Assignment 5

Shreyas Santosh More

University of Cumberlands

Algorithms and Data Structures (MSCS-532-B01)

Dr.Bass

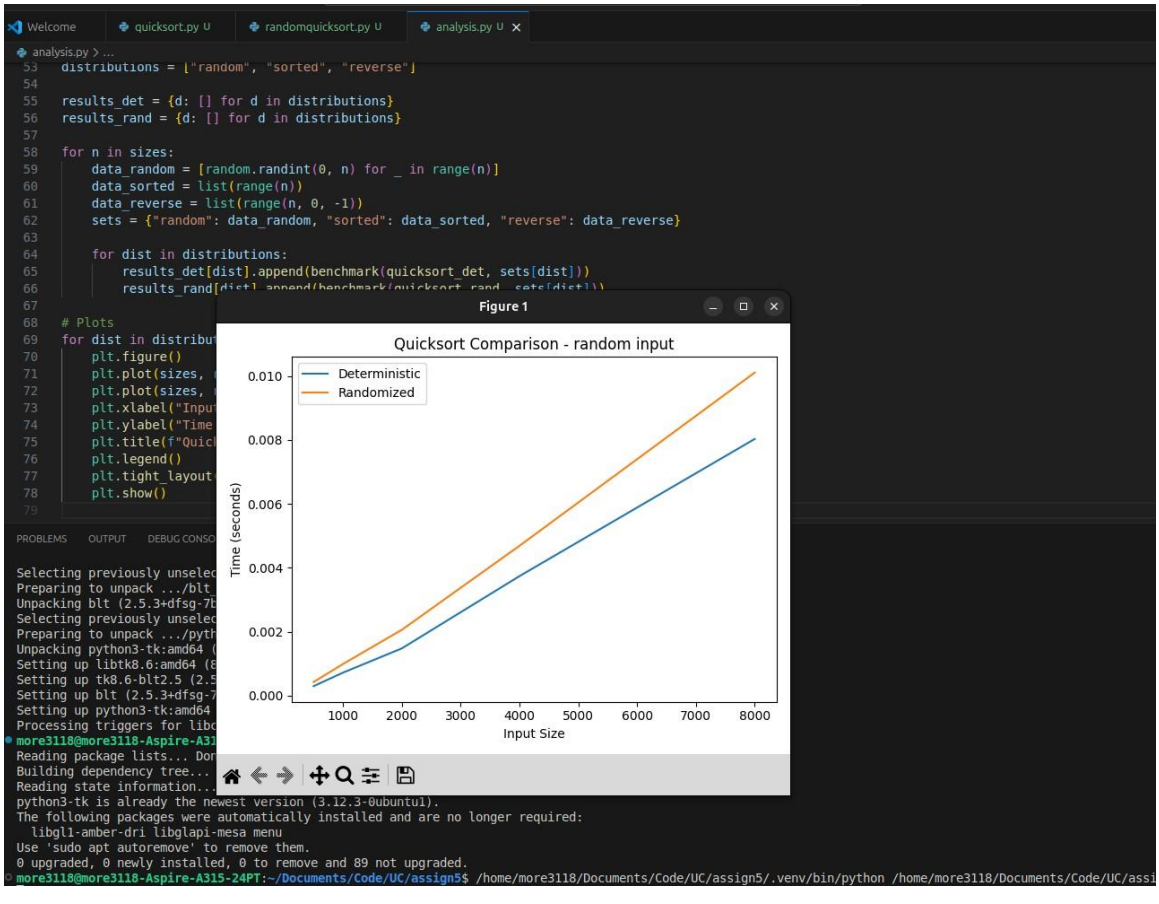
November 16, 2025

GitHub: https://github.com/smore38997/assign5

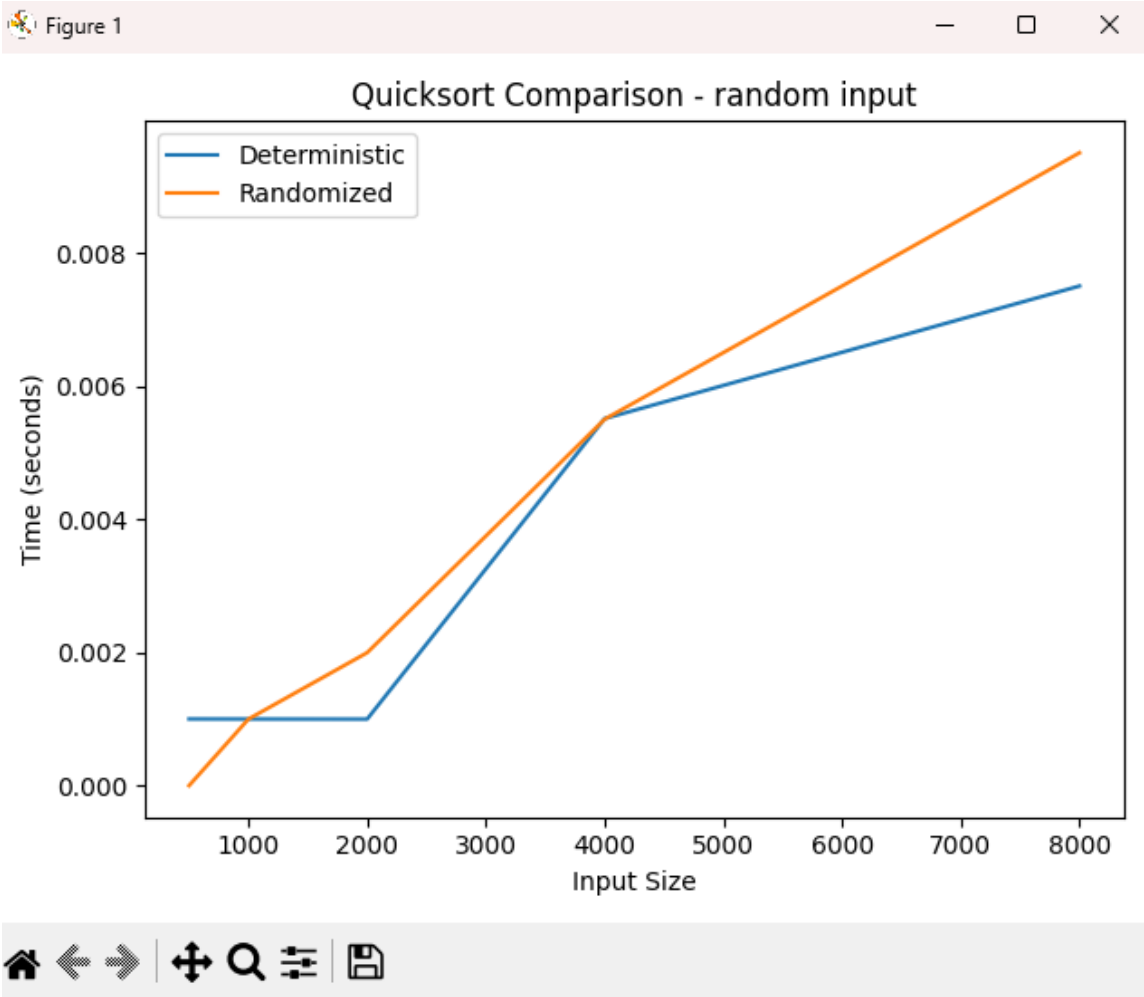
Install Matplotlib before proceeding. Use Virtual environment if required to run the programming

```
Sorted array: [1, 5, 7, 8, 9, 10]
more3118@more3118-Aspire-A315-24PT:~/Documents/Code/UC/assign5$ /bin/python3 /home/more3118/Documents/Code/UC/assign5/analysis.py
Traceback (most recent call last):
  File "/home/more3118/Documents/Code/UC/assign5/analysis.py", line 5, in <module>
    import matplotlib.pyplot as plt
ModuleNotFoundError: No module named 'matplotlib'
more3118@more3118-Aspire-A315-24PT:~/Documents/Code/UC/assign5$ py install matplotlib
Command 'py' not found, but can be installed with:
sudo apt install pythonpy
more3118@more3118-Aspire-A315-24PT:~/Documents/Code/UC/assign5$ pip install matplotlib
error: externally-managed-environment

× This environment is externally managed
╰─> To install Python packages system-wide, try apt install
    python3-xyz, where xyz is the package you are trying to
    install.
```

Result of comparison (Empircal Analysis)



Running it for a second time



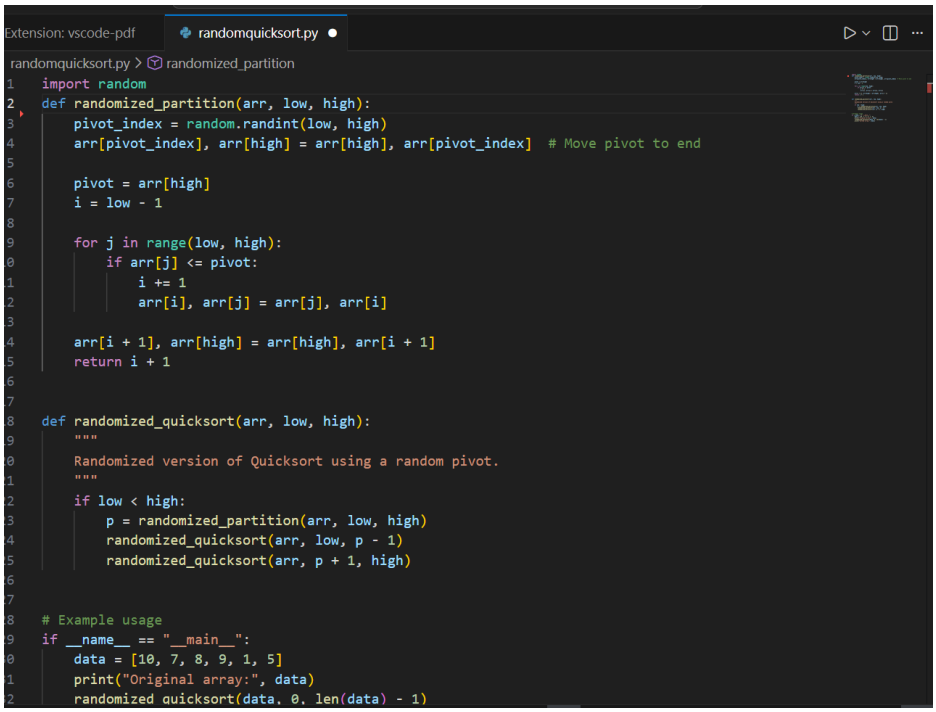
Code for Benchmarking both functions

```python
# Benchmark
def benchmark(sort_func, data):
    arr = data.copy()
    start = time.time()
    sort_func(arr, 0, len(arr)-1)
    return time.time() - start

sizes = [500, 1000, 2000, 4000, 8000]
distributions = ["random", "sorted", "reverse"]

results_det = {d: [] for d in distributions}
results_rand = {d: [] for d in distributions}

for n in sizes:
    data_random = [random.randint(0, n) for _ in range(n)]
    data_sorted = list(range(n))
    data_reverse = list(range(n, 0, -1))
    sets = {"random": data_random, "sorted": data_sorted, "reverse": data_reverse}

    for dist in distributions:
        results_det[dist].append(benchmark(quicksort_det, sets[dist]))
        results_rand[dist].append(benchmark(quicksort_rand, sets[dist]))
```

Determinstic Quick Sort

```python
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1  # pointer for smaller elements

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Place pivot in correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def quicksort(arr, low, high):
    """
    Sorts the array arr[low:high+1] in place using Quicksort.
    """
    if low < high:
        # Partition step: arr[p] is now at correct position
        p = partition(arr, low, high)

        # Recursively sort elements before and after partition
        quicksort(arr, low, p - 1)
        quicksort(arr, p + 1, high)


# Example usage:
if __name__ == "__main__":
    data = [10, 7, 8, 9, 1, 5]
    print("Original array:", data)
```

Randomized Quick sort



Observation is that the Randomized Quicksort has less performance than deterministic as input size gets bigger.

			Worst-case likelihood	
Version	**Average-case**	**Worst-case**		**Notes**
Deterministic	O(n \log n)	O(n^2)	Moderate on	Very sensitive to input **Worst-case**
Version	**Average-case**	**Worst-case**		**Notes**
			likelihood	
pivot			structured inputs	order
Randomized pivot	O(n log n)	O(n^2)	Extremely low	Hard to engineer worst-case

Median-of-three	O(n log n)	O(n^2)	Low	Heuristic, not proof-level protection

Further Quicksort Time and Space Complexity Analysis

1. Best Case: O(n log n)

In the best case, the pivot divides the array into two nearly equal halves. The recurrence T(n) = 2T(n/2) + O(n) solves to O(n log n).

2. Average Case: O(n log n)

On average, partitions tend to be reasonably balanced. Solving the expected recurrence leads to O(n log n).

3. Worst Case: O(n^2)

Occurs when the pivot always produces maximally unbalanced partitions. The recurrence T(n)=T(n−1)+O(n) expands to O(n^2).

4. Space Complexity

Average: O(log n) due to recursion stack.

Worst: O(n) with highly unbalanced recursion.

Quicksort is in-place aside from recursion overhead.

Analysis;

How Randomization Affects Quicksort Performance and Reduces Worst-Case Likelihood

Quicksort ranks among the top sorting methods based on comparisons, mainly due to solid real-world speed, straightforward design, yet dependable results overall. Still, runtime shifts noticeably depending on pivot choice - poor choices lead to slow outcomes. Normally, worst performance hits $O(n^2)$ if splits are highly uneven each time. That happens whenever

pivots end up being either minimal or maximal elements, especially in sorted or manipulated data sequences (Cormen et al., 2022). Using random picks helps prevent such weak behavior effectively.

Randomized Quicksort works better because it picks pivots at random instead of following a set method like taking the first or last item. This randomness helps dodge inputs that cause slow performance. When pivots are selected unpredictably, chances drop sharply that splits become extremely uneven. Across repeated steps, partitions tend to even out on average due to chance regularity. As a result, typical runtime lands around O(n log n), as shown in prior work (Goodrich et al., 2014).

Randomization adds unpredictability, disrupting how input layout affects pivot selection. Because of this, when someone tries to design data that triggers poor Quicksort performance, they can't know which pivots will be picked (Cormen et al., 2022). The selected pivot doesn't depend on how the data is arranged. Thanks to this trait, Randomized Quicksort works well even if inputs are nearly sorted, completely reversed, or built to slow it down. Moreover, using randomness lowers the average depth of recursive steps. If a problem has size n, there's a 50% chance the pivot lies in the central half of the data. In such cases, splits result in smaller parts, each at most 3n/4 in size - keeping total depth close to log n. Though some divisions might be uneven now and then, the odds that every level suffers bad splits drop sharply with more layers. Hence, while extreme slowdowns could happen in theory, they almost never do in real use (Sedgewick & Wayne, 2017).

In practice, picking pivots at random is now common in libraries and live systems since it prevents slowdowns while keeping code simple and memory use unchanged. Generating random values takes little time when compared to total sorting effort. Because of this,

Randomized Quicksort performs steadily on many types of data, which leads developers to choose it often in actual software.

In short, using randomness helps Quicksort pick pivots in a way that doesn't depend on how data is ordered. As a result, worst-case $O(n^2)$ behavior becomes much less likely, leading instead to an average runtime of $O(n \log n)$, no matter the input layout. By relying on chance to balance splits, this version performs reliably across different cases - so it’s often better than fixed pivot methods.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.

Sedgewick, R., & Wayne, K. (2017). *Algorithms* (4th ed.). Addison-Wesley.