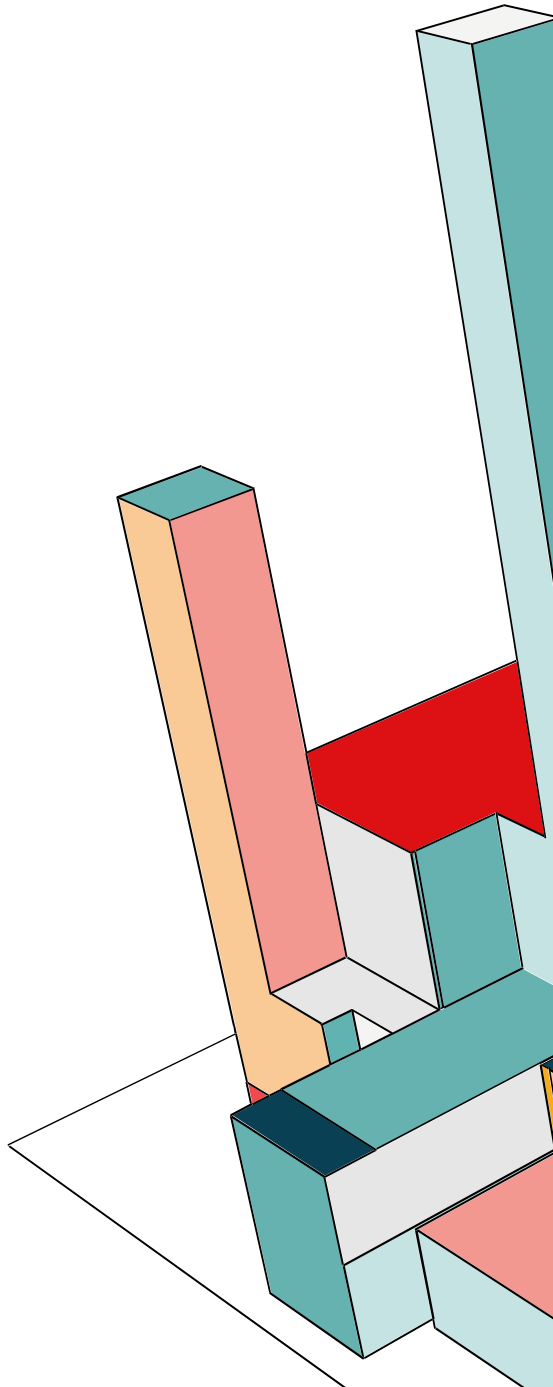
DATA STRUCTURE AND ALGORITHM

INTRODUCTION

- A stack ADT, a concrete data structure for a First In First out (FIFO) queue.
- Two sorting algorithms.
- Two network shortest path algorithms.



A STACK ADT, A CONCRETE DATA STRUCTURE FOR A FIRST IN FIRST OUT (FIFO) QUEUE.

STACKS AND QUEUES BASIC DATA STRUCTURES IN PROGRAMMING

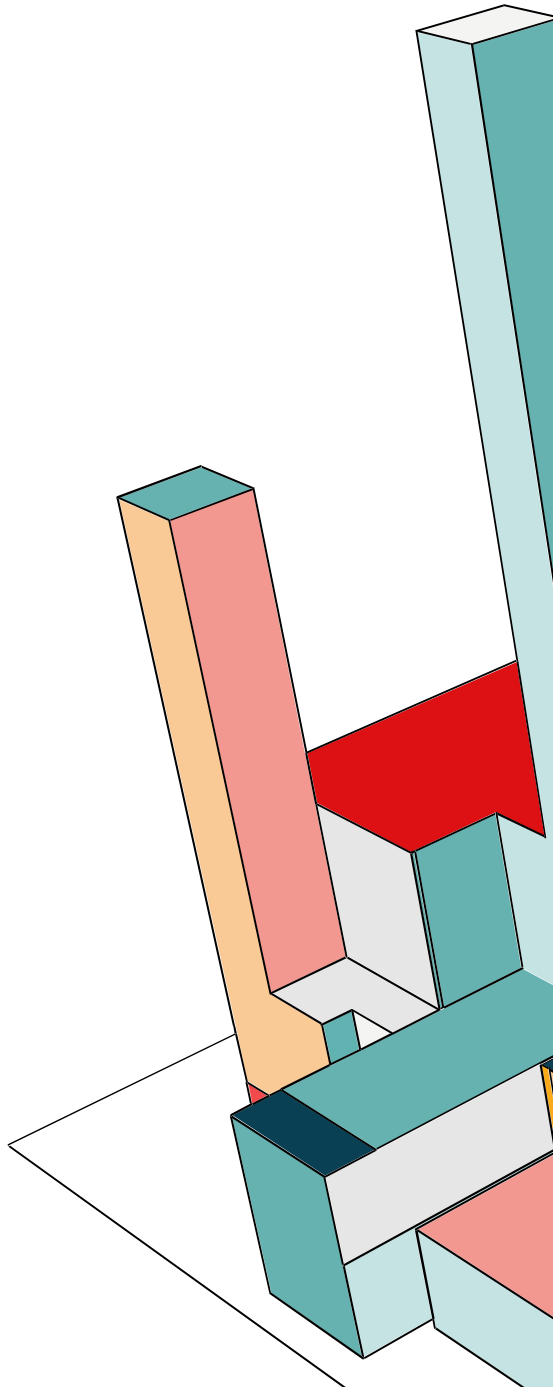
In programming, stacks and queues are two fundamental data structures that help manage data based on different rules. Understanding how these structures work is essential for developing efficient and optimized algorithms.

- **Stack**: Follows the **Last In First Out (LIFO)** principle, meaning the last element added is the first to be removed.
- **Queue**: Follows the **First In First Out (FIFO)** principle, meaning the first element added is the first to be removed.

WHAT IS A STACK?

Definition:

- A **stack** is a **linear data structure** that follows the **Last In First Out (LIFO)** principle. This means that the last element added to the stack will be the first one to be removed. Stacks are used in many programming applications where reversing order or managing function calls is important.



WHAT IS A STACK?

Key Concept - LIFO:

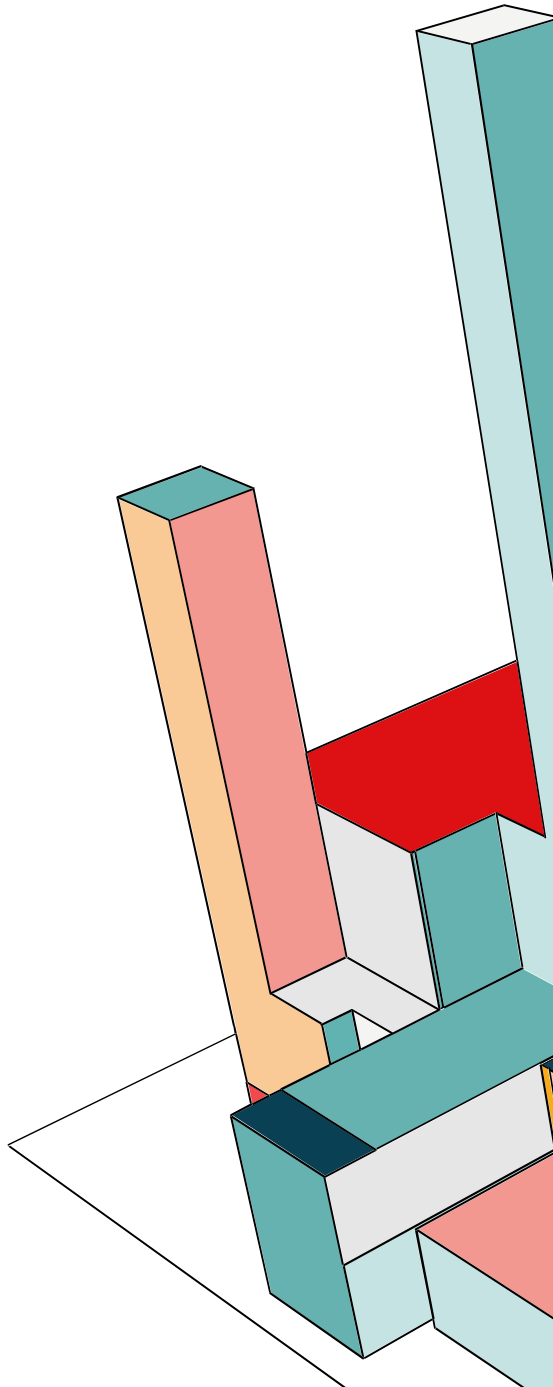
- **Last In First Out**: The most recently added item is the first to be removed.

Real-life Example:

- Imagine a **stack of books** where you place books on top of one another. The last book placed on the stack is the first book you'll take off.

Visualization:

1. Push item: Add an item on top of the stack.
2. Pop item: Remove the item from the top of the stack.



BASIC OPERATIONS OF A STACK

1. Push:

Description:

- The **Push** operation adds an element to the top of the stack. When you push an element, it becomes the new top item, and the previous top item is now second in the stack.

2. Pop:

Description:

- The **Pop** operation removes the element from the top of the stack and returns it. After popping, the next item below it becomes the new top item.

3. Peek/Top:

Description:

- The **Peek** (or **Top**) operation allows you to view the element at the top of the stack without removing it. This is useful for checking what the last pushed item is.

```java
public void push(Student student) {  3 usages
    Node newNode = new Node(student);
    newNode.next = top; // Point new node to the previous top
    top = newNode;      // Update top to be the new node
    size++;
    System.out.println("Inserted: " + student);
}

// Pop a student from the stack
public Student pop() {  1 usage
    if (isEmpty()) {
        System.out.println("Stack Underflow! No students to remove.");
        return null;
    }
    Student poppedStudent = top.student; // Get the student from the top node
    top = top.next;                    // Move top to the next node
    size--;
    return poppedStudent;
}

// Peek at the top student
public Student peek() {  no usages
    if (isEmpty()) {
        System.out.println("Stack is empty!");
        return null;
    }
    return top.student; // Return the student at the top node
}
```

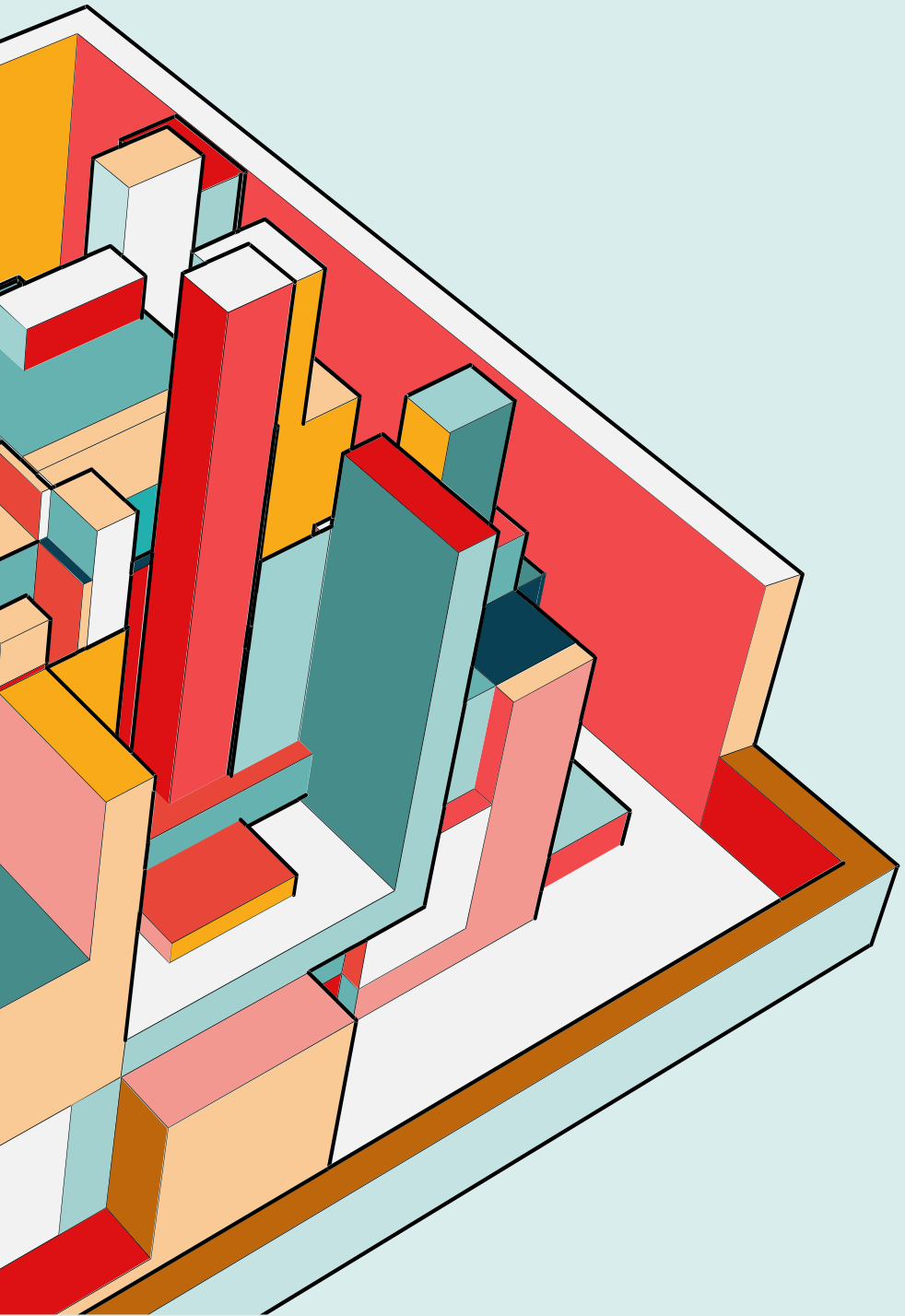
MAIN METHOD

Demonstrates the use of the stack operations, including pushing elements onto the stack, peeking at the top element, popping elements from the stack, and checking if the stack is empty.

```java
public void push(Student student) {  3 usages
    Node newNode = new Node(student);
    newNode.next = top; // Point new node to the previous top
    top = newNode;      // Update top to be the new node
    size++;
    System.out.println("Inserted: " + student);
}

// Pop a student from the stack
public Student pop() {  1 usage
    if (isEmpty()) {
        System.out.println("Stack Underflow! No students to remove.");
        return null;
    }
    Student poppedStudent = top.student; // Get the student from the top node
    top = top.next;                    // Move top to the next node
    size--;
    return poppedStudent;
}

// Peek at the top student
public Student peek() {  no usages
    if (isEmpty()) {
        System.out.println("Stack is empty!");
        return null;
    }
    return top.student; // Return the student at the top node
}
```



FIFO QUEUE: CONCRETE DATA STRUCTURE

Definition: A collection of elements with two principal operations: enqueue (add an element to the end) and dequeue (remove the front element).

Operations: Enqueue(x): Adds element x to the end. Dequeue(): Removes and returns the front element. Front(): Returns the front element without removing it. IsEmpty(): Checks if the queue is empty.

EXAMPLE QUEUE

- A queue is initialized using LinkedList.
- Elements are added to the queue using the add method.
- The remove method removes the head of the queue and returns it
- The peek method returns the head of the queue without removing it.
- The size method returns the number of elements in the queue.
- The isEmpty method checks whether the queue is empty.
- The poll method retrieves and removes the head of the queue, or returns null if the queue is empty

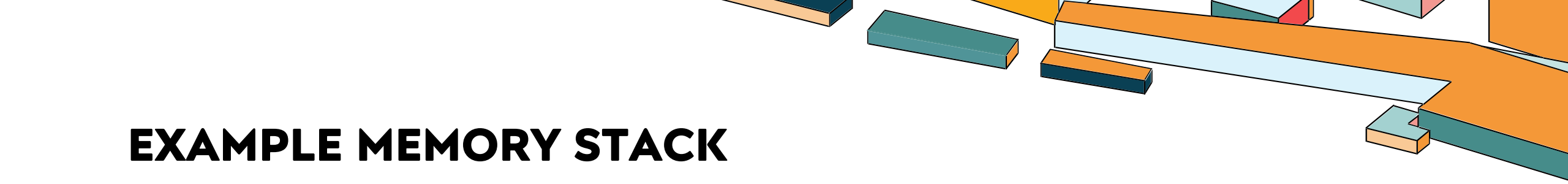
MEMORY STACK

A memory stack is a fundamental data structure in computer science and computing, operating on the Last In, First Out (LIFO) principle. This means that the most recently added item is the first to be removed. The memory stack is crucial for managing the execution of function calls in programming

MEMORY STACK IN IMPLEMENTING FUNCTION CALLS

The memory stack is essential in managing function calls, especially in recursive functions and nested function calls. The stack is used to store:

- Function Parameters: The arguments passed to the function.
- Return Address: The address in the code to return to after the function execution is complete.
- Local Variables: The variables that are declared within the function. Saved Registers: The state of CPU registers before the function call.



EXAMPLE MEMORY STACK

Package and Imports: The code is part of the stack package and imports the Stack class from java.util.

Class and Main Method: MemoryStackExample class contains the main method, the entry point of the application.

Input String: Defines the string input as "Hello, World!". Create Stack: Creates a Stack to store characters of the input string.

Push Characters: Loops through each character of input and pushes it onto the stack.

Reverse String: Pops each character from the stack and appends it to a StringBuilder, reversing the string in the process.

Class Definition: MemoryStack class manages a stack using an array.

Instance Variables: maxSize: Maximum size of the stack.

stackArray: Array to store stack elements.
top: Index of the top element in the stack.

Constructor: Initializes the stack with a specified size and sets top to -1 (indicating an empty stack).

Push Method: Adds an element to the top of the stack if it' s not full.

Pop Method: Removes and returns the top element of the stack if it' s not empty

TWO SORTING ALGORITHMS.

- Overview: This presentation explores the strengths and weaknesses of Tim Sort and Bubble Sort, providing a detailed comparison to understand when and why to use each algorithm in various scenarios

ALGORITHM OVERVIEW

Tim Sort:

Description:

Hybrid Algorithm: Combines the efficiency of Merge Sort with the simplicity of Insertion Sort. Target: Optimized for real-world data that often contains ordered sequences.

How It Works: Identifies and utilizes "runs" (pre-sorted sequences) to reduce the amount of work needed during the merge phase.

Complexity: Average Case: O(n log n)

Rationale: Efficient for typical scenarios, leveraging ordered runs.

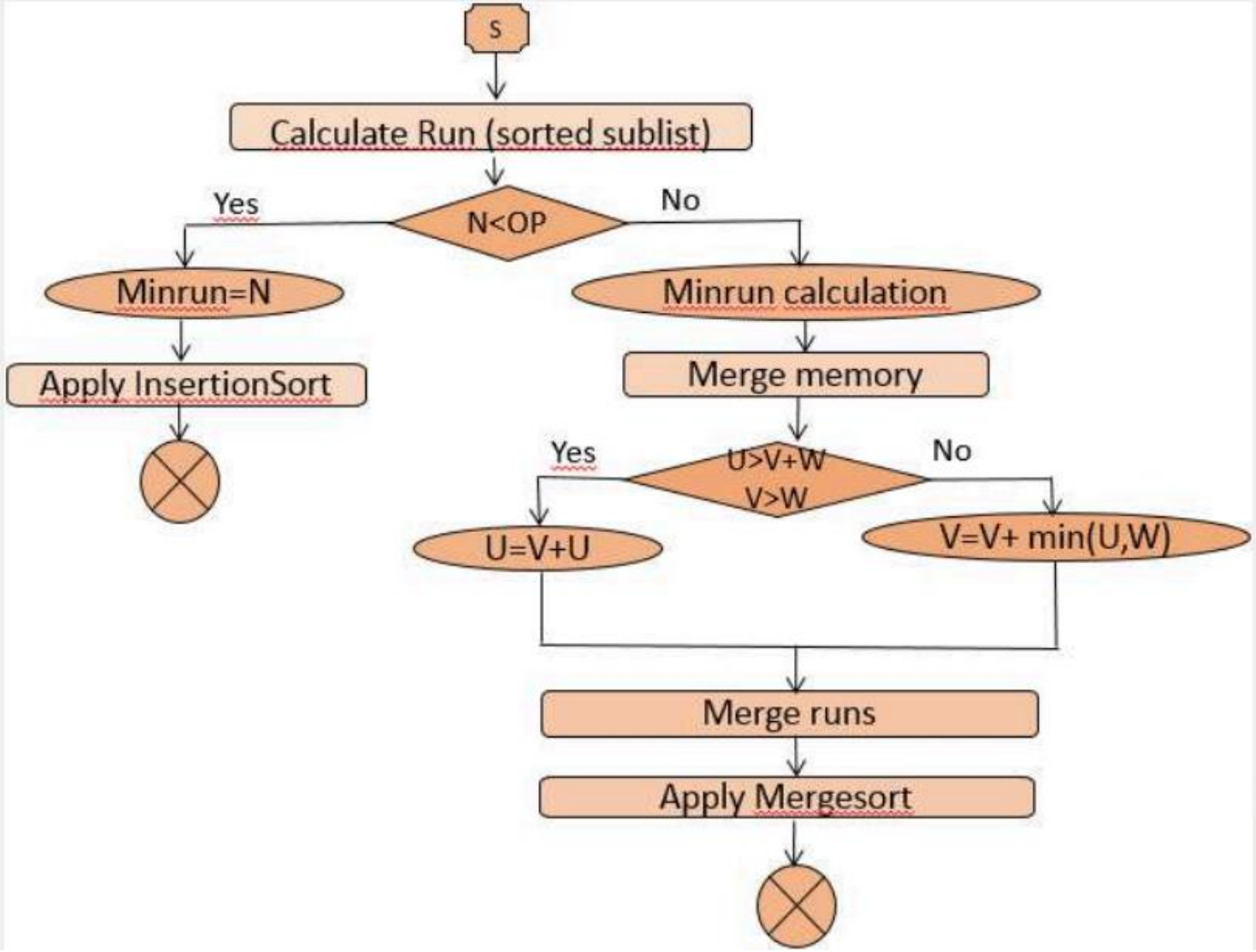
Worst Case: O(n log n)

Rationale: Maintains efficiency even with complex data arrangements.

Best Case: O(n)

Rationale: Optimal for already sorted data due to minimal merging

TIMSORT ALGORITHM



BUBBLE SORT:

1.DESCRIPTION:

SIMPLE ALGORITHM: REPEATEDLY STEPS THROUGH THE LIST, COMPARING ADJACENT ELEMENTS AND SWAPPING THEM IF NECESSARY.

TARGET: MAINLY USED FOR EDUCATIONAL PURPOSES DUE TO ITS SIMPLICITY.

HOW IT WORKS: CONTINUOUSLY "BUBBLES UP" THE LARGEST UNSORTED ELEMENT TO ITS CORRECT POSITION.

2.COMPLEXITY:

AVERAGE CASE: O(N^2)

RATIONALE: INEFFICIENT FOR LARGE DATASETS DUE TO REPEATED COMPARISONS.

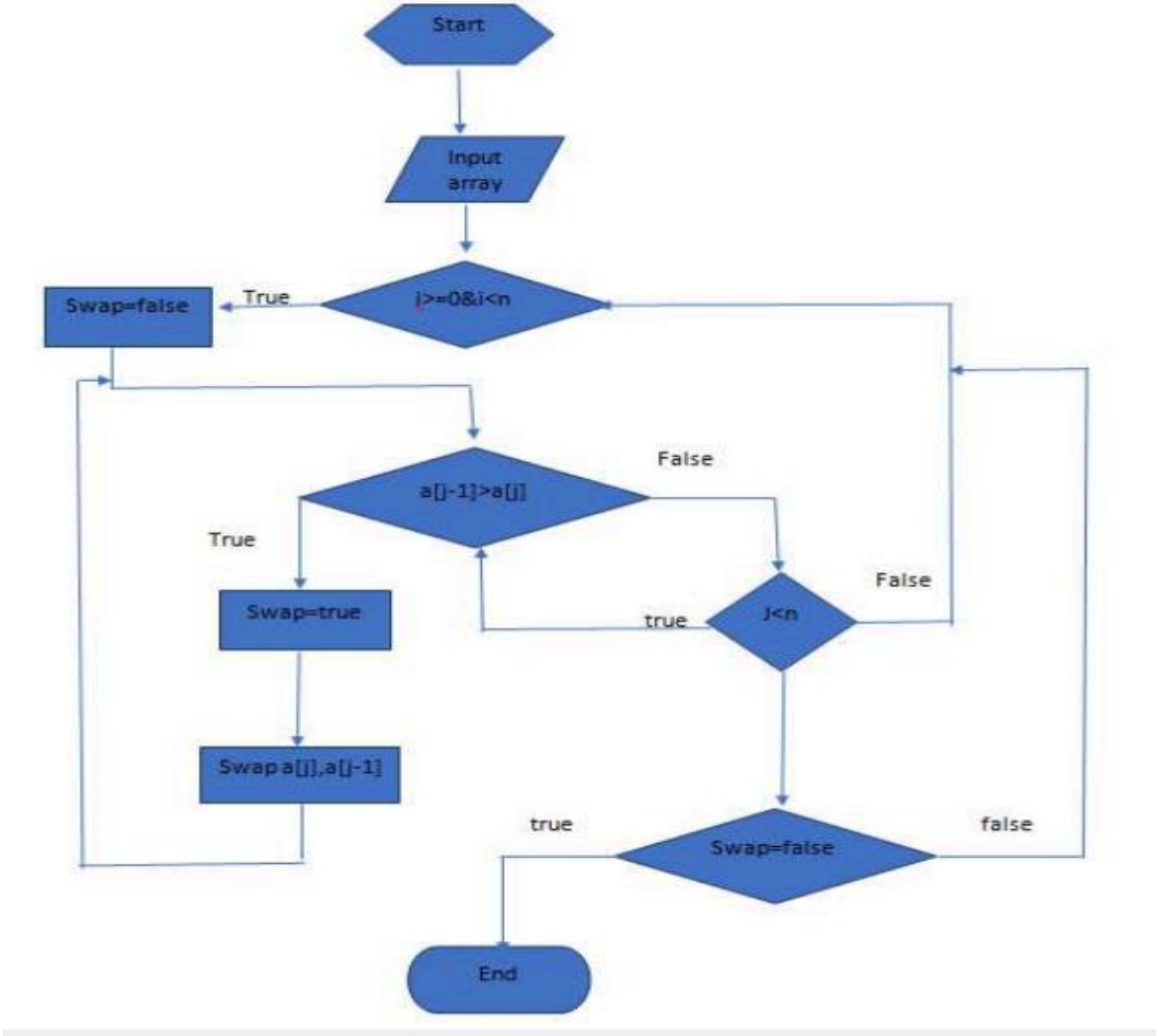
WORST CASE: O(N^2)

RATIONALE: MAXIMUM NUMBER OF SWAPS NEEDED FOR A REVERSE-ORDERED LIST.

BEST CASE: O(N)

RATIONALE: OPTIMAL IF THE LIST IS ALREADY SORTED OR NEARLY SORTED.

BUBBLE SORT ALGORITHM



BUBBLE SORT:

STRENGTHS:
SIMPLICITY: VERY EASY TO UNDERSTAND AND IMPLEMENT, USEFUL FOR EDUCATIONAL PURPOSES. IN-PLACE SORTING: REQUIRES NO ADDITIONAL MEMORY, MAKING IT IDEAL FOR SMALL DATASETS OR LIMITED MEMORY ENVIRONMENTS.

WEAKNESSES:
INEFFICIENCY: NOT PRACTICAL FOR LARGE DATASETS DUE TO ITS O(N^2) TIME COMPLEXITY. PERFORMANCE: SIGNIFICANTLY LESS EFFICIENT COMPARED TO MORE ADVANCED SORTING ALGORITHMS.

TIM SORT:

STRENGTHS:

EFFICIENCY: SUITABLE FOR LARGE DATASETS AND DATA WITH EXISTING ORDER.

STABILITY: PRESERVES THE RELATIVE ORDER OF EQUAL ELEMENTS.
WEAKNESSES:

MEMORY USAGE: REQUIRES ADDITIONAL SPACE, WHICH MAY BE A CONCERN FOR MEMORYLIMITED ENVIRONMENTS.

IMPLEMENTATION COMPLEXITY: MORE COMPLEX THAN SIMPLER ALGORITHMS, WHICH CAN
LEAD TO MORE POTENTIAL FOR BUGS AND MAINTENANCE CHALLENGES.

Two network shortest path algorithms.

•Title: Two Network Shortest Path Algorithms
•Subtitle: Dijkstra’s Algorithm & Bellman-Ford Algorithm

What are Shortest Path Algorithms?

- Algorithms designed to find the optimal path between nodes in a network or graph.
- **Goal**: To find the shortest or least costly path based on edge weights between nodes.
- **Applications**: GPS navigation systems, computer networks, traffic systems, etc.

Dijkstra’s Algorithm - Overview

- **Description**: Dijkstra’s Algorithm finds the shortest path from a source node to all other nodes in a graph with positive weights.
- **Condition**: The graph must not have negative edge weights.
- **Application**: Frequently used in navigation systems, finding the fastest route on maps.

Dijkstra’s Algorithm - How It Works

- **How It Works**:
  - Begin from the source node and assign a distance of 0 to it, and infinity to all other nodes.
  - In each step, choose the unvisited node with the smallest distance.
  - Update the distances to adjacent nodes if a shorter path is found through the current node.
  - Repeat until all nodes have been visited.
- **Time Complexity**: $O(V^2)$ for simple graphs, $O(V \log V)$ when using a priority queue.

Bellman-Ford Algorithm - Overview

- **Description**: Bellman-Ford Algorithm also finds the shortest path from a source node to all other nodes, and it can handle graphs with negative edge weights.
- **Condition**: The graph may have negative weights but must not have negative weight cycles.
- **Application**: Used in financial systems to analyze paths with negative interest rates, detecting negative weight cycles in graphs.

.

Bellman-Ford Algorithm - How It Works

- **How It Works**:
  - Initialize the distance to the source node as 0 and infinity for all other nodes.
  - For each edge, update the distance from the source to its neighboring nodes.
  - Repeat the process for V-1 times (V is the number of vertices).
  - If any edge can still be relaxed after V-1 iterations, the graph contains a negative weight cycle.
- **Time Complexity**: O(VE), where V is the number of vertices and E is the number of edges.

Comparison Between Dijkstra and Bellman-Ford

- **Dijkstra's Algorithm**:
  - Optimized for graphs with non-negative weights.
  - More efficient when using a priority queue for large graphs.
  - **Time Complexity**: O(V log V) with a priority queue.
- **Bellman-Ford Algorithm**:
  - Can handle negative weights and detect negative weight cycles.
  - Slower compared to Dijkstra's.
  - **Time Complexity**: O(VE).
- **When to Use?**:
  - **Dijkstra**: When the graph has non-negative weights.
  - **Bellman-Ford**: When the graph has negative weights or to detect negative cycles.

Conclusion

- **Key Takeaways**:
  - Both algorithms are essential for finding the shortest path in networks or graphs.
  - **Dijkstra** is faster and more efficient but only works with positive weights.
  - **Bellman-Ford** is more versatile, handling negative weights and detecting negative cycles but is slower.
- **Applications**:
  - Both are crucial in network systems, route optimization in traffic, and financial network analysis.

THANK YOU