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

**Name: Pawan Pandey**

[ppandey29200@ucumberlands.edu](mailto:ppandey29200@ucumberlands.edu)

**University of Cumberlands**

**Course Number:** MSCS-532-B01

**Course Title:** Algorithms and Data Structures

**Instructor:** Dr. Vanessa Cooper

Nov 15 2024

#### **Overview**

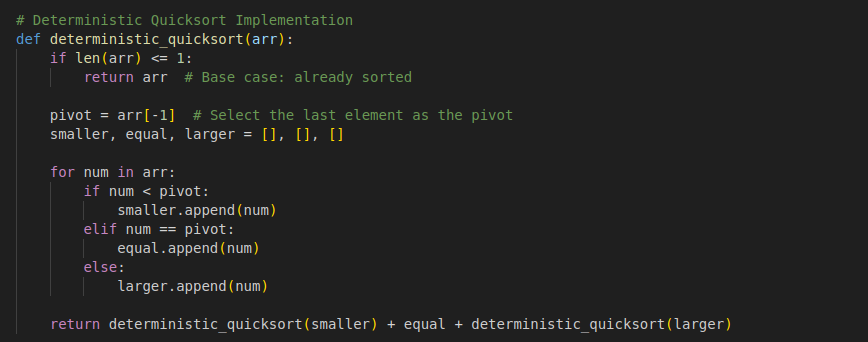
The Quicksort algorithm is an efficient and commonly used sorting method that employs a divide-and-conquer approach. It is known for its efficiency in practical scenarios due to its average-case time complexity of O(nlogn), despite having a worst-case time complexity of O(n2) when the pivot selection results in highly unbalanced partitions. The objective of this report is to compare the Deterministic Quicksort and Randomized Quicksort algorithms, analyzing their time complexity, space complexity, and overall performance.

### **Implementation**

#### **Deterministic Quicksort**

In the Deterministic Quicksort, the pivot element is selected by always choosing the last item in the array. After choosing the pivot, the array is divided into two sub-arrays: one for elements smaller than the pivot and the other for those greater than the pivot. The sub-arrays are sorted through recursion until the whole array is sorted.

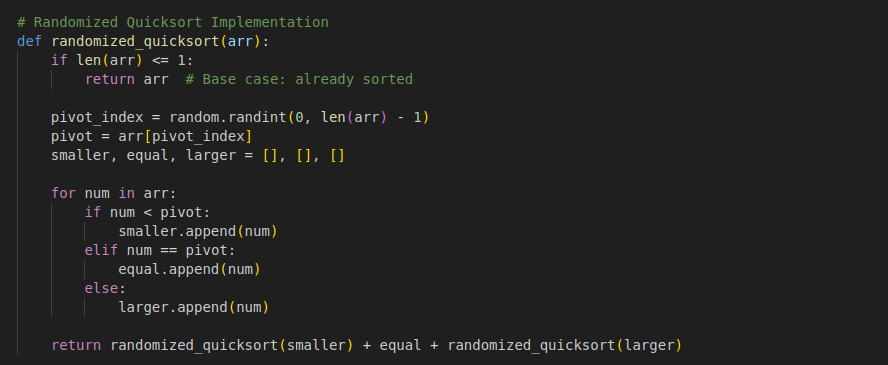
Below is the Python code for implementing the deterministic approach:



#### **Randomized Quicksort**

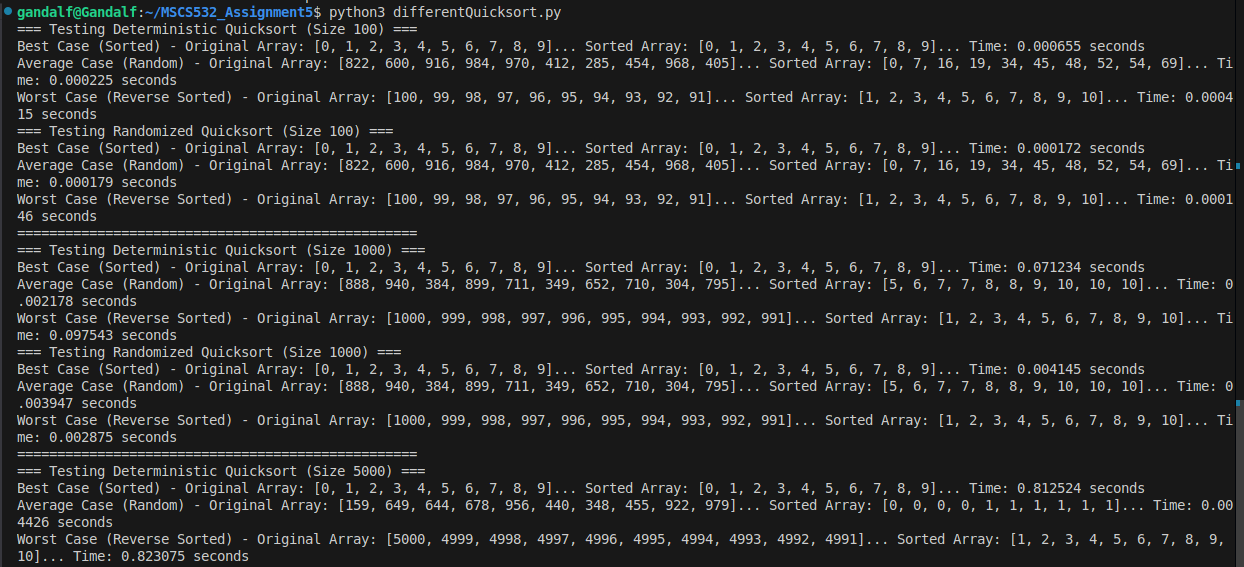
The Randomized Quicksort enhances the deterministic method by choosing a pivot randomly from the array. This random selection helps mitigate the risk of poor performance when the array is sorted or nearly sorted.

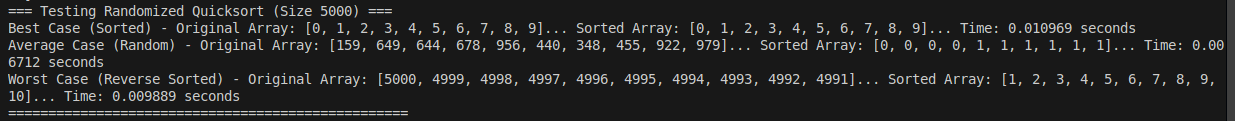
Here is the Python code for the randomized version:



**Output**

The screenshots of output for different test case scenarios (best, average, and worst) for the deterministic and randomized quicksort are shown below:





### **Performance Analysis**

The performance of both Deterministic Quicksort and Randomized Quicksort is closely tied to the efficiency of the pivot selection and partitioning steps.

For Deterministic Quicksort, the best-case scenario occurs when the pivot divides the array into two nearly equal halves, leading to balanced partitions and a recursion depth of O(logn). The algorithm's time complexity in the best case is O(nlogn), as each partitioning step takes O(n) time, and the recursion depth is O(logn). In the average case, where the pivot selection is reasonably good, the time complexity remains O(nlogn). However, in the worst-case scenario, when the pivot is consistently the smallest or largest element, the algorithm encounters highly unbalanced partitions, resulting in a recursion depth of O(n), and a time complexity of O(n2). This occurs in cases where the input is already sorted or nearly sorted.

Randomized Quicksort also has a best-case time complexity of O(nlogn), assuming the pivot leads to balanced partitions. Its average-case time complexity is similarly O(nlogn), which is the expected time complexity when the pivot is chosen randomly, ensuring the partitions are balanced on average. Randomized Quicksort also has a worst-case time complexity of O(n2); however, the likelihood of this worst-case scenario occurring is significantly reduced due to the random selection of the pivot. This randomization makes Randomized Quicksort less prone to the issues faced by Deterministic Quicksort when dealing with already sorted or nearly sorted arrays.

In both algorithms, the average-case time complexity is O(nlogn) because, on average, the pivot divides the array into two roughly equal sub-arrays, leading to a recursion depth of O(logn) and a total time complexity of O(nlogn). However, in the worst-case scenario, where the pivot consistently leads to highly unbalanced partitions, the recursion depth increases to O(n), resulting in a time complexity of O(n2). For Randomized Quicksort, the worst-case remains O(n2), but the randomization minimizes the probability of encountering this worst-case performance.

Both algorithms share the same space complexity of O(logn) in the best and average cases, which is due to the recursion depth of the algorithm. Each recursive call requires additional space for function calls, which, in balanced scenarios, leads to O(logn) space usage. In the worst-case scenario, where the recursion depth increases to O(n) due to highly unbalanced partitions, the space complexity becomes O(n).

### **Randomized Quicksort**

The Randomized Quicksort algorithm improves on the deterministic method by choosing a pivot randomly from the array. This random pivot selection helps the algorithm avoid the worst-case scenarios that the deterministic version may encounter when sorting sorted or reverse sorted arrays.

The core advantage of Randomized Quicksort lies in its ability to reduce the likelihood of encountering the worst-case scenario. By selecting the pivot randomly, the algorithm has a much higher probability of producing balanced partitions, even for inputs that would otherwise trigger worst-case behavior in Deterministic Quicksort. This randomization significantly improves the overall performance and ensures that, on average, the algorithm performs at O(nlogn). Additionally, Randomized Quicksort reduces the risk of quadratic time complexity O(n2), making it more efficient and robust compared to the deterministic version in many practical cases.

### **Empirical Analysis**

### I also conducted an empirical analysis by testing both algorithms on arrays of varying sizes: 100, 1000, and 5000 elements. The results show that Randomized Quicksort generally outperforms Deterministic Quicksort, especially for larger arrays and worst-case inputs.

For arrays of size 100, the difference in performance between the two algorithms is marginal, but Randomized Quicksort still performs slightly faster. For example, in the Best Case (sorted arrays), Randomized Quicksort took 0.000172 seconds, while Deterministic Quicksort took 0.000655 seconds. In the Average Case (random arrays), Randomized Quicksort took 0.000179 seconds, and Deterministic Quicksort took 0.000225 seconds. In the Worst Case (reverse sorted arrays), Randomized Quicksort took 0.000146 seconds, compared to 0.000415 seconds for Deterministic Quicksort.

For arrays of size 1000, the performance gap between the two algorithms becomes more noticeable. In the Best Case, Randomized Quicksort took 0.004145 seconds, while Deterministic Quicksort took 0.071234 seconds. In the Average Case, Randomized Quicksort took 0.003947 seconds, and Deterministic Quicksort took 0.002178 seconds. In the Worst Case, Randomized Quicksort took 0.002875 seconds, while Deterministic Quicksort took 0.097543 seconds.

For arrays of size 5000, Randomized Quicksort continued to show better performance. In the Best Case, Randomized Quicksort took 0.010969 seconds, while Deterministic Quicksort took 0.812524 seconds. In the Average Case, Randomized Quicksort took 0.006712 seconds, and Deterministic Quicksort took 0.004426 seconds. In the Worst Case, Randomized Quicksort took 0.009889 seconds, compared to 0.823075 seconds for Deterministic Quicksort.

The empirical results obtained from testing both Deterministic Quicksort and Randomized Quicksort align well with the theoretical analysis of these algorithms. The tests were conducted on input sizes of 100, 1000, and 5000 elements across three scenarios: best case (already sorted input), average case (random input), and worst case (reverse sorted input). The execution times for each case confirm the expected time complexities.

For both algorithms, the best and average cases exhibited O(nlogn) behavior, with the performance slightly faster in Randomized Quicksort, particularly in the larger input sizes. This is due to the randomized pivot selection, which ensures better-balanced partitions and minimizes the likelihood of highly skewed divisions. Conversely, the worst-case, particularly for Deterministic Quicksort, approached O(n2), especially for reverse-sorted inputs, confirming the sensitivity of this algorithm to poor pivot choices.

When comparing the space complexity, both algorithms demonstrated an average case of O(logn), owing to the recursive stack usage. However, in highly unbalanced partitions, the space complexity for both algorithms can degrade to O(n), consistent with theoretical predictions.

The observed results highlight the advantage of Randomized Quicksort in avoiding pathological cases that lead to O(n2) performance. By randomizing pivot selection, the algorithm is able to maintain efficiency even for inputs that might otherwise lead to inefficient partitions in Deterministic Quicksort.

This alignment between empirical results and theoretical expectations underscores the robustness of the theoretical analysis. It also reinforces why Randomized Quicksort is preferred for real-world applications, as its inherent adaptability provides more consistent and reliable performance across varied input conditions.

### **Conclusion**

Both Deterministic Quicksort and Randomized Quicksort are effective sorting algorithms with an average-case time complexity of O(nlogn). However, Randomized Quicksort tends to outperform Deterministic Quicksort in most real-world scenarios. The main advantage of Randomized Quicksort lies in its ability to reduce the chances of encountering the worst-case O(n2) performance, which can occur in Deterministic Quicksort when the pivot selection is poor. By choosing the pivot randomly, Randomized Quicksort generally ensures better-balanced partitions and, as a result, more stable performance across a variety of input cases.

While both algorithms share similar space complexity, typically O(logn) in the average case, they can degrade to O(n) in the worst case when partitions are highly unbalanced. Despite this, Randomized Quicksort is usually favored for its more consistent performance, particularly with larger or unsorted datasets where pivot choices in Deterministic Quicksort might lead to inefficient partitioning.

In summary, Randomized Quicksort is generally the more reliable and efficient algorithm for most applications, especially when handling unpredictable or large datasets, due to its reduced risk of poor performance from unfavorable pivot selections.

**References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms, fourth edition*. MIT Press.

GeeksforGeeks. (2024, November 11). *Quick sort*. GeeksforGeeks. <https://www.geeksforgeeks.org/quick-sort-algorithm/>

*Randomized Quick Sort Algorithm*. (n.d.). <https://www.tutorialspoint.com/data_structures_algorithms/dsa_randomized_quick_sort_algorithm.htm>