**Part 1: Randomized Quick Sort:**

Analysis:

The average-case time complexity of Randomized Quicksort is O(nlogn). This complexity arises from the fact that the algorithm recursively divides the array into smaller parts, with each division taking O(n) time on average, and the depth of recursion being O(nlogn).

The recurrence relation for Randomized Quicksort can be written as:

T(n)=T(k)+T(n−k−1) +O(n)

* At each recursive step, the algorithm performs a partitioning operation that involves scanning the array once.
* Partitioning requires O(n) time because every element in the current array segment must be compared with the pivot.

This contributes the O(n) term to the recurrence relation.

After partitioning:

* The array is divided into two subarrays:
* One contains elements smaller than or equal to the pivot.
* The other contains elements larger than the pivot.
* The sizes of these subarrays depend on the pivot's position k, which is chosen randomly.

The recurrence T(n) is therefore split into two smaller problems:

T(n)=T(k)+T(n−k−1) +O(n),

Where:

* T(k): Time to sort the left subarray of size k.
* T(n−k−1): Time to sort the right subarray of size n−k−1.
* O(n): Time for partitioning.

The random choice of pivot means that k (the size of the left subarray) is uniformly distributed between and n−1. While any split is possible, on average:

This average-case behavior creates a balanced recursion tree, where:

* Height of the tree: O (log n), because the array is halved at each level.
* Work at each level: O(n), because partitioning scans all elements once.

Comparison:

Comparison of Randomized and Deterministic Quicksort Performance Across Different Test Cases and Array Sizes.

A table with numbers and text

Description automatically generated

* In general, Randomized Quicksort demonstrates more consistent performance compared to Deterministic Quicksort across all test cases. For random inputs, both algorithms perform well, with running times increasing predictably as the array size grows, adhering to their theoretical time complexities. However, in edge cases like "Sorted" and "Reverse" inputs, Deterministic Quicksort struggles significantly due to its fixed pivot strategy.
* For the "Sorted" case, the performance of Randomized Quicksort remains relatively stable with growth close to O(n log n). In contrast, Deterministic Quicksort shows a dramatic increase in runtime as the array size grows, with runtimes escalating to over 1.8 seconds for size 10,000. A similar trend is observed for the "Reverse" test case, where Deterministic Quicksort suffers from unbalanced partitions, leading to an O(n²) runtime. For "Repeated" test cases, both algorithms exhibit slower runtimes, but Randomized Quicksort still outperforms Deterministic Quicksort due to its pivot randomization.

Discrepancy and Observations:

* While Randomized Quicksort adheres closely to its theoretical expectations, slight deviations occur in edge cases due to suboptimal random pivot choices. For Deterministic Quicksort, the empirical results align well with its theoretical limitations, showcasing its inefficiency with sorted and reverse inputs.
* In conclusion, Randomized Quicksort is generally more robust and versatile, while Deterministic Quicksort is vulnerable to poor input configurations, reinforcing the importance of pivot selection in quicksort algorithms.

**Part 2: Hashing with Chaining:**

Analysis:

The load factor (𝛼), defined as the ratio of the number of elements to the number of slots in a hash table, has a significant impact on performance. When the load factor is low (α<1), the chains in the hash table are short, resulting in efficient search, insert, and delete operations. Fewer collisions occur, which keeps the chain lengths minimal and maintains optimal performance. Conversely, when the load factor is high (α>1), the chains become longer due to more collisions. This increases the time required for search and delete operations, with performance potentially degrading to O(n) in the worst case, such as when all keys hash to the same bucket.

Search Operation:

Expected Time Complexity: O(1+α)

* α is the load factor, defined as​ α = where n is the number of elements in the hash table, and m is the number of slots (buckets).
* Under simple uniform hashing, keys are uniformly distributed across the buckets, so the expected chain length is approximately α. Therefore, searching for a key involves computing the hash function (constant time) and scanning through the chain (linear in chain length).

Insert Operation:

Expected Time Complexity: O (1)

* The insertion requires computing the hash value and appending the key-value pair to the appropriate chain. Even with collisions, insertions remain efficient, as we typically append to the end of the chain.

Delete Operation:

Expected Time Complexity: O(1+α)

* Deletion requires finding the key in the chain (same as search) and removing it, which involves scanning a chain of length proportional to 𝛼.

Effect of Load Factor on Performance:

The load factor α directly impacts performance:

* Low Load Factor (α<1):
* Chains are short, leading to efficient search, insert, and delete operations.
* Fewer collisions occur, minimizing chain length.
* High Load Factor (α>1):
* Chains grow longer, increasing the time for search and delete operations.
* Performance degrades towards O(n) in the worst case (if all keys hash to the same bucket).

Strategies to Maintain a Low Load Factor:

* Maintaining a low load factor is essential for ensuring the efficiency of hash table operations. One key strategy is dynamic resizing of the hash table. When the load factor (α) exceeds a predefined threshold (e.g., 0.75), the table size is increased, typically doubling it. All existing keys are rehashed into the new table, ensuring a more uniform distribution of keys. While resizing involves an O(n) operation, this cost is amortized across multiple insertions.
* Another approach is to use a universal hash function, which minimizes collisions by selecting a hash function from a family of functions designed to provide a uniform key distribution. Load balancing is also critical—regularly monitoring the load factor and redistributing keys if chains become disproportionately long can improve performance even before resizing becomes necessary.
* For collision resolution, separate chaining can be optimized by using data structures like balanced binary search trees (e.g., AVL trees) in place of linked lists. This reduces chain lookup complexity to O(log(chain length)).