# Searching in Java

Searching means **finding a required element** from a collection of data (array, list, tree, etc.).

There are mainly **two categories** of searching in Java:

## 1. Linear Search (Sequential Search)

## Concept

It checks elements **one-by-one** from start to end until the key is found.

## When to use?

- Array is **unsorted**
- Data is **small**
- Simple implementation is needed

## Time Complexity

- Best: **O(1)**
- Worst: **O(n)**

## Java Code

```
public class LinearSearch {
    public static int linearSearch(int[] arr, int key) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == key)
                return i;
        }
        return -1; // not found
    }

    public static void main(String[] args) {
        int[] nums = {10, 20, 30, 40, 50};
        int res = linearSearch(nums, 30);
        System.out.println("Element found at index: " + res);
    }
}
```

## Binary Search

## Concept

Binary Search works on **sorted arrays only**.
It repeatedly divides the array into **two halves**.

## Steps

1. Find mid

2. If key == mid → return index
3. If key < mid → search in left half
4. Else → search in right half

## Time Complexity

- Best: **O(1)**
- Worst: **O(log n)**

## Java Code (Iterative Binary Search)

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int key) {
        int low = 0, high = arr.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (arr[mid] == key)
                return mid;
            else if (key < arr[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] nums = {10, 20, 30, 40, 50};
        int pos = binarySearch(nums, 40);
        System.out.println("Found at index: " + pos);
    }
}
```

## Sorting in Java

Sorting means arranging data in a **specific order**, usually **ascending or descending**.

Example:
Unsorted → {5, 2, 9, 1, 6}
Sorted → {1, 2, 5, 6, 9}

Sorting improves searching, data processing, and optimizes algorithms.

## Types of Sorting Algorithms

Sorting algorithms are mainly divided into **two categories**:

## Simple Sorting Algorithms (Basic, Easy)

## A) Bubble Sort

- Repeatedly compare **adjacent elements**
- Swap if wrong order
- Largest element “bubbles up” to end

**Time Complexity:**
Best – O(n)
Worst – $O(n^2)$

## Java Code

```
public class BubbleSort {
    public static void bubbleSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

## B) Selection Sort

- Select **minimum element** from unsorted part
- Put it at correct position

**Time Complexity:** Worst – $O(n^2)$

## Code

```
public class SelectionSort {
    public static void selectionSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            int min = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[min])
                    min = j;
            }
            int temp = arr[min];
            arr[min] = arr[i];
            arr[i] = temp;
        }
    }
}
```

## C) Insertion Sort

- Insert each element into its **correct place** in sorted part

**Time Complexity:**
Best: O(n)
Worst: $O(n^2)$

### Code

```
public class InsertionSort {
    public static void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;

            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }
}
```

# Efficient Sorting Algorithms (Fast, Used in real world)

## A) Merge Sort (Divide & Conquer)

- Divide array into two halves
- Sort each half
- Merge them

**Time Complexity:** O(n log n)
**Stable sorting**

### Code

```
public class MergeSort {

    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int[] L = new int[n1];
        int[] R = new int[n2];

        for (int i = 0; i < n1; i++)
            L[i] = arr[left + i];
        for (int i = 0; i < n2; i++)
            R[i] = arr[mid + 1 + i];
```

```
        int i = 0, j = 0, k = left;

        while (i < n1 && j < n2) {
            if (L[i] <= R[j])
                arr[k++] = L[i++];
            else
                arr[k++] = R[j++];
        }

        while (i < n1)
            arr[k++] = L[i++];
        while (j < n2)
            arr[k++] = R[j++];
    }
}
```

## B) Quick Sort

- Pick a pivot
- Place pivot in correct position
- Elements smaller → left
- Elements larger → right

**Time Complexity:**
Best/Average → O(n log n)
Worst → $O(n^2)$

## Code

```
public class QuickSort {

    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int p = partition(arr, low, high);
            quickSort(arr, low, p - 1);
            quickSort(arr, p + 1, high);
        }
    }

    static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }
}
```

## C) Heap Sort

- Build Max Heap
- Swap root with last element
- Heapify remaining array

**Time Complexity:** O(n log n)

## Code (short)

```
public class HeapSort {

    public void sort(int[] arr) {
        int n = arr.length;

        for (int i = n/2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        for (int i = n - 1; i > 0; i--) {
            int temp = arr[0]; arr[0] = arr[i]; arr[i] = temp;
            heapify(arr, i, 0);
        }
    }

    void heapify(int[] arr, int n, int i) {
        int largest = i;
        int l = 2*i + 1;
        int r = 2*i + 2;

        if (l < n && arr[l] > arr[largest])
            largest = l;

        if (r < n && arr[r] > arr[largest])
            largest = r;

        if (largest != i) {
            int temp = arr[i]; arr[i] = arr[largest]; arr[largest] = temp;
            heapify(arr, n, largest);
        }
    }
}
```

## Sorting Comparison Table

| Sorting | Best | Worst | Stable | Use Case |
|---|---|---|---|---|
| Bubble | O(n) | $O(n^2)$ | ✔ | Very small data |
| Selection | $O(n^2)$ | $O(n^2)$ | ✘ | When swaps should be minimized |
| Insertion | O(n) | $O(n^2)$ | ✔ | Nearly sorted data |
| Merge | O(n log n) | O(n log n) | ✔ | Large data, stable need |
| Quick | O(n log n) | $O(n^2)$ | ✘ | Most used, fast |
| Heap | O(n log n) | O(n log n) | ✘ | Priority queues |

### Built-in Sorting in Java

Java has powerful built-in methods:

### Arrays.sort() (Dual-Pivot QuickSort)

```
Arrays.sort(arr);
```

### Collections.sort() (Merge Sort)

```
Collections.sort(list);
```

# Introduction to Graph

A **Graph** is a non-linear data structure that consists of:

- **Vertices (nodes)**
- **Edges (connections between nodes)**

A Graph is written as:
**G = (V, E)**
Where **V = vertices**, **E = edges**

**Real-life examples:**

- Social networks (Facebook friend suggestions)
- Road networks (Google Maps)
- Web page links
- Electricity / water network

### Basic Terms of Graph

| Term | Meaning |
|---|---|
| **Vertex (Node)** | A point in graph |
| **Edge** | Link between two vertices |
| **Adjacent Nodes** | Nodes directly connected by an edge |
| **Degree** | Number of edges attached to a node |
| **Path** | Sequence of edges connecting two nodes |
| **Cycle** | Path that starts and ends at same node |
| **Connected Graph** | Every vertex is reachable |
| **Component** | Sub-graph that is connected |

### Types of Graph

#### 1. Directed Graph (Digraph)

Edges have a direction → A → B

**2. Undirected Graph**

Edges have no direction → A — B

**3. Weighted Graph**

Each edge has a cost/weight

**4. Unweighted Graph**

Edges have no weight

**5. Cyclic Graph**

Graph contains at least one cycle

**6. Acyclic Graph**

No cycle exists
Example: **Tree**

**7. Connected Graph**

All nodes can be reached

**8. Disconnected Graph**

Some nodes are not reachable

**9. Complete Graph**

Every vertex is connected to every other vertex

## Graph Representation in Java

**1. Adjacency Matrix**

2D array of size **V × V**
`matrix[u][v] = 1` if edge exists

**2. Adjacency List (MOST USED in coding)**

Each vertex stores a list of connected nodes
Implemented using:

- ArrayList
- LinkedList

# BFS (Breadth First Search)

**BFS meaning:**

- Traversal **level-wise**
- Uses **Queue**
- Good for **shortest path in unweighted graph**

**BFS Steps:**

1. Start from source node
2. Visit all neighbors
3. Use queue to store next nodes
4. Mark visited nodes

# DFS (Depth First Search)

## DFS meaning:

- Traversal goes **deep first**
- Uses **Stack** or **Recursion**
- Good for detecting **cycles**, **connected components**

## DFS Steps:

1. Start from source
2. Visit a neighbor
3. Continue deeper
4. Backtrack when no more neighbors

# Java Code – Graph (Adjacency List)

## Graph Implementation in Java

```
import java.util.*;

class Graph {
    private int V;                    // number of vertices
    private ArrayList<ArrayList<Integer>> adj; // adjacency list

    Graph(int V) {
        this.V = V;
        adj = new ArrayList<>();

        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }

    // Add edge (undirected)
    void addEdge(int u, int v) {
```

```
        adj.get(u).add(v);
        adj.get(v).add(u);
    }

    // Print graph
    void printGraph() {
        for (int i = 0; i < V; i++) {
            System.out.print(i + " -> ");
            for (int node : adj.get(i)) {
                System.out.print(node + " ");
            }
            System.out.println();
        }
    }
}
```

# BFS Traversal – Java Code

```
void bfs(int start) {
    boolean visited[] = new boolean[V];
    Queue<Integer> q = new LinkedList<>();

    visited[start] = true;
    q.add(start);

    while (!q.isEmpty()) {
        int node = q.poll();
        System.out.print(node + " ");

        for (int neigh : adj.get(node)) {
            if (!visited[neigh]) {
                visited[neigh] = true;
                q.add(neigh);
            }
        }
    }
}
```

# DFS Traversal – Java Code (Recursive)

```
void dfsUtil(int node, boolean[] visited) {
    visited[node] = true;
    System.out.print(node + " ");

    for (int neigh : adj.get(node)) {
        if (!visited[neigh]) {
            dfsUtil(neigh, visited);
        }
    }
}

void dfs(int start) {
    boolean visited[] = new boolean[V];
    dfsUtil(start, visited);
}
```

# Most Used Graph Methods (For Exams / Placements)

| Method | Use |
|---|---|
| `addEdge(u, v)` | Add edge to graph |
| `bfs(start)` | Level-wise traversal |
| `dfs(start)` | Depth-first traversal |
| `printGraph()` | Print adjacency list |
| `isVisited[]` | Track visited nodes |
| `Queue` | Used in BFS |
| `Stack / Recursion` | Used in DFS |
| `adj.get(u)` | Gives neighbors of u |

**Full Graph Example**

```
public class Main {
    public static void main(String[] args) {
        Graph g = new Graph(5);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 3);
        g.addEdge(2, 4);

        System.out.println("Graph:");
        g.printGraph();

        System.out.println("\nBFS from 0:");
        g.bfs(0);

        System.out.println("\nDFS from 0:");
        g.dfs(0);
    }
}
```

**Output Example**

```
Graph:
0 -> 1 2
1 -> 0 3
2 -> 0 4
3 -> 1
4 -> 2

BFS from 0:
0 1 2 3 4

DFS from 0:
0 1 3 2 4
```

# What is a Cycle in a Graph?

A **cycle** occurs when:
A node can be reached again through a path of edges.

Example:

```
1 — 2
|   |
4 — 3
```

Here, 1 → 2 → 3 → 4 → 1 forms a **cycle**.

## Cycle Detection in Undirected Graph

In an undirected graph, a cycle exists if:

We reach a **visited node that is NOT the parent** of the current node.

## Why track parent?

Because in undirected graphs, edges go both ways.
Example: 1 — 2
From 1 you go to 2, and 2 goes back to 1 → this is NOT a cycle.
So we skip the **parent**.

## Approach 1: DFS for Undirected Graph

## Condition for cycle:

If

```
visited[neigh] == true AND neigh != parent
```

→ Cycle exists.

## Java Code (DFS – Undirected)

```
// Method to check cycle using DFS in an Undirected Graph
boolean dfsCycle(int node, int parent, boolean[] visited,
                 ArrayList<ArrayList<Integer>> adj) {

    visited[node] = true;

    for (int neigh : adj.get(node)) {
        if (!visited[neigh]) {
            if (dfsCycle(neigh, node, visited, adj))
                return true;
        }
        // If visited and not parent => cycle
        else if (neigh != parent) {
            return true;
        }
    }
    return false;
}

// Main function to check all components
boolean isCycleUndirected(int V, ArrayList<ArrayList<Integer>> adj) {
```

```
    boolean[] visited = new boolean[V];

    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            if (dfsCycle(i, -1, visited, adj)) {
                return true;
            }
        }
    }
    return false;
}
```

# Cycle Detection in Directed Graph

In a directed graph, cycles are more strict:

You detect a cycle when you visit a node that is **already in the current DFS path**.

So we maintain two arrays:

| Array | Meaning |
|---|---|
| `visited[]` | Node already visited in any DFS |
| `recStack[]` (recursion stack) | Node currently in active path |

## Condition for cycle:

If

```
recStack[neigh] == true
```

→ Cycle exists.

## Java Code (DFS – Directed Graph)

```
// DFS for Directed Graph cycle detection
boolean dfsDirected(int node, boolean[] visited, boolean[] recStack,
                    ArrayList<ArrayList<Integer>> adj) {

    visited[node] = true;
    recStack[node] = true;   // mark node in current path

    for (int neigh : adj.get(node)) {
        if (!visited[neigh]) {
            if (dfsDirected(neigh, visited, recStack, adj))
                return true;
        }
        else if (recStack[neigh]) {  // node already in current path
            return true;             // cycle found
        }
    }

    recStack[node] = false;   // remove from current path
    return false;
}
```

```
// Check for all components
boolean isCycleDirected(int V, ArrayList<ArrayList<Integer>> adj) {
    boolean[] visited = new boolean[V];
    boolean[] recStack = new boolean[V];

    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            if (dfsDirected(i, visited, recStack, adj)) {
                return true;
            }
        }
    }
    return false;
}
```

### Summary Table (Best for Exams)

| Graph Type | Method | Condition for Cycle |
|---|---|---|
| **Undirected Graph** | DFS | If `visited[neigh] == true` AND `neigh != parent` |
| **Directed Graph** | DFS with recursion stack | If `recStack[neigh] == true` |

## 5. Most Used Methods for Cycle Detection

| Method | Usage |
|---|---|
| `dfsCycle()` | DFS for undirected graph |
| `isCycleUndirected()` | Check all components |
| `dfsDirected()` | DFS with recursion stack |
| `isCycleDirected()` | Detect cycle in directed graph |
| `recStack[]` | Tracks active recursion path |
| `visited[]` | Marks visited nodes |
| `adj.get(u)` | Access neighbors |

## 6. Small Example Graphs (For Understanding)

### Undirected (has cycle)

```
0 — 1
|   |
3 — 2
```

### Directed (has cycle)

```
0 → 1 → 2
↑       |
|-------|
```

# Minimum Spanning Tree (MST) – Prim's & Kruskal's Algorithm

### What is MST?

A **Minimum Spanning Tree (MST)** of a connected, undirected, weighted graph is:

A tree (no cycles)
Covers all vertices
With **minimum total weight**

Example usage:

- Network design (cables, roads)
- Cluster analysis
- Electrical wiring optimization

### 2. Requirements for MST

1. Graph must be **Connected**
2. Graph must be **Undirected**
3. Graph must be **Weighted**

# Prim's Algorithm

Prim's algorithm builds MST **like Dijkstra**, but instead of shortest path, it picks the **minimum weight edge** that connects a new vertex to the MST.

### Prim's Algorithm Features:

- Works for **dense** graphs.
- Uses **Priority Queue (Min-Heap)**.
- Starts from **any one vertex**.
- Always picks **cheapest available edge**.

### Steps of Prim's Algorithm

1. Choose any starting vertex.
2. Mark it visited.
3. Use a Min-Heap to pick next lowest weight edge.
4. Add new vertex to MST.
5. Repeat until all vertices are included.

### Prim's Java Code (Using PriorityQueue)

```
import java.util.*;

class Pair {
```

```
    int vertex, weight;
    Pair(int v, int w) {
        vertex = v;
        weight = w;
    }
}

class Prims {
    static int primsMST(int V, ArrayList<ArrayList<Pair>> adj) {

        boolean[] visited = new boolean[V];
        PriorityQueue<Pair> pq = new PriorityQueue<>(
                (a, b) -> a.weight - b.weight
        );

        pq.add(new Pair(0, 0)); // start from node 0
        int mstCost = 0;

        while (!pq.isEmpty()) {
            Pair p = pq.poll();
            int node = p.vertex;
            int wt = p.weight;

            if (visited[node]) continue;

            visited[node] = true;
            mstCost += wt;

            for (Pair neigh : adj.get(node)) {
                if (!visited[neigh.vertex]) {
                    pq.add(new Pair(neigh.vertex, neigh.weight));
                }
            }
        }

        return mstCost;
    }
}
```

# Kruskal's Algorithm

Kruskal's algorithm builds MST by adding the **smallest edges first**, and avoids forming **cycles** using **Disjoint Set (Union-Find)**.

## Kruskal's Features:

- Works well on **sparse** graphs.
- Sorts all edges by increasing weight.
- Adds edges if they don't form a cycle.
- Uses **Union-Find** to check cycles.

## Steps of Kruskal's Algorithm

1. Sort all edges by weight.
2. Initialize Disjoint Set for vertices.
3. For each edge (u, v):

- If u and v belong to **different sets**, add edge to MST.
- Union the two sets.

4. Continue until MST has **V-1 edges**.

## Disjoint Set (Union-Find)

Used for cycle detection in Kruskal's.

Main Methods:

| Method | Use |
|---|---|
| `findParent(x)` | Finds leader of a set |
| `union(u, v)` | Combines two sets |
| `rank[]` or `size[]` | Helps balance tree height |

## Java Code for Kruskal's Algorithm

## Step 1: Edge Class

```
class Edge implements Comparable<Edge> {
    int src, dest, weight;

    Edge(int s, int d, int w) {
        src = s;
        dest = d;
        weight = w;
    }

    public int compareTo(Edge e) {
        return this.weight - e.weight;
    }
}
```

## Step 2: Disjoint Set (Union-Find)

```
class DisjointSet {
    int[] parent, rank;

    DisjointSet(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    int findParent(int node) {
        if (parent[node] == node)
            return node;
        return parent[node] = findParent(parent[node]); // path compression
    }

    void union(int u, int v) {
```

```
        u = findParent(u);
        v = findParent(v);

        if (rank[u] < rank[v])
            parent[u] = v;
        else if (rank[u] > rank[v])
            parent[v] = u;
        else {
            parent[v] = u;
            rank[u]++;
        }
    }
}
```

## Step 3: Kruskal Implementation

```
import java.util.*;

class Kruskal {
    static int kruskalMST(int V, ArrayList<Edge> edges) {

        Collections.sort(edges); // sort by weight
        DisjointSet ds = new DisjointSet(V);

        int mstCost = 0;
        ArrayList<Edge> mstEdges = new ArrayList<>();

        for (Edge e : edges) {

            int u = e.src;
            int v = e.dest;

            if (ds.findParent(u) != ds.findParent(v)) {
                mstCost += e.weight;
                mstEdges.add(e);
                ds.union(u, v);
            }
        }

        return mstCost;
    }
}
```

## Prim's vs Kruskal's (Important Table)

| Feature | Prim's | Kruskal's |
|---|---|---|
| Works On | Dense graphs | Sparse graphs |
| Approach | Grow tree from one vertex | Add smallest edges first |
| Data Structure | PriorityQueue | Union-Find |
| Graph Type | Connected | Can work with disconnected components |
| Detect Cycle | Not needed | Union-Find detects |
| Sorting | Not required | Required (sort edges) |

## MST Output Example

For a sample graph:

```
Edges:
0-1 (4)
0-2 (5)
1-2 (1)
1-3 (2)
2-3 (3)
```

# SHORTEST PATH ALGORITHMS: DIJKSTRA & BELLMAN–FORD

### Shortest Path Problem

Given a weighted graph, find the minimum distance from a **source vertex** to **all other vertices**.

Two major algorithms:

| Algorithm | Handles Negative Weight? | Complexity |
|---|---|---|
| **Dijkstra** | No negative edges | O(E log V) |
| **Bellman–Ford** | Yes negative edges (but no negative cycle) | O(V × E) |

# Dijkstra's Algorithm

Dijkstra finds **shortest path in a weighted graph without negative edges**.

### Key Ideas

- Works for **positive weight edges only**.
- Uses **Min-Heap (PriorityQueue)**.
- Greedy: always picks the next **minimum distance node**.
- Similar to Prim's, but compares **distance** instead of **weight**.

### Dijkstra Steps (Easy Explanation)

1. Set all distances = ∞
2. Distance of source = 0
3. Use PQ to pick node with **minimum distance**
4. Relax all edges:

```
if (dist[u] + weight < dist[v])
    update dist[v]
```

7. Repeat until PQ is empty.

### Java Code (Dijkstra using PriorityQueue)

```
import java.util.*;

class Pair {
```

```
    int node, dist;
    Pair(int n, int d) {
        node = n;
        dist = d;
    }
}

class Dijkstra {

    int[] dijkstra(int V, ArrayList<ArrayList<Pair>> adj, int src) {

        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);

        PriorityQueue<Pair> pq = new PriorityQueue<>(
                (a, b) -> a.dist - b.dist
        );

        dist[src] = 0;
        pq.add(new Pair(src, 0));

        while (!pq.isEmpty()) {
            Pair p = pq.poll();
            int node = p.node;
            int d = p.dist;

            for (Pair neigh : adj.get(node)) {
                int next = neigh.node;
                int wt = neigh.dist;

                if (d + wt < dist[next]) {
                    dist[next] = d + wt;
                    pq.add(new Pair(next, dist[next]));
                }
            }
        }

        return dist;
    }
}
```

## Example (Dijkstra Execution)

Graph:

```
0 →1 (4)
0 →2 (1)
2 →1 (2)
1 →3 (1)
2 →3 (5)
```

Shortest path from 0:

```
0 to 0 = 0
0 to 2 = 1
0 to 1 = 3
0 to 3 = 4
```

## Advantages & Limitations

### Pros:

- Fastest for non-negative graphs
- Very efficient with PQ

### Cons:

- Cannot handle **negative weights**
- Fails when negative cycle exists

# Bellman–Ford Algorithm

Bellman–Ford is used when the graph has **negative weights**.

### Important Points

- Works for **negative edges**
- Detects **negative weight cycles**
- Slower than Dijkstra: **O(V × E)**

### Bellman–Ford Logic

Repeat **(V − 1)** times:

```
for every edge (u → v):
    if (dist[u] + weight < dist[v]):
         update dist[v]
```

Reason:
A shortest path can have at most **V−1 edges** in a graph of V nodes.

After V−1 relaxations:

### To detect negative cycle:

Do one more iteration:

```
if any dist[u] + wt < dist[v]  → Negative Cycle
```

### Java Code (Bellman–Ford)

```
import java.util.*;

class Edge {
    int src, dest, weight;
    Edge(int s, int d, int w) {
        src = s;
        dest = d;
        weight = w;
    }
}
```

```
class BellmanFord {

    int[] bellmanFord(int V, ArrayList<Edge> edges, int src) {

        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;

        // Relax edges V-1 times
        for (int i = 1; i < V; i++) {
            for (Edge e : edges) {
                int u = e.src;
                int v = e.dest;
                int wt = e.weight;

                if (dist[u] != Integer.MAX_VALUE && dist[u] + wt < dist[v])
{
                    dist[v] = dist[u] + wt;
                }
            }
        }

        // Check for negative cycle
        for (Edge e : edges) {
            if (dist[e.src] != Integer.MAX_VALUE &&
                dist[e.src] + e.weight < dist[e.dest]) {

                System.out.println("Graph contains Negative Weight
Cycle!");
                return null;
            }
        }

        return dist;
    }
}
```

## Example (Bellman–Ford Execution)

Graph:

```
0 → 1 (1)
1 → 2 (-1)
2 → 3 (-1)
3 → 1 (2)
```

Shortest path from 0:

```
0 to 1 = 1
0 to 2 = 0
0 to 3 = -1
```

If a negative cycle exists, algorithm will detect it.

## Comparison Table

| Feature | Dijkstra | Bellman–Ford |
|---|---|---|
| Handles Negative Weights | No | Yes |
| Detect Negative Cycle | No | Yes |
| Complexity | O(E log V) | O(V × E) |
| Data Structure | PriorityQueue | Simple loops |
| Type | Greedy | Dynamic Programming |
| Usage | Maps, routing | Currency conversion fraud detection |

## Bellman–Ford

- Loop `v-1` times
- Iterate all edges
- Negative cycle check
- Works with **Edge list** instead of adjacency list