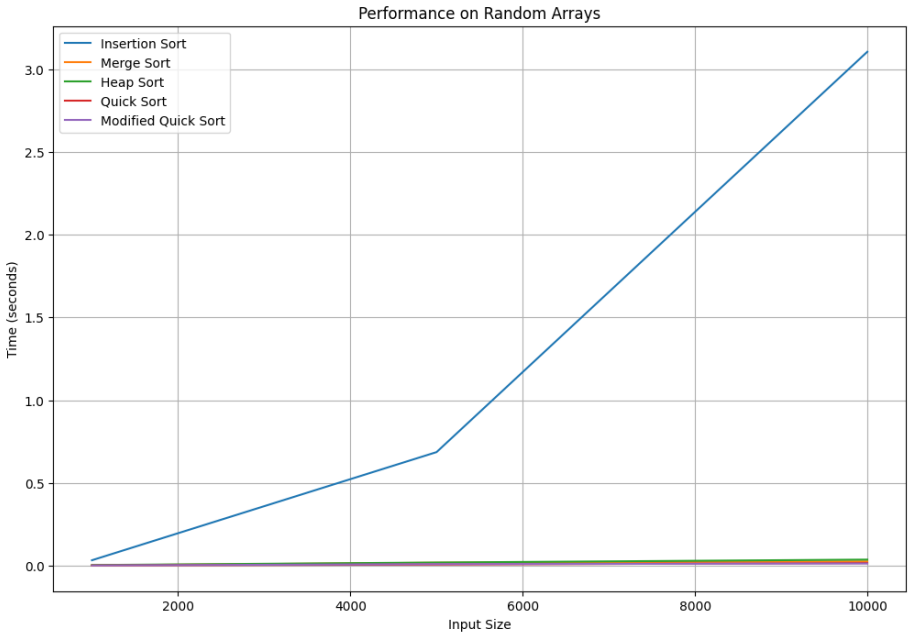

Figure 1: Random Arrays

Sorted Arrays:

Table 2: Time taken by each sorting algorithm on sorted arrays (in seconds)

Input Size	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Modified Quick Sort
1000	0.00000	0.00160	0.00340	0.04854	0.00081
5000	0.00060	0.01184	0.01947	1.59251	0.00643
10000	0.00200	0.02041	0.03813	6.01266	0.00914

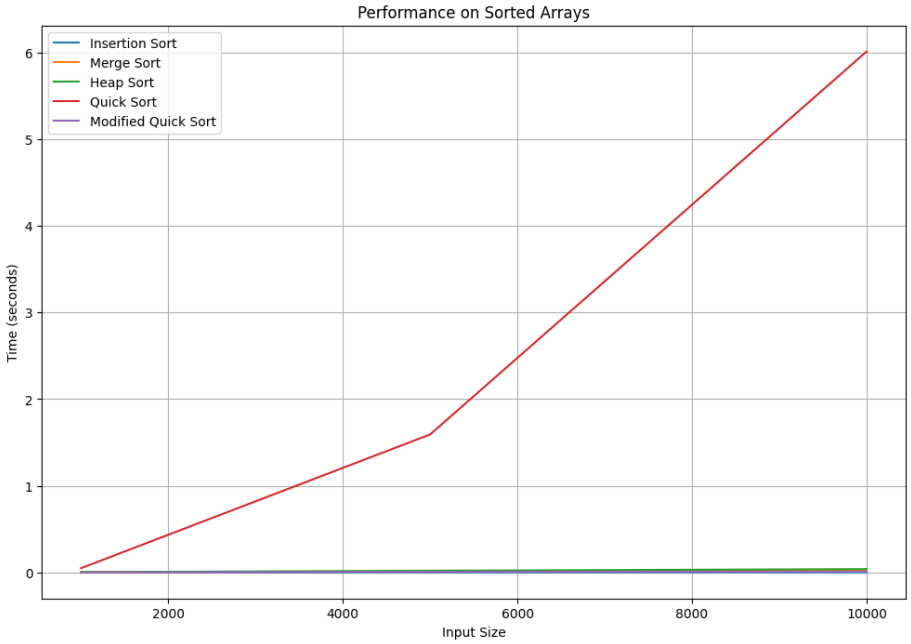

Figure 2: Sorted Arrays

Reversed Arrays:

Table 3: Time taken by each sorting algorithm on reversed arrays (in seconds)

Input Size	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Modified Quick Sort
1000	0.04292	0.00180	0.00200	0.03744	0.00140
5000	1.58539	0.01128	0.01602	1.16417	0.01460
10000	6.81070	0.02053	0.03114	3.66233	0.02133

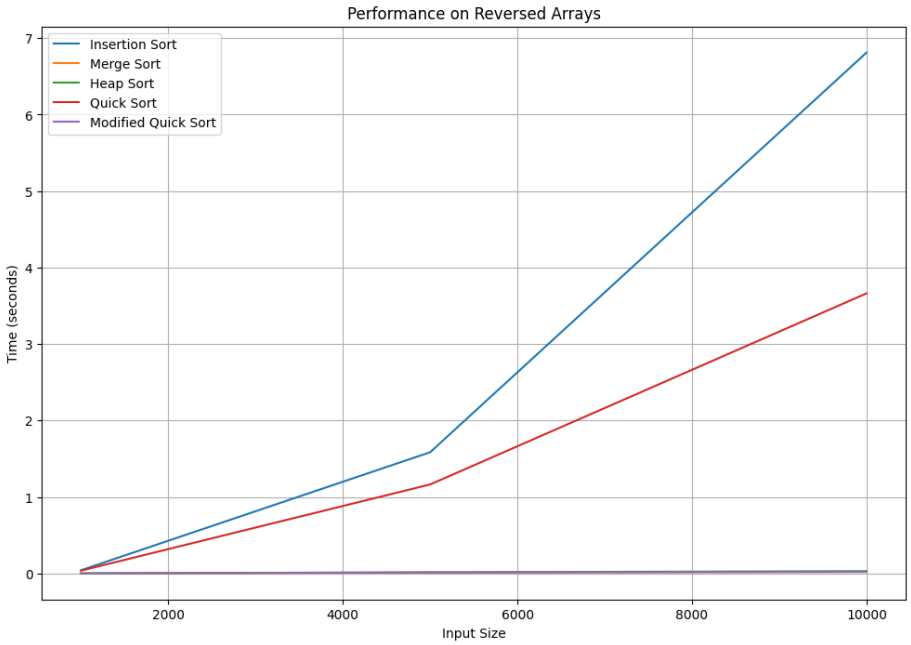

Figure 3: Reversed Arrays

Conclusion

The tests revealed that each sorting algorithm has its strengths and weaknesses based on the type of input array (random, sorted, or reversed).

For random arrays, **Quick Sort** and **Modified Quick Sort** were the fastest, with Quick Sort taking 0.01560 seconds and Modified Quick Sort 0.01270 seconds for an input size of 10,000.

For already sorted arrays, **Insertion Sort** was the quickest, taking only 0.00060 seconds for an input size of 5,000, compared to Quick Sort's 1.59251 seconds.

For reversed arrays, Insertion Sort struggled, taking 6.81070 seconds for an input size of 10,000. **Merge Sort** and **Heap Sort** performed more consistently, with Merge Sort taking 0.02053 seconds and Heap Sort 0.03114 seconds.

In summary, Quick Sort and Modified Quick Sort are good for more general use while Insertion Sort is better for nearly sorted data, and Merge Sort and Heap Sort provide steady performance across multiple different types.

References

[1] GeeksforGeeks. *Insertion Sort Algorithm*. Available: https://www.geeksforgeeks.org/insertion-sort-algorithm/.

[2] W3Schools. *Merge Sort Algorithm*. Available: https://www.w3schools.com/dsa/dsa_algo_mergesort.php.

[3] GeeksforGeeks. *Heap Sort*. Available: https://www.geeksforgeeks.org/heap-sort/.

[4] Wikipedia. *Quicksort*. Available: https://en.wikipedia.org/wiki/Quicksort.

[5] GeeksforGeeks. *Handling recursion limit*. Available: //www.geeksforgeeks.org/python-handling-recursion-limit/.