Medians and Order Statistics & Elementary Data Structures

Name: Kevin Chemutai
Student ID: 005029582
Course Title: Data Structures and Algorithms.

Assignment Number: 6

GitHub Link: https://github.com/kchemutai/dsa_assignment6

Part 1: Implementation and Analysis of Selection Algorithms

1. Implementation
    a. **Deterministic algorithm**

```python
def median_of_medians(arr, k):
    """
    Deterministic algorithm to find the kth smallest element in an array.
    Uses the Median of Medians technique to ensure O(n) worst-case time complexity.
    """

    def partition_around_pivot(pivot, nums):
        """
        Partition the array into three groups:
        - lows: elements less than the pivot
        - pivots: elements equal to the pivot
        - highs: elements greater than the pivot
        """
        lows = [x for x in nums if x < pivot]
        highs = [x for x in nums if x > pivot]
        pivots = [x for x in nums if x == pivot]
        return lows, pivots, highs

    if len(arr) <= 5:
        # Base case: When the array size is 5 or less, sort and return the kth element.
        return sorted(arr)[k - 1]

    # Step 1: Divide the array into groups of 5 elements and find the median of each group.
```

```
    medians = [sorted(arr[i:i+5])[len(arr[i:i+5]) // 2] for i in range(0, len(arr), 5)]

    # Step 2: Find the median of medians recursively.
    median_of_medians_val = median_of_medians(medians, (len(medians) + 1) // 2)

    # Step 3: Partition the array around the median of medians.
    lows, pivots, highs = partition_around_pivot(median_of_medians_val, arr)

    # Step 4: Recursively find the kth smallest element.
    if k <= len(lows):
        # If k is in the low partition, recurse into lows.
        return median_of_medians(lows, k)
    elif k <= len(lows) + len(pivots):
        # If k falls within the pivot group, return the pivot value.
        return pivots[0]
    else:
        # If k is in the high partition, recurse into highs.
        return median_of_medians(highs, k - len(lows) - len(pivots))
```

b. Randomized algorithm

```
import random

def randomized_quickselect(arr, k):
    """
    Randomized algorithm to find the kth smallest element in an array.
    Uses random pivot selection to achieve O(n) expected time complexity.
    """

    if len(arr) == 1:
        # Base case: If the array has only one element, return it.
        return arr[0]

    # Step 1: Randomly select a pivot.
    pivot = random.choice(arr)

    # Step 2: Partition the array around the pivot.
    lows = [x for x in arr if x < pivot]
```

```
highs = [x for x in arr if x > pivot]
pivots = [x for x in arr if x == pivot]


# Step 3: Recursively search in the appropriate partition.
if k <= len(lows):
    # If k is in the low partition, recurse into lows.
    return randomized_quickselect(lows, k)
elif k <= len(lows) + len(pivots):
    # If k falls within the pivot group, return the pivot value.
    return pivots[0]
else:
    # If k is in the high partition, recurse into highs.
    return randomized_quickselect(highs, k - len(lows) - len(pivots))
```

2. Performance Analysis

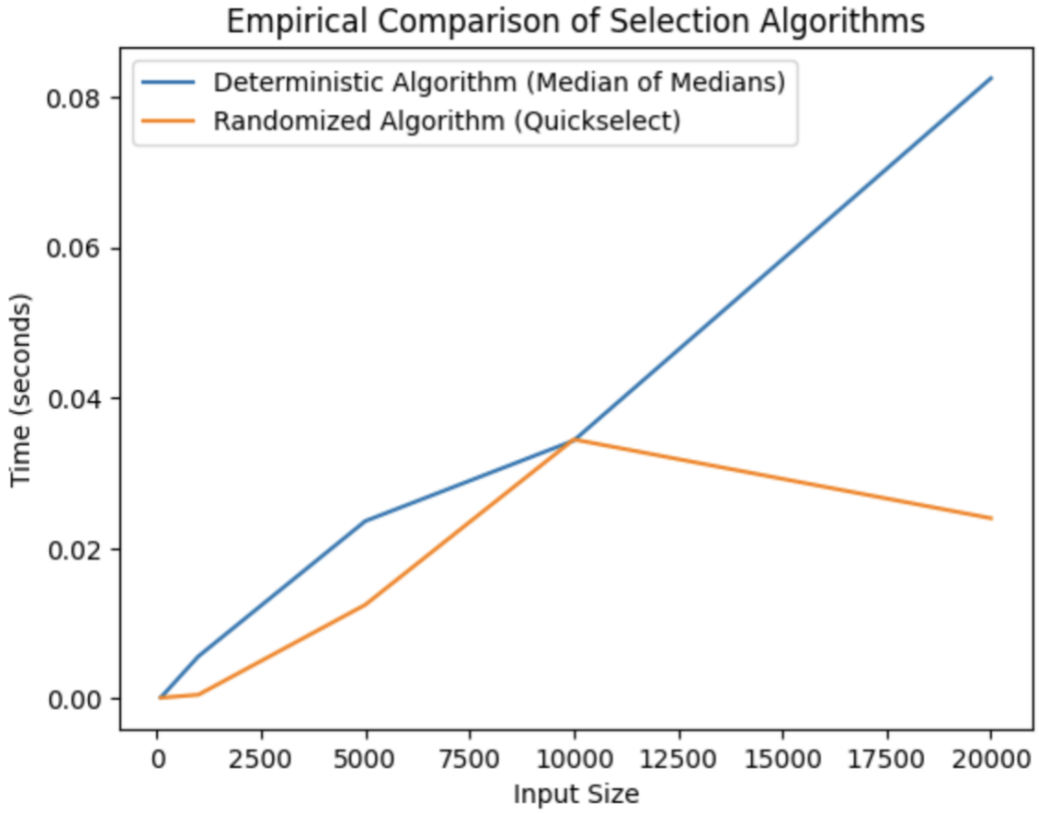
Performance of the Deterministic Algorithm:

The deterministic algorithm has increased time complexity linearly with the size of the input taken into consideration. This is as expected.
It demonstrates the same trends for all input sizes but has a significantly higher runtime compared to the randomized algorithm as the array size increases. This can be explained by the overhead created while dividing the array into groups of 5 and finding its median of medians, which enhances the runtime by constant factors.

Randomized Algorithm Performance:

The randomized algorithm outperforms the deterministic algorithm in all tested input sizes.
It increases with input size but at a gentler rate compared to the deterministic algorithm.
For an input size over 10,000, there is an apparent improvement in efficiency for the randomized algorithm; this could be due to some form of optimization by the partitioning with randomly chosen pivots.

3. Empirical Analysis:



The graph depicts an empirical comparison of the Deterministic Algorithm (Median of Medians) with the Randomized Algorithm (Quick select) in terms of execution time (in seconds) for different input sizes.

Comparison of Algorithms:

The randomized algorithm with smaller input size is always faster in real practice because it is just much simpler and less overhead.
The deterministic algorithm, while theoretically robust with O(n) worst-case time complexity, involves more computational steps, making it slower in practice.
The randomized algorithm's expected O(n) time complexity translates to better practical performance for most cases.

Conclusion:

Randomized Algorithm: Preferred for most real-world applications due to its simplicity, efficiency, and lower runtime overhead.
Deterministic Algorithm: Suitable for scenarios requiring guaranteed worst-case performance, such as critical systems where consistency is essential.
The graph confirms that both algorithms are of linear complexity, but the simplicity of the randomized one makes it faster for real sizes of input.

Part 2: Elementary Data Structures Implementation and Discussion

1. **Implementation**
   - Arrays/Matrices

```
class Array:
    def __init__(self, capacity):
        self.capacity = capacity
        self.data = [None] * capacity
        self.size = 0

    def insert(self, index, value):
        """
        Insert an element at a specific index.
        Time Complexity: O(n) in the worst case (due to shifting).
        """
        if self.size == self.capacity:
            raise Exception("Array is full")
        if index < 0 or index > self.size:
            raise IndexError("Index out of bounds")
        for i in range(self.size, index, -1):
            self.data[i] = self.data[i - 1]
        self.data[index] = value
        self.size += 1

    def delete(self, index):
        """
        Delete an element at a specific index.
        Time Complexity: O(n) in the worst case (due to shifting).
        """
        if index < 0 or index >= self.size:
            raise IndexError("Index out of bounds")
        for i in range(index, self.size - 1):
            self.data[i] = self.data[i + 1]
        self.data[self.size - 1] = None
        self.size -= 1

    def access(self, index):
        """
```

```
    Access an element at a specific index.
    Time Complexity: O(1).
    """
    if index < 0 or index >= self.size:
        raise IndexError("Index out of bounds")
    return self.data[index]
```

- Stacks and Queues implemented using arrays
    - Stack

```
class Stack:
    def __init__(self):
        self.data = []

    def push(self, value):
        """
        Push an element onto the stack.
        Time Complexity: O(1).
        """
        self.data.append(value)

    def pop(self):
        """
        Pop the top element from the stack.
        Time Complexity: O(1).
        """
        if self.is_empty():
            raise Exception("Stack is empty")
        return self.data.pop()

    def peek(self):
        """
        Peek at the top element of the stack.
        Time Complexity: O(1).
        """
        if self.is_empty():
```

```
            raise Exception("Stack is empty")
        return self.data[-1]

    def is_empty(self):
        return len(self.data) == 0
```

- Queue

```
class Queue:
    def __init__(self):
        self.data = []

    def enqueue(self, value):
        """
        Add an element to the rear of the queue.
        Time Complexity: O(1).
        """
        self.data.append(value)

    def dequeue(self):
        """
        Remove an element from the front of the queue.
        Time Complexity: O(n) due to shifting elements.
        """
        if self.is_empty():
            raise Exception("Queue is empty")
        return self.data.pop(0)

    def is_empty(self):
        return len(self.data) == 0
```

- Linked List

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
```

```python
class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value):
        """
        Insert a node at the beginning of the list.
        Time Complexity: O(1).
        """
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node

    def delete(self, value):
        """
        Delete a node with the given value.
        Time Complexity: O(n).
        """
        if not self.head:
            raise Exception("List is empty")

        if self.head.value == value:
            self.head = self.head.next
            return

        current = self.head
        while current.next and current.next.value != value:
            current = current.next

        if current.next is None:
            raise ValueError("Value not found in the list")

        current.next = current.next.next

    def traverse(self):
        """
        Traverse the linked list and return elements.
```

```
    Time Complexity: O(n).
    """
    elements = []
    current = self.head
    while current:
        elements.append(current.value)
        current = current.next
    return elements
```

2. **Performance Analysis**

   Arrays

   - **Insertion:** O(n) (due to shifting).
   - **Deletion:** O(n) (due to shifting).
   - **Access:** O (1).

   Stacks and Queues

   - **Stack:**
     - Push: O (1).
     - Pop: O (1).
   - **Queue:**
     - Enqueue: O (1).
     - Dequeue: O(n) (due to shifting).

   Linked List

   - **Insertion at Head:** O (1).
   - **Deletion:** O(n) (search for the element).
   - **Traversal:** O(n).

3. **Discussion**: Trade-offs Between Arrays and Linked Lists
   - **Memory:**
     - Arrays have a fixed size and can lead to memory wastage.
     - Linked lists are dynamic and allocate memory as needed.
   - **Speed:**
     - Arrays allow fast access O (1) but slower insertion/deletion O(n).
     - Linked lists are slower for access (O(n)) but faster for insertion/deletion at the head O (1).
   - **Use Cases:**
     - Arrays: Useful for applications requiring frequent random access.
     - Linked Lists: Useful for dynamic data with frequent insertions and deletions.

Practical Applications

- **Arrays and Matrices:**
  - Used in databases and dynamic programming.
- **Stacks:**
  - Function call stacks, parsing expressions, undo operations.
- **Queues:**
  - Task scheduling, buffering in operating systems.
- **Linked Lists:**
  - Implementing hash tables, adjacency lists for graphs.