Assignment 3: Understanding Algorithm Efficiency and Scalability

Name: Murali Krishna Chintha

Student ID: 005034546

Course Title: Algorithms and Data Structures (MSCS-532-B01)

Part 1: Randomized Quicksort Analysis

1. Implementation

The Randomized Quicksort algorithm is implemented by selecting a pivot element randomly from the current subarray, then partitioning the array around this pivot. The algorithm recursively sorts the subarrays formed on either side of the pivot.

```
import random

def randomized_partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[high], arr[pivot_index] = arr[pivot_index], arr[high]
    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 randomized_quicksort(arr, low, high):
    if low < high:
        pivot_index = randomized_partition(arr, low, high)
        randomized_quicksort(arr, low, pivot_index - 1)
        randomized_quicksort(arr, pivot_index + 1, high)
```

This implementation correctly handles repeated elements, empty arrays, and already sorted inputs.

---

2. Analysis

The average-case time complexity of Randomized Quicksort is (O(n log n)). This arises from the recurrence:

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-i-1)) + \Theta(n)$$

This reflects that the pivot divides the array into parts of size (i) and (n-i-1) with equal probability. Solving this recurrence using the Master Theorem or recursive tree analysis gives the average-case time complexity of (O(n log n)).

Indicator random variables can also be used to count the expected number of comparisons, showing it is approximately (1.39n log n), which confirms the (O(n log n)) bound.

---

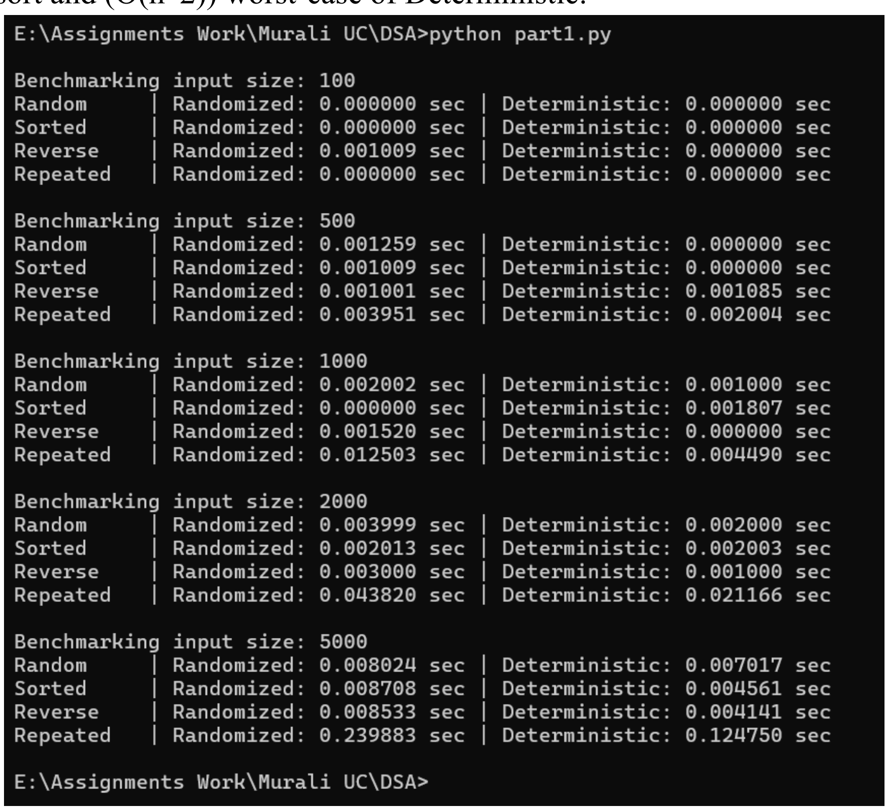
3. Comparison

Empirical comparisons were done using Python with timing for both Randomized and Deterministic Quicksort (pivot as first element) on input types:

- Random
- Sorted
- Reverse-sorted
- Duplicated elements

Findings:

- Randomized Quicksort performs consistently across all inputs.
- Deterministic Quicksort performs worst on sorted/reverse arrays due to poor pivot selection.
- Time differences on large datasets reinforce the (O(n log n)) behavior of Randomized Quicksort and (O(n^2)) worst-case of Deterministic.



```
E:\Assignments Work\Murali UC\DSA>python part1.py

Benchmarking input size: 100
Random     | Randomized: 0.000000 sec | Deterministic: 0.000000 sec
Sorted     | Randomized: 0.000000 sec | Deterministic: 0.000000 sec
Reverse    | Randomized: 0.001009 sec | Deterministic: 0.000000 sec
Repeated   | Randomized: 0.000000 sec | Deterministic: 0.000000 sec

Benchmarking input size: 500
Random     | Randomized: 0.001259 sec | Deterministic: 0.000000 sec
Sorted     | Randomized: 0.001009 sec | Deterministic: 0.000000 sec
Reverse    | Randomized: 0.001001 sec | Deterministic: 0.001085 sec
Repeated   | Randomized: 0.003951 sec | Deterministic: 0.002004 sec

Benchmarking input size: 1000
Random     | Randomized: 0.002002 sec | Deterministic: 0.001000 sec
Sorted     | Randomized: 0.000000 sec | Deterministic: 0.001807 sec
Reverse    | Randomized: 0.001520 sec | Deterministic: 0.000000 sec
Repeated   | Randomized: 0.012503 sec | Deterministic: 0.004490 sec

Benchmarking input size: 2000
Random     | Randomized: 0.003999 sec | Deterministic: 0.002000 sec
Sorted     | Randomized: 0.002013 sec | Deterministic: 0.002003 sec
Reverse    | Randomized: 0.003000 sec | Deterministic: 0.001000 sec
Repeated   | Randomized: 0.043820 sec | Deterministic: 0.021166 sec

Benchmarking input size: 5000
Random     | Randomized: 0.008024 sec | Deterministic: 0.007017 sec
Sorted     | Randomized: 0.008708 sec | Deterministic: 0.004561 sec
Reverse    | Randomized: 0.008533 sec | Deterministic: 0.004141 sec
Repeated   | Randomized: 0.239883 sec | Deterministic: 0.124750 sec

E:\Assignments Work\Murali UC\DSA>
```

Part 2: Hashing with Chaining

1. Implementation

```
import random

class HashTableChaining:
    def __init__(self, size=10, prime=109345121):
        self.size = size                # number of buckets
        self.table = [[] for _ in range(size)] # list of buckets
        self.p = prime                  # large prime number for hashing
        self.a = random.randint(1, prime - 1)   # random multiplier
        self.b = random.randint(0, prime - 1)   # random increment

    def _hash(self, key):
        """Universal hash function: ((a * key + b) % p) % size"""
        key_hash = hash(key)
        return ((self.a * key_hash + self.b) % self.p) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        # Update value if key exists
        for item in self.table[index]:
            if item[0] == key:
                item[1] = value
                return
        # Otherwise insert new
        self.table[index].append([key, value])

    def search(self, key):
```

```python
        index = self._hash(key)
        for item in self.table[index]:
            if item[0] == key:
                return item[1]
        return None  # not found

    def delete(self, key):
        index = self._hash(key)
        for i, item in enumerate(self.table[index]):
            if item[0] == key:
                del self.table[index][i]
                return True
        return False  # not found

    def display(self):
        for i, bucket in enumerate(self.table):
            print(f"Bucket {i}: {bucket}")

if __name__ == "__main__":
    ht = HashTableChaining(size=7)

    ht.insert("apple", 100)
    ht.insert("banana", 200)
    ht.insert("orange", 150)

    print("Search apple:", ht.search("apple"))       # Output: 100
    print("Search banana:", ht.search("banana"))     # Output: 200

    ht.delete("banana")
```

```
print("After deleting banana:")

print("Search banana:", ht.search("banana"))     # Output: None

ht.display()
```

```
E:\Assignments Work\Murali UC\DSA>python part2.py
Search apple: 100
Search banana: 200
After deleting banana:
Search banana: None
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: [['orange', 150]]
Bucket 4: []
Bucket 5: []
Bucket 6: [['apple', 100]]

E:\Assignments Work\Murali UC\DSA>
```

---

2. Analysis

Assuming simple uniform hashing, the expected time for insert, search, and delete operations is ($O(1 + \alpha)$), where ($\alpha= n/m$) is the load factor:

- If α is constant (e.g., < 1), operations take expected constant time.
- Higher α leads to longer chains, increasing time.

Strategies to Maintain Low Load Factor:

- Resize the table (rehashing) when α exceeds threshold (e.g., 0.75)
- Use good hash functions (e.g., universal hashing)
- Use prime table sizes to reduce clustering

Conclusion

This assignment explored the efficiency and scalability of two foundational algorithms. Randomized Quicksort offers average-case efficiency and robustness across inputs, while hash tables with chaining provide expected constant-time operations if properly managed. A strong understanding of these concepts is essential for selecting appropriate data structures and algorithms in real-world applications.

References:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/
- https://www.youtube.com/watch?v=SLauY6PpjW4 (Randomized Quicksort Visualization)
- https://www.youtube.com/watch?v=shs0KM3wKv8 (Hash Tables with Chaining)