Analysis of Algorithms

BLG 335E

Project 1 Report

Taha TEMİZ
temizt21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 03.11.2024

1. Implementation

1.1. Sorting Strategies for Large Datasets

	tweets	tweetsSA	tweetsSD	tweetsNS
Bubble Sort	20357ms	8870ms	25217ms	9192ms
Insertion Sort	5443ms	0.52ms	10805ms	278ms
Merge Sort	39ms	31ms	30ms	34ms

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

	5K	10K	20K	30K	50K
Bubble Sort	170ms	694ms	2883ms	6957ms	20634ms
Insertion Sort	53ms	212ms	887ms	2042ms	5506ms
Merge Sort	3ms	7ms	14ms	21ms	38ms

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

Discussion

First, it is obvious that the merge sort algorithm is the most efficient one and the bubble sort algorithm is the most inefficient one, as seen in Table 1.1 and Table 1.2. The bubble sort algorithm and the insertion sort algorithm sort the permutation tweet vectors in different durations, as seen in Table 1.1. Both of them execute slowly when the vector is sorted descending. As seen in Table 1.1, the permutation of the tweet vector does not affect the duration of the merge sort algorithm much. The merge sort algorithm works in permutation vectors more stably than the other algorithms do. It is interesting that the bubble sort algorithm wastes time even in the sorted tweet vector. Moreover, the insertion sort algorithm sorts the already sorted tweet vector faster than the other algorithms because it is the best-case scenario for the insertion sort algorithm, as seen in Table 1.1. It even works faster than the merge sort algorithm. The sort durations of different-sized vectors with the sort algorithms match with the time complexity calculations. The time complexity of the bubble sort algorithm is O(n^2). The time complexity of the insertion sort algorithm is O(n^2) (average case). The time complexity of the merge sort algorithm is O(n$\log n$). As seen in Table 1.2, the ratio of durations to each other matches with the time complexity.

1.2. Targeted Searches and Key Metrics

	5K	10K	20K	30K	50K
Binary Search	1.7µs	2.7µs	3.6µs	3µs	3.2µs
Threshold	61µs	95µs	220µs	286µs	466µs

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

Discussion

The time complexity of the binary search algorithm is O($\log n$). As seen in Table 1.3, the binary search durations for the vector with 5000 tweets, for the vector with 10000, and for the vector with 20000 tweets are understandable. But the binary search durations for the vector with 30000 tweets and for the vector with 50000 tweets do not fit the calculations. I think it is about the smallness's of unit measure. Search duration should increase with the data size. Moreover, the counting algorithm has O(n) time complexity. As seen in Table 1.3, the counting duration increases with the data size. The ratio of durations to each other matches with the time complexity.

1.3. Discussion Questions

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

For bubble sort, I implemented a method that compares the pairs from the start of the vector until the end of the vector. It is repeated n times. Every time comparing the pairs is done, the biggest element or the lowest element is sorted so the range is decreased one. It is like comparing n elements n times, so the time complexity of this method is O(n^2). For insertion sort, I implemented a method that compares each of every element in the tweet vector with previous tweets. So the time complexity of the insertion sort algorithm is O(n^2). For the merge sort algorithm, there are two functions: merge and merge sort. The merge function takes a vector as input and it divides the vector into two sub-vectors then creates a sorted vector with the same input size by using the sub-vectors. The merge sort function merge sorts the left part of the given vector then merge sort the right part of the given vector and finally merge the left part and the right part. So the merge sort function calls itself. The merge sort function divides the given vector into elements one by one then with the merge function collects the pieces into a vector while sorting. The time complexity of dividing is O($\log n$) because the algorithm divides the vector into two pieces repeatedly. Then it checks all the element at least once so the time complexity of merging is O(n). Therefore, the time complexity of the merge sort algorithm is O(n$\log n$). At first, for all of the three sorting algorithms, I created 6 blocks (ascending-tweetID,descending-tweetID,etc.) . But for the clarity of the code, I added

conditional operators to the if blocks. So the block number is decreased to 3. There was a slight increase in the sorting durations. My another idea was decreasing the block number to 1. But it would be very slow. So I did not. For binary search, I implemented a method that checks the middle indexed value recursively. If the key is bigger than the middle indexed value then the new lower bound is middle index + 1(Because we already checked the middle indexed value). If the key is smaller than the middle indexed value then the new upper bound is middle index - 1. The algorithm halves the vector repeatedly until finding the key. So the time complexity of this algorithm is O($\log n$). For counting above threshold, I implemented a method that checks every element of the vector. If the checked attribute of the current element is higher than the threshold then it counts. This method checks all the elements of a vector so the time complexity is O(n).

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

The data should be sorted. Because in an unsorted list, the middle indexed elements do not have to be middle valued, so luck would be needed. It is not impossible to find the element but, it is hard and unnecessary. For example, the binary search algorithm found the tweet given in assignment, inside the permutations/tweets.csv. The binary search algorithm needs to reach the middle indexed element easily. So if the data is stored in a dynamic structure like linked list. The efficiency would be reduced because it would be hard to reach the middle indexed element. Moreover, duplicated elements would be a problem for binary search. The algorithm can find the index of the searched element. But which one of them?

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?

The merge sort algorithm sorts an already sorted dataset or a dataset where all tweet counts are the same a little bit faster. But the improvement in these cases is not significant. Because the algorithm does not care about the current situation of the dataset. It always divides the dataset into pieces, then merges the pieces. The number of swaps has decreased, but as I mentioned before, it is not a significant change.

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

No, there were not any notable performance differences. I think it is because the number of comparisons is not changed notably when the order is ascending or descending.

Even the code is not changing that much. Just a "<" or ">". The algorithm works nearly the same, but it is choosing the bigger one or the smaller one.