

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

***Milan Bista***

***University of Cumberlands***

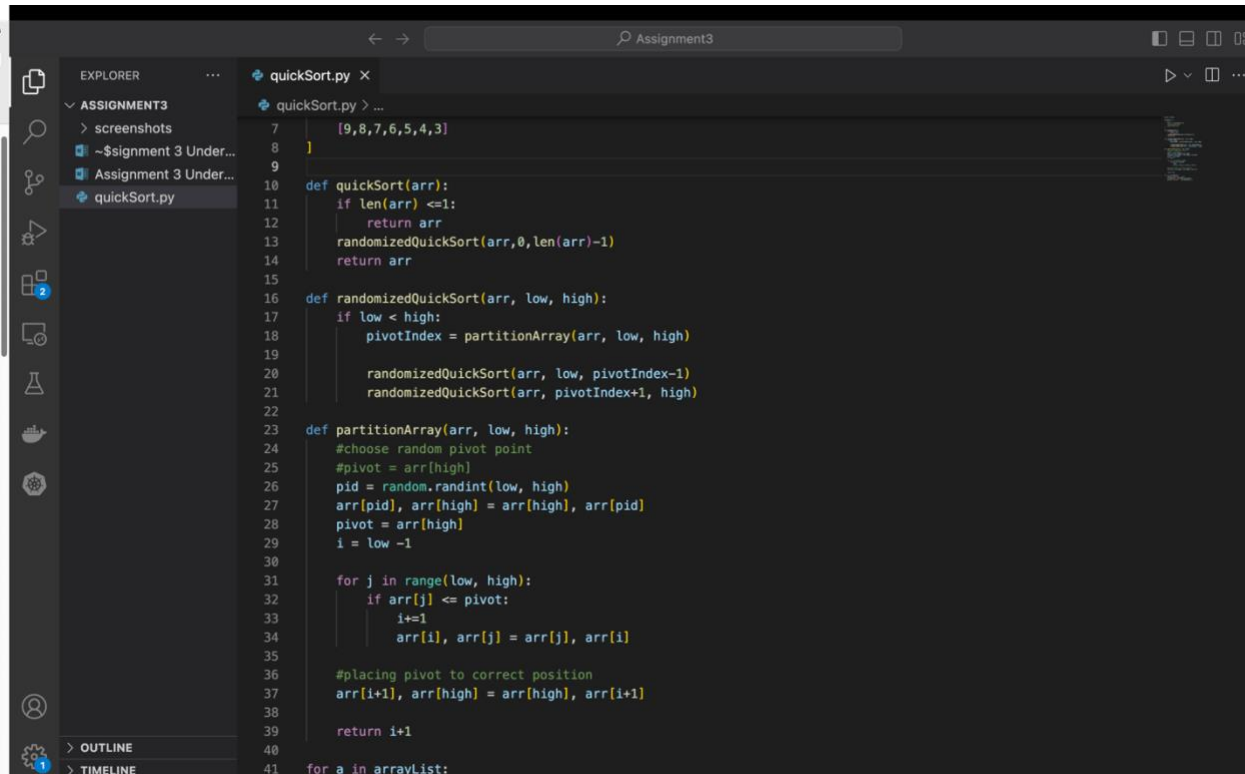
*2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term*

*Instructors: Machica McClain / Vanessa Cooper*

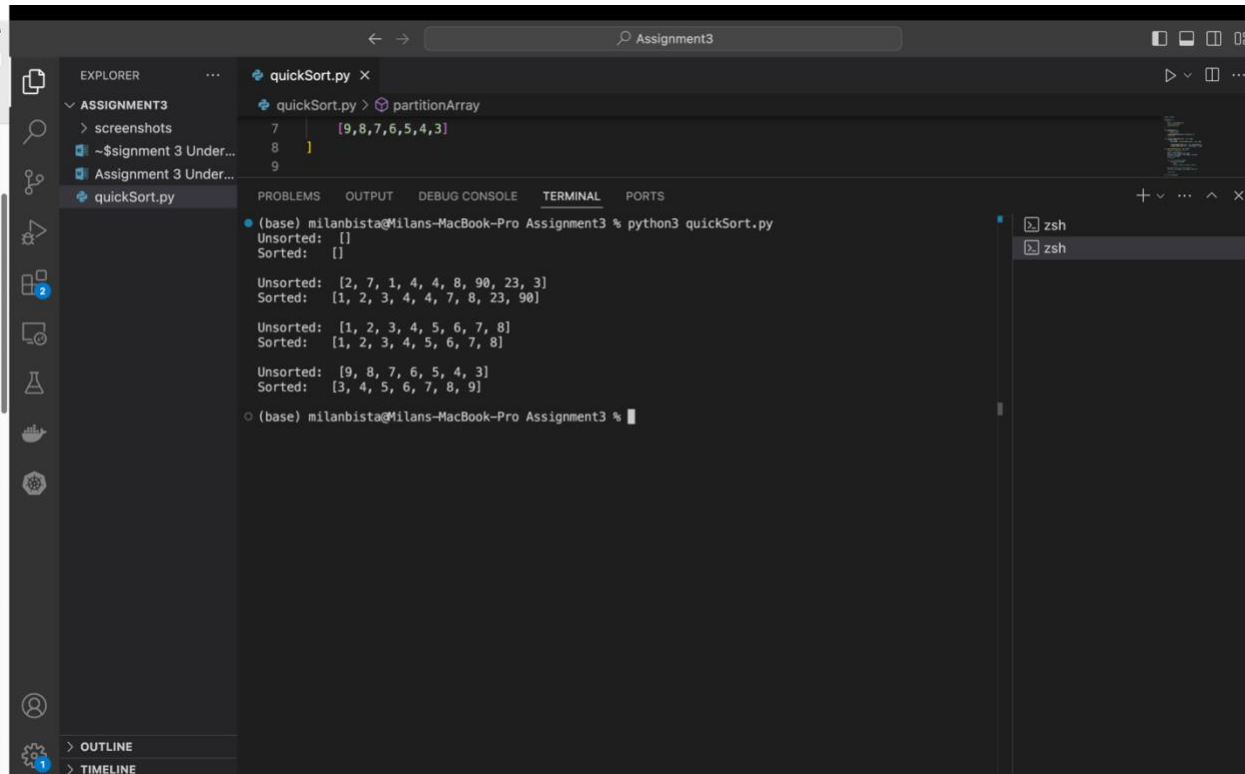
# Part 1: Randomized Quicksort Analysis

## 1. Implementation

- I have implemented a randomized quicksort algorithm with different edge cases such as empty list, reversed, random etc.



```
7 [9,8,7,6,5,4,3]
8 ]
9
10 def quickSort(arr):
11     if len(arr) <= 1:
12         return arr
13     randomizedQuickSort(arr,0,len(arr)-1)
14     return arr
15
16 def randomizedQuickSort(arr, low, high):
17     if low < high:
18         pivotIndex = partitionArray(arr, low, high)
19
20         randomizedQuickSort(arr, low, pivotIndex-1)
21         randomizedQuickSort(arr, pivotIndex+1, high)
22
23 def partitionArray(arr, low, high):
24     #choose random pivot point
25     #pivot = arr[high]
26     pid = random.randint(low, high)
27     arr[pid], arr[high] = arr[high], arr[pid]
28     pivot = arr[high]
29     i = low - 1
30
31     for j in range(low, high):
32         if arr[j] <= pivot:
33             i+=1
34             arr[i], arr[j] = arr[j], arr[i]
35
36     #placing pivot to correct position
37     arr[i+1], arr[high] = arr[high], arr[i+1]
38
39     return i+1
40
41 for a in arrayList:
```



The screenshot shows a VS Code editor window with a file named `quickSort.py` open. The file contains a recursive implementation of the QuickSort algorithm. The code is as follows:

```
def partitionArray(arr):  
    if len(arr) <= 1:  
        return arr  
    pivot = arr[0]  
    less = []  
    greater = []  
    for i in range(1, len(arr)):  
        if arr[i] < pivot:  
            less.append(arr[i])  
        else:  
            greater.append(arr[i])  
    return [partitionArray(less), pivot, partitionArray(greater)]  
  
def quickSort(arr):  
    return partitionArray(arr)
```

The terminal output shows the execution of the `quickSort.py` file. The output is as follows:

```
(base) milanbista@Milans-MacBook-Pro Assignment3 % python3 quickSort.py  
Unsorted: []  
Sorted: []  
  
Unsorted: [2, 7, 1, 4, 4, 8, 90, 23, 3]  
Sorted: [1, 2, 3, 4, 4, 7, 8, 23, 90]  
  
Unsorted: [1, 2, 3, 4, 5, 6, 7, 8]  
Sorted: [1, 2, 3, 4, 5, 6, 7, 8]  
  
Unsorted: [9, 8, 7, 6, 5, 4, 3]  
Sorted: [3, 4, 5, 6, 7, 8, 9]
```

## 2. Analysis

In Randomized Quicksort, the algorithm starts by picking a pivot element from the array at random. This pivot is used to partition the array into two parts: elements less than the pivot go to one side, and elements greater than the pivot go to the other. The algorithm then recursively applies this process to each of the two subarrays until the entire array is sorted.

The efficiency of Quicksort depends heavily on how well the pivot divides the array. If the pivot divides the array into roughly equal parts, each partitioning step reduces the problem size significantly, leading to fewer steps overall. Choosing the pivot randomly helps ensure that, on average, the divisions are well-balanced.

When we say the average-case complexity is  $O(n \log n)$ , it means that on average, Randomized Quicksort needs about  $O(n)$  comparisons to process each level of recursion and about  $O(\log n)$  levels of recursion.

Milan Bista

Quick Sort time Complexity =  $O(n \log n)$

Each partitioning step requires  $O(n)$  work to examine each element to place it either the left or right subarray. After partitioning, the resulting subarray are half the original size.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

where,

$T(n)$  is the total time needed to sort an array of size  $n$ .

$2T\left(\frac{n}{2}\right)$  represents the time needed to sort the two subarrays, each approx. size  $\frac{n}{2}$ .

$O(n)$  is the partitioning step that divides the array around pivot.

$$\text{Total time} = \text{number of levels} \times \text{time per level}$$

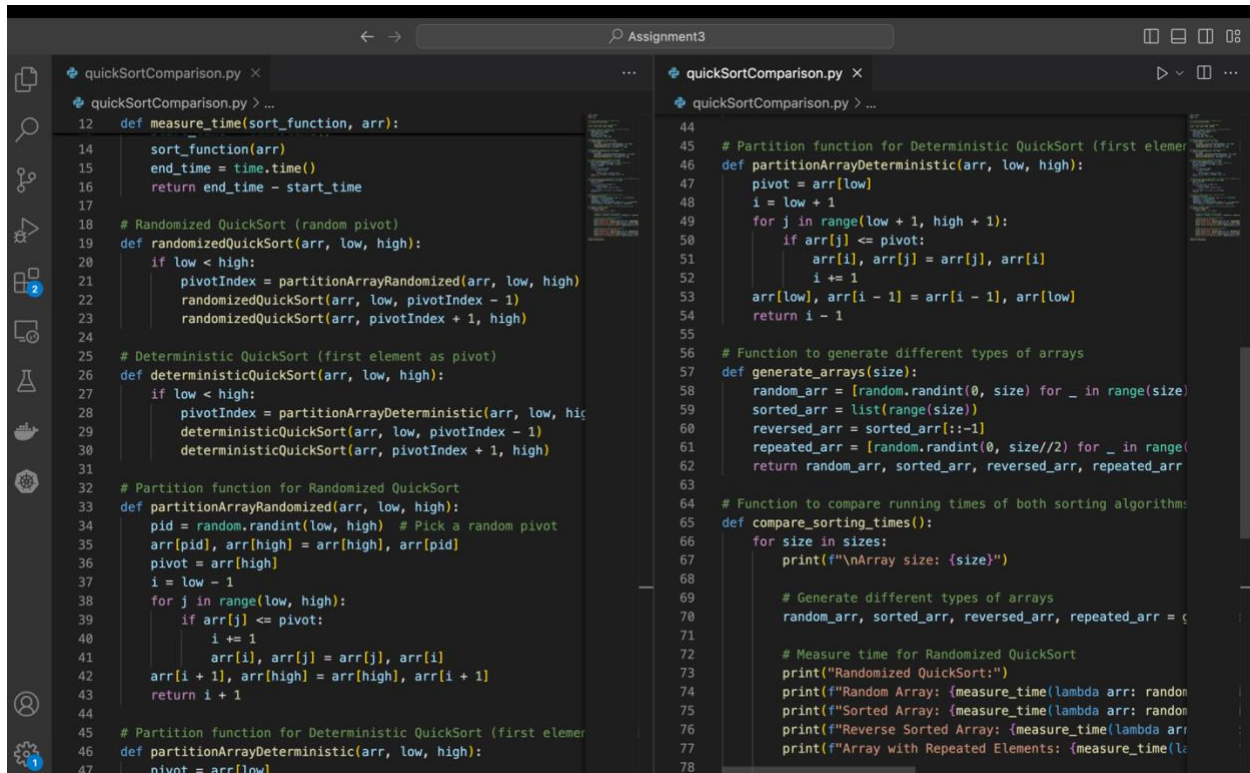
$$= O(\log n) \times O(n)$$

$$= O(n \log n)$$

where  $\log n$  is the time for each subarray.

### 3. Comparison

- Empirically comparing the running time of Randomized Quicksort with Deterministic Quicksort (using the first element as the pivot) on different input sizes and distributions:



```
12 def measure_time(sort_function, arr):
13     sort_function(arr)
14     end_time = time.time()
15     return end_time - start_time
16
17 # Randomized QuickSort (random pivot)
18 def randomizedQuickSort(arr, low, high):
19     if low < high:
20         pivotIndex = partitionArrayRandomized(arr, low, high)
21         randomizedQuickSort(arr, low, pivotIndex - 1)
22         randomizedQuickSort(arr, pivotIndex + 1, high)
23
24 # Deterministic QuickSort (first element as pivot)
25 def deterministicQuickSort(arr, low, high):
26     if low < high:
27         pivotIndex = partitionArrayDeterministic(arr, low, high)
28         deterministicQuickSort(arr, low, pivotIndex - 1)
29         deterministicQuickSort(arr, pivotIndex + 1, high)
30
31 # Partition function for Randomized QuickSort
32 def partitionArrayRandomized(arr, low, high):
33     pid = random.randint(low, high) # Pick a random pivot
34     arr[pid], arr[high] = arr[high], arr[pid]
35     pivot = arr[high]
36     i = low - 1
37     for j in range(low, high):
38         if arr[j] <= pivot:
39             i += 1
40             arr[i], arr[j] = arr[j], arr[i]
41     arr[i + 1], arr[high] = arr[high], arr[i + 1]
42     return i + 1
43
44 # Partition function for Deterministic QuickSort (first element as pivot)
45 def partitionArrayDeterministic(arr, low, high):
46     pivot = arr[low]
47     i = low + 1
48     for j in range(low + 1, high + 1):
49         if arr[j] <= pivot:
50             arr[i], arr[j] = arr[j], arr[i]
51             i += 1
52     arr[low], arr[i - 1] = arr[i - 1], arr[low]
53     return i - 1
54
55 # Function to generate different types of arrays
56 def generate_arrays(size):
57     random_arr = [random.randint(0, size) for _ in range(size)]
58     sorted_arr = list(range(size))
59     reversed_arr = sorted_arr[::-1]
60     repeated_arr = [random.randint(0, size//2) for _ in range(size)]
61     return random_arr, sorted_arr, reversed_arr, repeated_arr
62
63 # Function to compare running times of both sorting algorithms
64 def compare_sorting_times():
65     for size in sizes:
66         print(f"\nArray size: {size}")
67
68         # Generate different types of arrays
69         random_arr, sorted_arr, reversed_arr, repeated_arr = generate_arrays(size)
70
71         # Measure time for Randomized QuickSort
72         print("Randomized QuickSort:")
73         print(f"Random Array: {measure_time(lambda arr: randomizedQuickSort(arr, 0, len(arr) - 1), random_arr)}")
74         print(f"Sorted Array: {measure_time(lambda arr: randomizedQuickSort(arr, 0, len(arr) - 1), sorted_arr)}")
75         print(f"Reverse Sorted Array: {measure_time(lambda arr: randomizedQuickSort(arr, 0, len(arr) - 1), reversed_arr)}")
76         print(f"Array with Repeated Elements: {measure_time(lambda arr: randomizedQuickSort(arr, 0, len(arr) - 1), repeated_arr)}")
77
78         # Measure time for Deterministic QuickSort
79         print("Deterministic QuickSort:")
80         print(f"Random Array: {measure_time(lambda arr: deterministicQuickSort(arr, 0, len(arr) - 1), random_arr)}")
81         print(f"Sorted Array: {measure_time(lambda arr: deterministicQuickSort(arr, 0, len(arr) - 1), sorted_arr)}")
82         print(f"Reverse Sorted Array: {measure_time(lambda arr: deterministicQuickSort(arr, 0, len(arr) - 1), reversed_arr)}")
83         print(f"Array with Repeated Elements: {measure_time(lambda arr: deterministicQuickSort(arr, 0, len(arr) - 1), repeated_arr)}")
```

```
quickSortComparison.py x
quickSortComparison.py > ...
12 def measure_time(sort_function, arr):

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Array size: 100
Randomized QuickSort:
Random Array: 5.412101745605469e-05 seconds
Sorted Array: 5.1975250244140625e-05 seconds
Reverse Sorted Array: 5.1021575927734375e-05 seconds
Array with Repeated Elements: 5.888938903808594e-05 seconds

Deterministic QuickSort:
Random Array: 3.886222830355469e-05 seconds
Sorted Array: 7.700928104980469e-05 seconds
Reverse Sorted Array: 0.00013685226440429688 seconds
Array with Repeated Elements: 3.695487976074219e-05 seconds

Array size: 1000
Randomized QuickSort:
Random Array: 0.0007951259613037109 seconds
Sorted Array: 0.0007832050323486328 seconds
Reverse Sorted Array: 0.0007562637329101562 seconds
Array with Repeated Elements: 0.0008807182312011719 seconds

Deterministic QuickSort:
Random Array: 0.0006520748138427734 seconds
Sorted Array: 0.010614156723022461 seconds
Reverse Sorted Array: 0.01719188690185547 seconds
Array with Repeated Elements: 0.0007009506225585938 seconds

Array size: 10000
Randomized QuickSort:
Random Array: 0.010299205780029297 seconds
Sorted Array: 0.008749961853027344 seconds
Reverse Sorted Array: 0.009100198745727539 seconds
Array with Repeated Elements: 0.010136842727661133 seconds

Deterministic QuickSort:
Random Array: 0.008152961730957031 seconds
Traceback (most recent call last):
  File "/Users/milanbista/Documents/Cumberlands/Classes/MSCS-532-B01/Assignment3/quickSortComparison.py", line 30, in determin
    isticQuickSort
    deterministicQuickSort(arr, pivotIndex + 1, high)
  File "/Users/milanbista/Documents/Cumberlands/Classes/MSCS-532-B01/Assignment3/quickSortComparison.py", line 30, in determin
```

This study empirically compares **Randomized Quicksort** and **Deterministic Quicksort** (using the first element as the pivot) across various input sizes and data distributions: randomly generated arrays, already sorted arrays, reverse-sorted arrays, and arrays with repeated elements.

We tested both algorithms on input sizes of 100, 1000, 10000, and 100000 elements, generating arrays for each distribution. Running times were measured to compare the performance of both algorithms under different conditions.

**Randomized Quicksort** consistently performed near its expected  $O(n \log n)$  time complexity for all distributions. Its random pivot selection ensures balanced partitions, leading to efficient performance in all cases.

On the other hand, **Deterministic Quicksort** showed  $O(n \log n)$  performance for random and repeated element arrays but degraded to  $O(n^2)$  on sorted and reverse-sorted arrays. The use of the first element as the pivot caused unbalanced partitions, which led to inefficient sorting in these cases.

The main difference between the algorithms is how they handle unbalanced partitions. Randomized Quicksort avoids this issue by selecting a random pivot, which ensures more balanced partitions. In contrast, deterministic quicksort's use of the first element as the pivot causes poor performance on sorted and reverse-sorted arrays due to unbalanced partitions.

Both algorithms performed similarly on arrays with repeated elements, as the partitions remained balanced. However, Randomized Quicksort still had a slight advantage due to its random pivot selection, which helped avoid worst-case scenarios.

The results confirm the theoretical expectations: **Randomized Quicksort** maintains efficient  $O(n \log n)$  performance across all distributions, while **Deterministic Quicksort** performs poorly on sorted and reverse-sorted arrays, resulting in  $O(n^2)$  time complexity. Randomized Quicksort is more resilient to different input distributions, particularly when handling sorted or reverse-sorted data.

## Part 2: Hashing with Chaining

### 1. Implementation

- In the part, I am implementing a hash table using chaining for collision resolution. I am using a suitable hash function to minimize collisions. The function will support insert, delete and search along with to String method to print the content of the hash table.



```
hashWithChaining.py > ...
1 import random
2
3 class HashTable:
4     def __init__(self, size=10):
5         self.size = size
6         self.table = [[] for _ in range(size)]
7         # Universal hash function parameters
8         self.p = 109345121
9         self.a = random.randint(1, self.p - 1)
10        self.b = random.randint(0, self.p - 1)
11
12    def _hash_function(self, key):
13        """Universal hashing function."""
14        return ((self.a * hash(key) + self.b) % self.p) % self.size
15
16    def insert(self, key, value):
17        """Insert a key-value pair into the hash table, replacing an existing value if the key already exists"""
18        index = self._hash_function(key)
19        bucket = self.table[index]
20
21        # Check if the key already exists and replace its value
22        for i, (k, v) in enumerate(bucket):
23            if k == key:
24                bucket[i] = (key, value)
25                return
26
27        # If key was not found, append a new key-value pair
28        bucket.append((key, value))
29
30    def search(self, key):
31        """Search for a value with the given key in the hash table"""
32        index = self._hash_function(key)
33        bucket = self.table[index]
34
35        # Search for the key in the bucket
36
37
38    def delete(self, key):
39        """Remove a key-value pair from the hash table."""
40        index = self._hash_function(key)
41        bucket = self.table[index]
42
43        # Remove key if it exists in the bucket
44        for i, (k, v) in enumerate(bucket):
45            if k == key:
46                del bucket[i]
47                return
48
49    def __str__(self):
50        """String representation for the hash table."""
51        return "\n".join(f"{i}: {bucket}" for i, bucket in enumerate(self.table))
52
53    # Usage example:
54    hash_table = HashTable(size=10)
55
56    # Insert different animals with their respective values
57    hash_table.insert("lion", 300)
58    hash_table.insert("tiger", 250)
59    hash_table.insert("elephant", 500)
60    hash_table.insert("giraffe", 150)
61    hash_table.insert("zebra", 100)
62    hash_table.insert("monkey", 75)
63    hash_table.insert("panda", 120)
64    hash_table.insert("koala", 90)
65    hash_table.insert("kangaroo", 200)
```

```
EXPLORER
ASSIGNMENTS3
> screenshots
- Assignment 3 Under...
hashWithChaining.py
quickSort.py
quickSortCompariso...

hashWithChaining.py > ...
1 import random

(base) milanbista@Milans-MacBook-Pro Assignment3 % python3 hashWithChaining.py
Hash Table After Inserting Animals:
0: [('zebra', 100), ('bear', 320)]
1: [('monkey', 75), ('koala', 90)]
2: [('panda', 120), ('leopard', 310)]
3: [('rhino', 450)]
4: [('tiger', 250), ('elephant', 500), ('wolf', 270), ('fox', 220)]
5: [('giraffe', 150), ('cheetah', 330)]
6: []
7: []
8: [('kangaroo', 200)]
9: [('lion', 300)]

Search Results:
lion: 300
tiger: 250
bear: 320
panda: 120
leopard: 310

Hash Table After Deletions:
0: [('zebra', 100), ('bear', 320)]
1: [('monkey', 75), ('koala', 90)]
2: [('leopard', 310)]
3: [('rhino', 450)]
4: [('elephant', 500), ('wolf', 270), ('fox', 220)]
5: [('giraffe', 150), ('cheetah', 330)]
6: []
7: []
8: [('kangaroo', 200)]
9: [('lion', 300)]

Search Results After Deletions:
tiger: None
panda: None
(base) milanbista@Milans-MacBook-Pro Assignment3 %
```



## 2. Analysis

In a hash table that assumes simple uniform hashing, the expected time complexity for search, insert, and delete operations is heavily influenced by the load factor  $\alpha = n/m$ , where  $n$  is the number of elements and  $m$  is the number of slots (or buckets). When the load factor is low, meaning there are fewer elements relative to the number of buckets, each bucket will contain only a small number of elements, resulting in  $O(1)$  time complexity for these operations. This ensures efficient performance.

However, as the load factor increases (i.e., when the table becomes more populated), the performance degrades. When  $\alpha$  becomes large, multiple elements hash to the same bucket, resulting in longer chains (if using chaining) or clusters (if using open addressing). This leads to an increase in the time taken for operations, with the expected time complexity for search, insert, and delete becoming proportional to the number of elements in a bucket, which in the worst case can be  $O(n)$ .

### Impact of Load Factor and Collision Resolution

The **load factor** plays a crucial role in determining the efficiency of hash table operations. A low load factor generally results in a small number of collisions, meaning the hash table operates close to its expected  $O(1)$  time for each operation. As the load factor increases, however, the number of collisions also increases, causing the performance to deteriorate due to the longer chains or clusters formed in the hash table.

In practice, if the load factor exceeds a threshold (often around 0.7), performance can degrade significantly, especially with chaining, as longer linked lists form in each bucket. This results in slower operations, potentially reaching  $O(n)$  in the worst case, where all elements collide into the same bucket.

### Strategies for Optimizing Hash Table Performance

To avoid performance degradation as the load factor increases, **dynamic resizing** is typically employed. When the load factor exceeds a predetermined threshold, the hash table is resized (usually doubled), and all elements are rehashed into the new table. This reduces the load factor

and ensures that the number of collisions is minimized. Resizing helps keep the time complexity of operations close to  $O(1)$ .

Additionally, using a **good hash function** is critical to evenly distribute the keys across the hash table's slots. A well-designed hash function minimizes the likelihood of collisions and helps maintain efficient operations, even as the load factor increases.

In summary, maintaining a low load factor and dynamically resizing the hash table as it grows are key strategies to ensure that the hash table remains efficient, with time complexities for search, insert, and delete operations staying close to  $O(1)$ . These strategies, combined with a good hash function, minimize collisions and optimize the hash table's performance.

## References

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

Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.