**Analysis of Randomized Quicksort and Hash Tables**

Asrith Krishna Vejandla

Student ID: 005033913

MSCS-532 Algorithms and Data Structures

Assignment 3

Professor Satish Penmatsa

January 26, 2025

**Analysis of Randomized Quicksort**

**Partitioning Process:**

In each step of Quicksort, the array is divided into two smaller subarrays based on the pivot:

Elements smaller than or equal to the pivot.

Elements greater than the pivot.

Partitioning involves scanning the array once, which takes O(n) time.

A computer code with text

AI-generated content may be incorrect.

**Random Pivot Selection:**

Randomized Quicksort differs from Deterministic Quicksort in that the pivot element is chosen randomly. This ensures the split between the two subarrays is probabilistically balanced, avoiding the worst-case scenario of highly skewed partitions.

**Recurrence Relation:**

Let T(n) represent the time complexity of sorting an array of size n. The partitioning process divides the array into two subarrays of sizes i and n−i−1, where i is the number of elements in the first subarray. The recurrence relation for the algorithm is:

T(n)=T(i)+T(n−i−1)+O(n),

where O(n) accounts for the partitioning.

**Average-Case Analysis:**

Since the pivot is chosen randomly, the expected sizes of the two subarrays are approximately equal (on average, each subarray contains n/2 elements). Substituting this into the recurrence gives T(n)=2T(n/2)+O(n).

**Using the Master Theorem for divide-and-conquer algorithms:**

The recurrence has the form

T(n)=aT(n/b)+O(nd), where a=2, b=2, and d=1.

Since a=bd, the solution is T(n)=O(nlogn).

Intuition for O(nlogn):

Each recursion level processes all n elements during partitioning (O(n)).

The recursion depth is proportional to logn, as the array size is halved at each level.

This results in an average-case time complexity of O(nlogn)

**Comparison with Deterministic Quicksort:**

In Deterministic Quicksort, the pivot is chosen as a fixed element, such as the first or last element of the subarray. This approach can lead to highly unbalanced partitions for certain input distributions (e.g., already sorted arrays), resulting in a worst-case time complexity of O(n2).

**A screenshot of a computer program

AI-generated content may be incorrect.**

We empirically compare the performance of Randomized Quicksort and Deterministic Quicksort on arrays of varying sizes and distributions:

Random Arrays: Elements are randomly generated.

Already Sorted Arrays: Elements are sorted in ascending order.

Reverse-Sorted Arrays: Elements are sorted in descending order.

Arrays with Repeated Elements: All elements have the same value.

Randomly Generated Arrays: Both algorithms perform similarly, with a time complexity of O(nlogn).

A screenshot of a computer program

AI-generated content may be incorrect.

**Already Sorted and Reverse-Sorted Arrays:**

Deterministic Quicksort: Performs poorly (O(n2)) due to highly unbalanced partitions.

Randomized Quicksort: Performs well, maintaining O(nlogn) due to random pivot selection.

Arrays with Repeated Elements: Both algorithms handle repeated elements efficiently, but randomized pivot selection avoids pathological cases.

**Advantages of Randomized Quicksort:** The random pivot selection eliminates the dependency on input order, ensuring robust average-case performance.

Discrepancies: Minor deviations in running time may arise due to implementation-specific details, such as overhead from random number generation or memory access patterns.

**Analysis of Hash Tables**

**Performance Metrics**

Insert: Inserting six elements distributed the keys across the slots, with one collision handled by chaining (e.g., "banana" and "date" shared the same index).

Search: Searching for existing keys ("apple" and "fig") successfully retrieved their values in constant time. Searching for a non-existent key ("grape") returned None.

Delete: Deleting "banana" succeeded, while attempting to delete a non-existent key ("grape") returned False.

**Load Factor and Performance**

Initially, the load factor was 6/10 = 0.6, ensuring efficient operations.

After resizing, the load factor was reduced, ensuring chains remained short and performance remained optimal.

**Observations**

The dynamic resizing mechanism ensures the load factor remains below the threshold, avoiding performance degradation due to long chains.

The expected time complexity for each operation is O(1) on average.

Implementing a hash table using chaining demonstrates efficient handling of collisions, dynamic resizing to maintain a low load factor, and robust support for insert, search, and delete operations. The descriptive analysis confirms that the implementation meets the expected time complexity and handles collisions effectively.

**Source Code:**

The source code of both the Randomized quicksort and the Hash table is listed below:

<https://github.com/ImAsrith/MSCS532_Assignment3>

**References**

Introduction to Algorithms By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

Hash Tables - <https://www.geeksforgeeks.org/hash-table-data-structure/>

Quicksort using random pivoting - <https://www.geeksforgeeks.org/quicksort-using-random-pivoting/>