Report: Quicksort: Randomized and Deterministic

Avinna Bhattarai

005028547

MSCS-532-A01

Assignment 5

Dr. Vanessa Ruth Cooper

09/23/2024

**Introduction**

This report explores the implementation and analysis of the Quicksort algorithm, a widely-used sorting method known for its efficiency and speed. Quicksort is a comparison-based sorting algorithm that employs a divide-and-conquer strategy. It is particularly effective for large datasets and is often the algorithm of choice in many software applications due to its average-case performance of .

**Quicksort Implementation**

**Design Choices**  
We implement the Quicksort algorithm using a recursive approach. The choice of pivot selection significantly impacts performance; in the deterministic version, the middle element is used, while the randomized version selects a pivot randomly. This randomization helps mitigate the risk of encountering worst-case scenarios.

**Implementation Details**  
The Quicksort algorithm is divided into the following steps:

* **Select Pivot**: Choose a pivot element from the array.
* **Partitioning**: Rearrange the array so that elements less than the pivot come before it and elements greater come after it.
* **Recursion**: Recursively apply the above steps to the subarrays formed by the partitioning.

The implementations can be found in the repository named *randomized\_quicksort.py* and *deterministic\_quicksort.py*

**Time Complexity**

* **Best Case**: occurs when the pivot divides the array evenly.
* **Average Case**: is expected with random or unsorted input due to balanced partitions.
* **Worst Case**: arises when the pivot is poorly chosen (e.g., smallest or largest element).

**Space Complexity**

The space complexity for both implementations is for the recursion stack. The temporary arrays used in partitioning lead to an additional space complexity of when utilizing lists.

**Empirical Analysis**

To compare the performance of the deterministic and randomized versions of Quicksort, we tested each implementation with three types of input distributions:

* **Random**: An unsorted array of random integers.
* **Sorted**: An array already sorted in ascending order.
* **Reverse-Sorted**: An array sorted in descending order.

We used array sizes of n=[1000,5000,10000,50000,100000]n = [1000, 5000, 10000, 50000, 100000]n=[1000,5000,10000,50000,100000] and recorded the execution times for each version.

**Results of Sorting Algorithms**

**Table 1**

Array Size Algorithm Random Array Sorted Array Reverse-Sorted Array

1000 Deterministic 0.003174s 0.002103s 0.002088s

1000 Randomized 0.003890s 0.004060s 0.004084s

5000 Deterministic 0.017367s 0.012934s 0.011528s

5000 Randomized 0.013125s 0.011748s 0.010722s

10000 Deterministic 0.022065s 0.013498s 0.012479s

10000 Randomized 0.018167s 0.019514s 0.020040s

50000 Deterministic 0.081051s 0.084562s 0.089438s

50000 Randomized 0.105675s 0.637224s 0.126257s

100000 Deterministic 0.434352s 0.176478s 0.312132s

100000 Randomized 0.229785s 0.287702s 0.258193s

**Analysis of Quicksort Performance**

* **Deterministic Quicksort:** This version performed adequately for random and sorted arrays but showed slower performance on reverse-sorted arrays due to poor pivot selection. The execution times reflected the worst-case scenario when the pivot was consistently poor.
* **Randomized Quicksort:** This version outperformed the deterministic version across all input types, particularly on reverse-sorted arrays. The randomization of the pivot selection mitigated the worst-case scenario, resulting in more balanced partitions and consistent performance.

**Conclusion**

This empirical analysis highlights the strengths of both deterministic and randomized versions of the Quicksort algorithm. While deterministic Quicksort is simpler, randomized Quicksort offers better performance in diverse scenarios, particularly when faced with unfavorable input distributions. Understanding these differences aids in selecting the most appropriate sorting algorithm for various applications.