Algorithms and Data Structures

Assignment 3: Understanding algorithm efficiency and scalability

Nguyen The Duy Khanh

005019181

Table of Contents

1. Randomized Quicksort Analysis

- Each sort has its own use case, and the same can be said for QuickSort, where attackers can leverage its weakness to perform a DDOS attack and crash the server (e.g choosing the pivot poorly in a sorted array [1], a median of three killer sequence [2]). Therefore, in order to leverage the Quick Sort advantage, we need to resolve this weakness, and one of the solutions is using Random Pivot Selection[3], where we **increase the randomness and reduce the likelihood of consistently picking a poor pivot**.

a. Practical Implementation

- The Quick Sort algorithm with Randomized Pivot Selection has three steps:
  - **Step 1:** Choose a pivot (in this case, randomized pivot selection)

$$\mathbf{pivot} = \mathbf{L} + (\mathbf{random}() \mod (\mathbf{R} - \mathbf{L} + \mathbf{1}))$$

where L is the index of the first element in the range, and R is the index of the last element in the range.

```
# Step 1: Choose the pivot randomly by using random pivot selection
# https://www.baeldung.com/cs/randomized-quicksort
# since some algorithms for choosing pivot can be costly if not choosing
# correcly (e.g medium of 3 for fixed position can be leveraged
# https://programmingpraxis.com/2016/11/08/a-median-of-three-killer-sequence/)
pivot = array[random.randint(0, len(array) - 1) % (len(array))]
```

Figure 1: Randomized Pivot Selection

  - **Step 2:** Partition the array, which includes splitting and reordering the array so that all elements smaller than the pivot are on the left and all elements larger than the pivot are on the right.

```
# Step 2: Parition the array into two sides
leftArray = [element for element in array[1:] if element < pivot] # Smaller than the pivot
rightArray = [element for element in array[1:] if element >= pivot] # Larger than the pivot
```

Figure 2: Partition the array with Randomized Pivot Selection

- **Step 3:** After partitioning the array, return the pivot index to perform the exact implementation for both sides of the partitioned array

```
# Step 3: Repeat the same operation for left side and right side
return quickSort(leftArray) + [pivot] + quickSort(rightArray)
```

Figure 3: Quick Sort with left side and right side's partition

- As a result of Quick Sort with Randomized Pivot Selection

```
MSCS532_Assignment3 on  main [$!?] via  v3.12.5 on  kel.nguyen@vietnamtechsociety.org(us-east1)
❯ python3 main.py
Before sorting
3 2 1 2 2 1 1 1 2 3

After sorting
1 1 1 1 2 2 2 2 2 3
```

Figure 4: Quick Sort with left side and right side's partition

b. Theoretical analysis

- At each step, we partition the array based on the pivot elements, and each partition takes O(n) time complexity at the best-case scenario and average scenario where the sub-array is roughly balanced at each level, then the height of the recursive will be half for each level. Therefore, Quick Sort would have the following time complexity of Ω(n log n) and Θ(n log n).

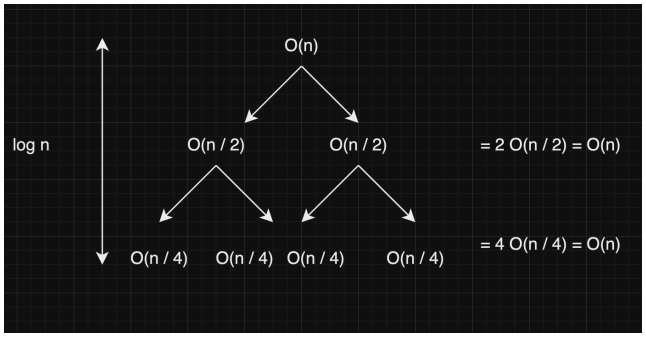

Figure 5: Recursion tree for Quick Sort best and average case complexity

However, if the array is already sorted and the pivot selection is poor, the sub-array won't be balanced, and the recursion tree will have n level, which leads to O(n^2)

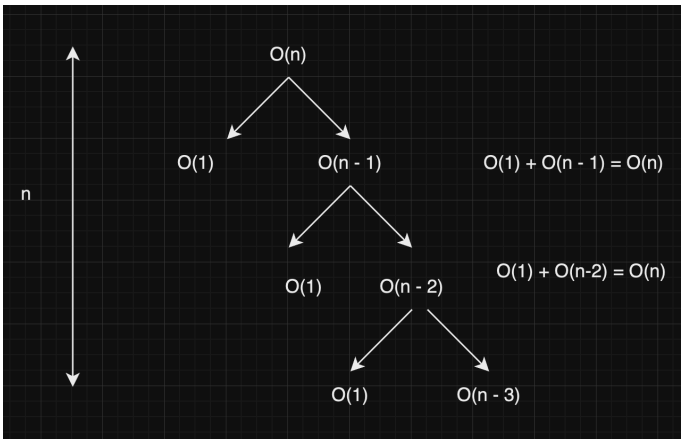

Figure 6: Recursion tree for Quick Sort's worst-case complexity

Since we are performing in-place sort for Quick Sort, it can achieve a space complexity of O(log n).

- The general recurrence relation is:

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

where k is the number of elements in the left partition.

- With the *Substitution Method[4],* assuming the k is roughly balanced which is k ≈ $\frac{n}{2}$, we will have the following equation when substituting k with the aforementioned recurrence relation:

$$T(n) = 2\,T(\tfrac{n}{2}) + cn$$

- However, we must prove that T(n) ≤ c n log n. Therefore,

$$T(n) \leq 2\,c\tfrac{n}{2} log\ \tfrac{n}{2} + cn$$

$$\leq cn\ log\ n$$

c. Comparison

- When experimenting with Randomized Quick Sort and Deterministic Quick Sort with different datasets, the difference between them can be clearly found:
    - With **random data** from n = 100 to 100,000 elements

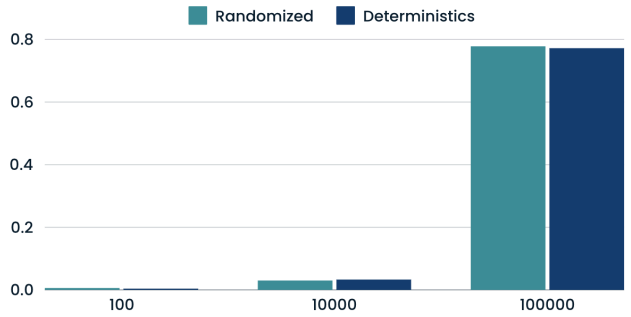

Figure 7: Random Dataset's Profiler with random data(CPU(s)))

- With **sorted order data** from n = 100 to 100,000 elements

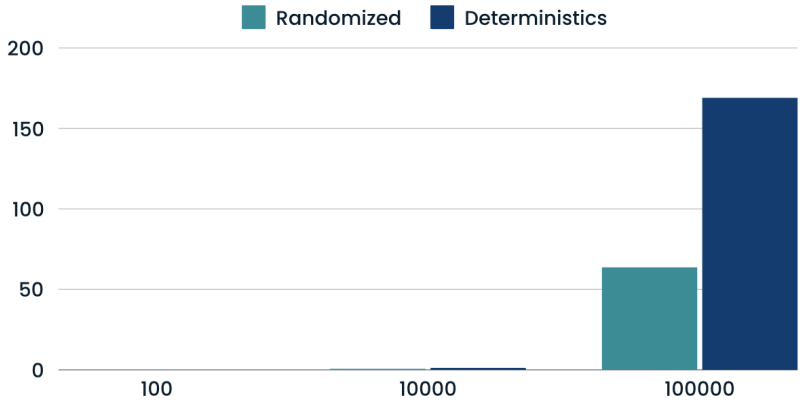

Figure 8: Random Dataset's Profiler with sorted order data (CPU(s)))

- With **reverse sorted order data** from n = 100 to 100,000 elements

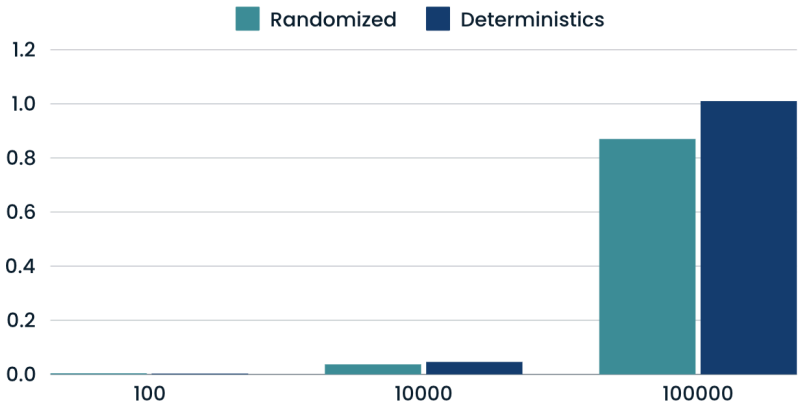

Figure 9: Random Dataset's Profiler with reverse sorted order data (CPU(s)))

- With **repeated elements** from n = 100 to 100,000 elements

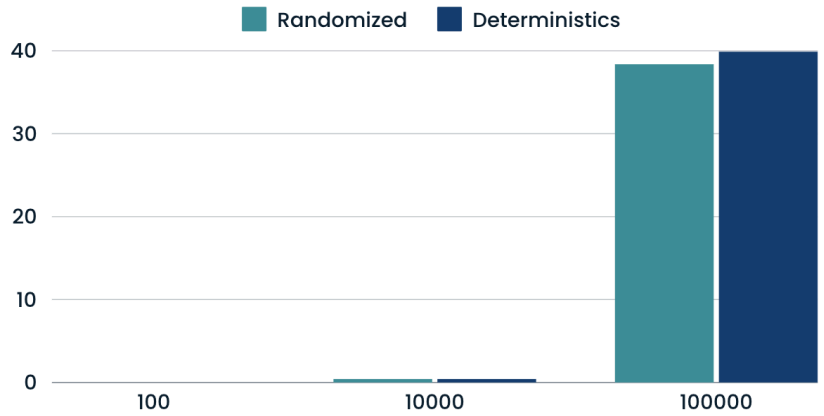

Figure 10: Random Dataset's Profiler with repeated elements(CPU(s)))

- As a result of the experiment, QuickSort with Randomized Pivot Selection has an increased time when sorting with sorted order data (from 0.08s to 64s for 100,000 elements). Even though the worst case is O(n^2), the randomness introduced by Randomized Pivot Selection has maintain the average complexity of Quick Sort to O(n log n). This can be confirmed when applied the same data to Determinisitcs Quick Sorth (from 0.08s to 167s). However, as we only increase the randomness in

choosing pivot, some edge cases such as data with repeated elements will behave the same with or without the pivot selection.

2. Hashing with Chaining

- Hash Map is a data structure that performs search, insertion, and removal of key-value pairs quickly by using the hashing concept[5], where each key is translated by a hash function into a distinct index in an array. However, in some cases, the index appears more than once. Therefore, to avoid that collision, we have the following techniques:
    - **Open Addressing [6]:** Collisions are handled by looking for the following empty space in the table (e.g, linear probing, double probing, etc.)
    - **Separate Chaining [7]:** Collisions are handled using a linked list of objects to include two keys if they hash to the same slot.

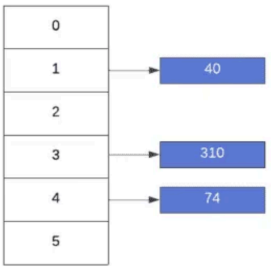

Figure 11: Random Dataset's Profiler with repeated elements(CPU(s)))

a. Practical Implementation

- Before analyzing each hashing algorithm, we need to understand the implementation of the hash table's operation with separate chaining under the assumption of a simple uniform hashing algorithm.
    - **Insert:** Before inserting any key, the bucket index needs to be determined by using the hashing function. If there are no elements at the calculated bucket index, the function will create a linked list node. However, if there is a

previous element, the function will insert the linked list node at the beginning of the linked list. As there are no extra operations, the time complexity is O(1).

```python
# Insert will insert the key to the hash map
# after determine the hashing index
# However, if there is a collision, the key will be added
# at the beginning of the linked list
# Time Complexity: O(1)
def insert(self, key: int, value: any):
  bucketIndex = self.hashingKey(key)
  newNode = Node(key = key, value = value)
  if self.array[bucketIndex] is None:
    self.array[bucketIndex] = newNode
  else:
    newNode.next = self.array[bucketIndex]
    self.array[bucketIndex] = newNode
```

Figure 12: Insert with Separate Chaining

- **Search:** Before searching any key, the bucket index needs to be determined by using the hashing function. Assuming the key is inserted before, the function will perform a search on the linked list, resulting in a time complexity of O(n) in the worst case if all the inserted elements have the same hashed index.

```python
# search will search the key from the hash map
# after determine the hashing index
# The more distributed for the data we have the better. We can use bisect
# to improve to logn
# https://docs.python.org/3/library/bisect.html
# Time Complexity: O(n) (best is O(1))
def search(self, key: int) -> int:
  bucketIndex = self.hashingKey(key)
  bucketHead = self.array[bucketIndex]

  while bucketHead is not None:
    if bucketHead.key == key:
      return bucketHead.value
    bucketHead = bucketHead.next

  return -1
```

Figure 13: Search with Separate Chaining

- **Remove:** Before removing any key, the bucket index needs to be determined by using the hashing function. Assuming the key is inserted before, the function will perform a search on the linked list and remove it if the function

is able to find the element. As the removing operation is O(1) but the search is O(n), the overall time complexity os O(n).

```
# search will remove the key from the hash map
# after determine the hashing index
# Time Complexity: O(n) (best is O(1))
def remove(self, key:int) -> None:
    bucketIndex = self.hashingKey(key)
    bucketHead = self.array[bucketIndex]

    if bucketHead is None:
        print("The key does not exist in table", key)
        return

    if bucketHead.key == key:
        self.array[bucketIndex] = bucketHead.next
        return

    previousNode = None
    currentNode = self.array[bucketIndex]
    while currentNode is not None:
        if currentNode.key == key:
            break
        previousNode = currentNode
        currentNode = currentNode.next

    if currentNode is None:
        print("The key does not exist in table", key)
    else:
        previousNode.next = currentNode.next
    return
```

Figure 14: Remove with Separate Chaining

b. Theoretical analysis

- In the hash table, the load factor is a crucial factor for determining the performance of various operations.

$$Load\ factor\ (\alpha) = \frac{n}{m}$$

where n is the number of elements that are currently in the table, and m is the bucket slots of the table.

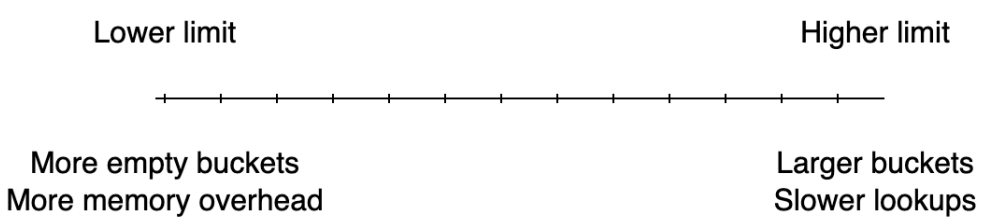

Figure 15: Load Factor's Impact

- The larger the load factor, the larger the probability of the next element's collision via the hash function [7]. However, resolving these collisions will be costly (e.g, separate chaining with an array can have a time complexity of O(n) for search, remove, and insert as a worst-case - resizing). Therefore, the lower the load factor, the lower the time the related operations are executed. Generally, the default load factor (0.75) offers a good trade-off between time and space costs.
- To maintain a low load factor and minimize collisions, we can utilize the following algorithms:
    - **Dynamic Resizing [8]:** since the load factor depends on the bucket slots of the table, we can increase or decrease the number of bucket slots by double based on the load factors. (e.g. we can rehash the entire functions when the load factor is 0.75). However, it will come with a cost of inefficient resizing if the table is frequently resized, and it will incur overhead and complexity of the hash table's implementation depending on the data structures (e.g., resizing the array can cause additional time complexity of O(n) [9])
    - **Hashing function with good distributions:** since the bucket index depends on the hash function, we can increase the randomness in the selection of hash functions, which reduces the likelihood of collisions. An example would be using Universal Hashing, which uses a family of hash functions at random [10]

3. References

- [1] *Pavel Minaev* (2009). Is Quick Sort a potential security risk? Stackoverflow. https://stackoverflow.com/questions/1527136/is-quicksort-a-potential-security-risk
- [2] David Muser (2016). A Median-Of-Three-Killer Sequence. Programming Praxis. https://programmingpraxis.com/2016/11/08/a-median-of-three-killer-sequence/

- [3] *Said Sryheni Grzegorz Piwowarek (2024). Understanding the randomized Quick Sort. BaelDung.* *https://www.baeldung.com/cs/randomized-quicksort*
- *[4] GeeksForGeeks* (2024). How to analyze the complexity of the recurrence equation https://www.geeksforgeeks.org/how-to-analyse-complexity-of-recurrence-relation/
- [5] *GeeksForGeeks (*2024). What is hashing? https://www.geeksforgeeks.org/what-is-hashing/
- [6] *Alexander Obregon* (2023). Exploring Open-Addresses Techniques in Java. Medium. https://medium.com/@AlexanderObregon/exploring-open-addressing-techniques-in-java-linear-probing-quadratic-probing-and-double-hashing-3559629cc8ff
- [7] *Harshit Kumar* (2024). Recursion tree method. Scaler Topics. https://www.scaler.com/topics/data-structures/recursion-tree-method/
- [8] Shrikant Sharma(2017). What is the purpose of load factor in hash tables? Quora. https://www.quora.com/Whats-the-purpose-of-load-factor-in-hash-tables
- [9] Karoly Horvath (2014). Time complexity of append operation in simple array. Stackoverflow. https://stackoverflow.com/questions/22173998/time-complexity-of-append-operation-in-simple-array
- [10] Ashish Kumar Gupta (2023). Introducing to universal hashing in data structures GeeksForGeeks https://www.geeksforgeeks.org/introduction-to-universal-hashing-in-data-structure/