Part 1: Randomized Quicksort Analysis

1. Implementation

Randomized Quicksort algorithm

- Arrays with repeated elements:

  The current implementation handles repeated elements correctly. The partition scheme used allows elements equal to the pivot to be placed on either side, which prevents infinite recursion for arrays with many duplicates.

- Empty arrays:

  The implementation handles empty arrays correctly. The base case if len(arr) <= 1: return arr ensures that empty arrays are returned as-is without further processing.

- Already sorted arrays:

  The current implementation does not have any specific optimizations for already sorted arrays. In fact, if the pivot is always chosen as the last element (which is the case here), already sorted arrays will lead to the worst-case time complexity of O(n^2).

2. Analysis

Average-Case Time Complexity Analysis

- Choosing the Pivot:

  In Randomized Quicksort, the pivot is chosen randomly. This randomness helps ensure that we don't consistently pick poor pivots (like the smallest or largest element), which can lead to unbalanced partitions.

- Partitioning:

When we partition the array around the pivot, we expect that the pivot will split the array into two roughly equal parts on average. This means that if we have n elements, we will have two subarrays of about n/2 elements each after one partitioning step.

- Recursion:

  The process of partitioning and sorting the subarrays continues recursively. If the array is split into two halves, the recurrence relation for the time complexity can be expressed as:

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

  Here, O(n) represents the time taken to partition the array.

- Solving the Recurrence:

  This recurrence relation can be simplified to: T(n)=2T(n/2)+O(n)$T(n)=2T(n/2)+O(n)$. Using the substitution method, we find that this resolves to $T(n)=O(n\log n)$. The log$n$ factor comes from the number of times we can split the array in half until we reach a base case (when the array size is 1).

Conclusion

The average-case time complexity of Randomized Quicksort is $O(n\log n)$. This means that, on average, the algorithm will take time proportional to n times the logarithm of n to sort an array of size n. The random selection of the pivot helps ensure that the partitions are balanced most of the time, preventing the worst-case scenario, i.e., O(n^2), from happening frequently. In summary, Randomized Quicksort achieves an average-case time complexity of $O(n\log n)$ because it effectively splits the array into smaller parts and sorts them recursively, with the randomness in pivot selection helping to maintain balance in the partitions.

3. Comparison

The empirical results from the quicksort implementations demonstrate a clear distinction in performance between Randomized Quicksort and Deterministic Quicksort across various input types. For Randomized Quicksort, the observed running times align well with the theoretical expectation of O(n log n) average-case complexity. This consistency is evident across all input types, including random, sorted, reverse-sorted, and arrays with repeated elements. The randomized pivot selection effectively maintains balanced partitions, ensuring that the algorithm performs efficiently regardless of the initial order of the array.

In contrast, Deterministic Quicksort exhibits a significant performance degradation when handling sorted and reverse-sorted arrays, particularly as the array size increases. While it performs comparably to the randomized version on random arrays for smaller sizes, the time complexity escalates to O(n^2) for sorted inputs due to unbalanced partitions. This behavior is consistent with theoretical expectations, as the deterministic approach, which likely uses the last element as the pivot, leads to poor performance when the input is already sorted or nearly sorted.

The observed discrepancies in running times can be attributed to several factors. For smaller arrays, the overhead of random number generation in Randomized Quicksort may result in slightly longer execution times than in Deterministic Quicksort. However, as the array size increases, the advantages of randomization become more pronounced, with Randomized Quicksort consistently outperforming its deterministic counterpart on sorted and reverse-sorted arrays. Additionally, both algorithms perform well on arrays with repeated elements, benefiting from efficient partitioning due to fewer unique comparisons.

In conclusion, the empirical analysis largely supports the theoretical framework surrounding the performance of both quicksort implementations. Randomized Quicksort demonstrates robust performance across all input types, confirming its average-case O(n log n)

complexity. Meanwhile, Deterministic Quicksort's vulnerability to sorted and reverse-sorted inputs highlights its worst-case O(n^2) behavior, reinforcing the importance of pivot selection in quicksort algorithms. The slight discrepancies observed, particularly for smaller array sizes, can be attributed to implementation-specific factors and constant overheads not captured in asymptotic analysis.

Part 2: Hashing with Chaining

1. Implementation

Gitlab link for hashTable.py:

https://github.com/ppradhancodes/MSCS532_Assignment3/blob/main/hashTable.py

2. Analysis

Expected Time Complexities

Search: O(1 + α)

Insert: O(1 + α)

Delete: O(1 + α)

Where α is the load factor (number of elements/number of slots).

Impact of Load Factor:

The load factor α significantly affects performance:

- As α increases, the average chain length grows, increasing operation times.
- When α is low (< 1), operations approach O(1) time.

- When α is high (> 1), performance degrades towards O(n) in the worst case.

For example, with α = 0.5, we expect chains of average length 0.5, resulting in fast operations. With α = 2, the average chain length is 2, doubling the expected operation time.

Strategies for Maintaining Low Load Factor

Maintaining a low load factor and minimizing collisions are crucial for optimal hash table performance. The primary strategy is dynamic resizing, where the table size is increased (typically doubled) when the load factor exceeds a predetermined threshold, such as 0.75. This process involves creating a larger array and rehashing all existing elements, which, while costly, pay off over many operations to maintain overall efficiency. Choosing an appropriate initial size based on the expected element count can delay the need for early resizing. Employing a high-quality hash function that uniformly distributes keys across the table is essential for minimizing collisions. Some implementations use prime number table sizes to improve key distribution, especially with certain hash functions. Alternative collision resolution techniques like open addressing might be considered in memory-constrained environments. Using balanced trees instead of linked lists for long chains can improve worst-case performance for separate chaining. When implemented together, these strategies help maintain a low load factor, keeping the average-case time complexity close to O(1) for all operations, although at the cost of increased memory usage and occasional resizing operations.