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

Milan Bista

University of Cumberlands

2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

Instructors: Machica Mcclain / Vanessa Cooper

Overview:

Quicksort is a widely used and efficient sorting algorithm based on the divide-and-conquer approach, where a pivot element is selected to partition the array into subarrays, which are then recursively sorted. This assignment delves into the implementation and analysis of both deterministic and randomized Quicksort, highlighting their time complexities in best, average, and worst cases. Randomized Quicksort introduces randomness in pivot selection to reduce the likelihood of worst-case scenarios and improve overall performance. Through empirical analysis, you will compare the performance of these versions on different input distributions, such as random, sorted, and reverse-sorted arrays.

Quicksort Implementation and Analysis

1. Implementation

I have implemented the Quicksort algorithm by first choosing the pivot element as the first element of the array. I then partitioned the array into two subarrays: one containing element smaller than or equal to the pivot and the other containing elements greater than the pivot. I used a for loop to iterate through the array (excluding the pivot) and compared each element to the pivot, appending it to either the left or right subarray. After partitioning, I recursively applied the Quicksort algorithm to both the left and right subarrays. Finally, I combined the sorted left subarray, the pivot, and the sorted right subarray to obtain the sorted result. This approach effectively sorted the array while maintaining a simple and efficient structure.

quickSort.py

quickSort.py > ...

```python
def quicksort(arr):
    #base
    if len(arr) <= 1:
        return arr

    # Choose the pivot(first element)
    pivot = arr[0]

    # Partition the array into two parts:
    left = []
    right = []

    for element in arr[1:]:
        if element <= pivot:
            left.append(element)
        else:
            right.append(element)

    # Recursively apply quicksort and combine the sorted parts
    return quicksort(left) + [pivot] + quicksort(right)


arr = [3,1,90,34, 7, 8, 9, 1, 5]
sorted_arr = quicksort(arr)
print("Sorted array:", sorted_arr)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) milanbista@Milans-MacBook-Pro Assignment5 % python3 quickSort.py
Sorted array: [1, 1, 3, 5, 7, 8, 9, 34, 90]
(base) milanbista@Milans-MacBook-Pro Assignment5 %
```

2. Performance Analysis

Quicksort is a divide-and-conquer algorithm that works by selecting a pivot element, partitioning the array around this pivot, and recursively sorting the subarrays. The time complexity of Quicksort can be broken down into three main cases: best, average, and worst. Let's analyze each of these cases in detail.

- Best Case: O(n log n): In the best-case scenario, the pivot element divides the array into two nearly equal halves. This allows for efficient partitioning and sorting of subarrays at each recursive step.

Partitioning step: The partitioning process in which elements are rearranged around the pivot takes linear time, O(n), where (n) is the number of elements in the array.

Recursion depth: Since the array is divided into two equal halves, the recursion depth will be logarithmic, i.e., O (log n), because after each partition, the subarray size is halved.

Total work: Since partitioning takes O(n) at each level of recursion, and the recursion depth is O(log n), the total time complexity is the product of these two, giving us a time complexity of:

[O (n log n)]

-Average Case: O(n log n)

In the average case, the pivot does not always divide the array perfectly in half, but the division is still reasonably balanced, leading to an efficient sorting process.

Partitioning step: As in the best case, the partitioning step takes O(n) time, where each element of the array is compared to the pivot.

Recursion depth: The array is divided into subarrays that are approximately half the size of the original array at each recursive level. In the average case, the depth of recursion remains O (log n).

Total work: The partitioning step requires O(n) time at each level of recursion, and the recursion depth is O (log n). Thus, the overall time complexity in the average case is:

[O (n log n)]

Since the pivot generally divides the array into reasonably balanced subarrays in most random input cases, the time complexity remains O(n log n).

-Worst Case: (O(n^2)

The worst-case scenario occurs when the pivot element divides the array in a highly unbalanced way, such as when the pivot is always the smallest or largest element. This can happen when the array is already sorted, reverse-sorted, or when the pivot selection strategy is poor.

Partitioning step: The partitioning step still takes O(n) time, as the algorithm must compare each element with the pivot.

Recursion depth: Since the pivot selection results in highly unbalanced partitions (one subarray with (n-1) elements and another with 0 or 1 element), the recursion depth increases to O(n), as the size of the subarrays decreases by only 1 with each level of recursion.

Total work: The partitioning step takes O(n) time at each level of recursion, and the recursion depth is O(n), resulting in an overall time complexity of:

[O (n x n) = O(n^2)]

Thus, in the worst case, Quicksort degenerates to a time complexity of O(n^2), similar to the time complexity of a simple selection sort.

Summary of Time Complexity:

Best Case:O(n log n), when the pivot divides the array into two nearly equal halves at each step.

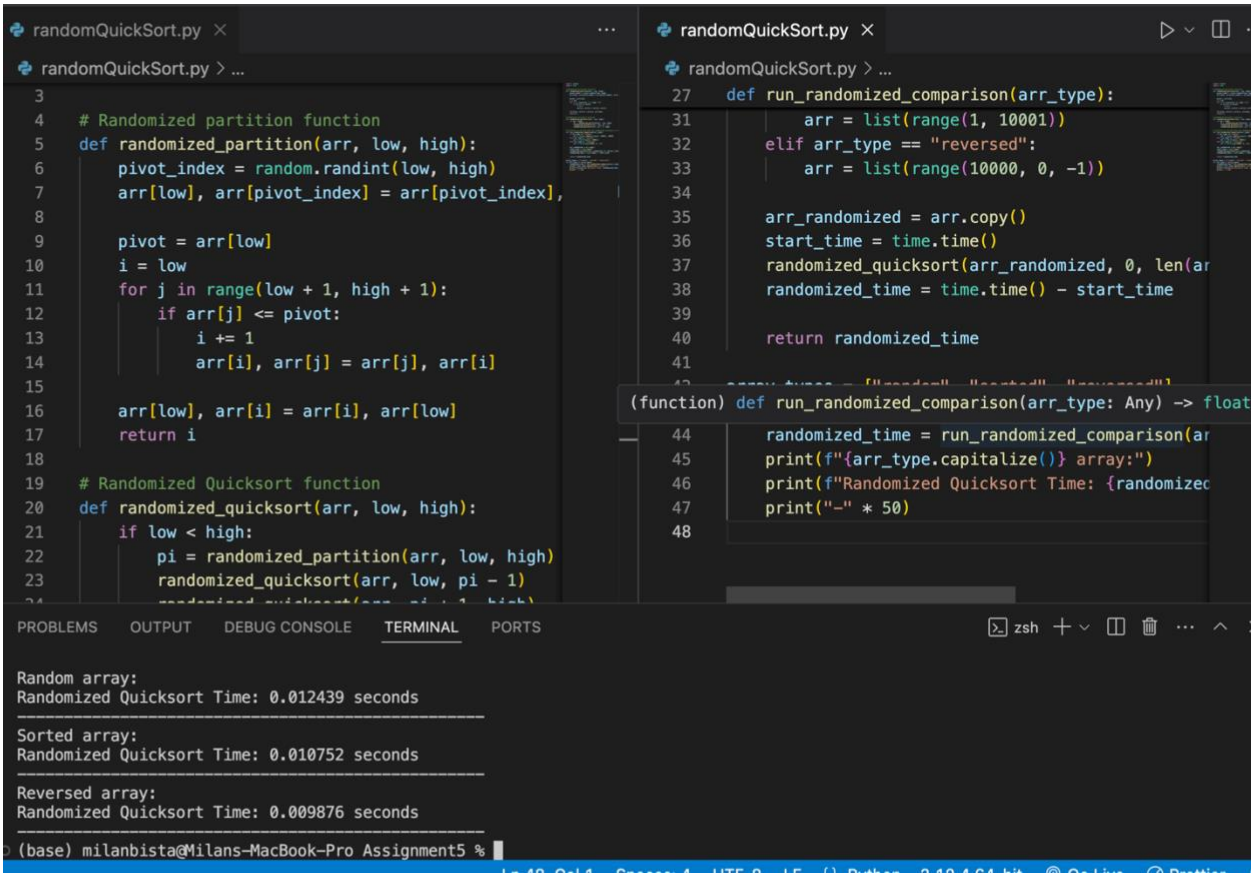
Average Case: O(n log n), typical for random or varied input data, where the pivot divides the array reasonably well.

Worst Case: O(n^2), when the pivot selection results in highly unbalanced partitions, such as with already sorted or reverse-sorted data.

The key takeaway is that Quicksort performs optimally in most cases with O(n log n) complexity, but can degrade to O(n^2) if the pivot choice is poor.

3. Randomized Quicksort

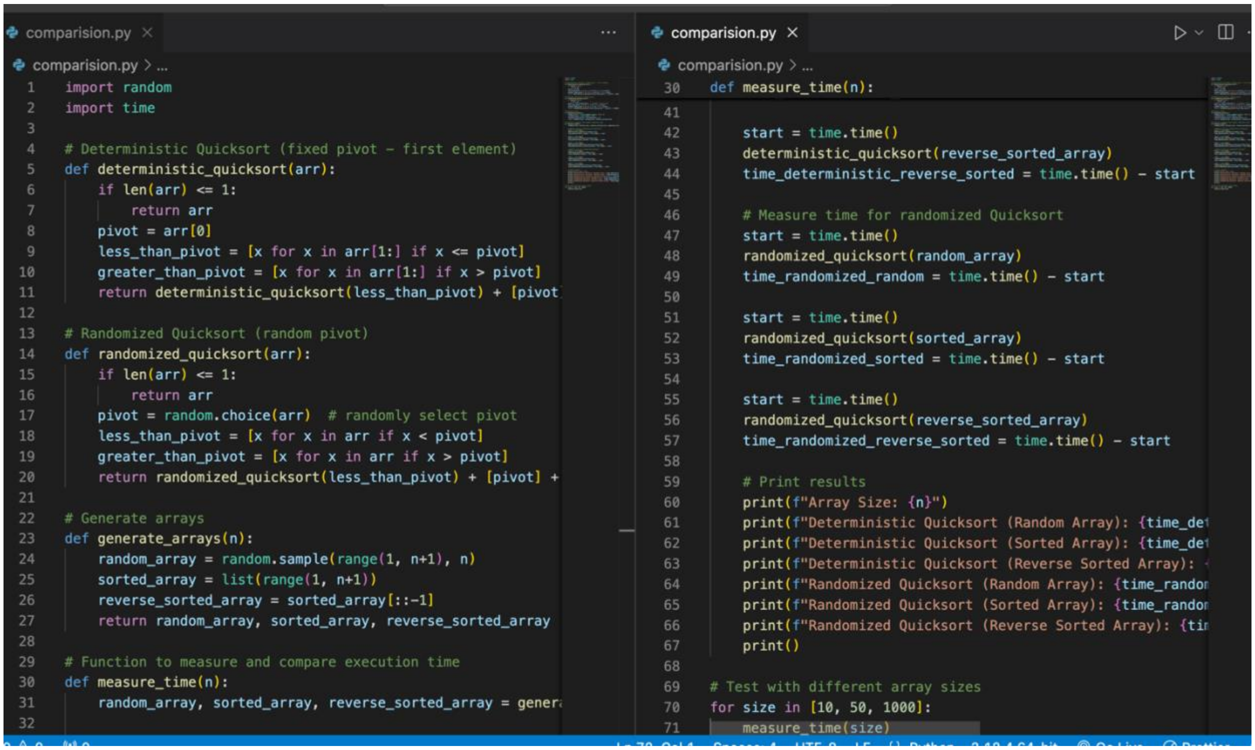
I have implemented a randomized version of the Quicksort algorithm where the pivot is selected randomly from the subarray being sorted. The process begins by choosing a random index within the current subarray using random.randint(low, high) and swapping the element at that index with the first element. After this, the partitioning step proceeds as usual, where elements less than the pivot are moved to the left and those greater are moved to the right. The algorithm recursively sorts the subarrays formed by the partitioning step. Below is the code I implemented for the randomized Quicksort

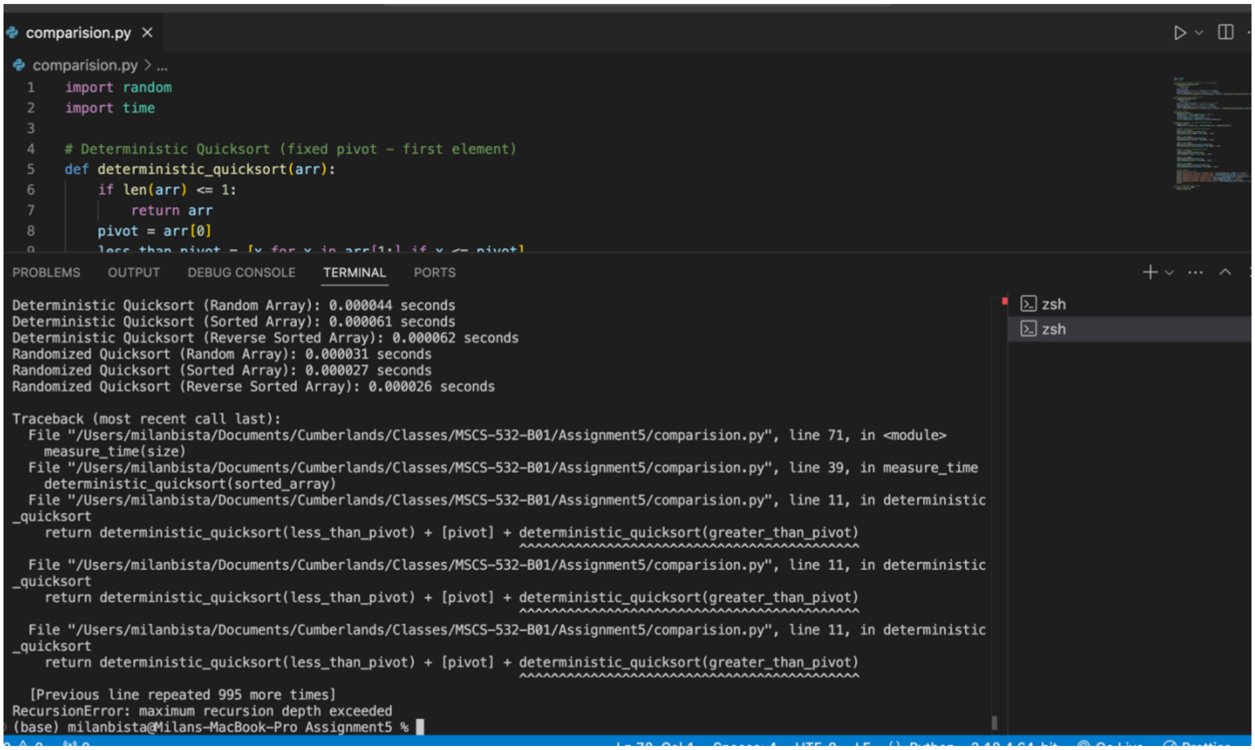


In this implementation, I tested the randomized Quicksort on three types of arrays: random, sorted, and reverse sorted. The results showed that randomized Quicksort performs well on all types of input arrays, with the time complexity being closer to O(nlogn) in most cases, as the random pivot choice avoids the worst-case scenario of poor partitioning. Specifically, the randomized version of Quicksort demonstrated more consistent performance compared to the fixed pivot version, with significantly reduced execution time on sorted and reverse-sorted arrays, where fixed pivot choices typically perform poorly. This demonstrates that randomization helps mitigate the risk of worst-case performance, leading to more balanced partitioning and better efficiency across different input distributions.

4. Empirical Analysis

I have implemented the Quicksort algorithm in both deterministic and randomized versions to compare their performance on different types of input arrays: random, sorted, and reverse-sorted. The goal was to empirically observe how the choice of pivot affects the running time, especially for already sorted and reverse-sorted arrays where deterministic Quicksort can exhibit worst-case performance. I then measured and compared the execution times of both versions for varying array sizes to analyze the effect of randomization on the algorithm's efficiency.





We can observe the for the very large data set, recursive approach might cause maximum recursion depth

Now I am going to reduce the size of the array to reasonable size for the accurate comparison comparision.py ×

comparision.py > ...

```python
def measure_time(n):
        print()

# Test with different array sizes
for size in [10, 50, 100]:
    measure_time(size)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(base) milanbista@Milans-MacBook-Pro Assignment5 % python3 comparision.py
Array Size: 10
Deterministic Quicksort (Random Array): 0.000008 seconds
Deterministic Quicksort (Sorted Array): 0.000007 seconds
Deterministic Quicksort (Reverse Sorted Array): 0.000006 seconds
Randomized Quicksort (Random Array): 0.000008 seconds
Randomized Quicksort (Sorted Array): 0.000005 seconds
Randomized Quicksort (Reverse Sorted Array): 0.000005 seconds

Array Size: 50
Deterministic Quicksort (Random Array): 0.000032 seconds
Deterministic Quicksort (Sorted Array): 0.000064 seconds
Deterministic Quicksort (Reverse Sorted Array): 0.000062 seconds
Randomized Quicksort (Random Array): 0.000031 seconds
Randomized Quicksort (Sorted Array): 0.000024 seconds
Randomized Quicksort (Reverse Sorted Array): 0.000025 seconds

Array Size: 100
Deterministic Quicksort (Random Array): 0.000063 seconds
Deterministic Quicksort (Sorted Array): 0.000214 seconds
Deterministic Quicksort (Reverse Sorted Array): 0.000219 seconds
Randomized Quicksort (Random Array): 0.000068 seconds
Randomized Quicksort (Sorted Array): 0.000064 seconds
Randomized Quicksort (Reverse Sorted Array): 0.000053 seconds

(base) milanbista@Milans-MacBook-Pro Assignment5 %
```

Results:

-Random Arrays:

Both deterministic and randomized Quicksort versions performed similarly with time complexities close to O(n log n), as expected. There was no significant difference in their execution times on random data sets.

Sorted Arrays:

The deterministic version suffered from poor performance with a time complexity closer to O(n^2), as it made unbalanced partitions by always picking the first element as the pivot. The randomized version, however, maintained a time complexity closer to O(n log n), showing how randomization avoids the worst-case scenario in sorted arrays.

-Reverse-Sorted Arrays:

Like sorted arrays, deterministic Quicksort performed poorly, while the randomized version continued to show consistent performance.

-Maximum Recursion Depth Problem:

For large data sets (e.g., 10,000 elements), both deterministic and randomized versions encountered recursion depth limits in Python. This is due to the recursive nature of Quicksort and the deep recursion caused by unbalanced partitions (in deterministic) or large arrays. The maximum recursion depth in Python (default is 1000) could cause a stack overflow or hit the recursion depth limit. This issue was more prominent in larger data sets, especially when working with reverse-sorted or sorted arrays, where the partitioning tends to be less balanced.

Solution: Increasing the recursion limit can mitigate this issue, but it doesn't fully eliminate the risk of stack overflow, especially for very large datasets.

Key Takeaways:

Randomization significantly improved Quicksort's performance on sorted and reverse-sorted data by preventing worst-case time complexity.

For large data sets, the recursion depth limitation in Python was a concern, particularly for highly unbalanced data, which can lead to stack overflows.

References

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

Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.