

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

**MSCS – 532-A01**

**Algorithm and Data Structure**

**Name- Shilpa Mesineni**

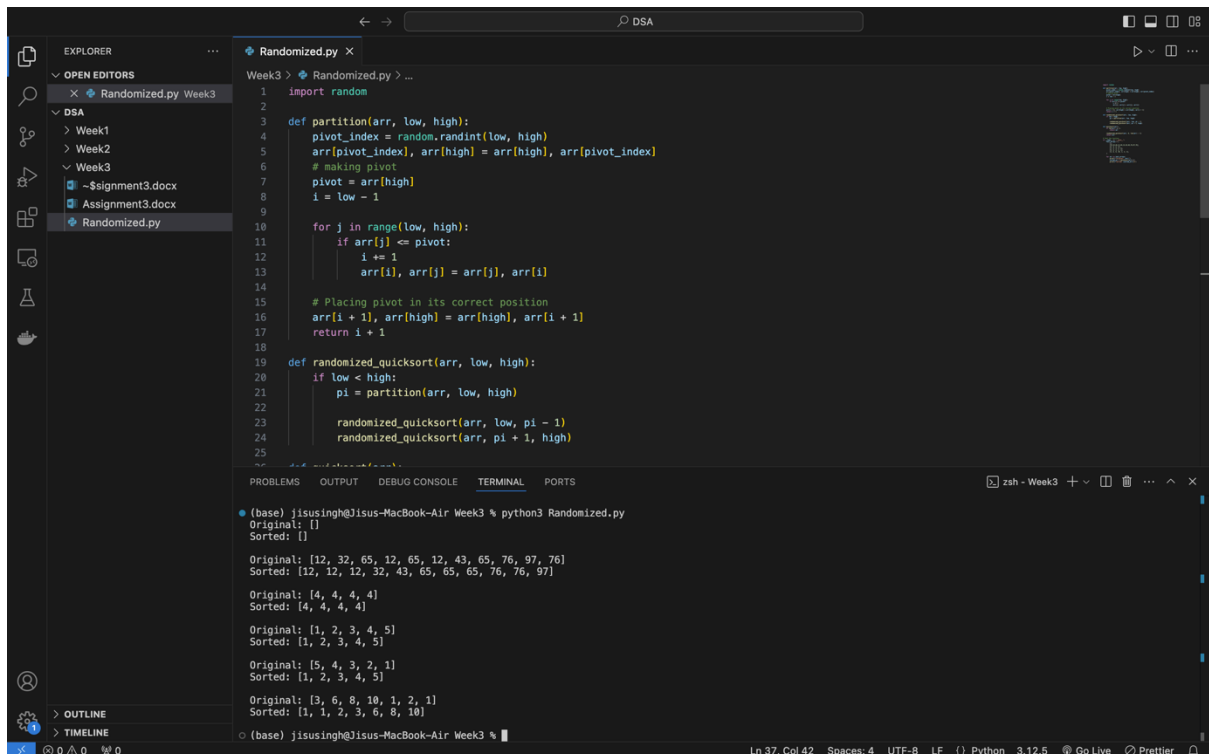
**Student ID- 005030618**

**Submission Date- 09/15/2024**

**GitHub: [https://github.com/shilpa-mesineni/MSCS532\\_Assignment3](https://github.com/shilpa-mesineni/MSCS532_Assignment3)**

## Part 1: Randomized Quicksort Analysis.

### Step 1: Implementation



```
1 import random
2
3 def partition(arr, low, high):
4     pivot_index = random.randint(low, high)
5     arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
6     # making pivot
7     pivot = arr[high]
8     i = low - 1
9
10    for j in range(low, high):
11        if arr[j] <= pivot:
12            i += 1
13            arr[i], arr[j] = arr[j], arr[i]
14
15    # Placing pivot in its correct position
16    arr[i + 1], arr[high] = arr[high], arr[i + 1]
17    return i + 1
18
19 def randomized_quicksort(arr, low, high):
20     if low < high:
21         pi = partition(arr, low, high)
22         randomized_quicksort(arr, low, pi - 1)
23         randomized_quicksort(arr, pi + 1, high)
24
25
26 (base) jisusingh@Jisus-MacBook-Air Week3 % python3 Randomized.py
Original: []
Sorted: []
Original: [12, 32, 65, 12, 65, 12, 43, 65, 76, 97, 76]
Sorted: [12, 12, 12, 32, 43, 65, 65, 76, 76, 97]
Original: [4, 4, 4, 4]
Sorted: [4, 4, 4, 4]
Original: [1, 2, 3, 4, 5]
Sorted: [1, 2, 3, 4, 5]
Original: [5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5]
Original: [3, 6, 8, 10, 1, 2, 1]
Sorted: [1, 1, 2, 3, 6, 8, 10]
```

Using `random.randint(low, high)`, the pivot is arbitrarily picked, guaranteeing that it is uniformly selected from the partitioned subarray that is currently in use.

All components smaller than the pivot are on its left, and all elements larger than the pivot are on its right, thanks to the partition function's placement of the pivot in the proper location.

The left and right subarrays are sorted recursively around the pivot by the `randomized_quicksort` function.

### Step 2: Analysis

An technique for sorting data called Randomized Quicksort typically takes  $O(n \log n)$  time, where  $O(\log_{f_0}^{f_1} f_0)$  is the average time. The worst-case behavior of deterministic Quicksort, which is  $O(n^2)$ , can be avoided and improved average performance is ensured by the unpredictability in pivot selection. We employ recurrence relations and probabilistic arguments based on indicator random variables to thoroughly investigate the average-case time complexity.

Defining the problem.

The amount of time required to partition the array (linear time, or  $O(n)$  for short).

The amount of time required to sort the two subarrays recursively. Let the predicted time complexity for sorting an array of size  $n$  be represented by  $T(n)$ . Each recursive call is

performed to the two subarrays of sizes  $k$  and  $n-k-1$  (where  $k$  is the pivot location) that are created during the partition stage.

As a result, the following can be used to express the recurrence relation for the expected temporal complexity  $T(n)$ :

$$T(n) = T(k) + T(n-k-1) + O(n)$$

$T(n) = T(k) + T(n-k-1) + O(n)$ , where the partitioning step is represented by  $O(n)$ , and the recursive sorting of the two subarrays is represented by  $T(k)$  and  $T(n-k-1)$ .

Indicator random Variables and expected Cost.

We add indicator random variables to examine the expected cost of partitioning around a randomly selected pivot, allowing us to rigorously compute the predicted time complexity.

Think about an array with size  $n$ .  $1/n$  is the likelihood of choosing any element to be the pivot. The total number of comparisons for every element in the array is the anticipated number of comparisons conducted during the partitioning process.

Recurrence Relation and Solution

The total amount of work completed at each recursion level equals the overall time complexity. The total time complexity is  $O(n \log n)$  due to the fact that there are  $O(\log n)$  layers of recursion and that each level completes  $O(n)$  work.

### Step 3: Comparison

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
ⓧ (base) jisusingh@Jisus-MacBook-Air Week3 % python3 Randomized.py

Array Size: 1000

Test Case: Random
Randomized Quicksort Time: 0.001266 seconds
Deterministic Quicksort Time: 0.001002 seconds

Test Case: Sorted
Randomized Quicksort Time: 0.001173 seconds
Deterministic Quicksort Time: 0.017327 seconds

Test Case: Reverse Sorted
Randomized Quicksort Time: 0.001149 seconds
Deterministic Quicksort Time: 0.029935 seconds

Test Case: Repeated
Randomized Quicksort Time: 0.017692 seconds
Deterministic Quicksort Time: 0.017071 seconds

Array Size: 5000

Test Case: Random
Randomized Quicksort Time: 0.007788 seconds
Deterministic Quicksort Time: 0.006330 seconds

Test Case: Sorted
```

## Part 2: Hashing with Chaining

### Step 1: Implementation

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
● (base) jisusingh@Jisus-MacBook-Air Week3 % python3 HashChaining.py
Hash Table after insertion:
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: [('apple', 1)]
Bucket 4: [('banana', 2)]
Bucket 5: [('grape', 4)]
Bucket 6: [('orange', 3)]
Bucket 7: []
Bucket 8: []
Bucket 9: []

Search results:
apple: 1
banana: 2
pear: None

Deleting 'banana'...

Hash Table after deletion:
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: [('apple', 1)]
Bucket 4: []
Bucket 5: [('grape', 4)]
Bucket 6: [('orange', 3)]
Bucket 7: []
Bucket 8: []
Bucket 9: []

Search for 'banana': None
○ (base) jisusingh@Jisus-MacBook-Air Week3 %
```

### Step 2: Analysis

The estimated time for search, insert, and delete operations in a hash table that uses chaining for collision resolution is dependent on how evenly the hash function distributes keys throughout the table. The performance analysis that follows is based on the assumption of simple uniform hashing, in which every key has an equal chance of being hashed to any bucket:

#### Insert

Because the hash function is computed in constant time and the key-value pair is appended to the relevant bucket, in the best-case scenario, where there are no collisions, the time to insert a key-value pair is  $O(1)$ . In the worst scenario, the time complexity drops to  $O(n)$ , where  $n$  is the number of entries in the table, if all keys hash to the same bucket.

Anticipated scenario: Each bucket typically has  $\lambda$  elements with a uniform distribution, where  $\lambda$  is the load factor (described below). Thus, for low load factors, the predicted insertion time is  $O(1 + \lambda)$ , which can be simplified to  $O(1)$ .

### Search

Computing the hash is the first step in the search process, after which the contents in the relevant bucket are scanned. If there are no collisions, the search time in the best scenario is  $O(1)$ . In the worst scenario, if every element hashes to the same bucket, it can take  $O(n)$  time. Anticipated scenario: Search times are typically  $O(1 + \lambda)$ , where  $\lambda$  is the average number of entries in each bucket. This equals  $O(1)$  for low load factors ( $O(1)$ ).

### Delete

The delete action finds the key by computing the hash and scanning the bucket's elements first, much like search does. The key-value pair is eliminated from the list if it is located. In the best scenario,  $O(1)$  is required for deletion. In the worst scenario, deletion takes  $O(n)$  time when all keys are in a single bucket. Expected case: Deletes take  $O(1 + \lambda)$  on average, where  $\lambda$  is the load factor.

In conclusion, the best hash table performance is achieved by keeping the load factor low and employing effective resizing and collision-resolution techniques. The estimated time for search, insert, and delete operations remains  $O(1)$  under the assumption of simple uniform hashing, but when the load factor increases, attention must be given to handle collisions effectively. Good hash functions and dynamic resizing are two crucial design decisions that help maintain a low load factor and reduce collisions, which guarantees the hash table operates effectively.

GitHub Repo: [https://github.com/shilpa-mesineni/MSCS532\\_Assignment3](https://github.com/shilpa-mesineni/MSCS532_Assignment3)