# Quicksort Algorithm: Implementation, Analysis and Randomization

**Name:** Kevin Chemutai
**Student ID:** 005029582
**Course Title:** Data Structures and Algorithms.

**Assignment Number:** 5

**GitHub Link**: https://github.com/kchemutai/Quicksort-Analysis

## Quicksort-Analysis

**Quick Sort implantation:** quick_sort.py

**Randomized quick sort Implementation:** randomized_quick_sort.py

**Performance Analysis**

***Time Complexity Analysis***

***Best Case: O (n log n)*** Occurs when the pivot splits the array into two roughly equal halves at every recursive step.

***Average Case: O (n log n)*** On average, the pivot divides the array into two reasonably balanced partitions.

***Worst Case: O(n^2)*** Happens when the pivot is always the smallest or largest element, leading to highly unbalanced partitions (e.g., already sorted arrays in deterministic Quicksort).

***Space Complexity***

In-place Implementation: O (log n) (stack space for recursive calls). Additional Overheads: If not in-place, extra space is required for temporary arrays during partitioning.

**Summary of Empirical Analysis Results**

+ Code + Text | Copy to Drive

```python
# Run comparisons
results = []
for size in input_sizes:
    for dist_name, generator in distributions.items():
        arr = generator(size)
        deterministic_time = measure_time(quicksort, arr)
        randomized_time = measure_time(randomized_quicksort, arr)
        results.append({
            "Input Size": size,
            "Distribution": dist_name,
            "Deterministic Time (s)": deterministic_time,
            "Randomized Time (s)": randomized_time
        })

# Print results
results_df = pd.DataFrame(results)
print(results_df)
```

```
    Input Size    Distribution  Deterministic Time (s)  Randomized Time (s)
0           10          random                0.000034             0.000032
1           10          sorted                0.000020             0.000024
2           10  reverse_sorted                0.000018             0.000026
3          100          random                0.000319             0.000350
4          100          sorted                0.000237             0.000369
5          100  reverse_sorted                0.000246             0.000321
6         1000          random                0.004604             0.006429
7         1000          sorted                0.003950             0.006851
8         1000  reverse_sorted                0.004638             0.024917
9        10000          random                0.090188             0.091728
10       10000          sorted                0.032288             0.053671
11       10000  reverse_sorted                0.042785             0.063005
```

The empirical analysis of deterministic and randomized Quicksort reveals the following insights:

***Small Input Sizes (10 elements):***

Both deterministic and randomized Quicksort have negligible runtimes (< 0.0001 seconds). There is minimal performance difference between the two approaches, with slight variations due to randomness in pivot selection.

***Medium Input Sizes (100 elements):***

Both deterministic and randomized Quicksort show a marginal increase in runtime compared to smaller inputs. Randomized Quicksort is slightly slower than deterministic Quicksort for sorted and reverse-sorted inputs due to the overhead of random pivot selection.

***Large Input Sizes (1000 elements):***

For random and sorted inputs, deterministic Quicksort performs marginally faster than randomized Quicksort. For reverse-sorted inputs, randomized Quicksort significantly underperforms, indicating that while randomization reduces the likelihood of the worst-case scenario, it may still encounter performance issues in certain cases.

***Very Large Input Sizes (10,000 elements):***

Deterministic Quicksort outperforms randomized Quicksort consistently for all input distributions. For sorted and reverse-sorted distributions, the deterministic approach exhibits better runtime efficiency, highlighting its stability under specific conditions. Randomized Quicksort takes longer to process reverse-sorted inputs, though it performs similarly to deterministic Quicksort for random distributions.

***Overall Trends:***

Randomized Quicksort tends to introduce additional overhead due to pivot randomization, making it slightly slower in practice compared to deterministic Quicksort for smaller input sizes and sorted/reverse-sorted distributions. Deterministic Quicksort is more predictable and generally faster, particularly for large input sizes and structured data distributions.

***Performance Breakdown by Input Distribution:***

***Random Distribution:*** Both algorithms perform similarly, with randomized Quicksort occasionally being slower. Sorted Distribution: Deterministic Quicksort consistently outperforms randomized Quicksort.

***Reverse-Sorted Distribution:*** Deterministic Quicksort is significantly faster, especially for larger input sizes.

**Key Observations:** ***Randomized Quicksort*** is effective in avoiding the worst-case scenario of O(n^2) but introduces slight overhead due to randomness.

***Deterministic Quicksort*** shows better practical performance across most cases and is more stable for sorted and reverse-sorted inputs.

These results align with theoretical expectations. while randomization helps reduce worst-case likelihood, deterministic Quicksort offers better overall runtime efficiency in practical scenarios.