11/15/2024

Assignment: 5

Quicksort Algorithm: Implementation, Analysis, and Randomization

Name: Rutu Shah
STUDENT ID: 005026421

Table of Contents

1 .1 Quicksort Implementation and Analysis

Quick Sort is one of the most efficient sorting algorithms based on the divide-and-conquer approach. It picks an element called a pivot, partitions the array around the chosen pivot, and then recursively sorts subarrays created until the whole array is sorted. In practice, it often outperforms many other sorting algorithms.

It is important to understand how Quick Sort works, time complexity, and its optimization techniques. Though it has an average case time complexity O (n log n), the worst case of quick sort and sensitivity to the selection of the pivot are of considerable interest in the implementation and analysis of the algorithm.

1.1.1 Implementation

GitHub URL: https://github.com/rutus-code/MSCS532_Assignment5/blob/main/quickSort.py

Quick sort implementation contains several key features and here is the explanation of how it works

Pivot Selection: Uses the last element as the pivot for simplicity; implementation of other strategies, such as median-of-three, would be simple and would yield better performance for nearly sorted arrays.

1. The last element in the array is selected as the pivot.

Partitioning: The partition function rearranges elements around the pivot without using extra space. It works in the below mentioned steps

2. The array is split into two different subarrays:
3. Elements which are less than or equal to the pivot.
4. correct: Components that exceed the pivot.

Recursion: The algorithm keeps on recursively sorting smaller subarrays until the entire array is sorted.

Combined Results:

Then, this sorted left subarray, the pivot, and the sorted right subarray merge into the sorted array.

2.1 Performance Analysis

Time Complexity for Quick Sort algorithm:

2.2.1 Best Case O (n log n)

The pivot divides the array into two nearly equal parts at each recursive step.

Analysis:

- Each level of recursion processes n elements in the partitioning step.
- The depth of the recursion tree is roughly logn-this is the number of times that an array can be halved.
- Total no of comparisons: n+n/2+n/4+⋯=O(nlogn).

2.2.2 Average Case O (n log n)):

In this scenario, the pivot does not always evenly divide the array into halves; rather, it usually achieves a good balance on average.

Analysis:

- On average, the pivot splits the array so that one subarray has about n/4 elements and the other has about 3n/4 elements.
- The recursion tree still has a height of roughly (logn).
- The amount of work is proportional to (nlogn), assuming the small differences in each level don't add up to increase the number of comparisons substantially.

2.2.3 Worst Case O (n^2)):

All such pivoting results in highly unbalanced partitions since the pivot is always the smallest or largest element.

For Example: Sorting an already sorted or reverse-sorted array with a naive pivot selection—for example, always choosing the last element.

Analysis:

- At each recursive step, one subarray contains $n-1$ elements, and the other subarray contains 0 elements.
- The recursion tree has depth n, and at each level, n elements are processed.
- Total number of comparisons: $n+(n-1) +(n-2) +\cdots+1=O (n^2)$.

2.2 Why the average-case time complexity is (O (n log n))

In a randomly ordered array, the pivot element has an equal probability of splitting the array into any ratio. The expected sizes of the two subarrays are directly proportional to the size of the original array. With balanced partitions on average, the recursion depth is (log n), where each level processes n elements.

Let’s prove this, suppose

- Level 1 has n comparisons
- Level 2 has n/2 comparisons in each half = n total
- Level 3 hasn/4 comparisons in each quarter = n total, and so on for log n levels
- Expected comparisons = $T(n)=n+T(\alpha n) +T((1-\alpha)n)$, where α is the expected division ratio.
- Recurrence simplifies to O (n log n).

2.3 Why the worst-case time complexity is (O(n^2))

The pivot divides the array such that one partition contains n−1 elements and the other contains 0. This happens when the pivot is always the smallest or biggest element of the subarray being sorted. The input array is already sorted or reverse-sorted and the pivot is chosen naively—say, it is always the first or last element.

Example:

- Take the list: [1,2,3,4,5] (sorted already).
- If the last element is always chosen as the pivot:
- Pivot = 5: Partition [1,2,3,4] and
- Pivot = 4: Partition into [1,2,3] and
- Pivot is 3: Partition into [1,2] and
- Pivot = 2: Divide into [1] and
- Pivot = 1: No more recursion possible.

At each step we perform n−1 comparisons during partitioning. The height of the recursion tree is n because the size of the subarray decreases by 1 at each step.

Total comparisons: $T(n)=n+(n-1)+(n-2)+\cdots+1$

This is the sum of the first n natural numbers:

$T(n)=n(n+1)/2 = O(n^2)$

Principal Findings

Inefficiency in the partitioning phase is caused by bad pivot selection, resulting in one very large partition and, of course, one empty partition.

Impact: Instead of balanced division (log n depth), the recursion tree becomes something more akin to a linked list with depth n.

Avoiding the Worst Case

- Picking a pivot that partitions the array into balanced sub-arrays will prevent the worst-case scenario from occurring.
- Randomized Quicksort reduces the probability of repeatedly picking a bad pivot.

Input Preprocessing:

Shuffling the array before applying Quicksort can help mitigate issues with already sorted or reverse-sorted input.

2.4 Space complexity and any additional overheads for quick sort algorithm

Its space complexity depends on the recursion depth and whether the algorithm is implemented in-place.

1. Recursive Call Stack O (log n))

- When the pivot divides the array into two nearly equal parts in each recursive step, then the recursion depth is directly proportional to log n.
- Every recursive function call uses stack space to store function arguments, local variables, and return addresses.

Worst Case O(n))

- When the pivot always creates unbalanced partitions for example, all elements on one side and none on the other then the recursion depth becomes n, this leads to a space complexity of O(n) for the call stack.

2. Partitioning Overhead

- Quicksort partitions the array in-place:
- It exchanges elements directly within the array itself to form two individual partitions.
- That saves the extra arrays and makes the partitioning phase space friendly.
- No extra memory is consumed to copy elements, contrary to Merge Sort, which requires O(n) auxiliary space.

3. Additional Overhead

The algorithm requires a couple of additional variables where loop counters to traverse the array. These variables represent little memory overhead, as they are usually O (1).

Randomized Quicksort: The addition of randomization, like the random selection of a pivot, comes at the cost of generating random numbers. However, this does not change the constant overhead, written as O (1) per call, and it does not affect the overall space complexity.

3.1 Randomized Quicksort

Randomized quicksort is a sorting algorithm derived from quicksort that uses randomization in the selection of a pivot element to partition the array. This, in turn, improves the average performance of the algorithm by reducing the likelihood of worst-case scenarios occurring frequently when using deterministic pivot selection. It makes use of principles of probability to raise efficiency and reliability in sorting and is thus one of the important examples showing how randomness can influence algorithm design and performance.

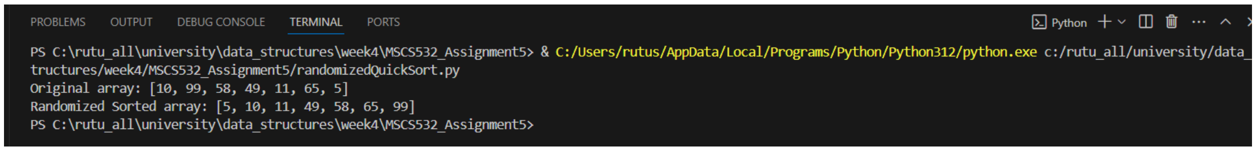
Here, is the implementation of randomized Quick Sort Algorithm:

```
# Created By : Rutu Shah

# Created Date : 14th November 2024

# Code :  Implementation of randomized quick sort algorithm

```

```
import random

def randomizedPartition(arr, low, high):

     #Here we will select the random pivot and swap it with the last element

    pivot_index = random.randint(low, high)

    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]

    # Standard partitioning

    pivot = arr[high]

    #smallest element index

    i = low - 1

    for j in range(low, high):

        if arr[j] <= pivot:

            i += 1

            arr[i], arr[j] = arr[j], arr[i]

   #to place the pivot array

    arr[i + 1], arr[high] = arr[high], arr[i + 1]

    return i + 1

def randomizedQuicksort(arr, low, high):

    if low < high:

        pi = randomizedPartition(arr, low, high)

        randomizedQuicksort(arr, low, pi - 1)

        randomizedQuicksort(arr, pi + 1, high)

# Example usage

arr = [10, 99, 58, 49, 11, 65,5]
```

```python
print("Original array:", arr)

randomizedQuicksort(arr, 0, len(arr) - 1)

print("Randomized Sorted array:", arr)
```

Output:



```
PS C:\rutu_all\university\data_structures\week4\MSCS532_Assignment5> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/data_
tructures/week4/MSCS532_Assignment5/randomizedQuickSort.py
Original array: [10, 99, 58, 49, 11, 65, 5]
Randomized Sorted array: [5, 10, 11, 49, 58, 65, 99]
PS C:\rutu_all\university\data_structures\week4\MSCS532_Assignment5>
```

3.2 How Randomization Affects Quick Sort?

- In standard Quicksort, using the first or last element as a pivot may cause the worst-case time complexity O (n^2) for already sorted or reverse-sorted arrays.
- Randomization ensures that the pivot is chosen in such a way that it is highly unlikely to always choose the smallest or largest element, reducing the probability of the worst case.

Time Complexity: The time complexity of Randomized Quicksort is O (n log n) even for inputs that are adversarial to deterministic pivot selection strategies. More efficient:

Randomization improves the load balancing by increasing the probability of partitioning the array into two nearly equal subarrays, thus reducing the recursion depth and optimizing the cache performance.

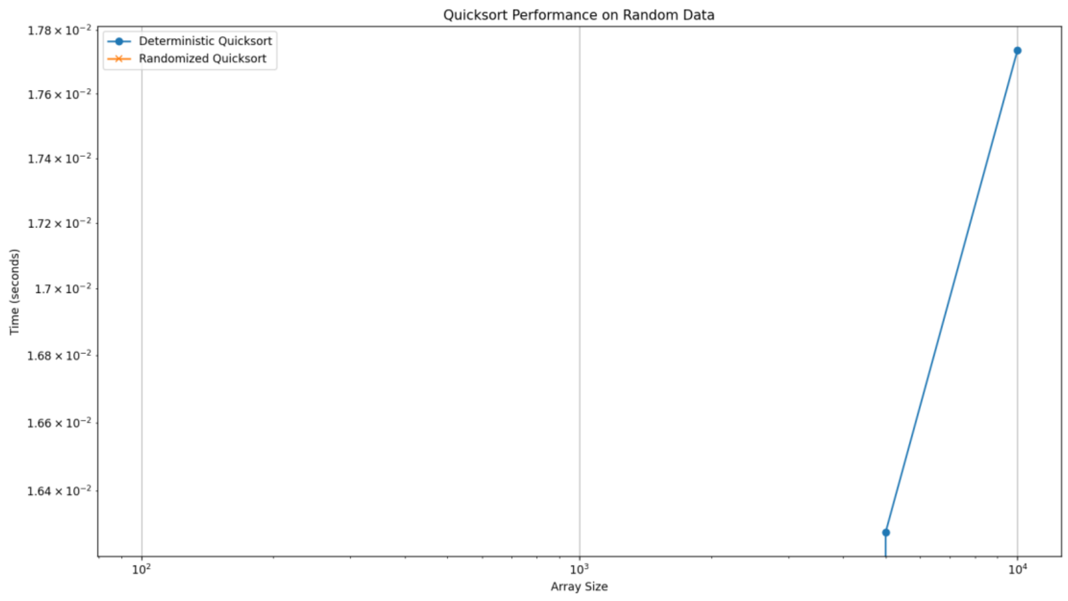
Theoretical Analysis:

- Randomized Quicksort also removes the sensitivity to input: pivot selection is uniformly distributed over the subarray.
- The average number of comparisons in Randomized Quicksort for a subarray of size n is 2nln+O(n).

4.1 Empirical Analysis

To compare the empirical analysis between deterministic vs Randomized Quick sort, I have implemented the empirical analysis python code for Random Array, Sorted Array and reverse sorted array.

Attaching the output of my first graphical representation for Quicksort performance on Random Data.



The graph presented illustrates the efficiency of deterministic quicksort and randomized quicksort when applied to randomly generated data sets of varying array sizes. An examination of the figure follows:

- **X-axis (Array Size):** The x-axis represents the size of the input array, varying from 10^2 $^($100 items) to 10^4 (10,000 items).
- **Vertical axis (Time measured in seconds):** The vertical axis is the time in seconds taken by the algorithm to sort the array. The values recorded are low, roughly of the order of 10^2 seconds.

Trend Analysis**:**

- Deterministic quicksort (blue line with circular markers) and randomized quicksort (orange line with "x" markers) follow roughly the same trends. However:
- Deterministic quicksort takes a bit longer than its randomized counterpart, as can be seen from the high position of the blue line.
- The discrepancy between the two methods is insignificant, suggesting that both are equally effective.

Logarithmic Scale: The logarithmic scale on the x-axis calls attention to the exponential growth over time accompanying the increase in input size.

Data Representation: There are only two measurements for deterministic quicksort (visible in the curve). If more size were measured, the trend would likely become smoother.

4.2 Observations:

- Randomized quicksort is a bit faster because of its pivot selection strategy that avoids worst-case scenarios (e.g., already sorted or reverse-sorted arrays).
- Using the last element as the pivot in deterministic quicksort can result in higher overhead for random input.

Scalability:

Both algorithms show the expected growth in time as the array size increases consistent with their average-case time complexity of O (n log n)

5.1 GitHub Repository:

https://github.com/rutus-code/MSCS532_Assignment5

5.2 References:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). The MIT Press.
- TutorialsPoint. (n.d.). *Randomized quick sort algorithm*.
- GeeksforGeeks. (n.d.). *Quicksort using random pivoting*.
- Khan Academy. (n.d.). *Analysis of Quicksort*.
- Rosenberg, A. (2011). *15-451 Algorithms: Quicksort* [Lecture notes]. Carnegie Mellon University.