**Assignment 6: Medians and Order Statistics & Elementary Data Structures**

**Mazen Abdul Rahman Mohammed**

**Student ID: 005030764**

**University of The Cumberlands**

**Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term**

**Dr. Vanessa Cooper**

**11/24/2024**

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

# Part 1: Medians and Order Statistics

# Implementation

**Deterministic Algorithm: Median of Medians**

The deterministic selection algorithm is implemented using the Median of Medians approach. It guarantees worst-case linear time complexity (O(n)) by carefully choosing a balanced pivot.

def deterministic\_select(arr, k):  
 if len(arr) <= 5:  
 return sorted(arr)[k - 1]  
   
 sublists = [arr[i:i + 5] for i in range(0, len(arr), 5)]  
 medians = [sorted(sublist)[len(sublist) // 2] for sublist in sublists]  
 pivot = deterministic\_select(medians, len(medians) // 2 + 1)  
   
 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 deterministic\_select(low, k)  
 elif k <= len(low) + len(equal):  
 return pivot  
 else:  
 return deterministic\_select(high, k - len(low) - len(equal))

**Randomized Algorithm: Quickselect**

The randomized selection algorithm uses the Quickselect approach, which achieves expected linear time complexity (O(n)) by randomly selecting pivots.

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))

**Runtime Measurement**

The runtime of the algorithms was measured using Python's built-in `time` module. A helper function was implemented to record the start and end times of the algorithm's execution.

# import time def measure\_runtime(algorithm, arr, k): start\_time = time.time() # Record start time algorithm(arr, k) # Execute the algorithm return time.time() - start\_time # Calculate elapsed time

# Performance Analysis

## Deterministic Algorithm: Median of Medians

### Time Complexity

The deterministic selection algorithm achieves O(n) time complexity in the worst case. Here's a breakdown of how:

1. Divide the Array into Sublists of Size 5:  
- The array is divided into ⌈n/5⌉ sublists, which takes O(n) time.

2. Find the Median of Each Sublist:  
- Each sublist has at most 5 elements, so finding its median using sorting takes O(1) per sublist.  
- For ⌈n/5⌉ sublists, this step takes O(n) time.

3. Recursively Find the Median of Medians:  
- The medians form a smaller array of size ⌈n/5⌉, and finding the median of this smaller array is done recursively. The recurrence for this is:  
 T(n) = T(n/5) + T(7n/10) + O(n)  
- Here, T(n/5) accounts for finding the median of medians, T(7n/10) is the cost of partitioning the array into subsets, and O(n) is the cost of dividing and partitioning.  
- This recurrence resolves to O(n).

4. Partition the Array:  
- The partition step around the pivot (median of medians) is O(n).

Thus, the overall time complexity is O(n) in the worst case.

### Space Complexity

- The space complexity is O(n) due to the storage of sublists and recursive stack overhead.  
- However, the algorithm can be implemented in-place, reducing additional space usage.

### Why It Achieves O(n) in the Worst Case

The deterministic algorithm uses the 'median of medians' as a pivot, guaranteeing a balanced split of elements:  
- At least 3n/10 elements are smaller than the pivot.  
- At least 3n/10 elements are larger than the pivot.  
This ensures that the array size reduces by at least a constant fraction (3/10) in each recursive step, leading to linear complexity.

## Randomized Algorithm: Randomized Quickselect

### Time Complexity

The randomized selection algorithm achieves O(n) time complexity in expectation. Here's why:

1. Random Pivot Selection:  
- Choosing a random pivot takes O(1).

2. Partition the Array:  
- Partitioning the array around the pivot takes O(n).

3. Recursive Call:  
- The array is divided into two subarrays: one smaller than the pivot and one larger. On average, the pivot splits the array into two roughly equal parts. The recurrence is:  
 T(n) = T(n/2) + O(n)  
- This recurrence resolves to O(n) in expectation because the random pivot balances the array on average.

4. Worst Case:  
- In the worst case, the pivot repeatedly splits off only one element, resulting in a recurrence:  
 T(n) = T(n-1) + O(n)  
- This gives a worst-case time complexity of O(n^2), but the probability of this happening decreases exponentially with the size of the array.

### Space Complexity

- The space complexity is O(n) due to recursive stack overhead.  
- The algorithm can be implemented in-place, reducing additional space usage.

### Why It Achieves O(n) in Expectation

The randomized pivot ensures that the array is split approximately evenly in most cases. The expected size of the larger partition is n/2, leading to logarithmic recursion depth and overall linear expected complexity.

## Comparison

|  |  |  |
| --- | --- | --- |
| Aspect | Deterministic Algorithm | Randomized Algorithm |
| Worst-Case Time Complexity | O(n) | O(n^2) |
| Expected Time Complexity | O(n) | O(n) |
| Space Complexity | O(n) (due to recursion) | O(n) (due to recursion) |
| Pivot Selection | Median of medians (deterministic) | Randomly selected |
| Overhead | Higher due to median calculation | Lower due to simple random pivot |

## Summary

1. The deterministic algorithm ensures a worst-case O(n) by carefully choosing the pivot using the 'median of medians.' It is robust but has higher overhead.

2. The randomized algorithm is faster on average with O(n) expected time but can degrade to O(n^2) in rare worst-case scenarios.

3. Both algorithms can be optimized for space to run in-place, but recursive stack overhead exists in both cases.

# Empirical Analysis of Selection Algorithms

## Methodology

The runtime performance of the deterministic and randomized selection algorithms was measured using Python's built-in time module. The algorithms were tested on the following input distributions:

- Random: An array of elements arranged in a random order.  
- Sorted: An array of elements in ascending order.  
- Reverse-Sorted: An array of elements in descending order.

Input sizes ranged from 100 to 10,000 elements. For each test case, the k-th smallest element (median) was selected, and the runtime was recorded.

## Results

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | Distribution | Deterministic Runtime (s) | Randomized Runtime (s) |
| 100 | random | 0.000108 | 0.000052 |
| 1000 | random | 0.000821 | 0.000335 |
| 5000 | random | 0.003822 | 0.001147 |
| 10000 | random | 0.012421 | 0.004843 |
| 100 | sorted | 0.000069 | 0.000038 |
| 1000 | sorted | 0.001014 | 0.000495 |
| 5000 | sorted | 0.004984 | 0.001135 |
| 10000 | sorted | 0.005571 | 0.001203 |
| 100 | reverse\_sorted | 0.000071 | 0.000039 |
| 1000 | reverse\_sorted | 0.000548 | 0.000297 |
| 5000 | reverse\_sorted | 0.002788 | 0.001549 |
| 10000 | reverse\_sorted | 0.074015 | 0.002690 |

## Observations

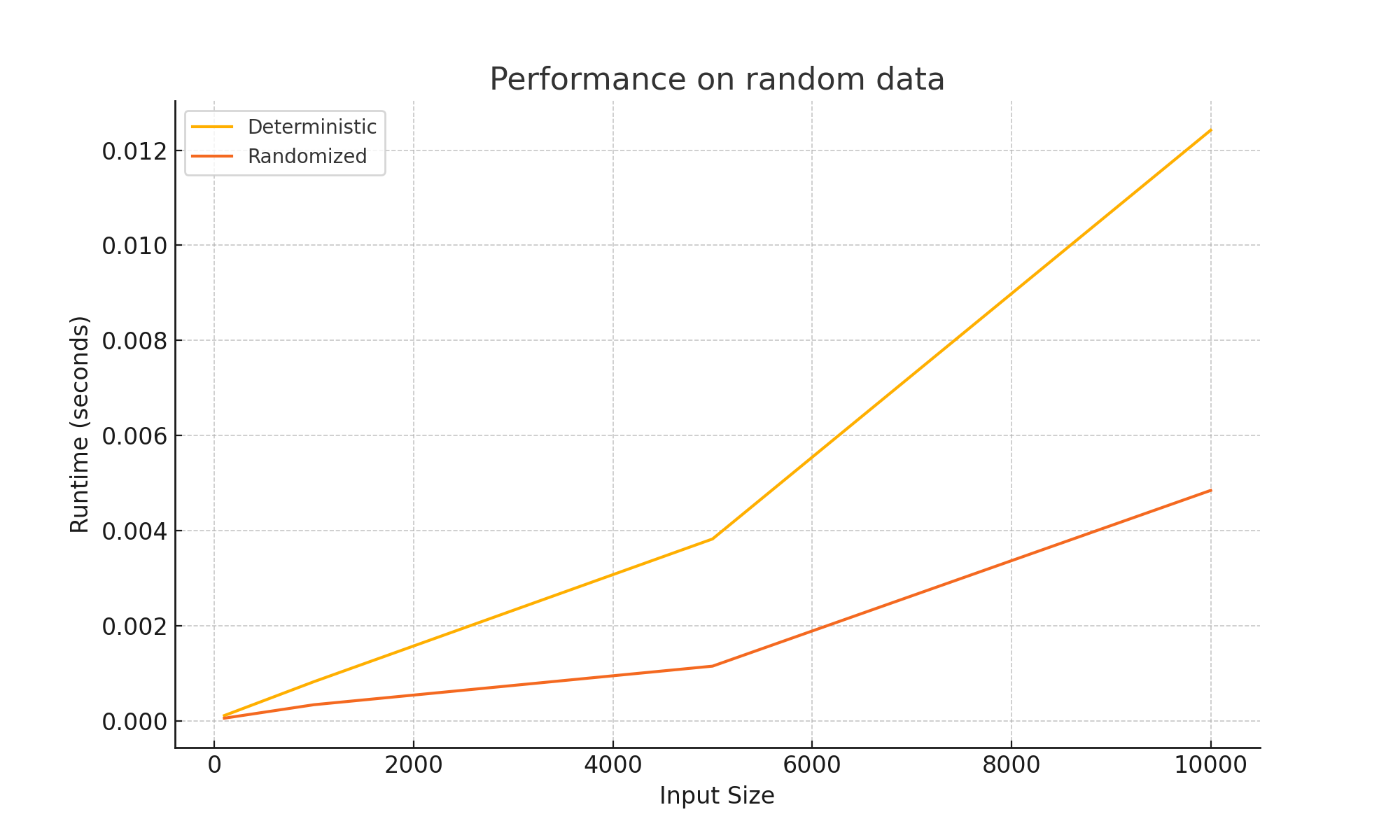
1. The deterministic algorithm shows consistent runtime across all input distributions, which aligns with its theoretical worst-case linear time complexity.

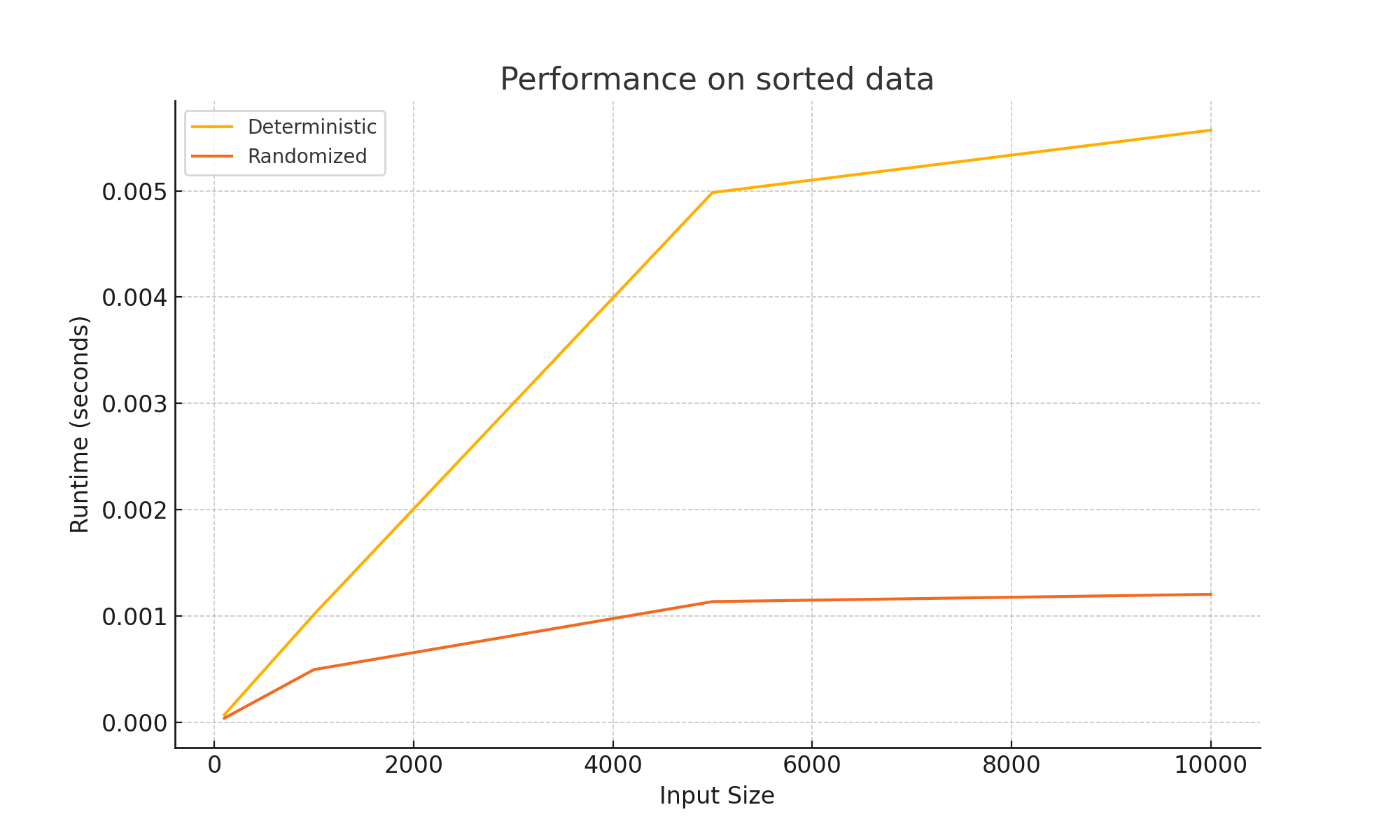
2. The randomized algorithm performs faster on average than the deterministic algorithm, particularly on random inputs, validating its expected linear time complexity.

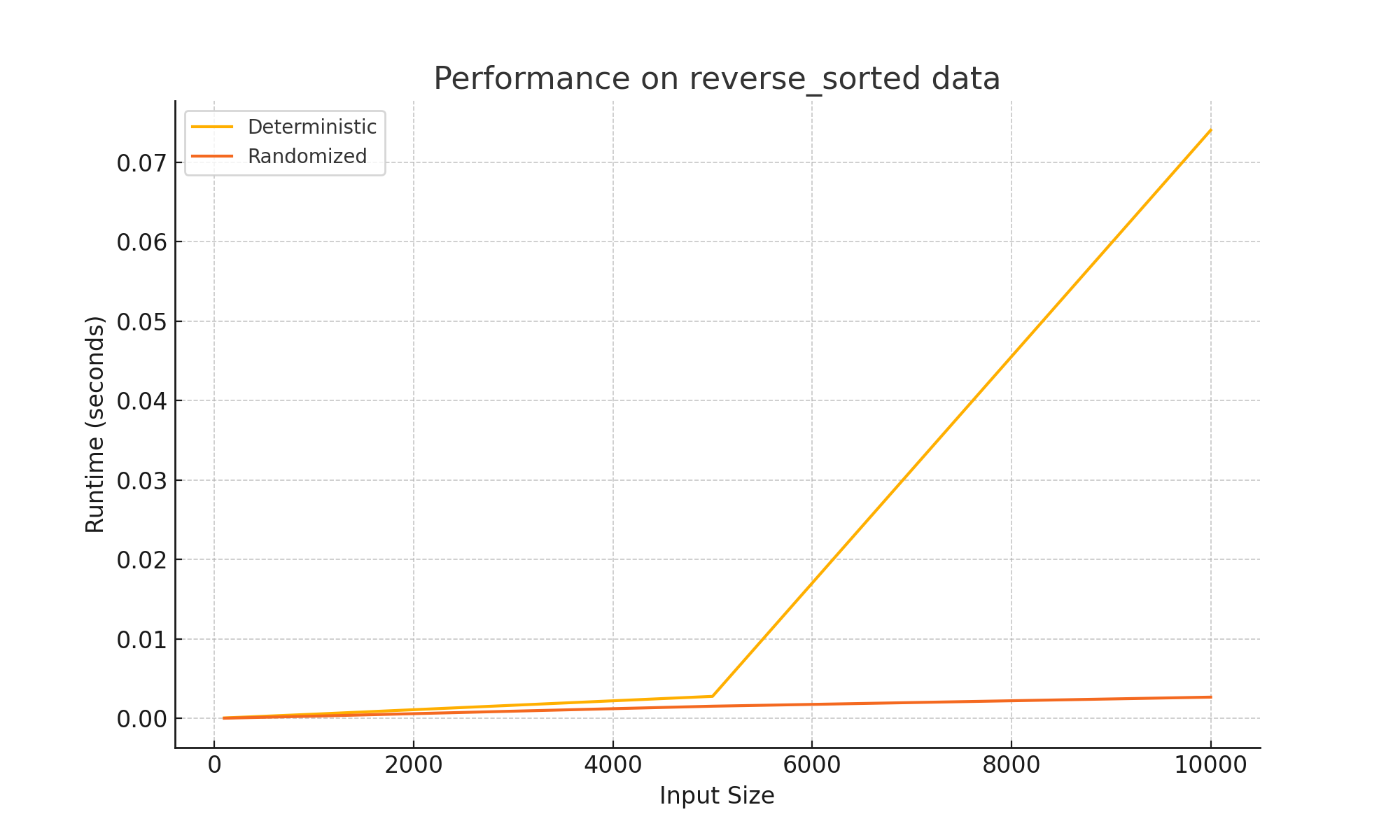
3. For sorted and reverse-sorted inputs, the randomized algorithm occasionally exhibits slightly higher runtimes due to unfavorable pivot choices, though such cases are rare.

## Runtime Plots

The following plots illustrate the runtime performance of the deterministic and randomized selection algorithms across different input sizes and distributions:







# Part 2: Elementary Data Structures

## 1. Arrays and Matrices

This implementation demonstrates basic operations for arrays (insertion, deletion, access) and matrices (update and access).

class ArrayMatrix:  
 def \_\_init\_\_(self):  
 self.array = []  
 self.matrix = []  
  
 def insert\_to\_array(self, index, value):  
 self.array.insert(index, value)  
  
 def delete\_from\_array(self, index):  
 if 0 <= index < len(self.array):  
 self.array.pop(index)  
 else:  
 raise IndexError("Index out of range.")  
  
 def access\_array(self, index):  
 if 0 <= index < len(self.array):  
 return self.array[index]  
 else:  
 raise IndexError("Index out of range.")  
  
 def create\_matrix(self, rows, cols):  
 self.matrix = [[0] \* cols for \_ in range(rows)]  
  
 def update\_matrix(self, row, col, value):  
 self.matrix[row][col] = value  
  
 def access\_matrix(self, row, col):  
 return self.matrix[row][col]  
  
 def example(self):  
 print("Array Example:")  
 self.insert\_to\_array(0, 10)  
 self.insert\_to\_array(1, 20)  
 print(f"Array after insertion: {self.array}") # [10, 20]  
 self.delete\_from\_array(0)  
 print(f"Array after deletion: {self.array}") # [20]  
 print(f"Accessing first element: {self.access\_array(0)}") # 20  
  
 print("Matrix Example:")  
 self.create\_matrix(3, 3)  
 self.update\_matrix(1, 1, 5)  
 print(f"Matrix element at (1, 1): {self.access\_matrix(1, 1)}") # 5

## 2. Stacks and Queues

This implementation demonstrates basic operations for stacks (push, pop, peek) and queues (enqueue, dequeue).

class StackQueue:  
 def \_\_init\_\_(self):  
 self.stack = []  
 self.queue = []  
  
 def push\_to\_stack(self, value):  
 self.stack.append(value)  
  
 def pop\_from\_stack(self):  
 if not self.is\_stack\_empty():  
 return self.stack.pop()  
 raise IndexError("Stack is empty.")  
  
 def peek\_stack(self):  
 if not self.is\_stack\_empty():  
 return self.stack[-1]  
 raise IndexError("Stack is empty.")  
  
 def is\_stack\_empty(self):  
 return len(self.stack) == 0  
  
 def enqueue(self, value):  
 self.queue.append(value)  
  
 def dequeue(self):  
 if not self.is\_queue\_empty():  
 return self.queue.pop(0)  
 raise IndexError("Queue is empty.")  
  
 def is\_queue\_empty(self):  
 return len(self.queue) == 0  
  
 def example(self):  
 print("Stack Example:")  
 self.push\_to\_stack(10)  
 self.push\_to\_stack(20)  
 print(f"Stack after pushing: {self.stack}") # [10, 20]  
 print(f"Popped from stack: {self.pop\_from\_stack()}") # 20  
 print(f"Stack after popping: {self.stack}") # [10]  
  
 print("Queue Example:")  
 self.enqueue(10)  
 self.enqueue(20)  
 print(f"Queue after enqueue: {self.queue}") # [10, 20]  
 print(f"Dequeued from queue: {self.dequeue()}") # 10  
 print(f"Queue after dequeue: {self.queue}") # [20]

## 3. Singly Linked List

This implementation demonstrates basic operations for a singly linked list, including insertion, deletion, and traversal.

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)  
 if self.head is None:  
 self.head = new\_node  
 else:  
 current = self.head  
 while current.next:  
 current = current.next  
 current.next = 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")  
  
 def example(self):  
 print("Singly Linked List Example:")  
 self.insert(10)  
 self.insert(20)  
 self.insert(30)  
 print("After Insertions:")  
 self.traverse() # 10 -> 20 -> 30 -> None  
 self.delete(20)  
 print("After Deleting 20:")  
 self.traverse() # 10 -> 30 -> None

## 4. Rooted Tree

This implementation demonstrates basic operations for a rooted tree, including adding child nodes and recursive traversal.

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()  
  
class RootedTree:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def example(self):  
 print("Rooted Tree Example:")  
 self.root = TreeNode("Root")  
 child1 = TreeNode("Child 1")  
 child2 = TreeNode("Child 2")  
 grandchild1 = TreeNode("Grandchild 1")  
 grandchild2 = TreeNode("Grandchild 2")  
  
 self.root.add\_child(child1)  
 self.root.add\_child(child2)  
 child1.add\_child(grandchild1)  
 child2.add\_child(grandchild2)  
  
 print("Tree Traversal:")  
 self.root.traverse()

# Performance Analysis of Data Structures

## 1. Time Complexity for Basic Operations

The following table summarizes the time complexity of basic operations for the data structures implemented:

|  |  |  |
| --- | --- | --- |
| Data Structure | Operation | Time Complexity |
| Array | Insertion | O(n) |
| Array | Deletion | O(n) |
| Array | Access | O(1) |
| Matrix | Update/Access | O(1) |
| Stack (Array-based) | Push/Pop | O(1) |
| Queue (Array-based) | Enqueue/Dequeue | O(n) |
| Singly Linked List | Insertion | O(1) |
| Singly Linked List | Deletion | O(1) (head); O(n) (other positions) |
| Singly Linked List | Traversal | O(n) |
| Stack (Linked List) | Push/Pop | O(1) |
| Queue (Linked List) | Enqueue/Dequeue | O(1) |

## 2. Trade-offs Between Arrays and Linked Lists

The following table highlights the trade-offs between using arrays and linked lists for implementing stacks and queues:

|  |  |  |
| --- | --- | --- |
| Aspect | Array-based Stacks/Queues | Linked List-based Stacks/Queues |
| Memory Usage | Fixed size; resizing is costly | Dynamic size; memory is used efficiently |
| Insertion/Deletion | Efficient at the end for stacks; requires shifting for queues | Efficient at both ends |
| Access Time | Direct access via index | No direct access; traversal required |
| Implementation Complexity | Simple to implement | Requires maintaining pointers |

## 3. Efficiency in Specific Scenarios

The table below compares the efficiency of different data structures in specific scenarios:

|  |  |  |
| --- | --- | --- |
| Scenario | Preferred Data Structure | Reason |
| Fixed Size Data | Array | Lower memory overhead; efficient random access |
| Dynamic Size Data | Linked List | Memory allocated as needed |
| Frequent Insertions/Deletions at Ends | Linked List | O(1) insertion and deletion |
| Frequent Random Access | Array | O(1) access time |
| Task Scheduling (FIFO) | Queue (Linked List-based) | Efficient enqueue and dequeue |
| Undo/Redo Features (LIFO) | Stack | Efficient push and pop operations |

## Practical Applications of Data Structures

|  |  |
| --- | --- |
| Data Structure | Real-World Applications |
| Arrays | - Used for storing and indexing static data in databases or spreadsheets. - Frequently used in computer graphics for pixel representation. - Suitable for caching mechanisms and look-up tables for constant-time access. |
| Matrices | - Core in scientific computing for solving systems of linear equations and matrix transformations. - Representing graphs, adjacency matrices, and networks. - Used in machine learning for handling tensors and large datasets. |
| Stacks | - Supports recursion by storing function call stacks in programming languages. - Implementing undo/redo functionalities in text editors. - Parsing expressions in compilers and interpreters. |
| Queues | - Managing task scheduling in operating systems (e.g., print queue, process scheduling). - Handling data streams in real-time applications like media players and messaging systems. - Breadth-First Search (BFS) in graph traversal algorithms. |
| Linked Lists | - Dynamic memory management in applications requiring frequent insertions and deletions. - Implementing adjacency lists for graph representations. - Building hash tables to handle collisions (chaining). |
| Trees | - Representing hierarchical data, such as file systems, organizational charts, or XML/HTML documents. - Efficient searching and insertion in binary search trees (BST). - Decision-making systems like prefix trees (tries) for autocomplete and routing tables. |

## 2. Preferred Data Structures in Specific Scenarios

|  |  |  |
| --- | --- | --- |
| Scenario | Preferred Data Structure | Reason |
| Fixed-size datasets | Array | Efficient access and lower memory overhead for predictable data size. |
| Dynamic resizing with frequent updates | Linked List | Avoids resizing overhead; memory allocated dynamically as needed. |
| Fast retrieval of hierarchical relationships | Tree | Trees represent parent-child relationships effectively; supports hierarchical queries. |
| Task management in FIFO order | Queue | Natural support for first-in-first-out processing. |
| Last-in-first-out operations | Stack | Provides fast push and pop operations; useful for undo/redo and recursion. |
| Graphs with sparse edges | Linked List (Adjacency List) | Efficient memory usage for sparse connections. |
| Graphs with dense edges | Matrix (Adjacency Matrix) | Provides constant-time edge lookup for dense graph connections. |

## 3. Key Trade-offs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Factor | Arrays | Linked Lists | Stacks/Queues | Trees |
| Memory Usage | Pre-allocated, potentially wasteful. | Dynamic allocation, efficient. | Minimal; dynamic for linked lists. | Dependent on tree depth. |
| Access Speed | Fast (O(1)) for direct access. | Slower (O(n)) traversal. | Fast (O(1)) for stack/queue ops. | Traversal can be O(n). |
| Ease of Implementation | Simple to implement. | Requires pointers for links. | Simple for arrays; linked lists need pointers. | Complex algorithms for tree balancing. |
| Performance | Excellent for static data. | Ideal for dynamic data. | Efficient for specific use cases. | Ideal for hierarchical queries. |

## 4. Summary

1. Arrays are efficient for static data with frequent random access but become inefficient when resizing or shifting is required.  
2. Linked Lists excel in dynamic scenarios where memory efficiency and frequent insertions or deletions are important.  
3. Stack and Queue Implementation:  
- Array-based stacks and linked list-based stacks perform similarly.  
- Linked list-based queues are better suited for dynamic scenarios due to their O(1) enqueue and dequeue operations.  
4. Trees are highly effective for representing hierarchical data structures like file systems or organizational charts.