**Assignment 3: Understanding Algorithm Efficiency and Scalability**

**Part 1: Randomized Quicksort Analysis**

1. **Implementation**

A Python script has been created to implement the Randomized Quicksort algorithm. Discussing the script, within it first a method called ‘partition\_randomized’ has been created where a random pivot index has been chosen between the low and the high. Then the pivot element is moved to the end of the array. Then the partition of the array will be done where at the end the pivot has been placed in its correct position. Then another method called ‘quicksort\_randomized’ has been created to make the partition around a random pivot index and get the index. Then the script recursively sorts the elements before and after the partition. While doing that, a method is called to check if the array contains one element or is empty and finally the sorting will be done. Here the edge cases like handling of empty arrays, single element arrays, repeated elements within the array, already sorted arrays, reversely sorted arrays etc. are there.

1. **Analysis**

In the Randomized Quicksort algorithm, a pivot element needs to be chosen uniformly. This element is a random element selected from the array in which each of the recursive calls is made. The created arrays are partitioned around the selected pivot and always after each recursive call, all of the elements are compared with the pivot element where the lesser values are placed to the left of the array and greater are placed to the right of the array. Then recursively the two partitions are sorted to make the sort successful. In this approach, the random choice of the pivot element helps in balancing the partitions on average where it can be expected that the average case time complexity will be .

Let consider that is the expected time complexity of the mentioned algorithm and is the size of the input array. Then,

The base case i.e. if then no sorting is required as there is only one element. Thus, the time complexity will be .

If then a random pivot will be chosen and taking time to make the partition. Considering that is one partition then another partition of the array will be . So, the time complexity can be described as the following.

By making the average of all partitions, the time complexity will be like

This can be solved as which is the average time complexity of this mentioned algorithm.

The average time complexity is because first we have chosen a pivot element randomly where each of the exists elements can be considered as pivot element which is likely to split the array into two equal portions and with each recursive step, the portions are half of the original size of the array. The depth of recursion and total work executed in partitioning is and it takes levels of recursion which makes it as the final time complexity of the approach.

1. **Comparison**

Comparing the running time of the Randomized Quicksort with the Deterministic Quicksort based on different input sizes and distribution are following:

|  |  |  |
| --- | --- | --- |
|  | | **Randomized Quicksort vs. Deterministic Quicksort** |
| **Randomly Generated Arrays** | **Expected** | Both of the algorithm will take same time as average-case time complexity. |
| **Observed** | Randomized Quicksort is performed slightly better and takes less time than Deterministic Quicksort. |
| **Already Sorted Arrays** | **Expected** | Worst-case behaviour will be observed by the first element of the Deterministic Quicksort which takes . |
| **Observed** | Randomized Quicksort is performed slightly better and takes less time than Deterministic Quicksort. |
| **Reverse-Sorted Arrays** | **Expected** | Like the Already Sorted Arrays, worst-case behaviour will be observed by the first element of the Deterministic Quicksort which takes . |
| **Observed** | Randomized Quicksort is performed better and takes less time () than Deterministic Quicksort. |
| **Arrays with Repeated Elements** | **Expected** | Both of the algorithm will take same time as these types of algorithms are well capable of handling these sorts of approaches. |
| **Observed** | Randomized Quicksort is performed slightly better and takes less time than Deterministic Quicksort. |

Considering the comparison, the results may vary due to the overheads from the generation of random numbers into the arrays, the performance of cache memory and Python interpreter overheads. But observing all of the scenarios, Randomized Quicksort is performed slightly better and takes less time than Deterministic Quicksort.

**Part 2: Hashing with Chaining**

1. **Implementation**

Implementation of the hash table required several hash functions. In this scope of work, first, a hash table and hash keys are created. Then within the implementation, insert, search, delete and display of key-pair values are incorporated. The entire implementation was created with the help of lists.

1. **Analysis**

To analyse the implementation, first, a load factor needs to be considered which is α in this case. Here, (where n is the number of elements and m is the number of buckets in the hash table). The required times are like the following:

In the case of insertion once a new element is inserted either in the beginning or at the end the expected time required is .

In the case of searching, the average time required for the operation is as here traversal of the list will be performed.

In case of deletion, similar to the searching the required time will be .

If the load factor is low (α < 1), fewer collisions, constant time is required for operations like insertion, searching and deletion. But if the load factor is high (α > 1) then a higher chance of collision may appear and the time required for the mentioned operations will be around .

To maintain a low factor and minimise collision, resizing the hash table may primarily require and for that, a threshold limit needs to be there. Apart from that, doubling the table size or rehashing with a new function may also be helpful in this scenario.