# Assignment 5 Report

# Introduction

Quicksort is an exceptionally efficient and often used sorting algorithm, characterized by its average-case time complexity of O(nlogn) and its capacity to sort arrays in situ, necessitating minimum supplementary memory. It works by a divide-and-conquer methodology, wherein an array is partitioned into subarrays according to a pivot element. The selection of the pivot substantially influences the algorithm's efficacy, especially for various input kinds, including random, sorted, or reverse-sorted data. In its deterministic form, Quicksort consistently chooses a certain element (often the final element) as the pivot. In its randomized variant, the pivot is selected randomly, which alleviates the worst-case performance often seen in deterministic Quicksort. Both iterations of the algorithm include advantages and disadvantages, and understanding their performance in various contexts is essential for enhancing sorting efficiency.

# Implementation

**Deterministic Quicksort**

In the deterministic variant of Quicksort, the partition function is tasked with segmenting the array according to a pivot, which is consistently selected as the last element in the array (or subarray) undergoing sorting. The variable pivot = arr[high] retains this value, while i = low - 1 monitors the location for items less than the pivot. The partition function traverses the array using a loop from low to high minus one. It verifies if each element is less than or equal to the pivot. If so, the index i is increased, and the element at position j is exchanged with the element at position i, therefore guaranteeing that items smaller than the pivot are positioned on the left side of the array.

A screenshot of a computer code

Description automatically generated

Upon concluding the loop, the pivot is positioned correctly by exchanging it with the element at index i + 1. At this juncture, all components to the left of the pivot are lesser, while all elements to the right are greater or equivalent. The partition function yields the index of the pivot (i + 1), indicating the division point of the array. The deterministic\_quicksort function use the partition index to iteratively sort the subarrays next to the pivot. Initially, it sorts the items before the pivot by using deterministic\_quicksort(arr, low, pi - 1) and then sorts the ones after the pivot with deterministic\_quicksort(arr, pi + 1, high). The recursion continues until the base case is reached, whereby low is no longer smaller than high, signifying that the subarray has one or zero entries and is thus already sorted. This method is effective for random data but may be inefficient for pre-sorted or reverse-sorted data, consistently resulting in a worst-case time complexity of O(n^2).

**Randomized Quicksort**

A close-up of a computer code

Description automatically generated

In the randomized variant of Quicksort, the function randomized\_partition is tasked with including randomization in pivot selection. Rather than consistently selecting the last element of the array as the pivot, a random element is chosen from the subarray using random\_index = random.randint(low, high). This function produces a random number inside the specified low and high indices, which identifies the element designated as the pivot. Upon picking the random pivot, it is exchanged with the final element in the array (i.e., arr[random\_index], arr[high] = arr[high], arr[random\_index]) to facilitate its use in the standard partitioning procedure. Upon completion of the swap, the partition function is invoked with the array, the low and high indices, and the pivot now positioned at the final index. The partition function operates similarly to the deterministic variant, segregating the array into items less than the pivot on the left and elements greater than or equal to the pivot on the right. The pivot is positioned accurately, and its index is provided. The randomized\_quicksort function thereafter recursively sorts the two subarrays generated by the partition function: the subarray to the left of the pivot and the subarray to the right of the pivot. This stochastic method averts the algorithm from repeatedly selecting suboptimal pivot items, such as the minimum or maximum member in the array, which may result in uneven partitions. This variant of Quicksort randomizes pivot selection, hence enhancing the probability of achieving balanced partitions and circumventing the O(n^2) worst-case situation faced by deterministic Quicksort when sorting pre-sorted or reverse-sorted datasets. Consequently, randomized Quicksort often attains a temporal complexity of O(nlogn) in such scenarios, making it more resilient across diverse input distributions.

# Performance Measures

A white background with text

Description automatically generated with medium confidence

A computer code with blue text

Description automatically generated

The screenshot presents a comparison of the performance of deterministic and randomized Quicksort across three distinct input types: a random array, a sorted array, and a reverse-sorted array. In deterministic Quicksort, the method operates effectively on a random array, executing in around 0.007 seconds. This aligns with its average-case time complexity of O(n log n), where the pivot, consistently selected as the final element, partitions the array into relatively balanced subarrays, facilitating rapid execution of the algorithm. Nonetheless, while sorting a pre-sorted array, deterministic Quicksort has a considerable increase in duration, around 0.159 seconds, attributable to the worst-case situation. As the pivot is the last element, which is the biggest in a sorted array, the technique results in significantly imbalanced partitions, with each division decreasing the array's size by just one element. This yields a temporal complexity of O(n^2), resulting in much slower performance. In a similar manner, deterministic Quicksort executes on the reverse-sorted array in around 0.061 seconds, demonstrating somewhat superior performance compared to the sorted array, however remains much slower than for the random array. This results once again from the uneven partitioning that occurs when consistently using the final element (the smallest in a reverse-sorted array) as the pivot. Conversely, randomized Quicksort demonstrates more uniform performance across various input types. In the case of the random array, randomized Quicksort executes in 0.016 seconds, little slower than deterministic Quicksort, however remains efficient due to the random pivot selection facilitating balanced partitions. In a sorted array, randomized Quicksort has markedly superior performance compared to its deterministic equivalent, requiring just 0.0049 seconds. The stochastic pivot selection guarantees that the method circumvents the worst-case situation and upholds an average time complexity of O(n log n). Likewise, for the reverse-sorted array, randomized Quicksort requires around 0.0138 seconds, once again surpassing deterministic Quicksort. The stochastic pivot selection mitigates the occurrence of persistently suboptimal partitions, leading to more equitable divisions of the array and enhanced performance. The findings indicate that while deterministic Quicksort excels with random data, it has difficulties with sorted and reverse-sorted arrays because of its static pivot selection. In contrast, randomized Quicksort ensures more consistent performance across various input types by alleviating worst-case circumstances via the selection of random pivots.

# Conclusion

Quicksort is a potent sorting algorithm owing to its efficient average-case time complexity and its versatility across many datasets. The deterministic variant of Quicksort, while efficient for random inputs, may encounter difficulties with sorted or reverse-sorted arrays because of its fixed pivot selection, leading to suboptimal performance characterized by a temporal complexity of O(n^2). Conversely, randomized Quicksort enhances this method by implementing random pivot selection, which significantly decreases the probability of worst-case situations and upholds a uniform O(nlogn) performance across various input distributions. In summary, while deterministic Quicksort performs well in several scenarios, randomized Quicksort is often more resilient and adaptable, particularly when the input data's structure is uncertain or maybe hostile, making it a favored option in various practical applications.