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

**Mazen Abdul Rahman Mohammed**

**Student ID: 005030764**

**University of The Cumberlands**

**Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term**

**Dr. Vanessa Cooper**

**11/10/2024**

## Part 1: Randomized Quicksort Analysis

### Implementation



### Analysis

### 1. Basic Structure of Randomized Quicksort

Randomized Quicksort selects a pivot element uniformly at random from the array and partitions the array into two subarrays. The partitioning process places the pivot in its final sorted position, with elements less than the pivot on one side and elements greater on the other. The algorithm then recursively sorts the two subarrays.

### 2. Recurrence Relation for Randomized Quicksort

Let's define:

* T(n): the expected time to sort an array of n elements.

In each recursive call, Randomized Quicksort:

1. Partitions the array in O(n) time.
2. Makes recursive calls on two sub-arrays formed by the partitioning.

The recurrence relation for Randomized Quicksort is therefore:

T(n)=O(n)+E[T(X)]+E[T(n−X−1)]

where X is the size of the first partition (the number of elements smaller than the pivot), and E[T(X)] represents the expected time on this partition size.

### 3. Expected Time Complexity Analysis

#### Expected Partition Sizes

With a randomly chosen pivot, the partition sizes vary, but on average, each pivot divides the array into two halves. We can estimate that, for a random pivot, the expected sizes of the partitions are about n/2n and n/2.

Thus, the recurrence simplifies to:

T(n)=T(n/2)+T(n/2)+O(n)

which can be written as:

T(n)=2⋅T(n/2)+O(n)

This recurrence is similar to the one for Merge Sort, where each level of recursion performs O(n) work, and there are O(log n) levels. Solving this recurrence yields T(n)=O(n log n)

#### Indicator Random Variables for Comparisons

Another way to analyze the expected number of comparisons is by using indicator random variables. Let C(n) represent the expected number of comparisons in Randomized Quicksort.

Define Xi,j​ as an indicator random variable, where Xi,j=1 if element ai is compared to aj​ at some point in the execution, and 0 otherwise.

Since we are interested in pairs of comparisons, we have:

C(n E[Xi,j]

For a pair (ai,aj) to be compared, one of them must be the first pivot chosen from the subarray that includes both elements. Since the pivot is chosen randomly, the probability of comparing any two elements ai and aj is 2/(j−i+1)

This gives us:

E[Xi,j]=2/j(−i+1)

Summing over all pairs (i,j) we find that:

C(n)= ∑1≤i<j≤n 2/(j−i+1)=O(n log n)

### 4. Summary

* **Average-Case Time Complexity**: The expected number of comparisons in Randomized Quicksort is O(n log n)
* **Intuition**: The random pivot selection ensures that, on average, the array is divided into roughly two halves at each recursive step, leading to a recurrence similar to Merge Sort.
* **Use of Indicator Variables**: By defining indicator variables for comparisons and calculating the expected number of comparisons, we confirm that the expected time complexity is O(n log n)

Thus, Randomized Quicksort achieves an average-case time complexity of O(n log n) due to the balance provided by the random pivot selection and the expected number of comparisons across all recursive calls.

### Comparision

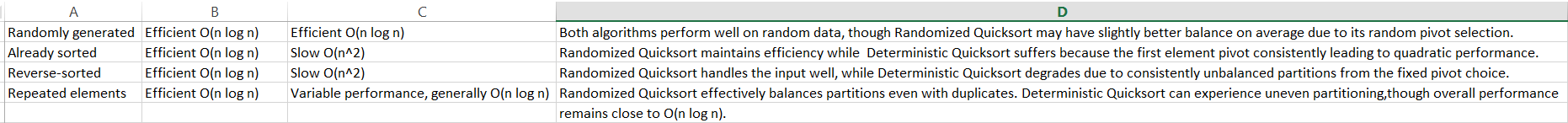
To empirically compare the performance of Randomized Quicksort and Deterministic Quicksort let’s prepare arrays with these different characteristics:

* **Randomly generated arrays**: This simulates an average case scenario for both algorithms.
* **Already sorted arrays**: This case can potentially worsen the performance of Deterministic Quicksort since the first element pivot choice will create unbalanced partitions.
* **Reverse-sorted arrays**: Similar to the sorted case but ordered oppositely, which should also challenge Deterministic Quicksort.
* **Arrays with repeated elements**: These will test the algorithms' ability to handle duplicates efficiently.

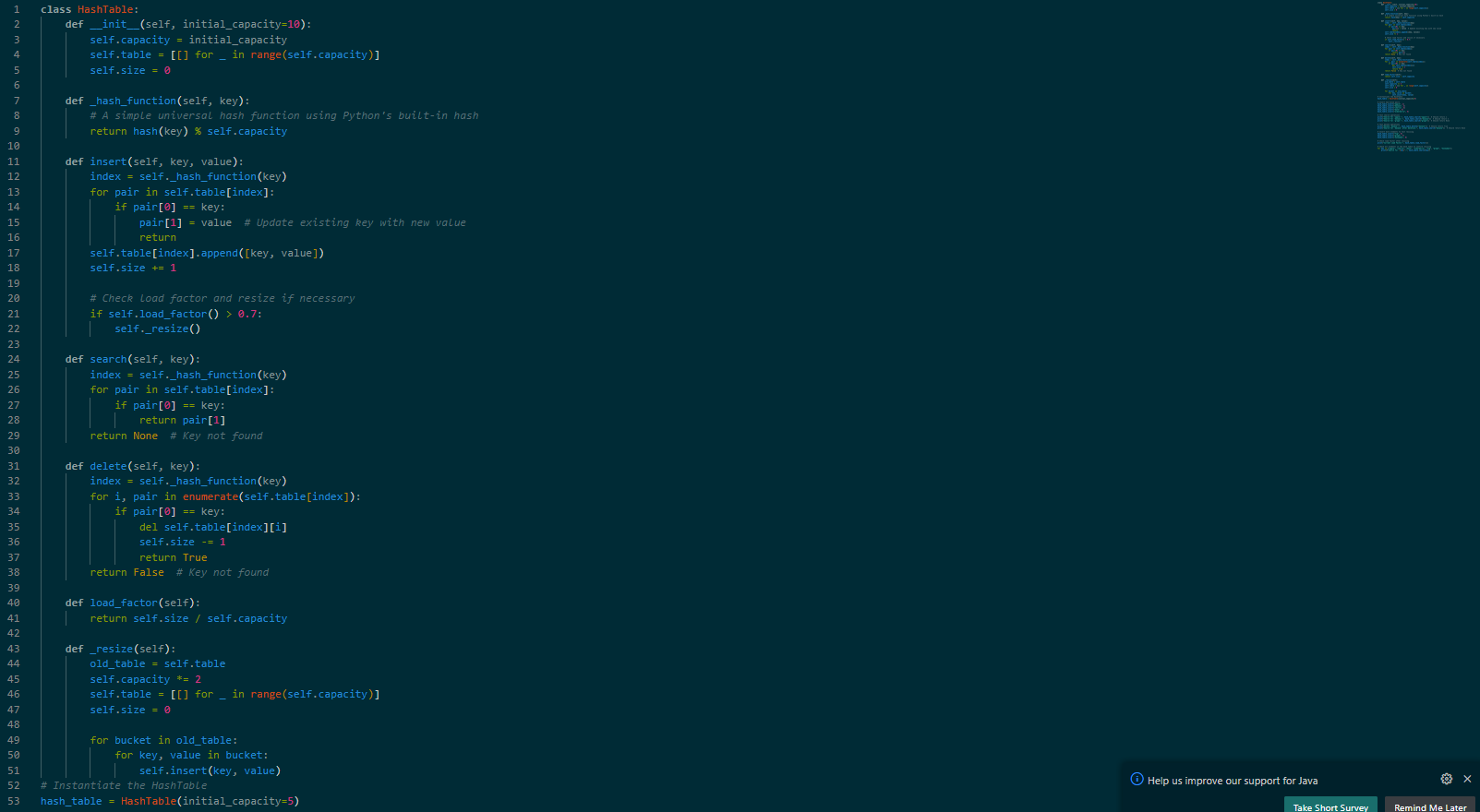
Run Tests and Measure Performance

Using Python’s timeit library or a similar tool, measure the running time for each input type across a range of array sizes (e.g., 1,000; 10,000; and 100,000 elements). Record the time each algorithm takes to complete the sort.

Results:



## Part 2: Hashing with Chaining

1. Implementation  


#### 2. Analysis

##### Expected Time Complexity

With simple uniform hashing, we assume each key is equally likely to be hashed into any slot independently, which helps distribute elements across the table.

* **Search**: Expected time is O(1+α) where α\alphaα is the load factor, since on average, only a few items will be in each bucket due to chaining.
* **Insert**: Expected time is also O(1+α) as we only add a new element to the end of a list within a bucket.
* **Delete**: Expected time remains O(1+α) as it involves searching within the bucket and deleting the element if found.

When the load factor is small, these operations approach O(1) on average.

##### Effect of Load Factor

The **load factor** (α=number of elements/number of slots) indicates how full the hash table is. A high load factor means more collisions, leading to longer chains in each slot and thus slower operations.

To maintain low α and optimize performance:

* **Resize dynamically**: As seen in the implementation, we double the hash table’s capacity when the load factor exceeds a threshold (e.g., 0.7).
* **Minimize collisions**: A suitable hash function distributes elements uniformly across slots. Using a universal hash function family ensures low collision probability and improves performance.

##### Summary

* By resizing the table based on the load factor, we maintain O(1) average time for insert, search, and delete.
* A low load factor ensures faster lookups and minimal chaining, which is achieved through dynamic resizing and an efficient hash function that distributes elements uniformly.