Analysis of Algorithms

BLG 335E

Project 1 Report

Ali Emre Kaya
kayaemr21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 29.10.2024

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.

ms	**tweets**	**tweetsSA**	**tweetsSD**	**tweetsNS**
Bubble Sort	70767	2	71187	66741
Insertion Sort	33575	2	67944	1739
Merge Sort	55	30	42	45

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

ms	**5K**	**10K**	**20K**	**30K**	**40K**	**50K**
Bubble Sort	684	2663	10717	24083	44287	67186
Insertion Sort	290	1141	4687	10352	20515	28736
Merge Sort	3	7	18	26	43	46

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

Discuss your results

First, I sort all the data in ascending order based on the retweetCount. I chose to focus on retweetCount because I observed that datas included several arranged counts. Displaying these outputs can provide valuable insights into time complexities.

The performance results illustrate the stark contrast between quadratic and linearithmic time complexities. Bubble Sort and Insertion Sort, both having $O(N^2)$ complexity, show significantly higher execution times across various input sizes and permutations. In contrast, Merge Sort, with its $O(N \log N)$ complexity, maintained consistently low execution times, averaging under 50 milliseconds for all cases. This confirms that Merge Sort is far superior for larger datasets and different permutations, making it the preferred choice for efficient sorting. But insertion sort and bubble sort shine in edge cases, particularly when sorting already sorted data, achieving optimal time complexity of $O(n)$ in such scenarios. When both algorithms struggle with arrays sorted in descending order, where insertion sort must shift every element and bubble sort makes numerous swaps, resulting in a worst-case time complexity of $O(n^2)$. This contrast highlights how the simplicity of these algorithms can lead to impressive efficiency in favorable conditions, while also revealing their limitations when faced with reverse-sorted datasets.

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.

ms	**5K**	**10K**	**20K**	**30K**	**40K**	**50K**
Binary Search	0.0037	0.0042	0.0037	0.0040	0.0037	0.0040
Threshold	0.0139	0.0274	0.0681	0.0816	0.1045	0.1311

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

Discuss your results

Binary Search locates the target in the array in $O(\log N)$ time by comparing the key value with left and right values. Since the array is already sorted in ascending (or descending) order, if the key is less than the middle value, the search interval is restricted to the left of the mid index in the case of ascending order. Conversely, if the array is sorted in descending order, the search would continue to the right half.

The logic for the threshold is straightforward: it involves traversing the array and counting values only when they exceed the specified threshold which time complexity is O(n).

1.3. Discussion Questions

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

The first method implemented is Bubble Sort, which has a time complexity of $O(N^2)$. The second method, Insertion Sort, also exhibits a time complexity of $O(N^2)$. However, both of these algorithms have a significant advantage in terms of memory usage, as they operate with constant space complexity of $O(1)$,and their best-case time complexity is $O(N)$ when the input is already sorted.

The third method is Merge Sort, which follows the divide, sort, and merge approach. Its time complexity is $O(N \log N)$, with a space complexity of $O(N)$ because it requires an auxiliary array during the sorting process.

Additionally, I implemented a binary search algorithm to locate the address of a specific value. This algorithm works by repeatedly halving the search interval until the target value is found, resulting in a time complexity of $O(\log N)$.

To identify values greater than a specified threshold, I employ a straightforward $O(N)$ for loop. However, if the array is sorted, a binary search approach could be more efficient which time complexity is $O(\log N)$. Since I'm not certain whether the array is sorted or not, I designed the program to be reliable for all input scenarios.

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

One fundamental limitation of binary search is that it can only be applied to sorted arrays. This requirement imposes a critical constraint on the dataset. Additionally, when multiple identical key values are present, the efficiency of binary search may be diminished. However, it remains usable in such scenarios. For instance, in the threshold function, I employed binary search even with numerous identical retweetCount values. After executing the binary search, I applied a straightforward logic to effectively manage these duplicates.

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 consistently maintains a time complexity of $O(N \log N)$, regardless of the dataset's characteristics, such as being already sorted or containing identical values. This stability exemplifies its robustness.

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

There is no distinction among the three sorting algorithms regarding their ability to sort data in either ascending or descending order; their time and space complexities remain unchanged. This consistency arises because the fundamental logic of sorting does not alter; only the method of comparison varies based on the desired order.