



9/15/2024

# Assignment 3

Randomized Quicksort and Hashing  
with Chaining



Sujan Dumar

Student ID: 005030060

MSCS 532: Algorithms and Data Structures

Dr. Vanessa Cooper

## INTRODUCTION

In modern computing, algorithm efficiency and scalability are critical to the performance of software programs, particularly when dealing with huge datasets. This report explores the efficiency and scalability of two fundamental algorithms: **Randomized Quicksort** and **Hashing with Chaining**. Randomized Quicksort is a sorting algorithm that overcomes the weakness of Quicksort and increases the efficiency by choosing pivot element at random, lowering the likelihood of worst-case scenarios while maintaining an average time complexity of  $O(n \log n)$ . In contrast, Hashing with Chaining, on the other hand, is used to efficiently handle key-value pairs by storing several elements in linked lists at each hash table index to prevent collisions. The report provides both theoretical analysis and empirical evaluation of these algorithms, providing insights into their performance under various conditions and input distributions.

## RANDOMIZED QUICKSORT IMPLEMENTATION

Randomized Quicksort is an enhanced version of the standard Quicksort algorithm that improves performance by selecting the pivot element randomly from the array, rather than using a fixed position such as the first or last element. This random selection reduces the risk of encountering consistently poor pivot choices, which can lead to the worst-case time complexity of  $O(n^2)$ , especially with already sorted or reverse-sorted data. By picking the pivot uniformly at random, Randomized Quicksort achieves an average-case time complexity of  $O(n \log n)$ , making it highly efficient for sorting large datasets with various input distributions. Additionally, this randomization enhances the algorithm's scalability and reliability in real-world scenarios. The primary difference in implementation compared to standard Quicksort is the random selection of the pivot element. The implementation of Randomized Quicksort is shown below:

## RANDOMIZED QUICKSORT AND HASH TABLE

```

1  """ Understanding Randomized Quicksort """
2
3  import timeit
4  import random
5  from memory_profiler import profile
6
7  def partition(array, first, last):
8      """ Split the array into 2 parts based on the pivot element """
9      pivot = array[last]                                # Select a pivot element to compare with other elements
10     low_index = first - 1                               # Select the highest index of opposite side of pivot
11
12     # Traverse through the array from beginning till end except the pivot element
13     # If the element is smaller than pivot then we swap it the left side,
14     # and save the index for pivot element
15     for index in range(first, last):
16         if array[index] <= pivot:
17             low_index += 1
18             array[low_index], array[index] = array[index], array[low_index]
19
20     # Once the traversal ends, pivot element is added on the last index
21     # which separated smaller and larger element
22     array[low_index + 1], array[last] = array[last], array[low_index + 1]
23
24     return (low_index + 1)                               # Return the new index of pivot element
25
26 def randomized_partition(array, first, last):
27     """ Function to select a random pivot element and do partition """
28     random_pivot = random.randint(first, last)           # Select pivot randomly
29     array[last], array[random_pivot] = array[random_pivot], array[last] # Swap last element with random pivot element
30     return partition(array, first, last)                 # Apply normal partition function
31

```

Figure 1: Implementation of randomized\_partition

```

32 # @profile
33 def randomized_quicksort(array, first, last):
34     """ Function that will take array, leftmost index and rightmost index of array
35         as an input and return sorted array """
36     if len(array) == 0:                                # Return array if it's empty
37         return array
38
39     if first < last:
40         partition_index = randomized_partition(array, first, last)
41
42         # Divide-and-Conquer method
43         randomized_quicksort(array, first, partition_index - 1)
44         randomized_quicksort(array, partition_index + 1, last)
45     return array
46
47 def print_execution_times(array_name):
48     """ Function to print execution times """
49     timer_stmt = 'randomized_quicksort({}, 0, len({}) - 1)'
50     times = timeit.repeat(stmt=timer_stmt.format(array_name), repeat=5, number=10000, globals=globals())
51     print('total execution time: ' + str(min(times)))
52
53 def huge_random_array():
54     """ Function to return huge number of random integers for testing purpose """
55     array = []
56     max_numbers = 500
57     for i in range(max_numbers):
58         array.append(random.randint(0, max_numbers))
59     return array
60
61 # Testing performance of algorithms on these arrays
62 empty_array = []
63 randomized_array = [23, 65, 98, 1, 36, 47, 76, 28, 83, 15]
64 sorted_array = [1, 15, 23, 28, 36, 47, 65, 76, 83, 98]
65 reversed_sorted_array = [98, 83, 76, 65, 47, 36, 28, 23, 15, 1]
66 repeated_elements_array = [23, 56, 23, 84, 56, 23, 56, 84, 23, 84]
67
68 # Print execution times for the array
69 print("EMPTY ARRAY", end = " ")
70 print_execution_times(empty_array)
71
72 print("RANDOM ARRAY", end = " ")
73 print_execution_times(randomized_array)
74
75 print("SORTED ARRAY", end = " ")
76 print_execution_times(sorted_array)
77
78 print("REVERSED SORTED ARRAY", end = " ")
79 print_execution_times(reversed_sorted_array)
80
81 print("REPEATED ELEMENTS ARRAY", end = " ")
82 print_execution_times(repeated_elements_array)
83
84 print("HUGE RANDOM ARRAY", end = " ")
85 print_execution_times(huge_random_array())
86

```

Figure 2: Implementation of randomized quicksort

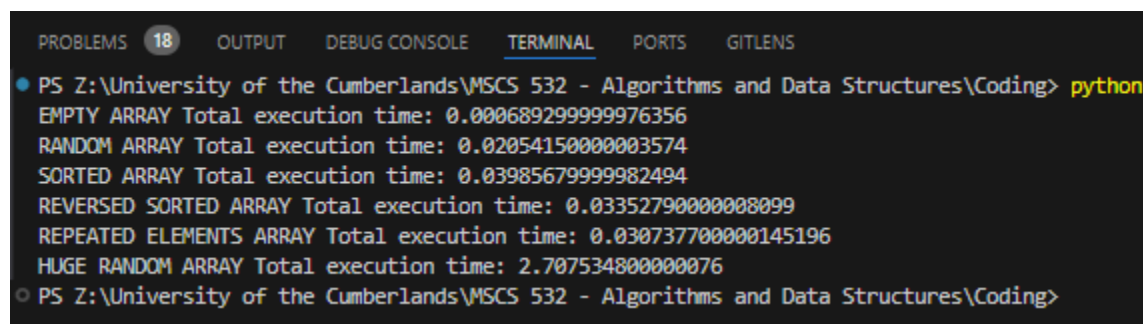
## Analysis of Randomized Quicksort

Randomized Quicksort enhances the standard Quicksort algorithm by choosing a pivot element uniformly at random from the subarray being partitioned. This approach mitigates the risk of encountering the worst-case time complexity of  $O(n^2)$ , which can occur with poor pivot choices leading to highly unbalanced partitions. By randomizing the pivot selection, the algorithm typically achieves more balanced partitions on average, resulting in efficient sorting. The average-case time complexity of Randomized Quicksort is  $O(n \log n)$  which can be derived from the following recurrence relation (Cormen et al., 2022):

$$T(n) = 2T(n/2) + O(n)$$

Despite the randomization, the worst-case time complexity remains  $O(n^2)$ . This worst case occurs if the pivot selection consistently results in the most unbalanced partitions, such as always picking the smallest or largest element. However, because this scenario is highly improbable, the average-case time complexity is a more realistic measure of performance. In practice, the average-case time complexity dominates, making Randomized Quicksort a preferred choice for sorting large datasets with varying input distributions.

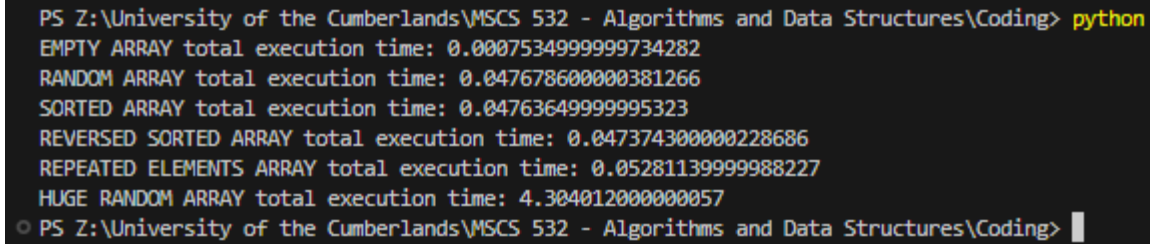
## Comparison with Quicksort



```
PROBLEMS 18 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding> python
EMPTY ARRAY Total execution time: 0.000689299999976356
RANDOM ARRAY Total execution time: 0.02054150000003574
SORTED ARRAY Total execution time: 0.03985679999982494
REVERSED SORTED ARRAY Total execution time: 0.03352790000008099
REPEATED ELEMENTS ARRAY Total execution time: 0.030737700000145196
HUGE RANDOM ARRAY Total execution time: 2.707534800000076
PS Z:\University of the Cumberlands\MSCS 532 - Algorithms and Data Structures\Coding>
```

Figure 3: Execution time for Quicksort in various datasets

## RANDOMIZED QUICKSORT AND HASH TABLE



```
PS Z:\University of the Cumberland\MSCS 532 - Algorithms and Data Structures\Coding> python  
EMPTY ARRAY total execution time: 0.0007534999999734282  
RANDOM ARRAY total execution time: 0.047678600000381266  
SORTED ARRAY total execution time: 0.04763649999995323  
REVERSED SORTED ARRAY total execution time: 0.047374300000228686  
REPEATED ELEMENTS ARRAY total execution time: 0.05281139999988227  
HUGE RANDOM ARRAY total execution time: 4.304012000000057  
PS Z:\University of the Cumberland\MSCS 532 - Algorithms and Data Structures\Coding> |
```

Figure 4: Execution time for Randomized Quicksort in various datasets

The figures clearly show that Randomized Quicksort has more consistent execution times compared to standard Quicksort. This consistency arises because Randomized Quicksort typically exhibits  $O(n \log n)$  time complexity on average, regardless of the input dataset. In contrast, standard Quicksort can experience significant variations in execution time, particularly with different input configurations such as random or sorted arrays. By mitigating these issues through random pivot selection, Randomized Quicksort provides more predictable and reliable performance.

## HASHING WITH CHAINING IMPLEMENTATION

A hash function is used in hashing to map keys to specific values. However, when two or more keys are mapped to the same value, various hashing algorithms may result in a collision. In hash tables, a collision resolution method called hashing with chaining is employed to minimize the collisions. A linked list is inserted at an index if more than one key-value pair points to the same index. When a collision occurs, the new element is simply added to the chain, allowing the table to handle multiple keys at the same hash value. This method ensures efficient insertion, search, and deletion operations with an average-case time complexity of  $O(1)$  under the assumption of uniform hashing, making it well-suited for dynamic datasets where collisions are likely. The implementation for the hash table using universal hash function is shown below:

## RANDOMIZED QUICKSORT AND HASH TABLE

```

You, 3 hours ago | 1 author (You)
5 class HashTable:
6     """ Data structure to store data based on the key """
7
8     def __init__(self, size = 10):
9         self.size = size
10        self.prime_number = 1000000007 # A large prime number
11        self.a = random.randint(1, self.prime_number - 1)
12        self.b = random.randint(0, self.prime_number - 1)
13        self.table = [[] for _ in range(size)]
14
15    def universal_hash(self, key):
16        """ Universal hash function using modulo (((ak + b) mod p) mod m) """
17        return ((self.a * key + self.b) % self.prime_number) % self.size
18
19    def insert(self, key, value):
20        """ Insert a key-value pair into the hash table """
21        hash_index = self.universal_hash(key)
22
23        # Check if the key already exists, and update the value if it does
24        for pair in self.table[hash_index]:
25            if pair[0] == key:
26                pair[1] = value
27                return
28
29        # If key doesn't exist, append the new key-value in the table
30        self.table[hash_index].append([key, value])
31
32    def search(self, key):
33        """ Search for the value based on the key provided """
34        hash_index = self.universal_hash(key)
35
36        # Search through the list at the hash index for the key
37        # Return the value if the key is found
38        for pair in self.table[hash_index]:
39            if pair[0] == key:
40                return pair[1]
41
42        # Return None if the key is not found
43        return None
44
45    def delete(self, key):
46        """ Delete a key-value pair from the hash table """
47        hash_index = self.universal_hash(key)
48
49        # Search through the list at the hash index for the key
50        # Delete the key-value pair if the key is found and return true
51        for index, pair in enumerate(self.table[hash_index]):
52            if pair[0] == key:
53                del self.table[hash_index][index]
54                return True
55
56        # Return false if the key is not found
57        return False

```

Figure 5: Implementation of Hash table

## RANDOMIZED QUICKSORT AND HASH TABLE

## Analysis of Hashing with Chaining

```

59 hash_table = HashTable() # Create a hash table of default slot size 10
60
61 # Insert key-value pairs
62 hash_table.insert(1, "Peoria")
63 hash_table.insert(10, "Dallas")
64 hash_table.insert(5, "Pheonix")
65 hash_table.insert(20, "Austin")
66 hash_table.insert(24, "Brooklyn")
67 hash_table.insert(13, "Williamsburg")
68
69 # Display the hash table
70 print("Hash Table after insertions:")
71 for i, bucket in enumerate(hash_table.table):
72     print(f"Index {i}: {bucket}")
73
74 # Search for location based on the key
75 print("\nSearch for key 24:", hash_table.search(24))
76 print("Search for key 13:", hash_table.search(13))
77 print("Search for key 5:", hash_table.search(5))
78 print("Search for key 30 (not present):", hash_table.search(30))
79
80 # Delete Dallas and Williamsburg
81 hash_table.delete(10)
82 hash_table.delete(13)
83
84 # Display the hash table after deletion
85 print("\nHash Table after deletion:")
86 for i, bucket in enumerate(hash_table.table):
87     print(f"Index {i}: {bucket}")
88

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

```

Hash Table after insertions:
Index 0: []
Index 1: []
Index 2: []
Index 3: []
Index 4: []
Index 5: []
Index 6: [[20, 'Austin']]
Index 7: [[24, 'Brooklyn']]
Index 8: [[1, 'Peoria'], [10, 'Dallas'], [13, 'Williamsburg']]
Index 9: [[5, 'Pheonix']]

Search for key 24: Brooklyn
Search for key 13: Williamsburg
Search for key 5: Pheonix
Search for key 30 (not present): None

Hash Table after deletion:
Index 0: []
Index 1: []
Index 2: []
Index 3: []
Index 4: []
Index 5: []
Index 6: [[20, 'Austin']]
Index 7: [[24, 'Brooklyn']]
Index 8: [[1, 'Peoria']]
Index 9: [[5, 'Pheonix']]

```

Figure 6: Result output from the hash table implementation

## RANDOMIZED QUICKSORT AND HASH TABLE

From the above output, we observe that at index 8, there are three elements causing a collision, resulting in a chain-like structure with three key-value pairs at that index. The hash index for these pairs is calculated using a universal hash function based on modular arithmetic:  $hash(k) = (((a.k + b) \bmod p) \bmod m)$ , where  $a$  and  $b$  are constants,  $p$  is a prime number and  $m$  is the number of slots in the hash table (Cormen et al., 2022).

For insertion, deletion, and search operations, the expected time complexity is  $O(1)$  under simple uniform hashing. However, with a load factor  $\alpha = n/m$ , where  $n$  is the number of elements and  $m$  is the number of slots, the time complexity can degrade to  $O(\alpha)$  in the worst case if many elements hash to the same slot. To maintain a low load factor and ensure efficient operations, dynamic resizing can be employed. This process involves doubling the size of the hash table and rehashing the elements, which helps to reduce collisions and improve performance.

## CONCLUSION

This report examined two crucial algorithms – Randomized Quicksort and Hashing with Chaining and included both a theoretical explanation and real-world examples. With its pivot selection based on randomness, Randomized Quicksort is a reliable option for sorting a variety of input data since it regularly achieves an average-case time complexity of  $O(n \log n)$ . On the other hand, under ideal circumstances, Hashing with Chaining provides a constant time complexity of  $O(1)$  for insertion, selection, and deletion, making it an effective method for maintaining dynamic datasets. Ultimately, choosing the right algorithms for a given set of use cases requires careful consideration of the scalability and efficiency of the algorithms.



## REFERENCES

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). Random House Publishing Services. <https://reader2.yuzu.com/books/9780262367509>