

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

Nitish Dhinakaran

University of the Cumberland

MSCS-532-M20 Algorithms and Data Structures

Dr. Brandon Bass

June 28, 2025

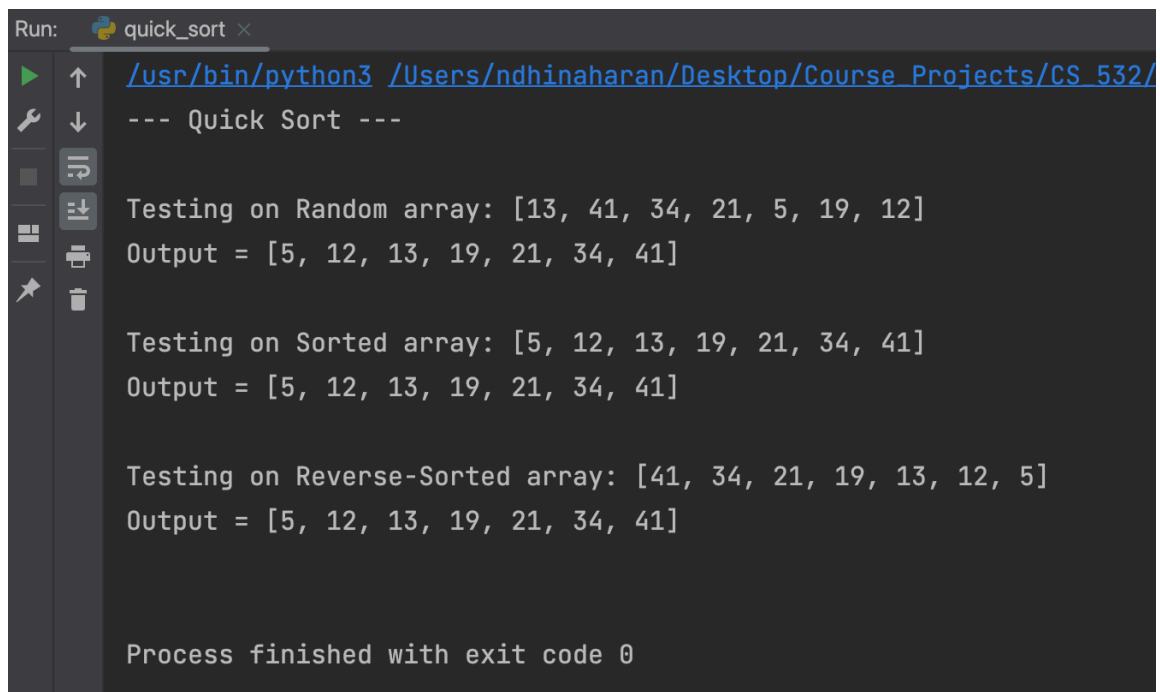
## Github Link to Source Code of Implementation

[https://github.com/ndhinaharan36295/MSCS-532\\_Assignment5](https://github.com/ndhinaharan36295/MSCS-532_Assignment5)

## Quicksort Implementation and Analysis

### Implementation and Execution

Quicksort is a classic divide-and-conquer sorting algorithm known for its efficiency in practical applications. I implemented this version of the quicksort in Python. This algorithm always selects the last element of the current array during each recursive call, as the pivot element (deterministic), and partitions accordingly (Cormen et al., 2022). As it can be seen in Figure 5.1, my implementation executes the sorting and correctly returns the sorted output. It can also be seen that it handles all the possible cases such as random arrays, sorted arrays, reverse-sorted arrays.



```
Run: quick_sort x
/usr/bin/python3 /Users/ndhinaharan/Desktop/Course Projects/CS 532/
--- Quick Sort ---
Testing on Random array: [13, 41, 34, 21, 5, 19, 12]
Output = [5, 12, 13, 19, 21, 34, 41]
Testing on Sorted array: [5, 12, 13, 19, 21, 34, 41]
Output = [5, 12, 13, 19, 21, 34, 41]
Testing on Reverse-Sorted array: [41, 34, 21, 19, 13, 12, 5]
Output = [5, 12, 13, 19, 21, 34, 41]
Process finished with exit code 0
```

**Figure 5.1** Quicksort execution

## Performance Analysis

The time complexity of Quicksort depends heavily on how the array is partitioned at each recursive step. It uses a divide-and-conquer strategy: it partitions the array around a pivot, then recursively sorts the two subarrays (Cormen et al., 2022).

### 1) Best case

The pivot divides the array into two equal halves (or nearly equal). At each level of recursion, we process all  $n$  elements once (in the partition step), and the recursion depth is  $\log n$  (Cormen et al., 2022).

An upper bound on the running time can then be described by the recurrence:

$$T(n) = 2T(n/2) + \Theta(n)$$

Therefore, solving this recurrence gives us the time-complexity:  $T(n) = \Theta(n \log n)$

### 2) Average case

The partition in this algorithm always produces a 9-to-1 proportional split (since the pivot is always determined as the last element), which seems quite unbalanced (Cormen et al., 2022).

We obtain the recurrence:

$$T(n) = T(9n/10) + T(n/10) + \Theta(n)$$

Therefore, solving this recurrence gives us the time-complexity:  $T(n) = \Theta(n \log n)$

### 3) Worst case

In the worst case, the pivot ends up being the largest element at each step. Typical scenarios of the worst case behavior for the deterministic pivot (always picking the last element) occurs when sorting already sorted or reverse-sorted arrays. The worst case partition always produces one subtree with  $n - 1$  elements and the other one with 0 elements. The recursion tree becomes skewed with depth  $n$ . The partitioning costs  $\Theta(n)$  time (Cormen et al., 2022).

We obtain the recurrence:

$$T(n) = T(n-1) + \Theta(n)$$

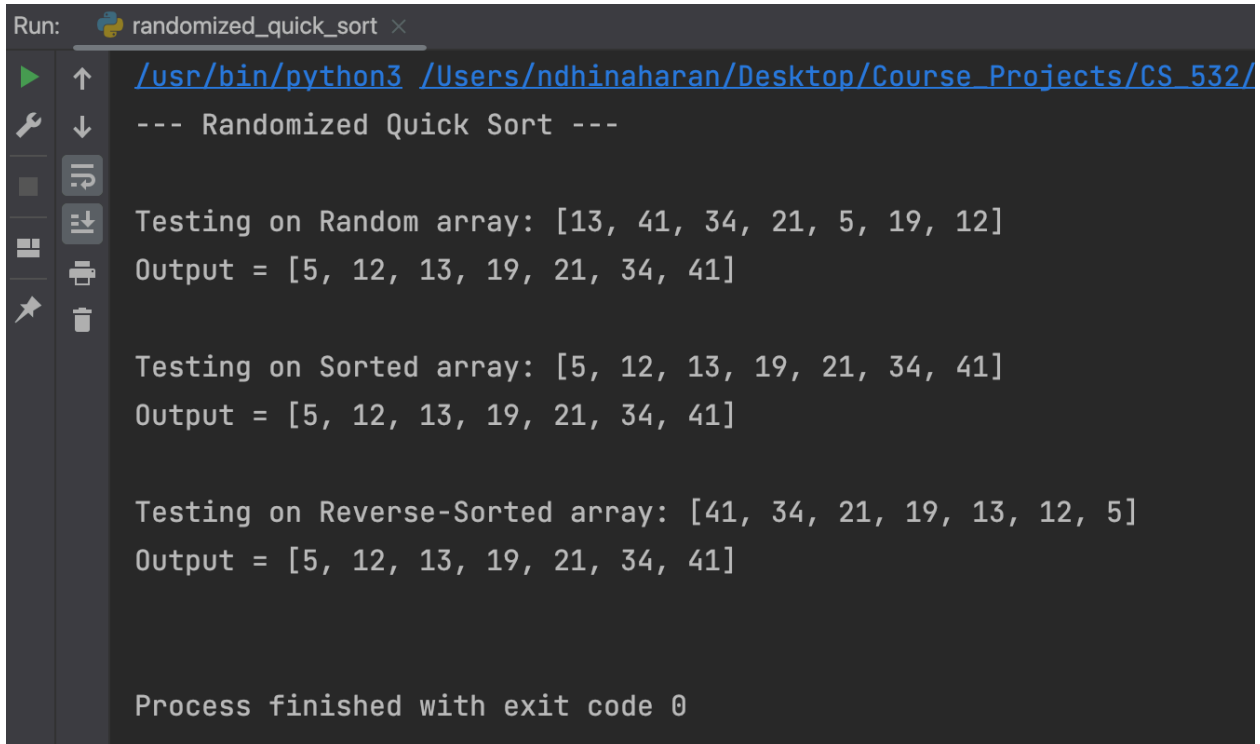
Therefore, solving this recurrence gives us the time-complexity:  $T(n) = \Theta(n^2)$

The space complexity of this algorithm depends on the auxiliary space and recursive stack depth used by the algorithm. Quicksort is an in-place sorting algorithm and hence, it doesn't use extra arrays for subarrays — meaning  $O(1)$  space. So the space overhead is only due to recursion (call stack). As mentioned above, the recursive stack depth for the best & average case is  $\Theta(\log n)$ , and  $\Theta(n)$  in the worst case. Therefore, the space complexity for the best & average case is  $O(\log n)$ , and  $O(n)$  in the worst case (Cormen et al., 2022).

### **Randomized Quicksort**

I implemented the randomized version of the quicksort in Python. This algorithm always selects a pivot randomly from the subarray before applying the standard partitioning and quicksort routine. This randomization reduces the likelihood of encountering worst-case scenarios on already sorted or patterned data (Cormen et al., 2022). As it can be seen in Figure

5.2, my implementation executes the sorting and correctly returns the sorted output for all the possible cases such as random arrays, sorted arrays, reverse-sorted arrays.



```
Run: randomized_quick_sort x
/usr/bin/python3 /Users/ndhinaharan/Desktop/Course Projects/CS_532/
--- Randomized Quick Sort ---

Testing on Random array: [13, 41, 34, 21, 5, 19, 12]
Output = [5, 12, 13, 19, 21, 34, 41]

Testing on Sorted array: [5, 12, 13, 19, 21, 34, 41]
Output = [5, 12, 13, 19, 21, 34, 41]

Testing on Reverse-Sorted array: [41, 34, 21, 19, 13, 12, 5]
Output = [5, 12, 13, 19, 21, 34, 41]

Process finished with exit code 0
```

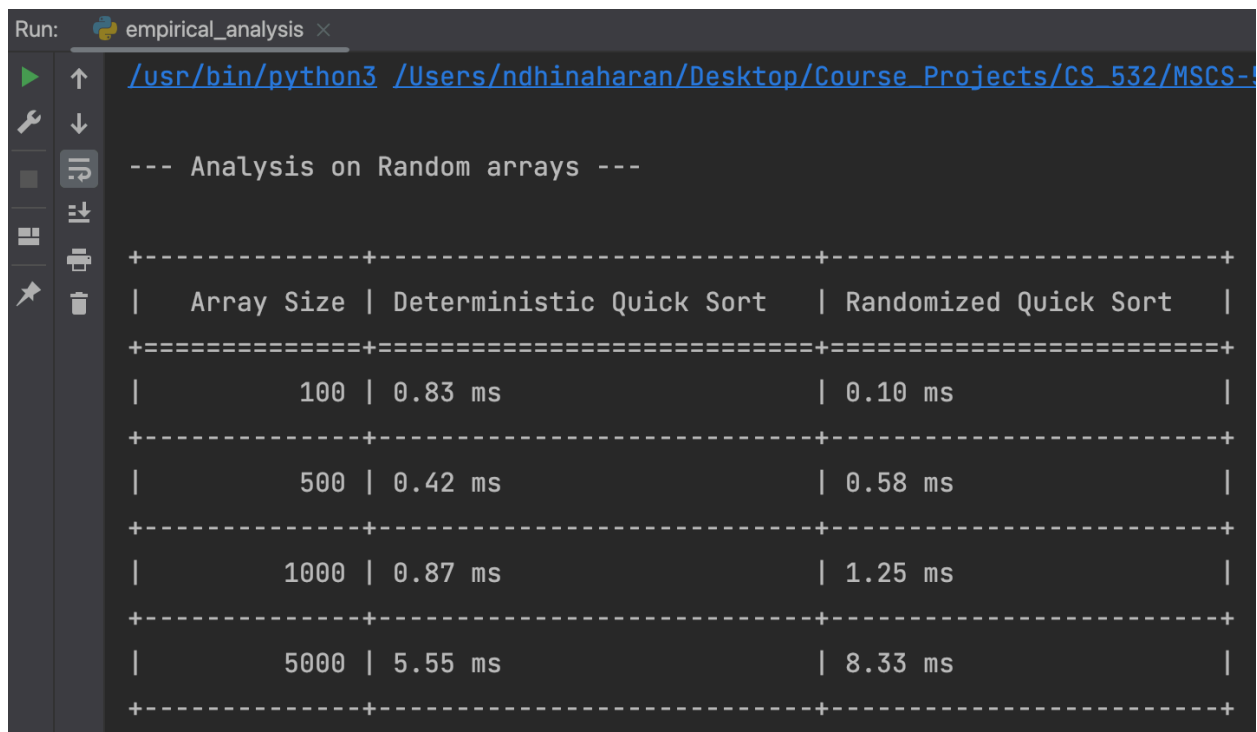
**Figure 5.2** Randomized quicksort execution

The effect of randomization in quicksort is that it combats input-dependent worst-case scenarios by eliminating fixed pivot bias. For example, in deterministic quicksort, an already sorted array results in  $O(n^2)$  time due to poor partitioning caused by fixed pivot choices (like either first or last element of the subarray). However, randomized Quicksort reduces the chance of such poor partitions, leading to expected  $O(n \log n)$  performance even on adverse inputs. Therefore, this randomization of the pivot makes performance more stable across varied input patterns, and maintains the same average-case time complexity but with greater practical robustness (Cormen et al., 2022).

## Empirical Comparison

I ran both the versions of the quicksort algorithms (deterministic and randomized quicksort) on different types of inputs (random arrays, sorted arrays, reverse-sorted arrays) and different input sizes (100, 500, 1000, 5000) to perform comparisons of the runtimes of these algorithms and how the runtime varies based on different inputs and different input sizes. I used Python's in-built tabulate package to showcase the runtimes which would make it easier to compare and analyse

As it can be seen in Figure 5.3, for random arrays, both versions of the quicksort are consistently fast, which is as expected from our theoretical analysis (Cormen et al., 2022).



```
Run: empirical_analysis x
/usr/bin/python3 /Users/ndhinaharan/Desktop/Course Projects/CS_532/MSCS-1

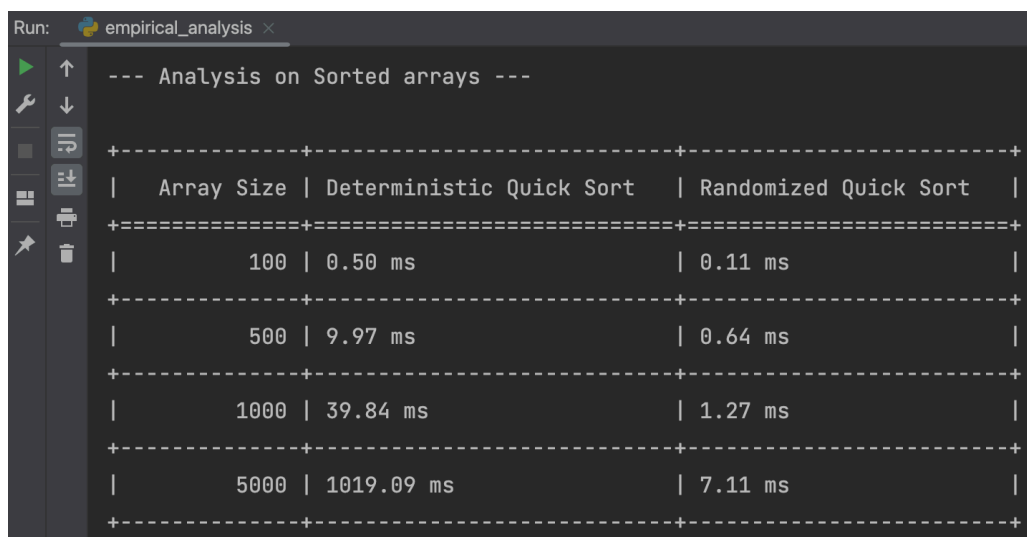
--- Analysis on Random arrays ---

+-----+-----+-----+
| Array Size | Deterministic Quick Sort | Randomized Quick Sort |
+-----+-----+-----+
|      100 | 0.83 ms | 0.10 ms |
+-----+-----+-----+
|      500 | 0.42 ms | 0.58 ms |
+-----+-----+-----+
|     1000 | 0.87 ms | 1.25 ms |
+-----+-----+-----+
|     5000 | 5.55 ms | 8.33 ms |
+-----+-----+-----+
```

**Figure 5.3** comparison of sorting algorithms on random arrays

But the runtime of deterministic quicksort drastically increases, as it can be seen in Figure 5.4, on sorted and reverse sorted arrays. This is due to the fact of the deterministic pivot

choice which is considered to be poor for already sorted arrays. This problem is addressed by randomizing the pivot choices, as per our theoretical analysis earlier. This can be proven from the massive difference in the runtimes between the randomized and deterministic version, as seen in Figure 5.4 and Figure 5.5, as the input sizes keep growing. Randomized quicksort is consistently fast even with growing input sizes, and outperforms the deterministic version of every input size.



Run: empirical\_analysis x

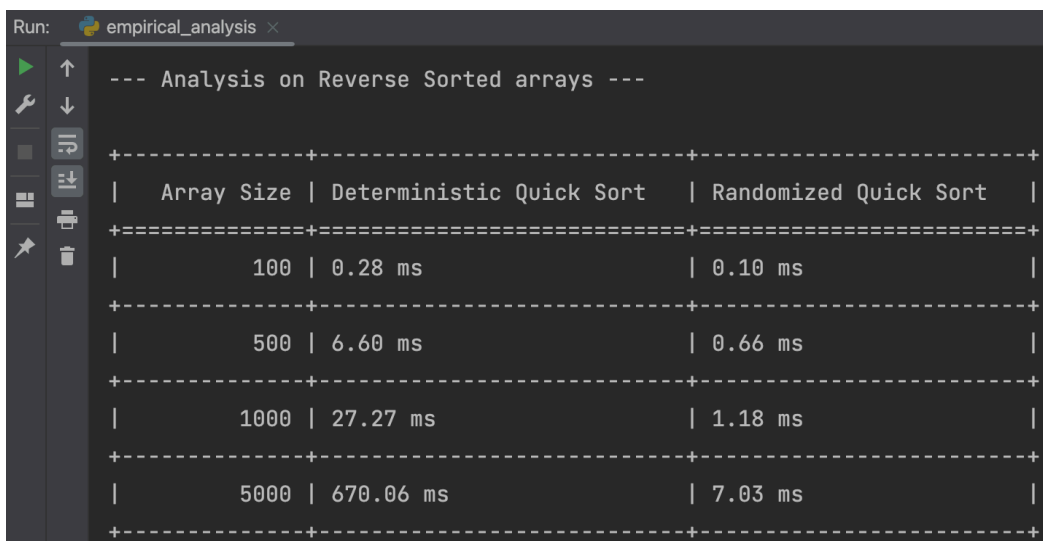
```

--- Analysis on Sorted arrays ---

+-----+-----+-----+
| Array Size | Deterministic Quick Sort | Randomized Quick Sort |
+-----+-----+-----+
|      100 | 0.50 ms                  | 0.11 ms                |
+-----+-----+-----+
|      500 | 9.97 ms                  | 0.64 ms                |
+-----+-----+-----+
|     1000 | 39.84 ms                 | 1.27 ms                |
+-----+-----+-----+
|     5000 | 1019.09 ms               | 7.11 ms                |
+-----+-----+-----+

```

**Figure 5.4** comparison of the algorithms on sorted arrays



Run: empirical\_analysis x

```

--- Analysis on Reverse Sorted arrays ---

+-----+-----+-----+
| Array Size | Deterministic Quick Sort | Randomized Quick Sort |
+-----+-----+-----+
|      100 | 0.28 ms                  | 0.10 ms                |
+-----+-----+-----+
|      500 | 6.60 ms                  | 0.66 ms                |
+-----+-----+-----+
|     1000 | 27.27 ms                 | 1.18 ms                |
+-----+-----+-----+
|     5000 | 670.06 ms                | 7.03 ms                |
+-----+-----+-----+

```

**Figure 5.5** comparison of the algorithms on reverse-sorted arrays

## References

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