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

**Part 1: Randomize Quicksort Analysis**

Randomized Quicksort also follows the standard Quicksort method, which means, it selects the pivot point for the array divisions into subarrays, then starts to compare and swapping the elements, then finally joining those subarrays into a single sorted array. However, this algorithm is the best to overcome the worst-case scenario of Quicksort, which is O(n^2), by randomly selecting a pivot point and the rest process follows the standard Quicksort method.

**Implementation**

The code snippet for implementation of Randomized Quicksort in python is given below:

A screen shot of a computer program

Description automatically generated

**Analysis**

The average Time Complexity of Randomized Quicksort is O(nlogn). The recurrence relation is T(n) = T(k) + T(n - k - 1) + O(n), where k is the size of the left subarray. When using the randomized quicksort, an array is divided into approximately half, which means k ≈ n/2. On solving this problem using the Master's Theorem, we get the recurrence T(n) = 2T(n/2) + O(n), which then solves to be O(nlogn).

**Comparison**

I compared the runtime for the Randomized Quicksort with the Deterministic Quicksort, with different data sizes and data types, and the findings are placed in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| Data size | Data Type | Quicksort Time (s) | Randomized Quicksort Time (s) |
| 100 | sorted | 0.000380 | 0.000077 |
| 100 | reverse\_sorted | 0.000283 | 0.000066 |
| 100 | random | 0.000065 | 0.000064 |
| 100 | repeated | 0.000078 | 0.000076 |
| 1000 | sorted | 0.024249 | 0.000903 |
| 1000 | reverse\_sorted | 0.024759 | 0.000933 |
| 1000 | random | 0.000914 | 0.000933 |
| 1000 | repeated | 0.002764 | 0.001956 |
| 10000 | sorted | 2.319578 | 0.012581 |
| 10000 | reverse\_sorted | 2.407096 | 0.012248 |
| 10000 | random | 0.011332 | 0.012763 |
| 10000 | repeated | 0.226307 | 0.128573 |

From this table, I see that runtime remains almost similar for the Randomized Quicksort for any data type except for the repeated elements. Time increases for the sorting when elements are repeated. But, the runtime for Quicksort differs accordingly to the data types and sizes.

**Part 2: Hashing with Chaining**

Hashing with chaining is a method for resolving collisions in a hash table. There will be a collision in a hash table when two mapped keys point to the same slot in the hash table. So, this method organizes those multiple elements to the same index, without overwriting or deleting the previous key.

**Implementation:**

The python code snippet for implementation of hashing with chaining is given below:

A screen shot of a computer program

Description automatically generated

**Analysis:**

There are three main functions in hashing with chaining. They are **Insert**, **Search**, and **Delete**. The expected Time Complexity for those operations is O(1 +α), where α (load factor) is the ratio of number of elements to the number of slots. In a hash function with a minimum or low load factor, which is usually α ≤ 0.7, has almost to the constant time operations. When the load factor is higher, then it tends to have longer chains and gets the performance to its worst case which is O(n). So, to maintain efficiency, hash tables should be resized, and elements should be rehashed, when the load factor exceeds the threshold. While rehashing the table, it tends to have the tradeoff of memory, which means we must double the size for the constant like efficiency of functions. Dynamic resizing usually is done when the load factor exceeds its threshold. This way, the efficiency of a hash operations remains constant.