Performance Analysis of Quicksort Algorithm: Deterministic vs Randomized

Student's Name: Pushya Mithra Kotakonda

Course: Algorithms and Data structures

StudentID: 005034297

Performance Analysis of Quicksort Algorithm: Deterministic vs Randomized

1. Introduction

The Quicksort algorithm is one of the most widely used sorting algorithms in computer science, known for its efficiency in sorting large datasets. It is a comparison-based, divide-and-conquer algorithm that partitions an array around a pivot element and recursively sorts the two subarrays. This report focuses on implementing the deterministic and randomized versions of the Quicksort algorithm, analyzing their performance, and examining the impact of randomization on Quicksort's behavior. By running tests on different input sizes and distributions, we explore how these versions perform in real-world scenarios and their theoretical implications.

2. Quicksort Implementation

Deterministic Quicksort:

The deterministic version of Quicksort works by selecting a fixed pivot, usually the last element of the array. It then partitions the array into two subarrays: one with elements smaller than the pivot and the other with elements greater than the pivot. These subarrays are then recursively sorted (Kneusel, 2024). The partitioning step rearranges elements so that the pivot is in its correct position, and both subarrays are sorted independently.

Algorithm Steps:

1. Select a pivot (last element in this case).
2. Partition the array so that elements less than the pivot are on its left, and elements greater than the pivot are on its right.
3. Recursively sort the left and right subarrays.

Randomized Quicksort:

In the randomized version, the pivot is chosen randomly from the array before partitioning. This helps reduce the likelihood of encountering the worst-case time complexity (O(n^2)), which occurs when the pivot selection consistently results in unbalanced partitions (e.g., if the pivot is always the smallest or largest element) (Vaidya, 2021). By randomizing the pivot, the algorithm becomes less predictable, leading to improved performance in many cases.

Algorithm Steps:

4. Randomly choose a pivot element.
5. Swap it with the last element in the array.
6. Partition the array around this new pivot.
7. Recursively sort the left and right subarrays.

3. Performance Analysis

To evaluate the performance of both the deterministic and randomized versions of Quicksort, we analyze their time complexity in different cases:

- **Best Case**: The best-case time complexity occurs when the pivot splits the array evenly, resulting in O(n log n) comparisons.
- **Average Case**: In most cases, Quicksort behaves similarly to the best case, with time complexity of O(n log n). This happens when the pivot divides the array in such a way that both subarrays are of similar size.
- **Worst Case**: The worst-case time complexity occurs when the pivot is consistently the smallest or largest element, causing the array to be partitioned in an unbalanced

way. This results in O(n^2) time complexity. In the deterministic version, this can happen when the input array is sorted or nearly sorted. In the randomized version, the likelihood of hitting this worst-case scenario is significantly reduced.

Space Complexity:

Both versions of Quicksort have a space complexity of O(log n) due to the recursive nature of the algorithm, which requires space for the call stack (Kneusel, 2024). However, in the worst case, where the recursion depth becomes O(n) (e.g., when the pivot is consistently unbalanced), the space complexity may degrade to O(n).

4. Empirical Analysis The empirical results from running both the deterministic and randomized Quicksort algorithms show the following:

- **Deterministic Quicksort**: The time taken to sort a random array of 1000 elements was 0.0039985 seconds. This is typical for Quicksort in the average case, where the pivot selections generally lead to balanced partitions.
- **Randomized Quicksort**: The time taken to sort the same array was 0.0020027 seconds. The randomized version was faster in this particular case, as it reduces the chances of hitting the worst-case scenario, which would occur more frequently in the deterministic version with highly ordered inputs.

Impact of Randomization:

Randomization in Quicksort helps avoid the worst-case performance (O(n^2)), which is more likely in deterministic Quicksort when the input array is already sorted or nearly sorted. Randomized Quicksort tends to provide more consistent performance across various input distributions, including sorted, reverse-sorted, and random arrays. By randomly

selecting the pivot, it ensures that the partitioning process remains relatively balanced, which is crucial for maintaining optimal performance (O(n log n)).

5. Conclusion

In this report, we implemented both deterministic and randomized versions of the Quicksort algorithm and analyzed their performance based on theoretical time complexities and empirical data. The deterministic Quicksort performs well in most average cases but can degrade to quadratic time complexity in the worst case. On the other hand, the randomized version mitigates the chances of encountering the worst-case scenario, providing more consistent and often faster results, particularly when the input data is already sorted or nearly sorted. As shown by our empirical tests, randomization significantly improves the efficiency of Quicksort in real-world applications, making it a more robust algorithm for general use.

References

Kneusel, R. T. (2024). *The Art of Randomness: Randomized Algorithms in the Real World*. No Starch Press.

Vaidya, K. E. (2021). *The case for a learned sorting algorithm* (Doctoral dissertation, Massachusetts Institute of Technology).