Analysis of Algorithms

BLG 335E

Project 1 Report

Kemal Tahir Bıcılıoğlu
bicilioglu21@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	54305ms	2ms	58517ms	43937ms
Insertion Sort	27011ms	1ms	55560ms	1460ms
Merge Sort	60ms	49ms	53ms	63ms

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

	5K	**10K**	**20K**	**30K**	**40K**	**50K**
Bubble Sort	515ms	2143ms	8058ms	14629ms	25780ms	40657ms
Insertion Sort	219ms	891ms	3506ms	7735ms	13676ms	21571ms
Merge Sort	5ms	12ms	24ms	32ms	50ms	59ms

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

Discuss your results

- It's also important to note that the datasets in Table 1.1 were sorted by retweetCount, showing different permutations based on this parameter (ascending, descending, nearly sorted, and random). In Table 1.2, the datasets were sorted by tweet ID, as no specific sorting parameter was given for those sizes. For Bubble Sort and Insertion Sort, the arrangement of the initial data (ascending, descending, or nearly sorted) significantly influences execution time. Bubble Sort performs particularly well on data already sorted in ascending order (tweetsSA), completing in only 2ms by recognizing the sorted order early and terminating quickly. However, Bubble Sort struggles with data sorted in descending order (tweetsSD), where every element must be repositioned, leading to the highest recorded time of 58,517ms. Similarly, Insertion Sort achieves optimal performance on ascending datasets (1ms on tweetsSA) as it proceeds linearly without shifting elements, but on descending data (tweetsSD), it has to make repeated shifts to reorder elements in the opposite order, yielding the worst runtime of 55,560ms. With nearly sorted data (tweetsNS), both Bubble and Insertion Sort perform reasonably well, benefiting from minimal reordering. We observe a significant performance difference between Insertion Sort and Bubble Sort in the average case, where Insertion Sort outperforms due to its more efficient swap strategy. Unlike Bubble Sort, which checks each adjacent

pair repeatedly, Insertion Sort directly places elements in their correct positions with fewer swaps.In contrast, Merge Sort's performance remains consistently efficient across all datasets, with only minor timing variations (49ms to 63ms). The merge-based divide-and-conquer approach effectively disregards the data's initial arrangement, yielding stable $\mathcal{O}(n \log n)$ runtimes regardless of whether the data is sorted, reverse-sorted, or randomized.

- In Table 2, the running times increase with dataset size consistent with the time complexities of the algorithms. For instance, as the data grows from 5K to 50K, the execution time grows proportional to n^2 (from 515ms to 40657ms) for bubble sort since its worst case time complexity is $\mathcal{O}(n^2)$. Even if the time complexity of the insertion sort is also $\mathcal{O}(n^2)$ it requires fewer operations in practice because it tends to move elements only when necessary. This exponential growth further illustrates Bubble Sort's and Insertion Sort's unsuitability for larger datasets. We can see the significant running time difference between the Merge Sort to Bubble Sort and Insertion Sort. It also does not grow exponentially for larger datasets because of its time complexity $\mathcal{O}(n \log n)$. We can conclude that the merge sort is suitable for larger 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.

	5K	**10K**	**20K**	**30K**	**40K**	**50K**
Binary Search	0ms	0ms	0ms	0ms	0ms	0ms
Threshold	0ms	0ms	0ms	1ms	1ms	2ms

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

Discuss your results

- The execution time for Binary Search remains at 0ms for datasets of size 5K to 50K. This indicates that the algorithm operates very efficiently, even as the dataset size increases. Binary Search achieves $\mathcal{O}(logn)$ complexity, which allows it to quickly narrow down the search space, resulting in negligible execution time for reasonably sized datasets.

- The execution time for the threshold counting function starts at 0ms for smaller datasets (5K to 20K) and increases to 1ms for the 30K dataset, 1ms for the 40K dataset, and 2ms for the 50K dataset. This trend suggests that while the function operates quickly, there is a slight increase in time as the dataset size grows. This

aligns with the expected performance of an $\mathcal{O}(n)$ algorithm, as the function iterates over the dataset to count elements above a specified threshold.

1.3. Discussion Questions

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

1. Bubble Sort
   (a) Time Complexity
      i. Best Case: $\Omega(n)$
      ii. Average Case: $\theta(n^2)$
      iii. Worst Case: $\mathcal{O}(n^2)$
      iv. Space Complexity: $\mathcal{O}(1)$
   (b) Advantages:
      i. Easy to understand and implement.
      ii. Not swapping any element in the inner loop and stopping the outer loop, Bubble Sort can detect already sorted arrays which saves time in a best-case scenario.
      iii. In-place sorting. It does not require additional space. Suitable for situations with limited memory.
      iv. Stable Sorting. Bubble sort preserves the relative order between the objects with the same key.
   (c) Disadvantages:
      i. It is inefficient for large datasets.
      ii. It can do many unnecessary checks and swaps.
2. Insertion Sort
   (a) Time Complexity
      i. Best Case: $\Omega(n)$
      ii. Average Case: $\theta(n^2)$
      iii. Worst Case: $\mathcal{O}(n^2)$
      iv. Space Complexity: $\mathcal{O}(1)$
   (b) Advantages:
      i. Easy to understand and implement.
      ii. Insertion Sort is efficient on small datasets and nearly sorted lists, where it often performs fewer operations than Bubble Sort.

iii. In-place sorting. It does not require additional space. Suitable for situations with limited memory.

iv. Stable Sorting. Insertion Sort preserves the relative order between the objects with the same key.

(c) Disadvantages:

i. More efficient algorithms like Merge Sort or Quick Sort outperform Insertion Sort for large datasets due to their better time complexities.

3. Merge Sort

(a) Time Complexity

i. Best Case: $\Omega(n \log n)$

ii. Average Case: $\theta(n \log n)$

iii. Worst Case: $\mathcal{O}(n \log n)$

iv. Space Complexity: $\mathcal{O}(n)$

(b) Advantages:

i. Merge Sort's complexity allows it to handle large datasets effectively, regardless of the initial order.

ii. Stable Sorting. Merge Sort preserves the relative order between the objects with the same key.

(c) Disadvantages:

i. Space Requirement: Merge Sort's additional space requirement can be a drawback, particularly for large datasets or memory-limited systems.

ii. The recursive nature of Merge Sort can lead to stack overflow issues on systems with limited stack space when sorting very large datasets.

4. Binary Search

(a) Time Complexity

i. Best Case: $\Omega(1)$

ii. Average Case: $\theta(logn)$

iii. Worst Case: $\mathcal{O}(logn)$

iv. Space Complexity: $\mathcal{O}(1)$

(b) Advantages:

i. Efficiency: Binary Search is highly efficient, especially on large datasets, thanks to its time complexity.

ii. Simplicity: The algorithm is relatively simple to implement and understand.

iii. When implemented iteratively, Binary Search requires minimal additional memory.

(c) Disadvantages:

i. Requires Sorted Data: Binary Search can only be applied to sorted arrays, which can add preprocessing time if the data isn't already sorted.

ii. Binary Search relies on direct access to middle elements, which makes it inefficient for linked lists, where elements are accessed sequentially.

5. Count Above Threshold

(a) Time Complexity

i. Best Case: $\Omega(n)$

ii. Average Case: $\theta(n)$

iii. Worst Case: $\mathcal{O}(n)$

iv. Space Complexity: $\mathcal{O}(1)$

(b) Advantages:

i. Straightforward and Minimal Memory Usage.

(c) Disadvantages:

i. It could have been implemented using a binary search function which would make the complexity $\mathcal{O}(logn)$ from $\mathcal{O}(n)$. We would find the upper bound for the key then $size - foundIndex$ would give the answer. However, we should do preprocessing to sort the data.

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

- Binary Search only works on data sorted in ascending or descending order. If the data isn't sorted, the algorithm cannot correctly determine which half of the array to search next.
- When there are duplicate elements, Binary Search can find one occurrence of the target value but not necessarily all. Additional logic is needed to handle duplicates. The reason why there should be additional logic is that Binary Search returns the first match it finds, but if there are duplicates, it cannot guarantee which occurrence (first, last, or any) it will locate.
- Data structures like sets, maps, or associative containers don't allow integer indexing, so Binary Search cannot be directly applied to them without transforming the data into an indexed format.

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 does not benefit from an already sorted dataset because it divides and merges each subarray regardless of initial order. The algorithm consistently applies the divide-and-conquer strategy, making its time complexity regardless of the order, so an already sorted dataset doesn't improve or degrade its performance.

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

- There were no notable differences in performance for most test cases; however, if the data is already sorted in ascending order, Bubble Sort and Insertion Sort may perform better, while sorting them in descending order can lead to worse performance than other runtime results. In contrast, Merge Sort remains unaffected by the sorting direction, consistently maintaining its time complexity regardless of whether the order is ascending or descending.