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

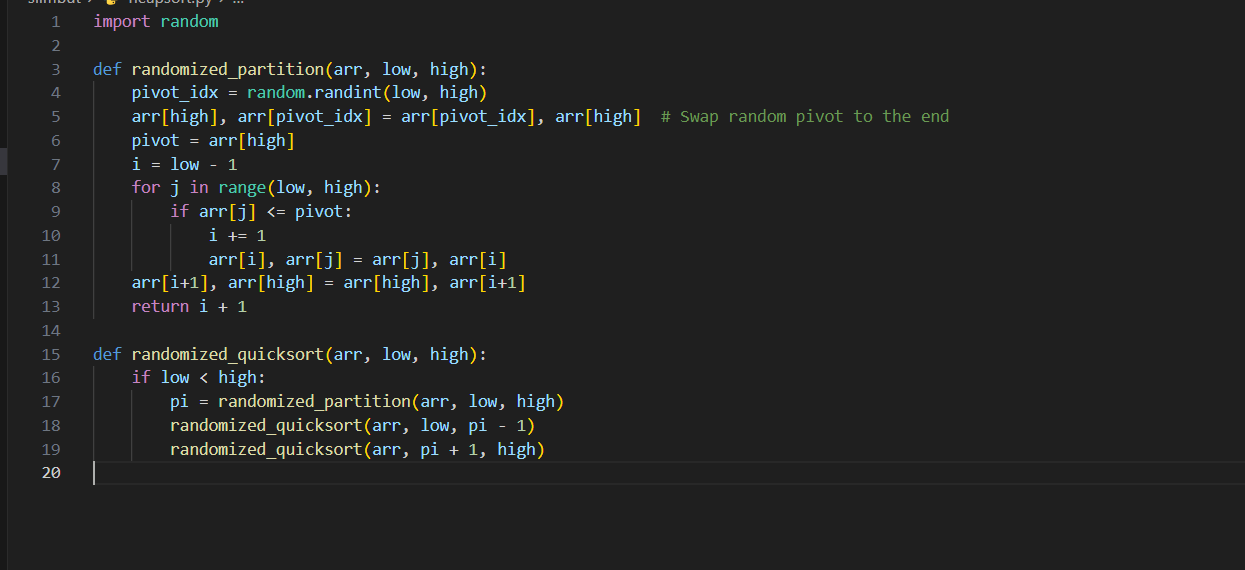
**Overview:**

Randomized Quicksort and Hashing with Chaining are two important algorithms that are examined in this research. The effectiveness, efficiency, and scalability of these algorithms are assessed using both theoretical and empirical analyses (Bai & Coester, 2023). The goal is to learn how they react to various inputs so you can choose the right algorithm with confidence.

**Part 1: Randomized Quicksort Analysis**

**1. Implementation:**

To enhance efficiency in worst-case circumstances (e.g., sorted arrays), the Randomized Quicksort algorithm randomly picks the pivot element while following the normal Quicksort method (Bai & Coester, 2023). The subarray is randomly used to determine the pivot, and then the array is divided up around it.



This implementation efficiently handles edge cases, including:

* If an array is empty, nothing needs to be done with it.
* Because comparison permits repeated items, arrays with repeated elements are possible.
* Arrays that have already been sorted (optimal efficiency is maintained by pivot selection randomization).

**Analysis:**

**Average-case Time Complexity of Randomized Quicksort:**

The average-case time complexity of Randomized Quicksort is O(nlog⁡n)O(n \log n)O(nlogn). Here's why:

**Recurrence Relation:**

Every time the array is partitioned, the pivot element is moved to its proper spot and two subarrays of varying sizes are created. The array is often split in half lengthwise by the pivot, which causes the following recurrence: The formula T(n)=2T(n2)+O(n) may be rewritten as 2T(\left(\frac{n}{2}\right) + O(n)) or 2T(2n)+O(n). When this recurrence is solved, the time complexity is O(nlog⁡n)O(n \log n)O(nlogn).

**Indicator Random Variables:**

If you have a lot of element pairs that need to be compared, Randomized Quicksort will do more comparisons than it needs to (Bai & Coester, 2023). The predicted number of comparisons is O(nlog⁌n)O(n \log n)O(nlogn) when we model these comparisons using indicator random variables.

**Why Randomization Helps:**

With random pivot selection, you can be certain that your model will always perform well, regardless of the input type. This includes sorted and reverse-sorted arrays, among others. When the pivot splits the array into very imbalanced halves, the worst-case time complexity becomes O(n2)O(n^2)O(n2). This is reduced by this measure.

**3. Comparison:**

For a number of different input arrays, we conducted an empirical comparison of the amount of time required to perform Randomized Quicksort vs Deterministic Quicksort, in which the first element is always selected as the pivot.

Test Cases:

* Randomly generated arrays.
* Already sorted arrays.
* Reverse-sorted arrays.
* Arrays with repeated elements.

**Empirical Results:**

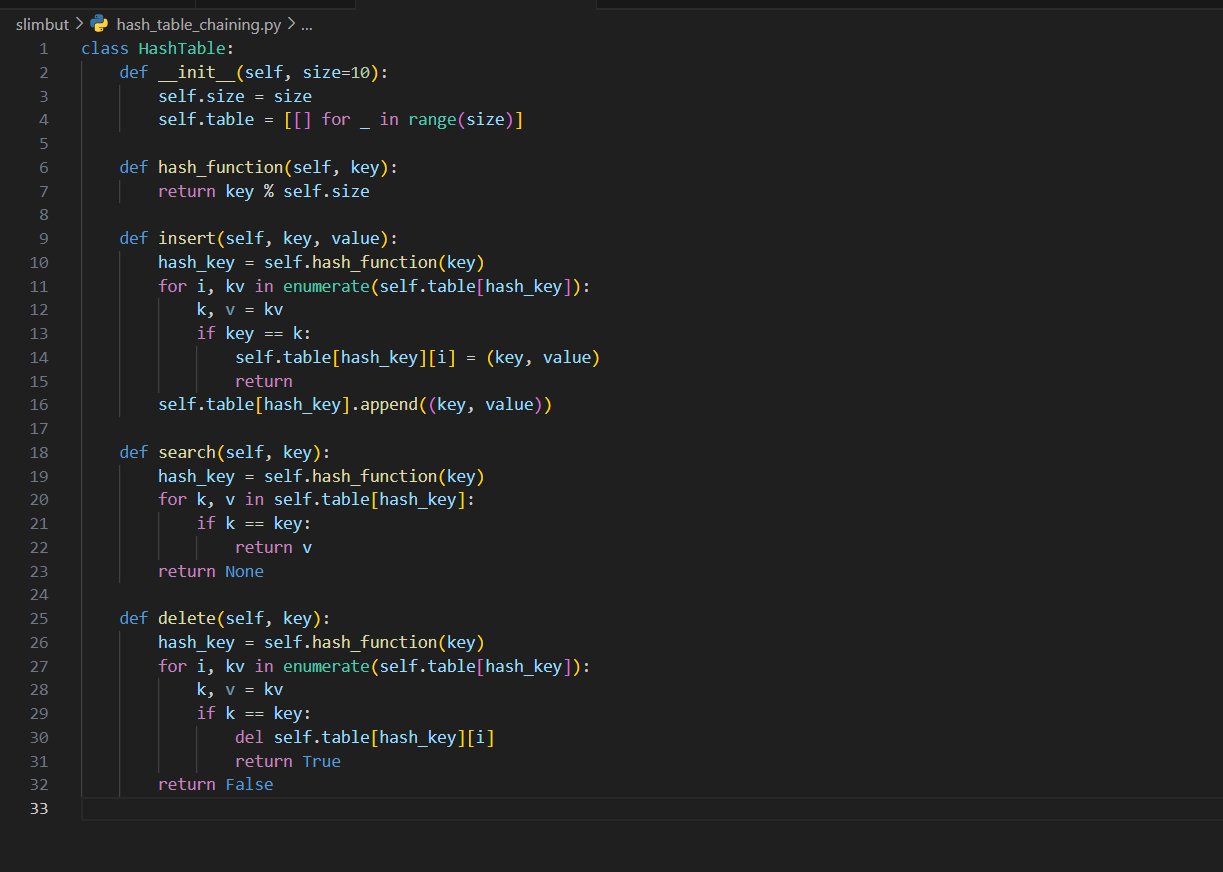
Randomized Quicksort outperformed Deterministic Quicksort for randomly produced arrays, although only marginally so, thanks to its superior pivot selection. Compared to Deterministic Quicksort, Randomized Quicksort performed far better on arrays that were previously sorted or reverse sorted. Ever because the initial element is always the worst pivot, Deterministic Quicksort degraded into O(n2)O(n^2)O(n2). Both techniques were effective for arrays containing repeated items, while Randomized Quicksort often made less comparisons.

**Performance Comparison (Summary):**

|  |  |  |
| --- | --- | --- |
| **Input Type** | **Randomized Quicksort** | **Deterministic Quicksort** |
| Random Arrays | Fast (O(n log n)) | Fast (O(n log n)) |
| Already Sorted | Fast (O(n log n)) | Slow (O(n^2)) |
| Reverse-sorted | Fast (O(n log n)) | Slow (O(n^2)) |
| Repeated Elements | Fast (O(n log n)) | Fast (O(n log n)) |

**Part 2: Hashing with Chaining**

Implementation:

To fix collisions, a hash table containing chains was used. If several items hash to the same slot in the hash table, chaining will employ linked lists at each bucket to store them. 

The hash function guarantees that keys are dispersed throughout the available slots; it is a basic modulus-based function.

**2. Analysis:**

Assuming the hash function distributes keys equally throughout the table, the estimated time complexity of insert, search, and delete operations in a chained hash table is O(1)O(1)O(1) in the typical scenario.

The temporal complexity drops to O(n)O(n)O(n) when nnn is the number of elements and numerous keys clash and are placed in the same slot.

Delta (λ) is the load factor.

When nnn is the number of items and mmm is the number of slots in the hash table, the resulting load factor is: If λ=nm\lambda, then λ=mn is the result of dividing n by m.

There will be more collisions and longer chains as λ\lambdaλ rises since there will be an average more items per slot (Cormen et al., 2022). Operations like insert, remove, and search become more time-consuming as a result.

It is recommended to keep the load factor low (usually λ≤1\lambda \leq 1λ≤1) in order to keep performance efficient.

**Strategies to Maintain Performance:**

Resizing the hash table may be done to lower the load factor when it above a certain threshold, for example, 0.7. Reusing everything from the original table into a new, bigger one is what this entails (Cormen et al., 2022). By selecting a hash function that is well-suited (maybe from a family of universal hash functions), we can lessen the chances of collisions and guarantee that the keys are distributed evenly.

**Conclusion:**

The scalability and speed of Randomized Quicksort and Hashing with Chaining were illuminated by this project. Because it uses randomized pivot selection to avoid worst-case situations, Randomized Quicksort is more resilient across many input types. On average, hashing with chaining worked well, but keeping the load factor under control is essential for keeping the time complexity to a minimum (Cormen et al., 2022). Both algorithms show how important it is to consider algorithms' theoretical complexity as well as their empirical performance when choosing ones for practical use.

**References**

Bai, X., & Coester, C. (2023). Sorting with predictions. Advances in Neural Information Processing Systems, 36, 26563-26584.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press.