Analysis of Algorithms

BLG 335E

Project 1 Report

ALPER TUTUM
tutum21@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	8678 ms	3130 ms	7143 ms	3416 ms
Insertion Sort	1962 ms	0.4 ms	3929 ms	114 ms
Merge Sort	41 ms	37 ms	42 ms	44 ms

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

	5K	**10K**	**20K**	**30K**	**50K**
Bubble Sort	105 ms	346 ms	1393 ms	3092 ms	8727 ms
Insertion Sort	38 ms	106 ms	330 ms	714 ms	1968 ms
Merge Sort	8 ms	16 ms	22 ms	35 ms	49 ms

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

Discuss your results

Looking at Table 1.1, it can be seen that while the different permutations of data impact the efficiency of bubble and insertion sort, merge sort takes approximately the same time in each scenario. Additionally, it can be observed that insertion sort's efficiency is more sensitive against permutation changes in the data when compared to bubble sort.

Table 1.2, shows how the time complexity of each sorting algorithm scales depending on the size of the input. In terms of how time increases as a reaction to increasing size of data, bubble sort is the fastest growing algorithm while merge sort is the slowest.

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	3 µs	3 µs	2 µs	1 µs	2 µs
Threshold	38 µs	62 µs	102 µs	132 µs	217 µs

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

Discuss your results

As can be seen in Table 1.3, the time it takes for binary search to execute depends on the permutation of the data. Larger data doesn't necessarily mean binary search will take more time however it is still a factor. The table also demonstrates how the time length of $countAboveThreshold()$ increases linearly with the size of the input data.

1.3. Discussion Questions

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

The time complexity of each method implemented in this projects are:

- **Bubble sort:** $\Theta(n^2)$
- **Insertion sort:** $O(n^2)$, $\Omega(n)$
- **Merge sort:** $\Theta(n * log_2 n)$
- **Binary search:** $O(log_2 n)$
- **Threshold:** $O(n)$

From the sorting algorithms, it can be said that merge sort is overall the fastest algorithm by looking at both the time complexities and results in Tables 1.1 and 1.2. Although in its best case, insertion sort is faster than all others.

Binary search and threshold methods have smaller complexities when compared to the sorting algorithms. From these two, binary search is clearly more efficient due to its logarithmic time complexity against threshold's linear.

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

Binary search can only be applied to sorted lists of data since its algorithm relies on it. The algorithm checks the middle element of the array and compares it with the value it is trying to find. One of the cases is that if the value being searched is greater than the middle element's value, binary search keeps searching the indexes greater than the index of the middle element. This is because binary search takes for granted that the array is sorted in the correct order and values greater than the middle element will be to its right. That is why binary search will result in errors if the list isn't already sorted in ascending order.

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 long as the data size is constant, the performance of merge sort stays the same regardless of if the dataset is sorted or not. This means the upper and lower bounds of merge sort's complexity are the same.

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

For insertion sort and bubble sort, the changes in performance are visible when the data is sorted. For insertion sort the reason is that the inner loop of the algorithm performs the insertion operation nearly n times depending on the while condition. If the data is already sorted in the correct order, the code inside the inner loop always gets skipped, leaving only the outer loop to run $n - 1$ times, resulting in linear time complexity. Otherwise, insertion sort works in quadratic time complexity. In the worst case, which is if the data is sorted but in the wrong order, the time complexity is still quadratic but it takes twice the time of the average case. The results in Table 1.1 support this.

Even though bubble sort has a notable change in performance when the order of input is different, the complexity function is still quadratic. This is because the outer loop runs for approximately n times regardless, and the inner loop also runs for nearly n times. The difference is that in the inner loop, only the if condition is guaranteed to run proportional to n. The if block however only runs when the condition holds thus in the best case, which is when the data is already sorted in the correct order, the block of code after the condition is never executed, making a big difference in performance.

Merge sort algorithm is recursive and the function $mergeSort()$ is always called $log_2(n + 1)$ times. For every time it is called, $merge()$ will take n steps to compare the elements in the sub-arrays and merge them, again regardless of the order of the input data. This will result in a time complexity of $\Theta(n * (log_2 n))$ for every case.