Analysis of Randomized Quicksort vs Deterministic Quicksort

Randomized Quicksort is created in order to avoid performance penalties in edge cases, such as when the input data is already sorted. In this algorithm, the pivot element is chosen randomly from the array, which helps ensure that the input order does not affect the performance. After selecting the pivot, the array is divided into two subarrays: one containing elements smaller than or equal to the pivot and the other containing elements greater than the pivot. This process is repeated recursively on the subarrays, resulting in an average-case time complexity of O(n.logn). However, the actual running time depends on the partitioning, which determines the sizes of the subarrays at each step.

Let's consider a subarray of size n to find the average-case time complexity complexity. On average, a randomly chosen pivot divides the array into two subarrays of sizes i and $n - i - 1$, where i ranges from 0 to $n - 1$. The partitioning step takes $O(n)$ time, as each element is compared to the pivot.

Then, the recurrence relation for the expected runtime T(n) is as below:

$T(n) = T(i) + T(n - i - 1) + O(n)$ where i is the size of one partition.

Since the pivot is chosen uniformly at random, all i are equally likely. The expected size of the partitions is balanced, leading to an expectation that each recursive step divides the array into approximately equal halves. The recurrence relation simplifies to:

$T(n) = (1/n)\,(O\ to\ n - 1)\,\Sigma(T(i) + T(n - i - 1)) + O(n)$

Through substitution and simplification, this recurrence resolves to $T(n) = O(n.logn)$.

Here, the log(n) term is valid because the array is repeatedly divided into half during partitioning, while the n term accounts for the comparisons at each recursion level.

Now using indicator random variables to further validate this result:

Let Xij represent whether the ith and jth elements are compared during execution. The probability of this comparison depends on the first pivot chosen that splits i and j, yielding $Pr(Xij = 1) = 2 / (|j - i + 1|)$. Summing over all pairs, the expected number of comparisons is $O(n.logn)$, matching the result derived from recurrence relations. Thus, the average-case time complexity of Randomized Quicksort is $O(nlogn)$, as calculated by the probabilistic balancing of partitions and the logarithmic depth of recursion.

Two different input samples of 10000 and 20000 integers were used to compare the runtime for the randomized and deterministic quicksort. The data volume is doubled to understand whether runtime increases linearly or exponentially.

Here is the time taken to sort for the sample with 10000 integers:

Quick sort type	Time taken for sorted data	Time taken for reverse sorted data	Time taken for random data with no duplicates	Time taken for random data with duplicates
Deterministic	1.7808 second	1.6349 second	0.0093 second	0.0102 second
Randomized	0.0109 second	0.0169 second	0.0150 second	0.0136 second

The performance of deterministic and randomized Quicksort was tested on four types of data: sorted, reverse-sorted, random data with no duplicates, and random data with duplicates. Due to inefficient pivot selection, the deterministic version showed poor performance on sorted and

reverse-sorted data, taking 1.7808 and 1.6349 seconds, respectively. It could only perform well on random data, taking about 0.0093 and 0.0102 seconds. On the other hand, randomized Quicksort consistently performed well across all cases, with times ranging from 0.0109 to 0.0169 seconds, demonstrating its resilience to edge cases and better handling of diverse input datasets.

Here is the time taken to sort the data with 20000 integers:

Quick sort type	Time taken for sorted data	Time taken for reverse sorted data	Time taken for random data with no duplicates	Time taken for random data with duplicates
Deterministic	7.395 second	7.9912 second	0.0205 second	0.0201 second
Randomized	0.0218 second	0.0208 second	0.0212 second	0.0217 second

From the above table, we can see that randomized quick sort is very consistent even when the data volume gets large. On the other hand quick sort performs worse when the data volume is high and if the data is present in sorted order. The outcomes are consistent with our theoretical understanding of deterministic and randomized quicksort. For a deterministic quicksort, it takes O(n^2) time to execute when the data is sorted. This is validated by above table data because when the data volume doubled from 10000 to 2000, the time taken to execute increased by 4X. Meanwhile, the randomized quicksort has a time complexity of O(n.logn). This is why the time taken to sort is consistent across all types of data.

Analysis of Hashing with Chaining:

In simple uniform hashing, every key has an equal chance of being placed in any slot in the table. The speed of search, insert, and delete operations depends on the load factor ($\alpha=n/m$, where n is the number of items and m is the number of slots). For search, it usually takes $O(1+\alpha)$ time because we first find the slot and then check the items in the list stored there, which is about as long as the load factor. Insert is faster at $O(1)$ since we just add the new item to the list in its slot. Delete takes $O(1+\alpha)$ because we need to find the item in the list before removing it. In the worst case, if all the keys end up in the same slot, the list can get very long, making search and delete operations take $O(n)$. To keep the operations fast, it's important to keep the load factor low (preferably less than 1) by resizing the table when it becomes too full and using a good hash function to spread items evenly across the slots.

The load factor ($\alpha=n/m$, where n is the number of elements and m is the number of slots) has a big impact on how well a hash table performs. When the load factor is low i.e. ($\alpha \ll 1$), most slots are empty or have very short chains, so operations like search, insert, and delete are quick, averaging $O(1)$. However, as the load factor gets higher ($\alpha>1$), more elements get stored in the same slots, causing longer chains and more collisions. This makes search and delete slower, with average time $O(1+\alpha)$. In the worst case, if all elements take up the same slot, the chain will be as long as the total number of elements (n), and operations could take $O(n)$. To maintain good performance, the load factor should be kept low by resizing the table when it gets too full and using a good hash function to spread elements evenly across the slots.

One effective method to keep the load factor low and reduce collisions is dynamic resizing, where the hash table is expanded (often doubled in size) when the load factor exceeds a specific range (0.75 to 0.85). All elements are then rehashed into the larger table, which takes

O(n) time but happens rarely, keeping average operations at O(1). A good hash function is critical for evenly distributing elements and to avoid clustering. Additionally, choosing an initial table size that is large and preferably a prime number helps. Collision resolution methods like chaining or open addressing also improve efficiency.