**Medians and Order Statistics & Elementary Data Structures**

Shyam Nath

University of Cumberland

Algorithms and Data Structures (MSCS-532-B01)

Student ID: 005032491

Assignment 6

August 03, 2025

**Medians and Order Statistics & Elementary Data Structures**

**Part 1: Performance Analysis and Empirical Evaluation of Selection Algorithms**

**1. Overview**

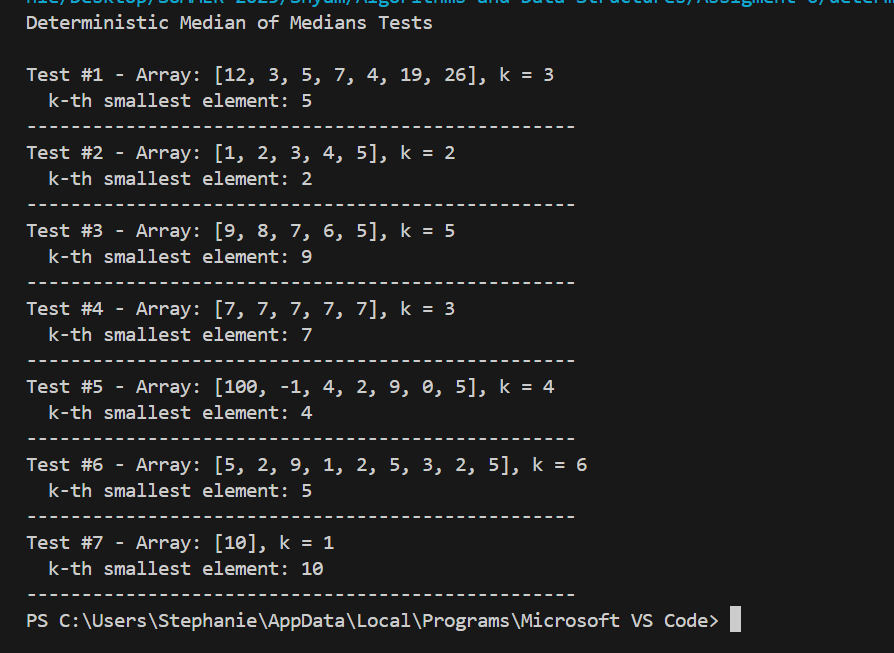
This report analyzes two fundamental selection algorithms used to find the k-th smallest element (order statistic) in an unsorted array:  
- Deterministic Selection using the Median of Medians algorithm.  
- Randomized Selection using the Randomized Quickselect algorithm.

The algorithms are implemented in Python and tested on arrays of varying size and order to compare theoretical expectations with empirical behavior.

**2. Algorithm Overview**

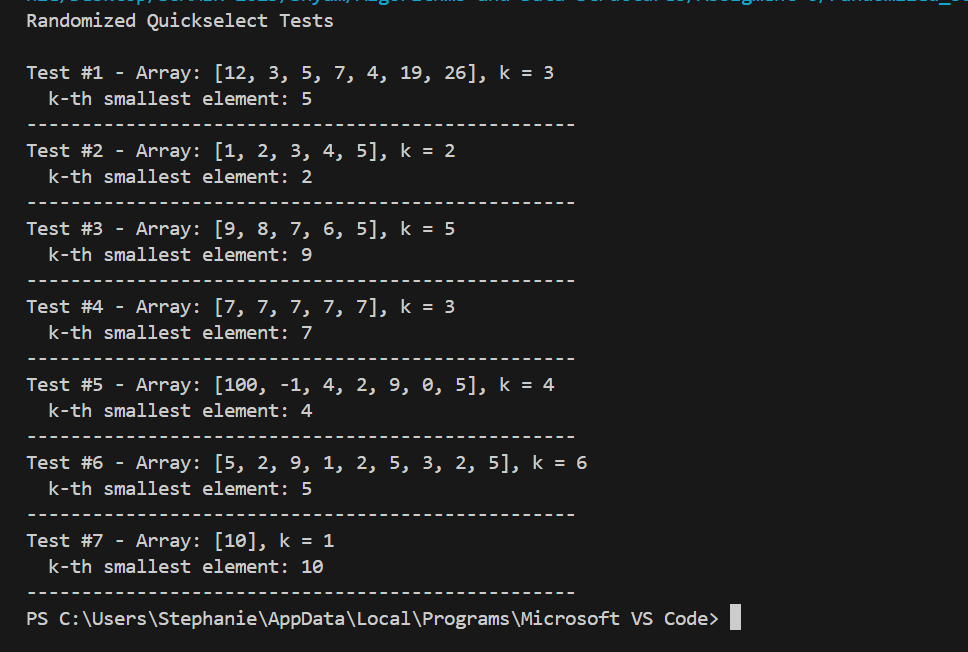
**2.1 Deterministic Selection (Median of Medians)**

Guarantees worst-case linear time O(n).  
Operates by:  
- Dividing the array into groups of 5.  
- Selecting the median of each group.  
- Recursively selecting the median of medians as a pivot.  
- Partitioning the array and recursing on the relevant partition.



**2.2 Randomized Selection (Quickselect)**

Achieves expected linear time O(n).  
Operates by:  
- Randomly choosing a pivot.  
- Partitioning around the pivot.  
- Recursing based on the rank of the pivot.



**3. Time Complexity Analysis**

**1. Randomized Quickselect**

**Algorithm Overview:**

Randomized Quickselect is a variation of the QuickSort algorithm. It selects a pivot at random and partitions the array into elements smaller and greater than the pivot, then recursively continues the search in the relevant partition (Amin et al., 2023).

**Best/Average Case Time Complexity:**

* **Expected Time Complexity:** **O(n)**
* The expectation comes from the fact that, on average, the pivot splits the array reasonably well (e.g., 25%–75%).
* The recurrence relation is approximately:

T(n)=T(n/2) +O(n)

which solves to **O(n)**.

**Worst Case Time Complexity:**

* **O(n²)** if the pivot ends up being the smallest or largest element every time, leading to:

T(n)=T(n−1) +O(n)⇒O(n2)

**Space Complexity:**

* **O(1)** auxiliary space (in-place).
* **O(log n)** call stack space in average case due to recursion.
* **O(n)** in worst case due to unbalanced recursion.

**Overheads:**

* Pivot selection is simple (random).
* No explicit array splitting - partitioning is done in-place.

**2. Deterministic Median of Medians**

**Algorithm Overview:**

This algorithm uses the “median of medians” as a pivot to guarantee a good split of the array regardless of input. It is designed to ensure worst-case linear performance (Mostert et al., 2021).

**Worst Case Time Complexity:**

* **O(n)**
* The recurrence relation is:

T(n)≤ T(n/5)+T(7n/10)+O(n)

* **T(n/5):** for finding medians of 5-element groups.
* **T(7n/10):** for the largest possible recursive partition.
* The linear term comes from partitioning and grouping.

This recurrence solves to **T(n) = O(n)**.

**Best/Average Case Time Complexity:**

* Still **O(n)**, but with a larger constant factor than randomized quickselect due to extra operations:
  + Grouping elements
  + Sorting small groups
  + Recursively finding median of medians

**Space Complexity:**

* **O(n)** for partitioning into sublists and recursive calls.
* Not in-place (due to Python list slicing and temporary arrays).

**Overheads:**

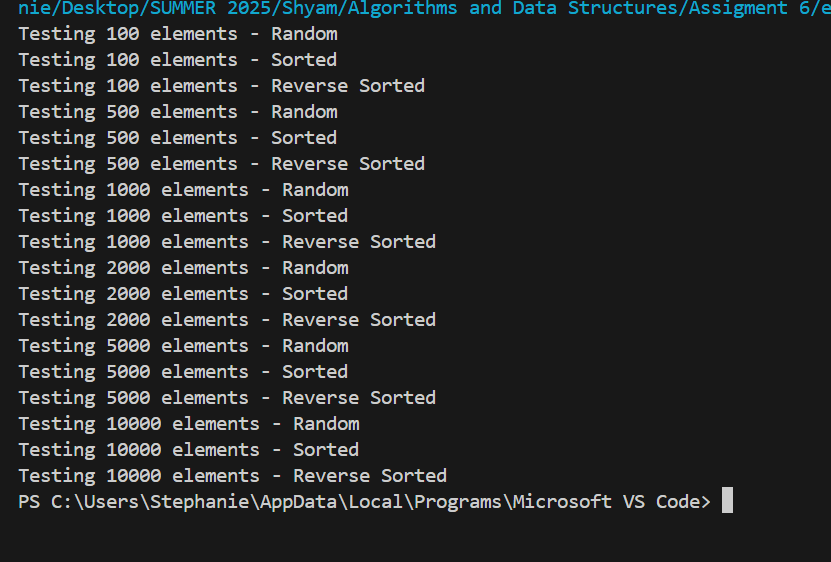
* Grouping into chunks of 5 and computing medians is costly.
* More complex logic compared to randomized version

**5. Empirical Results**

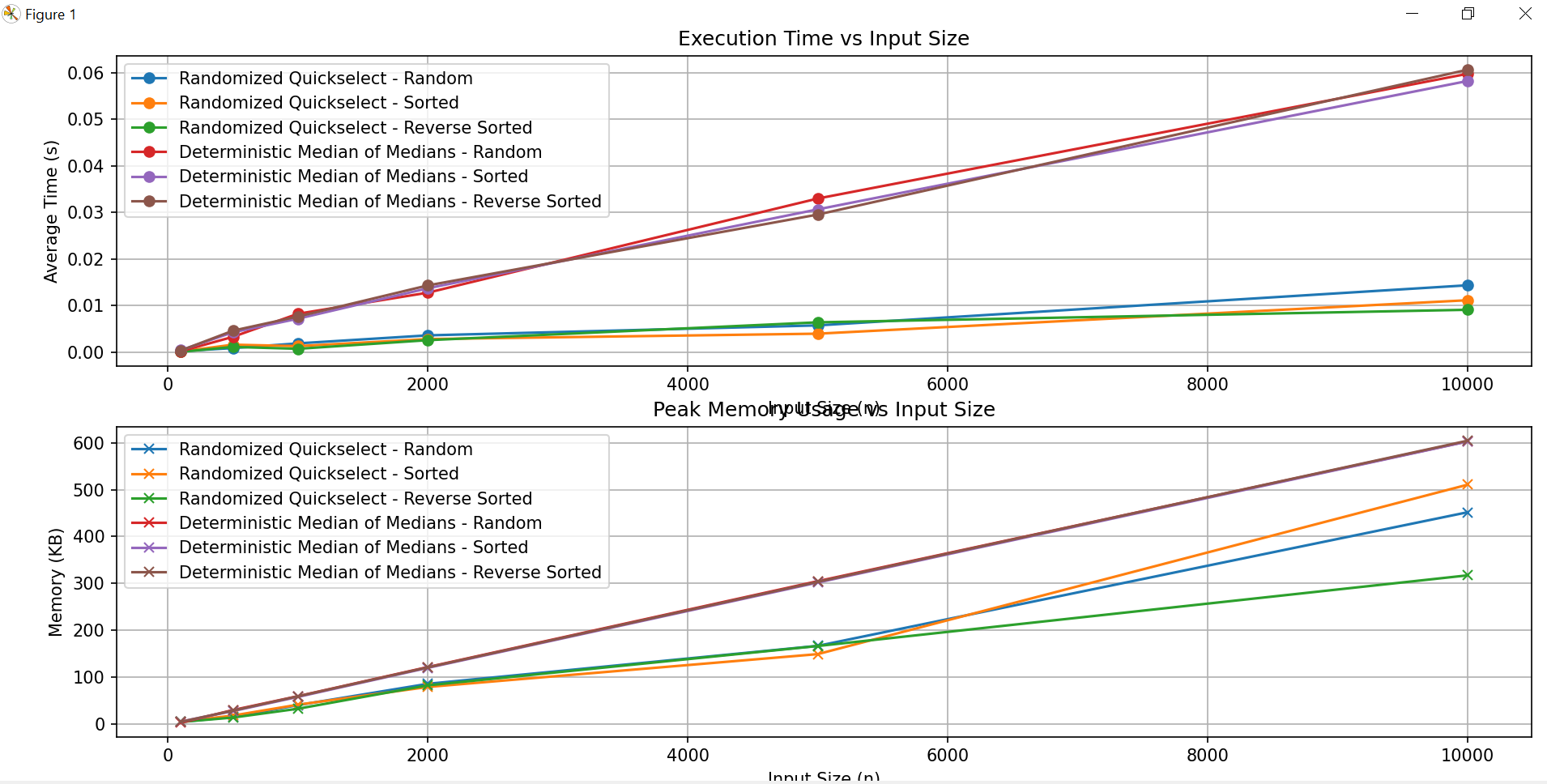
**5.1 Test Parameters**

- Input sizes: 100, 500, 1000, 2000, 5000, 10000  
- Distributions: Random, Sorted, Reverse-Sorted  
- Trials: 5 per configuration  
- Metrics:  
 - Average execution time  
 - Peak memory usage (using tracemalloc)

**5. 2 Results of Implementation**



**5.3 Graphical Results**

  
These visuals clearly illustrate the trade-offs in time and memory between the two algorithms.

**5.4 Observations**

- Randomized Quickselect is faster in practice for average-case scenarios but less stable for adversarial cases.  
- Deterministic Median of Medians offers consistent performance regardless of input distribution, but with higher overhead.  
- Memory usage is generally higher for deterministic selection due to recursion and grouping.

**Part 2: Elementary Data Structures: Implementation and Analysis**

**Objective**

This report presents the implementation, performance analysis, and real-world application of elementary data structures — Arrays, Matrices, Stacks, Queues, and Singly Linked Lists. These structures were implemented from scratch using Python, and their strengths, weaknesses, and suitable use cases are discussed in depth.

**Implementation**

All data structures were implemented in a modular, object-oriented fashion using Python:

**1. Arrays and Matrices**

* **Array:** Implemented as a fixed-size list. Supports insert, delete, and access operations.
* **Matrix:** Implemented as a list of lists. Supports insert and access operations using 2D indexing.

**2. Stack (using Array)**

* Implements LIFO behavior.
* Uses a list internally with append (for push) and pop (for pop).
* Also includes peek and is\_empty methods.

**3. Queue (using Array)**

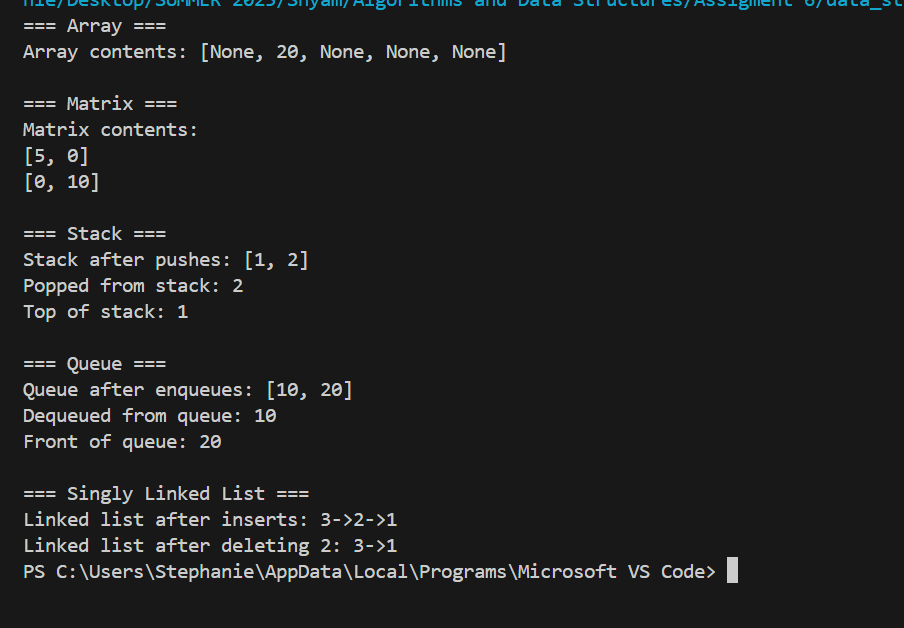
* Implements FIFO behavior.
* Uses a list with append (enqueue) and pop(0) (dequeue).
* Includes front and is\_empty methods.

**4. Singly Linked List**

* Composed of Node objects connected via next pointers.
* Supports insert (at head), delete (by value), and traverse.

All implementations are thoroughly tested using a main() function that demonstrates the behavior of each structure with example data.

**Screenshot of the Implementation**



**Performance Analysis**

**1. Time Complexity of Basic Operations**

Each data structure supports a set of core operations. Their time complexity depends on how the data is organized internally (Matuszek, 2025).

**Array**

* **Access (O(1))**:  
  Arrays provide constant-time access because elements are stored in contiguous memory. The memory address of any element can be calculated using the formula:  
  address = base\_address + index \* element\_size
* **Insertion (Best: O(1), Worst: O(n))**:
  + If inserting by overwriting a value at a specific index, it's O(1).
  + If inserting at the beginning or middle and shifting elements to the right, it takes O(n) in the worst case.
* **Deletion (Best: O(1), Worst: O(n))**:
  + Deleting by marking a position as None is O(1).
  + But if you want to maintain contiguous order (e.g., by shifting), the operation becomes O(n).

**Example**: Inserting at index 0 in an array of 1,000 items requires shifting all 1,000 items one index forward — an O(n) operation.

**Matrix**

* **Access (O(1))**:  
  Accessing or modifying any element using [row][col] is constant time, assuming row and column indices are valid.
* **Insertion (O(1))**:  
  Similar to array access — assigning a value to a specific cell requires no shifting or resizing.

**Example**: In a 2D matrix representing a game board, updating a tile is instantaneous regardless of board size.

**Stack (Array-based)**

* **Push (O(1))**:  
  Python’s list append is amortized O(1). It may occasionally involve resizing the underlying array, but on average it is constant time.
* **Pop (O(1))**:  
  Removing the last element with pop() is also O(1) as no shifting is required.

**Amortized O(1)**: Even though resizing (e.g., doubling capacity) takes O(n), it happens infrequently. So on average, each push is still O(1).

**Queue (Array-based)**

* **Enqueue (O(1))**:  
  Appending to the end of a list is constant time.
* **Dequeue (O(n))**:  
  Removing from the front using pop(0) takes linear time, as all subsequent elements must be shifted one position left.

**Example**: If you dequeue 10,000 items, Python internally shifts each remaining item each time — very inefficient.

**Singly Linked List**

* **Insert at Head (O(1))**:  
  Create a new node and point it to the current head — no shifting required.
* **Delete (O(n))**:  
  Requires traversal to find the node (unless it's at the head). Deletion itself is O(1) once found.
* **Search/Traversal (O(n))**:  
  No random access — must start at the head and walk through each node.

**Example**: Deleting the last node in a singly linked list with 100,000 elements requires 100,000 comparisons.

**Trade-offs: Arrays vs Linked Lists**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Array** | **Linked List** |
| **Memory Layout** | Contiguous block in memory | Scattered nodes linked by pointers |
| **Access Time** | O(1) for any index (fast, cache-friendly) | O(n), must traverse from head |
| **Insertion/Deletion (front)** | O(n), due to shifting | O(1), update head pointer |
| **Insertion/Deletion (middle)** | O(n), due to shifting | O(n), need to traverse to position |
| **Insertion/Deletion (end)** | O(1) if size known, O(n) otherwise | O(n) for singly linked list |
| **Resizing** | Expensive (copy to new larger array) | No resizing needed |
| **Memory Usage** | Compact — only element values | Extra memory for each next pointer |
| **Cache Efficiency** | High (contiguous storage) | Low (random memory access) |
| **Implementation Complexity** | Simple (built-in in many languages like Python) | More complex (manual node and pointer management) |

**Detailed Explanations**

* **Access**:  
  Arrays allow direct access via index. In contrast, linked lists require following the next reference from node to node, which slows down access.
* **Insert/Delete (Head)**:  
  In arrays, inserting at the head means all elements must be shifted — inefficient. In linked lists, it's just one pointer update — very efficient.
* **Insert/Delete (Middle/End)**:  
  Arrays can do this quickly *only* if you overwrite existing values. Otherwise, you must shift elements. Linked lists need traversal to reach the target, but once found, updates are constant time (Lafore et al., 2022).
* **Memory and Cache**:  
  Arrays are faster in practice due to **better CPU cache utilization**. Linked lists have pointer overhead and less cache locality — impacting performance in large-scale applications.

**Use Case Comparison**

|  |  |  |
| --- | --- | --- |
| **Use Case** | **Recommended Structure** | **Why?** |
| Fast element lookup by index | Array | Constant-time access |
| Undo/Redo stack (text editor) | Stack | LIFO structure matches natural undo operations |
| Scheduling tasks in order | Queue | Ensures FIFO task handling |
| Dynamic data size, frequent inserts | Linked List | Flexible, avoids resizing overhead |
| Graph adjacency list | Array of Linked Lists | Memory-efficient, flexible for sparse graphs |
| Polynomial operations | Linked List | Simplifies addition and multiplication of terms |

**Real-World Applications and Discussion**

**Arrays**

* **Use Cases:** Lookup tables, buffers, image processing (matrices), and hash table implementations.
* **Strengths:** Fast access and compact memory usage.
* **Limitations:** Fixed size (unless resized), slow insert/delete at arbitrary positions.

**Matrices**

* **Use Cases:** Graph adjacency matrices, game boards, numerical computing (e.g., NumPy).
* **Strengths:** Structured 2D representation, easy indexing.
* **Limitations:** Fixed row/column sizes, difficult dynamic resizing.

**Stacks**

* **Use Cases:**
  + Function call stacks (recursion).
  + Undo/Redo in editors.
  + Syntax parsing (e.g., matching brackets).
* **Strengths:** Efficient and simple for LIFO operations.
* **Limitations:** Not suitable when random access or queue behavior is needed.

**Queues**

* **Use Cases:**
  + Print queues.
  + Scheduling tasks (e.g., OS process scheduling).
  + Breadth-First Search (BFS).
* **Strengths:** Natural for handling tasks in order.
* **Limitations:** Dequeue from array-based queue can be slow unless circular or deque used.

**Singly Linked Lists**

* **Use Cases:**
  + Music/playlist navigation.
  + Memory-efficient insertions/deletions in certain scenarios.
  + Implementing hash table chaining.
* **Strengths:** Dynamic resizing, quick insertion at head.
* **Limitations:** No backward traversal, higher memory due to pointers, slower access.

**References**

Amin, A. A. H., Aladdin, A. M., Hasan, D. O., Mohammed-Taha, S. R., & Rashid, T. A. (2023). Enhancing algorithm selection through comprehensive performance evaluation: Statistical analysis of stochastic algorithms. *Computation*, *11*(11), 231.

Lafore, R., Broder, A., & Canning, J. (2022). *Data Structures & Algorithms in Python*. Addison-Wesley Professional.

Matuszek, D. (2025). *Quick Data Structures*. CRC Press.

Mostert, W., Malan, K. M., & Engelbrecht, A. P. (2021). A feature selection algorithm performance metric for comparative analysis. *Algorithms*, *14*(3), 100.