**Avijit Saha**

**Student ID #005023281**

**Algorithms and Data Structures (MSCS-532-B01)**

**Assignment 3**

**November 10, 2024**

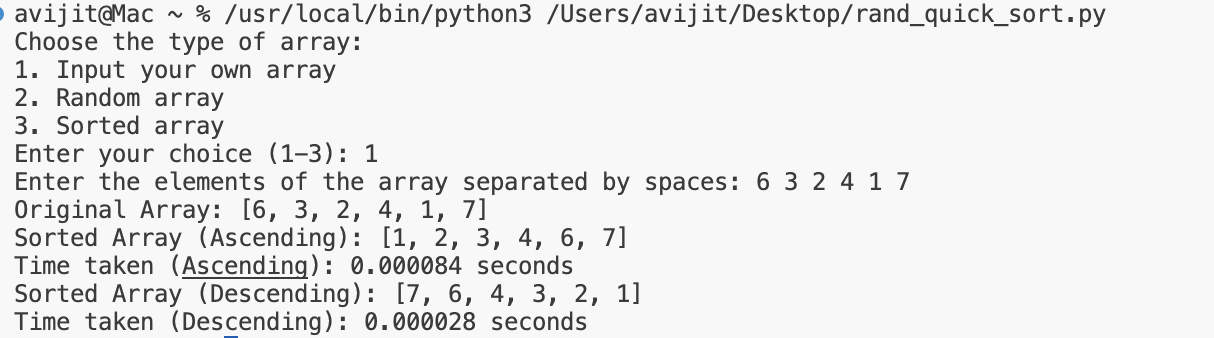
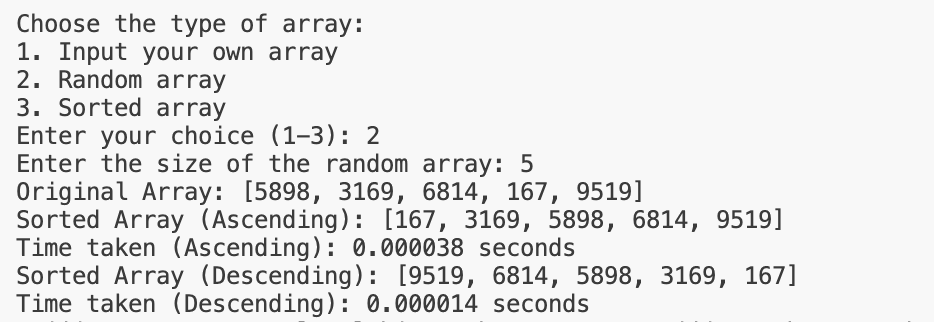
**Part 1**

**Randomized Quick Sort:** The average-case time complexity of a randomized quick sort is O(n log n). Each call divides the array or subarray into two parts by selecting a pivot and rearranging elements. All elements less than or equal to the pivot are on the left, and the more significant elements go on the right side. This is the partitioning, and it takes linear time O(n), where n is the number of elements. Randomized Quicksort avoids the worst case by randomly selecting the pivot, which, in expectation, leads to roughly balanced splits. This means that, on average, each partition splits the array into two parts of approximately equal size. With an average-case split, the recursion tree's recursive levels or depth is proportional to log n. Each level of recursion roughly halves the subarray size. Therefore, with n elements, it takes approximately log n splits to reach subarray size 1, which forms the base case of recursion.

**A close-up of a computer code

Description automatically generated**Since the partitioning steps at each level of recursion take O(n) time, and we have O(log n) levels of recursion, the total time complexity is O(n log n) for average-case time complexity. It minimizes the chance of unbalanced partitioning by selecting the pivot randomly, which leads to balanced recursion depth on average.

**Figure 1:** Random Pivot Choosing by the random function



**Figure 2:** Sorting for Random Array vs Given Unsorted array

From the analysis, I have found that randomized quick sort takes almost similar times for all cases. It is faster for random arrays. In all cases, reverse sorting takes less time than ascending sorting.

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated**Comparison:**

**Figure 3:** Randomized Time Taken. **Figure 4:** Deterministic Time Taken

I have found some interesting data from the comparison with the deterministic quick sort. It takes less time than the randomized quick-sort in all cases, such as randomly generated arrays, already sorted arrays, reverse sorted arrays, and arrays with repeated elements.

A close up of a number

Description automatically generated

**Figure 5:** Deterministic with Repeated Data

But, when I compare it with itself with repeated data, it takes more time than before. It is because of the worst case where O(n^2) applies. For randomized, the time is still more than the deterministic quick sort, but the difference is not less than the deterministic quick sort. It is because of the average case of time complexity. It chooses the pivot randomly to reduce the time complexity. My analysis shows that deterministic quick sorting takes less time than randomized. But if the size of the array is different or if it has a different structure, such as a repeated array or already sorted array, then it can be time-consuming. So, to be safe, a randomized quick sort is chosen in most cases.

**Part 2**

**Hashing with Chaining:** In hashing, a hash function maps the key to some values. However, these hashing functions may lead to a collision in which two or more keys are mapped to the same value. Chain hashing avoids collision. The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value.

**Expected Time Complexity**

* **Insert:** To insert a node into the hash table, we must find the hash index for the given key. It could be calculated using the hash function. The average-case time complexity for insert operation is O(1) because a new item is typically added at the beginning or end of a linked list at a given hash index. Uniform hashing assumes that keys are uniformly distributed across the table, so each chain's length is small on average. But for the worst case, it is O(n) if all keys hash to the same index.
* **Search:** In the average case with uniform hashing, the time complexity for searching an element is O(1). This is because each chain at a given index is short when keys are uniformly distributed, allowing quick access to any chain component. In the worst case, it is O(n) since the entire chain may need to be traversed.
* **Delete:** Like insertion and search, deletion also has an average time complexity of O(1). Removing an element from a chain only requires accessing the linked list node and adjusting pointers, which is fast if chains are short. In worse cases, it is similar to inserting and searching because we might have to traverse the entire chain to locate and remove the target element.

**Load Factor (α)**

The load factor α is a crucial metric for hash tables, defined as α = n/m where n is the total number of elements or keys in the hash table, and m is the number of buckets or slots in the hash table. It represents the average number of elements per chain and can indicate how full the hash table is.

When it is less than 1, it has fewer elements than the buckets, resulting in many empty or sparsely populated chains. This configuration results in very short chains, leading to efficient O(1) performance on average for insertion, search, and deletion.

When the load factor exceeds one, more elements are present than buckets, causing longer chains and more frequent collisions. As chains grow in length, operations can become slower. A high load factor can lead to a degradation in performance, trending towards O(n) in the worst case if n becomes significantly more extensive than m.

Maintaining an optimal load factor (often around α=1) is essential for balancing memory use and performance. Hash tables typically use resizing (doubling the number of buckets) when the load factor exceeds a certain threshold to maintain efficient operation by reducing the length of chains, thus keeping the average-case complexity close to O(1) for most operations.

Strategies to Maintain a Low Load Factor:

Dynamic Resizing: This is one of the common strategies to resize a hash table when a load factor exceeds a certain threshold, commonly 0.7. When resizing, create a new table with a larger size and rehash the existing elements into the new table.

Choosing an appropriate initial size: Estimate the number of elements you expect to store and set the initial size of the hash table accordingly to minimize resizing.

Using a Good Hash Function: A well-distributed hash function helps minimize collisions by ensuring that keys are spread across the available slots.

Load factor Threshold: Set a maximum load factor threshold that triggers resizing. This can help maintain efficiency.

The hash table implementation provided uses chaining for collision resolution and supports efficient insertion, searching, and deletion operations. The average time complexity is O(n). Implementing strategies like dynamic resizing, selecting an appropriate hash function, managing collision, and maintaining efficient operation can be implemented effectively.

**Reference:**

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The design and analysis of computer algorithms*. Addison-Wesley.

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

Morin, P. (n.d.). *Open data structures*. Retrieved from <http://opendatastructures.org/>

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

Weiss, M. A. (2013). *Data structures and algorithm analysis in C++* (4th ed.). Pearson.