Introduction :

On the twelfth day of my internship at Surfboard Payments, I focused on learning sorting algorithms in JavaScript. I studied five different sorting algorithms and analyzed their performance based on the time taken to sort different input sizes. The goal was to determine which algorithm performed the best and which two performed the worst. I also examined their time complexity and efficiency under various conditions.

Understanding Sorting Algorithms

Sorting algorithms are used to arrange elements in a specific order, such as ascending or descending. Different sorting algorithms have different efficiencies depending on the size and nature of the input data. The five sorting algorithms I analyzed were Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort.

Process Followed

1 I implemented each sorting algorithm in JavaScript.

2 I tested each algorithm with different input sizes to measure performance.

3 I analyzed the time complexity of each algorithm.

4 I identified the best performing and the two worst performing algorithms based on execution time and efficiency.

5 I documented observations regarding their performance.

Performance Analysis

Best Performing Algorithm

Quick Sort was the best performing algorithm among the five.

1 Time Complexity Quick Sort has an average time complexity of O n log n, which makes it highly efficient for large datasets.

2 Efficiency Quick Sort efficiently divides the array into smaller subarrays using a pivot element, reducing the number of comparisons.

3 Space Complexity Quick Sort has a space complexity of O log n due to its recursive nature, making it efficient in terms of memory usage compared to Merge Sort.

4 Observations Quick Sort performed significantly better than the other sorting algorithms, especially on large datasets. It quickly reduced the number of comparisons by partitioning the data and sorting smaller subsets efficiently.

Worst Performing Algorithms

Bubble Sort and Selection Sort were the two worst performing algorithms.

Bubble Sort

1 Time Complexity Bubble Sort has a worst-case time complexity of O n squared. This makes it inefficient for large datasets.

2 Efficiency Bubble Sort repeatedly compares adjacent elements and swaps them if they are out of order, leading to a high number of operations.

3 Space Complexity It has a space complexity of O 1, meaning it does not require extra memory. However, its inefficiency in execution outweighs this advantage.

4 Observations Bubble Sort took the longest time to sort large datasets. Its inefficiency became evident when the input size increased, as it required multiple passes through the array.

Selection Sort

1 Time Complexity Selection Sort also has a worst-case time complexity of O n squared, making it slow for large inputs.

2 Efficiency The algorithm selects the smallest element from the unsorted portion and swaps it with the current position. This results in a high number of swaps and comparisons.

3 Space Complexity It has a space complexity of O 1, meaning it does not require additional memory. However, its performance is still poor.

4 Observations Although Selection Sort performed slightly better than Bubble Sort, it was still inefficient for large datasets. The number of comparisons and swaps increased significantly as the input size grew.

Time Complexity and Efficiency Analysis

1 Bubble Sort O n squared worst-case and average-case complexity, making it inefficient for large datasets.

2 Selection Sort O n squared worst-case and average-case complexity, leading to slow performance.

3 Insertion Sort O n squared worst-case, but O n for nearly sorted data, making it more efficient than Bubble and Selection Sort for certain cases.

4 Merge Sort O n log n time complexity in all cases, making it efficient but requiring additional memory O n for merging.

5 Quick Sort O n log n average and best-case complexity, making it the fastest among these sorting algorithms. However, its worst-case complexity is O n squared if the pivot selection is poor.

Key Observations

1 Quick Sort consistently outperformed other sorting algorithms due to its efficient divide-and-conquer approach.

2 Merge Sort was also efficient but required additional memory for merging, which could be a disadvantage in memory-constrained environments.

3 Insertion Sort performed well for small and nearly sorted datasets but was inefficient for large random datasets.

4 Bubble Sort and Selection Sort were the slowest due to their high number of comparisons and swaps.

Conclusion

On day 12 of my internship at Surfboard Payments, I successfully learned about sorting algorithms in JavaScript and analyzed their performance. I identified Quick Sort as the best-performing algorithm due to its efficient time complexity and partitioning strategy. Bubble Sort and Selection Sort were the worst-performing algorithms due to their O n squared time complexity, making them inefficient for large datasets. This analysis helped me understand the importance of choosing the right sorting algorithm based on the input size and nature of the data.