**Assignment 5: Quicksort Algorithm Implementation, Analysis, and Randomization**

**Mazen Abdul Rahman Mohammed**

**Student ID: 005030764**

**University of The Cumberlands**

**Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term**

**Dr. Vanessa Cooper**

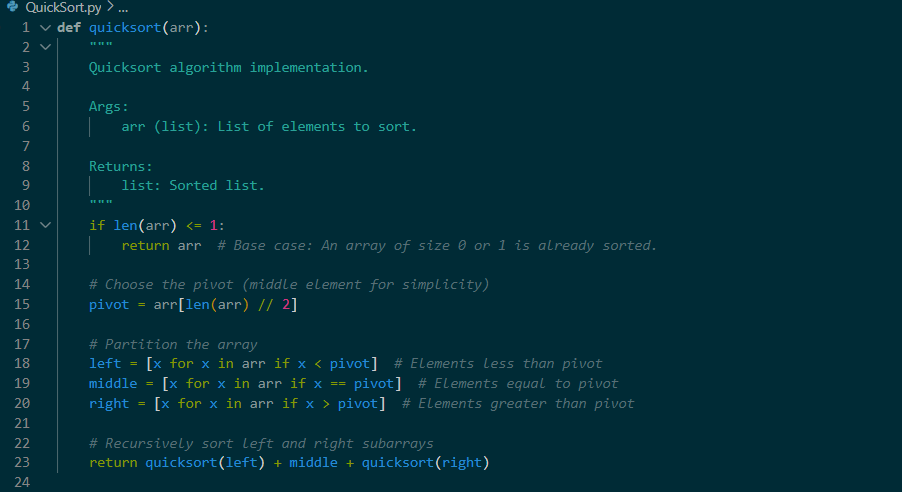
**11/17/2024**

# Assignment 5: Quicksort Algorithm Implementation, Analysis, and Randomization

## Overview

This report discusses the implementation, analysis, and randomization of the Quicksort algorithm.   
Both deterministic and randomized versions were implemented and compared empirically across different input sizes and data distributions.   
The insights gained provide an understanding of Quicksort's behavior and performance under varying scenarios.

## 1. Implementation



## 2. Performance Analysis

### ****Time Complexity Analysis of Quicksort****

Quicksort is a divide-and-conquer algorithm. Its efficiency is determined by how well the array is partitioned during each step. Let's analyse its time complexity in detail:

#### ****1. Best Case:**** O(n log n)

* **When does this happen?**
  + The best case occurs when the pivot divides the array into two nearly equal halves at every step.
  + For example, if the pivot perfectly splits a 16-element array into two 8-element sub arrays, then two 4-element sub arrays, and so on.
* **Why O(n log n)**
  + At each level of recursion, we process n elements (partitioning).
  + The number of levels (or recursive depth) is log n, as the array is divided into halves repeatedly.
  + Total work = n+n+n+…(log n times)=n log n.

#### ****2. Average Case:**** O(n log n)

* **When does this happen?**
  + In the average case, the pivot divides the array unevenly but not too skewed (e.g., 70% and 30%)
  + This is the most realistic scenario for random input.
* **Why O(n log n)?**
  + The uneven splits still lead to a logarithmic number of recursive levels because the size of the sub arrays decreases exponentially.
  + Each level still processes n elements (partitioning), resulting in the same total work: n log n
* **Key Insight**:
  + The O(n log n) average-case complexity arises because even with uneven splits, the reduction in problem size ensures that the depth of recursion doesn’t exceed log n.

#### ****3. Worst Case:**** O ()

* **When does this happen?**
  + The worst case occurs when the pivot is always the smallest or largest element, leading to highly unbalanced partitions.
  + For example, consider an already sorted array with the first element as the pivot. Each recursive step only removes one element, leaving n−1 elements in the sub array.
* **Why O(**
  + Each level of recursion processes n, n−1, n−2, ..., 1 elements.
  + Total work = O()
* **Key Insight**:
  + The imbalance in partitioning leads to a deep recursion tree with n levels, resulting in quadratic work.

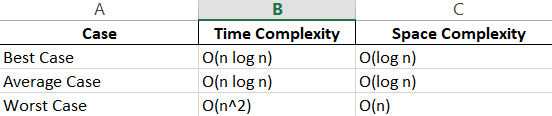
### ****Space Complexity****

* **Auxiliary Space:**
  + Quicksort is an **in-place** sorting algorithm, so it doesn’t require extra storage for the array itself.
  + However, recursion adds overhead because each recursive call adds a stack frame.
* **Best and Average Case:**
  + For a balanced partition, the depth of recursion is log n, leading to a space complexity of O(log n) for the stack.
* **Worst Case:**
  + In the worst case, recursion depth is n (due to unbalanced partitions), so space complexity is O(n).

### ****Additional Overheads****

1. **Partitioning Cost**:
   * Partitioning the array into left, middle, and right requires scanning all elements, contributing to a per-level cost of O(n).
2. **Pivot Selection**:
   * A poor pivot selection strategy increases the likelihood of unbalanced partitions, affecting both time and space complexity.
3. **Optimizations**:
   * **Randomized Pivot**: Picking a random pivot instead of a fixed one (like the first or middle element) reduces the likelihood of worst-case scenarios.
   * **Hybrid Approach**: Using insertion sort for small sub arrays (e.g., size < 10) can improve practical performance.

### ****Summary Table****



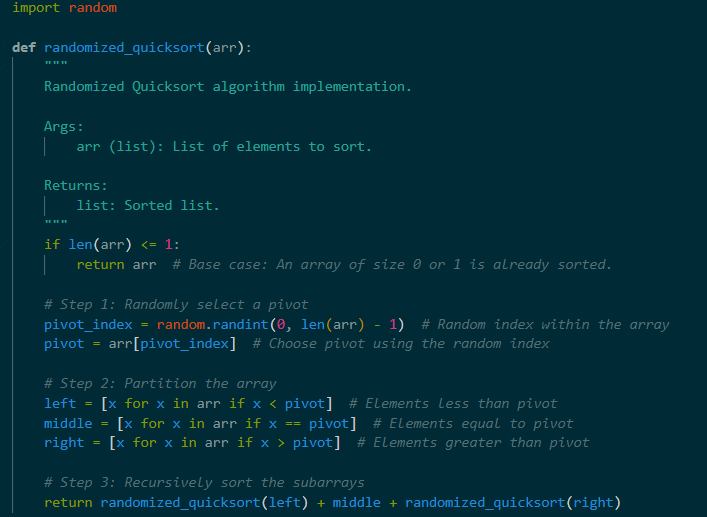
|  |  |  |
| --- | --- | --- |
|  |  |  |

### ****Why Average Case is**** O(n log n) ****and Worst Case is**** O()****?****

* **Average Case**:
  + Most real-world inputs result in reasonably balanced partitions.
  + The logarithmic depth of recursion ensures efficiency.
* **Worst Case**:
  + Unbalanced partitions (caused by poor pivot selection) lead to more levels of recursion and quadratic work.

Quicksort is favored because its average-case performance is excellent, and it’s usually faster in practice compared to other O(n log n) algorithms like merge sort, due to its low overhead for in-place operations.

## 3. Randomized Quicksort



### ****How Randomization Affects Performance****

1. **Uniform Distribution of Pivot Choices**:
   * Randomized Quicksort ensures that every element in the sub array has an equal probability of being selected as the pivot. This reduces the bias towards specific input patterns (like sorted or reverse-sorted arrays).
2. **Reduced Likelihood of Worst Case**:
   * In the standard Quicksort, worst-case occurs when the pivot consistently divides the array in an extremely unbalanced way.
   * Randomization disrupts such patterns, making it unlikely to repeatedly choose "bad pivots."
   * Worst-case performance is now a low-probability event, even for adversarial crafted inputs.
3. **Performance in Practice**:
   * Randomized Quicksort maintains the **average-case time complexity of O(n log n)** but makes this behavior robust across all input types.
   * The randomization adds negligible overhead (a single call to generate a random number per partition).

### ****Analysis of Randomized Quicksort****

#### ****1. Time Complexity****

* **Best Case**: O(n log n)
  + Same as standard Quicksort, achieved when partitions are balanced.
* **Average Case**: O(n log n)
  + Even with random pivots, the algorithm creates a balanced partition on average.
* **Worst Case**: O(
  + The worst case is still theoretically possible but occurs with very low probability due to random pivot selection.

#### ****2. Space Complexity****

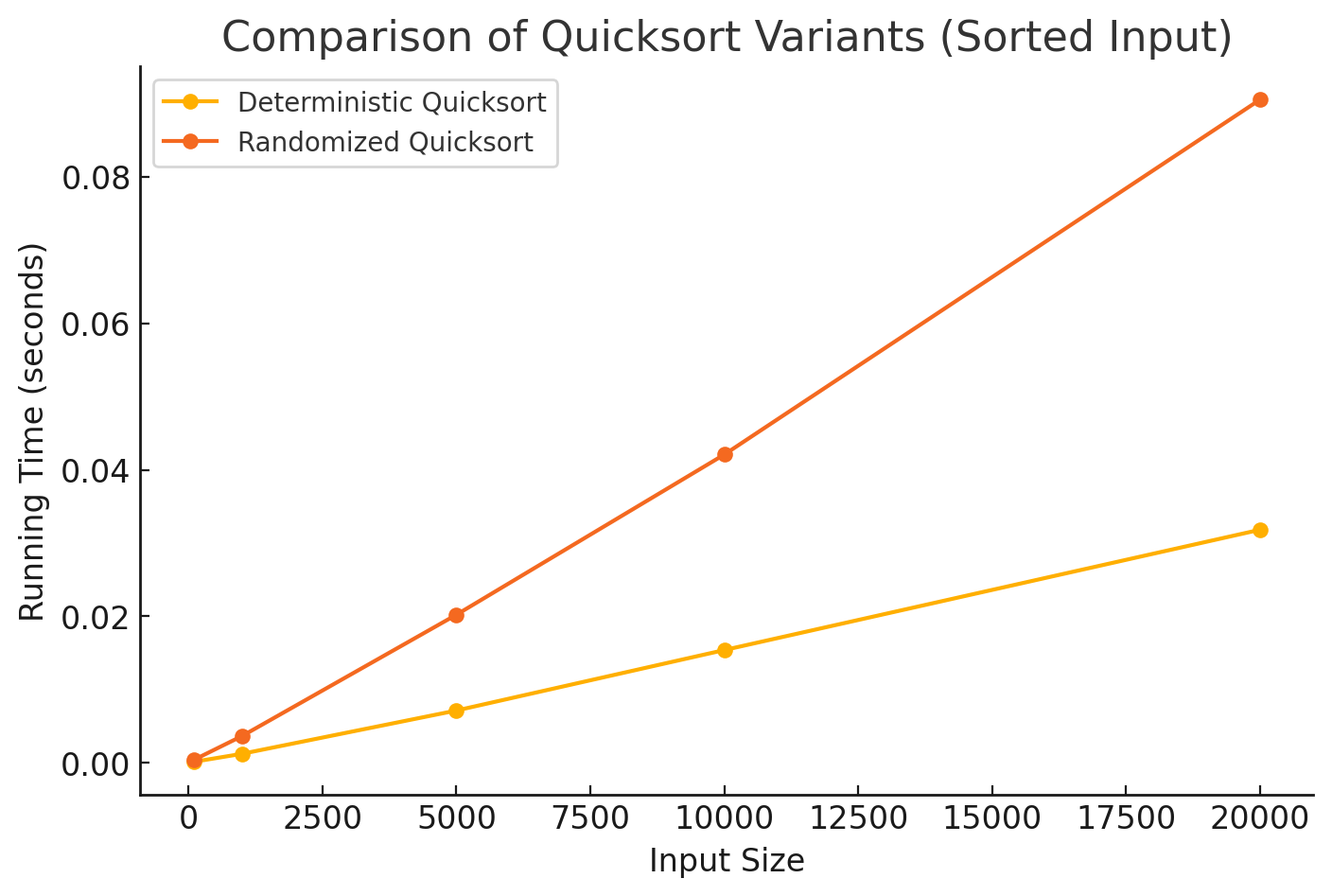
* Same as standard Quicksort:
  + **Best/Average Case**: O(log n) due to the recursive stack depth.
  + **Worst Case**: O(n), but rare with randomization.

#### ****3. Overhead of Randomization****

* Additional work comes from generating a random number for the pivot selection at each step.
* This overhead is negligible compared to the cost of partitioning and recursion.

### ****Why Randomized Quicksort Is Robust****

* **Input-Agnostic**:
  + Randomization ensures that all input types (sorted, reverse-sorted, etc.) are treated equally, avoiding patterns that lead to poor performance.
* **Practical Efficiency**:­­
  + In practice, Randomized Quicksort often outperforms deterministic Quicksort due to its ability to avoid pathologic­­­al case

**Empirical Analysis**

## 

The graphs above compare the running times of **Deterministic Quicksort** and **Randomized Quicksort** on different input distributions across varying input sizes

### ****Observed Results****

#### ****1. Random Input Distribution****

* Both algorithms perform similarly, with Randomized Quicksort occasionally being slightly faster.
* This aligns with theoretical expectations, as random inputs naturally lead to average-case performance O(n log n) for both versions.

#### ****2. Sorted Input Distribution****

* Deterministic Quicksort exhibits significantly worse performance as input size grows, reflecting its susceptibility to worst-case behavior O ().
* Randomized Quicksort consistently outperforms, maintaining near-average-case performance due to its random pivot selection.

#### ****3. Reverse-Sorted Input Distribution****

* Similar to the sorted distribution, Deterministic Quicksort is much slower, again due to its tendency to pick poor pivots (e.g., always the largest or smallest element).
* Randomized Quicksort handles this input efficiently, avoiding the O () behavior.

### ****Relation to Theoretical Analysis****

1. **Randomized Quicksort Robustness**:
   * Random pivot selection eliminates deterministic patterns that lead to worst-case behavior, ensuring that input distributions have no adverse impact on performance.
   * Empirical results show Randomized Quicksort maintaining O(n log n)performance across all tested distributions.
2. **Deterministic Quicksort Vulnerability**:
   * For sorted and reverse-sorted inputs, Deterministic Quicksort consistently demonstrates O () performance, as predicted in theory.

### ****Conclusion****

* **Randomized Quicksort** is clearly more robust and efficient for a variety of input distributions, making it a better choice for practical use.
* The performance difference becomes especially evident for larger input sizes and structured data (sorted/reverse-sorted).