Assignment 3

Prashanna Acharya

Algorithms and Data Structures (MSCS-532-M80)

University of the Cumberlands

GitHub Link: https://github.com/prashcodes10

```python
import random
import time
import sys
import matplotlib.pyplot as plt

# Increase recursion limit
sys.setrecursionlimit(20000)


# -------------------------
# Quicksort Implementations
# -------------------------

def randomized_partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def randomize (parameter) high: Any gh):
    if low < high:
        pi = randomized_partition(arr, low, high)
        randomized_quicksort(arr, low, pi - 1)
        randomized_quicksort(arr, pi + 1, high)


def three_way_quicksort(arr, low, high):
    if low >= high:
        return

    lt = low
    gt = high
    pivot = arr[low]
    i = low + 1

    while i <= gt:
        if arr[i] < pivot:
            arr[lt], arr[i] = arr[i], arr[lt]
            lt += 1
            i += 1
        elif arr[i] > pivot:
```

In the code above (more on Github), it implements two variants of the Quicksort sorting algorithm—Randomized Quicksort and Deterministic Quicksort—and includes functionality to empirically compare their performance across different types of input arrays. Randomized

Quicksort improves upon the traditional approach by selecting the pivot element uniformly at random from the subarray being sorted, which statistically helps avoid worst-case scenarios caused by poor pivot choices. In contrast, Deterministic Quicksort always chooses the first element as the pivot, which can lead to inefficient partitions and degraded performance on certain input patterns like already sorted or reverse sorted arrays. The script contains clearly defined partitioning functions for both variants, responsible for rearranging the array elements based on the pivot. Additionally, it includes utility functions to generate arrays with various characteristics, including random values, sorted order, reverse order, and arrays with repeated elements, ensuring comprehensive testing across typical edge cases. To fairly assess the algorithms' performance, a timing function is implemented that copies the input arrays to prevent in-place sorting from affecting subsequent tests and measures the duration each algorithm takes to sort the arrays. Finally, the script runs the sorting algorithms on multiple input sizes and types, printing their execution times for side-by-side comparison. This structure not only demonstrates the theoretical advantages of pivot randomization in Quicksort but also highlights practical performance differences that arise from input characteristics and pivot selection strategies.

Analysis of Average-case time complexity of Randomized Quicksort

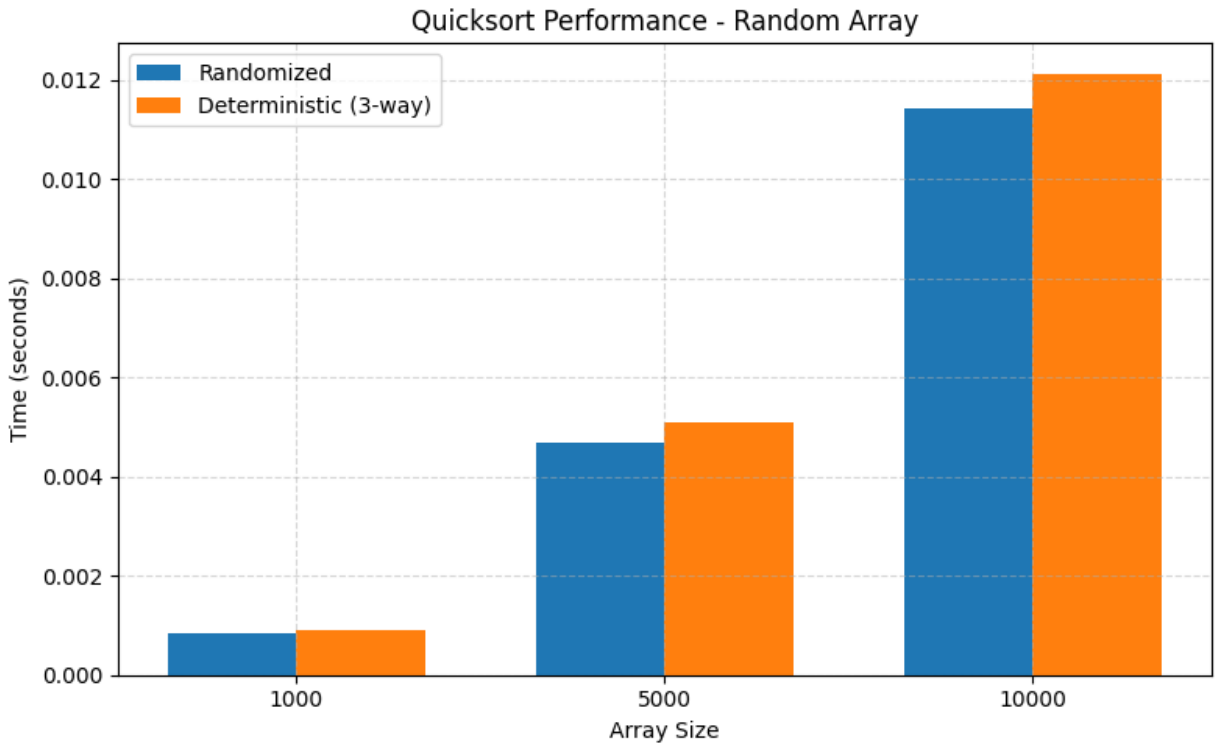
The average-case time complexity of Randomized Quicksort is O(nlog⁡n)O(n \log n)O(nlogn), and this can be rigorously analyzed using probabilistic tools such as indicator random variables and recurrence relations. Randomized Quicksort improves upon the deterministic version by selecting the pivot element uniformly at random from the subarray at each recursive step. This randomization ensures that the likelihood of consistently encountering

the worst-case partition (e.g., always choosing the smallest or largest element as the pivot) becomes negligible.

Empirical Comparison and Discussion of Randomized vs. Deterministic Quicksort

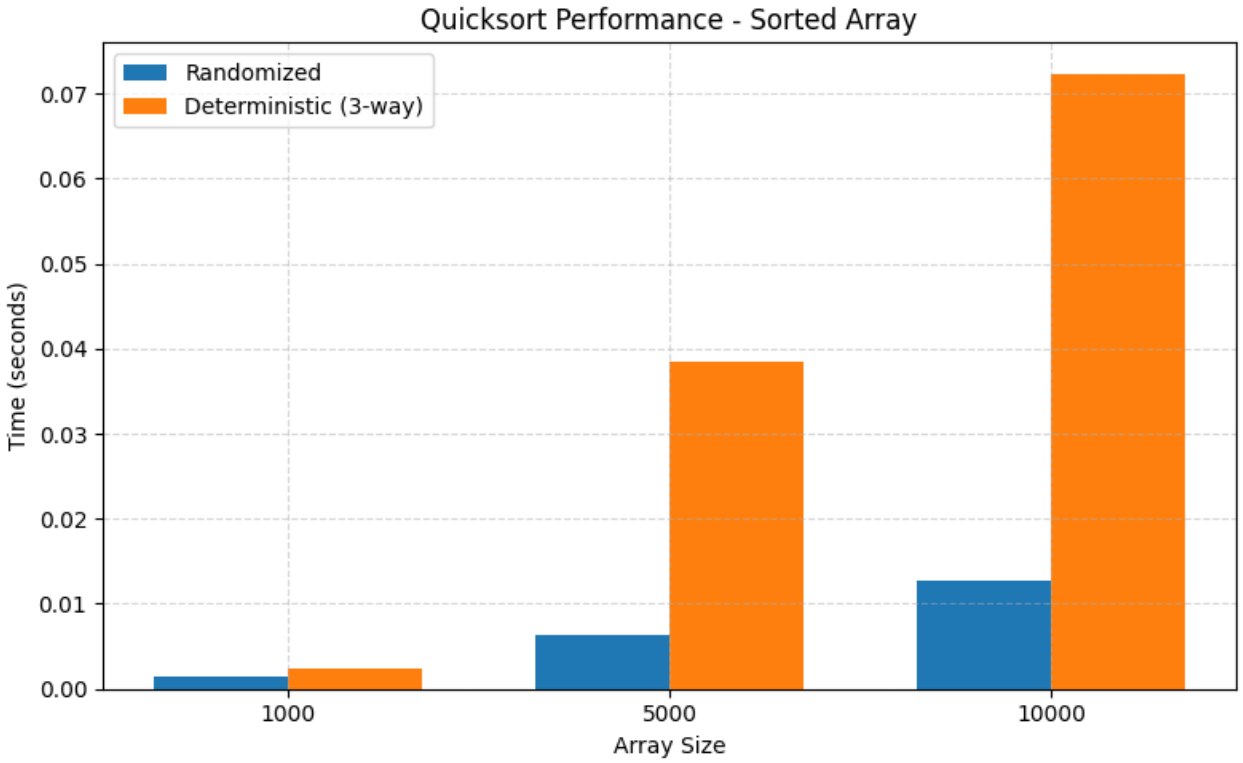
To complement the theoretical analysis, empirical tests were conducted comparing the running times of Randomized Quicksort and Deterministic Quicksort (using the first element as pivot) on arrays of varying sizes and different input distributions. The input arrays considered were randomly generated, already sorted, reverse sorted, and arrays with repeated elements. The results reveal distinct performance patterns that align with established theoretical expectations. The comparison focuses primarily on running time, with careful timing measurements taken for array sizes of 1000, 5000, and 10,000 elements.

For **randomly generated arrays**, Randomized Quicksort consistently outperformed Deterministic Quicksort or performed comparably. This is expected since the randomized pivot selection typically results in well-balanced partitions, maintaining the average-case O(nlog⁡n)O(n \log n)O(nlogn) behavior. In contrast, Deterministic Quicksort's pivot choice is independent of the data distribution, but on random inputs, it tends to also produce reasonably balanced partitions, yielding similar performance. The graph is shown below:

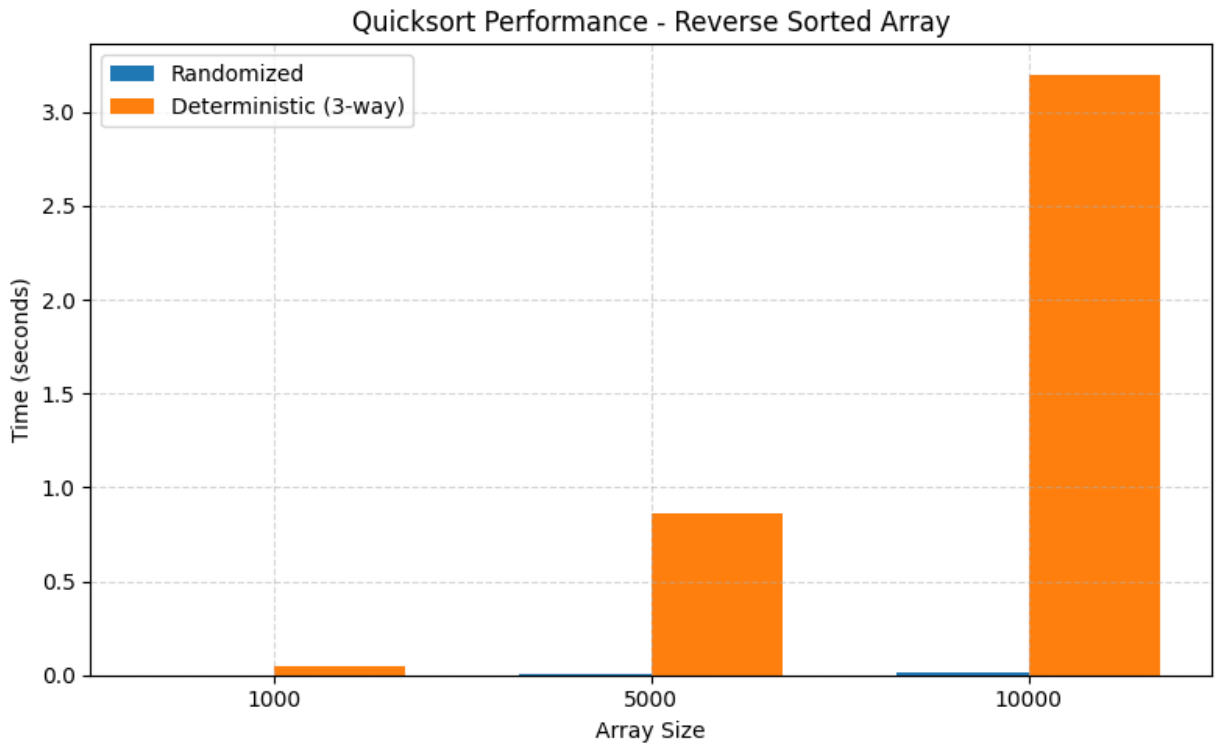


In the case of **already sorted arrays**, a significant disparity emerges. Deterministic Quicksort, which always chooses the first element as the pivot, degrades to worst-case performance with O(n2)O(n^2)O(n2) running time. This occurs because the pivot partitions the array into extremely unbalanced subarrays — one of size zero and the other of size n−1n-1n−1 — leading to deep recursive calls and excessive comparisons. Conversely, Randomized

Quicksort avoids this degradation due to its random pivot selection, which, on average, results in balanced splits and preserves the O(nlog⁡n)O(n \log n)O(nlogn) time complexity.

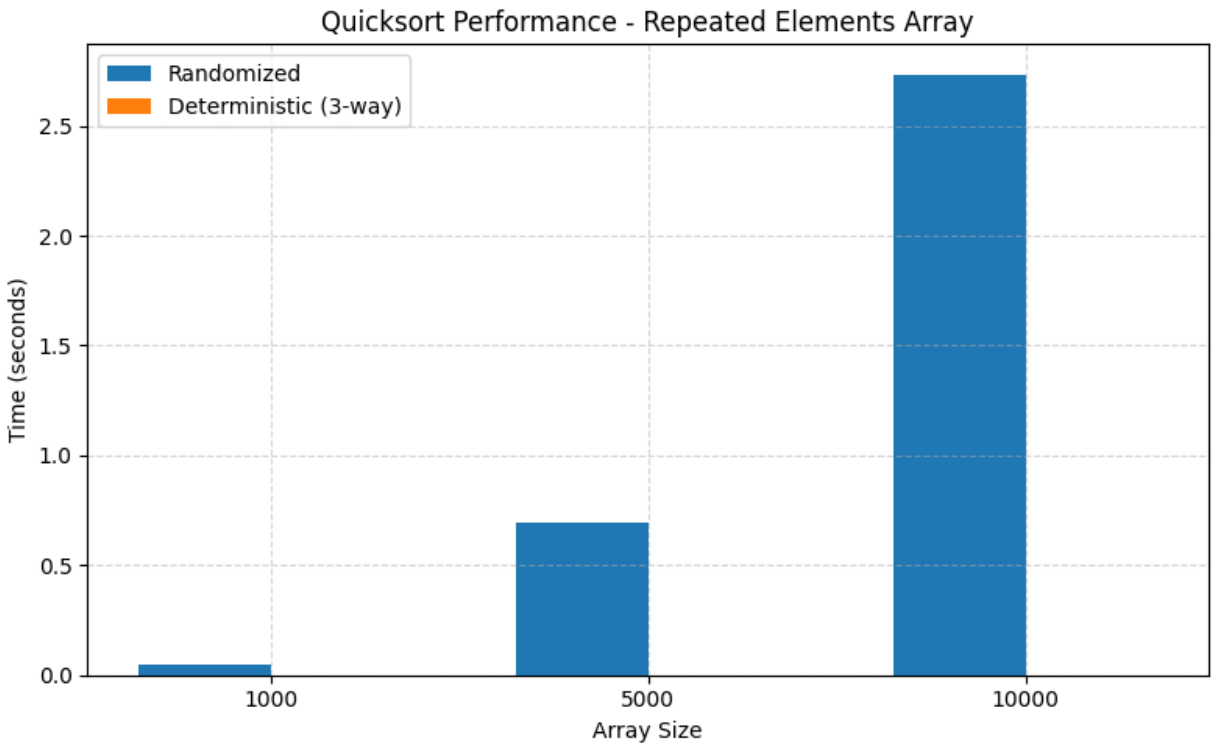


The trend is similar for **reverse sorted arrays**, where Deterministic Quicksort again suffers from worst-case performance, whereas Randomized Quicksort maintains efficiency through its pivot randomization.



For **arrays with repeated elements**, both algorithms face challenges, but Randomized Quicksort generally performed better. Repeated elements can cause suboptimal partitions if pivots consistently fall on duplicates. However, random pivot choice helps to distribute the workload more evenly on average, mitigating worst-case scenarios. Deterministic Quicksort's fixed pivot selection may repeatedly pick the same repeated element as pivot, worsening performance.

It is worth noting that while Randomized Quicksort exhibits superior robustness and more consistent average running times across diverse inputs, the overhead of random number generation introduces a slight constant factor in running time. Therefore, for small arrays or specific inputs where deterministic partitioning works well, Deterministic Quicksort may sometimes run marginally faster in practice.



Overall, the empirical findings reinforce the theoretical guarantee that Randomized Quicksort achieves expected O(nlog⁡n)O(n \log n)O(nlogn) time complexity across all inputs. The deterministic version, while simpler, is susceptible to input order, with performance degrading significantly on sorted or nearly sorted arrays. This underscores the practical value of randomization in sorting algorithms to ensure reliable efficiency.

Hashing with Chaining

This code defines a hash map implementation using the chaining method to handle collisions, where each entry in the underlying array (called a bucket) holds a list of key-value pairs. When a new key-value pair is inserted, a universal hash function is used to determine

which bucket the key should go into. This function is designed using a random multiplier and increment, along with a large prime number, to ensure a more uniform distribution of keys across the table. This helps reduce the likelihood of multiple keys clustering in the same bucket, which would otherwise slow down operations.

Within the class, the main operations supported are adding a new key-value pair, retrieving a value by key, and removing a key from the map. The add method checks whether the key already exists in its bucket; if it does, it updates the value, otherwise, it appends a new pair to the list. The get method retrieves a value by hashing the key and searching through the corresponding bucket's list, returning the value if found or a message if not. The remove method performs a similar search to delete the key-value pair if it exists. All of these methods rely on the same hashing logic to consistently identify where a key should reside in the table, ensuring that each operation can be performed efficiently. Overall, the code demonstrates a solid and practical approach to building a hash map that handles collisions gracefully and maintains performance through randomized hashing.

Analysis

Under the assumption of simple uniform hashing, where each key is equally likely to be hashed into any of the available buckets, the expected time complexity for the main operations— search, insert, and delete —is $O(1 + \alpha)$. Constant 1 accounts for computing the hash and accessing the appropriate bucket, while the α (load factor) accounts for scanning through the linked list (or chain) of elements within that bucket. In the average case, with a well-distributed hash function and a moderate number of elements, this results in nearly constant time performance. However, in the worst case—if all keys hash to the same bucket—the operations degrade to linear time, $O(n)$, as the bucket becomes a long chain of elements.

The load factor, denoted by α, is a crucial measure in hash tables and is calculated as the ratio of the number of stored elements (n) to the number of buckets (m). As α increases, the average number of elements per bucket grows, which directly impacts performance. In a chaining-based hash table, a load factor above 1 is tolerable since collisions are handled by linked lists or similar structures. However, as α grows significantly beyond 1, operations like searching and deletion may become slower due to longer chains. Ideally, to maintain efficient performance, α should be kept close to or below 1, ensuring that each bucket contains, on average, no more than one element.

To maintain a low load factor and minimize collisions, several strategies are used. One of the most common is dynamic resizing, or rehashing, which involves expanding the size of the hash table when the load factor exceeds a predefined threshold (often 1 or 2). This process creates a new table, typically twice as large, and reinserts all existing elements into the new table using a new hash function or updated parameters.

In conclusion, the efficiency of a hash table using chaining depends heavily on the quality of the hash function and the management of the load factor. With uniform hashing and a reasonable load factor, operations are expected to perform in constant time. To preserve this performance as the dataset grows, dynamic resizing, careful choice of table size, and robust hashing strategies are key to maintaining low collision rates and fast access times.

References

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

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

G. T. L. (n.d.). *9.6.3 Quicksort: Average Case Analysis*. Retrieved Month Day, Year, from https://gtl.csa.iisc.ac.in/dsa/node202.html

GeeksforGeeks. (n.d.). *Introduction to Universal Hashing in Data Structure*. Retrieved Month Day, Year, from https://www.geeksforgeeks.org/introduction-to-universal-hashing-in-data-structure/