Unique Karanjit

Department of Computer Science, University of the Cumberlands

MSCS-532-A01: Algorithms and Data Structures

Dr. Satish Penmatsa

Feb 2, 2025

Table of Contents

Quicksort Implementation and Analysis

Deterministic Quick Sort Introduction

Quicksort is a divide-and-conquer algorithm that works as follows:

1. First, select a pivot element from the array.
2. Partition the array into two subarrays: one with elements less than the pivot and one with elements greater than the pivot.
3. Recursively apply the same procedure to the subarrays.

Performance Analysis of Quick Sort – Deterministic

Best, Average, and Worst Cases:

- Best-case: The best case occurs when the pivot divides the array evenly into two halves at each recursion. In this case, the algorithm has a time complexity of $\boldsymbol{O}$ ($\boldsymbol{n \log n}$) where n is the number of elements in the array. When the pivot is chosen well (e.g., median or random pivot), the algorithm partitions the array into roughly equal halves. The depth of recursion is $\boldsymbol{O}$ ($\boldsymbol{n \log n}$) requiring $\boldsymbol{O}$ ($\boldsymbol{\log n}$) stack space so that the space complexity will be $\boldsymbol{O}$ ($\boldsymbol{\log n}$).
- Average-case: The average cases happen when a pivot typically divides the array into roughly equal halves. The time complexity is $\boldsymbol{O}$ ($\boldsymbol{n \log n}$) on average for a random selection of pivots. When the pivot is chosen well (e.g., median or random pivot), the algorithm partitions the array into roughly equal halves. The depth of recursion is $\boldsymbol{O}$ ($\boldsymbol{n \log n}$) requiring $\boldsymbol{O}$ ($\boldsymbol{\log n}$) stack space so that the space complexity will be $\boldsymbol{O}$ ($\boldsymbol{\log n}$).

- Worst-case: The worst case occurs when the pivot always divides the array in the worst possible way, for example, always selecting the smallest or largest element as the pivot. In this case, the time complexity is $\boldsymbol{O(n^2)}$. If the pivot is always the smallest or largest element, the algorithm degrades into a recursive call for **n-1** elements each time. This leads to $\boldsymbol{O(n)}$ recursive calls, requiring $\boldsymbol{O(n)}$. stack space.

The time complexity and space complexity of quick sort can be shown in the table as follows.

Case	Time Complexity	Space Complexity
Best Case	$\boldsymbol{O\ (n \log n)}$	$\boldsymbol{O\ (\log n)}$
Worst Case	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n)}$
Average Case	$\boldsymbol{O\ (n \log n)}$	$\boldsymbol{O\ (\log n)}$

Randomized Quick Sort

Randomized quicksort is an algorithm that works by selecting a pivot element uniformly at random from the array and partitioning the elements into subgroups. The implementation of randomized quicksort is helpful when there is a worst-case scenario like a reverse-sorted or sorted array.

Let us suppose T(n) is the average time complexity of the randomized quicksort for an array of size n. Then, the steps involved during the sorting are as follows.

1. Time taken for choosing a pivot: O (1)

2. Time taken for partitioning of the array: O(n)
3. Quicksort is applied here recursively. This is the main step that dominates the behavior of the algorithm.

We can use recursive relations to analyze the time complexity of the algorithm. The recurrence relation for T(n) representing the time complexity of the quicksort for an array of size n is:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

where,

k = size of one subarray.

T(k) and T(n-k-1) = Time taken to recursively sort two sub arrays.

O(n) = Time taken for partitioning.

Average-Case Analysis:

In randomized quicksort, the pivot element is chosen uniformly at random, so that the size of the left and right subarray is almost equal. Hence, we can assume k is almost equal to the halve of n. We can use this average-case assumption and then the recurrence becomes:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Using the master theorem, we get,

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Where:

a = 2 → number of sub-problems

b = 2 → division factor

d = 1 → partition time

Also, $log_b a = log_2 2 = 1$

Since, $d = log_b a$, the recurrence falls into Case 2 of the master theorem. Therefore, we can conclude:

$$\boldsymbol{T(n) = O(n \log n)}$$

Best-case and worst-case complexity

The best case happens when the pivot splits the array perfectly, so it is going to be the same calculation as above. So, for best-case complexity,

$$\boldsymbol{T(n) = O(n \log n)}$$

The worst case happens when the pivot divides the array into extremely unbalanced subarrays. For example, when one subarray has n-1 elements, whereas the other has 0. So, the recurrence resolves to:

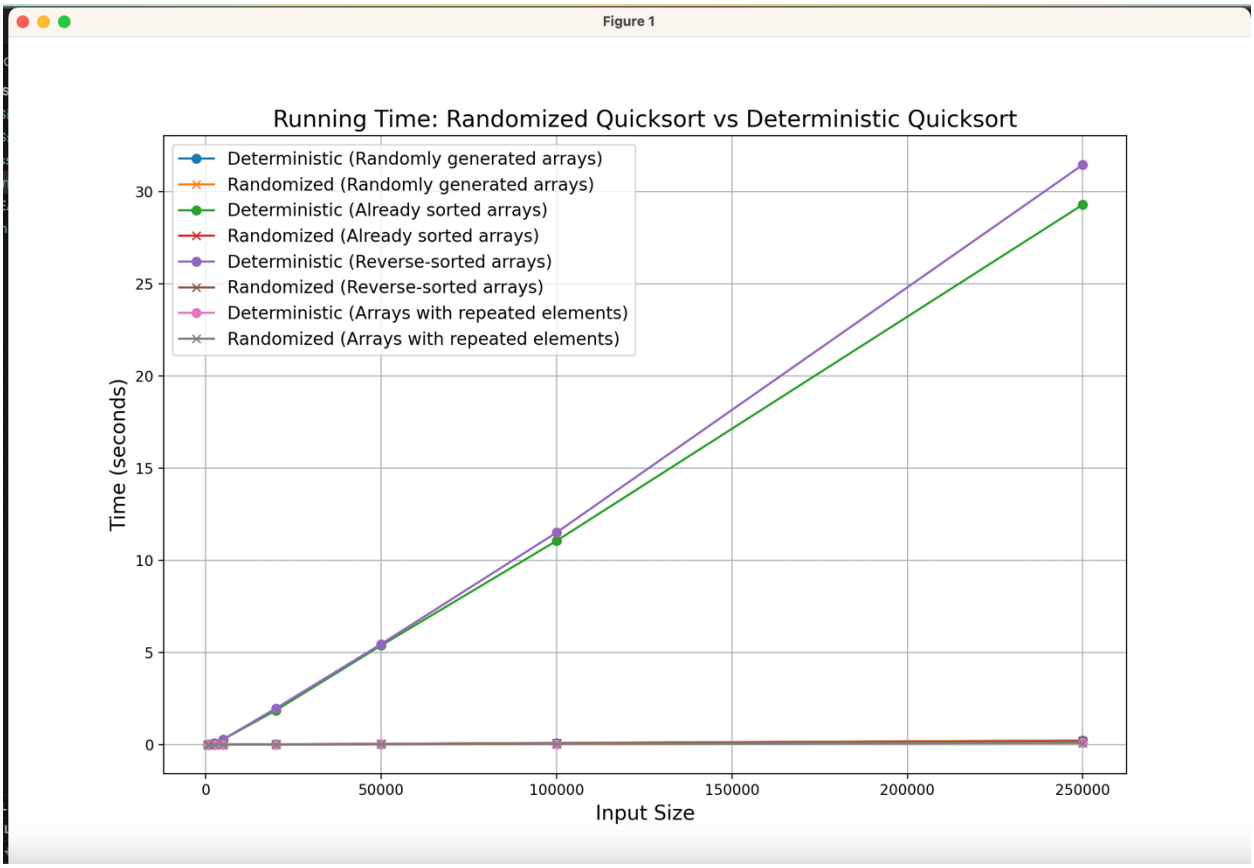
$$\boldsymbol{T(n) = O(n^2)}$$

Case	Time Complexity	Space Complexity
Best Case	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(\log n)}$
Worst Case	$\boldsymbol{O(n^2)}$	$\boldsymbol{O(n)}$
Average Case	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(\log n)}$

Table showing time complexity and space complexity for randomized quick sort.

The randomization significantly reduces the chance of hitting the worst case.

The graph showing the runtime between randomized and deterministic quicksort for different input sizes is shown below.



Based on this graph, deterministic quick sort performs worst for sorted and reverse-sorted arrays, whereas, for the same input, the randomized quick sort performs well. This concludes that the randomized quick sort outperforms the deterministic quick-sort approach.

Empirical Results with Data

- The empirical tests are supposed to show that the randomized Quicksort performs better in practice for large input sizes, especially on edge cases like sorted and reverse-sorted arrays. Randomized Quicksort avoids the worst-case behavior more effectively than deterministic Quicksort, which is proved by the graph shown above. The runtime for those operations is shown below.

```
Size: 500
Deterministic Quicksort (Sorted):         0.005402 seconds
Randomized Quicksort (Sorted):            0.000730 seconds
Deterministic Quicksort (Random):         0.000809 seconds
Randomized Quicksort (Random):            0.008469 seconds
Deterministic Quicksort (Reverse-Sorted): 0.005229 seconds
Randomized Quicksort (Reverse-Sorted):    0.000689 seconds
Deterministic Quicksort (Repeated):       0.000251 seconds
Randomized Quicksort (Repeated):          0.000347 seconds
-----------------------------------------

Size: 1000
Deterministic Quicksort (Sorted):         0.017662 seconds
Randomized Quicksort (Sorted):            0.001441 seconds
Deterministic Quicksort (Random):         0.000825 seconds
Randomized Quicksort (Random):            0.001488 seconds
Deterministic Quicksort (Reverse-Sorted): 0.018099 seconds
Randomized Quicksort (Reverse-Sorted):    0.001456 seconds
Deterministic Quicksort (Repeated):       0.000486 seconds
Randomized Quicksort (Repeated):          0.000535 seconds
-----------------------------------------

Size: 2500
Deterministic Quicksort (Sorted):         0.087067 seconds
Randomized Quicksort (Sorted):            0.003450 seconds
Deterministic Quicksort (Random):         0.002164 seconds
Randomized Quicksort (Random):            0.003779 seconds
Deterministic Quicksort (Reverse-Sorted): 0.089330 seconds
Randomized Quicksort (Reverse-Sorted):    0.003486 seconds
Deterministic Quicksort (Repeated):       0.001221 seconds
Randomized Quicksort (Repeated):          0.001070 seconds
-----------------------------------------

Size: 5000
Deterministic Quicksort (Sorted):         0.293376 seconds
Randomized Quicksort (Sorted):            0.006514 seconds
Deterministic Quicksort (Random):         0.004599 seconds
Randomized Quicksort (Random):            0.007306 seconds
Deterministic Quicksort (Reverse-Sorted): 0.292867 seconds
Randomized Quicksort (Reverse-Sorted):    0.006657 seconds
Deterministic Quicksort (Repeated):       0.001860 seconds
Randomized Quicksort (Repeated):          0.001923 seconds
-----------------------------------------

Size: 20000
Deterministic Quicksort (Sorted):         1.863530 seconds
Randomized Quicksort (Sorted):            0.019427 seconds
Deterministic Quicksort (Random):         0.017284 seconds
Randomized Quicksort (Random):            0.022282 seconds
Deterministic Quicksort (Reverse-Sorted): 1.970533 seconds
Randomized Quicksort (Reverse-Sorted):    0.019384 seconds
Deterministic Quicksort (Repeated):       0.007042 seconds
Randomized Quicksort (Repeated):          0.007061 seconds
-----------------------------------------
```

```
Size: 50000
Deterministic Quicksort (Sorted):         5.383780 seconds
Randomized Quicksort (Sorted):            0.044975 seconds
Deterministic Quicksort (Random):         0.042147 seconds
Randomized Quicksort (Random):            0.044968 seconds
Deterministic Quicksort (Reverse-Sorted): 5.460912 seconds
Randomized Quicksort (Reverse-Sorted):    0.043209 seconds
Deterministic Quicksort (Repeated):       0.018607 seconds
Randomized Quicksort (Repeated):          0.019380 seconds
------------------------------------------

Size: 100000
Deterministic Quicksort (Sorted):         11.064318 seconds
Randomized Quicksort (Sorted):            0.075673 seconds
Deterministic Quicksort (Random):         0.091163 seconds
Randomized Quicksort (Random):            0.094403 seconds
Deterministic Quicksort (Reverse-Sorted): 11.513323 seconds
Randomized Quicksort (Reverse-Sorted):    0.082616 seconds
Deterministic Quicksort (Repeated):       0.039060 seconds
Randomized Quicksort (Repeated):          0.038263 seconds
------------------------------------------

Size: 250000
Deterministic Quicksort (Sorted):         29.307747 seconds
Randomized Quicksort (Sorted):            0.193500 seconds
Deterministic Quicksort (Random):         0.226839 seconds
Randomized Quicksort (Random):            0.233850 seconds
Deterministic Quicksort (Reverse-Sorted): 31.471617 seconds
Randomized Quicksort (Reverse-Sorted):    0.179624 seconds
Deterministic Quicksort (Repeated):       0.101401 seconds
Randomized Quicksort (Repeated):          0.098050 seconds
------------------------------------------
```

As concluded, for large inputs where the array is already sorted, the randomized approach performs well which is visualized with the runtimes for the same input.

References

GeeksforGeeks. (2023, September 14). *Quicksort using random pivoting*. https://www.geeksforgeeks.org/quicksort-using-random-pivoting/