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

Ujjwal Khadka

University of the Cumberlands

MSCS-532-M80- Algorithms and Data Structures

Satish Penmatsa

19 Oct 2025

Quicksort is a divide and conquer sorting algorithm that is efficient and commonly used in today's world. Quicksort is used to sort the array by picking a pivot element, which divides the array into two sub arrays and those two sub arrays are sorted recursively, providing a sorted array. The running time and efficiency of quicksort highly depend on the partitioning of the array, or how the pivot element is selected. If the array is partitioned or divided into equal sub array, then the efficiency or running time of the quicksort algorithm is better, providing better performance. There are various methods for picking a pivot element to improve the efficiency and performance of quicksort algorithms, since pivot elements are an important factor in determining efficiency and performance. Two of the common methods are deterministic and randomized quicksort. In a randomized quicksort, the pivot element is chosen randomly. In a deterministic quicksort, the pivots are chosen by either always choosing the pivot at the same relative index, such as the first, last, or middle element, or by using the median of any number of predetermined element choices (Grech, 2010). In this assignment, both the deterministic and randomized versions of quicksort are implemented in the Python code to determine the running time on different input sizes and distributions and compare the running time between deterministic and randomized quicksort. Deterministic Quicksort is implemented in Python, where the middle element is chosen as the pivot.

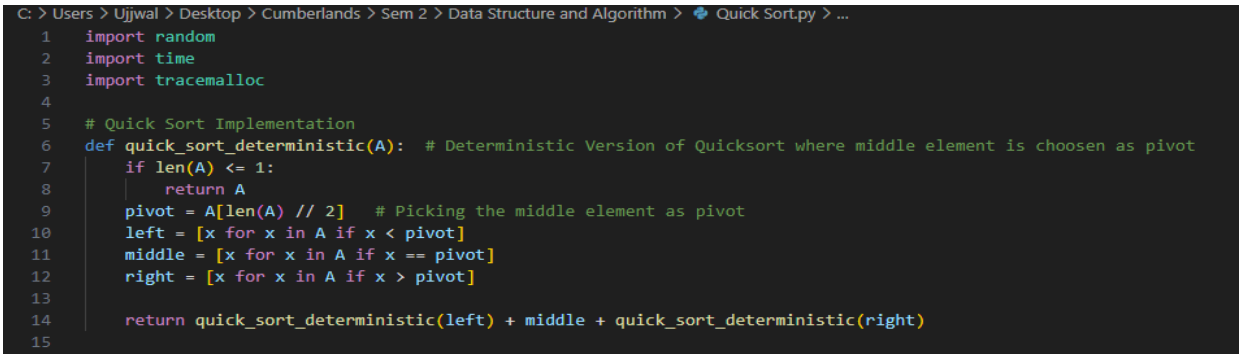

Figure 1: Screenshot of the Deterministic Quicksort implementation in python

Time Complexity

The running time and efficiency of quicksort are greatly dependent on the pivot element, which determines if the partitioning is balanced or not. If the partitioning is balanced, the running time and efficiency of quicksort are better, while if the partitioning is unbalanced, the running time and efficiency of quicksort are poor and slow.

- Worst Case: Worst case time complexity occurs when the pivot element is chosen poorly, such as picking the first or last element as the pivot element, resulting in a very unbalanced partitioning of the array. The worst-case behavior for quicksort occurs when the partitioning produces one subproblem with n – 1 elements and one with 0 elements (Cormen et al., 2022). With the recursive call on the array of size 0 elements returns without doing anything, resulting in worst case time complexity to be $T(n) = O(n^2)$.
- Best Case: Best case time complexity occurs when the pivot element is chosen in such a way that the portioning of the array is perfectly balanced, resulting in equal and even two sub arrays. With the balanced partitioning, the recurrence relation results in $T(n) = 2T(n/2) + O(n)$. Using the master method, nlogba = nlog22 = n1, which is case 2 of the master theorems, resulting in the best case time complexity to be $T(n) = O(nlogn)$.
- Average Case: Average case time complexity occurs when the pivot element is selected in such a way that the portioning occurs to create a reasonably balanced sub array, providing the performance and efficiency closer to the best case rather than the worst case. The time complexity of the average case is $T(n) = O(nlogn)$.

The worst case space complexity of quicksort is O(n) due to the unbalanced portioning resulting in a skewed recursion tree, while the average and best space complexity are O(logn) due to the balanced portioning resulting in a balanced recursion tree.

Randomized Quicksort is implemented in python where the pivot element is chosen randomly.

```python

def quick_sort_randomized(A):  # Randomized Version of Quciksort where pivot element is picked in random
    if len(A) <= 1:
        return A
    pivot = random.choice(A) #Picking the pivot element in random
    left = [x for x in A if x < pivot]
    middle = [x for x in A if x == pivot]
    right = [x for x in A if x > pivot]

    return quick_sort_randomized(left) + middle + quick_sort_randomized(right)

```

Figure 2: Screenshot of the Randomized Quicksort implementation in python

Empirical Analysis

Below is the screenshot of the Python code used to determine the running time for different input sizes and distributions and compare the deterministic and randomized Quicksort.

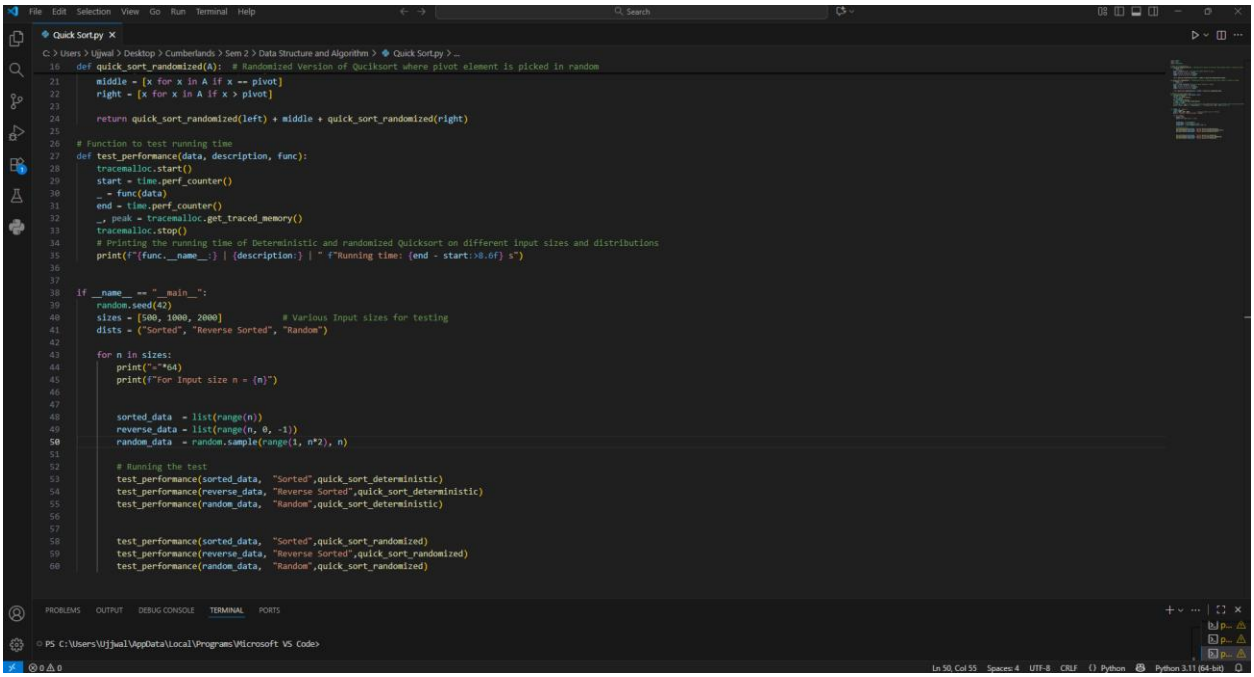

Figure 3: Screenshot of the python code to determine and display the running time

From running the code above, various data and results were obtained. With the help of the data and results obtained, various analyses can be performed, as the effects of deterministic

and randomized quicksort can be seen, as well as comparing and determining if the experimental data and results relate to the theoretical analysis. The screenshot below shows the experimental results and data obtained from the code.

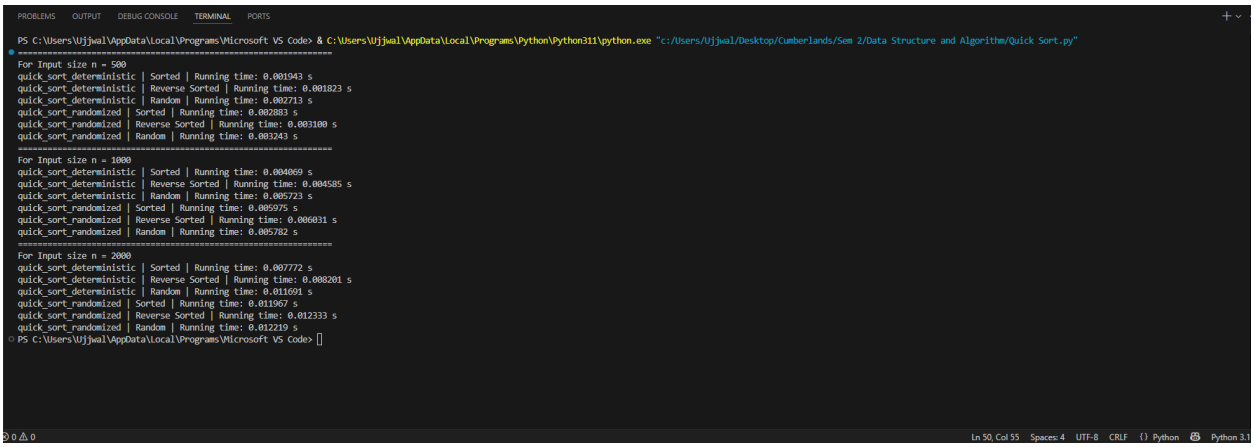

Figure 4: Screenshot of the output of the code showing the running time

From the running time result seen above between different input sizes and distributions, we can see that the running time for randomized quicksort is consistent compared to deterministic quicksort. Also, the running time between each distribution is different or varies in deterministic quicksort while the running time between each distribution is similar in randomized quicksort which matches with the theoretical analysis that suggests that randomized quicksort where the pivot is picked randomly is similar to best case and average case which provide better efficiency and performance preventing the occurrence of worst case scenarios by picking the pivot at random. In conclusion, it can be seen from theoretical and experimental analysis that the randomized quicksort provides better efficiency and performance compared to the deterministic quicksort. Since deterministic quicksort picks the determined pivot element, resulting in an unbalanced array, as compared to randomized quicksort, which results in a reasonably balanced array, showing consistent running time and providing better efficiency and performance.

Link to the GitHub repository: https://github.com/Ujjwal0004/MSCS532_Assignment5

References

Grech, A. (2010, February 22). *What is a Deterministic Quicksort?* Stack Overflow. https://stackoverflow.com/questions/2313940/what-is-a-deterministic-quicksort

Cormen, T. H., Charles Eric Leiserson, Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. The Mit Press.