Unique Karanjit

Department of Computer Science, University of the Cumberlands

MSCS-532-A01: Algorithms and Data Structures

Dr. Satish Penmatsa

Jan 26, 2025

Randomized Quicksort Analysis

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 sorted or reverse-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:

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

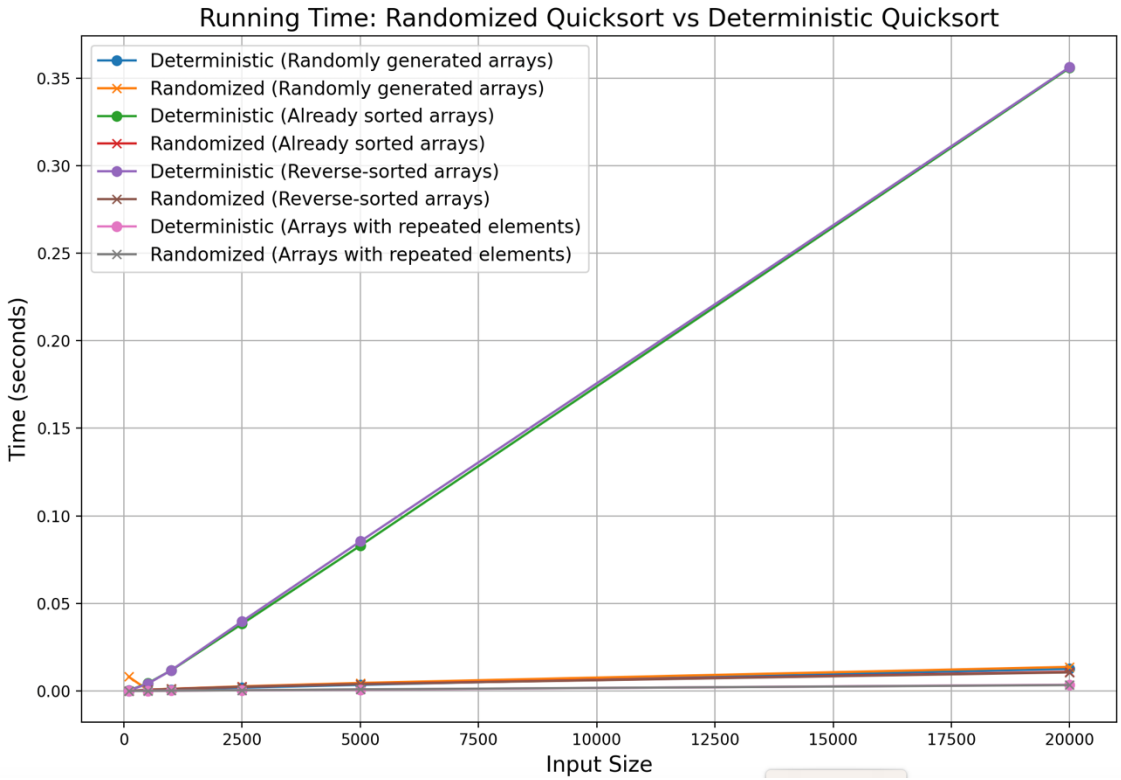

Fig: Graph showing run time for randomized quicksort vs Deterministic Quicksort based on input size and time.

Empirical Results vs Expected Theoretical Performance:

Randomized quicksort consistently performs near $O(n \log n)$ in all cases due to its random pivot selection. Also, deterministic quick sort also aligns with the theory showing $O(n \log n)$ for random input and $O(n^2)$ for sorted or reverse-sorted arrays. The minor discrepancy might have occurred because of how the algorithms may access memory, but the discrepancy is little to none. Hence, the empirical results match with the expected theoretical performance.

```
Size: 100
Deterministic Quicksort (Sorted):         0.000323 seconds
Randomized Quicksort (Sorted):            0.000142 seconds
Deterministic Quicksort (Random):         0.000131 seconds
Randomized Quicksort (Random):            0.007834 seconds
Deterministic Quicksort (Reverse-Sorted): 0.000324 seconds
Randomized Quicksort (Reverse-Sorted):    0.000132 seconds
Deterministic Quicksort (Repeated):       0.000034 seconds
Randomized Quicksort (Repeated):          0.000041 seconds
-------------------------------------------

Size: 500
Deterministic Quicksort (Sorted):         0.004325 seconds
Randomized Quicksort (Sorted):            0.000658 seconds
Deterministic Quicksort (Random):         0.000406 seconds
Randomized Quicksort (Random):            0.000768 seconds
Deterministic Quicksort (Reverse-Sorted): 0.004261 seconds
Randomized Quicksort (Reverse-Sorted):    0.000624 seconds
Deterministic Quicksort (Repeated):       0.000105 seconds
Randomized Quicksort (Repeated):          0.000117 seconds
-------------------------------------------

Size: 1000
Deterministic Quicksort (Sorted):         0.012079 seconds
Randomized Quicksort (Sorted):            0.001226 seconds
Deterministic Quicksort (Random):         0.000779 seconds
Randomized Quicksort (Random):            0.001272 seconds
Deterministic Quicksort (Reverse-Sorted): 0.012172 seconds
Randomized Quicksort (Reverse-Sorted):    0.001169 seconds
Deterministic Quicksort (Repeated):       0.000184 seconds
Randomized Quicksort (Repeated):          0.000222 seconds
-------------------------------------------
```

```
Size: 2500
Deterministic Quicksort (Sorted):         0.037921 seconds
Randomized Quicksort (Sorted):            0.002444 seconds
Deterministic Quicksort (Random):         0.001788 seconds
Randomized Quicksort (Random):            0.002765 seconds
Deterministic Quicksort (Reverse-Sorted): 0.040570 seconds
Randomized Quicksort (Reverse-Sorted):    0.002436 seconds
Deterministic Quicksort (Repeated):       0.000446 seconds
Randomized Quicksort (Repeated):          0.000428 seconds
-------------------------------------------

Size: 5000
Deterministic Quicksort (Sorted):         0.084245 seconds
Randomized Quicksort (Sorted):            0.003836 seconds
Deterministic Quicksort (Random):         0.003592 seconds
Randomized Quicksort (Random):            0.004644 seconds
Deterministic Quicksort (Reverse-Sorted): 0.084518 seconds
Randomized Quicksort (Reverse-Sorted):    0.004068 seconds
Deterministic Quicksort (Repeated):       0.000909 seconds
Randomized Quicksort (Repeated):          0.000833 seconds
-------------------------------------------

Size: 20000
Deterministic Quicksort (Sorted):         0.355518 seconds
Randomized Quicksort (Sorted):            0.010766 seconds
Deterministic Quicksort (Random):         0.012551 seconds
Randomized Quicksort (Random):            0.014771 seconds
Deterministic Quicksort (Reverse-Sorted): 0.354558 seconds
Randomized Quicksort (Reverse-Sorted):    0.012062 seconds
Deterministic Quicksort (Repeated):       0.003076 seconds
Randomized Quicksort (Repeated):          0.003417 seconds
-------------------------------------------
```

As per the runtime data, we can conclude that the randomized quicksort outperforms deterministic quicksort for sorted and reverse-sorted arrays.

Hashing with Chaining Analysis:

The time complexity for insert, delete, and search operations on the hash table is $O(1)$ on average and can get up to $O(n)$ in case of collisions or worst case.

1. Search Time:

$$\boldsymbol{Expected\ Search\ Time = O(1 + \frac{n}{m})}$$

$where, n\ is\ the\ number\ of\ key\ in\ the\ table, m\ is\ the\ number\ of\ available\ buckets.$

This equation proves that if the load factor n/m is small, the time is constant, but it increases with the number of elements as load factor increase.

2. Insert Time

$$\boldsymbol{Expected\ Insert\ Time = O(1 + \frac{n}{m})}$$

$, where, n\ is\ the\ number\ of\ key\ in\ the\ table, m\ is\ the\ number\ of\ available\ buckets.$

3. Delete Time

$$\boldsymbol{Expected\ Delete\ Time = O(1 + \frac{n}{m})}$$

$where, n\ is\ the\ number\ of\ key\ in\ the\ table, m\ is\ the\ number\ of\ available\ buckets.$

Hence, the time depends on the load factor, represented as $\boldsymbol{\alpha = \frac{n}{m}}$

When the load factor is less, the hash table operates efficiently since there are fewer collisions and time operation goes to $O(1)$. As α increases, the hash table becomes inefficient, and the operations could degrade as high as $O(n)$.

Since the load factor plays a vital role in efficiency, we can apply certain strategies to handle the efficiency. For example, if the load factor is increasing significantly, we can increase the number of slots or buckets, which will reduce the load factor. Such a strategy can be applied by setting up a threshold like 0.5 or as per the requirement.

```
--------------------------------------
Showing the Table
--------------------------------------
Index 1: [('AudiMake', 'Audi')]
Index 4: [('BMWMake', 'BMW')]
Index 5: [('LexusMake', 'Lexus')]
Index 8: [('ToyotaMake', 'Toyota')]
Index 10: [('AcuraMake', 'Acura')]
Index 15: [('TeslaMake', 'Tesla')]


--------------------------------------
Searching for entries LexusMake and AcuraMake from the Table
--------------------------------------
Make: Lexus
Make: Acura


--------------------------------------
Deleting entry ToyotaMake the Table
--------------------------------------
Deleted Make: True


--------------------------------------
Searching entry ToyotaMake the Table after deletion
--------------------------------------
After Deletion:: Make: None


--------------------------------------
Showing the Table
--------------------------------------
Index 1: [('AudiMake', 'Audi')]
Index 4: [('BMWMake', 'BMW')]
Index 5: [('LexusMake', 'Lexus')]
Index 10: [('AcuraMake', 'Acura')]
Index 15: [('TeslaMake', 'Tesla')]
```