Assignment 3: Understanding Algorithm Efficiency and Scalability

Algorithms and Data Structures (MSCS-532-B01)

Shreya Sapkota

005026644

11/10/2024

Randomized Quicksort Algorithm

Randomized quicksort is a variation of the Quicksort algorithm using a random pivot element. A traditional quicksort using a pivot element splits an array into subarrays, sorting each one recursively until the entire array is in order. While a randomized quicksort selects the pivot at random from the current subarray rather than at a predetermined location. By preventing problems like the worst-case time complexity that traditional Quicksort encounters with specific inputs, this randomization aids in obtaining consistent performance across a variety of input types.

Average-Case Time Complexity O(nlogn)

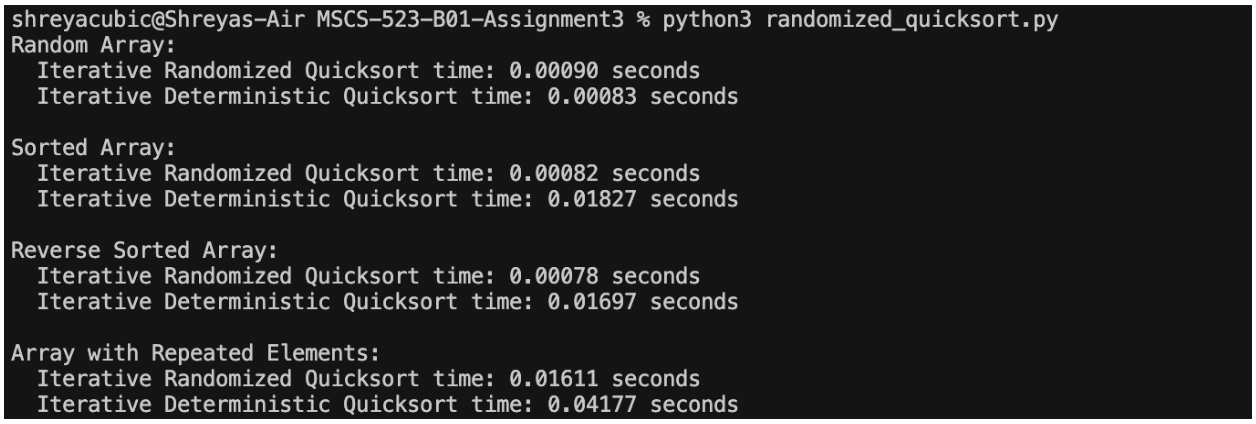
Randomized Quicksort splits the array into two subarrays in each recursive step by partitioning it around a randomly selected pivot. The time complexity for sorting an array of size n is denoted by T(n). We get the recurrence by assuming that the pivot splits the array roughly in two halves:

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

where O(n) represents the time for partitioning. Using the Master theorem, this recurrence relation solves to T(n)=O(nlogn).

Randomness in pivot selection is important for achieving O(nlogn) performance since it reduces the possibility of poor partitions. By giving each element an equal chance of being chosen as the pivot, randomization avoids skewed divisions, such as those that occur in sorted or reverse-sorted arrays in conventional Quicksort with a fixed pivot. In most situations, this method results in balanced partitioning, making the total effort across all levels O(nlogn) and the recursive depth O(logn).

Comparison of Randomized Quicksort with Deterministic Quicksort



```
shreyacubic@Shreyas-Air MSCS-523-B01-Assignment3 % python3 randomized_quicksort.py
Random Array:
  Iterative Randomized Quicksort time: 0.00090 seconds
  Iterative Deterministic Quicksort time: 0.00083 seconds

Sorted Array:
  Iterative Randomized Quicksort time: 0.00082 seconds
  Iterative Deterministic Quicksort time: 0.01827 seconds

Reverse Sorted Array:
  Iterative Randomized Quicksort time: 0.00078 seconds
  Iterative Deterministic Quicksort time: 0.01697 seconds

Array with Repeated Elements:
  Iterative Randomized Quicksort time: 0.01611 seconds
  Iterative Deterministic Quicksort time: 0.04177 seconds
```

We found that randomized quicksort and deterministic quicksort performed significantly differently after executing the code in different scenarios. For a random array, the deterministic quicksort took 0.00083 seconds, whereas a randomized quicksort took 0.00090 seconds. Given that both algorithms often manage random distributions effectively, this is to be expected. While deterministic quicksort dropped considerably to 0.01827 seconds, randomized quicksort maintained a fast speed of 0.00082 seconds for a sorted array. Deterministic quicksort's tendency to select the first element as a pivot leads to imbalanced partitions in already sorted data, resulting in O(n 2) performance.

Similar results were observed in the reverse sorted array, randomized quicksort finished in 0.00078 seconds, however deterministic quicksort took significantly longer at 0.01697 seconds. Finally, randomized quicksort outperformed deterministic quicksort by a significant margin, taking 0.01611 seconds as opposed to 0.04177 seconds for an array with repeated elements. This implies that by using random pivot selection to prevent worst-case partitions, randomized quicksort manages duplicate cases more well. All things considered, our investigation shows that randomized quicksort is reliable and consistent with a wide range of inputs, particularly for sorted and repeated components where performance decrease may result from deterministic selections.

Hashing with Chaining

A collision-resolution technique in hash tables is hashing with chaining, in which every slot refers to a chain of linked components that map to the same hash index. Each index can carry many items if necessary since when an element hashes to an occupied slot, it is simply added to the linked list at that position.

Time Complexity of Operations

We can examine the anticipated timings for each operation if we assume simple uniform hashing, in which items have an equal chance of hashing to any slot. The average number of items per slot is represented by the load factor $\alpha = n/m$ for n elements and m slots.

- **Insert**: Since the additional element is introduced at the end of the chain, the time complexity is $O(1)$
- **Search** and **Delete**: The majority of chains will be short and the search will typically take $O(1)$ time if α is low, or near 1. The length of each chain, however, rises with α, resulting in an estimated time complexity of $O(1+\alpha)$.

Strategies to Maintain a Low Load Factor

On average, $O(1)$ operations are ensured by short chains with a low load factor ($\alpha = 1$). Longer chains with a high load factor ($\alpha \gg 1$) result in slower operations at $O(\alpha)$. The following technique ensures that, even as the dataset expands, hash tables with chaining can continue to perform effective $O(1)$ operations:

1. Dynamic Resizing: Resizing and rehashing elements maintain α low and increase efficiency when the load factor surpasses a threshold.
2. Effective Hash Function: A well-designed hash function distributes elements evenly and reduces clustering.
3. Effective Data Structures: Although it adds complexity, using balanced trees for chains might enhance performance in high-collision situations.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

GeeksforGeeks. (n.d.). *Hashing with chaining*. Retrieved November 6, 2024, from https://www.geeksforgeeks.org/hashing-with-chaining/

Khan Academy. (n.d.). *Hashing and hash tables*. Retrieved November 6, 2024, from https://www.khanacademy.org/computing/computer-science/algorithms/hash-tables/a/hash-tables