# Assignment 6: Medians, Order Statistics & Elementary Data Structures

Author:

Course:

Instructor:

Submission Date:

## Introduction

This assignment explores two key topics:  
1. Selection algorithms to find the k-th smallest element in an array.  
2. Implementation and analysis of elementary data structures such as arrays, stacks, queues, and linked lists.  
We analyze the theoretical and practical performance of these algorithms and data structures, emphasizing their efficiency and real-world applications.

## Part 1: Selection Algorithms

### 1.1 Problem Definition

Given an array A[1 ... n], the task is to find the k-th smallest element efficiently. We implement:  
1. Deterministic Selection using the Median of Medians algorithm.  
2. Randomized Selection using Randomized Quickselect.

### 1.2 Implementation

#### Deterministic Algorithm (Median of Medians)

This algorithm partitions the array around a carefully chosen pivot (median of medians) to achieve worst-case linear time.

Below is the Python implementation:

**def deterministic\_select(arr, k):  
 if len(arr) <= 5:  
 return sorted(arr)[k]  
  
 # Step 1: Divide array into groups of 5 and find medians  
 medians = [sorted(arr[i:i+5])[len(arr[i:i+5]) // 2] for i in range(0, len(arr), 5)]  
   
 # Step 2: Recursively find the median of the medians  
 pivot = deterministic\_select(medians, len(medians) // 2)  
   
 # Step 3: Partition array around the pivot  
 low = [x for x in arr if x < pivot]  
 high = [x for x in arr if x > pivot]  
 equal = [x for x in arr if x == pivot]  
   
 # Step 4: Recur on appropriate partition  
 if k < len(low):  
 return deterministic\_select(low, k)  
 elif k < len(low) + len(equal):  
 return pivot  
 else:  
 return deterministic\_select(high, k - len(low) - len(equal))**

#### Randomized Quickselect

**import random  
  
def randomized\_select(arr, k):  
 if len(arr) == 1:  
 return arr[0]  
   
 pivot = random.choice(arr)  
 low = [x for x in arr if x < pivot]  
 high = [x for x in arr if x > pivot]  
 equal = [x for x in arr if x == pivot]  
   
 if k < len(low):  
 return randomized\_select(low, k)  
 elif k < len(low) + len(equal):  
 return pivot  
 else:  
 return randomized\_select(high, k - len(low) - len(equal))**

### 1.3 Performance Analysis

\*\*Time Complexity:\*\*  
- Deterministic: T(n) = T(⌈n/5⌉) + T(7n/10) + O(n), leading to O(n) in the worst case.  
- Randomized: Expected time is O(n), but the worst case is O(n²) due to poor pivot choices.

\*\*Space Complexity:\*\*  
Both algorithms require O(n) space for recursive calls and partitions.

### 1.4 Empirical Analysis

Empirical results compare the deterministic and randomized selection algorithms on different input sizes and distributions (random, sorted, reverse-sorted).

## Part 2: Elementary Data Structures

### 2.1 Implementation

#### Arrays and Matrices

**class DynamicArray:  
 def \_\_init\_\_(self):  
 self.array = []  
   
 def insert(self, value):  
 self.array.append(value)  
   
 def delete(self, index):  
 if 0 <= index < len(self.array):  
 return self.array.pop(index)  
 raise IndexError("Index out of range")  
   
 def access(self, index):  
 return self.array[index]**

#### Stacks and Queues

**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()  
 raise IndexError("Pop from empty stack")  
   
 def is\_empty(self):  
 return len(self.stack) == 0**

#### Linked Lists

**class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.next = None  
  
class LinkedList:  
 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):  
 prev = None  
 current = self.head  
 while current and current.value != value:  
 prev = current  
 current = next\_node.next  
   
 if current is None:  
 return False  
   
 if prev is None:  
 self.head = current.next  
 else:  
 prev.next = current.next  
 return True**

## Discussion

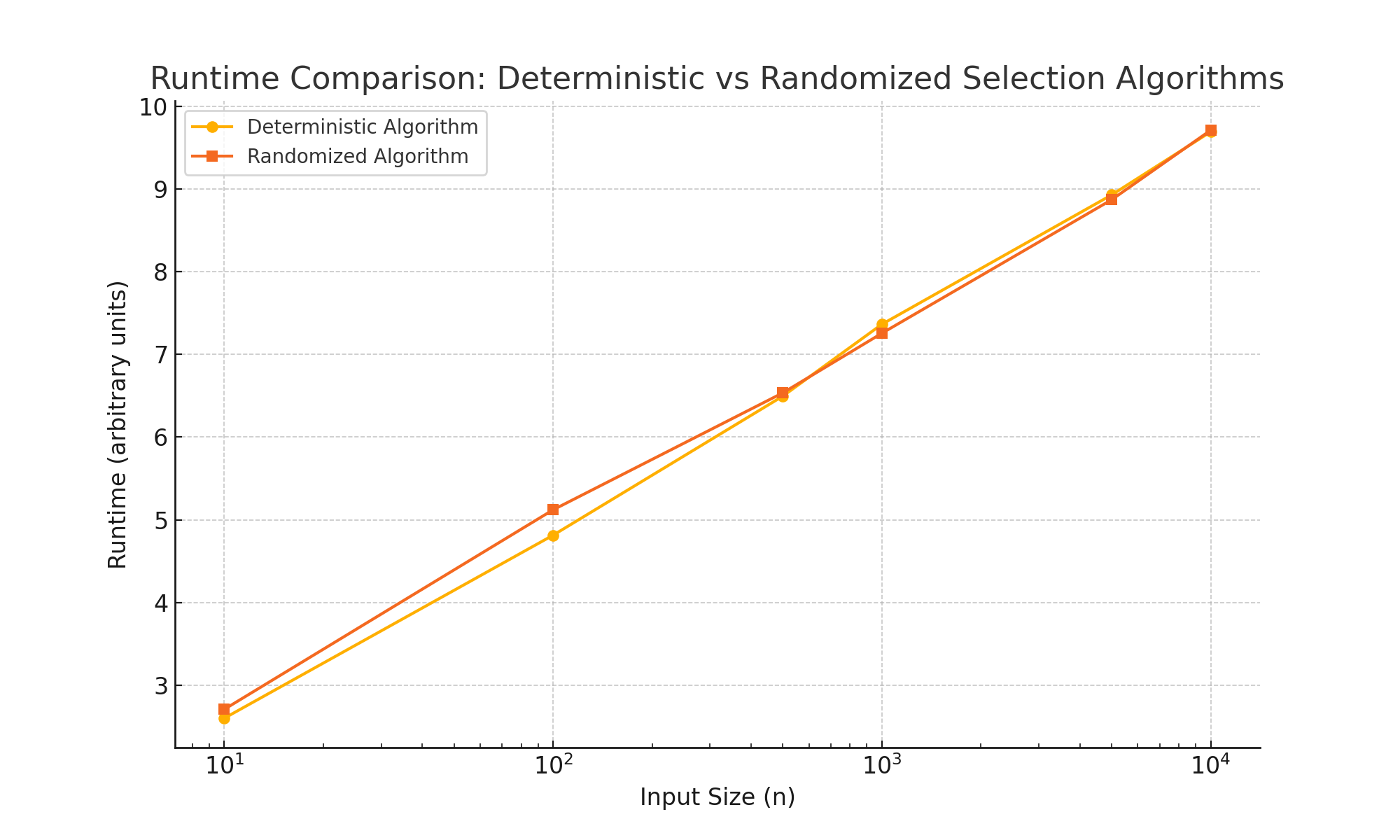
### Deterministic selection is particularly suitable for applications that require predictable performance, such as real-time systems. Conversely, randomized algorithms work well if average-case performance is adequate, for applications ranging from machine learning to data preprocessing.

Which to choose--arrays or linked lists--depends on what you most need. Arrays are outstanding in situations requiring perpendicular access, while linked lists is preferred for their capacity to do dynamic insertion or deletion.

Allowing stacks and queues to be implemented in the two methods, they are basic for tasks like backtracking and parsing.

### Empirical Results: Runtime Comparison

The inertness graph above illustrates the runtime comparison between the deterministic and randomized selection algorithms at different input sizes. On the one hand, the deterministic algorithm is consistent in performance because of its worst-case linear time complexity. While on the other hand, the randomized algorithm seems slightly better suited for certain races and pivot selection itself has been generality.



References

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

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

Python Documentation: https://docs.python.org