Graph Interview Prepration

---

1. Find the Shortest Path Between Two Nodes in a Graph

Problem Statement

Given a graph represented as an adjacency list or adjacency matrix, find the shortest path between two nodes. The graph may be **unweighted** (use BFS) or **weighted** (use Dijkstra's algorithm).

Algorithm (Dijkstra's Algorithm for Weighted Graph)

1. **Initialize a Min-Heap (Priority Queue):** Store (distance, node) pairs.
2. **Set Distances:** Create an array dist to store the shortest distance from the source to every node, initialized as Infinity, except dist[source] = 0.
3. **Process Nodes:**
   - Extract the node with the smallest distance from the priority queue.
   - Update the distances of its neighbors if a shorter path is found.
   - Push the updated (distance, neighbor) back into the queue.
4. **Continue Until All Nodes Are Processed.**
5. **Return the Shortest Distance to the Target Node.**

Why This Algorithm?

We use **Dijkstra's Algorithm** because we are dealing with a **graph with weighted edges** and need the shortest path. BFS works only for unweighted graphs, but Dijkstra efficiently finds the shortest path in O((V + E) log V) time using a priority queue. If negative weights exist, we use **Bellman-Ford Algorithm** instead.

Time & Space Complexity

- **Time Complexity:** O((V + E) log V), where V = number of vertices and E = number of edges.
- **Space Complexity:** O(V) for distance array and O(E) for the adjacency list.

Java Code

```
import java.util.*;


class ShortestPathGraph {
    static class Node {
```

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

    public static int dijkstraShortestPath(int V, List<List<Node>> adj, int source, int target) {
        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n ->
n.weight));
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[source] = 0;
        pq.add(new Node(source, 0));

        while (!pq.isEmpty()) {
            Node curr = pq.poll();
            int u = curr.vertex;
            if (u == target) return dist[u];

            for (Node neighbor : adj.get(u)) {
                int v = neighbor.vertex, weight = neighbor.weight;
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.add(new Node(v, dist[v]));
                }
            }
        }
        return -1;  // Return -1 if no path exists
    }

    public static void main(String[] args) {
        int V = 5;
```

```java
        List<List<Node>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());

        adj.get(0).add(new Node(1, 2));
        adj.get(0).add(new Node(3, 6));
        adj.get(1).add(new Node(2, 3));
        adj.get(3).add(new Node(2, 1));

        int shortestPath = dijkstraShortestPath(V, adj, 0, 2);
        System.out.println("Shortest Path: " + shortestPath);
    }
}
```

Leetcode Link:

- **Dijkstra's Algorithm** (Similar Problem)

---

2. Determine Whether a Graph is Bipartite

Problem Statement

A graph is **bipartite** if we can split its vertices into two independent sets such that no two adjacent nodes belong to the same set. Given a graph represented as an adjacency list, determine whether it is bipartite.

Algorithm (BFS-Based Two-Coloring Approach)

1. **Initialize a Color Array:** Assign -1 to all nodes (uncolored).
2. **Use BFS to Color the Graph:**
    - Start from any uncolored node and color it 0.
    - For each neighbor, assign the opposite color (1 for 0 and vice versa).
    - If a conflict arises (a neighbor has the same color), return false.
3. **Repeat for all Components:** If the graph is disconnected, check each component.
4. **Return true if No Conflicts Found.**

Why This Algorithm?

We use **BFS two-coloring** because it efficiently checks whether a graph is bipartite in **O(V + E) time**. If any cycle of **odd length** exists, the graph is **not bipartite**. We can also use **DFS**, but BFS is preferred for shorter paths.

Time & Space Complexity

- **Time Complexity:** O(V + E), where V = number of vertices, E = number of edges.
- **Space Complexity:** O(V) for the color array and queue.

Java Code

```
import java.util.*;


class BipartiteGraph {
    public static boolean isBipartite(int[][] graph) {
        int V = graph.length;
        int[] color = new int[V];
        Arrays.fill(color, -1); // Uncolored


        for (int i = 0; i < V; i++) {
            if (color[i] == -1) { // Check unvisited components
                Queue<Integer> queue = new LinkedList<>();
                queue.add(i);
                color[i] = 0;


                while (!queue.isEmpty()) {
                    int node = queue.poll();
                    for (int neighbor : graph[node]) {
                        if (color[neighbor] == -1) {
                            color[neighbor] = 1 - color[node];
                            queue.add(neighbor);
                        } else if (color[neighbor] == color[node]) {
                            return false; // Conflict found
                        }
```

```
                }
            }
        }
    }
    return true;
}

public static void main(String[] args) {
    int[][] graph = {
        {1, 3}, {0, 2}, {1, 3}, {0, 2}
    };
    System.out.println("Is Bipartite: " + isBipartite(graph));
}
}
```

Leetcode Link:

- [Is Graph Bipartite?]()

---

3. Detect a Cycle in a Directed Graph

Problem Statement

Given a directed graph, determine whether it contains a cycle. A cycle exists if we can start from a node and reach the same node through a sequence of edges.

Algorithm (DFS-Based Cycle Detection)

1. **Use a Visited Array:** Track the state of each node:
   - 0 → Unvisited
   - 1 → Visiting (part of current recursion stack)
   - 2 → Visited (fully explored)
2. **Perform DFS:**
   - Mark a node as 1 when visiting.
   - If a neighbor is also 1, a **cycle exists**.

    - After exploring all neighbors, mark the node as 2.
3. **Repeat for All Nodes:** If DFS completes without detecting a cycle, return false.

Why This Algorithm?

DFS with recursion stack efficiently detects cycles in directed graphs in **O(V + E) time**. An alternative is **Kahn's Algorithm (Topological Sort)**, but it requires an in-degree array.

Time & Space Complexity

- **Time Complexity:** O(V + E) (DFS traversal).
- **Space Complexity:** O(V) (recursion stack).

Java Code

```
import java.util.*;


class DetectCycleDirected {
   public static boolean hasCycle(int V, List<List<Integer>> adj) {
      int[] state = new int[V];


      for (int i = 0; i < V; i++) {
         if (state[i] == 0 && dfs(i, adj, state)) {
            return true;
         }
      }
      return false;
   }


   private static boolean dfs(int node, List<List<Integer>> adj, int[] state) {
      state[node] = 1; // Mark as visiting
      for (int neighbor : adj.get(node)) {
         if (state[neighbor] == 1 || (state[neighbor] == 0 && dfs(neighbor, adj, state))) {
            return true;
         }
      }
```

```java
        state[node] = 2; // Mark as visited
        return false;
    }

    public static void main(String[] args) {
        int V = 4;
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        adj.get(0).add(1);
        adj.get(1).add(2);
        adj.get(2).add(3);
        adj.get(3).add(1); // Cycle

        System.out.println("Has Cycle: " + hasCycle(V, adj));
    }
}
```

Leetcode Link:

- [Detect Cycle in Directed Graph]()

---

4. Count the Number of Connected Components in an Undirected Graph

Problem Statement

Given an undirected graph, find the number of **connected components** (sets of nodes that are directly or indirectly connected).

Algorithm (DFS Traversal)

1. **Use a Visited Array:** Track nodes that have been visited.
2. **Perform DFS/BFS for Each Component:**
    - If a node is unvisited, increase the count and perform DFS to mark all reachable nodes.
3. **Return the Count of Components.**

Why This Algorithm?

DFS efficiently finds connected components in **O(V + E)** time. We could also use **BFS or Union-Find**, but DFS is simpler for adjacency lists.

Time & Space Complexity

- **Time Complexity:** O(V + E) (DFS traversal).
- **Space Complexity:** O(V) (visited array).

Java Code

```
import java.util.*;


class ConnectedComponents {
    public static int countComponents(int V, List<List<Integer>> adj) {
        boolean[] visited = new boolean[V];
        int count = 0;


        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                dfs(i, adj, visited);
                count++;
            }
        }
        return count;
    }


    private static void dfs(int node, List<List<Integer>> adj, boolean[] visited) {
        visited[node] = true;
        for (int neighbor : adj.get(node)) {
            if (!visited[neighbor]) {
                dfs(neighbor, adj, visited);
            }
        }
    }
```

```java
    public static void main(String[] args) {
        int V = 5;
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        adj.get(0).add(1);
        adj.get(1).add(0);
        adj.get(2).add(3);
        adj.get(3).add(2);
        adj.get(4).add(4); // Isolated node

        System.out.println("Number of Components: " + countComponents(V, adj));
    }
}
```

Leetcode Link:

- Number of Connected Components

---

5. Find the Minimum Spanning Tree of a Graph

Problem Statement

Given a weighted, connected, and undirected graph, find its Minimum Spanning Tree (MST). A Minimum Spanning Tree is a subset of edges that connects all vertices with the minimum possible total edge weight and no cycles.

Algorithm (Kruskal's Algorithm)

1. **Sort** all edges of the graph in increasing order based on their weights.
2. **Initialize** an empty MST and use a Disjoint Set Union (DSU) to manage connected components.
3. **Iterate** through the sorted edges:
    - If adding the edge does not create a cycle, include it in the MST.
    - Use the **Union-Find** data structure to keep track of connected components.

4. Stop when we have exactly V−1V - 1 edges in the MST (where VV is the number of vertices).

Why This Algorithm?

We use **Kruskal’s Algorithm** because:

- It **greedily selects** the smallest edge, ensuring a globally optimal solution.
- It efficiently detects cycles using **Disjoint Set Union (DSU)**.
- Kruskal’s works best for **sparse graphs** (E=O(V)E = O(V)), unlike Prim’s Algorithm, which is better for dense graphs.
- If we need an alternative, **Prim’s Algorithm** would work, but it processes vertices one by one, making it slower for graphs with fewer edges.

Time & Space Complexity

- **Sorting Edges** → O(ElogE)O(E \log E)
- **DSU Operations** → O(Eα(V))O(E \alpha(V)) (almost constant time using path compression)
- **Overall Complexity** → O(ElogV)O(E \log V)
- **Space Complexity** → O(V+E)O(V + E) (for storing edges and DSU)

Java Code (Using Kruskal's Algorithm)

```
import java.util.*;


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


    public Edge(int src, int dest, int weight) {
        this.src = src;
        this.dest = dest;
        this.weight = weight;
    }


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

```
}

class DisjointSet {
    int[] parent, rank;

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

    public int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]); // Path compression
        return parent[x];
    }

    public void union(int x, int y) {
        int rootX = find(x), rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) parent[rootY] = rootX;
            else if (rank[rootX] < rank[rootY]) parent[rootX] = rootY;
            else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
}
```

```java
public class KruskalMST {
    public static List<Edge> kruskalMST(int V, List<Edge> edges) {
        Collections.sort(edges); // Sort edges by weight
        DisjointSet ds = new DisjointSet(V);
        List<Edge> mst = new ArrayList<>();

        for (Edge edge : edges) {
            if (ds.find(edge.src) != ds.find(edge.dest)) { // No cycle
                mst.add(edge);
                ds.union(edge.src, edge.dest);
            }
            if (mst.size() == V - 1) break; // Stop when MST is complete
        }
        return mst;
    }

    public static void main(String[] args) {
        int V = 4;
        List<Edge> edges = Arrays.asList(
            new Edge(0, 1, 10), new Edge(0, 2, 6), new Edge(0, 3, 5),
            new Edge(1, 3, 15), new Edge(2, 3, 4)
        );

        List<Edge> mst = kruskalMST(V, edges);
        System.out.println("Edges in MST:");
        for (Edge edge : mst) {
            System.out.println(edge.src + " - " + edge.dest + " (" + edge.weight + ")");
        }
    }
```

```
}
```

Leetcode Link

- No direct problem for Kruskal's MST, but a similar one:
  [Connecting Cities With Minimum Cost]
- Prim's Algorithm alternative:
  [Minimum Cost to Connect All Points]

---

6. Topological Sorting of a Directed Acyclic Graph (DAG)

Problem Statement

Given a Directed Acyclic Graph (DAG), find a valid **topological ordering** of its vertices, meaning an order in which for every directed edge u→vu \to v, vertex uu comes before vv.

Algorithm (Kahn's Algorithm - BFS Approach)

1. **Compute in-degree** of all vertices.
2. **Push all vertices** with in-degree **0** into a queue.
3. **Process the queue**:
   - Remove a node and add it to the result list.
   - Decrease the in-degree of its neighbors.
   - If a neighbor's in-degree becomes **0**, push it into the queue.
4. **Repeat until all nodes are processed**.

Why This Algorithm?

- We use **Kahn's Algorithm (BFS-based)** because it ensures that we always process nodes with zero dependencies first.
- **DFS-based Topological Sort** is an alternative where we use a stack to store nodes after DFS completion.
- **Time Complexity:** O(V+E)O(V + E) (same for both BFS and DFS approaches).
- **Space Complexity:** O(V)O(V) for storing the result.

Code (Java - BFS Approach)

```java
import java.util.*;


public class TopologicalSort {

    public static List<Integer> topoSort(int V, List<List<Integer>> adj) {
```

```
        int[] inDegree = new int[V];
        for (List<Integer> neighbors : adj)
            for (int neighbor : neighbors) inDegree[neighbor]++;

        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < V; i++)
            if (inDegree[i] == 0) queue.add(i);

        List<Integer> result = new ArrayList<>();
        while (!queue.isEmpty()) {
            int node = queue.poll();
            result.add(node);
            for (int neighbor : adj.get(node))
                if (--inDegree[neighbor] == 0) queue.add(neighbor);
        }
        return result;
    }
}
```

Leetcode Link

[Course Schedule II (Leetcode 210)]

---

7. Shortest Path Between All Pairs of Nodes in a Weighted Graph (Floyd-Warshall Algorithm)

Problem Statement

Given a weighted graph, find the shortest paths between all pairs of nodes.

Algorithm (Floyd-Warshall Algorithm)

1. Create a **distance matrix**, initialized with graph weights.
2. Iterate over **each node kk as an intermediate node**:
   - Update dist[i][j]=min⁡(dist[i][j],dist[i][k]+dist[k][j])dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j]).

3. If dist[i][i]dist[i][i] is negative, there's a **negative cycle**.

Why This Algorithm?

- **Floyd-Warshall** is best for small graphs as it runs in **O(V3)O(V^3)**.
- For sparse graphs, **Dijkstra's Algorithm** is better (O(V+ElogV)O(V + E \log V)).
- **Space Complexity**: O(V2)O(V^2) for the distance matrix.

Code (Java - Floyd-Warshall)

```java
public class FloydWarshall {
    static final int INF = 1000000;

    public static void floydWarshall(int[][] graph) {
        int V = graph.length;
        int[][] dist = new int[V][V];

        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++) dist[i][j] = graph[i][j];

        for (int k = 0; k < V; k++)
            for (int i = 0; i < V; i++)
                for (int j = 0; j < V; j++)
                    if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
    }
}
```

Leetcode Link

[Find the City With the Smallest Number of Neighbors (Leetcode 1334)]

---

8. Clone a Directed Graph

Problem Statement

Given a directed graph, create a **deep copy** of the graph.

Algorithm (DFS Approach)

1. Use a **hash map** to store visited nodes.
2. Perform **DFS traversal**:
    - Copy each node and its neighbors recursively.

Why This Algorithm?

- DFS is used because we need to explore and copy all neighbors before returning the node.
- **Time Complexity:** O(V+E)O(V + E)
- **Space Complexity:** O(V)O(V) for storing nodes.

Code (Java - DFS Approach)

```
import java.util.*;


class Node {

   int val;

   List<Node> neighbors;

   public Node(int val) {

      this.val = val;

      neighbors = new ArrayList<>();

   }

}


public class CloneGraph {

   private Map<Node, Node> map = new HashMap<>();


   public Node cloneGraph(Node node) {

      if (node == null) return null;

      if (map.containsKey(node)) return map.get(node);


      Node clone = new Node(node.val);

      map.put(node, clone);
```

```
        for (Node neighbor : node.neighbors)
            clone.neighbors.add(cloneGraph(neighbor));

        return clone;
    }
}
```

Leetcode Link

[Clone Graph (Leetcode 133)]()

---

9. Check if There is a Path Between Two Nodes in a Graph

Problem Statement

Given a graph and two nodes, check if there exists a path between them.

Algorithm (DFS or BFS)

1. Use **DFS** (or BFS) to traverse from the source node.
2. If the destination node is reached, return **true**.
3. If all paths are explored and the destination is not found, return **false**.

Why This Algorithm?

- **DFS** is used when we need a recursive approach and a **single path search**.
- **BFS** is better if we need the **shortest path** in an **unweighted graph**.

Code (Java - DFS Approach)

```
import java.util.*;

public class GraphPath {
    public static boolean hasPath(Map<Integer, List<Integer>> graph, int src, int dest,
Set<Integer> visited) {
        if (src == dest) return true;
        if (visited.contains(src)) return false;
        visited.add(src);
```

```
        for (int neighbor : graph.getOrDefault(src, new ArrayList<>()))
            if (hasPath(graph, neighbor, dest, visited)) return true;

        return false;
    }
}
```

Leetcode Link

Find If Path Exists in Graph (Leetcode 1971)

---

10. Find the Diameter of a Tree or Graph

Problem Statement

Find the **longest path** between any two nodes in a tree or graph.

Algorithm (Two BFS or DFS Traversals)

1. **First BFS/DFS** from any node to find the farthest node XX.
2. **Second BFS/DFS** from XX to find the farthest node YY, which gives the diameter.

Why This Algorithm?

- Works in O(V+E)O(V + E) time using **two DFS or BFS traversals**.
- **Greedy approach** ensures the longest path is found efficiently.

Code (Java - DFS Approach)

```
public class TreeDiameter {
    static int maxDiameter = 0;
    public static int dfs(Map<Integer, List<Integer>> graph, int node, Set<Integer> visited) {
        visited.add(node);
        int max1 = 0, max2 = 0;
        for (int neighbor : graph.get(node)) {
            if (!visited.contains(neighbor)) {
                int depth = dfs(graph, neighbor, visited);
                if (depth > max1) { max2 = max1; max1 = depth; }
```

```
            else if (depth > max2) max2 = depth;
        }
    }
    maxDiameter = Math.max(maxDiameter, max1 + max2);
    return max1 + 1;
  }
}
```

Leetcode Link

[Diameter of Binary Tree (Leetcode 543)]

---

11. Find the Strongly Connected Components (SCCs) of a Directed Graph (Kosaraju's Algorithm)

Problem Statement

Find all **Strongly Connected Components (SCCs)** in a directed graph, where each SCC is a maximal set of nodes such that every node can reach every other node in the component.

Algorithm (Kosaraju's Algorithm - Two DFS Traversals)

1. **Perform DFS** and store the finish order in a stack.
2. **Transpose the graph** (reverse all edges).
3. **Perform DFS** in the stack order on the transposed graph.
4. **Each DFS traversal identifies an SCC**.

Why This Algorithm?
Kosaraju's Algorithm efficiently finds SCCs in a directed graph using two depth-first search (DFS) traversals. The first DFS stores nodes in a finish-time order, and the second DFS on the transposed graph extracts SCCs. This approach ensures linear time complexity O(V+E)O(V + E)O(V+E), making it optimal for large graphs.

Code (Java - Kosaraju's Algorithm)

```
import java.util.*;

public class StronglyConnectedComponents {
  public static void findSCCs(int V, List<List<Integer>> adj) {
```

```
        Stack<Integer> stack = new Stack<>();
        boolean[] visited = new boolean[V];

        for (int i = 0; i < V; i++)
            if (!visited[i]) dfs(adj, i, visited, stack);

        List<List<Integer>> transposedGraph = transposeGraph(V, adj);
        Arrays.fill(visited, false);

        while (!stack.isEmpty()) {
            int node = stack.pop();
            if (!visited[node]) {
                List<Integer> scc = new ArrayList<>();
                dfs(transposedGraph, node, visited, scc);
                System.out.println("SCC: " + scc);
            }
        }
    }

    private static void dfs(List<List<Integer>> adj, int node, boolean[] visited, Stack<Integer>
stack) {
        visited[node] = true;
        for (int neighbor : adj.get(node))
            if (!visited[neighbor]) dfs(adj, neighbor, visited, stack);
        stack.push(node);
    }

    private static List<List<Integer>> transposeGraph(int V, List<List<Integer>> adj) {
        List<List<Integer>> transposed = new ArrayList<>();
        for (int i = 0; i < V; i++) transposed.add(new ArrayList<>());
```

```
        for (int i = 0; i < V; i++)
            for (int neighbor : adj.get(i)) transposed.get(neighbor).add(i);
        return transposed;
    }
}
```

Leetcode Link

[Strongly Connected Components (Leetcode - Hard)]

---

12. Convert a Graph to Its Complement

Problem Statement

The **complement of a graph** is a graph with the same vertices but contains all edges **not** present in the original graph.

Algorithm

1. **Create an adjacency matrix** of the original graph.
2. **Invert the adjacency matrix** to obtain the complement graph.

Why This Algorithm?
Using an **adjacency matrix** allows easy inversion of edges in O(V2)O(V^2)O(V2) time. If using an adjacency list, an **efficient set-based lookup** prevents redundant edges, improving performance. The approach guarantees correctness while handling **sparse and dense graphs efficiently**.

Code (Java - Adjacency Matrix Approach)

```
public class GraphComplement {
    public static int[][] complementGraph(int[][] graph) {
        int V = graph.length;
        int[][] complement = new int[V][V];

        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (i != j) complement[i][j] = (graph[i][j] == 1) ? 0 : 1;
```

```
            }
        }
        return complement;
    }
}
```

Leetcode Link

Graph Complement Problem (Custom Implementation)

---

13. Find the Maximum Flow in a Network Flow Graph (Ford-Fulkerson Algorithm)

Problem Statement

Find the **maximum flow** from the **source (s)** to the **sink (t)** in a flow network.

Algorithm (Edmonds-Karp Implementation - BFS)

1. Use **BFS** to find an **augmenting path**.
2. **Find the bottleneck capacity** of that path.
3. **Update the residual graph**.
4. Repeat until no more augmenting paths exist.

Why This Algorithm?
Ford-Fulkerson finds the **maximum flow** by augmenting flow paths iteratively, while the **Edmonds-Karp variation** ensures O(VE2)O(VE^2)O(VE2) worst-case time complexity using **BFS for shortest augmenting paths**. This method efficiently handles real-world network constraints like **pipeline optimization and traffic routing**.

Code (Java - BFS Approach)

```
import java.util.*;

public class MaxFlow {
    static final int INF = Integer.MAX_VALUE;

    public static int fordFulkerson(int[][] capacity, int s, int t) {
```

```java
    int V = capacity.length;
    int[][] residual = new int[V][V];
    for (int i = 0; i < V; i++)
        System.arraycopy(capacity[i], 0, residual[i], 0, V);

    int maxFlow = 0;
    int[] parent = new int[V];

    while (bfs(residual, s, t, parent)) {
        int flow = INF;
        for (int v = t; v != s; v = parent[v])
            flow = Math.min(flow, residual[parent[v]][v]);

        for (int v = t; v != s; v = parent[v]) {
            residual[parent[v]][v] -= flow;
            residual[v][parent[v]] += flow;
        }
        maxFlow += flow;
    }
    return maxFlow;
}

private static boolean bfs(int[][] residual, int s, int t, int[] parent) {
    Arrays.fill(parent, -1);
    Queue<Integer> queue = new LinkedList<>();
    queue.add(s);
    parent[s] = s;

    while (!queue.isEmpty()) {
```

```
            int u = queue.poll();
            for (int v = 0; v < residual.length; v++) {
                if (parent[v] == -1 && residual[u][v] > 0) {
                    parent[v] = u;
                    queue.add(v);
                    if (v == t) return true;
                }
            }
        }
        return false;
    }
}
```

Leetcode Link

[Maximum Flow Problem]

14. Implement Dijkstra's Algorithm

Why This Algorithm?
Dijkstra's Algorithm is optimal for single-source shortest path problems in weighted graphs. Using a min-heap priority queue, it efficiently finds the shortest path in O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), making it well-suited for navigation systems and network routing.

Code:

```
import java.util.*;


public class Dijkstra {
    public static int[] dijkstra(int V, List<List<int[]>> adj, int src) {
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;
        pq.offer(new int[]{src, 0});
```

```
        while (!pq.isEmpty()) {
            int[] curr = pq.poll();
            int u = curr[0], d = curr[1];


            if (d > dist[u]) continue;


            for (int[] neighbor : adj.get(u)) {
                int v = neighbor[0], weight = neighbor[1];
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.offer(new int[]{v, dist[v]});
                }
            }
        }
        return dist;
    }
}
```

---

15. Implement Kruskal's Algorithm

Why This Algorithm?
Kruskal's Algorithm is efficient for **finding the Minimum Spanning Tree (MST)** by **sorting edges and using Union-Find for cycle detection**. With O(ElogE)O(E \log E)O(ElogE) complexity, it performs well in **sparse graphs**, making it ideal for **network design and clustering**.

Code:

```
import java.util.*;


public class Kruskal {
    public static int kruskal(int V, int[][] edges) {
```

```
        Arrays.sort(edges, Comparator.comparingInt(a -> a[2]));
        UnionFind uf = new UnionFind(V);
        int minCost = 0;

        for (int[] edge : edges) {
            if (uf.union(edge[0], edge[1])) minCost += edge[2];
        }
        return minCost;
    }
}
```

---

16. Implement Breadth-First Search (BFS)

Why This Algorithm?
BFS is optimal for **finding the shortest path in an unweighted graph**. Its **level-order traversal** ensures **all nodes at depth ddd are visited before depth d+1d+1d+1**. With O(V+E)O(V + E)O(V+E) complexity, it's widely used in **maze solving, AI, and network traversal**.

Code:

```
public static void bfs(Map<Integer, List<Integer>> graph, int start) {
    Queue<Integer> queue = new LinkedList<>();
    Set<Integer> visited = new HashSet<>();
    queue.add(start);
    visited.add(start);

    while (!queue.isEmpty()) {
        int node = queue.poll();
        System.out.print(node + " ");
        for (int neighbor : graph.get(node)) {
            if (!visited.contains(neighbor)) {
                queue.add(neighbor);
                visited.add(neighbor);
```

```
            }
        }
    }
}
```