Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Lab Report on "Quick Sort & Merge Sort Time Complexity Analysis"

[Code No. : COMP 314]

Submitted by:

Prafful Raj Thapa (59)

Submitted to:

Dr. Rajni Chulyadyo

Department of Computer Science and Engineering

December 28, 2024

Contents

List of Figures

List of Tables

Chapter 1 : Quick Sort

1.1 Quick Sort Algorithm

Quick Sort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. Quick Sort breaks down the problem into smaller sub-problems. It selects an element from an array as the pivot(the choices can vary), then rearranges the around the pivot. After partitioning, all the elements smaller than the pivot will be on the left, and elements grater than pivot will be on the right. Quick Sort recursively apply the same process to the partitioned sub-array(left and right of pivot). The best case time complexity of the quick sort algorithm is $O(nlogn)$, where n is the number of elements in the list or array. This means the time taken for sorting increases logarithmic with the scale of the array.

1.2 Time Complexity Analysis

1.2.1 Methodology

The implementation and evaluation of the Quick sort algorithm were conducted systematically to ensure reliable and reproducible results. The process involved the following steps:

1. **Algorithm Implementation and Testing:** The quick sort algorithm was implemented in Rust. Its correctness was verified using the Rust `testing` library to ensure accuracy and consistency under various conditions.

2. **Data Generation for Time Analysis:** A separate script was written to generate datasets of varying sizes, specifically for analyzing the algorithm's time complexity. The datasets included lists with sizes of 100 elements, 1,000 elements, 10,000 elements, 100,000 elements, 300,000 elements and progressively larger sizes up to 5,000,000 elements (e.g., 500,000, 1,000,000, 1,500,000, and so on, in increments of 500,000).

3. **Case Analysis:** For each dataset, the algorithm's performance was measured in three scenarios:

    - **Best Case:** An array of randomly generated data was sorted using quick sort.
    - **Average Case:** As the average case time complexity was the same as in best case this test case was eliminated.
    - **Worst Case:** The target array is an already sorted. But due to n depth of recursion for array of size n. This test is not viable for data of larger size.

4. **Time Measurement:** The time taken by the algorithm to sort the target array was recorded. These timings were consolidated into a table for comparison and analysis.

5. **Visualization:** The recorded data was visualized in a graph, providing a clear representation of the relationship between dataset size and algorithm performance under each scenario.

This approach ensured that the analysis captured the quick sort algorithm's behavior across a wide range of input size, offering valuable insight into its efficiency & scalability.

1.2.2 Data

The algorithm's execution time was measured for best-case, average-case, and worst-case scenarios. The results were compiled into a table 1.1 and visualized through a graphs 1.1 to illustrate the relationship between input size and search performance.

Data Size	**Time Taken**		
	Best Case	**Average Case**	**Worst Case**
100	0.000055047S	NA	NA
1,000	0.00027251S	NA	NA
10,000	0.00340056S	NA	NA
100,000	0.04116563S	NA	NA
300,000	0.129006793S	NA	NA
500,000	0.24777136S	NA	NA
1,000,000	0.519681379S	NA	NA
1,500,000	0.740276624S	NA	NA
2,000,000	1.04142036S	NA	NA
2,500,000	1.317598602S	NA	NA
3,000,000	1.535092795S	NA	NA
3,500,000	1.847271043S	NA	NA
4,000,000	2.136620001S	NA	NA
4,500,000	2.437808963S	NA	NA
5,000,000	2.722196567S	NA	NA
5,500,000	2.924174204S	NA	NA
6,000,000	3.266431163S	NA	NA
6,500,000	3.680379315S	NA	NA
7,000,000	3.740404506S	NA	NA
7,500,000	4.080480053S	NA	NA
8,000,000	4.6585537S	NA	NA
8,500,000	4.865868873S	NA	NA
9,000,000	5.531628484S	NA	NA
9,500,000	5.616552614S	NA	NA
10,000,000	5.885646513S	NA	NA

Table 1.1: Time complexity Analysis Data of Quick Sorting Algorithm

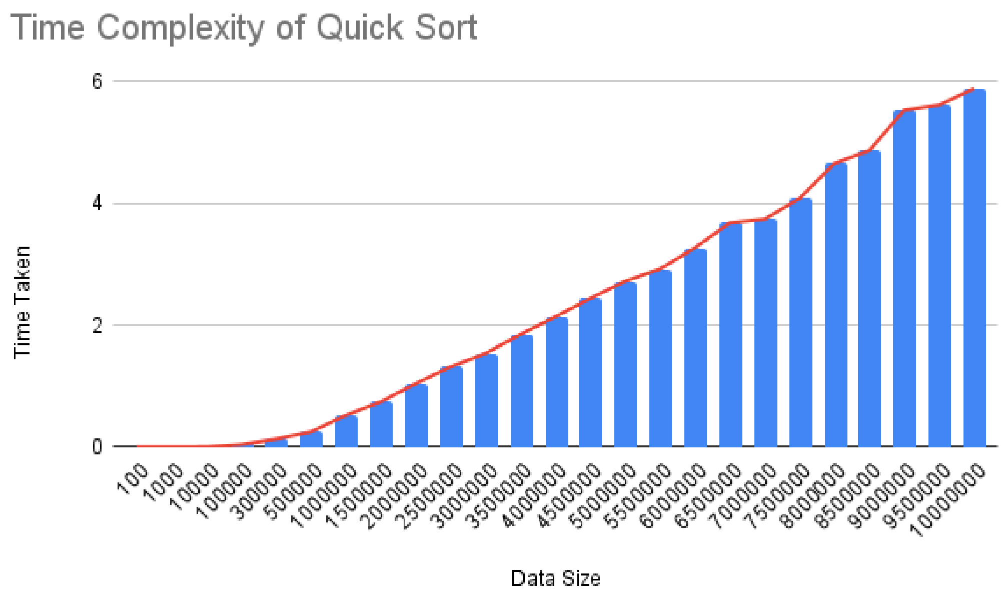

Figure 1.1: Time Complexity of Quick Sort

1.2.3 Analysis Result

The analysis of the gathered data and corresponding graphs reveals distinct patterns in the performance of the quicksort algorithm. In the best-case scenario, where the pivot consistently partitions the list into two equal halves, the algorithm demonstrates O(nlogn) execution time. This behavior aligns with expectations, as balanced partitions minimize recursion depth and maintain efficient processing.

For the average-case scenario, the graph shows a similar O(nlogn)relationship between dataset size and execution time. This reflects the algorithm's tendency to create moderately balanced partitions on average, leading to efficient sorting with manageable recursion depth.

In the worst-case scenario, where the pivot repeatedly partitions the array into one large and one empty subarray (e.g., for sorted or reverse-sorted data), the graph reveals a quadratic relationship. The algorithm must perform O(n2) operations due to excessive recursion depth and unbalanced partitions, significantly increasing execution time.

Overall, the analysis confirms that quicksort performs optimally with well-chosen pivots, scaling efficiently for average and best cases, but suffers severe performance degradation in the worst-case scenario without pivot optimization.

Chapter 2 : Merge Sort

2.1 Merge Sort Algorithm

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted. The time complexity of the merge sort algorithm is $O(nlogn)$ in all three scenario (Best, Average, Worst case). This means the time taken for sorting increases logarithmic with the scale of the array.

2.2 Time Complexity Analysis

2.2.1 Methodology

The implementation and evaluation of the Merge sort algorithm were conducted systematically to ensure reliable and reproducible results. The process involved the following steps:

1. **Algorithm Implementation and Testing:** The merge sort algorithm was implemented in Rust. Its correctness was verified using the Rust `testing` library to ensure accuracy and consistency under various conditions.

2. **Data Generation for Time Analysis:** A separate script was written to generate datasets of varying sizes, specifically for analyzing the algorithm's time complexity. The datasets included lists with sizes of 100 elements, 1,000 elements, 10,000 elements, 100,000 elements, 300,000 elements and progressively larger sizes up to 5,000,000 elements (e.g., 500,000, 1,000,000, 1,500,000, and so on, in increments of 500,000).

3. **Case Analysis:** For each dataset, the algorithm's performance was measured in three scenarios:

    - **Best Case:** An array of randomly generated data was sorted using merge sort.
    - **Average Case:** As the average case time complexity was the same as in best case this test case was eliminated.
    - **Worst Case:** As the worst case time complexity was the same as in best case this test case was eliminated.

4. **Time Measurement:** The time taken by the algorithm to sort the target array was recorded. These timings were consolidated into a table for comparison and analysis.

5. **Visualization:** The recorded data was visualized in a graph, providing a clear representation of the relationship between dataset size and algorithm performance under each scenario.

This approach ensured that the analysis captured the binary search algorithm's behavior across a wide range of input size, offering valuable insight into its efficiency & scalability.

2.2.2 Data

The algorithm's execution time was measured for best-case, average-case, and worst-case scenarios. The results were compiled into a table 2.1 and visualized through a graphs 2.1 to illustrate the relationship between input size and search performance.

Data Size	**Time Taken**		
	Best Case	**Average Case**	**Worst Case**
100	0.00007006S	NA	NA
1,000	0.000423117S	NA	NA
10,000	0.004986037S	NA	NA
100,000	0.057760778S	NA	NA
300,000	0.187911155S	NA	NA
500,000	0.323688189S	NA	NA
1,000,000	0.674096929S	NA	NA
1,500,000	1.073920581S	NA	NA
2,000,000	1.457581284S	NA	NA
2,500,000	1.869643479S	NA	NA
3,000,000	2.250004228S	NA	NA
3,500,000	2.65169339S	NA	NA
4,000,000	3.010853775S	NA	NA
4,500,000	3.511088165S	NA	NA
5,000,000	3.886548585S	NA	NA
5,500,000	4.312413096S	NA	NA
6,000,000	4.771077962S	NA	NA
6,500,000	5.068592918S	NA	NA
7,000,000	5.476072905S	NA	NA
7,500,000	5.985026945S	NA	NA
8,000,000	6.282098211S	NA	NA
8,500,000	6.66134367S	NA	NA
9,000,000	7.261410196S	NA	NA
9,500,000	7.510780779S	NA	NA
10,000,000	8.076241524S	NA	NA

Table 2.1: Time complexity Analysis Data of Merge Sorting Algorithm

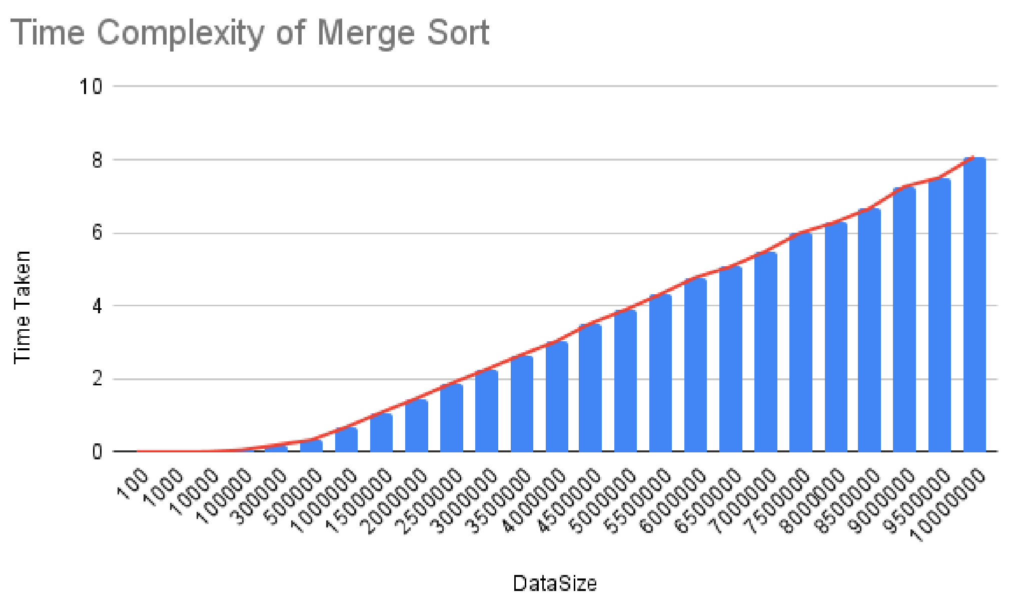

Figure 2.1: Best Case Of Merge Sorting Algorithm

2.2.3 Analysis Result

The analysis of the gathered data and corresponding graphs highlights the performance of the Merge Sort algorithm across different scenarios. Merge Sort, a divide-and-conquer sorting algorithm, demonstrates consistent behavior due to its deterministic approach to sorting.

In the best-case scenario, where the dataset is already sorted, the algorithm still recursively divides and merges the dataset, as it does not optimize for pre-sorted input. Consequently, the execution time reflects a time complexity of O(nlogn), as indicated by the graph. This consistent performance stems from the fixed overhead of dividing the dataset and merging subarrays.

For the average-case scenario, the graph similarly illustrates a time complexity of O(nlogn). The algorithm divides the dataset into halves recursively, sorts each half, and merges them back together. The logarithmic depth of recursive calls combined with the linear merging step ensures predictable and efficient sorting, even for randomly ordered data.

In the worst-case scenario, such as when the dataset is sorted in reverse order, the performance remains consistent with a time complexity of O(nlogn). The graph confirms this behavior, showing no significant deviation in execution time compared to the average case. This robustness is attributed to Merge Sort's structured and uniform approach, which treats all input scenarios equivalently in terms of computational effort.

Overall, the analysis confirms that Merge Sort is a highly efficient and stable sorting algorithm. Its O(nlogn) time complexity across all scenarios makes it well-suited for handling large datasets, especially when stability and worst-case guarantees are critical. Compared to simpler algorithms like Bubble Sort or Insertion Sort, Merge Sort's divide-

and-conquer methodology provides a significant performance advantage for larger or more complex datasets.