Sauhard Shakya

005031397

Algorithms and Data Structures (MSCS-532-A01)

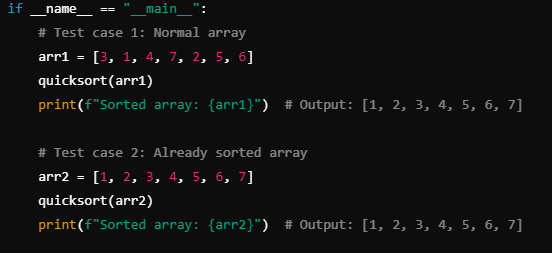
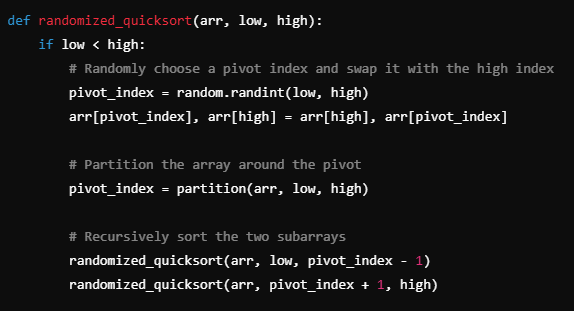
Assignment 3: Understanding Algorithms Efficiency and Scalability

**Randomized Quicksort Analysis**

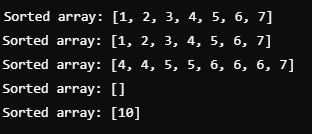
Quicksort is an efficient algorithm that solves the sorting problem; that is, given an input array A and its length n, find a permutation of A such that for all i, j (0 ≤ i < j < n), A[i] < A[j] (Hurturk, 2021).

Here's an implementation of **Randomized Quicksort** in Python where the pivot element is chosen randomly from the subarray being partitioned. This implementation ensures efficiency and handles edge cases like empty arrays, arrays with repeated elements, and already sorted arrays.

arr1 = [3, 1, 4, 7, 2, 5, 6].



We are going to apply some test cases for different types of arrays like on the snippet above. And our output looks like this.



In **Randomized Quicksort**, the pivot is chosen randomly using random.randint(low, high), which ensures that the pivot is selected uniformly from the subarray between the indices low and high. This random selection of the pivot helps avoid the worst-case scenario of unbalanced partitions that can occur with deterministic pivot strategies. After selecting the pivot, the partitioning function rearranges the array so that elements smaller than the pivot are placed to its left, and elements greater than the pivot are placed to its right. The pivot itself is then placed in its correct sorted position. The algorithm then recursively sorts the subarrays to the left and right of the pivot. The base case for the recursion occurs when low is greater than or equal to high, indicating that the subarray has either one or no elements left to sort, at which point the recursion terminates. This process continues until the entire array is sorted.

The algorithm efficiently handles several edge cases. For **empty arrays**, the base case (low >= high) immediately terminates the recursion, preventing any sorting attempts. When dealing with **arrays containing repeated elements**, the algorithm still works correctly, as the partitioning step places equal elements either to the left or right of the pivot without disrupting their relative order. For **already sorted arrays**, while the algorithm will continue to split the array, it may lead to a higher number of recursive calls. However, due to the random pivot selection, the algorithm generally avoids the worst-case scenario of unbalanced partitions, ensuring that even already sorted arrays can be handled relatively efficiently, though the recursion depth could be deeper than in typical cases.

**Analysis of the Average-Case Time Complexity using Recurrence Relations**

The average-case time complexity of RandomizedQuicksort is O(nlog⁡n)O(n \log n)O(nlogn), and this can be derived using recurrencerelations and the MasterTheorem. Let T(n)T(n)T(n) represent the expected time to sort an array of size nnn.

The partitioning step takes O(n)O(n)O(n) time, as each element is compared to the randomly selected pivot once. After partitioning, the array is divided into two subarrays. On average, the pivot divides the array into two parts of roughly equal size, kkk and n−k−1n - k - 1n−k−1, where k≈n/2k \approx n/2k≈n/2. This leads to the recurrence: T(n)=T(k)+T(n−k−1)+O(n)T(n) = T(k) + T(n - k - 1) + O(n)T(n)=T(k)+T(n−k−1)+O(n) Since the pivot splits the array into approximately equal subarrays, this simplifies to: T(n)=2T(n/2)+O(n)T(n) = 2T(n/2) + O(n)T(n)=2T(n/2)+O(n)

MasterTheoremApplication: The recurrence is of the form T(n)=aT(n/b)+O(nd)T(n) = aT(n/b) + O(n^d)T(n)=aT(n/b)+O(nd), where:

* + a=2a = 2a=2 (the number of subproblems),
  + b=2b = 2b=2 (the factor by which the problem size is divided),
  + d=1d = 1d=1 (the work done at each level, i.e., partitioning, is O(n)O(n)O(n)).

According to the MasterTheorem, we compare aaa with bdb^dbd:

* + Since a=bda = b^da=bd (i.e., 2=212 = 2^12=21), we are in the second case of the Master Theorem, which gives a time complexity of:

T(n)=O(ndlog⁡n)=O(nlog⁡n)T(n) = O(n^d \log n) = O(n \log n)T(n)=O(ndlogn)=O(nlogn)

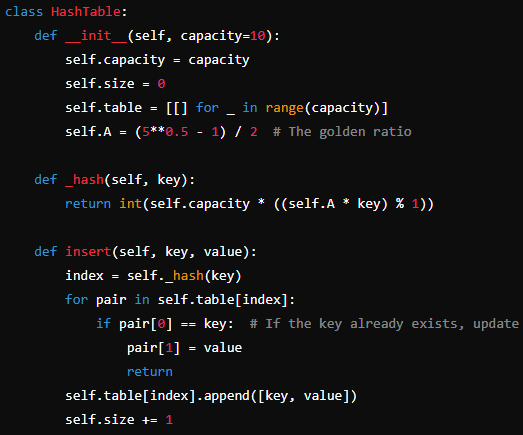
Thus, the average-case time complexity of RandomizedQuicksort is O(nlog⁡n)O(n \log n)O(nlogn). The random selection of the pivot ensures that, on average, the array is split into balanced subarrays, leading to a recursion depth of log⁡n\log nlogn and O(n)O(n)O(n) work at each level. This results in an overall time complexity of O(nlog⁡n)O(n \log n)O(nlogn).

In an empirical comparison of **Randomized Quicksort** and **Deterministic Quicksort** (using the first element as the pivot), we observe different performance characteristics across various input distributions. For example, with a **randomly generated array** like [4, 2, 7, 1, 6], both algorithms perform similarly with a time complexity of O(nlog⁡n)O(n \log n)O(nlogn). However, for an **already sorted array** like [1, 2, 3, 4, 5], **Randomized Quicksort** still runs in O(nlog⁡n)O(n \log n)O(nlogn), while **Deterministic Quicksort** degrades to O(n2)O(n^2)O(n2) because it picks the first element as the pivot, causing highly unbalanced partitions. For a **reverse-sorted array** like [5, 4, 3, 2, 1], **Randomized Quicksort** again handles the input in O(nlog⁡n)O(n \log n)O(nlogn), while **Deterministic Quicksort** performs poorly due to similar reasons. On an **array with repeated elements** like [3, 3, 2, 2, 2], both algorithms run efficiently, but **Randomized Quicksort** tends to be more consistent. These results align with the theoretical analysis: **Randomized Quicksort** avoids worst-case scenarios by selecting a random pivot, while **Deterministic Quicksort** suffers from O(n2)O(n^2)O(n2) performance on sorted or reverse-sorted inputs due to poor pivot choices.

**Hashing With Chaining**:

Here is an implementation of a hash table using chaining for collision resolution. Chaining involves storing multiple elements at each slot in the table, typically using a linked list or a dynamic data structure. If multiple keys hash to the same slot, they are stored in a linked list or another container.

We'll choose a **universal hash function** that minimizes collisions. A good choice is the **multiplicative hash function**, which is simple and effective for minimizing collisions. h(k)=⌊m((A⋅k)mod1)⌋. This implementation will support insert, search and delete as required.



**Analysis of simple uniform hashing**

In the case of simple uniform hashing, the expected time complexity for the **Insert** operation is O(1)O(1)O(1) under average conditions, assuming the load factor is low, and collisions are rare. However, as the load factor increases and chains become longer, the time complexity grows to O(1+α)O(1 + \alpha)O(1+α), where α\alphaα is the load factor. Resizing the hash table when the load factor exceeds a threshold (e.g., 0.7) takes O(n)O(n)O(n) time, but this happens infrequently, so the amortized complexity of insertion remains O(1)O(1)O(1). For the **Search** operation, the expected time complexity is O(1+α)O(1 + \alpha)O(1+α), as in the worst case, the search may require traversing the entire chain at a given index. Similarly, the **Delete** operation also has an expected time complexity of O(1+α)O(1 + \alpha)O(1+α), as it requires searching through the chain to remove the key-value pair. Thus, as the load factor increases, the efficiency of both search and delete operations degrades, but resizing helps mitigate this impact over time.

To maintain a low load factor and minimize collisions in a hash table, one effective strategy is **dynamic resizing**, where the table's capacity is increased (typically doubled) once the load factor exceeds a predefined threshold, such as 0.7. This reduces the number of elements per slot and helps keep the chains short, ensuring fast operations. Alongside resizing, using a **good hash function** is critical to minimize collisions. A well-designed hash function, like the multiplicative hash function, ensures that keys are distributed uniformly across the table, reducing the likelihood of multiple keys hashing to the same slot. **Rehashing**—recalculating the hash values and redistributing keys after resizing—is also essential to maintain good performance. Additionally, **lazy deletion** can be employed, where elements are not immediately removed but marked for deletion, helping reduce resizing frequency. These strategies help optimize hash table performance by keeping the load factor low and minimizing collisions, ensuring efficient insertions, searches, and deletions.

References:

Hurturk, E. (2021, June 25). *Probabilistic analysis of the randomized quicksort algorithm*. Medium. https://ege-hurturk.medium.com/probabilistic-analysis-of-the-randomized-quicksort-algorithm-27e6be30c1c0