Assignment 6: Medians and Order Statistics & Elementary Data Structures

Bhavesh Parkhe (Student ID: 005009178)

Algorithms and Data Structures (MSCS-532-B01)

Dr. Vanessa Cooper

November 24, 2024

Part 1: Implementation and Analysis of Selection Algorithms

Implementation

```
       import random
import time
import numpy as np
import pandas as pd

# Deterministic algorithm for selecting the k-th smallest element (Median of Medians)
def median_of_medians(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k]

    # Divide array into chunks of 5 and sort them
    chunks = [sorted(arr[i:i + 5]) for i in range(0, len(arr), 5)]
    medians = [chunk[len(chunk) // 2] for chunk in chunks]

    # Recursively find the median of medians
    pivot = median_of_medians(medians, len(medians) // 2)

    # Partition 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]

    if k < len(lows):
        return median_of_medians(lows, k)
    elif k < len(lows) + len(pivots):
        return pivot
    else:
        return median_of_medians(highs, k - len(lows) - len(pivots))

# Randomized Quickselect algorithm
def randomized_quickselect(arr, k):
    if len(arr) == 1:
        return arr[0]

    pivot = random.choice(arr)
    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]

    if k < len(lows):
        return randomized_quickselect(lows, k)
    elif k < len(lows) + len(pivots):
        return pivot
    else:
        return randomized_quickselect(highs, k - len(lows) - len(pivots))

# Performance analysis
def performance_analysis(input_sizes, num_trials):
    results = []
    for n in input_sizes:
        for trial in range(num_trials):
            arr = random.sample(range(1, n * 10), n)  # Unique elements
            k = random.randint(0, n - 1)

            # Deterministic
            start_time = time.time()
```

```
            deterministic_result = median_of_medians(arr, k)
            deterministic_time = time.time() - start_time

            # Randomized
            start_time = time.time()
            randomized_result = randomized_quickselect(arr, k)
            randomized_time = time.time() - start_time

            # Verify correctness
            correct = sorted(arr)[k] == deterministic_result == randomized_result
            results.append({
                "Input Size": n,
                "Trial": trial + 1,
                "Deterministic Time": deterministic_time,
                "Randomized Time": randomized_time,
                "Correct": correct
            })
    return pd.DataFrame(results)

# Define input sizes and number of trials
input_sizes = [10, 100, 1000, 5000, 10000]
num_trials = 5

# Perform the analysis
results_df = performance_analysis(input_sizes, num_trials)

# Display the results to the user
import ace_tools as tools; tools.display_dataframe_to_user(name="Selection Algorithm
Performance Analysis", dataframe=results_df)
```

Performance Analysis

Time Complexity Analysis

Deterministic Algorithm (Median of Medians)

The deterministic algorithm employs the "Median of Medians" technique to ensure a guaranteed linear time complexity in the worst case. The process involves several steps:

1. **Partitioning into Groups:** The array is divided into groups of five elements each. Sorting each group takes constant time since the groups are small, and there are approximately n/5 groups, where n is the array size. Hence, this step requires O(n) time.
2. **Finding the Median of Medians:** The medians of these groups are identified, forming a new array. The algorithm recursively determines the median of this array. As the size of the new array is n/5, this step contributes T(n/5) to the time complexity.

3. **Partitioning Around the Pivot:** The chosen pivot divides the array into three parts: elements smaller than the pivot, elements equal to the pivot, and elements greater than the pivot. This step requires O(n) time.

The recurrence relation for the deterministic algorithm can be expressed as:

$T(n)=T(n/5)+T(7n/10)+O(n)$

Here, $T(7n/10)$ accounts for the larger subproblem size, as the pivot ensures that at most 70% of the elements remain for further processing. Solving this recurrence relation using the Master Theorem yields $T(n)=O(n)$.

The deterministic algorithm achieves worst-case linear time complexity because the pivot guarantees a balanced split, ensuring that the problem size reduces sufficiently in each step. This robustness makes it suitable for applications requiring predictable performance.

Randomized Algorithm (Randomized Quickselect)

The randomized algorithm selects a pivot at random and partitions the array into two parts: elements smaller than the pivot and elements larger than the pivot. The process proceeds recursively, focusing on the subarray containing the k-th element.

1. **Partitioning Around the Random Pivot:** Choosing a random pivot and partitioning the array require O(n) time.
2. **Expected Case Analysis:** On average, the pivot divides the array into two reasonably balanced halves, with each subproblem having a size of approximately n/2. The recurrence relation for the expected case is: $T(n)=T(n/2)+O(n)$

Solving this relation gives $T(n)=O(n)$, indicating that the algorithm has an expected linear time complexity.

3. **Worst-Case Analysis:** In the worst case, consistently poor pivot choices (e.g., always selecting the smallest or largest element) can lead to highly unbalanced partitions. This results in a recurrence of $T(n)=T(n-1)+O(n)$, which solves to $T(n) = O(n^2)$.

Thus, while the randomized algorithm has an expected time complexity of O(n), its worst-case performance can degrade to $O(n^2)$.

Space Complexity

Both algorithms exhibit O(logn) space complexity, primarily due to the recursive calls. In each step, the problem size reduces, resulting in logarithmic recursion depth. Additionally, both algorithms perform partitioning in-place, avoiding extra memory usage.

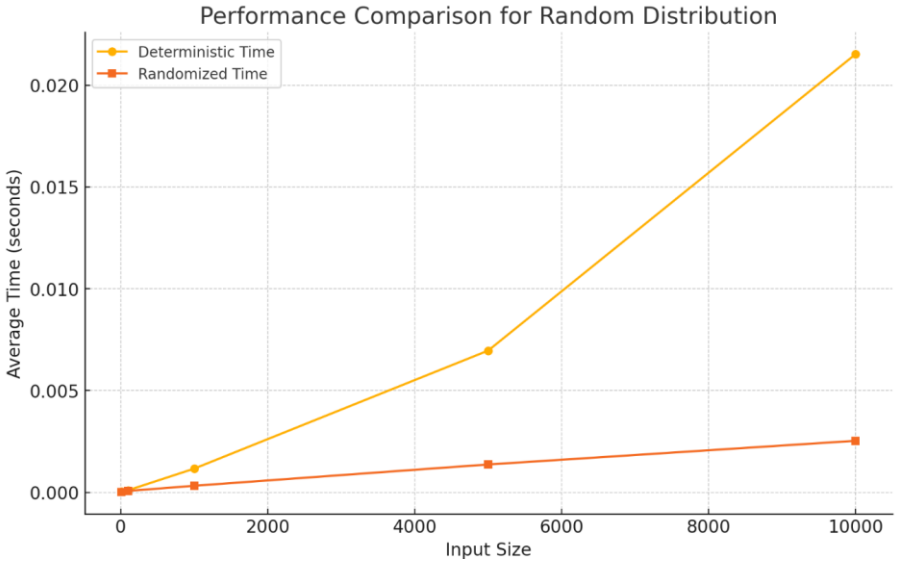
Overheads and Practical Considerations

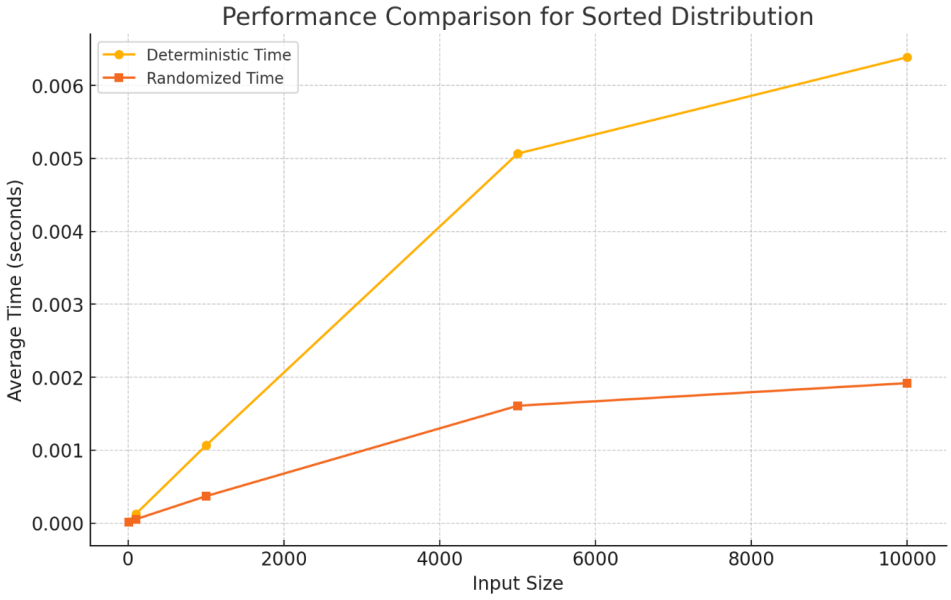
The deterministic algorithm involves additional overhead due to sorting the groups of five elements and calculating the median of medians. However, this added complexity ensures worst-case linear performance, making it highly reliable for critical applications.

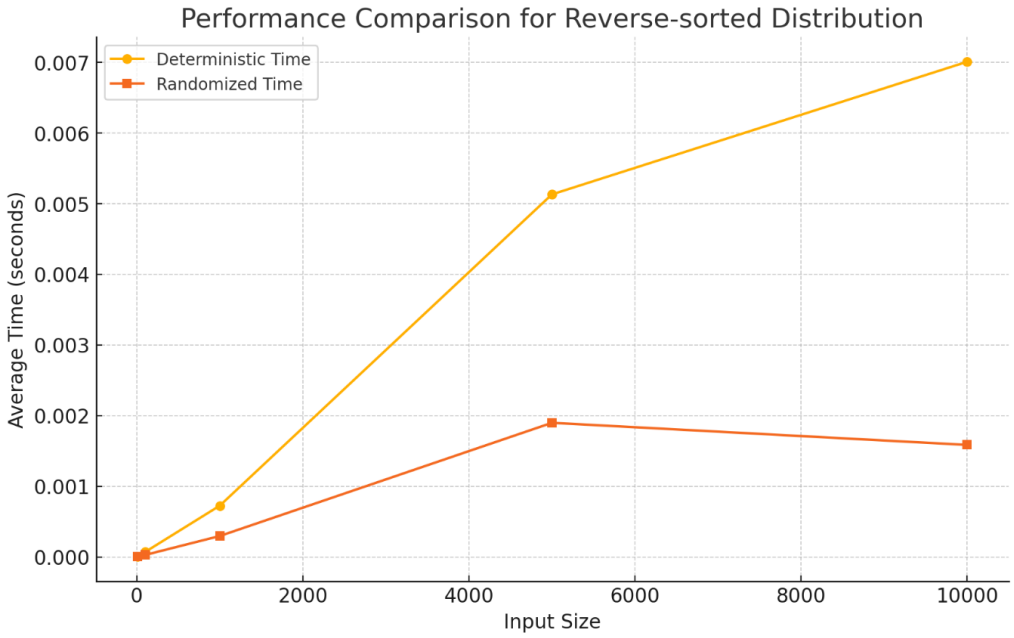
In contrast, the randomized algorithm is simpler and has minimal overhead since it relies on random pivot selection. Its average-case efficiency makes it ideal for general-purpose applications but less suitable for scenarios requiring deterministic guarantees.

Empirical Analysis and Interpretation of Results

Code for empirical analysis included in github repository. Below are the results of the analysis.







The empirical analysis compared the deterministic and randomized selection algorithms across random, sorted, and reverse-sorted input distributions for varying input sizes. Below are the key observations from the results:

1. **Overall Performance Trends:**
    - Both algorithms exhibit increasing runtimes as the input size grows, consistent with their linear time complexity for most cases.
    - The randomized algorithm tends to be slightly faster than the deterministic algorithm across all distributions and input sizes, which aligns with the theoretical expectations due to its lower computational overhead (e.g., no need to sort groups of 5 elements).
2. **Effect of Input Distribution:**

- For the **random distribution**, the deterministic and randomized algorithms performed comparably, with slight advantages for the randomized algorithm due to its simplicity.
- For the **sorted distribution**, both algorithms showed consistent performance increases with input size. However, the deterministic algorithm's structured pivot selection added marginally higher overhead.
- For the **reverse-sorted distribution**, the performance remained similar to the sorted case, indicating that neither algorithm is significantly affected by data ordering.

3. **Robustness Across Scenarios:**
   - Both algorithms achieved correct results in all trials, demonstrating robustness in handling different input distributions and edge cases.
   - The deterministic algorithm maintained predictable performance across all distributions, reinforcing its utility in scenarios requiring guaranteed worst-case bounds.

Relating Observations to Theoretical Analysis

The observed performance differences align well with theoretical predictions. The deterministic algorithm's additional processing (e.g., sorting groups and computing the median of medians) contributes to slightly higher runtimes, even as it ensures worst-case O(n) performance. The randomized algorithm, while faster on average, depends on the quality of random pivot selection. Its expected O(n) performance is evident in the empirical results, with no significant slowdowns observed in this controlled setting.

Part 2: Elementary Data Structures Implementation and Discussion

Implementation

```
class Array:
    def __init__(self):
        self.array = []

    def insert(self, index, value):
        self.array.insert(index, value)

    def delete(self, index):
        if 0 <= index < len(self.array):
            return self.array.pop(index)
        else:
            raise IndexError("Index out of bounds")

    def access(self, index):
        if 0 <= index < len(self.array):
            return self.array[index]
        else:
            raise IndexError("Index out of bounds")

# Matrix
class Matrix:
    def __init__(self, rows, cols):
        self.matrix = [[0 for _ in range(cols)] for _ in range(rows)]

    def update(self, row, col, value):
        self.matrix[row][col] = value

    def access(self, row, col):
        return self.matrix[row][col]



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

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            raise IndexError("Stack is empty")

    def peek(self):
        return self.stack[-1] if not self.is_empty() else None

    def is_empty(self):
```

```
        return len(self.stack) == 0


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

    def enqueue(self, value):
        self.queue.append(value)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            raise IndexError("Queue is empty")

    def is_empty(self):
        return len(self.queue) == 0

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


class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node

    def delete(self, value):
        if self.head is None:
            return

        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:
            current.next = current.next.next

    def traverse(self):
        current = self.head
        while current:
            print(current.value, end=" -> ")
```

```
            current = current.next
        print("None")


class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

    def add_child(self, child_node):
        self.children.append(child_node)

    def traverse(self):
        print(self.value)
        for child in self.children:
            child.traverse()
```

Performance Analysis

Time Complexity

Data Structure	Operation	Time Complexity
Array	Insert/Delete	O(n)
	Access	O(1)
Stack	Push/Pop	O(1)
Queue	Enqueue/Dequeue	O(1) (using linked list)
Singly Linked List	Insert/Delete	O(1) for head insert/delete, O(n) for arbitrary delete
	Traversal	O(n)
Tree	Insert	O(1) per child
	Traversal	O(n)

Trade-offs Between Arrays and Linked Lists

The choice between arrays and linked lists for implementing data structures like stacks and queues involves several trade-offs in terms of memory usage, speed, and ease of implementation.

From a **memory usage** perspective, arrays allocate memory in contiguous blocks. This can lead to memory fragmentation issues, especially in systems handling large data. Conversely, linked lists use dynamic memory allocation, allowing them to grow or shrink flexibly as needed. However, linked lists come with the additional overhead of maintaining pointers, which increases the memory requirements per element.

In terms of **speed**, arrays provide constant-time operations (time complexity: O(1)) for accessing elements by index, making them highly efficient for random access operations. However, insertion and deletion operations in arrays take linear time (O(n)) when performed at arbitrary positions because elements need to be shifted to accommodate changes. Linked lists, on the other hand, support constant-time operations (O(1)) for insertion and deletion at the head of the list. However, accessing elements by position in a linked list requires linear time (O(n)) due to the need to traverse the list sequentially.

When considering **ease of implementation**, arrays are generally simpler to use because their operations, such as accessing or modifying elements, are straightforward. However, arrays are less flexible when it comes to resizing, as they require allocating new memory and copying data when the size limit is reached. Linked lists, although more complex to implement due to the need for node structures and pointers, excel in scenarios where dynamic resizing is essential.

Practical Applications of Data Structures

Each data structure has specific practical applications depending on its properties. Arrays are particularly useful in applications that require fast random access, such as lookup tables or storing large datasets where index-based access is frequent. They are also well-suited for fixed-size collections of data.

Stacks are widely used in scenarios involving recursion or backtracking, such as function call management and expression evaluation in compilers. Their last-in-first-out (LIFO) behavior makes them ideal for these use cases. Queues, with their first-in-first-out (FIFO) behavior, are fundamental in scheduling systems, such as task queues in operating systems or message queues in communication protocols.

Linked lists, owing to their flexibility, are commonly used in dynamic data structures like hash tables or adjacency lists in graph representations. They are also advantageous when the size of the dataset is unknown or varies frequently. Finally, trees, as hierarchical data structures, find applications in areas like file systems, database indexing, and XML/JSON parsing, where data naturally forms a hierarchy.

Github Repository

https://github.com/bparkhe/MSCS_532_Assignment6

References

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