Analysis of Algorithms

BLG 335E

Project 1 Report

Ece Nil Kombak 820220330
kombak22@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 31.10.2024

1. Implementation

1.1. Sorting Strategies for Large Datasets

Apply ascending search with the algorithms you implemented on the data expressed in the header rows of Tables 1.1 and 1.2. Provide the execution time in the related cells. Make sure to provide the unit.

	tweets	**tweetsSA**	**tweetsSD**	**tweetsNS**
Bubble Sort	8.41373s	2.59588s	6.97291s	2.88696s
Insertion Sort	1.71276s	0.000549125s	3.45941s	0.0944376s
Merge Sort	0.0247283s	0.016536s	0.0161561s	0.017537s

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

	5K	**10K**	**20K**	**30K**	**50K**
Bubble Sort	0.108989s	0.346118s	1.34693s	3.03738s	8.50528s
Insertion Sort	0.0407783s	0.0818411s	0.290622s	0.636087s	1.74791s
Merge Sort	0.0059565s	0.00925833s	0.0169451s	0.0200087s	0.0249959s

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

Discuss your results

Table 1.1- Different Permutations

Bubble Sort is the slowest algorithm in the comparison. It really struggles with datasets tweetsSD and tweets, making it a poor choice for most real-world situations. The results show that Bubble Sort does not handle different types of data well where the data is not in favor of the task, in our case sorted like the opposite of wanted. Overall, Bubble Sort is not practical for sorting tasks.

Insertion Sort performs better than Bubble Sort, especially on datasets like tweetsSD and tweets, where Bubble Sort handled poorly. However, it is still slow with more mixed-up datasets. We can say Insertion Sort works well for mostly sorted data, but it may not be the best choice for more complicated tasks.

Merge Sort is the best of the three algorithms, performing well across all datasets. It handles different sorted data easily and does not slow down, making it a great choice for different situations. No matter how the data is arranged, Merge Sort consistently gives good results. It is suitable for complex datasets where speed and efficiency really matter.

Table 1.2- Different Sizes

Bubble Sort's execution time increases quickly as the dataset grows, with a more than doubling jump from 3 seconds at 30K elements to over 8 seconds at 50K. This shows that Bubble Sort struggles with larger data sizes, making it inefficient for larger datasets.

Insertion Sort performs better than Bubble Sort but still experiences increasing delays as the dataset grows. The execution time is under 2 seconds at 50K elements, which is better than Bubble Sort but may still be too slow for very large datasets. Therefore we can say Insertion Sort works well on small to medium datasets, but it may not be the best choice as data size increases.

Merge Sort is the fastest algorithm in this comparison, with almost no difference in execution time as the data size grows. Even with 50K elements, it completes in less than a second, therefore it is highly efficient for larger datasets. Merge Sort's stable performance across different sizes makes it a good choice for handling large amounts of data.

1.2. Targeted Searches and Key Metrics

Run a binary search for the index 1773335. Search for the number of tweets with more than 250 favorites on the datasets given in the header row of Table 1.3. Provide the execution time in the related cells. Make sure to provide the unit.

	5K	**10K**	**20K**	**30K**	**50K**
Binary Search	13.7510^{-7} s	13.3310^{-7} s	8.3410^{-7}s	8.7510^{-7} s	8.7610^{-7} s
Threshold	$3.05x10^{-5}$ s	$4.87x10^{-5}$ s	$7.17x10^{-5}$ s	$7.73x10^{-5}$ s	$10.8x10^{-5}$ s

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

Discuss your results

Table 1.3- Different Metric Algorithms

Binary Search shows low and stable execution times across all dataset sizes, with small differences. As the dataset grows from 5K to 50K, the time remains close, so we can say that larger datasets does not create an impact on the algorithm. This makes Binary Search a reliable choice for searching tasks, even as the data size increases significantly. Also it can be observed from the table that after 20K level Binary Search works faster. When researched on this topic, I discovered that my algorithm performs better with larger data because the CPU manages larger lists more efficiently within the memory of my computer.

Threshold's execution time is directly affected by the dataset size. This suggests that Threshold becomes slower with larger data, so it may not be ideal for very large datasets. While it may still work well for smaller datasets, the increase in time shows it may struggle with efficiency as data size increases.

1.3. Discussion Questions

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

Bubble Sort repeatedly goes through the list, comparing and swapping adjacent elements if they are out of order. It keeps sorting even if the list is mostly sorted, making it slow for large or random lists. With a complexity of $O(n^2)$, Bubble Sort is faster on nearly sorted lists because it does not need as many swaps.

Insertion Sort works by gradually building a sorted part of the list, inserting each new element in the right place by shifting larger elements over. This is efficient for small or nearly sorted lists since it minimizes unnecessary comparisons. While it has a worst-case complexity of $O(n^2)$, its efficiency improves to $O(n)$ with nearly sorted data, making it useful for situations requiring less sorting.

Merge Sort splits the list into halves, sorts each half, and then combines them in order. This method is consistent and handles all types of input efficiently, with a time complexity of $O(nlogn)$. Merge Sort is better for larger lists or data with random order because it divides the work evenly.

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

Binary Search is efficient but it has specific limitations. It only works on sorted data, so it fails on unsorted lists. It is also designed for data structures that support quick and direct access so it may not be usable with linked lists. When there are duplicate values, Binary Search will not consistently find the first or last occurrence without additional steps. On very large datasets, calculating the midpoint as *(left + right) / 2* can cause overflow and to avoid this, it is calculated as *left + (right - left) / 2* but this can still be a limitation in some cases with large numbers.

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?

Merge sort performs consistently on sorted or identical datasets, because it always splits and merges data the same way regardless of the initial order. Unlike algorithms that can optimize for sorted data, merge sort does not speed-benefit in these cases and maintains a stable $O(nlogn)$ time complexity.

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

When tested with the tweets.csv file, bubble sort, insertion sort, and merge sort performed similarly for both ascending and descending orders. **Bubble Sort** made the same number of movements no matter which way it was sorting, showing consistency. **Insertion Sort** also worked well with ascending and descending, completing its tasks effectively. **Merge Sort** maintained its usual performance, remaining stable whether sorting ascending or descending. This consistency across all algorithms shows that the randomized input provided a fair challenge for each sorting method.

However, as seen in Table 1.1, Bubble Sort's performance can be affected by the inputs permutation, since it depends on the number of swaps needed. Insertion sort is also sensitive to input order, while merge sort stays stable regardless of the input arrangement.