## **Quicksort Implementation and Analysis**

### **1. Implementation**

To make the implementation and the analysis of the Quicksort algorithm, I have implemented the following versions/approach of the quicksort.

* Deterministic Quicksort: This approach will use the middle element from the array as the pivot
* Randomized Quicksort: this approach will use a random element from the array as the pivot. Even though it is randomly selected, it is implemented to avoid the worst-case scenarios.

The way each approach works is it will use recursively partitions the array around the pivot, sorting subarrays until the entire elements on an array is ordered.

### **2. Performance Analysis**

#### **2.1 Time Complexity**

Following table shows the time complexities on different case

| Case | Time Complexity |
| --- | --- |
| Best Case | O (n log n ) |
| Average Case | O (n log n) |
| Worst Case | O (n ^2) |

* The best and average cases occur when the pivot approximately evenly splits the array.
* The average-case complexity is derived from a probabilistic analysis of all pivot positions over many inputs.
* There will be the worst case scenario if the selected pivot element make the partitions very unbalanced.

#### **2.2 Space Complexity**

Quicksort is an in-place algorithm, which means it does not require extra memory for use by another array, but it is still using stack space due to recursive function calls. The recursion depth has an average case of O(log n) and, therefore, the space used on the stack is O(log n) as well.

In the worst case (again, when the input is already sorted or reverse sorted with poor pivot selection) the recursion depth can be O(n). This causes more stack usage, and if the input is very large, a stack overflow could occur. Apart from the call stack, no additional memory overhead is used, so in most cases, Quicksort uses very little space.

* In the average case, it will be O (log n)
* The algorithm is in-place apart from the recursion stack.

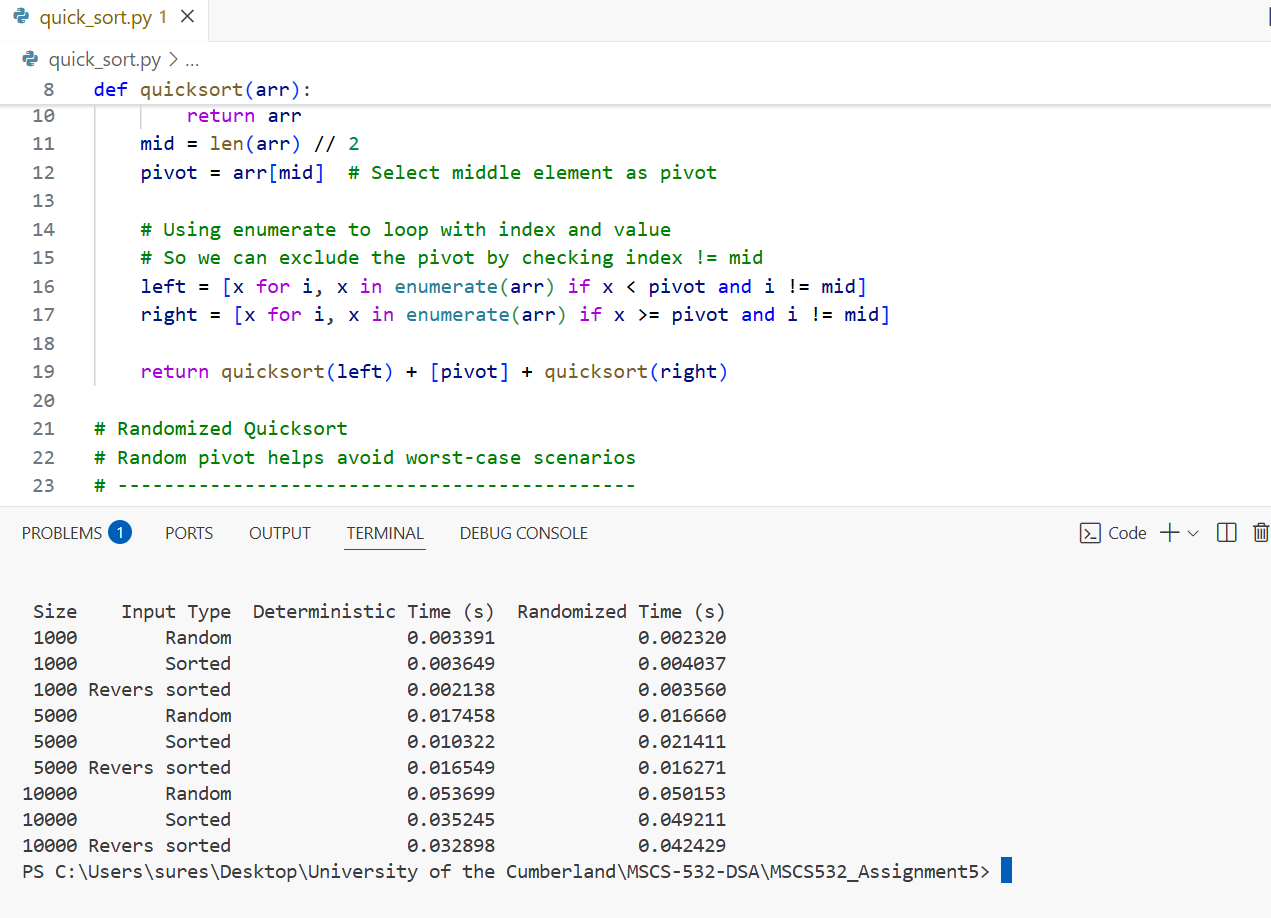
### **3. Randomized Quicksort**

The randomized version of Quicksort selects a pivot at random from the array. This randomness reduces the chances of encountering the worst-case scenario O(n^2), especially for already sorted or reverse-sorted inputs.

### **4. Empirical Analysis**

The following table shows the benchmark between different algorithms with different input types and sizes.

| Size | Input Size | Deterministic Time (in seconds) | Randomized Time (in seconds) |
| --- | --- | --- | --- |
| 1000 | Random | 0.003391 | 0.002320 |
| 1000 | Sorted | 0.003649 | 0.004037 |
| 1000 | Reverse Sorted | 0.002138 | 0.003560 |
|  |  |  |  |
| 5000 | Random | 0.017458 | 0.016660 |
| 5000 | Sorted | 0.010322 | 0.021411 |
| 5000 | Reverse Sorted | 0.016549 | 0.016271 |
|  |  |  |  |
| 10000 | Random | 0.053699 | 0.050153 |
| 10000 | Sorted | 0.035245 | 0.049211 |
| 10000 | Reverse Sorted | 0.032898 | 0.042429 |



The pandas library was needed in a usable format and is a framework for working with structured data.

#### **4.1 Observations**

All of the results are consistent with the theoretical assessment. In the case of random input, both the deterministic and randomized Quicksort operation produced performance consistent with the theoretical expected average-case time complexity of O(n log n). The results also corroborate the idea that Quicksort is an efficient option in this type of running environment (random data).

While the deterministic version of Quicksort did demonstrate superior performance on this particular set of sorted/reverse-sorted inputs, this can be explained by the fact that the deterministic version could occasionally select a middle pivot that produced well balanced partitioning. All things considered, the randomized version also produced better-more consistent performance across all types of inputs. The randomized version has an advantage to avoid poor performance in worst case scenarios.

* The deterministic and randomized versions both function very similarly with random data.
* The randomized version is more consistent across input types and provides an advantage for worst case scenario avoidance.
* The deterministic version is faster on sorted/reverse-sorted data in this run but this may very well be attributed to good pivot choices leading in this case to a good balanced partitioning.

### **5. Conclusion**

Quicksort is a fast and efficient algorithm for most practical cases. Randomizing the pivot selection improves its robustness against specific input patterns that would degrade performance. Our tests show that both implementations work effectively, with randomized Quicksort providing added protection against worst-case inputs.