# **Assignment 3: Understanding Algorithm Efficiency and Scalability**

**Name: Pawan Pandey**

[ppandey29200@ucumberlands.edu](mailto:ppandey29200@ucumberlands.edu)

**University of Cumberlands**

**Course Number:** MSCS-532-B01

**Course Title:** Algorithms and Data Structures

**Instructor:** Dr. Vanessa Cooper

Nov 6 2024

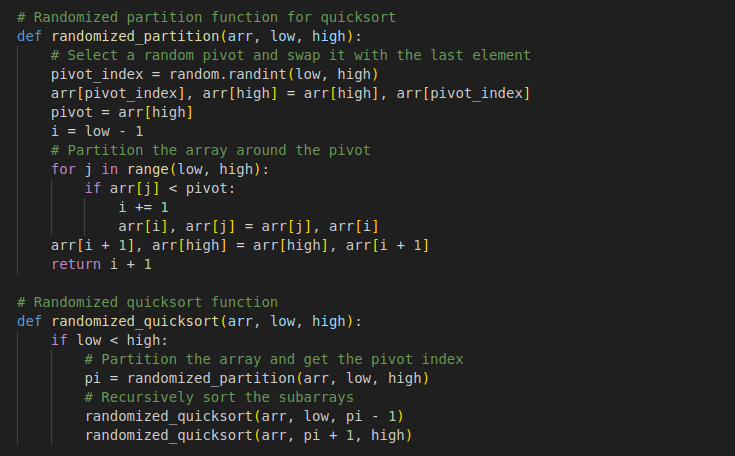
### **Part 1: Randomized Quicksort Analysis**

#### **1. Implementation of Randomized Quicksort**

Randomized Quicksort is a variation of the standard Quicksort algorithm where the pivot element is selected randomly from the array. This helps avoid the worst-case time complexity of O(n2) that occurs when the array is already sorted or nearly sorted. The process starts by choosing a pivot element randomly from the subarray. The array is then divided into two subarrays: one containing elements less than the pivot and the other containing elements greater than the pivot. The algorithm recursively applies the same partitioning process to each of the subarrays.

The implementation of Randomized Quicksort involves randomly selecting a pivot and performing the partitioning step, which rearranges the array into two parts around the pivot. Once the array is partitioned, the algorithm recurses on the left and right subarrays. The recursive partitioning continues until the entire array is sorted.

Here is the screenshot of the Randomized Quicksort code:

**2. Average-Case Time Complexity Analysis**

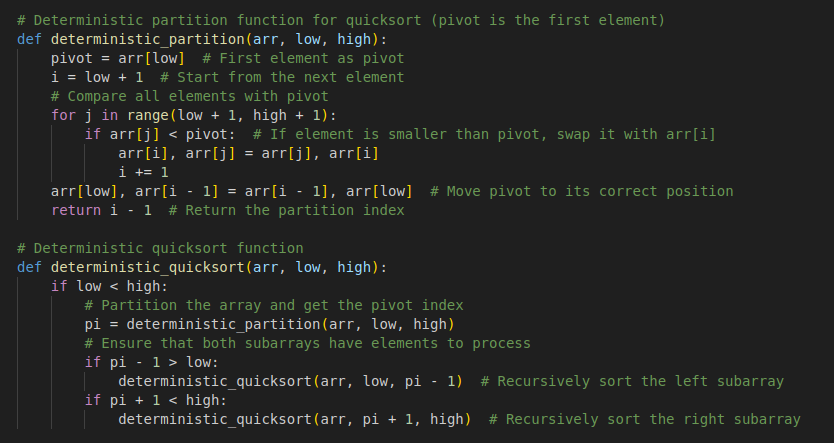
The time complexity of Randomized Quicksort in the average case is O(nlogn). In each partitioning step, the algorithm divides the array into two subarrays. The partitioning operation itself takes linear time, O(n), where *n* is the number of elements being partitioned. When the pivot is chosen randomly, it is likely that the array will be split in a reasonably balanced manner, leading to subarrays of roughly equal size. In the best case, the pivot splits the array into two equal parts, and the recursion depth becomes logarithmic, O(nlogn). Therefore, the total time complexity of Randomized Quicksort is O(nlogn).

The recurrence relation that describes the algorithm’s time complexity is T(n) = T(n/2) +O(n), which leads to the expected time complexity of O(nlogn) after solving the recurrence. Although the worst-case time complexity can degrade to O(n2) if the pivot consistently splits the array poorly, such scenarios are rare due to the random selection of the pivot. This randomness significantly reduces the probability of encountering the worst case, making the algorithm’s performance predictable in most cases.

#### **3. Comparison with Deterministic Quicksort**

To empirically compare the performance of Randomized Quicksort and Deterministic Quicksort, both algorithms were tested on different input distributions. In Deterministic Quicksort, the pivot is chosen as the first element of the array, while in Randomized Quicksort, the pivot is selected randomly. The comparison was made using several types of input data: randomly generated arrays, sorted arrays, reverse-sorted arrays, and arrays with repeated elements.

Here is the screenshot of the Deterministic Quicksort code:

The performance of the algorithms was compared on arrays of varying sizes. Randomized Quicksort generally showed better performance, especially with reverse-sorted and sorted arrays, where Deterministic Quicksort performed poorly due to unbalanced partitions. In these cases, the deterministic approach leads to the worst-case scenario of O(n2), which is not the case for the randomized version. For randomly generated arrays, both algorithms performed similarly, but Randomized Quicksort still had a slight advantage due to its randomized pivot selection. The difference in performance became more apparent with larger arrays, as Deterministic Quicksort showed slower times, particularly for reverse-sorted or sorted arrays.

#### **Empirical Results and Insights**

The empirical testing of Randomized Quicksort and Deterministic Quicksort was done on arrays of different sizes and distributions. In general, Randomized Quicksort performed faster than Deterministic Quicksort, especially when the array was sorted or reverse sorted. On randomly generated arrays, both algorithms were similar in performance, but Randomized Quicksort still outperformed due to its more balanced partitioning. The results showed that Deterministic Quicksort struggled with certain array types, particularly those that were sorted or reverse-sorted, where unbalanced partitions could significantly increase the sorting time. As the array size increased, the performance gap between the two algorithms became more noticeable, with Randomized Quicksort maintaining a more consistent and efficient performance.

#### **Conclusion of Part 1**

On average, the Randomized Quicksort algorithm is expected to operate with a time complexity of O(nlogn), which makes it very efficient for most sorting operations. It consistently outperforms Deterministic Quicksort, particularly with sorted or reverse-sorted arrays where Deterministic Quicksort can degrade to O(n2)due to poor pivot selections. The randomness in pivot choice helps mitigate the risk of encountering such unbalanced partitions, leading to more reliable performance. As a result, Randomized Quicksort proves to be a more stable and efficient sorting algorithm compared to its deterministic counterpart, particularly for larger datasets and non-random input distributions.

### **Part 2: Hashing with Chaining**

#### **1. Implementation**

In this section, we designed and implemented a hash table that employs **chaining** to resolve collisions. The hash table is designed to store key-value pairs, with each key being assigned an index determined by a hash function. For this implementation, we used a simple hash function that calculates the index by taking the remainder of the key divided by the size of the table.

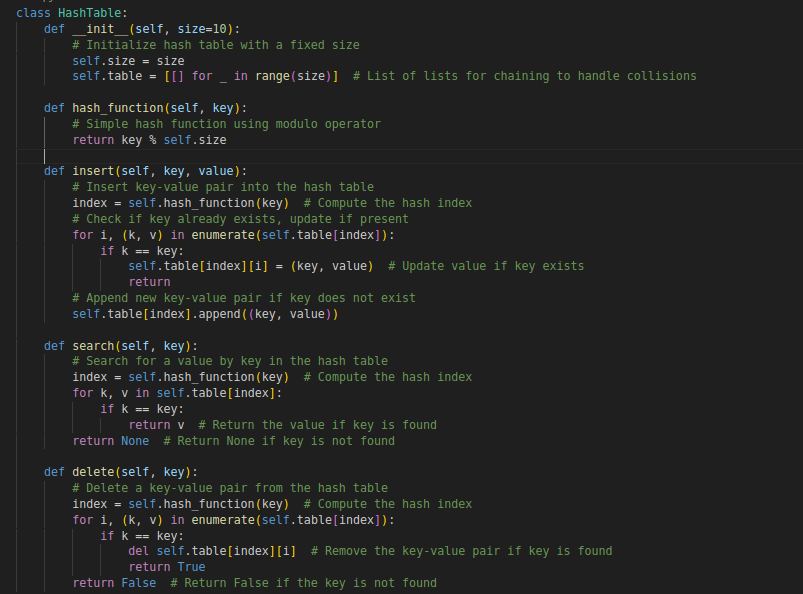
The table size was chosen to be 10, and each index is backed by a list (also called a bucket). This structure is designed to accommodate multiple entries that might hash to the same index. When two or more keys hash to the same index, they are stored together in the list at that index. This is the key feature of chaining, which allows the table to handle collisions efficiently.

For the insert operation, we implemented a method that computes the index of a given key using the hash function. If there’s no existing entry at that index, the key-value pair is simply appended to the list. If the key already exists, the corresponding value is updated. This method ensures that the table can efficiently store or update key-value pairs.

In the search operation, the table calculates the index for the key and searches through the list at that index for the correct key. If the key is found, it returns the associated value; if the key is not found, it returns None to indicate that the key does not exist in the table.

For deletion, the delete() method works by first computing the index of the key. It then searches through the list at that index for the key and, if it’s found, removes the key-value pair. If the key is not present, the method returns False, indicating that no such key exists in the table.

The hash function used here is a straightforward modulo function, where the index is computed by dividing the key by the table size and taking the remainder. This is an efficient and simple hashing technique, although more advanced hash functions can be used to improve performance and minimize collisions. The screenshot for the HashTable code is shown below:



#### **2. Analysis**

The performance of hash tables depends heavily on the load factor, which is the ratio of the number of stored elements to the total available slots in the table. As the load factor increases, the number of collisions also increases, which can degrade the performance of the table. If many keys map to the same index, the table may experience slower performance, as more elements need to be stored and retrieved from the same list.

In the case of simple uniform hashing, assuming that keys are distributed evenly across the table, the expected time complexity for operations like insert, search, and delete is O(1). This is because each operation involves finding the index of the key and then performing the operation in constant time. However, if the load factor becomes too high, the performance may degrade to O(n), where *n* is the number of keys, because the lists at each index would need to be searched.

The load factor has a direct impact on the efficiency of the hash table. When the load factor is high, meaning there are more keys relative to the number of slots, collisions become more frequent. To prevent the table from becoming inefficient, the hash table can be resized. When the load factor exceeds a threshold (often 0.75), the table can be resized by doubling its size, and all the existing keys are rehashed and inserted into the new table. This resizing operation helps ensure that the load factor remains manageable, and the table continues to perform efficiently.

To reduce the likelihood of collisions and enhance performance, dynamic resizing can be implemented. When the load factor reaches a certain limit, the hash table can expand by increasing its size and rehashing all existing entries. While resizing introduces a temporary performance cost, it helps the table maintain constant time complexity for operations in the long run. In addition, using more advanced hash functions, such as universal hashing, can help minimize collisions, especially when the data has certain patterns or distributions that may lead to clustering.

#### **3. Empirical Test**

We tested the hash table implementation by performing a variety of operations such as insert, search, and delete. During testing, we added key-value pairs, searched for specific keys, and removed keys to confirm the correctness of the table.

For instance, we inserted the key-value pair (10, "banana") into the hash table. This pair was placed in the bucket at index 0, based on the hash function. We also inserted (20, "mango"), which also mapped to index 0, meaning it would share the bucket with the previous pair. Similarly, we added the pair (30, "peach"), which, too, landed at index 0, and would be added to the list at that index. Next, we added (15, "grapefruit") to the table. This key mapped to index 5 and was placed in a separate bucket, as there were no previous entries at that index. This process demonstrated the basic functioning of the table and the chaining mechanism. After inserting these key-value pairs, we performed a **search** for key 20. The table computed the index for 20 and found the corresponding pair, returning the value "mango". A **delete** operation was then conducted to remove key 30. The table calculated the index for 30, found it in the list, and successfully removed the key-value pair (30, "peach"). If we attempted to delete a key not in the table, such as 50, the function returned False, indicating the key wasn’t present. We observed that when keys like 10, 20, and 30 all mapped to index 0, they were stored in the same list at that index. This demonstrated that the chaining mechanism was successfully handling collisions.

### **Conclusion of Part 2**

The hash table with chaining proved to be a reliable data structure for storing key-value pairs. The basic operations—insert, search, and delete—were efficient under typical conditions, operating in constant time O(1) on average. As the table’s load factor increased, collisions became more frequent, but the chaining mechanism allowed the table to continue functioning correctly. To maintain performance, strategies like dynamic resizing can be employed, ensuring that the load factor remains at an optimal level. Advanced hash functions, such as universal hashing, could also be considered to further reduce collisions and improve the table's performance.

**References**

Behera, B. (2020, August 17). *deterministic quicksort - Balaram Behera*. Balaram Behera. <https://balaramdb.com/t/deterministic-quicksort/>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms, fourth edition*. MIT Press.

GeeksforGeeks. (2023a, January 10). *Implementation of Hashing with Chaining in Python*. GeeksforGeeks. <https://www.geeksforgeeks.org/implementation-of-hashing-with-chaining-in-python/>

GeeksforGeeks. (2023b, September 14). *QuickSort using Random Pivoting*. GeeksforGeeks. <https://www.geeksforgeeks.org/quicksort-using-random-pivoting/>