QUICKSORT ALGORITHM ANALYSIS

Assignment 5

JULY 25, 2025

BISHNU SHARMA
ID: 005036007

Quicksort Algorithm: Implementation, Analysis, and Randomization

Introduction

This report focuses on building and evaluating two versions of the Quicksort algorithm, a popular and efficient sorting method based on the divide and conquer strategy. In this method, a pivot element is selected, and the input array is divided into parts based on comparisons with the pivot. Each part is then sorted recursively. The goal of this assignment was to implement both deterministic and randomized versions of Quicksort, evaluate their performance on different types of inputs, and analyze how randomization can help avoid poor performance in worst case scenarios.

Implementation

Two versions of the Quicksort algorithm were implemented in Python. The deterministic version always selects the first element of the array as the pivot. While this method is easy to implement, it often performs poorly when the input is already sorted or close to being sorted. The randomized version, on the other hand, selects a pivot randomly from the array at each recursive step. This helps avoid consistently poor partitions and improves average performance (GeeksforGeeks, 2024).

Both versions follow the same logic. The input array is divided into three sections: elements smaller than the pivot, elements equal to it, and elements that are larger. Each part is then sorted recursively, and the result is the combination of the sorted subarrays. The time module from Python's standard library was used to measure execution time of each sorting function (Python Software Foundation, n.d.).

A third Python script was developed to test and compare the two implementations using various input types. The test cases covered different types of input, such as randomly ordered data, already sorted lists, reverse ordered lists, and lists with duplicate values.

Theoretical Analysis

The performance of Quicksort depends heavily on how the pivot divides the array. The most efficient case occurs when the pivot splits the array into two evenly sized sections, resulting in a time complexity of O(n log n). On average, especially with randomly arranged data, the algorithm also runs in O(n log n) time. However, the worst case arises when the pivot causes highly unbalanced partitions, for example, if the input is sorted and the pivot is always chosen as the first element. In this situation, the time complexity becomes $O(n^2)$ (Cormen et al., 2009).

In terms of space complexity, Quicksort uses O(log n) space on average due to the recursive calls. However, in the worst case, it can use up to O(n) space if the recursion goes too deep.

Empirical Results

Both deterministic and randomized versions of Quicksort were tested on arrays of size 10000. The execution times for different input types were recorded in milliseconds. The results are summarized below:

```
Performance Comparison (Sorting - in milliseconds):

Random Array:
Deterministic Quicksort: 18.52
Randomized Quicksort   : 22.87

Sorted Array:
Deterministic Quicksort: 3771.28
Randomized Quicksort   : 24.42

Reverse Sorted Array:
Deterministic Quicksort: 3853.66
Randomized Quicksort   : 19.81

Repeated Elements Array:
Deterministic Quicksort: 0.75
Randomized Quicksort   : 5168.64
```

The deterministic version worked well with random input but performed poorly on sorted and reverse sorted data due to its predictable pivot choice. In contrast, the randomized version maintained stable performance across most input types. The only exception was the repeated elements array, where randomized Quicksort was slower because it failed to make progress in partitioning the array efficiently.

Impact of Randomization

Randomizing the pivot is an effective way to avoid worst case behavior in Quicksort. According to GeeksforGeeks (2024), a random pivot reduces the likelihood of always selecting a bad pivot that leads to unbalanced partitions. This helps keep the performance close to O(n log n) in most cases. However, randomization is not a guaranteed solution. In cases where the input has many repeated elements, even randomized pivot selection may not lead to efficient partitions.

Conclusion

This assignment provided a hands-on understanding of how Quicksort behaves under different conditions. It showed how important the choice of pivot is to the performance of the algorithm. While the deterministic version is easier to implement, it is not suitable for all types of input data. The randomized version performs more consistently and avoids poor performance in many scenarios. This experience has helped me learn how to analyze algorithm behavior and understand when and why certain strategies, such as randomization, are useful in real world programming.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Python Software Foundation. (n.d.). *time — Measure execution time*. Python 3.12. Retrieved from https://docs.python.org/3/library/time.html

GeeksforGeeks. (2024). *QuickSort - Data Structure and Algorithm Tutorials*. Retrieved from https://www.geeksforgeeks.org/quick-sort/