Analysis of Algorithms

BLG 335E

Project 1 Report

EMRE YAZICI
150220063@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 26.10.2024

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**
Insertion Sort	1559.41 ms	8.602 ms	3115.11 ms	95.462 ms
Merge Sort	10.749 ms	6.965 ms	6.057 ms	6.316 ms

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

	5K	**10K**	**20K**	**30K**	**50K**
Bubble Sort	93.3 ms	403.859 ms	1700.49 ms	3739.25 ms	11136.1 ms
Insertion Sort	19.609 ms	70.313 ms	284.394 ms	619.626 ms	1737.25 ms
Merge Sort	0.942 ms	2.373 ms	4.651 ms	6.518 ms	10.278 ms

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

When comparing these three sorting algorithms, we observe that merge sort has a significant advantage in average run-time complexity. This advantage is because both the average and worst-case time complexities of merge sort are $O(n \log n)$, whereas those of the other algorithms are $O(n^2)$. However, as shown in Table 1.1, insertion sort may perform with a significant advantage on certain permutations of the same dataset. This is because the insertion sort operates in $O(n)$ time in the best case.

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	0.001 ms	0.002 ms	0.002 ms	0.004 ms	0.004 ms
Threshold	0.011 ms	0.020 ms	0.048 ms	0.075 ms	0.187 ms

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

As observed in the results from the table, the time complexity of binary search is $O(\log n)$, since the search space is halved with each iteration. This provides a query advantage in sorted datasets.

On the other hand, in the code used to count the occurrences of favoriteCounts exceeding a threshold, the time complexity is $O(n)$, as it simply iterates through all elements. This is also reflected in the results in the table.

1.3. Discussion Questions

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

Binary Search

Since we halve the search space at each step, the search process consists of approximately $\log_2 n$ steps, resulting in a time complexity of $O(\log n)$.

Count Above Threshold

Using a for loop, we iterate through the entire dataset and increment our output for each element that meets the condition. In brief, we traverse all elements once. Thus, the time complexity is $O(n)$.

Bubble Sort

In each step, we traverse all elements, and with each iteration, we reduce the size of the unsorted portion by one. This results in a time complexity of $1 + 2 + 3 + \cdots + n$, which simplifies to $O(n^2)$.

Insertion Sort

We use a for loop to iterate through the array from start to finish, placing each element in its appropriate position within the already sorted portion of the array. This way, as we proceed, the section behind each new element is always sorted. For each new element, we find the insertion point using binary search and then insert it. If the new element is greater than the previous one, neither search nor insertion is needed.

In general, the iteration has complexity $O(n)$, the binary search $O(\log n)$, and the insertion $O(n)$, leading to an overall time complexity of $O(n) \cdot (O(\log n) + O(n)) = O(n^2)$.

In the best case, if the array is already sorted, neither search nor insertion occurs. Thus, the time complexity is $O(n) \cdot O(1) = O(n)$.

Merge Sort

In each step, we split our array into halves until we obtain single-element parts. We then merge these parts in the recursive order to form a sorted array. During the merge process, we iterate through both parts by comparing elements from each and adding them in sorted order to a new array. Thus, for each merge, we must traverse all elements in the parts being merged. Since the array is divided in $\log_2 n$ steps, and each step requires $O(n)$ time for the merge operations, the overall time complexity is $O(n \log n)$.

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

If our dataset is not sorted, comparing the target element with the middle element does not provide any information about its location. Therefore, binary search cannot be applied in this case.

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?

Regardless of the case, we will need $\log n$ steps to reach single-element arrays. Additionally, for each merge operation, we must iterate through the elements individually. This means that independent of the case, there will be $\log n$ steps, and each step will require $O(n)$ time complexity. Therefore, merge sort is not visibly affected by edge cases.

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

Since we only need to switch between greater-than and less-than comparisons between elements, there will be no additional workload. Therefore, there will be no performance difference.