Analysis of Algorithms

BLG 335E

Project 1 Report

RACHA BADREDDINE
badreddine21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 02/11/2024

1. Implementation

1.1. Sorting Strategies for Large Datasets

Below, I will discuss the comparative performance of Bubble sort, Insertion sort and merge sorts algorithms on different data permutations with same size (randomly ordered, nearly sorted, sorted in ascending order, and sorted in descending order), as shown in Table 1.1, followed by an analysis of their behavior as dataset sizes increase, as shown in Table 1.2.

	tweets	**tweetsSA**	**tweetsSD**	**tweetsNS**
Bubble Sort	23.6182s	18.4265s	24.7799s	17.7478s
Insertion Sort	9.56975s	0.00195884s	19.7938s	0.480827s
Merge Sort	0.0273784s	0.0227984s	0.0215379s	0.0233747s

Table 1.1: Comparison of different sorting algorithms on input data (Same Size, Different Permutations).

	5K	**10K**	**20K**	**30K**	**50K**
Bubble Sort	0.235776s	0.914398s	3.79938s	8.42469s	23.5039s
Insertion Sort	0.0942464s	0.379908s	1.52478s	3.41046s	9.61793s
Merge Sort	0.00486979s	0.00593089s	0.014663s	0.0189515s	0.028297s

Table 1.2: Comparison of different sorting algorithms on input data (Different Size).

Discussion

Table 1.1:

From the results in Table 1.1, we observe that Merge Sort is the fastest among the three algorithms by a significant difference. This performance advantage is due to Merge Sort's time complexity of $O(nlogn)$. In contrast, both Bubble Sort and Insertion Sort have a worst-case complexity of $O(n^2)$, making them slower. However, Insertion Sort consistently performs better than Bubble Sort despite having the same worst-case complexity. This difference arises because Bubble Sort performs more swaps: it repeatedly compares and swaps adjacent elements, moving the largest unsorted element to the end of the list in each pass. In contrast, Insertion Sort implementation shifts elements until it finds the correct position for the current element, which involves fewer operations.

Even though this was the general observation, we can notice that when an ascendingly sorted array was given as input, Insertion Sort was the fastest. This was expected since this case represents the best case for this algorithm, where no swaps or

shifts are performed, and the running time ends up being linear $O(n)$, which is faster than Merge Sort's $O(nlogn)$.

Secondly, let's analyze the behavior of each algorithm with different permutations of the data. Starting with Bubble Sort, which was the slowest, taking multiple seconds to sort the data, we can see that the running time was almost the same for all permutations, with a slight difference for the sorted and nearly sorted cases. This is likely due to the smaller number of swaps, or even zero swaps when the data is already sorted. This behavior arises from my implementation of the algorithm, where the two loops are executed independently of the input, resulting in a running time of $O(n^2)$ regardless of the input (the inner loop always starts from the first element up to (size $- i$), where i represents the i-th iteration).

Moving to Insertion Sort, it was faster than Bubble Sort in all cases and slower than Merge Sort on average. As discussed earlier, the ascendingly sorted array is the best case for this algorithm, which is why it was the fastest in this case. On the other hand, the descendingly sorted input represents the worst case for the algorithm, with a running time of $O(n^2)$ due to the maximum number of shifts where each time we visit the whole soerted part and add the element as the first one. The running time for this case was close to that of Bubble Sort, with a slight difference due to the number of swaps. The nearly sorted array input also ran faster compared to other cases but remained slightly slower than Merge Sort, representing a scenario close to the best case.

Finally, we have Merge Sort, which was the fastest and most preferred among these algorithms. We can observe that the running time for this algorithm was nearly the same for all different permutations. This is due to the running time being independent of the input, as the divide, conquer, and combine steps are executed in the same manner regardless of the input configuration giving a running time of $O(nlogn)$.

Table 1.2:

From the results in Table 1.2, we can observe the increase in running time with larger datasets and see how dataset size affects the performance of each algorithm. As a general observation, we obtain the same results as in the previous table, where the fastest algorithm was Merge Sort, followed by Insertion Sort, and finally Bubble Sort. The reasons for these differences were discussed in the previous section.

Here, we can also notice the differences in the growth rate of each algorithm. Both Bubble Sort and Insertion Sort have quadratic growth, $O(n^2)$, which leads to a significant increase in running time as the dataset size grows, as observed in the difference between the 5K and 50K datasets, and the rate growth can be observed through the other data sizes.

However, the rate of growth for the Merge Sort algorithm was small, with only a slight increase in running time between the smallest and largest dataset sizes. This is due to its running time complexity of $O(n \log n)$, which is more efficient for handling large

datasets.

1.2. Targeted Searches and Key Metrics

In this part i ran a binary search for the tweet with ID 1773335 in asorted array. After that i searched for the number of tweets with favorite count more than 250 favorites on the datasets given in the Table1.3:

	5K	**10K**	**20K**	**30K**	**50K**
Binary Search	2.5176e-05s	2.9846e-05s	2.7566e-05s	2.8489e-05s	3.5745e-05s
Threshold	0.000198592s	0.000452287s	0.000770256s	0.00169961s	0.00185331s

Table 1.3: Comparison of different metric algorithms on input data (Different Size).

Discussion

From a general observation of Table 1.3, it can be noticed that the binary search algorithm is faster than the threshold algorithm, which is a linear algorithm with a complexity of $O(n)$, while binary search has a logarithmic complexity of $O(\log n)$. The running time growth rate of binary search is so small between larger and smaller datasets. On the other hand, we can observe a slight increase in the running time of the linear algorithm as the data size grows, with a small but consistent increase as the input size becomes larger.

In the binary search algorithm, I observed that it was sometimes a bit faster with larger data sizes. I believe this is due to the position of the target ID within the dataset. As the dataset size increases, the index of the target ID can vary, sometimes appearing closer to the middle of the search range in earlier iterations compared to smaller datasets. For example, the position of ID 426 in a 5k dataset is not the same as the position of ID 840 in a 10k dataset or ID 4382 in a 50k dataset.

1.3. Discussion Questions

Discuss the methods you've implemented and the complexity of those methods.

Sort Algorithms

The 3 sort algorithms implemented in this assignment are 3 stable sorts having different runtime complexities. The results discussed in the first section are summarized in Figure 1.1

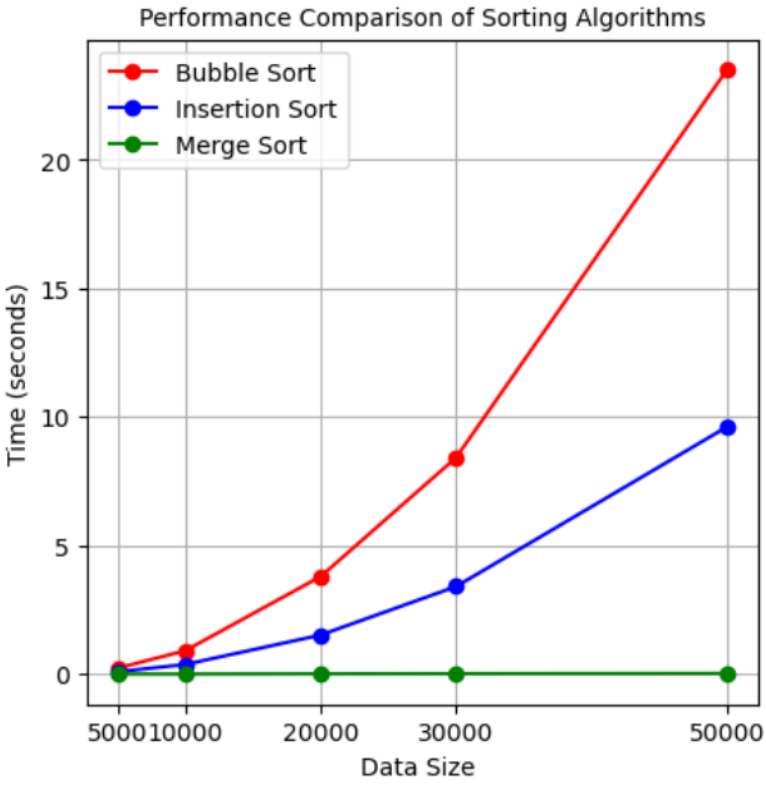

Figure 1.1: Runtime growth

- **Bubble sort:** The runtime complexity is $O(n^2)$ for the best, average, and worst cases (independent of the input data).
- **Insertion sort:** It has a linear best case runtime with complexity $O(n)$ (for sorted input), while the average and worst cases are $O(n^2)$.
- **Merge sort:** The runtime complexity is $O(n \log n)$ for the best, average, and worst cases (independent of the input data).

Targeted Searches and Key Metrics

In Figure 1.2 we can see the results shown before in Table 1.3. As it has been discussed before:

- **Binary Search:** has a comlexity of $O(\log n)$, thus small change with larger data.
- **Treshold:** this was implemented as linera Algorithm with a complexity of $O(n)$.

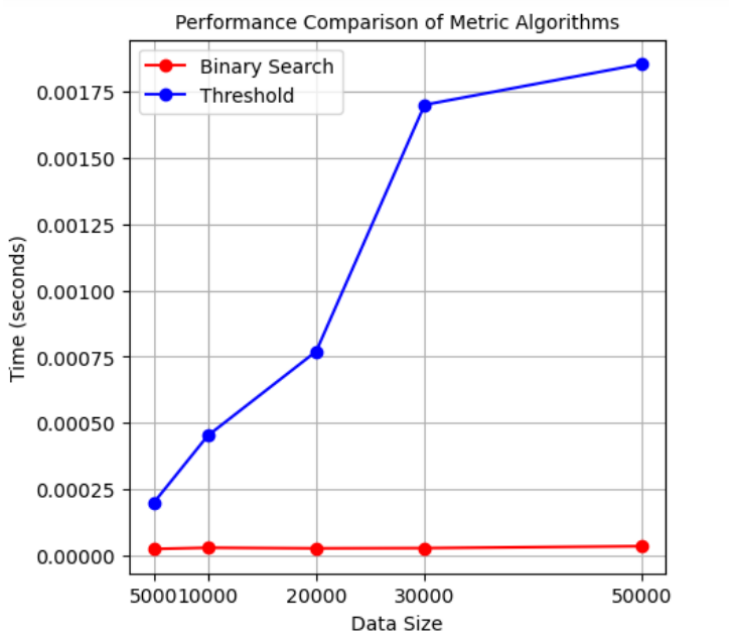

Figure 1.2: Runtime growth

What are the limitations of binary search? Under what conditions can it not be applied, and why?

The limitation of binary search is that it can only be applied to sorted data. If the input data is unsorted, we can not use binary search directly because the target can be in any part of the data. While it is possible to sort the data first and then apply binary search, this approach is generally inefficient since the most efficient sorting algorithm we have seen so far has a time complexity of $O(n \log n)$, which, when combined with the $O(\log n)$ complexity of binary search, results in an overall complexity of $O(n \log n)$. Therefore, for unsorted data, using linear search with $O(n)$ complexity is often more practical.

How does merge sort perform on edge cases, such as an already sorted dataset or a dataset where all tweet counts are the same? Is there any performance improvement or degradation in these cases?

As mentioned previously, the running time of the merge sort algorithm is independent of the input data. The algorithm begins by dividing the input array into sub-arrays of size 1. This process is completed without checking or evaluating the data itself, and then it proceeds to merge these sub-arrays back together. The only difference here is that we always merge by taking elements from the left sub-array first, followed by the right sub-array. As a result, the running time remains consistent across all cases, with neither improvement nor degradation based on the data's initial ordering, this can be proven by results shown in Table 1.1.

Were there any notable performance differences when sorting in ascending versus descending order? Why do you think this occurred or didn't occur?

In Table 1.4, the results for sorting data in descending order across three different data permutations are shown. Basically there is no difference noticeable for bubble and merge sort algorithms because the only part changes is the comparison. The only difference is that the descendingly sorted input became the best case for insertion sort with a running time of $O(n)$, while the ascendingly sorted input represented its worst case requiring maximum shifts. All the other observations are the same as the ascending order.

	tweets	**tweetsSA**	**tweetsSD**
Bubble Sort	27.6459s	28.4225s	22.2836s
Insertion Sort	22.7735s	23.0328s	0.00210095s
Merge Sort	0.0346999s	0.0270396s	0.0332566s

Table 1.4: Comparison of different sorting algorithms on input data (Same Size, Different Permutations).