FIFO Queue: A Concrete Data Structure

A FIFO queue is a fundamental data structure used in many applications, from managing tasks in operating systems to processing data in real-time systems.

What is a FIFO Queue?

A FIFO queue follows the "First In, First Out" principle. This means that the first element added to the queue is the first element to be removed.
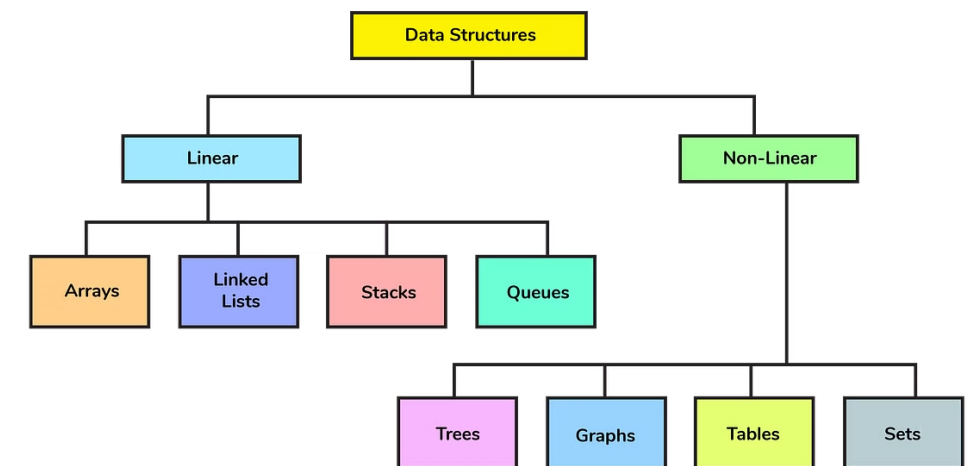
1 Real-World Example

Imagine a line at a grocery store. The first person in line is the first to be served, and the last person to join the line will be served last.

2 Data Structure Property

The order of elements in the queue is maintained, ensuring that elements are processed in the sequence they were added.

Underlying Data Structure: Array

A FIFO queue can be implemented using an array, where the front of the queue is the first element in the array, and the rear of the queue is the last element in the array.

Front

The front of the queue represents the element that was added first and is ready to be removed.

Rear

The rear of the queue represents the element that was added most recently and is waiting to be processed.

Enqueue Operation

Adding an element to the queue, called enqueue, involves placing the new element at the rear of the queue.

1

Step 1

Check if the queue is full. If full, resize the array or handle the overflow.

2

Step 2

Increment the rear index.

3

Step 3

Add the new element at the rear index of the array.

Adding
elements
into an array

JS TUTORIALS
CODEUNDERSCORED.COM

Dequeue Operation

Removing an element from the queue, called dequeue, involves taking out the element from the front of the queue.

Step 1

1

Check if the queue is empty. If empty, handle the underflow.

Step 2

2

Store the element at the front index.

Step 3

3

Increment the front index.

Step 4

4

Return the stored element, which was at the front.

C#
remove
elements
from an array

codevscolor.com

Handling Full and Empty Queues

In a fixed-size array, the queue can become full or empty. It's essential to manage these situations to ensure the queue operates correctly.

Full Queue	When the rear index reaches the end of the array, the queue is full. This condition requires either resizing the array or handling overflow gracefully by rejecting new elements.
Empty Queue	When the front index equals the rear index, the queue is empty. This condition requires handling underflow by ensuring that dequeue operations are not attempted when the queue is empty.

Time Complexity Analysis

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

The time complexity of the enqueue and dequeue operations in a FIFO queue implemented using an array is generally O(1), meaning they take constant time.

Enqueue

Adding an element to the rear of the queue is a constant-time operation, regardless of the queue size.

Dequeue

Removing an element from the front of the queue is also a constant-time operation.

Space Complexity Analysis

The space complexity of a FIFO queue implemented using an array is O(n), meaning it takes linear space proportional to the number of elements in the queue.

Fixed Size

For a fixed-size array, the space used is constant, independent of the number of elements.

Dynamic Size

For a dynamically resizing array, the space used is proportional to the number of elements in the queue.

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Example Implementation in Python

Here is a simple Python implementation of a FIFO queue using an array:

PROGRAMMING AND DESIGN

Conclusion and Key Takeaways

The FIFO queue is a fundamental data structure. It is used in various applications like task scheduling, data processing, and more. It can be efficiently implemented using an array with proper handling of full and empty conditions.

1 **Efficiency**

The enqueue and dequeue operations in a FIFO queue have constant time complexity.

2 **Applications**

Queues are crucial for managing resources, processing data, and implementing various algorithms.

3 **Understanding**

Understanding the concept of FIFO queues is essential for working with various data structures and algorithms.

Comparing Bubble Sort and Quick Sort

We'll explore two fundamental sorting algorithms: Bubble Sort and Quick Sort. These algorithms showcase different approaches to organizing data efficiently.

Bubble Sort: An Overview

Comparison-Based

Bubble Sort compares adjacent elements and swaps them if they're in the wrong order.

Iterative

The process repeats until the entire list is sorted, with no more swaps needed.

Simplicity

Known for its straightforward implementation, making it easy to understand and code.

Definition of Bubble Sort
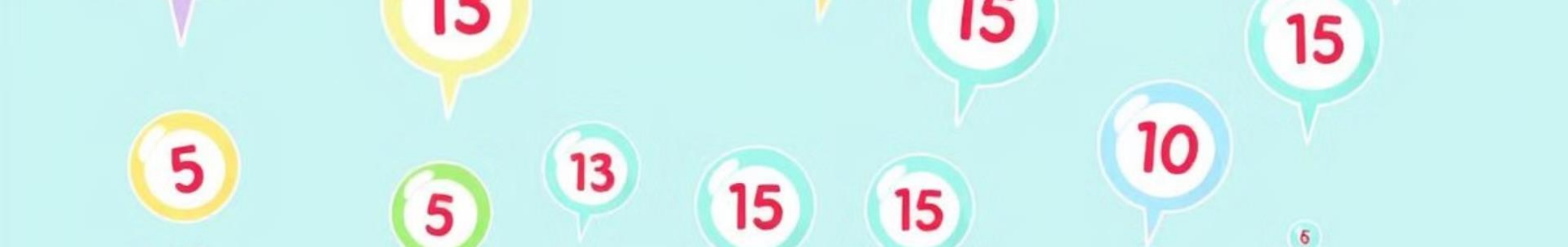
Comparison-Based

Bubble Sort repeatedly steps through the list, comparing adjacent elements and swapping them if they're in the wrong order.
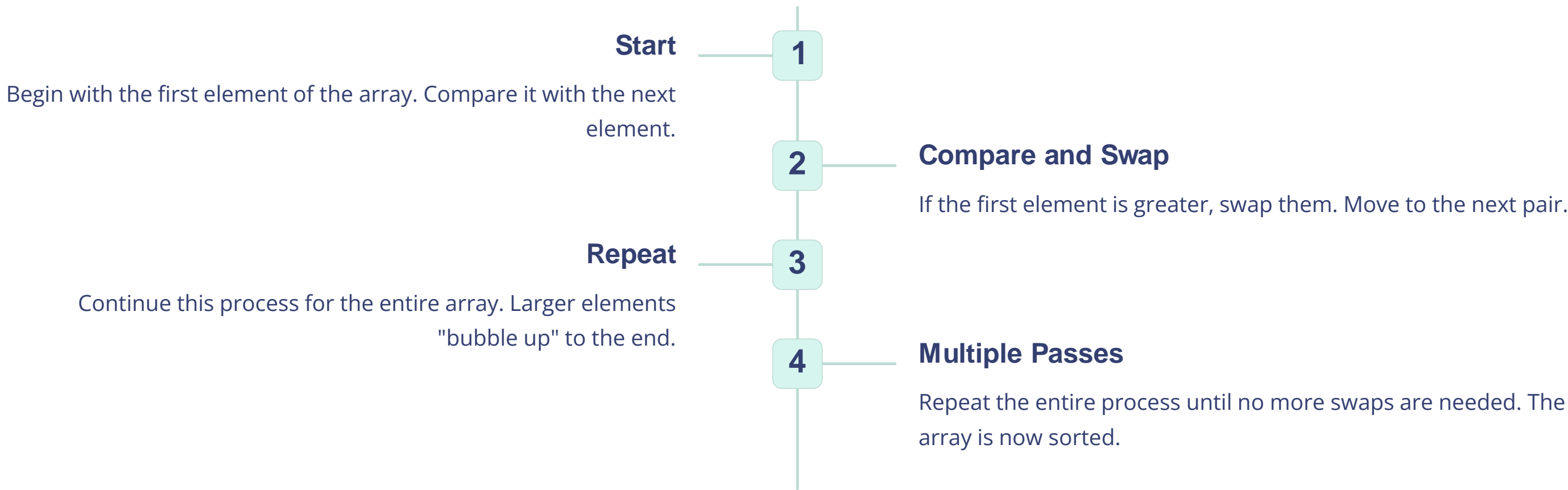
In-Place Algorithm

It sorts the elements within the given array without requiring additional memory space.

Stable Sort

Bubble Sort maintains the relative order of equal elements in the sorted output.

Mechanism of Bubble Sort

Start — **1**

Begin with the first element of the array. Compare it with the next element.

2 — **Compare and Swap**

If the first element is greater, swap them. Move to the next pair.

Repeat — **3**

Continue this process for the entire array. Larger elements "bubble up" to the end.

4 — **Multiple Passes**

Repeat the entire process until no more swaps are needed. The array is now sorted.

Advantages and Disadvantages of Bubble Sort

Advantages

- Simple to understand and implement

- Requires minimal additional memory

- Detects if the list is already sorted

Disadvantages

- Poor time complexity of $O(n^2)$

- Inefficient for large datasets

- Performs poorly compared to advanced algorithms

Code

```java
public class BubbleSort {
    // Method to perform Bubble Sort
    public static void bubbleSort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    // Swap array[j+1] and array[j]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }

    // Method to print the array
    public static void printArray(int[] array) {
        for (int value : array) {
            System.out.print(value + " ");
        }
        System.out.println();
    }

    // Main method to demonstrate Bubble Sort

    public static void main(String[] args) {
        int[] array = {115, 22, 103, 21, 39, 17, 100};
        System.out.println("Unsorted array:");
        printArray(array);

        bubbleSort(array);

        System.out.println("Sorted array:");
        printArray(array);
    }
}
```

Pass 1:

- Compare 115 and 22: Swap → {22, 115, 103, 21, 39, 17, 100}
- Compare 115 and 103: Swap → {22, 103, 115, 21, 39, 17, 100}
- Compare 115 and 21: Swap → {22, 103, 21, 115, 39, 17, 100}
- Compare 115 and 39: Swap → {22, 103, 21, 39, 115, 17, 100}
- Compare 115 and 17: Swap → {22, 103, 21, 39, 17, 115, 100}
- Compare 115 and 100: Swap → {22, 103, 21, 39, 17, 100, 115}

Pass 2:

- Compare 22 and 103: No Swap
- Compare 103 and 21: Swap → {22, 21, 103, 39, 17, 100, 115}
- Compare 103 and 39: Swap → {22, 21, 39, 103, 17, 100, 115}
- Compare 103 and 17: Swap → {22, 21, 39, 17, 103, 100, 115}
- Compare 103 and 100: Swap → {22, 21, 39, 17, 100, 103, 115}

Pass 3:

- Compare 22 and 21: Swap → {21, 22, 39, 17, 100, 103, 115}
- Compare 22 and 39: No Swap
- Compare 39 and 17: Swap → {21, 22, 17, 39, 100, 103, 115}
- Compare 39 and 100: No Swap
- Compare 100 and 103: No Swap

Pass 4:

- Compare 21 and 22: No Swap
- Compare 22 and 17: Swap → {21, 17, 22, 39, 100, 103, 115}
- Compare 22 and 39: No Swap
- Compare 39 and 100: No Swap
- Compare 100 and 103: No Swap

Pass 5:

- Compare 21 and 17: Swap → {17, 21, 22, 39, 100, 103, 115}
- Compare 21 and 22: No Swap
- Compare 22 and 39: No Swap
- Compare 39 and 100: No Swap
- Compare 100 and 103: No Swap

Pass 6:

- Compare 17 and 21: No Swap
- Compare 21 and 22: No Swap
- Compare 22 and 39: No Swap
- Compare 39 and 100: No Swap
- Compare 100 and 103: No Swap

Total Passes: 6 (with multiple comparisons per pass)

-> Sorted Array (Bubble Sort): {17, 21, 22, 39, 100, 103, 115}

Quick Sort: An Overview

Divide and Conquer

Quick Sort uses a divide-and-conquer strategy to efficiently sort elements.

Pivot-Based

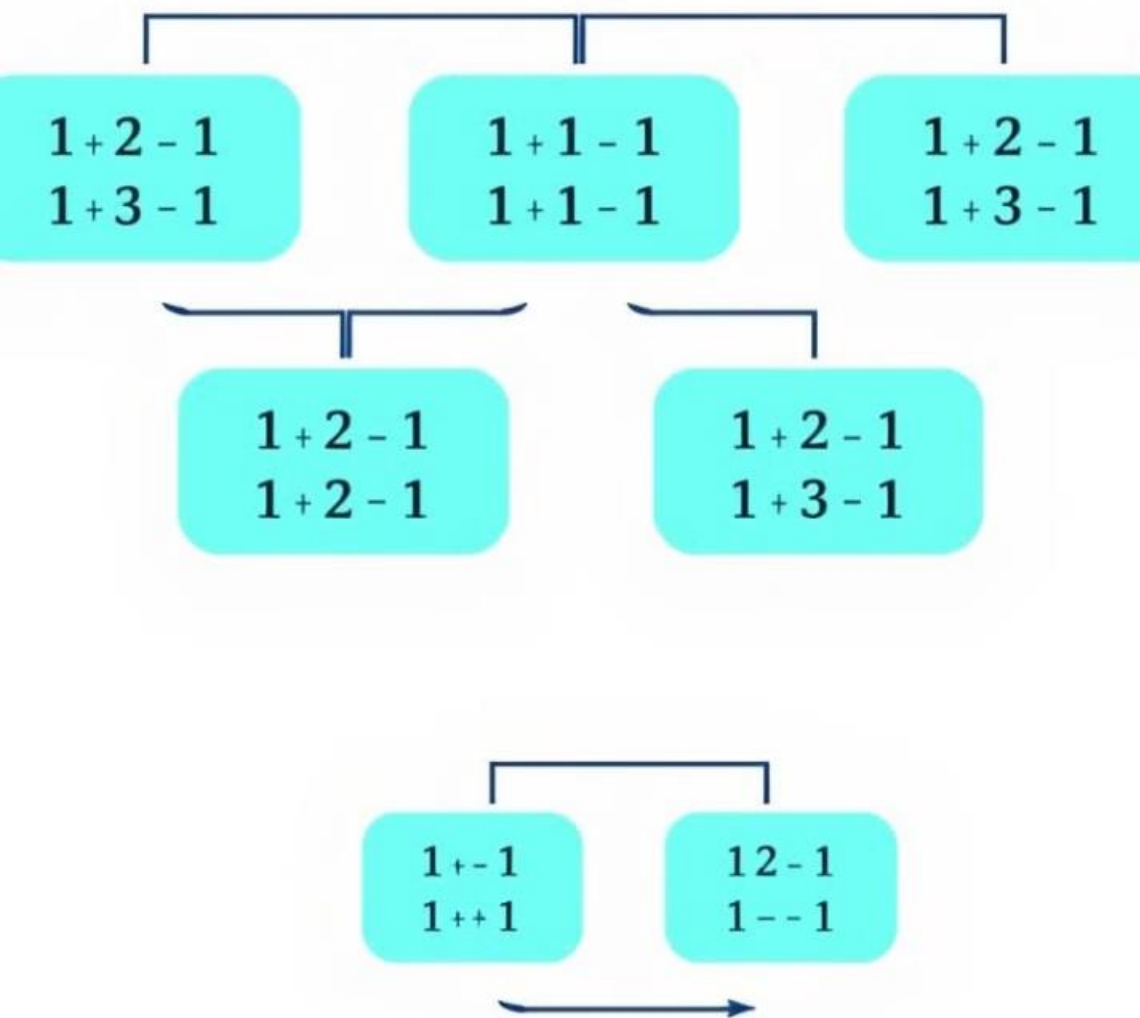It selects a 'pivot' element and partitions the array around it.

Recursive

The algorithm recursively applies the same process to smaller subarrays.

Definition of Quick Sort

Divide-and-Conquer

Quick Sort divides the array into smaller subarrays, sorts them independently, and combines the results.

Partitioning

It uses a pivot element to partition the array into two halves.

In-Place Sorting

Quick Sort typically sorts the array in-place, minimizing additional memory usage.

Mechanism of Quick Sort

1 — Choose Pivot

Select a pivot element from the array, often the last or a random element.

2 — Partitioning

Rearrange the array so elements smaller than the pivot are on the left, larger on the right.

3 — Recursive Sorting

Recursively apply Quick Sort to the subarrays on the left and right of the pivot.

4 — Combine

The sorted subarrays are already in place, forming the final sorted array.

Advantages and Disadvantages of Quick Sort

Advantages

- Efficient for large datasets

- Average time complexity of O(n log n)

- In-place sorting with low memory usage

Disadvantages

- Worst-case time complexity of $O(n^2)$

- Not stable (may change order of equal elements)

- Performance depends on pivot selection

Code

```java
public class QuickSort {
    // Method to perform Quick Sort
    public static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(array, low, high);
            quickSort(array, low, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, high);
        }
    }

    // Method to partition the array
    private static int partition(int[] array, int low, int high) {
        int pivot = array[high]; // Choosing the last element as
pivot
        int i = (low - 1); // Index of the smaller element
        for (int j = low; j < high; j++) {
            // If the current element is smaller than or equal to the
pivot
            if (array[j] < pivot) {
                i++;
                // Swap array[i] and array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        // Swap array[i + 1] and array[high] (or pivot)
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;
        return i + 1; // Return the pivot index
    }

    // Method to print the array
    public static void printArray(int[] array) {
        for (int value : array) {
            System.out.print(value + " ");
        }
        System.out.println();
    }

    // Main method to demonstrate Quick Sort
    public static void main(String[] args) {
        int[] array = {115, 22, 103, 21, 39, 17, 100};
        System.out.println("Unsorted array:");
        printArray(array);

        quickSort(array, 0, array.length - 1);

        System.out.println("Sorted array:");
        printArray(array);
    }
}
```

Initial Array: {115, 22, 103, 21, 39, 17, 100}

Pivot: 100

Partitioning: Rearranging elements around the pivot 100:

- Compare 115: No swap
- Compare 22: Swap with 115 → {22, 115, 103, 21, 39, 17, 100}
- Compare 103: No swap
- Compare 21: Swap with 115 → {22, 21, 103, 115, 39, 17, 100}
- Compare 39: Swap with 115 → {22, 21, 39, 115, 103, 17, 100}
- Compare 17: Swap with 115 → {22, 21, 39, 17, 115, 103, 100}
- Swap pivot 100 with 115 → {22, 21, 39, 17, 100, 103, 115}

Pivot Index: 4

- Recursively Sort Left Partition {22, 21, 39, 17}:
- Pivot: 17
- Partitioning results in: {17, 21, 39, 22}
- Now sort {21, 39, 22} with pivot 22:
- Resulting in: {21, 22, 39}

Recursively Sort Right Partition {103, 115}: No changes needed as they are already sorted.

-> Sorted Array (Quick Sort): {17, 21, 22, 39, 100, 103, 115}

# Comparison of Bubble Sort and Quick Sort

Aspect	Bubble Sort	Quick Sort
Time Complexity (Average)	$O(n^2)$	$O(n \log n)$
Space Complexity	$O(1)$	$O(\log n)$
Stability	Stable	Not Stable
Efficiency for Large Datasets	Poor	Excellent
Implementation Complexity	Simple	Moderate
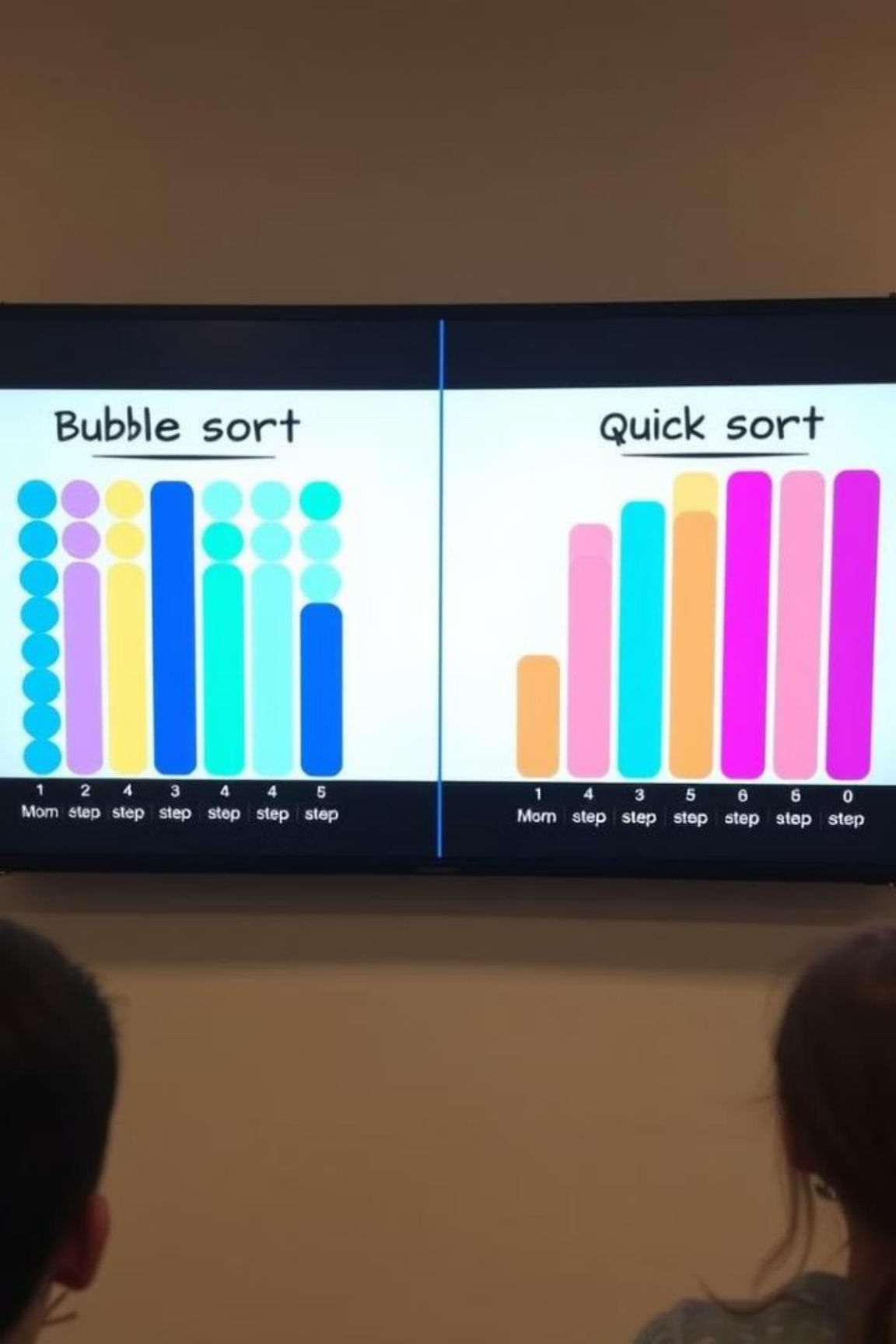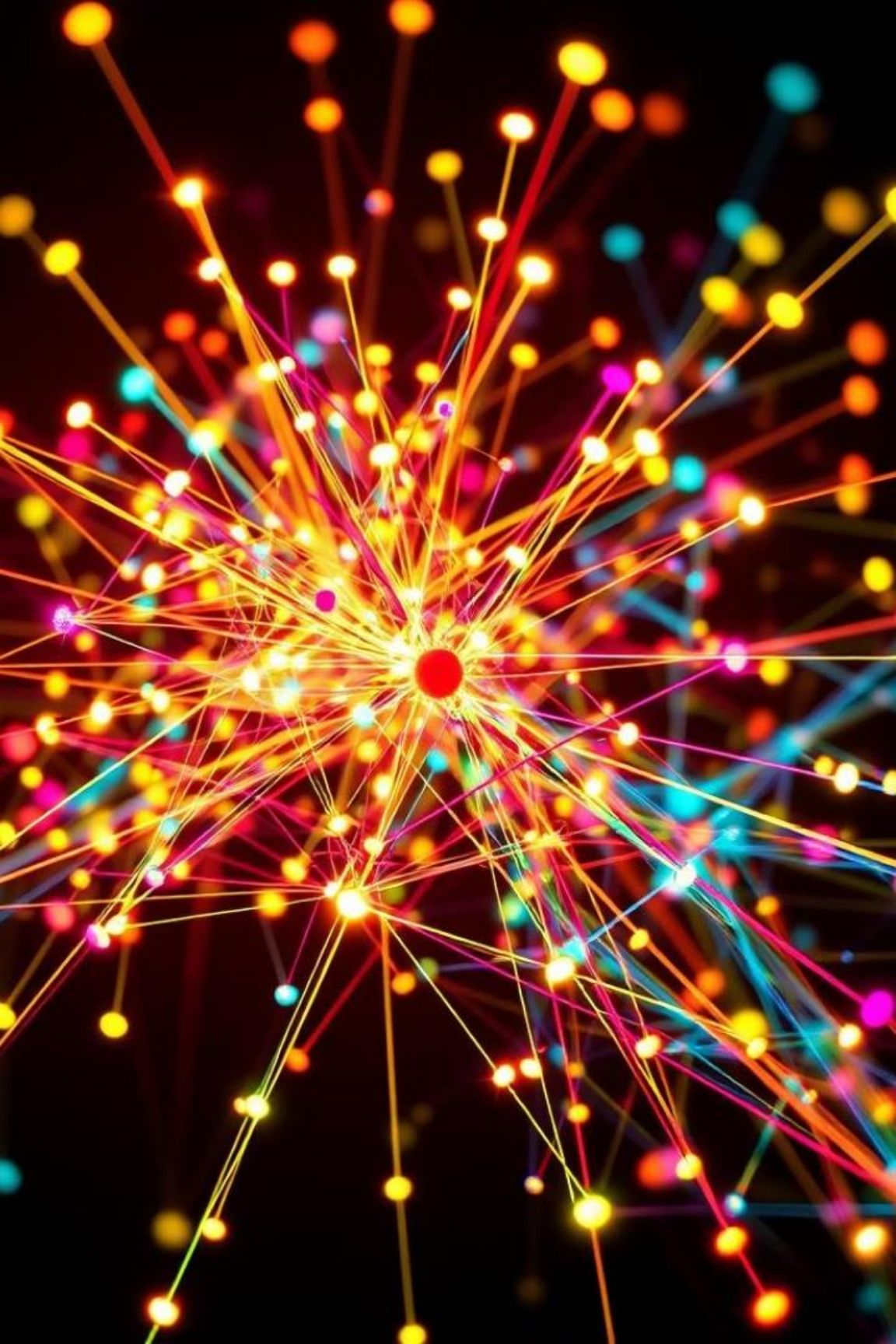
Network Shortest Path Algorithms

Shortest path algorithms find the least-cost path between two nodes in a network. Two common algorithms are Dijkstra's Algorithm and Prim-Jarnik Algorithm.
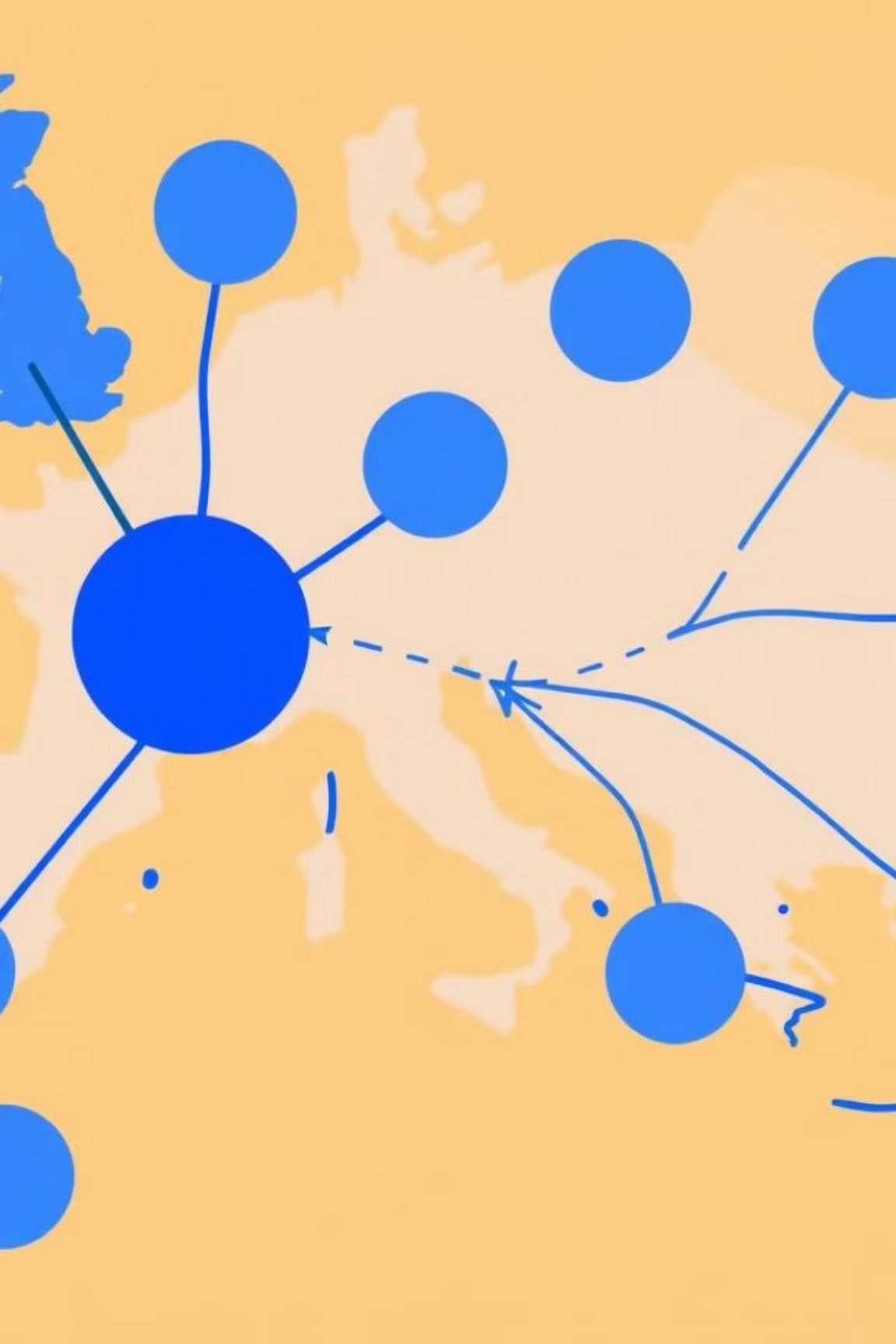
Dijkstra's Algorithm

1 **Source to All Nodes**

Dijkstra's Algorithm finds the shortest path from a starting point to every other node in the network.

2 **Greedy Approach**

The algorithm iteratively builds the shortest path tree by making the most optimal choice at each step.

3 **Weighted Edges**

Dijkstra's Algorithm works with graphs where edges have associated weights, representing costs or distances.

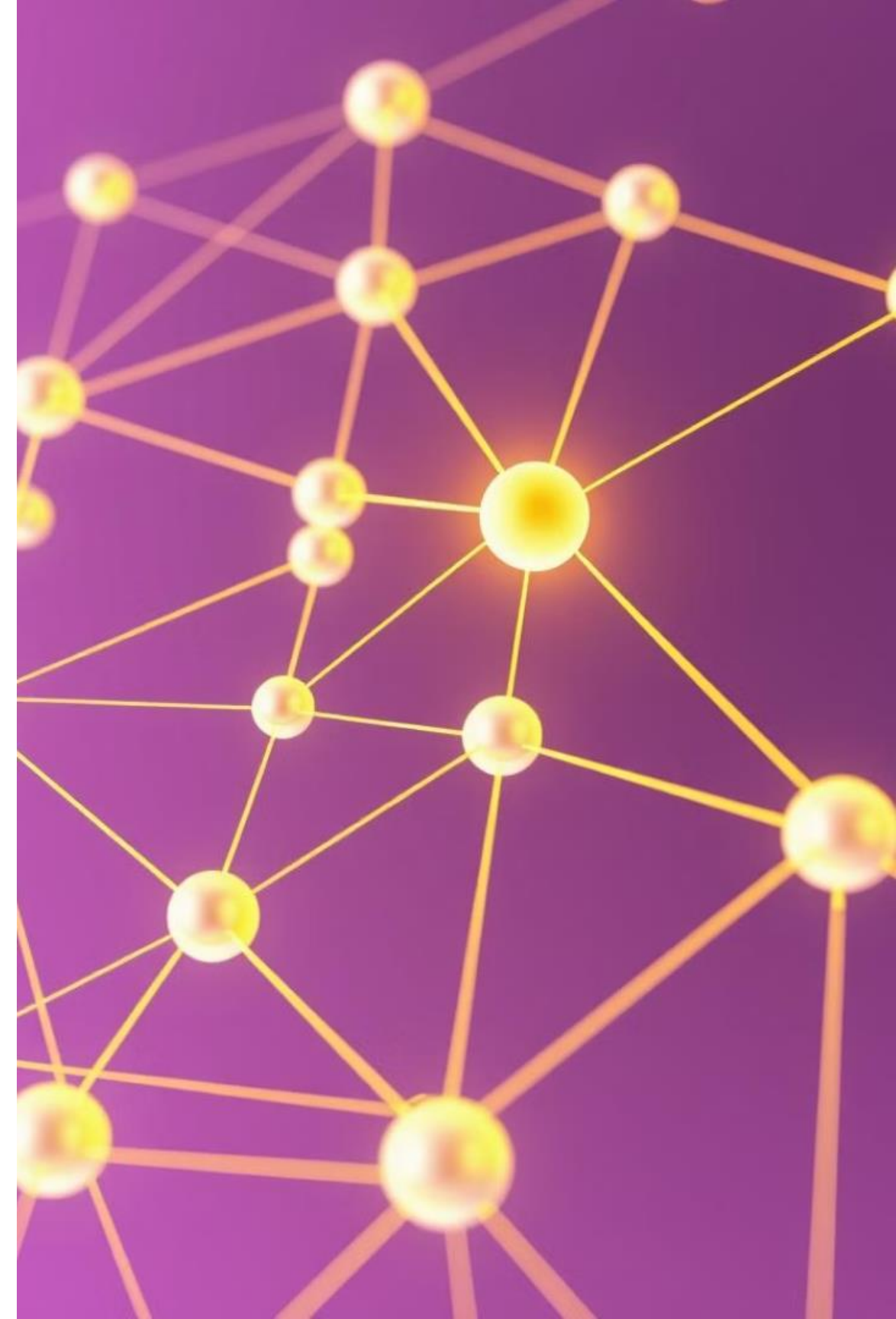# Purpose of Dijkstra's Algorithm

Navigation

Finding the shortest route for a car, bus, or airplane between two locations.

Network Routing

Determining the most efficient path for data packets to travel across a network.

Resource Allocation

Identifying the least-cost way to assign resources to different tasks in a project.

Mechanism of Dijkstra's Algorithm

1   **Initialization**

Set the distance to the starting node to 0, and the distance to all other nodes to infinity.

2   **Iteration**

Select the node with the smallest distance, and update the distances to its neighbors.

3   **Termination**

Repeat the iteration process until all nodes have been visited, or the shortest path to all nodes has been found.

Example of Dijkstra's Algorithm

Code

```java
import java.util.*;

class Graph {
    private final int vertices;
    private final List<List<Node>> adjacencyList;

    public Graph(int vertices) {
        this.vertices = vertices;
        adjacencyList = new ArrayList<>(vertices);
        for (int i = 0; i < vertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

    public void addEdge(int source, int destination, int weight) {
        adjacencyList.get(source).add(new Node(destination, weight));
        adjacencyList.get(destination).add(new Node(source, weight)); // for
undirected graph
    }

    public void dijkstra(int start) {
        int[] distances = new int[vertices];
        boolean[] visited = new boolean[vertices];
        PriorityQueue<Node> queue = new PriorityQueue<>(vertices,
Comparator.comparingInt(node -> node.cost));

        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[start] = 0;
        queue.add(new Node(start, 0));

        while (!queue.isEmpty()) {
            int currentNode = queue.poll().node;
            visited[currentNode] = true;

            for (Node neighbor : adjacencyList.get(currentNode)) {
                if (!visited[neighbor.node]) {
                    int newDist = distances[currentNode] + neighbor.cost;
                    if (newDist < distances[neighbor.node]) {
                        distances[neighbor.node] = newDist;
                        queue.add(new Node(neighbor.node,
distances[neighbor.node]));
                    }
                }
            }
        }

        System.out.println("Shortest distances from node " + start + ": " +
Arrays.toString(distances));
    }
    static class Node {
        int node;
        int cost;

        Node(int node, int cost) {
            this.node = node;
            this.cost = cost;
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph(5);
        graph.addEdge(0, 1, 10);
        graph.addEdge(0, 2, 3);
        graph.addEdge(1, 2, 1);
        graph.addEdge(1, 3, 2);
        graph.addEdge(2, 1, 4);
        graph.addEdge(2, 3, 8);
        graph.addEdge(2, 4, 2);
        graph.addEdge(3, 4, 7);
        graph.addEdge(4, 3, 9);

        graph.dijkstra(0); // Start from node 0
    }
}
```

Output

```
C:\Users\Admin\.jdks\openjdk-21.0.2\bin\java.exe "
Shortest distances from node 0: [0, 4, 3, 6, 5]

Process finished with exit code 0
```

Prim-Jarnik Algorithm

Minimum Spanning Tree

The Prim-Jarnik Algorithm finds the minimum spanning tree of a graph, connecting all nodes with the least total edge weight.
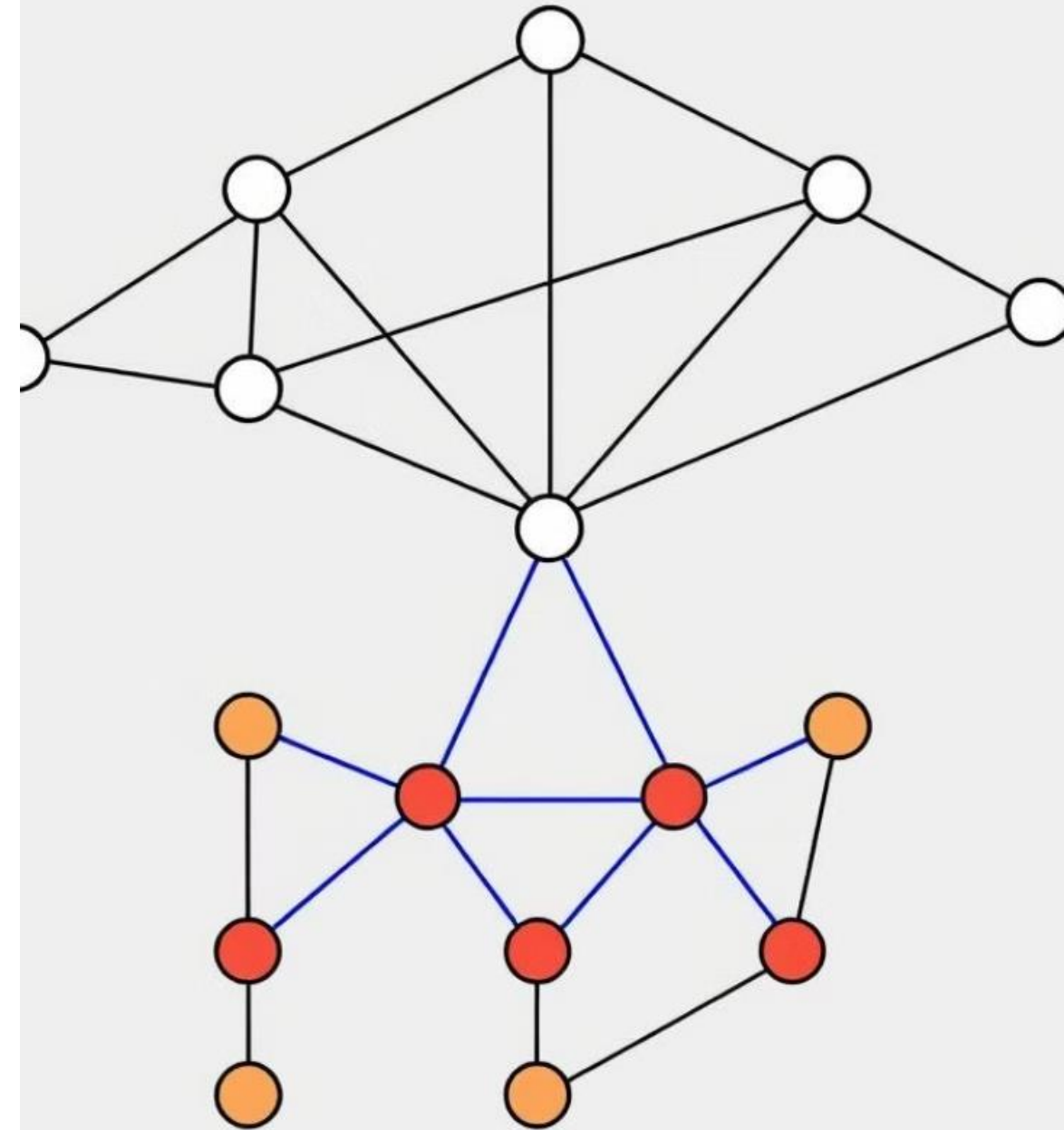
Greedy Approach

The algorithm builds the tree one edge at a time, always selecting the cheapest edge connecting a new node to the existing tree.

Connected Graph

Prim's Algorithm works with graphs that are connected, meaning there is a path between every pair of nodes.

Prim's Algorithm.

Purpose of Prim-Jarnik Algorithm

Network Design

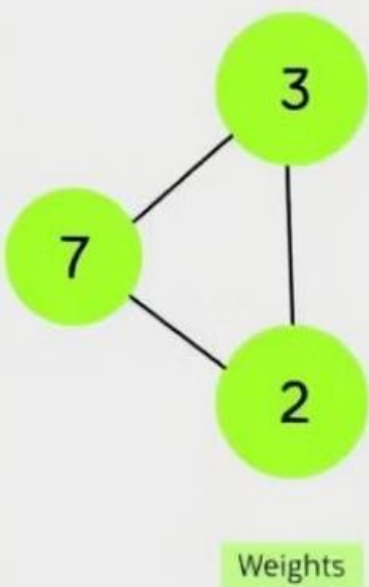Designing the most cost-effective network for connecting multiple locations.

Clustering

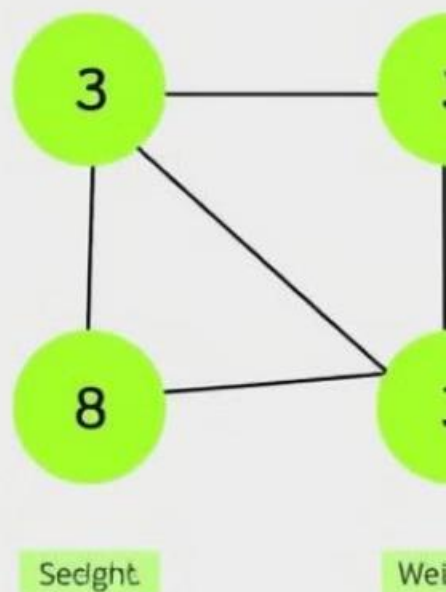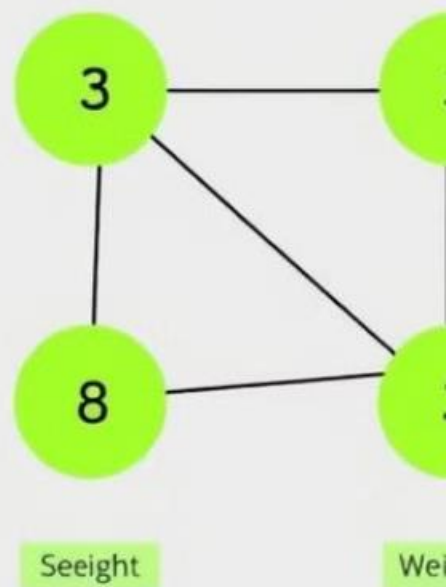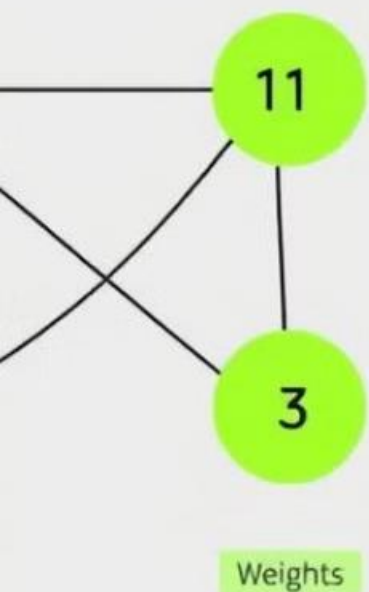Grouping similar data points into clusters based on their distances or similarities.

Circuit Design

Optimizing the connections between components in an electronic circuit.

Mechanism of Prim-Jarnik Algorithm

1

Initialization

Choose any node as the starting point, and add it to the tree.

2

Iteration

Select the cheapest edge that connects a node outside the tree to a node inside the tree, and add this edge to the tree.

3

Termination

Repeat the iteration process until all nodes are included in the tree, resulting in the minimum spanning tree.

Minmum spanning tree

Example of Prim-Jarnik Algorithm

Code

```java
import java.util.*;

class PrimJarnikAlgorithm {
    private final int vertices;
    private final List<List<Edge>> adjacencyList;

    public PrimJarnikAlgorithm(int vertices) {
        this.vertices = vertices;
        adjacencyList = new ArrayList<>(vertices);
        for (int i = 0; i < vertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

    public void addEdge(int source, int destination, int weight) {
        adjacencyList.get(source).add(new Edge(destination, weight));
        adjacencyList.get(destination).add(new Edge(source, weight)); // for
undirected graph
    }

    public void prim(int start) {
        boolean[] inMST = new boolean[vertices];
        Edge[] minEdge = new Edge[vertices];
        Arrays.fill(minEdge, new Edge(-1, Integer.MAX_VALUE));
        minEdge[start] = new Edge(start, 0);

        PriorityQueue<Edge> queue = new
PriorityQueue<>(Comparator.comparingInt(edge -> edge.weight));
        queue.add(minEdge[start]);

        while (!queue.isEmpty()) {
            Edge edge = queue.poll();
            int vertex = edge.destination;

            if (inMST[vertex]) continue; // Ignore if vertex already in MST

            inMST[vertex] = true; // Include vertex in MST

            for (Edge neighbor : adjacencyList.get(vertex)) {
                if (!inMST[neighbor.destination] && neighbor.weight <
minEdge[neighbor.destination].weight) {
                    minEdge[neighbor.destination] = neighbor;
                    queue.add(neighbor);
                }
            }
        }

        System.out.println("Minimum Spanning Tree edges:");
        for (Edge e : minEdge) {
            if (e.destination != -1) {
                System.out.println("Edge: " + e.source + " - " + e.destination + "
Weight: " + e.weight);
            }
        }
    }

    static class Edge {
        int source;
        int destination;
        int weight;

        Edge(int destination, int weight) {
            this.source = -1; // Not used in Prim's algorithm
            this.destination = destination;
            this.weight = weight;
        }
    }

    public static void main(String[] args) {
        PrimJarnikAlgorithm graph = new PrimJarnikAlgorithm(5);
        graph.addEdge(0, 1, 10);
        graph.addEdge(0, 2, 6);
        graph.addEdge(0, 3, 5);
        graph.addEdge(1, 3, 15);
        graph.addEdge(2, 3, 4);

        graph.prim(0); // Start from node 0
    }
}
```

Output

```
C:\Users\Admin\.jdks\openjdk-21.0.2\bin\java.exe
Minimum Spanning Tree edges:
Edge: -1 - 0 Weight: 0
Edge: -1 - 1 Weight: 10
Edge: -1 - 2 Weight: 4
Edge: -1 - 3 Weight: 5

Process finished with exit code 0
```