9/22/2024

Assignment 5

Quicksort Algorithm: Implementation, Analysis and Randomization

Sujan Dumaru

Student ID: 005030060

MSCS 532: Algorithms and Data Structures

Dr. Vanessa Cooper

INTRODUCTION

Quicksort is a widely used comparison-based sorting algorithm. It follows the divide-and-conquer paradigm that works by selecting a pivot element from the array and partitioning the other elements into two subarrays and then recursively sorted. This algorithm is used to sort large datasets in memory-constrained environments because of its speed and minimal memory usage. The efficiency of Quicksort depends heavily on the pivot selection strategy, which affects the time complexity of the algorithm. In this assignment, we explore both the deterministic and randomized versions of Quicksort, analyzing their performance across different scenarios.

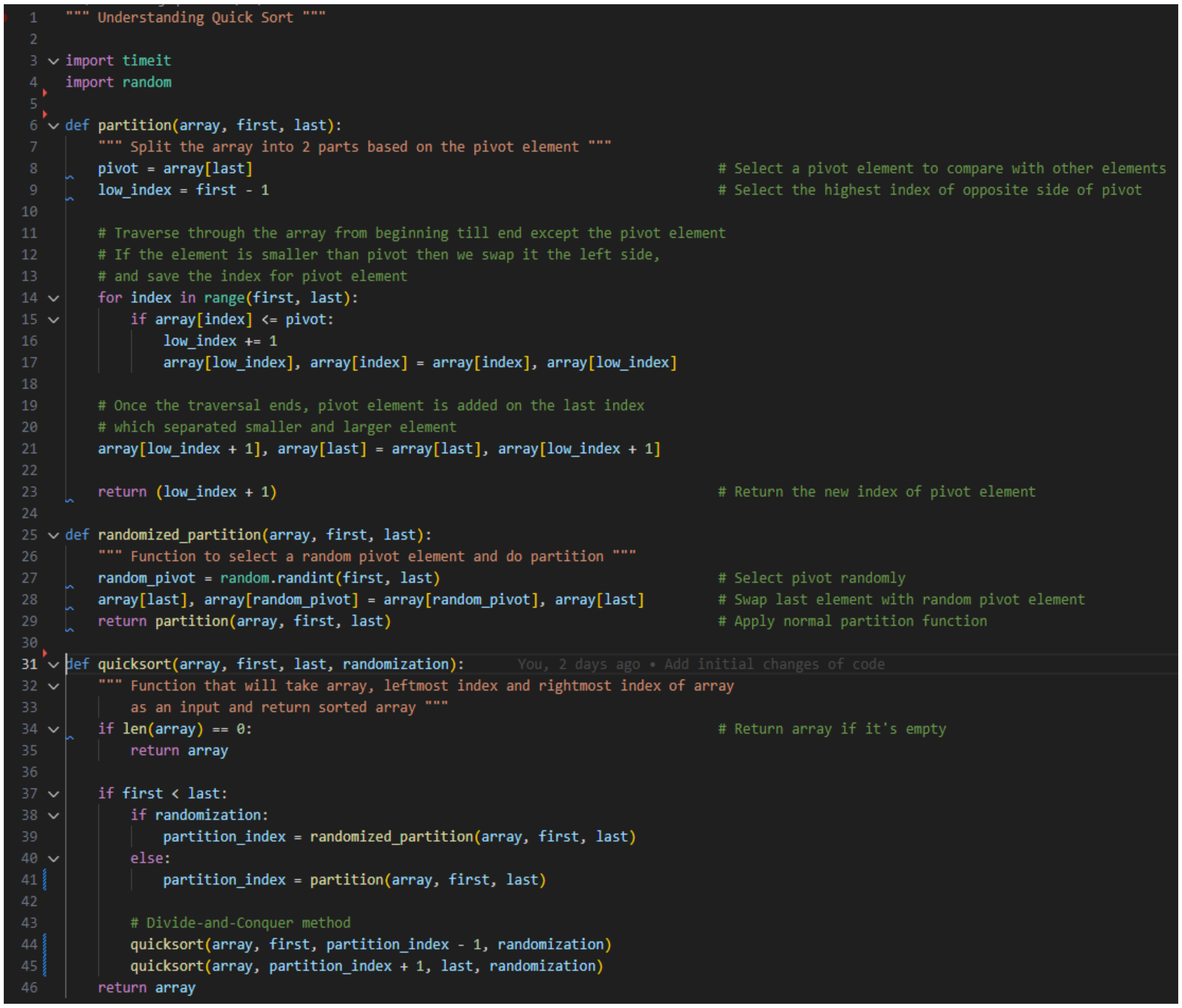
IMPLEMENTATION

1. Deterministic Quicksort:

In the deterministic version of Quicksort, the pivot is chosen as the last element or first element of the subarray. The array is then partitioned such that elements less than the pivot are on its left, and elements greater than pivot are on its right. The algorithm then recursively sorts the two subarrays.

2. Randomized Quicksort:

In the randomized version of Quicksort, the pivot is randomly selected from the subarray. Randomized Quicksort is an enhanced version of the Deterministic Quicksort that improves worst case performance. This random selection reduces the risk of encountering consistently poor pivot choices, which can lead to the worst-case time complexity of $O(n^2)$, especially with already sorted or reverse-sorted data.



```python
""" Understanding Quick Sort """

import timeit
import random

def partition(array, first, last):
    """ Split the array into 2 parts based on the pivot element """
    pivot = array[last]                                                  # Select a pivot element to compare with other elements
    low_index = first - 1                                                # Select the highest index of opposite side of pivot

    # Traverse through the array from beginning till end except the pivot element
    # If the element is smaller than pivot then we swap it the left side,
    # and save the index for pivot element
    for index in range(first, last):
        if array[index] <= pivot:
            low_index += 1
            array[low_index], array[index] = array[index], array[low_index]

    # Once the traversal ends, pivot element is added on the last index
    # which separated smaller and larger element
    array[low_index + 1], array[last] = array[last], array[low_index + 1]

    return (low_index + 1)                                               # Return the new index of pivot element

def randomized_partition(array, first, last):
    """ Function to select a random pivot element and do partition """
    random_pivot = random.randint(first, last)                           # Select pivot randomly
    array[last], array[random_pivot] = array[random_pivot], array[last]  # Swap last element with random pivot element
    return partition(array, first, last)                                 # Apply normal partition function

def quicksort(array, first, last, randomization):      You, 2 days ago • Add initial changes of code
    """ Function that will take array, leftmost index and rightmost index of array
        as an input and return sorted array """
    if len(array) == 0:                                                  # Return array if it's empty
        return array

    if first < last:
        if randomization:
            partition_index = randomized_partition(array, first, last)
        else:
            partition_index = partition(array, first, last)

        # Divide-and-Conquer method
        quicksort(array, first, partition_index - 1, randomization)
        quicksort(array, partition_index + 1, last, randomization)
    return array
```

Figure 1: Deterministic and Randomized Quicksort.

The code snippets above implement both the Deterministic and Randomized versions of Quicksort. The **quicksort()** function includes a parameter called *randomization* , which determines the type of Quicksort to be executed. If *randomization* is set to **True**, **randomized_partition()** will be called in which the pivot element is chosen randomly from the array, resulting in a Randomized Quicksort. Conversely, if *randomization* is set to **False**, **partition()** will be called in which the pivot is selected as the last element of the array, following the Deterministic Quicksort approach.

PERFORMANCE ANALYSIS

1. Time Complexity

Like I mentioned before, the time complexity of the Quicksort heavily depends on the pivot selection. Poor pivot element selection can make Quick Sort ineffective against other sorting algorithms. The time complexity of Quick Sort can be expressed by the recurrence relation:

$T(n) = T(k) + T(n\text{-}k\text{-}1) + O(n)$, where k is the number of elements less than the pivot.

- **Best Case:**

  When the pivot divides the array into two equal parts, the number of elements is equal on both sides. So, the recurrence relation comes out to be:

  $$T(n) = 2T(n/2) + O(n)$$

  If we apply master theorem in this, we can see that $log_a b = log_2 2 = 1$. Since, $n^{log_b a} = n^1$ and $n^d = n^1$, this falls into Case 2 of Master Theorem which is $n^{log_b a} = n^d$ resulting in the time complexity for the best case scenario to be $\Omega\ (n\ logn)$.

- **Worst Case:**

  When the pivot consistently corresponds to either the smallest or largest element, it results in highly unbalanced partitions, with one subarray containing $n-1$ elements each time. Consequently, the time complexity at each level of the recursion is $O\ (n)$. Given that the recursion tree will have n levels due to these unbalanced partitions, the overall time complexity for the worst-case scenario is $O\ (n)$ per level multiplied by n levels, resulting in a total time complexity of $O\ (n * n) = O\ (n^2)$.

- **Average Case:**

When pivot selection is random, the average-case time complexity can be determined using the Master Theorem, just as in the best-case scenario. This analysis reveals that the average-case time complexity is *Θ (n logn)*.

Randomized Quicksort can generally ensure balanced partitions, but it will not avoid the worst-case scenario completely. The average case arises when the pivot splits the array in such a way that both subarrays are of roughly equal size. Each level of recursion processes a fraction of the array, and there are *O (log n)* levels in total. At each level, the partitioning step takes *O(n)* time, leading to an overall time complexity of *O (n logn).* Quicksort is widely used in practice, particularly in situations where average-case speed and memory efficiency are crucial.

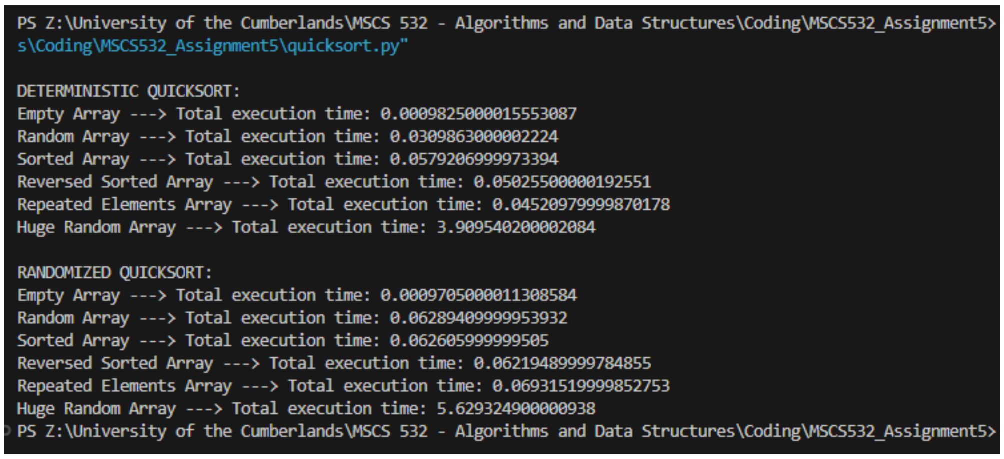
2. Space Complexity

The space complexity of Quicksort primarily arises from the recursive calls. In the best and average cases, the recursive depth is *O (logn)*, resulting in a space complexity of *O (logn).* However, in the worst case, the recursion depth can reach *O(n)*, resulting in a space complexity of *O(n).*

EMPIRICAL ANALYSIS

I tested both the deterministic and randomized versions of Quicksort on input arrays:

- Empty array – No elements in the array.
- Random array – Elements are randomly ordered.
- Sorted array – Elements are already sorted in ascending order.
- Reverse-sorted array – Elements are sorted in descending order.
- Repeated elements array – Presence of same elements in the array
- Huge random array – Large number of randomly ordered elements.



```
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding\MSCS532_Assignment5>
s\Coding\MSCS532_Assignment5\quicksort.py"

DETERMINISTIC QUICKSORT:
Empty Array ---> Total execution time: 0.0009825000015553087
Random Array ---> Total execution time: 0.0309863000002224
Sorted Array ---> Total execution time: 0.0579206999973394
Reversed Sorted Array ---> Total execution time: 0.05025500000192551
Repeated Elements Array ---> Total execution time: 0.04520979999870178
Huge Random Array ---> Total execution time: 3.909540200002084

RANDOMIZED QUICKSORT:
Empty Array ---> Total execution time: 0.0009705000011308584
Random Array ---> Total execution time: 0.06289409999953932
Sorted Array ---> Total execution time: 0.062605999999505
Reversed Sorted Array ---> Total execution time: 0.06219489999784855
Repeated Elements Array ---> Total execution time: 0.06931519999852753
Huge Random Array ---> Total execution time: 5.629324900000938
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding\MSCS532_Assignment5>
```

Figure 2: Output result for quicksort on various inputs.

The output demonstrates the total execution time of both Quicksort versions across different input types. Randomized Quicksort shows more consistent performance compared to Deterministic Quicksort. This consistency stems from the fact that Randomized Quicksort generally achieves an average time complexity of *O (n logn)*, regardless of the input. In contrast, Deterministic Quicksort's execution time can vary significantly depending on the input, especially with sorted or random arrays. By using a random pivot selection, Randomized Quicksort minimizes the likelihood of worst-case scenarios, resulting in more predictable and reliable performance.

CONCLUSION

In this assignment, we implemented both deterministic and randomized versions of the Quicksort algorithm and analyzed their performance. The deterministic version can encounter worst-case scenarios with sorted or reverse-sorted inputs, leading to $O(n^2)$ time complexity. However, randomization significantly improves the average performance by reducing the likelihood of poor pivot selection, making randomized Quicksort more robust across different input types. Hence, the Quicksort algorithm can be used in real-world scenarios where efficient sorting of large datasets is required.

REFERENCES

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509