Assignment 5

Prashanna Acharya

Algorithms and Data Structures (MSCS-532-M80)

University of the Cumberlands

GitHub Link: https://github.com/prashcodes10

Quicksort Implementation and Analysis

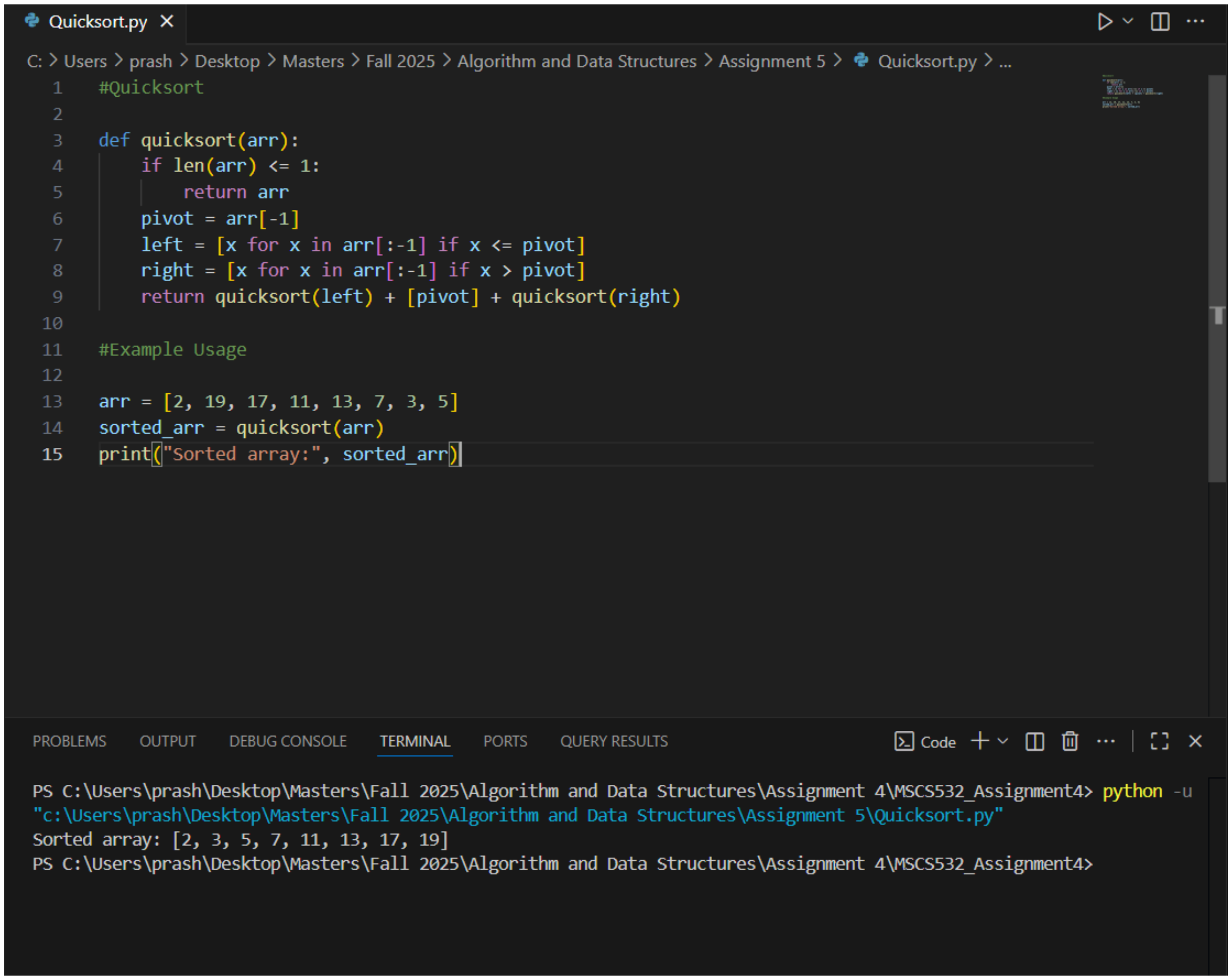
Introduction

Sorting is one of the most fundamental problems in computer science and plays a crucial role in various applications such as searching, data organization, and optimization. Among the many sorting algorithms developed, Quicksort is widely regarded for its efficiency and elegant divide-and-conquer approach. Its ability to sort large datasets efficiently in-place has made it a popular choice in both theoretical and practical contexts. However, the performance of Quicksort can vary significantly depending on how the pivot is selected during the partitioning process. This report presents a comparative study of the deterministic and randomized versions of the Quicksort algorithm. The study includes a detailed implementation of both approaches, theoretical performance analysis, and empirical evaluation across various input distributions and sizes.

Quicksort Algorithm

The Quicksort algorithm operates by selecting a pivot element from the input array, and partitioning the remaining elements into two subarrays— where one contains elements less than or equal to the pivot and the other contains elements greater than the pivot. These subarrays are then recursively sorted using the same strategy, and the final sorted array is obtained by concatenating the results. In the deterministic version of Quicksort, the pivot is typically chosen as the last element of the array.

The Python implementation of deterministic Quicksort begins by checking if the input array has one or zero elements, in which case it is already sorted and returned immediately. If not, the last element is chosen as the pivot. The array is then partitioned into two lists: one for elements less than or equal to the pivot, and another for elements greater than the pivot. These two lists are recursively sorted using the same function, and the results are combined with the pivot in the middle to produce the final sorted array.


Quicksort.py
C: > Users > prash > Desktop > Masters > Fall 2025 > Algorithm and Data Structures > Assignment 5 > Quicksort.py > ...
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
QUERY RESULTS
Code

```python
#Quicksort

def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[-1]
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]
    return quicksort(left) + [pivot] + quicksort(right)

#Example Usage

arr = [2, 19, 17, 11, 13, 7, 3, 5]
sorted_arr = quicksort(arr)
print("Sorted array:", sorted_arr)
```

```
PS C:\Users\prash\Desktop\Masters\Fall 2025\Algorithm and Data Structures\Assignment 4\MSCS532_Assignment4> python -u
"c:\Users\prash\Desktop\Masters\Fall 2025\Algorithm and Data Structures\Assignment 5\Quicksort.py"
Sorted array: [2, 3, 5, 7, 11, 13, 17, 19]
PS C:\Users\prash\Desktop\Masters\Fall 2025\Algorithm and Data Structures\Assignment 4\MSCS532_Assignment4>
```

This implementation is straightforward and correctly follows the fundamental logic of Quicksort. However, it can suffer from poor performance when the pivot does not effectively split the array, particularly when the input is already sorted or reverse sorted.

Performance Analysis

The efficiency of the Quicksort algorithm is highly dependent on how well the pivot divides the array. In the best-case scenario, the pivot divides the array into almost two equal halves at each recursive step. This results in a recursive depth of $\log n$, and since each level of recursion involves scanning all $n$elements for partitioning, the total time complexity becomes $O(n\log\ n)$.

The average-case time complexity of Quicksort is also $O(n\log\ n)$, which is one of its most appealing characteristics. On average, even though the pivot may not always divide the array perfectly, it is likely to produce reasonably balanced partitions over many recursive calls. Mathematically, this can be explained by the fact that the recursive splitting of the array, combined with the linear cost of partitioning at each level, results in a geometric series that sums up to $O(n\log\ n)$.

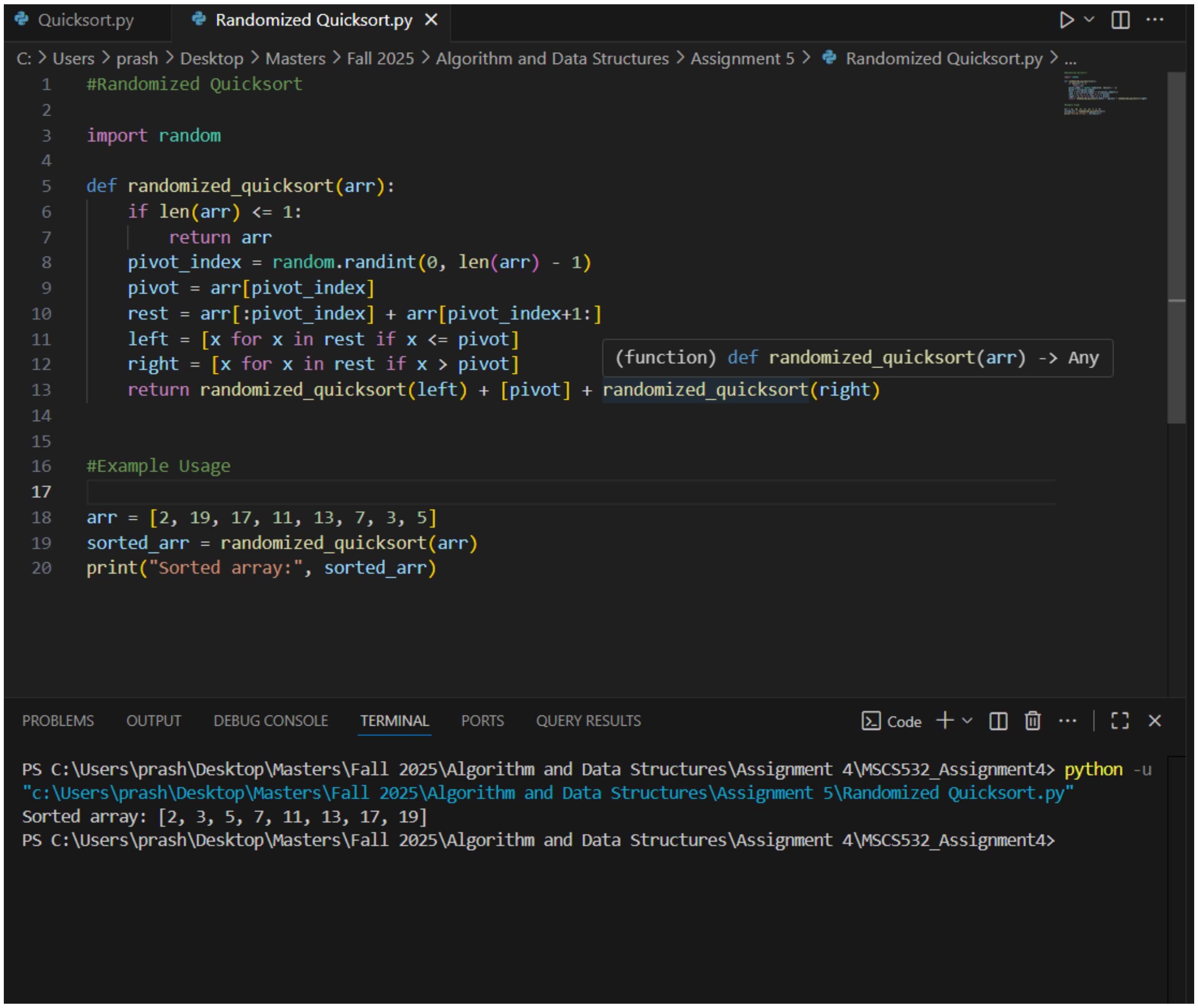
In contrast, the worst-case time complexity of Quicksort is $O(n^2)$. It occurs when the pivot consistently produces highly unbalanced partitions. For example, if the array is already sorted and the last element is always chosen as the pivot, one subarray will always be empty, and the other will contain the rest of the elements. This leads to a recursive depth of n, and each level still requires $O(n)$operations to partition, resulting in quadratic time complexity.

Regarding space complexity, Quicksort is generally efficient. In the average and best cases, the recursion stack grows to a depth of $O(\log\ n)$, since the array is halved at each step. However, in the worst case, the depth of recursion can become $O(n)$, leading to higher memory usage. It is also worth noting that while this implementation uses additional lists for partitioning, optimized in-place versions of Quicksort are commonly used in practice to reduce memory overhead.

Randomized Quicksort

To mitigate all the risks of encountering such worst-case scenario, a randomized version of Quicksort can be deployed. In this approach, the pivot is chosen randomly from the current subarray instead of deterministically selecting the last element. This randomization significantly reduces the likelihood of consistently poor partitions, even if the input data is sorted or contains repeated patterns.

The implementation of randomized Quicksort begins by checking if the array length is less than or equal to one, in which case the array is returned as it is already sorted. Otherwise, a pivot index is chosen randomly, and the corresponding element is used as the pivot. The pivot is removed from the array, and the remaining elements are partitioned into two subarrays based on their comparison with the pivot. These subarrays are recursively sorted, and the final result is obtained by concatenating the sorted left subarray, the pivot, and the sorted right subarray.


Quicksort.py
Randomized Quicksort.py
C: > Users > prash > Desktop > Masters > Fall 2025 > Algorithm and Data Structures > Assignment 5 > Randomized Quicksort.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
Code

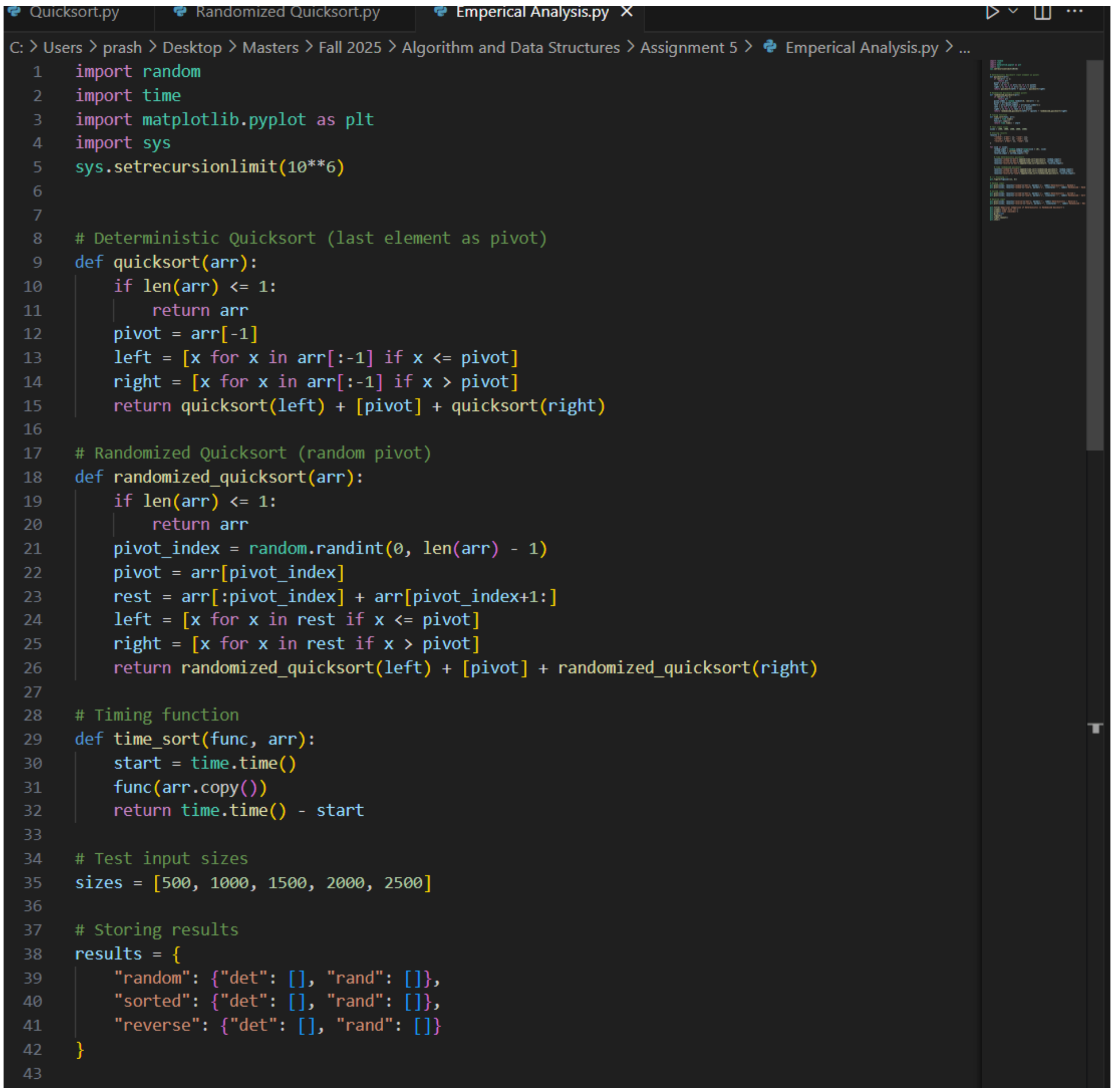
```python
#Randomized Quicksort

import random

def randomized_quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot_index = random.randint(0, len(arr) - 1)
    pivot = arr[pivot_index]
    rest = arr[:pivot_index] + arr[pivot_index+1:]
    left = [x for x in rest if x <= pivot]
    right = [x for x in rest if x > pivot]
    return randomized_quicksort(left) + [pivot] + randomized_quicksort(right)


#Example Usage

arr = [2, 19, 17, 11, 13, 7, 3, 5]
sorted_arr = randomized_quicksort(arr)
print("Sorted array:", sorted_arr)
```

```
(function) def randomized_quicksort(arr) -> Any
```

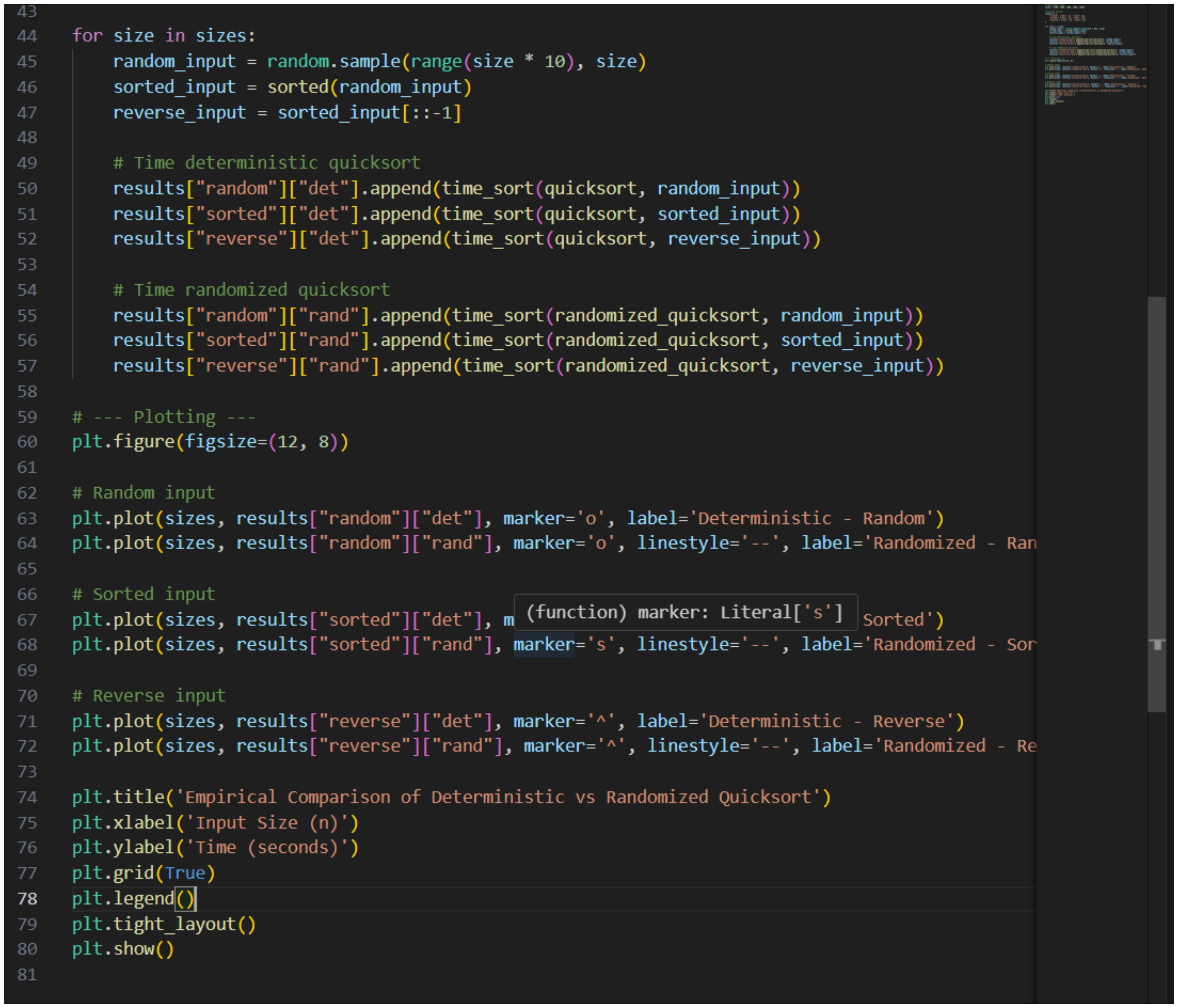
```
PS C:\Users\prash\Desktop\Masters\Fall 2025\Algorithm and Data Structures\Assignment 4\MSCS532_Assignment4> python -u "c:\Users\prash\Desktop\Masters\Fall 2025\Algorithm and Data Structures\Assignment 5\Randomized Quicksort.py"
Sorted array: [2, 3, 5, 7, 11, 13, 17, 19]
PS C:\Users\prash\Desktop\Masters\Fall 2025\Algorithm and Data Structures\Assignment 4\MSCS532_Assignment4>
```

By randomizing the pivot selection, this version of Quicksort ensures that even adversarial input patterns do not degrade its performance. While the theoretical worst-case time complexity remains $O(n^2)$, the probability of such a scenario occurring becomes extremely low, making randomized Quicksort highly effective in practice.

Empirical Analysis


Quicksort.py
Randomized Quicksort.py
Emperical Analysis.py
C: > Users > prash > Desktop > Masters > Fall 2025 > Algorithm and Data Structures > Assignment 5 > Emperical Analysis.py > ...

```python
import random
import time
import matplotlib.pyplot as plt
import sys
sys.setrecursionlimit(10**6)


# Deterministic Quicksort (last element as pivot)
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[-1]
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]
    return quicksort(left) + [pivot] + quicksort(right)

# Randomized Quicksort (random pivot)
def randomized_quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot_index = random.randint(0, len(arr) - 1)
    pivot = arr[pivot_index]
    rest = arr[:pivot_index] + arr[pivot_index+1:]
    left = [x for x in rest if x <= pivot]
    right = [x for x in rest if x > pivot]
    return randomized_quicksort(left) + [pivot] + randomized_quicksort(right)

# Timing function
def time_sort(func, arr):
    start = time.time()
    func(arr.copy())
    return time.time() - start

# Test input sizes
sizes = [500, 1000, 1500, 2000, 2500]

# Storing results
results = {
    "random": {"det": [], "rand": []},
    "sorted": {"det": [], "rand": []},
    "reverse": {"det": [], "rand": []}
}

```



```
for size in sizes:
    random_input = random.sample(range(size * 10), size)
    sorted_input = sorted(random_input)
    reverse_input = sorted_input[::-1]

    # Time deterministic quicksort
    results["random"]["det"].append(time_sort(quicksort, random_input))
    results["sorted"]["det"].append(time_sort(quicksort, sorted_input))
    results["reverse"]["det"].append(time_sort(quicksort, reverse_input))

    # Time randomized quicksort
    results["random"]["rand"].append(time_sort(randomized_quicksort, random_input))
    results["sorted"]["rand"].append(time_sort(randomized_quicksort, sorted_input))
    results["reverse"]["rand"].append(time_sort(randomized_quicksort, reverse_input))

# --- Plotting ---
plt.figure(figsize=(12, 8))

# Random input
plt.plot(sizes, results["random"]["det"], marker='o', label='Deterministic - Random')
plt.plot(sizes, results["random"]["rand"], marker='o', linestyle='--', label='Randomized - Ran

# Sorted input
plt.plot(sizes, results["sorted"]["det"], m (function) marker: Literal['s'] Sorted')
plt.plot(sizes, results["sorted"]["rand"], marker='s', linestyle='--', label='Randomized - Sor

# Reverse input
plt.plot(sizes, results["reverse"]["det"], marker='^', label='Deterministic - Reverse')
plt.plot(sizes, results["reverse"]["rand"], marker='^', linestyle='--', label='Randomized - Re

plt.title('Empirical Comparison of Deterministic vs Randomized Quicksort')
plt.xlabel('Input Size (n)')
plt.ylabel('Time (seconds)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

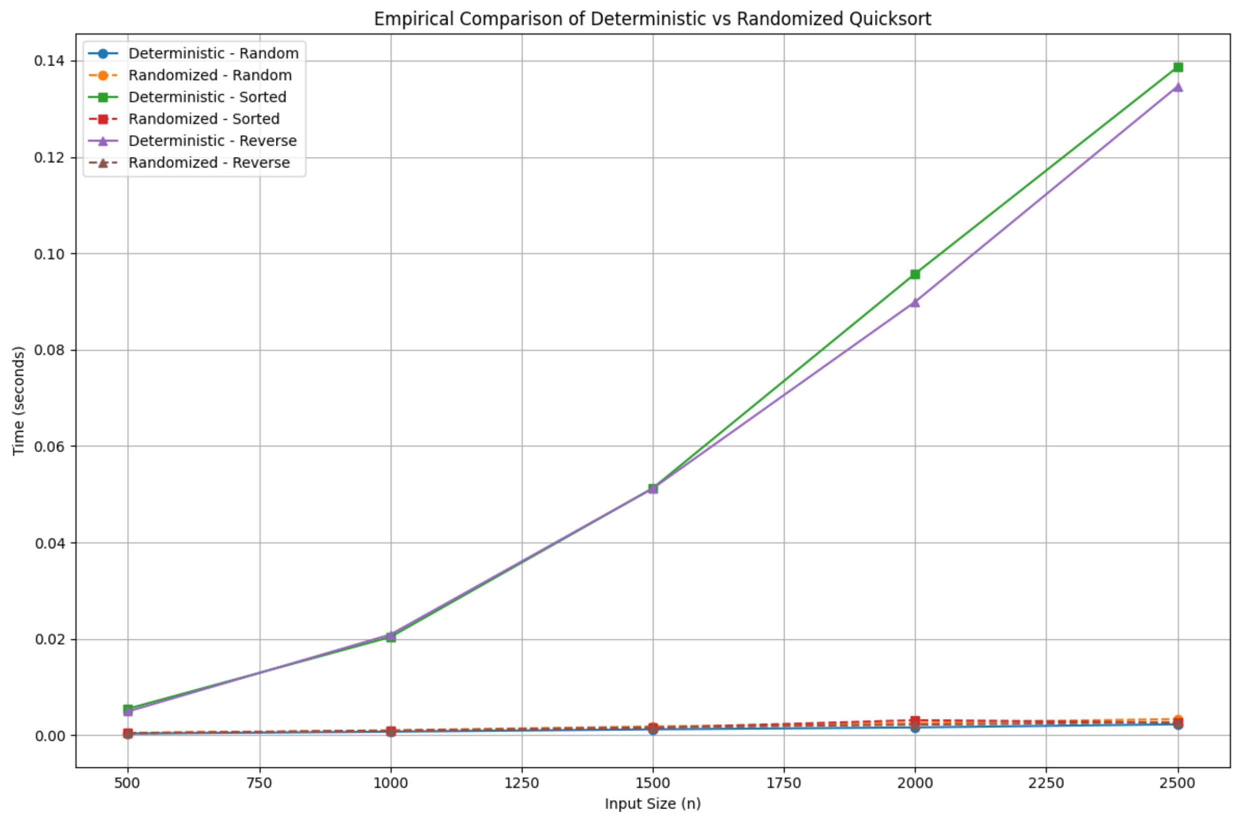
This script empirically compares the performance of two Quicksort variants: deterministic (pivot is always the last element) and randomized (pivot chosen randomly). It tests both algorithms on three types of input arrays—randomly shuffled, sorted in ascending order, and sorted in descending order—across multiple input sizes. For each test, it measures the time taken by each sorting method, stores the results, and then plots the running times on a graph. This allows to visualize and compare how both algorithms perform on different data distributions and input sizes.

Output

The empirical results, as shown below, clearly demonstrate the differences in performance between the deterministic and randomized versions of Quicksort across various input distributions. On randomly ordered inputs, both algorithms performed similarly, with their running times closely aligned. This supports the idea that both versions have an average-case time complexity of $O(n\log\ n)$when partitions are reasonably balanced.

However, on sorted and reverse-sorted inputs, the deterministic version of Quicksort exhibited a dramatic increase in execution time, especially as the input size grew. This behavior is consistent with its worst-case time complexity of $O(n^2)$, which arises due to consistently unbalanced partitions when the pivot is always chosen as the last element. On the other hand, the randomized version maintained near-linearithmic growth in all input types, including sorted and reverse-sorted arrays. This shows how random pivot selection avoids repeated poor partitions and thus mitigates the likelihood of worst-case performance.

Overall, the results reinforce the advantages of randomized Quicksort. While both algorithms are efficient on average, the randomized version is significantly more robust and reliable, especially in scenarios where the input order is unknown or adversarial.



Conclusion

In conclusion, Quicksort remains one of the most efficient and widely used sorting algorithms due to its average-case performance and simple recursive structure. The deterministic version, while straightforward to implement, can suffer from poor performance when applied to already sorted or similarly structured data due to its fixed pivot selection strategy. On the other hand, randomized Quicksort introduces a simple yet powerful modification by randomizing the pivot selection, which significantly reduces the chance of encountering worst-case performance. Theoretical analysis demonstrates that both versions achieve an average-case time complexity of $O(n\log\ n)$, but randomized Quicksort offers better protection against unbalanced partitions. Empirical tests further confirm that randomized Quicksort consistently performs well across different input distributions, making it a preferred choice for general-purpose sorting tasks.

Ultimately, randomization introduces minimal overhead while providing substantial improvements in reliability, particularly when dealing with large and unpredictable datasets.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

GeeksforGeeks. (n.d.). *QuickSort*. Retrieved from https://www.geeksforgeeks.org/quick-sort/

Python Software Foundation. (n.d.). *random — Generate pseudo-random numbers*. Retrieved from https://docs.python.org/3/library/random.html