**Part1: Randomized Quicksort Analysis**

**1. Implemented code is already attached.**

**2. Analysis of the Average-Case Time Complexity of Randomized Quicksort**

To understand why the average-case time complexity of Randomized Quicksort is O(n), we need to analyze the recurrence relation and the expected behavior of the algorithm.

**Recurrence Relation**

For an array of length n, the average time complexity can be expressed using a recurrence relation:

T(n)=T(k)+T(n−k−1)+O(n)

Here:

* K is the number of elements on one side of the pivot.
* T(k) is the time complexity of sorting the left subarray.
* T(n−k−1) is the time complexity of sorting the right subarray.
* O(n) accounts for the partitioning process.

Since the pivot is chosen uniformly at random, on average, the pivot will divide the array into two subarrays of approximately equal size:

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

**Applying the Master Theorem**

The above recurrence relation can be solved using the Master Theorem for divide-and-conquer recurrences of the form T(n)=aT(n/b)+f(n)

In our case:

* a=2
* b=2
* f(n)=O(n)

The Master Theorem states that if f(n)=O() where c = , then:

* If c < , T(n)=O()
* If c = , T(n)=O()
* If c = , T(n)=O(f(n))

Since f(n)=O(n) and c=1 , and also we fall into the second case.

Therefore, the average-case time complexity of Randomized Quicksort is: T(n) = O(nlogn).

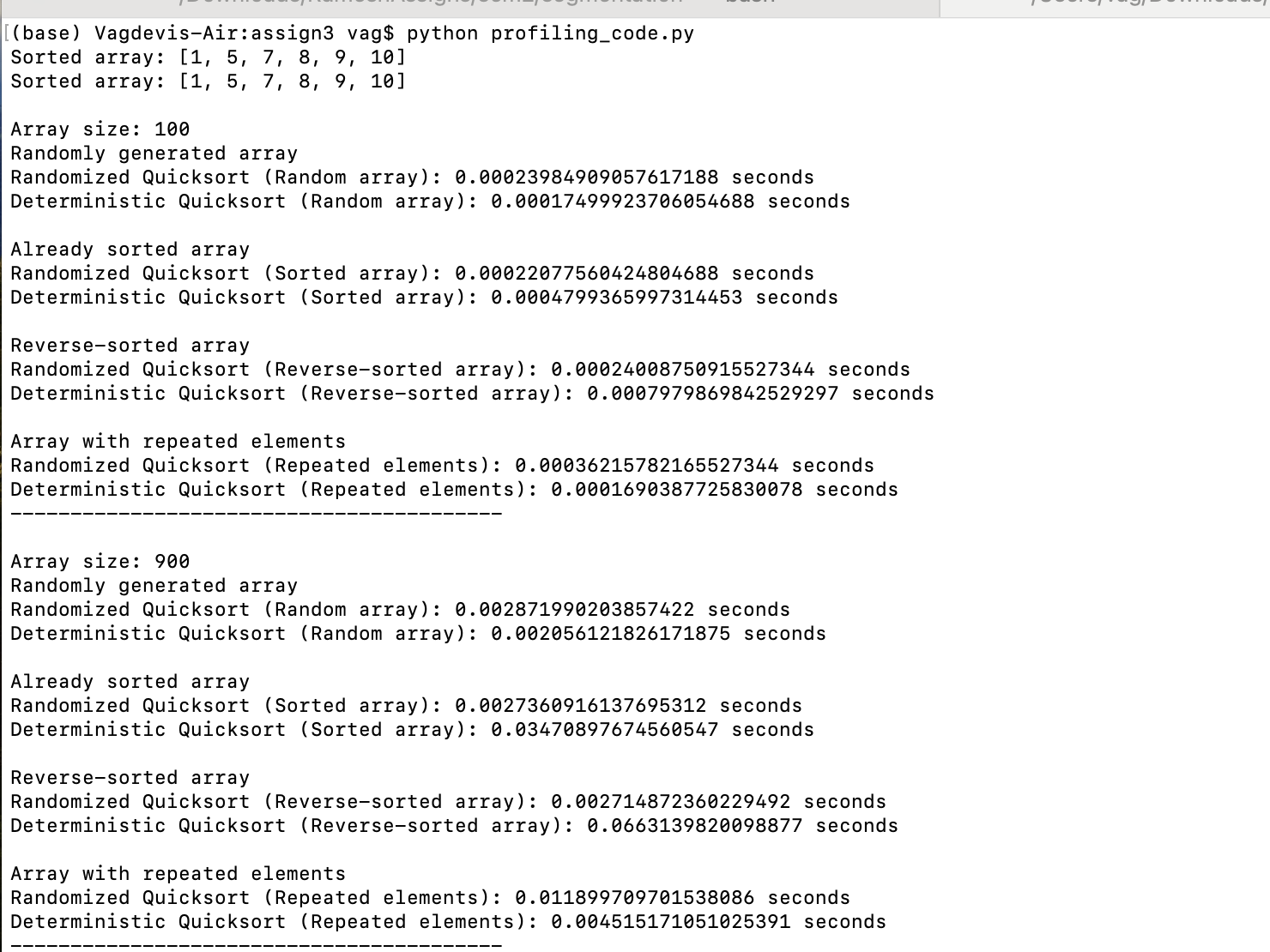
**3. Comparison:**

- Randomly generated arrays : Similar kind of performance by both algorithms

- Already sorted arrays : Randomized quicksort was better

- Reverse-sorted arrays : Randomized quicksort was better

- Arrays with repeated elements : Similar kind of performance by both algorithms



**Discussion of Observed Differences**

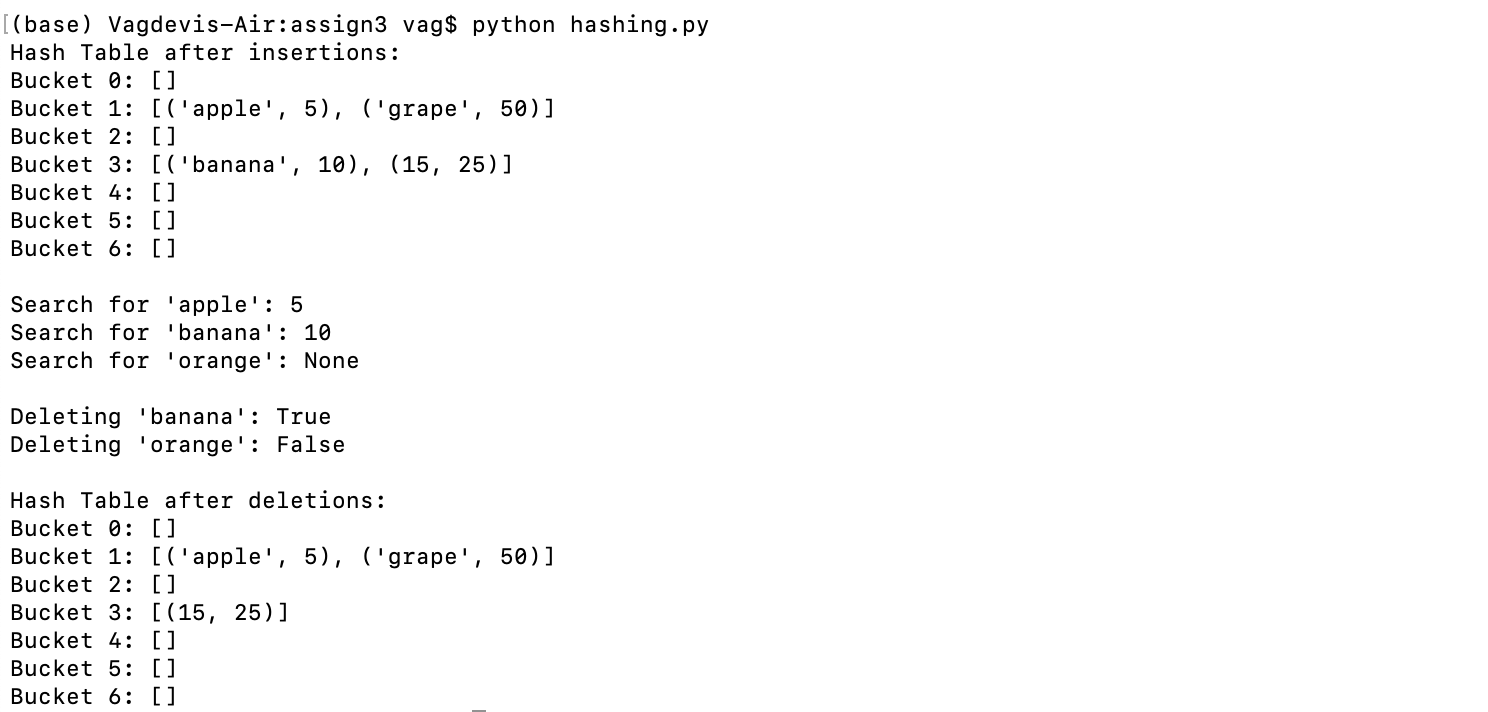
* **Randomly generated arrays**: Both algorithms should perform similarly, with Randomized Quicksort potentially having a slight edge due to its robustness against specific input patterns.
* **Already sorted arrays**: Randomized Quicksort will generally perform well because the pivot choice is random, preventing worst-case scenarios. Deterministic Quicksort, using the first element as a pivot, will perform poorly, degrading to O() time complexity.
* **Reverse-sorted arrays**: Similar to sorted arrays, Randomized Quicksort will perform well, while Deterministic Quicksort will again degrade to O().
* **Arrays with repeated elements**: Both algorithms should perform similarly, but Randomized Quicksort's performance can vary due to the random pivot selection.

**Explaining Discrepancies**

* **Empirical Results vs. Theoretical Performance**: In practice, Randomized Quicksort's average-case performance aligns with the theoretical O(n log n) complexity. However, deterministic choices of pivot (like always picking the first element) can lead to worst-case scenarios for specific input patterns (already sorted or reverse-sorted arrays), resulting in O() time complexity.
* **Randomness and Robustness**: The random selection of pivots in Randomized Quicksort ensures that it avoids consistently hitting worst-case scenarios, making it more robust across various input distributions. This leads to better average performance in practice compared to Deterministic Quicksort with a fixed pivot choice.

**Part2: Randomized Quicksort Analysis**

1. **Implemented code is already attached.**

****

The hash function is defined as:

h(k)=((a⋅k+b)mod  p) mod m

where:

* a and b are random integers chosen during initialization.
* p is a large prime number to reduce collisions.
* m is the number of buckets (hash table size).

1. **Analysis of Expected Search, Insert, and Delete Times (Simple Uniform Hashing)**

Under the assumption of **simple uniform hashing** (i.e., each key is equally likely to hash to any bucket, independent of other keys):

1. **Expected Search Time**:
   * The **expected search time** depends on the load factor α=n/m, where:
     + n = number of elements in the hash table.
     + m = number of buckets (slots) in the hash table.
   * If the key is present:
   * Tsearch=O(1+α) where O(1) accounts for hashing the key and accessing the bucket, and O(α) accounts for searching through the linked list in the bucket.
   * If the key is not present, the time is still O(1+α), as all entries in the bucket must be checked.
2. **Expected Insert Time**:
   * The **expected insert time** involves:
     + Computing the hash function (O(1)).
     + Appending the new element to the end of the linked list in the corresponding bucket (O(1)).
   * Therefore, the expected insert time is: Tinsert=O(1+α)
3. **Expected Delete Time**:
   * The **expected delete time** involves:
     + Computing the hash function (O(1)).
     + Searching for the key in the linked list (O(α)).
     + Removing the key-value pair once found (O(1)).
   * Therefore, the expected delete time is: Tdelete=O(1+α).

### **Effect of the Load Factor (α) on Performance**

The **load factor** (α=n/m ​) significantly affects the performance of hash table operations:

1. **Low Load Factor (**α≪1**):**
   * Each bucket contains very few elements, so search, insert, and delete operations approach O(1).
   * Memory usage is higher due to many empty buckets.
2. **High Load Factor (**α≫1**):**
   * More collisions occur, causing buckets to have longer chains.
   * Operations degrade to O(α), and in the worst case (all elements in one bucket), they become O(n).

To maintain efficient operations, it's critical to keep the load factor low (typically α≤1).

**Strategies to Maintain a Low Load Factor and Minimize Collisions**

#### 1. **Dynamic Resizing**

Dynamic resizing adjusts the size of the hash table based on the current load factor:

* **When to Resize:**
  + If the load factor exceeds a threshold (e.g., α>0.75), resize to reduce collisions.
  + Optionally, shrink the table when α\alphaα becomes too small (e.g., α<0.25).
* **How to Resize:**
  + Create a new hash table with more buckets (typically double the size).
  + Rehash all existing elements into the new table.

#### 2. **Use of Prime Numbers for Bucket Count**

Choosing m (the number of buckets) as a **prime number** reduces patterns that cause collisions.

#### 3. **Open Addressing (Alternative to Chaining)**

Instead of chaining, **open addressing** resolves collisions by storing all elements directly in the hash table. Techniques like **linear probing, quadratic probing,** or **double hashing** reduce clustering but require careful handling of load factors.