**Part 1**

Randomized Quicksort sorts an array by recursively partitioning it around a randomly chosen pivot. At each step, elements smaller than the pivot are placed in one subarray, and elements larger than the pivot go into another. The algorithm continues to sort these subarrays recursively. The randomness in pivot selection ensures balanced partitions on average, leading to efficient sorting.

**Analysis - Average-case time complexity of Randomized Quicksort**

1. **Breaking Down the Algorithm**

* In each step, the algorithm partitions the array into two subarrays and recursively sorts them.
* Partitioning the array involves comparing each element with the pivot, which takes O(n) time.
* The sizes of the subarrays depend on the pivot's position, which is random and the array is split into two equal parts on average.

1. **The Recurrence Relation**

Let T(n) represent the time taken to sort an array of size n. Partitioning the array takes O(n), and the two subarrays are sorted recursively. Since the pivot is chosen randomly, the expected sizes of the two subarrays are approximately equal. The recurrence relation for the runtime of randomized quicksort is given as T(n) = T(k) + T(n - k - 1) + O(n), where k is the size of the first subarray.

1. **Simplifying the Average Case**

On average, the pivot divides the array into two subarrays of approximately equal size k is close to n/2.

Substituting this into the recurrence:

T(n) ≈ 2T(n/2) + O(n).

This is a standard recurrence relation in divide-and-conquer algorithms, which solves to T(n) = O(n log n).

The algorithm performs log n levels of recursion because the array size halves at each level. At each level, the partitioning step processes all n elements.

Multiplying the number of levels (log n) by the work done per level (n) gives a total runtime of O(n log n). This gives us the runtime of O(n log n) for Randomized Quicksort because the random pivot selection ensures that, on average, the array is split into balanced subarrays. This leads to log n levels of recursion, with O(n) work done at each level for partitioning. The recurrence relation T(n) = 2T(n/2) + O(n) explains this behavior clearly, making Randomized Quicksort an efficient sorting algorithm for most inputs.

**Comparison**

**A screenshot of a computer

Description automatically generated**

1. **Randomly Generated Arrays**

* **Performance:** Both Deterministic and Randomized Quicksort show comparable performance, with execution times growing as the input size increases.
* **Explanation**: Randomly distributed data allows Deterministic Quicksort to select reasonably balanced partitions in most cases, achieving 𝑂(n log n). Randomized Quicksort achieves a time complexity of O(n log n) on average by using a randomly selected pivot, which typically results in balanced partitions.
* **Expected Behavior:** This matches theoretical expectations, as both algorithms are optimized for handling random data.

1. **Sorted Arrays**

* **Performance**: Randomized Quicksort significantly outperforms Deterministic Quicksort as input size increases.
* **Explanation**: Deterministic Quicksort always selects the last element as the pivot, creating unbalanced partitions (worst-case O(n2)) for sorted inputs. In contrast, Randomized Quicksort chooses a random pivot, leading to more balanced partitions and retaining O(n log n) complexity.
* **Expected Behavior**: Yes, the results align with expectations. Randomized Quicksort's robustness to input order ensures stable performance, while Deterministic Quicksort suffers due to its fixed pivot selection strategy.

1. **Reverse-Sorted Arrays**

* **Performance**: Randomized Quicksort generally performs better than Deterministic Quicksort, especially on average, because the random pivot selection helps avoid worst-case scenarios that can occur with poorly chosen pivots..
* **Explanation**: Due to the unbalanced partitions in Deterministic Quicksort we observe the runtime O(n2), similar to what we noticed in case of sorted arrays. Randomized Quicksort avoids this issue by randomly selecting pivots, ensuring balanced splits.
* **Expected Behavior:** The observed behavior matches theoretical predictions, as Randomized Quicksort is designed to mitigate the impact of unfavorable input distributions.

1. **Arrays with Repeated Elements**

* **Performance:** Both algorithms perform similarly, with Randomized Quicksort showing slightly higher execution times for larger inputs.
* **Explanation:** Deterministic Quicksort efficiently handles repeated elements by grouping equal values around the pivot. Randomized Quicksort, while also efficient, incurs additional overhead due to random pivot selection, resulting in slightly slower performance.
* **Expected Behavior**: Yes, the results are consistent with expectations. Both algorithms handle repeated elements well, but Randomized Quicksort's randomness introduces minor overhead.

**Part 2**

**A screenshot of a computer

Description automatically generated**

**Analysis**

**Expected Time for Search, Insert, and Delete**

For hash tables, when implemented correctly with a good hash function, the average time for operations like searching, inserting, and deleting is O(1). This is because:

* Calculating the hash value to locate the correct bucket takes constant time.
* In most cases, elements in each bucket are distributed evenly, so operations within a bucket are minimal.

However, in the rare case of too many collisions (many elements hashing to the same bucket), the operations can take longer. A good hash function and proper table management can prevent this.

**Effect of Load Factor on Performance**

The load factor measures how full the hash table is, calculated as the number of elements divided by the number of buckets.

* Low load factor: The table has plenty of empty buckets, so collisions are rare, and operations are consistently fast.
* High load factor: The table becomes crowded, leading to more collisions and slower operations as multiple elements end up in the same bucket.

To maintain efficiency, it’s essential to keep the load factor within an optimal range, usually below 0.75.

**Strategies for Maintaining Performance**

1. **Dynamic Resizing:**

* If the table becomes too full, the number of buckets is increased (often doubled), and all elements are rehashed into the new table. This prevents buckets from becoming overcrowded, keeping operations fast.

1. **Use a Good Hash Function:**

* A good hash function spreads elements evenly across buckets, minimizing collisions. It ensures that keys are distributed randomly, avoiding patterns that cause clustering.

1. **Prime Number for Table Size:**

* Using a prime number for the table size helps spread keys more evenly, as it reduces the chance of multiple keys ending up in the same bucket due to common factors.

1. **Chaining:**

* In cases where collisions occur, store multiple elements in the same bucket using a linked list or another structure. This ensures operations remain efficient, even when collisions happen.

**Conclusion**

Hash tables are known for their high efficiency due to their constant average time complexity for operations. However, maintaining good performance depends on managing the load factor, using a good hash function, and resizing the table dynamically when needed. These strategies ensure the table operates smoothly, even as the number of elements grows.