Avijit Saha

Student ID #005023281

Algorithms and Data Structures (MSCS-532-B01)

Assignment 6

November 24, 2024

Part 1: Implementation and Analysis of Selection Algorithms

Implementation: In this part, for the deterministic algorithm (median of median), first divide the array into groups of five elements, find the median of each group, then recursively find the median of these medians and use it as a pivot to partition the array, then recursively select the kth element in the appropriate partition.

```
# Step 1: Divide the array into groups of 5 and find medians
medians = [find_median(arr[i:i + 5]) for i in range(0, len(arr), 5)]

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

Figure 1: Deterministic by median of medians

```
# Step 1: Pick a random pivot
pivot = random.choice(arr)

# Step 2: Partition the array
low = [x for x in arr if x < pivot]
high = [x for x in arr if x > pivot]
pivot_count = len(arr) - len(low) - len(high)  # Count occurrences of the pivot
```

Figure 2: Randomized Quick Select by Selecting a Random Pivot

For the randomized algorithm (randomized quick select), Select a random pivot, partition the array, and recursively find the kth element in the appropriate partition. Randomization helps ensure an average-case time complexity of O(n).

Time Complexity

Deterministic Algorithm:

Worst-case Complexity is O(n). The deterministic Median of Medians algorithm assures a linear time in finding the k-th smallest element, even in the worst case. This is achieved through careful

choice of pivot. The input array is divided into groups of 5. The median of each group is then computed. The median of these medians is chosen as the pivot. This pivot guarantees that in each recursion, at least ≈ 3n/10 elements are discarded (3/10th of the array). The recurrence relation for this algorithm is:

$$\mathbf{T(n)=T(n/5)+T(7n/10)+O(n)}, \text{ which solves to } O(n)$$

Randomized Algorithm:

Expected Complexity is O(n). The randomized Quick Select algorithm randomly selects a pivot and partitions the array around it. On average, the pivot divides the array into roughly equal halves, halving the input size in each recursive step.

This yields a recurrence:

$$\mathbf{T(n)=T(n/2)+O(n)}, \text{ which solves to } O(n) \text{ in expectation.}$$

But the Worst-case Complexity is O(n^2). In the worst case, the chosen pivot could always be the smallest or largest element, with repetition only removing one element at each step. This is the case for very adversarial input, such as input already sorted in reverse, which gives:

$$\mathbf{n+(n-1)+(n-2)+\ldots+1=O(n^2)}$$

Space Complexity

Deterministic Algorithm:

The algorithm runs in place, and partitioning requires O(1) additional space. The recursion depth depends on the size of the array being reduced in each step. The recursion depth is logarithmic in the Median of Medians since 3n/10 elements are removed after every step. This follows a total space complexity of O(log n).

Randomized Algorithm:

As in the deterministic version, the partition step is in place, taking O(1) additional space. The average recursion depth is O(log n) since the array size is halved according to the expectation at each step. In the worst case, recursion depth can increase to O(n) if the pivot is repeatedly chosen poorly.

Empirical Analysis

```
Enter the array elements separated by spaces:
5 3 2 14 5 4 2
Enter the value of k (1-based index):
4

Input Array: [5, 3, 2, 14, 5, 4, 2]

Generated Random Array: [1, 7, 8, 5, 2, 10, 13]

--- Testing with User-Input Array ---
The 4-th smallest element is: 4
Time taken (Deterministic): 0.000011 seconds
The 4-th smallest element is: 4
Time taken (Randomized): 0.000007 seconds

--- Testing with Random Array ---
The 4-th smallest element is: 7
Time taken (Deterministic): 0.000004 seconds
The 4-th smallest element is: 7
Time taken (Randomized): 0.000003 seconds

--- Testing with Sorted Array ---
The 4-th smallest element is: 4
Time taken (Deterministic): 0.000002 seconds
The 4-th smallest element is: 4
Time taken (Randomized): 0.000003 seconds

--- Testing with Reverse-Sorted Array ---
The 4-th smallest element is: 4
Time taken (Deterministic): 0.000001 seconds
The 4-th smallest element is: 4
Time taken (Randomized): 0.000002 seconds
```

Figure 3: Implementation Output

According to the analysis, I have found that randomized quick selection takes less time than deterministic algorithm in some common cases. But if we consider the already sorted array or the reverse sorted array, a randomized algorithm takes more time than a deterministic one. Because of the worst-case scenario of the randomized quick selection, where the chosen pivot could always be the most minor or most significant element, repetition leaves only one element removed at each step. But on the other hand, deterministic always ensures linear time O(n) time complexity for finding the kth smallest element. So, the time is consistent in any array selection. F**igure 3** shows that randomized sampling takes more time than deterministic sampling for a

sorted array and a reversely sorted array. The time difference for the given array is more significant than that of the other version of the variety. The randomized algorithm's performance deteriorates for edge cases where pivots are poorly chosen repeatedly, leading to deeper recursion.

Part 2: Elementary Data Structures Implementation and Discussion

Implementation

Array and Matrix (array_matrix.py), Stack and Queue (stack_queue.py), and Linked List (linked_list.py) are implemented in these files.

Performance Analysis

Time Complexity:

Operation	Array	Stack	Queue	Linked List	Matrix
Insertion (End)	O(1)	O(1)	O(1)	O(1)	O(1)
Insertion (Middle)	O(n)			O(n)	O(1)
Deletion (End)	O(1)	O(1)		O(n)	
Deletion (Middle)	O(n)		O(1)	O(n)	O(1)
Access (Specific Pos)	O(1)	O(1)	O(1)	O(n)	O(1)

Tradeoffs Between Arrays and Linked Lists:

- Memory Use: Arrays require contiguous memory; linked lists do not.
- Access Time: Arrays are O(1) access; linked lists are O(n) traversal.
- Insertion/Deletion: Among arbitrary position insertions/deletions, linked lists perform better, O(1), than arrays, O(n).

Discussion Practical Applications

- **Arrays:** Utilized where random access is needed, such as indexing vast data sets.
- Stacks: Applied in parsing expressions, backtracking, and recursion simulation.

- **Queues:** Useful in task scheduling, resource management (such as printers), and BFS traversal.
- **Linked Lists:** Preferable in situations involving dynamic memory functions or when many insertions/deletions must be performed frequently.
- **Rooted Trees:** Used in file systems, decision algorithms, and data hierarchy.

Real-World Comparison

- **Arrays** for fast access with fixed-size data.
- **Linked lists** are used when the size of data changes dynamically and insertions/deletions are more frequent.
- **Queues** for task scheduling in real-time.
- **Stacks** for Last-In-First-Out operations, such as providing undo functionality in editors.

References:

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

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2006). Algorithms. McGraw-Hill.

Goodrich, M. T., & Tamassia, R. (2014). Algorithm Design and Applications. Wiley.

Knuth, D. E. (1998). The Art of Computer Programming: Volume 3 - Sorting and Searching (2nd ed.). Addison-Wesley.

Mitzenmacher, M., & Upfal, E. (2017). Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.