

■ Graphs

Complete DSA Notes — Beginner to Advanced

12 Chapters • Java Code • Visual Diagrams • Interview Ready

01	BFS of Graph	EASY
02	DFS of Graph	EASY
03	Number of Provinces	MEDIUM
04	Number of Islands	MEDIUM
05	Flood Fill	EASY
06	Rotten Oranges	MEDIUM
07	Detect Cycle — Undirected Graph	MEDIUM
08	Detect Cycle — Directed Graph	MEDIUM
09	Topological Sort (DFS)	MEDIUM
10	Topological Sort (Kahn's BFS)	MEDIUM
11	Dijkstra's Algorithm	MEDIUM
12	Bellman Ford Algorithm	MEDIUM

BFS of Graph

● EASY

■ Introduction

Imagine you drop a stone in a pond. Ripples spread outward layer by layer — first the adjacent ring, then the next ring, then the next. BFS (Breadth-First Search) explores a graph exactly like this: level by level, using a Queue.

Why it matters: BFS is the foundation of shortest path algorithms, level-order traversal, finding connected components, and multi-source spreading problems.

■ Core Intuition

BFS Properties

- Uses a QUEUE (FIFO) — process nodes in the order they were discovered.
- Visits all neighbors at distance 1, then distance 2, then distance 3...
- Guarantees SHORTEST PATH (in terms of edge count) in unweighted graphs.
- Needs a visited[] array to avoid re-processing nodes (prevent infinite loops).

■■ BFS Algorithm Steps

Step 1: Add starting node to queue, mark it visited.

Step 2: While queue is not empty: dequeue front node, add to result.

Step 3: For each unvisited neighbor of current node: mark visited, enqueue.

Step 4: Repeat until queue is empty. All reachable nodes are processed.

■ Visual — BFS Traversal

BFS Trace

Graph (Adjacency List): BFS from node 0:

0 -- 1 Queue: [0] Visited: {0}

0 -- 2 Deq 0 → enq 1,2 Queue:[1,2]

1 -- 3 Deq 1 → enq 3 Queue:[2,3]

1 -- 4 Deq 2 → enq 5 Queue:[3,5] (4 not neighbor of 2)

2 -- 5 Deq 3 → no new Queue:[5]

Deq 4 → no new Queue:[]

BFS Order: 0 → 1 → 2 → 3 → 4 → 5 (Level by Level)

Level 0: [0]

Level 1: [1, 2]

Level 2: [3, 4, 5]

■ Java Implementation

Java

```
import java.util.*;

class Solution {
    // BFS traversal – returns list of nodes in BFS order

    public ArrayList<Integer> bfsOfGraph(int V, ArrayList<ArrayList<Integer>> adj) {
        ArrayList<Integer> result = new ArrayList<>();

        boolean[] visited = new boolean[V];
        Queue<Integer> queue = new LinkedList<>();

        // Start from node 0
        queue.add(0);
        visited[0] = true;

        while (!queue.isEmpty()) {
            int node = queue.poll(); // Dequeue
            result.add(node); // Process

            // Enqueue all unvisited neighbors
            for (int neighbor : adj.get(node)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }

        return result;
    }

    // For disconnected graphs – BFS from every unvisited node
    for (int i = 0; i < V; i++) {
        if (!visited[i]) bfs(i, adj, visited, result);
    }
}
```

■■ Time & Space Complexity

Case	Time	Space	Why
Time	$O(V + E)$	$O(V)$	Visit each vertex once (V) + traverse each edge once (E)
Space	$O(V)$	$O(V)$	Queue + visited array, both at most V nodes

Common Mistakes

Avoid These

- Not marking visited BEFORE enqueueing — causes same node to be added multiple times.
- Forgetting disconnected components — always loop over all nodes for full traversal.
- Using Stack instead of Queue — that gives DFS, not BFS.
- Not handling edge case: $V=0$ or empty graph.

Practice Problems

Case	Time	Space	Why
Easy	GFG	BFS of Graph	Direct implementation
Easy	LC 102	Binary Tree Level Order	BFS on tree
Medium	LC 994	Rotten Oranges	Multi-source BFS
Medium	LC 127	Word Ladder	BFS on implicit graph

Mental Model

Remember Forever

- "BFS is a pandemic spreading level by level. Patient 0 infects neighbors on Day 1, they infect their neighbors on Day 2... Queue tracks who to process next, visited[] tracks who is already infected."

■ Introduction

■ Core Intuition

- Uses STACK (implicit via recursion, or explicit Stack).
- Goes deep into one branch completely before exploring siblings.
- Good for: cycle detection, topological sort, connected components, path finding.
- NOT guaranteed to find shortest path (unlike BFS).

■ Visual — DFS Traversal

■ Java Implementation

```
// ■■ Recursive DFS ████████████████████████████████████████
```

```
class Solution {  
  
    public ArrayList<Integer> dfsOfGraph(int V, ArrayList<ArrayList<Integer>> adj) {  
  
        ArrayList<Integer> result = new ArrayList<>();  
  
        boolean[] visited = new boolean[V];  
  
        dfs(0, adj, visited, result);  

```

■ Time & Space Complexity

Case	Time	Space	Why
------	------	-------	-----

Time	$O(V + E)$	$O(V)$	Each vertex and edge processed once
Rec. Space	$O(V)$	$O(V)$	Call stack depth = longest DFS path (up to V)

Common Mistakes

Avoid These

- Not marking visited — causes infinite recursion/loop on cyclic graphs.
- Confusing DFS order with inorder/preorder tree traversal — graph DFS is different.
- Stack overflow on very deep graphs — iterative version is safer.

Practice Problems

Case	Time	Space	Why
Easy	GFG	DFS of Graph	Direct implementation
Medium	LC 200	Number of Islands	DFS on grid
Medium	LC 130	Surrounded Regions	DFS from boundary
Hard	LC 329	Longest Path in Matrix	DFS + memoization

Mental Model

Remember Forever

- "DFS is an explorer who always takes the first available unexplored path, keeps going until a dead end, then backtracks. BFS = ripple in water. DFS = explorer in a maze."

Number of Provinces

● MEDIUM

■ Introduction

Given an adjacency MATRIX (not list), find how many connected components exist. Cities in the same connected component form one "province." This is a classic connected components problem — solved with DFS/BFS or Union-Find.

■ Core Intuition

Key Insight

- Input is an $n \times n$ adjacency matrix. `isConnected[i][j]=1` means city i and j are directly connected.
- A province = a connected component of cities.
- Algorithm: iterate over all cities. For each unvisited city, do DFS/BFS to mark all reachable cities as visited. Increment province count.
- Each DFS/BFS call discovers one complete province.

■ Java Implementation

Java

```
class Solution {
    public int findCircleNum(int[][] isConnected) {
        int n = isConnected.length;
        boolean[] visited = new boolean[n];
        int provinces = 0;

        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                dfs(isConnected, visited, i);
                provinces++; // Found a new province
            }
        }

        return provinces;
    }

    private void dfs(int[][] grid, boolean[] visited, int city) {
        visited[city] = true;

        for (int other = 0; other < grid.length; other++) {
```

Time & Space Complexity

Case	Time	Space	Why
DFS	$O(n^2)$	$O(n)$	Check every cell of $n \times n$ matrix; visited array $O(n)$
Union-Find	$O(n^2)$	$O(n)$	Same matrix scan; union operations nearly $O(1)$ each

Common Mistakes

Watch Out

- Adjacency MATRIX input (not list) — iterate j from 0 to n for neighbors.
- isConnected[i][i] = 1 (self-loop) — skip when i == j or handle gracefully.
- Forgetting to check !visited before DFS — avoid re-entering visited cities.

■ Practice Problems

Case	Time	Space	Why
Medium	LC 547	Number of Provinces	Direct application
Medium	LC 200	Number of Islands	Grid version
Medium	LC 684	Redundant Connection	Union-Find cycle detection
Hard	LC 827	Making a Large Island	Connected components + merge

■ Mental Model

Remember Forever

- "Each unvisited node is the start of a new island (province). DFS from it marks the entire island. Count how many times you start a new DFS = number of provinces."

Number of Islands

● MEDIUM

■ Introduction

Given a 2D grid of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and formed by connecting adjacent land cells horizontally or vertically. This is connected components on a GRID.

■ Core Intuition

Grid as Graph

- Each cell is a NODE. Adjacent (up/down/left/right) land cells are EDGES.
- An island = a connected component of '1' cells.
- Algorithm: scan grid. On finding unvisited '1', do DFS/BFS to sink the entire island (mark as '0' or visited). Increment count.
- Four-directional movement: up(-1,0), down(+1,0), left(0,-1), right(0,+1).

■ Visual

Islands Visual

Grid: After DFS from (0,0):

1 1 0 0 0 0 0 0 ← Island 1 sunk

1 1 0 0 0 0 0 0

0 0 1 0 0 0 0 0 ← Island 2 found at (2,2), sunk

0 0 0 1 0 0 0 0 ← Island 3 found at (3,3), sunk

Total islands = 3

■ Java Implementation

Java

```
class Solution {
    int[][] dirs = {{-1,0},{1,0},{0,-1},{0,1}}; // up,down,left,right

    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;
        int rows = grid.length, cols = grid[0].length;
        int count = 0;

        for (int r = 0; r < rows; r++) {
```

```

for (int c = 0; c < cols; c++) {
    if (grid[r][c] == '1') {
        dfs(grid, r, c); // Sink the island
        count++;
    }
}

return count;
}

private void dfs(char[][] grid, int r, int c) {
    // Out of bounds or water — stop
    if (r < 0 || r >= grid.length || c < 0 || c >= grid[0].length
        || grid[r][c] != '1') return;

    grid[r][c] = '0'; // Mark as visited (sink land)

    for (int[] d : dirs)
        dfs(grid, r + d[0], c + d[1]);
}

```

■ Time & Space Complexity

Case	Time	Space	Why
Time	$O(m \cdot n)$	$O(m \cdot n)$	Each cell visited at most once; DFS stack $O(m \cdot n)$ worst case
Space	$O(m \cdot n)$	$O(m \cdot n)$	DFS recursion stack, BFS queue — both at most $m \cdot n$ deep

■ Common Mistakes

Avoid These

- Checking diagonal neighbors — islands connect only 4-directionally (not diagonal).
- Not resetting grid — if you modify grid to mark visited, be careful if grid is reused.
- Missing bounds check before accessing `grid[r][c]` — always validate bounds first.

■ Practice Problems

Case	Time	Space	Why
Medium	LC 200	Number of Islands	Direct application
Medium	LC 695	Max Area of Island	DFS + area count
Medium	LC 130	Surrounded Regions	DFS from boundary
Hard	LC 305	Number of Islands II	Union-Find incremental

■ Mental Model

Remember Forever

- "You're a flood that starts at each unvisited land cell. Flood fills the entire island (marks all connected land as water). Each time you start a new flood = new island."

Flood Fill

● EASY

■ Introduction

Flood Fill is the algorithm behind the "paint bucket" tool in image editors. Given a pixel position and a new color, change that pixel and all connected pixels of the same original color to the new color. It's DFS/BFS on a grid with a color-matching condition.

■ Core Intuition

Key Steps

- Start at (sr, sc). Note the original color = image[sr][sc].
- DFS/BFS in 4 directions. For each cell: only proceed if it has the ORIGINAL color.
- Change color as you visit — prevents re-visiting.
- Edge case: if new color == original color, return immediately (no change needed).

■ Java Implementation

Java

```
class Solution {  
    public int[][] floodFill(int[][] image, int sr, int sc, int color) {  
        int origColor = image[sr][sc];  
        if (origColor == color) return image; // Nothing to change  
        dfs(image, sr, sc, origColor, color);  
        return image;  
    }  
  
    private void dfs(int[][] img, int r, int c, int orig, int newColor) {  
        if (r < 0 || r >= img.length || c < 0 || c >= img[0].length) return;  
        if (img[r][c] != orig) return; // Not same original color  
  
        img[r][c] = newColor; // Fill  
        dfs(img, r-1, c, orig, newColor); // Up  
        dfs(img, r+1, c, orig, newColor); // Down  
        dfs(img, r, c-1, orig, newColor); // Left  
        dfs(img, r, c+1, orig, newColor); // Right  
    }  
}
```

```
}
```

Time & Space Complexity

Case	Time	Space	Why
Time	$O(m \cdot n)$	$O(m \cdot n)$	Each cell visited at most once
Space	$O(m \cdot n)$	$O(m \cdot n)$	Recursion depth in worst case fills entire grid

Common Mistakes

Avoid These

- Forgetting the "new color == original color" early return — causes infinite recursion.
- Changing color AFTER recursive calls — always change BEFORE to mark as visited.
- Using `!= newColor` as condition instead of `== orig` — wrong logic.

Practice Problems

Case	Time	Space	Why
Easy	LC 733	Flood Fill	Direct application
Medium	LC 542	01 Matrix	BFS flood from all 0s
Medium	LC 286	Walls and Gates	Multi-source BFS fill

Mental Model

Remember Forever

- "Paint bucket: click a pixel, color spreads to all connected same-colored pixels. The check is: same original color AND not already painted."

Rotten Oranges

● MEDIUM

■ Introduction

Every minute, fresh oranges adjacent to rotten ones become rotten. Find the minimum time to rot all oranges, or return -1 if impossible. This is a classic **Multi-Source BFS** problem — multiple sources spread simultaneously.

■ Core Intuition

Multi-Source BFS

- Start BFS from ALL rotten oranges simultaneously (add all to queue at time 0).
- Each BFS level = 1 minute of rotting. Level count = time taken.
- After BFS, check if any fresh orange remains → return -1 if yes.
- Key: multi-source BFS ensures simultaneous spreading from all initial rotten oranges.

■ Visual

Rotten Oranges Trace

Initial: Min 1: Min 2:

2 1 1 2 2 1 2 2 2

1 1 0 → 2 1 0 → 2 2 0

0 1 1 0 1 1 0 2 2

Queue at start: [(0,0), (1,0)] – both rotten oranges

BFS level 1: rot (0,1),(1,1) → level 2: rot (0,2),(2,1),(2,2)

Answer: 2 minutes

■ Java Implementation

Java

```
class Solution {
    public int orangesRotting(int[][] grid) {
        int rows = grid.length, cols = grid[0].length;
        Queue<int[]> queue = new LinkedList<>();
        int fresh = 0;

        // Step 1: Add all rotten oranges to queue, count fresh
        for (int r=0; r<rows; r++)
```

```

for (int c=0; c<cols; c++) {
    if (grid[r][c] == 2) queue.add(new int[]{r,c});
    if (grid[r][c] == 1) fresh++;
}

if (fresh == 0) return 0; // No fresh oranges

int[][] dirs = {{-1,0},{1,0},{0,-1},{0,1}};
int minutes = 0;

// Step 2: BFS – spread rotting level by level
while (!queue.isEmpty() && fresh > 0) {
    minutes++;

    int size = queue.size();
    for (int i=0; i<size; i++) {
        int[] cell = queue.poll();
        for (int[] d : dirs) {
            int nr = cell[0]+d[0], nc = cell[1]+d[1];

            if (nr>=0 && nr<rows && nc>=0 && nc<cols && grid[nr][nc]==1) {
                grid[nr][nc] = 2;
                queue.add(new int[]{nr,nc});
                fresh--;
            }
        }
    }
}

return fresh == 0 ? minutes : -1;
}

```

Time & Space Complexity

Case	Time	Space	Why
Time	$O(m*n)$	$O(m*n)$	Every cell enqueued at most once
Space	$O(m*n)$	$O(m*n)$	Queue can hold all cells in worst case

Common Mistakes

Avoid These

- Single-source BFS — must start from ALL rotten oranges simultaneously.
- Not counting minutes correctly — increment once per BFS level, not per cell.
- Forgetting to check if fresh > 0 after BFS — isolated fresh oranges cant be reached.
- Modifying grid without tracking freshCount — count fresh separately.

■ Practice Problems

Case	Time	Space	Why
Medium	LC 994	Rotten Oranges	Direct application
Medium	LC 542	01 Matrix	Multi-source BFS from 0s
Medium	LC 1162	Far from Land	Multi-source BFS from land
Hard	LC 317	Shortest Distance from All Buildings	Multi-source BFS

■ Mental Model

Remember Forever

- "Multi-source BFS = multiple epidemics starting at time 0. Use BFS levels to count time. After BFS, isolated survivors = -1."


```

}

return false;
}

private boolean hasCycleBFS(int src, ArrayList<ArrayList<Integer>> adj, boolean[] vis) {
    Queue<int[]> q = new LinkedList<>(); // [node, parent]
    q.add(new int[]{src, -1});
    vis[src] = true;

    while (!q.isEmpty()) {
        int[] curr = q.poll();
        int node = curr[0], parent = curr[1];

        for (int neighbor : adj.get(node)) {
            if (!vis[neighbor]) {
                vis[neighbor] = true;
                q.add(new int[]{neighbor, node});
            } else if (neighbor != parent) {
                return true; // Visited + not parent = CYCLE
            }
        }
    }

    return false;
}
}

```

■ ■ Time & Space Complexity

Case	Time	Space	Why
Time	$O(V + E)$	$O(V)$	Each vertex and edge processed once
Space	$O(V)$	$O(V)$	Queue and visited array both $O(V)$

■ ■ Common Mistakes

Avoid These

- Not tracking parent — without parent, the reverse edge falsely triggers cycle detection.
- Using visited alone — in undirected, ALL edges look like they revisit the other end.
- For DFS version: pass parent as parameter, not a global variable.

■ Practice Problems

Case	Time	Space	Why
Medium	GFG	Detect Cycle Undirected	Direct application
Medium	LC 684	Redundant Connection	Find the cycle-creating edge
Medium	LC 261	Graph Valid Tree	Cycle + connectivity check

Detect Cycle — Directed Graph

● MEDIUM

■ Introduction

Cycle detection in DIRECTED graphs is trickier. A visited node in a different DFS branch is NOT a cycle. A back-edge (to an ancestor in the CURRENT DFS path) IS a cycle. We need to track the current recursion stack separately.

■ Core Intuition

Two Arrays Needed

- `visited[]`: marks nodes that have been completely processed (any DFS path).
- `inStack[]`: marks nodes currently in the ACTIVE DFS recursion stack.
- Cycle detected when we reach a neighbor that is in `inStack[]` (ancestor in current path).
- After returning from a node's DFS, remove it from `inStack[]`.

■ Visual

Directed Cycle Visual

Directed Graph: DFS from 0:

0 → 1 → 2 → 3 visit 0, inStack={0}

↑ | visit 1, inStack={0,1}

+-----+ visit 2, inStack={0,1,2}

(3 points back to 1) visit 3, inStack={0,1,2,3}

neighbor of 3 is 1 → 1 is in inStack → CYCLE!

Cross-edge (NOT a cycle):

0→1→3 1 visited from 0's DFS, 3 visited later

0→2→3 (3 reachable from both paths – NOT a cycle)

■ Java Implementation

Java

```
class Solution {
    public boolean isCyclic(int V, ArrayList<ArrayList<Integer>> adj) {
        boolean[] visited = new boolean[V];
        boolean[] inStack = new boolean[V];

        for (int i=0; i<V; i++)
```

```

if (!visited[i] && dfs(i, adj, visited, inStack))
    return true;
return false;
}

private boolean dfs(int node, ArrayList<ArrayList<Integer>> adj,
    boolean[] visited, boolean[] inStack) {
    visited[node] = true;
    inStack[node] = true; // Add to current DFS path

    for (int neighbor : adj.get(node)) {
        if (!visited[neighbor]) {
            if (dfs(neighbor, adj, visited, inStack)) return true;
        } else if (inStack[neighbor]) {
            return true; // Back edge = CYCLE
        }
    }

    inStack[node] = false; // Remove from DFS path on backtrack
    return false;
}
}

```

Time & Space Complexity

Case	Time	Space	Why
Time	$O(V + E)$	$O(V)$	Each vertex and edge once; two $O(V)$ arrays
Space	$O(V)$	$O(V)$	visited + inStack arrays; recursion stack $O(V)$

Common Mistakes

Watch Out

- Confusing with undirected cycle detection — directed needs inStack[], undirected needs parent.
- Forgetting inStack[node]=false on backtrack — causes false cycle detection.
- Using visited[] instead of inStack[] — will falsely flag cross-edges as cycles.

Practice Problems

Case	Time	Space	Why
------	------	-------	-----

Medium	LC 207	Course Schedule	Detect cycle in directed graph
Medium	LC 210	Course Schedule II	Topological sort (no cycle)
Hard	LC 802	Find Eventual Safe States	Reverse cycle detection

Topological Sort (DFS)

● MEDIUM

■ Introduction

Topological Sort is the ordering of tasks where dependencies come first. Example: to compile code, first compile dependencies. In a DAG (Directed Acyclic Graph), topological order ensures every node comes before its successors.

■ Core Intuition — DFS Post-order

Key Insight

- DFS and push nodes to a STACK after all their descendants are visited.
- A node is pushed to stack only after ALL its dependencies are processed.
- Finally, pop the stack to get topological order (reverse post-order).
- Only valid for DAGs — cycles make topological sort impossible.

■ Visual

Topological DFS Trace

DAG: $5 \rightarrow 2$, $5 \rightarrow 0$, $4 \rightarrow 0$, $4 \rightarrow 1$, $2 \rightarrow 3$, $3 \rightarrow 1$

DFS from 5:

visit 5 $\rightarrow$ visit 2 $\rightarrow$ visit 3 $\rightarrow$ visit 1 (no unvisited neighbors)

push 1 $\rightarrow$ stack:[1]

back to 3: push 3 $\rightarrow$ stack:[1,3]

back to 2: push 2 $\rightarrow$ stack:[1,3,2]

back to 5: visit 0 $\rightarrow$ push 0 $\rightarrow$ stack:[1,3,2,0]

push 5 $\rightarrow$ stack:[1,3,2,0,5]

DFS from 4: visit 4 $\rightarrow$ push 4 $\rightarrow$ stack:[1,3,2,0,5,4]

Pop stack: 4, 5, 0, 2, 3, 1 $\leftarrow$ topological order

■ Java Implementation

Java

```
class Solution {
    public int[] topoSort(int V, ArrayList<ArrayList<Integer>> adj) {
        boolean[] visited = new boolean[V];
        Deque<Integer> stack = new ArrayDeque<>();
```

```

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

// Pop stack to get topological order
int[] result = new int[V];
for (int i=0; i<V; i++) result[i] = stack.pop();
return result;
}

private void dfs(int node, ArrayList<ArrayList<Integer>> adj,
    boolean[] visited, Deque<Integer> stack) {
    visited[node] = true;
    for (int neighbor : adj.get(node))
        if (!visited[neighbor]) dfs(neighbor, adj, visited, stack);
    stack.push(node); // Push AFTER all descendants are processed
}
}

```

■ Time & Space Complexity

Case	Time	Space	Why
Time	$O(V + E)$	$O(V)$	DFS visits each node and edge once
Space	$O(V)$	$O(V)$	Visited array + stack both $O(V)$

■ Common Mistakes

Watch Out

- Pushing node to stack BEFORE recursion — must push AFTER (post-order).
- Running on cyclic graph — topological sort undefined for graphs with cycles.
- Not iterating all nodes — disconnected nodes get missed.

■ Practice Problems

Case	Time	Space	Why
Medium	LC 207	Course Schedule	Check if topo sort possible
Medium	LC 210	Course Schedule II	Return topo sort order

Topological Sort (Kahn's BFS)

● MEDIUM

■ Introduction

Kahn's Algorithm uses BFS and the concept of IN-DEGREE (number of incoming edges). Nodes with in-degree 0 have no prerequisites — they can be processed first. After processing, reduce neighbors' in-degree; new zero-degree nodes join the queue.

■ Core Intuition

Kahn's Steps

- Compute in-degree of every node.
- Add all nodes with in-degree = 0 to queue.
- While queue not empty: dequeue node, add to result, reduce in-degree of all neighbors.
- If neighbor's in-degree becomes 0 → add to queue.
- If result size != V at end → cycle exists (topological sort impossible).

■ Java Implementation

Java

```
class Solution {
    public int[] topoSort(int V, ArrayList<ArrayList<Integer>> adj) {
        int[] inDegree = new int[V];

        // Calculate in-degree of each node
        for (int i=0; i<V; i++)
            for (int neighbor : adj.get(i))
                inDegree[neighbor]++;

        // Add all nodes with in-degree 0 to queue
        Queue<Integer> queue = new LinkedList<>();
        for (int i=0; i<V; i++)
            if (inDegree[i] == 0) queue.add(i);

        int[] result = new int[V];
        int idx = 0;

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

```

int node = queue.poll();
result[idx++] = node;

for (int neighbor : adj.get(node)) {
    inDegree[neighbor]--;
    if (inDegree[neighbor] == 0) queue.add(neighbor);
}
}

// If idx != V, there was a cycle
return (idx == V) ? result : new int[0];
}
}

```

■ Time & Space Complexity

Case	Time	Space	Why
Time	$O(V + E)$	$O(V)$	In-degree calc $O(E)$, BFS $O(V+E)$
Space	$O(V)$	$O(V)$	In-degree array + queue both $O(V)$

Kahn's vs DFS Topo Sort

- Kahn's (BFS): naturally detects cycles (result size $< V$), iterative, intuitive.
- DFS Topo: needs separate cycle detection, recursive (stack overflow risk on large graphs).
- Both produce valid topological orders. Prefer Kahn's for cycle detection built-in.

■ Practice Problems

Case	Time	Space	Why
Medium	LC 207	Course Schedule	Check cycle using in-degree
Medium	LC 210	Course Schedule II	Return Kahn's topo order
Hard	LC 1136	Parallel Courses	Min semesters = max topo level

Dijkstra's Algorithm

● MEDIUM

■ Introduction

Dijkstra's algorithm finds the shortest path from a source node to all other nodes in a WEIGHTED graph (non-negative weights). It is the foundation of GPS navigation, network routing, and all weighted shortest-path problems.

■ Core Intuition

Greedy + Priority Queue

- Maintain a `dist[]` array: shortest known distance from source to each node.
- Use a Min-Heap (Priority Queue): always process the node with minimum current distance.
- When processing node `u`: for each neighbor `v`, check if $\text{dist}[u] + \text{weight}(u,v) < \text{dist}[v]$.
- If yes, update `dist[v]` and add $(\text{dist}[v], v)$ to the heap.
- Key insight: once a node is popped from heap, its shortest distance is FINALIZED.

■ Visual

Dijkstra's Trace

Graph (weighted):

0 --3-- 1 --1-- 2

| |

4 2

| |

3 --5-- 4

Dijkstra from 0:

`dist[] = [0, INF, INF, INF, INF]`

Heap: $[(0,0)]$

Pop (0,0): update $1 \rightarrow 3$, $3 \rightarrow 4$. Heap: $[(3,1), (4,3)]$

Pop (3,1): update $2 \rightarrow 4$, $4 \rightarrow 5$. Heap: $[(4,2), (4,3), (5,4)]$

Pop (4,2): no better path. Heap: $[(4,3), (5,4)]$

Pop (4,3): update nothing. Heap: $[(5,4)]$

`dist[] = [0, 3, 4, 4, 5]`

■ Java Implementation

Java

```
import java.util.*;

class Solution {

    public int[] dijkstra(int V, List<List<int[]>> adj, int src) {

        int[] dist = new int[V];

        Arrays.fill(dist, Integer.MAX_VALUE);

        dist[src] = 0;

        // Min-heap: [distance, node]

        PriorityQueue<int[]> pq = new PriorityQueue<>((a,b) -> a[0]-b[0]);

        pq.offer(new int[]{0, src});

        while (!pq.isEmpty()) {

            int[] curr = pq.poll();

            int d = curr[0], node = curr[1];

            if (d > dist[node]) continue; // Stale entry, skip

            for (int[] edge : adj.get(node)) {

                int neighbor = edge[0], weight = edge[1];

                int newDist = dist[node] + weight;

                if (newDist < dist[neighbor]) {

                    dist[neighbor] = newDist;

                    pq.offer(new int[]{newDist, neighbor});

                }

            }

        }

        return dist;

    }

}
```

■ ■ Time & Space Complexity

Case	Time	Space	Why
Time	$O((V+E) \log V)$	$O(V)$	Each edge processed once; heap operations $O(\log V)$
Space	$O(V + E)$	$O(V)$	Adjacency list $O(V+E)$; dist array + heap $O(V)$

Common Mistakes

Critical Pitfalls

- Using Dijkstra with **NEGATIVE** weights — undefined behavior! Use Bellman-Ford instead.
- Not skipping stale heap entries ($d > \text{dist}[\text{node}]$) — causes incorrect results.
- Using max-heap instead of min-heap — always use min-heap (compare by distance).
- Forgetting to initialize `dist[]` with `MAX_VALUE` except source.

Practice Problems

Case	Time	Space	Why
Medium	LC 743	Network Delay Time	Classic Dijkstra's
Medium	LC 787	Cheapest Flights K Stops	Modified Dijkstra / Bellman-Ford
Hard	LC 1514	Path with Max Probability	Dijkstra with probabilities
Hard	LC 778	Swim in Rising Water	Dijkstra on grid

Mental Model

Remember Forever

- "Dijkstra is a greedy algorithm: always expand the nearest unfinalized node. Like expanding a growing circle of confirmed-shortest paths from the source."

Bellman Ford Algorithm

● MEDIUM

■ Introduction

Bellman-Ford is the answer to Dijkstra's limitation: it handles NEGATIVE edge weights. It works by relaxing ALL edges $V-1$ times. If a shorter path is still found after $V-1$ iterations, a negative cycle exists.

■ Core Intuition

Why $V-1$ Iterations?

- Any shortest path in a graph has at most $V-1$ edges (visiting each vertex at most once).
- After k iterations, `dist[]` contains shortest paths using at most k edges.
- After $V-1$ iterations: shortest paths using at most $V-1$ edges = all shortest paths.
- If ANY edge still relaxes on the V th iteration $\rightarrow$ negative cycle detected.

■ Visual

Bellman-Ford Trace

Graph with negative edge:

```
0 --5-- 1 ---4-- 2
```

```
\ /
```

```
-6 3
```

```
\ /
```

```
3
```

Iter 1: relax all edges $\rightarrow$ dist updates

Iter 2: further relaxation via negative edge

...after $V-1$ iters: stable shortest distances

Negative Cycle Detection:

If `dist[v] > dist[u] + w(u,v)` still holds on V th iteration $\rightarrow$ NEGATIVE CYCLE

■ Java Implementation

Java

```
class Solution {
    public int[] bellmanFord(int V, int[][] edges, int src) {
        // edges[i] = {u, v, weight}
```

```

int[] dist = new int[V];

Arrays.fill(dist, Integer.MAX_VALUE);

dist[src] = 0;

// Relax all edges V-1 times
for (int i=1; i < V; i++) {
    for (int[] edge : edges) {
        int u = edge[0], v = edge[1], w = edge[2];
        if (dist[u] != Integer.MAX_VALUE && dist[u]+w < dist[v]) {
            dist[v] = dist[u] + w;
        }
    }
}

// Vth relaxation: detect negative cycle
for (int[] edge : edges) {
    int u = edge[0], v = edge[1], w = edge[2];
    if (dist[u] != Integer.MAX_VALUE && dist[u]+w < dist[v]) {
        System.out.println("Negative cycle detected!");
        return new int[0];
    }
}

return dist;
}

```

■ Time & Space Complexity

Case	Time	Space	Why
Time	$O(V * E)$	$O(V)$	$V-1$ iterations, each relaxing E edges
Space	$O(V)$	$O(V)$	dist[] array only; no heap needed

Dijkstra vs Bellman-Ford Summary

- Dijkstra: $O((V+E)\log V)$, NO negative weights, greedy with priority queue.
- Bellman-Ford: $O(V * E)$, handles negative weights, detects negative cycles.
- Use Dijkstra when all weights non-negative (faster). Use Bellman-Ford with negative weights.

Common Mistakes

Avoid These

- Only doing V-2 iterations — need exactly V-1 to guarantee all shortest paths.
- Not checking `dist[u] != MAX_VALUE` before adding weight — causes integer overflow.
- Confusing with Dijkstra — Bellman-Ford does NOT use a priority queue.

Practice Problems

Case	Time	Space	Why
Medium	LC 787	Cheapest Flights K Stops	Bellman-Ford with K constraint
Medium	LC 743	Network Delay Time	Compare Dijkstra vs Bellman
Hard	LC 1334	Find the City Min Neighbors	All-pairs shortest path

Graphs — Overall Quick Revision

- BFS: Queue, level-by-level, shortest path (unweighted), `visited[]` before enqueue
- DFS: Recursion/Stack, deep-first, `visited[]` before recursion
- Connected Components: run BFS/DFS from each unvisited node
- Cycle (undirected): BFS with parent tracking
- Cycle (directed): DFS with `visited[]` + `inStack[]`
- Topo Sort DFS: post-order push to stack
- Topo Sort Kahn's: `in-degree` → `queue` → `reduce` → BFS
- Dijkstra: Min-Heap, non-negative weights, $O((V+E)\log V)$
- Bellman-Ford: V-1 relaxations, handles negatives, $O(V \cdot E)$
- Grid problems: treat as graph, 4-directional DFS/BFS

Final Mental Model

The Graph Cheat Sheet

- Unweighted shortest path → BFS
- Weighted, non-negative → Dijkstra
- Weighted, with negatives → Bellman-Ford
- Task ordering → Topological Sort
- Connected components → DFS/BFS from each unvisited
- Grid problems → 2D BFS/DFS with direction arrays