Algorithms for graph problems

Depth-First Search

Here is a detailed, step-by-step guide to the Depth First Search (DFS) algorithm, designed to be easy to visualize and understand without complex mathematical notation.

Part 1: What is DFS and Why Do We Need It?

What it is: Depth First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. Think of it like walking through a maze. When you hit a fork in the road, you choose one path and follow it **as deep as possible** until you hit a dead end. Only then do you backtrack to the last fork and try the next path.

Why we need it:

- **Pathfinding:** It is great for finding *a* path (not necessarily the shortest) between two points.
- **Topological Sorting:** Used in scheduling tasks (e.g., determining which library to compile first).
- **Solving Puzzles:** Sudoku or mazes, where you need to exhaust one possibility before trying another.
- **Detecting Cycles:** Checking if a graph has a loop (A connects to B, B connects to A).

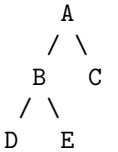
Part 2: Visual Walkthrough (How it Works)

To understand DFS, we use a data structure called a **Stack** (Last-In, First-Out). This can be done explicitly with a list or implicitly using recursion.

The Rules:

1. Start at a node. Mark it as **Visited**.
2. Move to an adjacent unvisited neighbor.
3. If you reach a node with no unvisited neighbors (a "dead end"), **Backtrack** to the previous node.
4. Repeat until all nodes are visited.

The Example Graph:



Step-by-Step Execution:

Step 1: Start at A We begin at the root. We mark 'A' as visited and look at its neighbors (B, C). By convention, we'll go left first.

```
GRAPH STATE:            TRACKER:
      A [CURRENT]       Visited List: [A]
     / \                Stack (Path): [A]
    B   C
   / \
  D   E
```

Step 2: Dive Deeper to B From A, we move to B. We ignore C for now (we will come back to it later).

```
      A
     / \
    B [CURRENT]         Visited List: [A, B]
   / \                  Stack (Path): [A, B]
  D   E
```

Step 3: Dive Deeper to D From B, we move to neighbors D and E. Let's pick D.

```
      A
     / \
    B   C
   / \
  D [CURRENT]           Visited List: [A, B, D]
     E                  Stack (Path): [A, B, D]
```

Step 4: Dead End at D (Backtrack) D has no neighbors (or no unvisited ones). We cannot go deeper. We pop D off the stack and go back up to B to see if it has other options.

```
      A
     / \
    B [CURRENT]         Visited List: [A, B, D]
   / \                  Stack (Path): [A, B]  <-- D removed
  D   E
```

Step 5: Visit B's other neighbor (E) Back at B, we see D is done, but E is still unvisited. We move to E.

```
      A
     / \
    B   C
   / \
  D   E [CURRENT]       Visited List: [A, B, D, E]
                        Stack (Path): [A, B, E]
```

Step 6: Dead End at E (Backtrack) E is a leaf node (dead end). We backtrack to B.

```
      A
     / \
    B [CURRENT]        Visited List: [A, B, D, E]
   / \                 Stack (Path): [A, B] <-- E removed
  D   E
```

Step 7: Backtrack to A We are back at B. B has no more unvisited neighbors (D and E are done). So we backtrack to A.

```
      A [CURRENT]      Visited List: [A, B, D, E]
     / \               Stack (Path): [A]    <-- B removed
    B   C
   / \
  D   E
```

Step 8: Visit A's other neighbor (C) Back at A, we see B is done. We move to C.

```
      A
     / \
    B   C [CURRENT]    Visited List: [A, B, D, E, C]
   / \                 Stack (Path): [A, C]
  D   E
```

Step 9: Finish C is a dead end. Backtrack to A. A has no more neighbors. The stack is empty. We are done.

Part 3: Code Implementation

Here is how you write this algorithm in Python and JavaScript. This uses the **Recursive** approach, which is the most common and "cleanest" way to write DFS.

Python

```python
def dfs(graph, node, visited):
    # 1. Mark current node as visited
    if node not in visited:
        print(node)
        visited.add(node)

        # 2. Iterate through all neighbors
        for neighbor in graph[node]:
            # 3. Recursively call DFS
            dfs(graph, neighbor, visited)
```

```
# Graph represented as an Adjacency List (Dictionary)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': [],
    'D': [],
    'E': []
}

visited_set = set()
dfs(graph, 'A', visited_set)
```

JavaScript

```
function dfs(graph, node, visited) {
    // 1. Mark current node as visited
    if (!visited.has(node)) {
        console.log(node);
        visited.add(node);

        // 2. Iterate through all neighbors
        const neighbors = graph[node];
        for (let neighbor of neighbors) {
            // 3. Recursively call DFS
            dfs(graph, neighbor, visited);
        }
    }
}

// Graph represented as an Object (Adjacency List)
const graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': [],
    'D': [],
    'E': []
};

const visitedSet = new Set();
dfs(graph, 'A', visitedSet);
```

Part 4: Analysis & Application

Complexity Analysis We use **V** for Vertices (nodes) and **E** for Edges (connections).

Time Complexity: O(V + E)

- **Why?** In the worst case, we visit every node (V) exactly once and cross every connection (E) once.
- **Diagram:**

```
Processing Nodes (V)  +  Scanning Connections (E)
[ A, B, C, D, E ]     +  [ A-B, A-C, B-D, B-E ]
```

Space Complexity: O(V)

- **Why?** In the worst case (a long straight line graph), the recursion stack (or memory stack) will hold all vertices at once before backtracking.
- **Diagram:**

```
Worst Case Stack (Linear Graph):
A -> B -> C -> D -> E
Stack: [A, B, C, D, E]  <-- Takes O(V) space
```

How to use DFS on LeetCode When you see a new problem, ask yourself: *"Does this problem require me to explore every possible path to find a solution?"* or *"Do I need to find connected components (like islands)?"*

Strategy: "The Island Thinking" Imagine a grid of numbers where 1 is land and 0 is water. To count the number of islands, you land on a '1', then run DFS to mark all connected '1's as visited (sinking the island) so you don't count them again.

LeetCode Keywords indicating DFS:

1. "Find all paths…"
2. "Calculate the depth…"
3. "Number of islands…"
4. "Connected components…"

DFS vs. BFS (Breadth First Search)

Feature	DFS (Depth First)	BFS (Breadth First)
Strategy	Plunge deep, then backtrack.	Explore neighbors level by level (ripples in a pond).
Data Structure	Stack (Recursion).	Queue (First-In, First-Out).
Best For	Mazes, Puzzles, Path existence.	Finding the **Shortest Path** (GPS).

Feature	DFS (Depth First)	BFS (Breadth First)
Memory	Lower memory on wide trees.	Higher memory (stores all neighbors at once).

Breadth-First Search

Here is a detailed guide to the Breadth First Search (BFS) algorithm, designed to be easy to visualize and understand without complex mathematical notation.

What is Breadth First Search (BFS)?

Breadth First Search (BFS) is a fundamental graph traversal algorithm. Imagine dropping a stone into a calm pond. The ripples move outward in perfect circles, hitting everything close to the center first, then moving further out.

BFS works exactly like those ripples. It starts at a specific node (the "root") and explores all its immediate neighbors first (Layer 1) before moving to the neighbors' neighbors (Layer 2).

Why do we need it?

1. **Shortest Path:** In an unweighted graph (where every edge has the same "cost"), BFS guarantees finding the shortest path from the start to the target.
2. **Level-Order Traversal:** If you need to process data in layers (e.g., organizational hierarchy, networking hops), BFS is the tool.
3. **Connected Components:** It helps find all items connected to a specific starting point.

How It Works: The "Queue" Strategy

BFS relies on a data structure called a **Queue**. A Queue is **FIFO** (First In, First Out)—like a line of people waiting for a bus. The first person in line is the first one to board.

The Rules:

1. **Enqueue (Add):** Add a node to the back of the line.
2. **Dequeue (Remove):** Remove a node from the front of the line to process it.
3. **Mark Visited:** Keep a list of nodes you have already seen so you don't process them twice (preventing infinite loops).

Step-by-Step Visual Walkthrough

Let's use a simple graph. We want to traverse every node starting from **A**.

The Graph:

```
    A
   / \
  B   C
  |   |
  D   E
```

- **A** connects to **B** and **C**.
- **B** connects to **D**.
- **C** connects to **E**.

Step 1: Initialization We start at **A**. We look at it, mark it as "Visited," and add it to our Queue.

```
Status: Start at A

QUEUE: [ A ]
VISITED: { A }

   (A)  <-- In Queue
   / \
  B   C
  |   |
  D   E
```

Step 2: Process A We take **A** out of the Queue (dequeue). We look at A's neighbors: **B** and **C**. They haven't been visited, so we add them to the Queue and mark them visited.

```
Status: Dequeued A. Found neighbors B, C.

QUEUE: [ B, C ]   <-- A is gone, neighbors added
VISITED: { A, B, C }

    A   <-- Processed
   / \
 (B) (C) <-- In Queue
  |   |
  D   E
```

Step 3: Process B The next item in line is **B**. We dequeue **B**. We look at B's neighbor: **D**. D is not visited, so we add D to the Queue.

```
Status: Dequeued B. Found neighbor D.

QUEUE: [ C, D ]   <-- C was waiting, D joined back of line
VISITED: { A, B, C, D }

    A
   / \
  B  (C) <-- Next to be processed
  |   |
 (D)  E  <-- Just added
```

Step 4: Process C The next item in line is **C**. We dequeue **C**. We look at C's neighbor: **E**. E is not visited, so we add E to the Queue.

```
Status: Dequeued C. Found neighbor E.

QUEUE: [ D, E ]
VISITED: { A, B, C, D, E }

    A
   / \
  B   C
  |   |
 (D) (E) <-- Both in Queue
```

Step 5: Process D We dequeue **D**. D has no neighbors (or no unvisited neighbors). We do nothing but remove it.

```
Status: Dequeued D. No new neighbors.

QUEUE: [ E ]
VISITED: { A, B, C, D, E }

    A
   / \
  B   C
  |   |
  D  (E) <-- Last one
```

Step 6: Process E We dequeue **E**. No new neighbors. The Queue is now empty. We are done!

```
Status: Queue Empty. Finished.

QUEUE: [ ]
Traversal Order: A -> B -> C -> D -> E
```

Complexity Analysis

Here is the cost of running BFS, visualized.

Time Complexity: O(V + E)

- **V** = Vertices (Nodes)
- **E** = Edges (Connections)

We visit every Node once, and we inspect every Connection once.

```
| TIME COMPLEXITY VISUALIZED
|
|  Processing Nodes (V)  |  Checking Connections (E)
|  [A] [B] [C] [D] [E]   |  A->B, A->C, B->D, C->E
|           +            |             +
|  ------------------------------------------
|           Total Work = V + E
```

Space Complexity: O(V) In the worst case, we might have to hold many nodes in the Queue at once (for example, if A connected to 100 other nodes immediately).

```
| SPACE COMPLEXITY VISUALIZED (Queue Size)
|
|  Worst Case: The graph is very "wide"
|
|  Queue: [ node1, node2, node3 ... nodeV ]
|
|  Memory Usage scales with the width of the graph (up to V)
```

Algorithm Code

Here is how you write this in code.

Python Python uses `deque` (double-ended queue) because popping from the start of a standard list is slow.

```python
from collections import deque

def bfs(graph, start_node):
    # 1. Initialize Queue and Visited Set
    queue = deque([start_node])
    visited = set([start_node])

    print(f"Starting BFS from: {start_node}")
```

```
    while queue:
        # 2. Dequeue the current node
        current_node = queue.popleft()
        print(f"Processing: {current_node}")

        # 3. Explore neighbors
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Example Graph (Adjacency List)
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
}

bfs(graph, 'A')
```

JavaScript In JavaScript, we can use a standard Array. `.shift()` removes from the front, `.push()` adds to the back.

```
function bfs(graph, startNode) {
    // 1. Initialize Queue and Visited Set
    let queue = [startNode];
    let visited = new Set([startNode]);

    console.log("Starting BFS from:", startNode);

    while (queue.length > 0) {
        // 2. Dequeue the current node
        let currentNode = queue.shift();
        console.log("Processing:", currentNode);

        // 3. Explore neighbors
        let neighbors = graph[currentNode];
        for (let neighbor of neighbors) {
            if (!visited.has(neighbor)) {
                visited.add(neighbor);
                queue.push(neighbor);
            }
        }
```

```
    }
}

// Example Graph
const graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': [],
    'E': []
};

bfs(graph, 'A');
```

LeetCode Strategy: How to use BFS

Target Problem: *Rotting Oranges* or *Shortest Path in Binary Matrix*.

The Mental Model: When you read a problem, look for these keywords:

1. **"Shortest path"** or **"Minimum steps"**
2. **"Nearest X"**
3. **"Level order"**

How to approach a grid problem (like a maze): Think of the grid cells as Nodes and the Up/Down/Left/Right movements as Edges.

1. **Start:** Put your starting cell `(row, col)` into the Queue.
2. **Track:** Create a `distance` matrix or map to keep track of how far you have traveled.
3. **Loop:** While the Queue isn't empty, pop a cell, look at its 4 neighbors. If a neighbor is valid (not a wall) and unvisited, add it to the Queue and increase the distance count by +1.

Comparison: BFS vs. DFS

Feature	BFS (Breadth First)	DFS (Depth First)
Data Structure	Queue (FIFO)	Stack (LIFO) or Recursion
Movement	Layer by layer (Wide)	Deep into one path (Deep)
Best for...	Shortest path in unweighted graphs	Maze solving, game simulations
Memory	High (stores a whole layer)	Low (stores just one active path)

Feature	BFS (Breadth First)	DFS (Depth First)
Visual	Ripples in a pond	A snake going into a tunnel

Topological Sort: Kahn's Algorithm

What is Topological Sort and Why Do We Need It?

Topological Sort is a linear ordering of vertices (nodes) in a directed graph such that for every directed edge from vertex A to vertex B, vertex A comes before vertex B in the ordering.

Key Requirement: It only works on **DAGs** (Directed Acyclic Graphs). If the graph has a cycle (e.g., A -> B -> A), you cannot topologically sort it because there is no clear start or end.

Why we need it (The Problem it Solves): Imagine you are baking a cake or compiling code. You cannot bake the cake before mixing the batter, and you cannot mix the batter before buying eggs. Topological sort solves dependency problems.

- **Task Scheduling:** "Task B depends on Task A."
- **Build Systems:** "Library X needs Library Y to be built first."
- **Course Prerequisites:** "You must take Math 101 before Math 102."

How Kahn's Algorithm Works

Kahn's Algorithm uses the concept of **"In-Degree"**.

- **In-Degree:** The number of edges coming *into* a node.
- **The Logic:** If a node has an In-Degree of 0, it has no dependencies (or all its dependencies are already met). It is safe to process this node.

The Algorithm in Plain English:

1. Calculate the In-Degree for every node.
2. Put all nodes with **In-Degree 0** into a Queue.
3. While the Queue is not empty:

- Remove a node from the Queue and add it to the Sorted List.
- "Remove" that node from the graph (virtually) by looking at its neighbors and decreasing their In-Degree by 1.
- If a neighbor's In-Degree drops to 0, add it to the Queue.

Visual Walkthrough

We will sort this simple dependency graph.

The Graph:

- Node **A** points to **B** and **C**.
- Node **B** points to **D**.
- Node **C** points to **D**.

Edges: `A -> B`, `A -> C`, `B -> D`, `C -> D`

```
      [A]
     /   \
   [B]   [C]
     \   /
      [D]
```

Step 0: Initialization We calculate the In-Degree for all nodes.

- A: 0 (No arrows pointing in)
- B: 1 (Arrow from A)
- C: 1 (Arrow from A)
- D: 2 (Arrows from B and C)

Since **A** has 0 in-degree, it goes into the Queue.

```
STATUS: Start
------------------------------------------------
Graph State (In-Degrees shown in parens):
      [A](0)
     /      \
   [B](1)   [C](1)
     \      /
      [D](2)

Queue:       [ A ]
Sorted List: []
------------------------------------------------
```

Step 1: Process Node A We pop **A** from the Queue and add it to the Sorted List. We look at A's neighbors (**B** and **C**) and decrease their in-degrees by 1.

- B becomes 0.
- C becomes 0. Both B and C are now ready, so we add them to the Queue.

```
STATUS: Processing A
------------------------------------------------
      (Processed)
```

```
          |
         [A] -- removed dependency --> [B](0)  (Added to Queue)
          |
          +--- removed dependency --> [C](0)  (Added to Queue)

      [D](2)  (Unchanged)

Queue:       [ B, C ]
Sorted List: [ A ]
-------------------------------------------------
```

Step 2: Process Node B We pop **B** from the Queue. We look at B's neighbor (**D**) and decrease its in-degree by 1.

- D was 2, now becomes 1. D is not 0 yet, so we do *not* add it to the Queue.

```
STATUS: Processing B
-------------------------------------------------
      (Processed)
          |
         [A]
        /   \
(Processed) [C](0)  (Waiting in Queue)
   [B]
     \
      +--- removed dependency --> [D](1)

Queue:       [ C ]
Sorted List: [ A, B ]
-------------------------------------------------
```

Step 3: Process Node C We pop **C** from the Queue. We look at C's neighbor (**D**) and decrease its in-degree by 1.

- D was 1, now becomes 0. D is now ready! Add D to the Queue.

```
STATUS: Processing C
-------------------------------------------------
      (Processed)
          |
         [A]
        /   \
(Processed) (Processed)
   [B]       [C]
     \       /
      \     +--- removed dependency --> [D](0) (Added to Queue)
       \
```

```
        (from B, already handled)

Queue:       [ D ]
Sorted List: [ A, B, C ]
-------------------------------------------------
```

Step 4: Process Node D We pop **D** from the Queue. D has no outgoing edges, so we stop there.

```
STATUS: Processing D
-------------------------------------------------
      (Processed)
          |
         [A]
        /   \
(Processed) (Processed)
   [B]       [C]
     \       /
      (Processed)
         [D]

Queue:       [ ]  (Empty -> Done!)
Sorted List: [ A, B, C, D ]
-------------------------------------------------
```

Final Topological Sort: A, B, C, D (Note: A, C, B, D is also a valid valid answer depending on queue order).

Complexity Analysis

Here is the breakdown of the efficiency.

TIME COMPLEXITY: O(V + E)	
1. Calculate In-Degrees	Visit every node and edge once Cost: O(V + E)
2. Queue Operations	Each Node is added/removed once. Cost: O(V)
3. Neighbor Updates	We iterate through neighbors of each node. Total edges. Cost: O(E)

TOTAL	V (Vertices) + E (Edges)

SPACE COMPLEXITY: O(V + E)	
1. Adjacency List	Stores the graph structure Cost: O(V + E)
2. In-Degree Array	Stores count for each node Cost: O(V)
3. Queue	Worst case stores all nodes Cost: O(V)

Code Implementation

The graph is represented as a dictionary/map where keys are nodes and values are lists of neighbors.

Python

```python
from collections import deque

def topological_sort(vertices, edges):
    # 1. Initialize Graph and In-Degree
    graph = {v: [] for v in vertices}
    in_degree = {v: 0 for v in vertices}

    # Build the graph and count in-degrees
    for src, dest in edges:
        graph[src].append(dest)
        in_degree[dest] += 1

    # 2. Add 0 in-degree nodes to Queue
    queue = deque([node for node in vertices if in_degree[node] == 0])
    sorted_order = []

    # 3. Process Queue
    while queue:
        current = queue.popleft()
        sorted_order.append(current)

        # Decrease in-degree of neighbors
```

```
        for neighbor in graph[current]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Check for cycles
    if len(sorted_order) != len(vertices):
        return "Cycle detected! Topological sort not possible."

    return sorted_order

# Example Usage
nodes = ["A", "B", "C", "D"]
edge_list = [("A", "B"), ("A", "C"), ("B", "D"), ("C", "D")]
print(topological_sort(nodes, edge_list))
# Output: ['A', 'B', 'C', 'D']
```

JavaScript

```
function topologicalSort(vertices, edges) {
    // 1. Initialize Graph and In-Degree
    const graph = new Map();
    const inDegree = new Map();

    vertices.forEach(v => {
        graph.set(v, []);
        inDegree.set(v, 0);
    });

    // Build graph
    edges.forEach(([src, dest]) => {
        graph.get(src).push(dest);
        inDegree.set(dest, inDegree.get(dest) + 1);
    });

    // 2. Add 0 in-degree nodes to Queue
    const queue = [];
    vertices.forEach(v => {
        if (inDegree.get(v) === 0) queue.push(v);
    });

    const sortedOrder = [];

    // 3. Process Queue
    while (queue.length > 0) {
        const current = queue.shift(); // Remove from front
```

```
            sortedOrder.push(current);

            const neighbors = graph.get(current);
            neighbors.forEach(neighbor => {
                inDegree.set(neighbor, inDegree.get(neighbor) - 1);
                if (inDegree.get(neighbor) === 0) {
                    queue.push(neighbor);
                }
            });
        }

        // Check for cycles
        if (sortedOrder.length !== vertices.length) {
            return "Cycle detected!";
        }

        return sortedOrder;
    }

    const nodes = ["A", "B", "C", "D"];
    const links = [["A", "B"], ["A", "C"], ["B", "D"], ["C", "D"]];
    console.log(topologicalSort(nodes, links));
```

Comparison & LeetCode Strategy

LeetCode Strategy: "Course Schedule" When you see a problem asking:

1. "Can all courses be finished?"
2. "Find the order to take courses."
3. "Detect a circular dependency."

Think Kahn's Algorithm immediately.

Pro-Tip for Cycle Detection: If the length of your `sorted_order` list is **less** than the total number of nodes (V), it means a cycle exists. This is because nodes involved in a cycle will never reach an In-Degree of 0 and will never enter the queue.

Kahn's Algorithm (BFS) vs. DFS Approach

Feature	Kahn's Algorithm (BFS based)	DFS Approach
Concept	Iteratively peel off "free" nodes (0 in-degree).	Traverse deep, add node to list when backtracking.

Feature	Kahn's Algorithm (BFS based)	DFS Approach
Data Structure	Queue + In-Degree Array.	Recursion Stack + Visited Set.
Cycle Detection	Very Intuitive (check result length).	Requires an extra "recursion stack" tracker.
Order Direction	Generates order naturally (First to Last).	Generates Reverse Order (must reverse list at end).

Winner: Kahn's is generally easier to implement and reason about for **cycle detection** and simple ordering. DFS is better if you are already doing a deep traversal for other graph properties (like finding connected components).

Distinct Set Union: Union-Find Algorithm

Here is a detailed, step-by-step guide to the Disjoint Set Union (DSU) algorithm, also known as Union-Find.

---

1. The Problem: Dynamic Connectivity

Imagine you are building a computer network. You start with isolated computers. Gradually, you add cables (connections) between them. At any point, you need to quickly answer two questions:

1. **Are Computer A and Computer B connected?** (Directly or indirectly).
2. **Connect Computer A and Computer B.**

If you used standard Graph algorithms like BFS or DFS, checking connectivity is fast, but adding new connections and maintaining the structure efficiently is hard. If you have millions of nodes and edges, running a full traversal every time you add a cable is too slow.

We need a data structure that effectively supports:

- **Union:** Connecting two objects.
- **Find:** Determining if two objects are in the same component (group).

---

2. What is Union-Find?

Union-Find is a data structure that tracks a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

It thinks of every group of connected nodes as a "clan" or a "tree." Every clan has a **Representative** (often called the **Root** or **Leader**).

- **Find(x):** "Who is the leader of x?" (Traverse up the tree until you find the root).
- **Union(x, y):** "Merge the clan of x and the clan of y." (Find the leaders of both, and make one leader report to the other).

The Two Critical Optimizations

To make this incredibly fast, we use two tricks:

1. **Path Compression:** When looking up a node's leader, we flatten the path so future lookups are instant.
2. **Union by Rank (or Size):** When merging two trees, always attach the shorter tree to the taller tree to prevent the tree from getting too deep.

---

3. Visual Walkthrough (ASCII)

Let's trace the algorithm with a small example: **5 nodes (0 to 4)**.

State 0: Initialization

Initially, every node is its own parent. Everyone is a leader of their own set.

Parent Array: `[0, 1, 2, 3, 4]`

Graph State:

```
  0      1      2      3      4
 (L)    (L)    (L)    (L)    (L)
```

(L) denotes a Leader/Root.

Step 1: Union(0, 1)

We want to connect 0 and 1.

1. Find leader of 0 -> 0.
2. Find leader of 1 -> 1.
3. Roots are different. Make 0 the parent of 1.

Parent Array: `[0, 0, 2, 3, 4]` (Index 1 now holds value 0)

Graph State:

```
   0       2       3       4
  /       (L)     (L)     (L)
 1
```

Node 0 is now the Leader of the set {0, 1}.

Step 2: Union(2, 3)

Connect 2 and 3.

1. Find leader of 2 -> 2.
2. Find leader of 3 -> 3.
3. Make 2 the parent of 3.

Parent Array: `[0, 0, 2, 2, 4]`

Graph State:

```
   0           2           4
  /           /           (L)
 1           3
```

We now have three sets: {0,1}, {2,3}, and {4}.

Step 3: Union(1, 4)

Connect 1 and 4.

1. **Find(1):** Go up from 1 to 0. Leader is **0**.
2. **Find(4):** Leader is **4**.
3. Merge leader 4 into leader 0.

Parent Array: `[0, 0, 2, 2, 0]` (Index 4 now points to 0)

Graph State:

```
     0           2
    / \         /
   1   4       3
```

Note: Even though we called Union(1, 4), we actually connected the ROOTS (0 and 4). Node 4 becomes a sibling of 1.

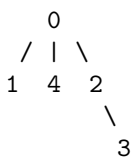
Step 4: Union(3, 4)

Connect 3 and 4.

1. **Find(3):** Go up from 3 -> 2. Leader is **2**.
2. **Find(4):** Go up from 4 -> 0. Leader is **0**.
3. Merge leader 2 into leader 0. (We attach the tree rooted at 2 to the tree rooted at 0).

Parent Array: `[0, 0, 0, 2, 0]` (Index 2 now points to 0)

Graph State (Before Path Compression):



Everyone is now in one big set rooted at 0.

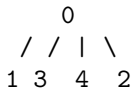
Step 5: Find(3) with Path Compression

Now, let's see the magic. We ask: "Who is the leader of 3?"

1. Start at 3. Parent is 2.
2. Go to 2. Parent is 0.
3. Go to 0. Parent is 0 (Root found).
4. **Compression:** As we return, we update the parent of visited nodes to point directly to the root.

- Node 3's parent becomes 0.

Graph State (After Find(3)):



Node 3 moved! It now points directly to 0. Next time we search for 3, it's just one step.

---

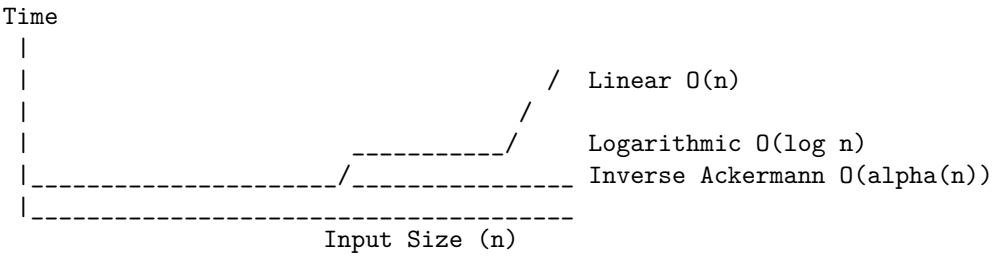
4. Complexity Analysis

Time Complexity: Inverse Ackermann Function

The time complexity is technically `O(alpha(n))`.

Visualizing alpha(n): Imagine a graph of growth.

- Linear O(n) goes up steadily.
- Logarithmic O(log n) goes up slowly.
- **alpha(n)** is incredibly flat.



For all practical purposes in the universe (up to 10^80 atoms), `alpha(n)` is **less than 5**. It is **nearly constant time**, i.e., almost O(1).

Space Complexity

We need two arrays (Parent and Rank/Size), both of size N. **Space:** O(N)

```
Memory Usage:
[ P0 | P1 | P2 | ... | Pn ]  <- Parent Array
[ R0 | R1 | R2 | ... | Rn ]  <- Rank Array
```

---

5. Implementation

Here is the code implementing both **Path Compression** and **Union by Rank**.

Python

```python
class UnionFind:
    def __init__(self, size):
        # Initially, each node is its own parent
        self.parent = list(range(size))
        # Rank helps keep the tree flat
        self.rank = [1] * size

    def find(self, x):
        # Path Compression
        if self.parent[x] != x:
            # Recursively find the root and point x directly to it
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        rootX = self.find(x)
```

```
        rootY = self.find(y)

        if rootX != rootY:
            # Union by Rank: Attach smaller tree to larger tree
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1
            return True # Connection successful
        return False # Already connected
```

JavaScript

```
class UnionFind {
    constructor(size) {
        // Initially, parent[i] = i
        this.parent = Array.from({ length: size }, (_, i) => i);
        this.rank = new Array(size).fill(1);
    }

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

    union(x, y) {
        let rootX = this.find(x);
        let rootY = this.find(y);

        if (rootX !== rootY) {
            // Union by Rank
            if (this.rank[rootX] > this.rank[rootY]) {
                this.parent[rootY] = rootX;
            } else if (this.rank[rootX] < this.rank[rootY]) {
                this.parent[rootX] = rootY;
            } else {
                this.parent[rootY] = rootX;
                this.rank[rootX] += 1;
            }
            return true;
```

```
        }
        return false;
    }
}
```

---

6. Comparison & Strategy

When to use DSU on LeetCode?

Use DSU when you see problems involving:

1. **Connected Components:** "Count the number of provinces/islands."
2. **Cycle Detection:** "Is there a cycle in this undirected graph?" (If `find(u) == find(v)` before you union them, there is a cycle).
3. **Equivalence Classes:** "Group A is equivalent to B, B to C… is A equivalent to C?"

Comparison Table

Feature	DFS / BFS	Union-Find (DSU)
Connectivity Check	O(V + E) (Slow for dynamic)	Nearly O(1) (Fast)
Adding an Edge	O(1) (But requires re-traversal)	Nearly O(1)
Path Finding	**Yes** (Can tell you the path)	**No** (Only tells you *if* connected)
Graph Type	Directed & Undirected	Best for **Undirected**

> **Strategy Tip:** If the problem asks "What is the path from A to B?", use BFS. If the problem asks "Are A and B connected?" or "Group these items," use DSU.

Single Source Shortest Path: Dijkstra's Algorithm

Here is a detailed, step-by-step explanation of Dijkstra's Algorithm, designed to be easy to visualize and understand without complex mathematical notation.

What is Dijkstra's Algorithm?

The Problem: Imagine you are in a city (Node A) and want to get to another city (Node D). There are many roads (edges), and each road takes a different amount of time to travel (weight). You want to find the absolute fastest route not just to Node D, but potentially to every other city from your starting point.

The Solution: Dijkstra's Algorithm is a "greedy" algorithm. It finds the shortest path from a **Single Source** (starting node) to all other nodes in a graph with **non-negative weights**.

Why we need it:

- GPS Navigation (Google Maps finding the fastest route).
- Network Routing (sending data packets via the fastest path).
- Social Networks (degrees of separation).

How It Works (The Logic)

1. **Initialization:** Set the distance to your starting node to **0** and the distance to all other nodes to **Infinity**.
2. **Visit Priority:** Look at the node with the **smallest** known distance that you haven't "finished" yet.
3. **Exploration (Relaxation):** Check all neighbors of this current node. Calculate the distance to them *through* the current node.
4. **Update:** If this calculated distance is shorter than the distance you previously had on record for that neighbor, update the neighbor's distance.
5. **Repeat:** Mark the current node as "finished" (visited) and repeat the process until all reachable nodes are visited.

Visual Walkthrough

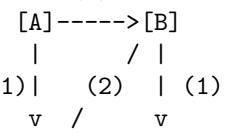
We will use a simple Directed Graph with 4 nodes: **A, B, C, D**. **Goal:** Find the shortest path from **A** to all other nodes.

The Graph Map:

- A connects to B (cost 4)
- A connects to C (cost 1)
- C connects to B (cost 2)
- B connects to D (cost 1)
- C connects to D (cost 5)

Initial State We track two things:

1. **Distances:** The best distance found so far (Start = A).
2. **Priority Queue:** A list of nodes to process, sorted by shortest distance.



```
  [C]----->[D]
      (5)

Distances: { A: 0,  B: inf,  C: inf,  D: inf }
Queue:     [(0, A)]  <-- We start with A because distance is 0
Visited:   {}
```

Step 1: Process Node A We pop **A** (dist 0). We look at A's neighbors: **B** and **C**.

1. **A -> B:** Current dist to A (0) + Edge (4) = 4. Since 4 < inf, update B.
2. **A -> C:** Current dist to A (0) + Edge (1) = 1. Since 1 < inf, update C.

```
       *
      (4)
  [A]=====>[B]  <-- Updated B to 4
   |       / |
(1)|    (2)  | (1)
   * /      v
  [C]----->[D]
   ^
   Updated C to 1

Distances: { A: 0,  B: 4,  C: 1,  D: inf }
Queue:     [(1, C), (4, B)]
Visited:   { A }
```

Note: C is at the front of the queue because 1 < 4.

Step 2: Process Node C We pop **C** (dist 1). Neighbors of C are **B** and **D**.

1. **C -> B:** Dist to C (1) + Edge (2) = 3.

- Compare: Is 3 smaller than B's current distance (4)? **Yes.**
- Update B to 3.

2. **C -> D:** Dist to C (1) + Edge (5) = 6.

- Compare: Is 6 smaller than D's current distance (inf)? **Yes.**
- Update D to 6.

```
      (4)
  [A]----->[B]
   |       ^ |
(1)|    (2)| | (1)
   v  /     v
```

```
  [C]=====>[D]
     =====
     ^
     Processing C

Distances: { A: 0,  B: 3,  C: 1,  D: 6 }
Queue:     [(3, B), (4, B_old), (6, D)]
Visited:   { A, C }
```

Note: We found a shortcut to B! Going A->C->B (cost 3) is faster than A->B (cost 4).

Step 3: Process Node B We pop **B** (dist 3). Neighbor of B is **D**.

1. **B -> D:** Dist to B (3) + Edge (1) = 4.

- Compare: Is 4 smaller than D's current distance (6)? **Yes.**
- Update D to 4.

```
     (4)
  [A]----->[B]
   |      / |
(1)|   (2)  | (1) *
   v  /     v
  [C]----->[D]
            ^
            Updated D to 4

Distances: { A: 0,  B: 3,  C: 1,  D: 4 }
Queue:     [(4, D), (6, D_old)]
Visited:   { A, C, B }
```

Note: We found a shortcut to D! Path A->C->B->D (cost 4) is faster than A->C->D (cost 6).

Step 4: Process Node D We pop **D** (dist 4). D has no outgoing neighbors. We are done with D.

Any "old" entries in the queue (like the old distance of 6 for D) are ignored because we have already processed D with a shorter distance.

Final Result The shortest path from A to every other node:

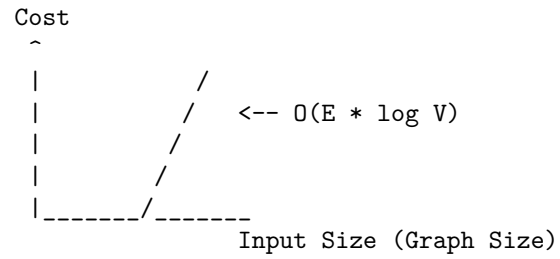
- **A:** 0
- **B:** 3 (Path: A->C->B)
- **C:** 1 (Path: A->C)

- **D:** 4 (Path: A->C->B->D)

Complexity Analysis

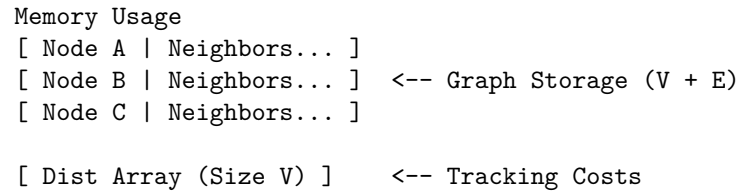
Here is how the efficiency breaks down, visualized. **V** = Number of Vertices (Nodes) **E** = Number of Edges

Time Complexity: O(E * log V) We visit every edge once (E), and for every edge, we might update the Priority Queue. Updating a Priority Queue (Heap) takes `log V` time.



Why it matters: It is much faster than checking every single combination of paths, but slower than a simple Breadth-First Search (BFS) which only works on unweighted graphs.

Space Complexity: O(V + E) We need to store the graph (Adjacency List) and the distances for every node.



Code Implementation

Python Solution Python is excellent for this because it has a built-in "min-heap" library called `heapq` that handles the sorting for us efficiently.

```python
import heapq

def dijkstra(graph, start_node):
    # Initialize distances to infinity, start_node to 0
    distances = {node: float('infinity') for node in graph}
    distances[start_node] = 0
```

```python
    # Priority Queue: stores tuples of (current_distance, current_node)
    # We start with the source node
    pq = [(0, start_node)]

    while pq:
        # 1. Pop the node with the smallest distance
        current_dist, current_node = heapq.heappop(pq)

        # Optimization: If we found a shorter way to this node already, skip
        if current_dist > distances[current_node]:
            continue

        # 2. Explore neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_dist + weight

            # 3. Relaxation: If new path is shorter, update and push to Q
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances

# Graph from our example
graph_example = {
    'A': {'B': 4, 'C': 1},
    'B': {'D': 1},
    'C': {'B': 2, 'D': 5},
    'D': {}
}

print(dijkstra(graph_example, 'A'))
# Output: {'A': 0, 'B': 3, 'C': 1, 'D': 4}
```

JavaScript Solution JavaScript does not have a built-in Priority Queue. In a real interview or production, you would use a library. For this example, we will use a simple array sort to simulate the priority queue (note: this makes the time complexity worse, but the logic is the same).

```javascript
function dijkstra(graph, startNode) {
    // Initialize distances
    let distances = {};
    for (let node in graph) {
        distances[node] = Infinity;
    }
```

```
    distances[startNode] = 0;

    // Simple array as a queue (simulating a priority queue)
    let pq = [[0, startNode]]; // [distance, node]

    while (pq.length > 0) {
        // 1. Sort to ensure we process the smallest distance first
        // (In a real scenario, a MinHeap data structure is O(log N))
        pq.sort((a, b) => a[0] - b[0]);

        // Pop the smallest
        let [currentDist, currentNode] = pq.shift();

        // Optimization check
        if (currentDist > distances[currentNode]) continue;

        // 2. Explore neighbors
        let neighbors = graph[currentNode];
        for (let neighbor in neighbors) {
            let weight = neighbors[neighbor];
            let newDist = currentDist + weight;

            // 3. Relaxation step
            if (newDist < distances[neighbor]) {
                distances[neighbor] = newDist;
                pq.push([newDist, neighbor]);
            }
        }
    }
    return distances;
}

const graph = {
    'A': {'B': 4, 'C': 1},
    'B': {'D': 1},
    'C': {'B': 2, 'D': 5},
    'D': {}
};

console.log(dijkstra(graph, 'A'));
// Output: { A: 0, B: 3, C: 1, D: 4 }
```

LeetCode Strategy: How to Spot It

Use Dijkstra's Algorithm when you see a problem with these characteristics:

1. **"Shortest Path" or "Minimum Cost":** You need to go from A to B (or A to all).
2. **Weighted Edges:** The connections between nodes have a cost (time, price, distance) and the costs are **not** all the same.
3. **No Negative Weights:** The costs are all 0 or positive. (If negative, Dijkstra fails).

Typical Problem phrasing:

- "Find the minimum time to send a signal…" (Network Delay Time).
- "Find the cheapest flight with K stops…"
- "Minimum effort path in a grid…"

Comparison with Other Algorithms

Algorithm	Best Used For	Key Difference
BFS (Breadth-First Search)	Unweighted Graphs	Only works if all edges have weight 1. It spreads out in layers. Dijkstra is essentially BFS with a Priority Queue.
Dijkstra	Weighted Graphs (Positive)	Smarter than BFS because it prioritizes "cheaper" paths first.
Bellman-Ford	Graphs with Negative Weights	Can handle negative edge weights, but is much slower (O(V * E)).

Single Source Shortest Path: Bellman Ford Algorithm

Single Source Shortest Path: Bellman-Ford Algorithm

Why do we need it? Imagine you are planning a road trip. Most algorithms (like Dijkstra's) are great at finding the fastest route, assuming all roads take positive time to travel. But what if a road actually *gives* you time back?

In computer science graphs, this is a "negative weight edge." This could represent a rebate, a time-travel mechanic in a game, or an energy gain in a physics simulation.

- **The Problem:** Standard algorithms break when edges have negative weights. They assume once a node is visited, the shortest path is set in stone.
- **The Solution:** Bellman-Ford handles these negative weights gracefully and can also detect "Negative Cycles" (infinite loops where you keep gaining benefits forever).

How it Works: The "Relaxation" Concept

Bellman-Ford is brute-force but smart. It relies on a concept called **Relaxation**.

Relaxing an Edge Think of a rubber band stretching between two cities. The "distance" is currently infinite (or very high). When we find a road, we "relax" the tension by shortening the estimated distance to the correct value.

The Rule: If the distance to node **u** + weight of edge **(u, v)** is less than the current distance to node **v**:

> Update distance to **v** = distance to **u** + weight of **(u, v)**

The Algorithm Steps:

1. Set the distance to the Start Node to 0 and all others to Infinity.
2. Repeat this process **V - 1** times (where V is the number of vertices/nodes):

- Check **every single edge** in the graph.
- If you can find a shortcut using that edge, update the distance.

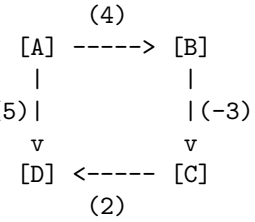
3. (Optional) Run one more time to check for negative cycles. If a distance changes, a negative cycle exists.

Visual Walkthrough

The Scenario: We want to go from Node **A** (Source) to all other nodes.

- Nodes (V): 4 (A, B, C, D)
- Edges (E): 5

The Graph: Notice the path `B -> C` has a weight of `-3`.



```
*Also, let's assume an edge C -> D with weight 4 for connectivity,
 but for this simple walkthrough, let's use:
 A->B (4)
 A->D (5)
 B->C (-3)
 D->C (2)
 C->D (Is typically needed to create a cycle, let's keep it simple:
 Let's just connect C->? or leave it.

 Let's refine the example to be perfectly clear with one update chain.*
```

Refined Example Graph:

- **A** is the Source.
- **A -> B** (Weight: 4)
- **A -> C** (Weight: 5)
- **B -> D** (Weight: -2)
- **C -> B** (Weight: -2)
- **D -> C** (Weight: 3)

```
    (4)        (-2)
[A] -----> [B] -----> [D]
 |          ^  \         |
 |         /    \        |
(5)|   (-2)/     \       |(3)
 |     /          \      |
 v    /            \     v
[C] -/              \-> [C] (Wait, let's keep topology simple)
```

Let's use this clean Graph:

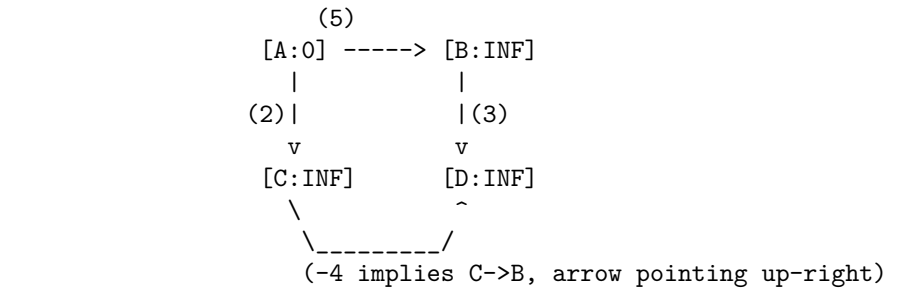
1. **A -> B** (Cost: 5)
2. **A -> C** (Cost: 2)
3. **C -> B** (Cost: -4) *The shortcut!*
4. **B -> D** (Cost: 3)

Goal: Shortest path from A to D.

Initialization We track distances in a list. `INF` means we haven't reached it yet.

```
DISTANCES:
 [A]: 0    (Source)
 [B]: INF
 [C]: INF
 [D]: INF

GRAPH STATE:
(Values in [] are current shortest distance from A)
```



Iteration 1 (Process all edges) We iterate through the edge list: `(A,B,5)`, `(A,C,2)`, `(C,B,-4)`, `(B,D,3)`.

1. **Check A -> B (weight 5):**

- Current B is INF.
- A is 0. 0 + 5 = 5.
- 5 < INF. **Update B to 5.**

2. **Check A -> C (weight 2):**

- Current C is INF.
- A is 0. 0 + 2 = 2.
- 2 < INF. **Update C to 2.**

3. **Check C -> B (weight -4):**

- Current B is 5.
- Current C is 2.
- Path A->C->B = 2 + (-4) = -2.
- -2 < 5. **Update B to -2.**

4. **Check B -> D (weight 3):**

- Current D is INF.
- Current B is -2 (just updated!).
- -2 + 3 = 1.
- 1 < INF. **Update D to 1.**

```
End of Iteration 1 Status:

DISTANCES:
 [A]: 0
 [B]: -2  (Updated via C)
 [C]: 2   (Updated via A)
 [D]: 1   (Updated via B)
```

```
VISUALIZATION:
      (5)
  [A:0] -----> [B:-2]
    |            |
 (2)|            |(3)
    v            v
  [C:2]        [D:1]
     \_________/
        (-4)
```

Note: In a real algorithm, the order of edge processing matters for how fast we converge, but after V-1 iterations, it is guaranteed to be correct.

Iteration 2 (Process all edges again) We do this again to ensure changes propagate.

1. **Check A -> B (5):** Is $0 + 5 < -2$? No.
2. **Check A -> C (2):** Is $0 + 2 < 2$? No.
3. **Check C -> B (-4):** Is $2 + (-4) < -2$? No (they are equal).
4. **Check B -> D (3):** Is $-2 + 3 < 1$? No.

No changes occurred. The algorithm has stabilized.

Final Result:

- Shortest path to B is -2 (Path: A->C->B).
- Shortest path to D is 1 (Path: A->C->B->D).

Complexity Analysis (Visualized)

Time Complexity: O(V * E) We run a loop `V` times. Inside that loop, we check `E` edges.

```
  | <------ V Iterations ------> |
  _______________________________
 |                               |
 |  [ Check Edge 1 ]             |
 |  [ Check Edge 2 ]             |
 |  [ ...          ]   (E Edges) |
 |  [ Check Edge E ]             |
 |_______________________________|

 Calculation: V vertices * E edges = O(V * E)
```

Space Complexity: O(V) We only need to store the distance array for the vertices.

```
  Storage Array:
  [ Slot 1 ] [ Slot 2 ] [ Slot 3 ] ... [ Slot V ]

  Calculation: Linear space relative to vertices = O(V)
```

Code Implementation

Python

```python
def bellman_ford(graph, V, src):
    # Step 1: Initialize distances
    # distances[i] will hold the shortest distance from src to i
    dist = [float("inf")] * V
    dist[src] = 0

    # Step 2: Relax all edges |V| - 1 times
    # graph is represented as a list of [u, v, w]
    for _ in range(V - 1):
        for u, v, w in graph:
            if dist[u] != float("inf") and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # Step 3: Check for negative weight cycles
    # If we can still relax an edge, then we have a negative cycle
    for u, v, w in graph:
        if dist[u] != float("inf") and dist[u] + w < dist[v]:
            print("Graph contains negative weight cycle")
            return None

    return dist

# Example Usage
# Edge format: [source, destination, weight]
edges = [
    [0, 1, 5],   # A->B
    [0, 2, 2],   # A->C
    [2, 1, -4],  # C->B
    [1, 3, 3]    # B->D
]
# Vertices: 0=A, 1=B, 2=C, 3=D
print(bellman_ford(edges, 4, 0))
# Output: [0, -2, 2, 1]
```

JavaScript

```javascript
function bellmanFord(edges, V, src) {
    // Step 1: Initialize distances
    let dist = new Array(V).fill(Infinity);
    dist[src] = 0;

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

    // Step 3: Check for negative cycles
    for (let edge of edges) {
        let [u, v, w] = edge;
        if (dist[u] !== Infinity && dist[u] + w < dist[v]) {
            console.log("Graph contains negative weight cycle");
            return null;
        }
    }

    return dist;
}

const edges = [
    [0, 1, 5],
    [0, 2, 2],
    [2, 1, -4],
    [1, 3, 3]
];
console.log(bellmanFord(edges, 4, 0));
```

LeetCode Strategy

When solving problems (like *Cheapest Flights Within K Stops* or *Network Delay Time*), think of Bellman-Ford when:

1. **Negative Costs:** The problem explicitly mentions costs can be negative (Dijkstra fails here).
2. **Restricted Hops:** The problem asks for the shortest path "using at most K edges."

- *Why?* The outer loop of Bellman-Ford controls the number of edges used. Iteration 1 gives shortest paths using 1 edge. Iteration 2 gives shortest paths using 2 edges. If you stop the loop at `K`, you get the answer for K-stops.

3. **Detecting Arbitrage:** If the problem asks if you can make infinite money/gain by looping through conversions, it is asking for a **Negative Cycle Detection**.

Comparison: Bellman-Ford vs. Dijkstra

Feature	Dijkstra	Bellman-Ford
Speed	Faster: `O(E + V log V)`	Slower: `O(V * E)`
Negative Edges?	**Fails** (Produces wrong answers)	**Works** (Handles them correctly)
Complexity	Uses a Priority Queue (Heap)	Uses simple Loops
Use Case	GPS, standard routing	Finance, Arbitrage, Constraints

All-Pairs Shortest Path: Floyd-Warshall Algorithm

Here is a detailed guide to the Floyd-Warshall Algorithm.

1. What is it and Why do we need it?

The Problem: Imagine you have a map of cities and roads. You don't just want to know the shortest path from City A to City B; you want to know the shortest path between **every single pair of cities** on the map.

The Solution: Floyd-Warshall is an "All-Pairs Shortest Path" algorithm.

- **Why not just use Dijkstra?** Dijkstra's algorithm is great for one start point. Running Dijkstra N times (once for every node) works, but Floyd-Warshall handles **negative edge weights** (unlike Dijkstra) and is much simpler to implement for dense graphs (lots of connections).
- **The Catch:** It detects negative cycles (a loop where the total weight is negative), but it cannot find the correct path if one exists.

---

2. How It Works (The Core Logic)

The algorithm relies on a simple idea: **The Intermediate Node.**

For any two nodes `i` and `j`, is it faster to go directly from `i` to `j`, or is it faster to go from `i` to an intermediate node `k`, and then from `k` to `j`?

The Formula: `New Distance = Distance(i to k) + Distance(k to j)`

If `New Distance` is smaller than the current `Distance(i to j)`, we update our map. We repeat this process trying **every** node as the intermediate node `k`.

---

3. Complexity Analysis

Time Complexity: O(N^3) We have three nested loops: one for the intermediate node `k`, one for the source `i`, and one for the destination `j`.

```
Complexity Visualization:
      (k)      (i)      (j)
    |-------|-------|-------|
       N    x   N   x   N    = N^3 operations

    If N = 10,  Ops ~ 1,000
    If N = 100, Ops ~ 1,000,000
    If N = 500, Ops ~ 125,000,000 (Limit for typical coding interviews)
```

Space Complexity: O(N^2) We need a 2D matrix to store the distances between every pair of nodes.

```
Space Visualization:
    [ ] [ ] [ ] [ ]
    [ ] [ ] [ ] [ ]  N rows * N columns = N^2
    [ ] [ ] [ ] [ ]
    [ ] [ ] [ ] [ ]
```

---

4. Step-by-Step Walkthrough

Let's find the shortest paths for this 4-node directed graph.

Input Graph (ASCII Diagram):

```
      (5)
 [0] -----> [1]
  |          |
  | (10)     | (3)
  v          v
 [3] <----- [2]
      (1)
```

- Node 0 goes to 1 (weight 5) and 3 (weight 10).
- Node 1 goes to 2 (weight 3).

- Node 2 goes to 3 (weight 1).
- Node 3 has no outgoing edges.
- **Goal:** Notice that `0 -> 3` is cost 10 directly. But `0 -> 1 -> 2 -> 3` is `5 + 3 + 1 = 9`. The algorithm should find this.

Step 0: Initialization (Distance Matrix) We create a table (matrix).

- If there is a direct edge, put the weight.
- If it's the same node (0 to 0), put 0.
- If there is no direct edge, put Infinity (INF).

Current State (Matrix D0):

```
      To: 0   1   2   3
From:
  0     [ 0,  5, INF, 10 ]
  1     [INF, 0,  3, INF ]
  2     [INF,INF, 0,  1  ]
  3     [INF,INF,INF, 0  ]
```

Step 1: Loop k = 0 (Using Node 0 as a bridge) We check: Can we shorten any path by going through Node 0?

- Logic: `D[i][j] = min(D[i][j], D[i][0] + D[0][j])`
- Since Node 0 only has outgoing edges and no incoming edges (except from itself), it cannot act as a shortcut for other nodes yet.

State after k=0: (No changes)

```
      To: 0   1   2   3
From:
  0     [ 0,  5, INF, 10 ]
  1     [INF, 0,  3, INF ]
  2     [INF,INF, 0,  1  ]
  3     [INF,INF,INF, 0  ]
```

Step 2: Loop k = 1 (Using Node 1 as a bridge) We check: Can we shorten paths by going through Node 1?

- We look for paths like `i -> 1 -> j`.
- Check `0 -> 2`:
- Current direct cost: INF
- Path via 1: `(0->1)` + `(1->2)` = `5 + 3 = 8`
- **Update!** `0 -> 2` becomes 8.

State after k=1:

```
      To: 0   1   2   3
From:
  0     [ 0,  5, *8*, 10 ]  <-- Updated (was INF)
```

```
  1     [INF, 0,  3, INF ]
  2     [INF,INF, 0,  1  ]
  3     [INF,INF,INF, 0  ]
```

Step 3: Loop k = 2 (Using Node 2 as a bridge) We check: Can we shorten paths by going through Node 2?

- We look for paths like `i -> 2 -> j`.
- Check `0 -> 3`:
- Current cost: 10 (direct)
- Path via 2: `(0->2)` + `(2->3)`
- Wait, what is `0->2`? We just updated it to 8 in the previous step!
- New cost: `8 + 1 = 9`.
- Is $9 < 10$? Yes. **Update!**
- Check `1 -> 3`:
- Current cost: INF
- Path via 2: `(1->2)` + `(2->3)` = `3 + 1 = 4`.
- **Update!**

State after k=2:

```
      To: 0   1   2   3
From:
  0     [ 0,  5,  8, *9* ]  <-- Updated (was 10)
  1     [INF, 0,  3, *4* ]  <-- Updated (was INF)
  2     [INF,INF, 0,  1  ]
  3     [INF,INF,INF, 0  ]
```

Step 4: Loop k = 3 (Using Node 3 as a bridge) Node 3 has no outgoing edges, so it cannot act as a bridge to anywhere.

Final State:

```
      To: 0   1   2   3
From:
  0     [ 0,  5,  8,  9  ]
  1     [INF, 0,  3,  4  ]
  2     [INF,INF, 0,  1  ]
  3     [INF,INF,INF, 0  ]
```

The algorithm is complete. We now have the shortest path between every pair.

---

5. Code Implementation

Python

```python
def floyd_warshall(n, edges):
    # Initialize matrix with Infinity
    dist = [[float('inf')] * n for _ in range(n)]

    # Distance to self is 0
    for i in range(n):
        dist[i][i] = 0

    # Set initial edge weights
    for u, v, w in edges:
        dist[u][v] = w

    # The Algorithm: 3 Nested Loops
    # k = intermediate node
    # i = source node
    # j = destination node
    for k in range(n):
        for i in range(n):
            for j in range(n):
                # If going through k is shorter, update dist[i][j]
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

# Example Usage
# Nodes: 0, 1, 2, 3
edges = [[0,1,5], [0,3,10], [1,2,3], [2,3,1]]
result = floyd_warshall(4, edges)
for row in result:
    print(row)
```

JavaScript

```javascript
function floydWarshall(n, edges) {
    // Create n x n matrix filled with Infinity
    let dist = Array.from({ length: n }, () => Array(n).fill(Infinity));

    // Distance to self is 0
    for (let i = 0; i < n; i++) {
        dist[i][i] = 0;
    }
```

```
    // Set initial edge weights
    for (let [u, v, w] of edges) {
        dist[u][v] = w;
    }

    // The Algorithm
    for (let k = 0; k < n; k++) {
        for (let i = 0; i < n; i++) {
            for (let j = 0; j < n; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
    return dist;
}
```

---

6. LeetCode Strategy & Comparison

When to use this on LeetCode? Look for the **Constraint Check**:

1. **Input size:** If `N <= 500`, O(N^3) solutions are usually acceptable (500^3 = 125,000,000 ops, which is borderline but often passes in Python/C++). If `N > 1000`, do not use this.
2. **Keywords:** "All pairs," "Transitive closure," "Find the city with the fewest neighbors within a threshold."
3. **Example Problem:** *Find the City With the Smallest Number of Neighbors at a Threshold Distance* (LeetCode 1334).

Comparison Table

Feature	Floyd-Warshall	Dijkstra	Bellman-Ford
Goal	All-Pairs Shortest Path	Single-Source Shortest Path	Single-Source Shortest Path
Complexity	**O(N^3)**	**O(E + V log V)**	**O(V * E)**
Negative Edges?	Yes	No	Yes
Implementation	Very Short (4-5 lines of logic)	Moderate (Priority Queue needed)	Moderate

Feature	Floyd-Warshall	Dijkstra	Bellman-Ford
Best For	Small graphs (N $<$ 500), Dense graphs	Large graphs, Sparse graphs	Detecting negative cycles

Minimum Spanning Tree: Prim's Algorithm

Here is a complete, easy-to-understand breakdown of Prim's Algorithm, showing exactly how it works without getting bogged down in unnecessary theory.

The Problem It Solves

Imagine you are a city planner who needs to connect a set of new towns with a power grid. Laying down power lines costs money, and the cost varies depending on the distance and terrain between specific towns.

You need to connect all the towns together so power can flow anywhere, but you want to spend the absolute minimum amount of money. You do not care if the path between Town A and Town Z is indirect, as long as they are connected.

This is the **Minimum Spanning Tree (MST)** problem. A "Spanning Tree" is a subset of paths that connects all points together without forming any redundant loops (cycles). The "Minimum" part means it has the lowest possible total cost.

What It Is & How It Works

Prim's Algorithm is a **greedy algorithm** used to find the Minimum Spanning Tree for a weighted, undirected graph.

It works by building the tree one step at a time, always making the cheapest local choice:

1. **Start anywhere:** Pick a random starting node. Mark it as "visited".
2. **Look around:** Look at all the edges connecting your visited nodes to unvisited nodes.
3. **Be greedy:** Pick the edge with the absolute lowest cost.
4. **Expand:** Add the new node to your "visited" group.
5. **Repeat:** Repeat steps 2-4 until every single node is visited.

Example Walkthrough

Let's use a simple graph with 4 nodes (A, B, C, D) and 5 connecting edges. We will build the MST step by step.

```
=== Step 0: The Initial Graph ===
Here are our nodes and the cost to travel between them.
    (4)
 A-------C
  \     / \
(1)\   /(2)\(3)
    \ /     \
     B-------D
        (5)
```

Step 1: Start at Node A We start at A. We look at all edges connecting A to unvisited nodes. We can go to B (cost 1) or C (cost 4). We greedily pick the cheapest: A-B.

```
=== Step 1: Pick A-B (Cost 1) ===

 A       C
  \
(1)\
    \
     B       D

Visited: {A, B}
Edges available to unvisited: A-C (4), B-C (2), B-D (5)
```

Step 2: Expand the Tree Now we look at all edges connecting our growing tree {A, B} to the unvisited nodes {C, D}. The cheapest available edge is B-C (cost 2). We pick it.

```
=== Step 2: Pick B-C (Cost 2) ===

 A       C
  \     /
(1)\   /(2)
    \ /
     B       D

Visited: {A, B, C}
Edges available to unvisited: A-C (4), B-D (5), C-D (3)
*Note: A-C (4) connects two already visited nodes, which would form a loop. We ignore it.*
```

Step 3: Finish the Tree We need to connect D. The available edges are B-D (cost 5) or C-D (cost 3). We pick C-D.

```
=== Step 3: Pick C-D (Cost 3) ===

 A       C
  \     / \
(1)\   /(2)\(3)
```

```
    \ /     \
     B       D

Visited: {A, B, C, D}
All nodes visited! Algorithm finished.
Total Minimum Cost = 1 + 2 + 3 = 6
```

Code Implementation

Python (Min-Heap Approach)

Using a priority queue (min-heap) is the most efficient way to continuously find the "cheapest available edge."

```python
import heapq

def prims_algorithm(n, edges):
    # Create an adjacency list: node -> [(weight, neighbor)]
    adj = {i: [] for i in range(n)}
    for u, v, weight in edges:
        adj[u].append((weight, v))
        adj[v].append((weight, u))

    mst_cost = 0
    visited = set()
    # Min-heap stores tuples of (weight, node). Start at node 0.
    min_heap = [(0, 0)]

    while len(visited) < n:
        weight, node = heapq.heappop(min_heap)

        # If we've already added this node to our tree, skip it
        if node in visited:
            continue

        visited.add(node)
        mst_cost += weight

        # Add all unvisited neighbors to the priority queue
        for next_weight, neighbor in adj[node]:
            if neighbor not in visited:
                heapq.heappush(min_heap, (next_weight, neighbor))

    return mst_cost
```

JavaScript (Array Approach)

JavaScript doesn't have a built-in priority queue. For smaller graphs, we can implement Prim's using an array to track the minimum edge connecting each node to the growing tree.

```
function primsAlgorithm(n, edges) {
    const adj = Array.from({length: n}, () => []);
    for (let [u, v, weight] of edges) {
        adj[u].push([v, weight]);
        adj[v].push([u, weight]);
    }

    const visited = new Set();
    const minEdge = Array(n).fill(Infinity);
    minEdge[0] = 0; // Start at node 0
    let mstCost = 0;

    for (let i = 0; i < n; i++) {
        let currNode = -1;
        let currMin = Infinity;

        // Find the unvisited node with the cheapest connection to our tree
        for (let j = 0; j < n; j++) {
            if (!visited.has(j) && minEdge[j] < currMin) {
                currMin = minEdge[j];
                currNode = j;
            }
        }

        if (currNode === -1) break; // Graph might be disconnected

        visited.add(currNode);
        mstCost += currMin;

        // Update the minimum connection cost for unvisited neighbors
        for (let [neighbor, weight] of adj[currNode]) {
            if (!visited.has(neighbor) && weight < minEdge[neighbor]) {
                minEdge[neighbor] = weight;
            }
        }
    }
    return mstCost;
}
```

Complexity Analysis

Time and Space Complexity (Prim's Algorithm)		
Approach	Time	Space
Array (JS Code)	O(V^2)	O(V + E)
Min-Heap (Python Code)	O(E log V)	O(V + E)
Fibonacci Heap	O(E + V log V)	O(V + E)
V = Number of Vertices (Nodes)		
E = Number of Edges		

Note: A Fibonacci heap gives the theoretically best time complexity, but is rarely used in standard interviews due to its complex implementation.

Application: LeetCode

Problem Suggestion: LeetCode 1584 - *Min Cost to Connect All Points* **How to think about it:** You are given an array of points on a 2D plane. The cost to connect two points is the Manhattan distance: `|xi - xj| + |yi - yj|`.

1. Treat every point as a node.
2. Because any point can connect to any other point, treat it as a fully connected graph where the "edge weight" is the Manhattan distance.
3. Apply Prim's algorithm starting from point 0. Keep track of the cheapest distance to connect unvisited points to your growing cluster. This is the exact scenario Prim's was built for!

Comparison With Other Algorithms

- **Prim's vs. Kruskal's Algorithm:** Both find the Minimum Spanning Tree. Kruskal's sorts *all* edges globally and picks them from cheapest to most expensive (skipping loops using a Union-Find data structure). Prim's builds a single tree outwards like a puddle expanding. Prim's is generally faster for dense graphs (lots of edges), while Kruskal's is great for sparse graphs.
- **Prim's vs. Dijkstra's Algorithm:** They look incredibly similar and both use priority queues. However, Dijkstra's finds the shortest path from a starting node to *all other nodes*. It keeps track of the total accumulated distance from the start. Prim's only cares about the shortest distance

from the *current tree edge* to the next node, ignoring how far away the starting node is.

Minimum Spanning Tree: Kruskal's Algorithm

Here is a detailed guide to Kruskal's Algorithm, designed to be easy to visualize and apply.

1. What is Kruskal's Algorithm and Why Do We Need It?

The Problem: Imagine you are a city planner. You have several houses (nodes) that need to be connected to the power grid. You can build power lines (edges) between them, but each line costs a different amount of money based on distance or terrain (weight).

The Goal: You need to connect **all** the houses so that everyone has power, but you want to spend the **minimum** amount of money possible. You do not need to create loops (redundant paths); you just need a single connected network.

The Solution: This "cheapest connected network" is called a **Minimum Spanning Tree (MST)**. Kruskal's Algorithm is a strategy to find this MST efficiently.

How It Works (The "Greedy" Strategy): Kruskal's is simple: it always picks the cheapest available option first.

1. **Sort:** List every possible connection from cheapest to most expensive.
2. **Select:** Start buying the cheapest connections.
3. **Check:** Before buying a connection, check: "Does this create a loop (cycle)?"

- If **No**: Buy it.
- If **Yes**: Skip it (because those houses are already connected indirectly, so this line is a waste of money).

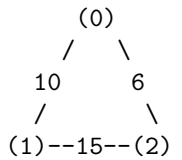
4. **Repeat:** Keep going until all houses are part of a single network.

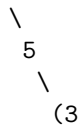
---

2. Visual Walkthrough

Let's trace this step-by-step using a graph with **4 Nodes (0, 1, 2, 3)**.

Initial Graph (with Edge Weights):





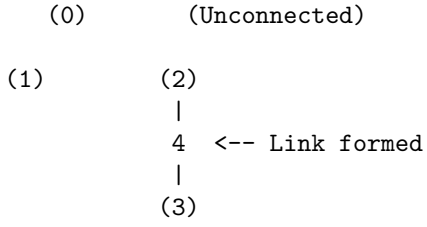
List of Edges: (0,1): 10 | (0,2): 6 | (1,2): 15 | (1,3): 5 | (2,3): 4

Step 1: Sort all edges by weight (Low to High)

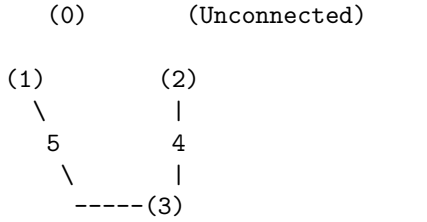
1. (2, 3) - Weight: 4
2. (1, 3) - Weight: 5
3. (0, 2) - Weight: 6
4. (0, 1) - Weight: 10
5. (1, 2) - Weight: 15

Step 2: Process Edges

Iteration A: Pick Edge (2, 3) [Weight 4] *Check:* Are 2 and 3 connected? No. *Action:* **Add Edge.**

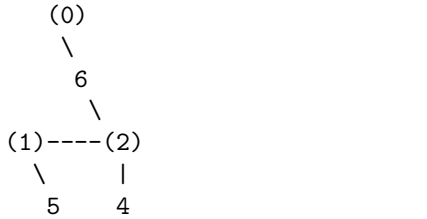


Iteration B: Pick Edge (1, 3) [Weight 5] *Check:* Are 1 and 3 connected? No. *Action:* **Add Edge.**



Current Network: 1-3-2 are all connected.

Iteration C: Pick Edge (0, 2) [Weight 6] *Check:* Are 0 and 2 connected? No. *Action:* **Add Edge.**



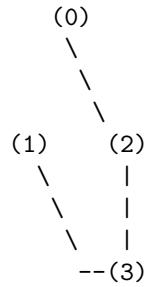
Current Network: All nodes (0, 1, 2, 3) are now connected!

Iteration D: Pick Edge (0, 1) [Weight 10] *Check:* Are 0 and 1 connected?

- 0 is connected to 2.
- 2 is connected to 3.
- 3 is connected to 1.
- **Yes**, they are already in the same group. Adding this would create a closed loop (0-2-3-1-0). *Action:* **Skip / Discard.**

Iteration E: Pick Edge (1, 2) [Weight 15] *Check:* Are 1 and 2 connected? Yes (via 3). *Action:* **Skip / Discard.**

Final Minimum Spanning Tree: Total Cost = 4 + 5 + 6 = **15**



---

3. Complexity Analysis

To implement the "Check" step efficiently (checking if nodes are already connected), we use a data structure called **Union-Find** (Disjoint Set Union).

Time Complexity: O(E log E) E = Number of Edges, V = Number of Vertices (Nodes).

Operation	Complexity
1. Sorting Edges	O(E log E) <-- The dominant step
2. Iterating Edges	O(E)
3. Union-Find Operations	O(alpha(V)) <-- Nearly constant time
TOTAL	O(E log E)

Note: Sorting is usually the most expensive part. O(E log E) is effectively the same as O(E log V) for simple graphs.

Space Complexity: O(V)

```
  [ Union-Find Arrays (Parent/Rank) ]
  | Node 0 | Node 1 | Node 2 | ... | Node V |  --> Requires O(V) space
```

---

4. Implementation Code

Here is the clean implementation using the Union-Find helper structure.

Python:

```python
class UnionFind:
    def __init__(self, n):
        # Initially, every node is its own parent
        self.parent = list(range(n))

    def find(self, node):
        # Find the representative (root) of the set
        if self.parent[node] != node:
            # Path compression: point directly to the root
            self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union(self, node1, node2):
        root1 = self.find(node1)
        root2 = self.find(node2)

        if root1 != root2:
            # Union: Attach one root to the other
            self.parent[root1] = root2
            return True # Union was successful
        return False # Cycle detected (already connected)

def kruskals_algorithm(num_nodes, edges):
    # edges format: [ [u, v, weight], ... ]

    # Step 1: Sort edges by weight (ascending)
    edges.sort(key=lambda x: x[2])

    mst = []
    min_cost = 0
    uf = UnionFind(num_nodes)
    edges_count = 0
```

```
    for u, v, weight in edges:
        # Step 2 & 3: Check cycle and Add
        if uf.union(u, v):
            mst.append([u, v, weight])
            min_cost += weight
            edges_count += 1

            # Optimization: Stop if we have V-1 edges (tree is complete)
            if edges_count == num_nodes - 1:
                break

    return min_cost, mst

# Example usage
# Nodes: 0, 1, 2, 3
graph_edges = [
    [0, 1, 10],
    [0, 2, 6],
    [0, 3, 5], # Adding an extra edge for example
    [1, 3, 15],
    [2, 3, 4]
]

cost, tree = kruskals_algorithm(4, graph_edges)
print(f"Minimum Cost: {cost}")
print(f"Edges in MST: {tree}")
```

JavaScript:

```
class UnionFind {
  constructor(n) {
    this.parent = Array.from({ length: n }, (_, i) => i);
  }

  find(node) {
    if (this.parent[node] !== node) {
      this.parent[node] = this.find(this.parent[node]);
    }
    return this.parent[node];
  }

  union(node1, node2) {
    const root1 = this.find(node1);
    const root2 = this.find(node2);

    if (root1 !== root2) {
```

```
      this.parent[root1] = root2;
      return true;
    }
    return false;
  }
}

function kruskalsAlgorithm(numNodes, edges) {
  // Step 1: Sort edges by weight
  edges.sort((a, b) => a[2] - b[2]);

  const uf = new UnionFind(numNodes);
  let minCost = 0;
  const mstEdges = [];
  let edgesCount = 0;

  for (const [u, v, weight] of edges) {
    // Step 2 & 3: Check cycle and Add
    if (uf.union(u, v)) {
      minCost += weight;
      mstEdges.push([u, v, weight]);
      edgesCount++;

      if (edgesCount === numNodes - 1) break;
    }
  }

  return { minCost, mstEdges };
}

// Example usage
const edges = [
  [0, 1, 10], [0, 2, 6], [0, 3, 5],
  [1, 3, 15], [2, 3, 4]
];

console.log(kruskalsAlgorithm(4, edges));
```

---

5. Application & Comparison

LeetCode Strategy: When you see a problem asking to "connect points" or "find the minimum cost to reach all nodes," think Kruskal's.

- **Specific Problem:** *Min Cost to Connect All Points* (LeetCode 1584).
- **Mental Model:** Treat every point as a node. Calculate the distance

between every pair of points to create your edges. Sort those distances and run Kruskal's.

Comparison with Prim's Algorithm:

Feature	Kruskal's Algorithm	Prim's Algorithm
Approach	**Edge-centric:** Sorts all edges and picks the smallest.	**Node-centric:** Grows a tree from a single starting node.
Graph Type	Better for **Sparse Graphs** (fewer edges).	Better for **Dense Graphs** (lots of edges).
Data Structure	Uses **Union-Find**.	Uses **Priority Queue** (Min-Heap).
Disconnected?	Can generate a "Forest" (multiple trees) if the graph isn't fully connected.	Only generates a single tree from the start node.

Strongly Connected Components: Tarjan Algorithm

Introduction to Tarjan's Algorithm

What it is: Tarjan's Algorithm is a highly efficient method used to find **Strongly Connected Components (SCCs)** in a directed graph.

What is an SCC? A Strongly Connected Component is a group of nodes where every node can reach every other node within that specific group.

Why do we need it?

1. **Cycle Detection:** If an SCC has more than one node, or a single node with a self-loop, it contains a cycle.
2. **Condensing Graphs:** It helps convert a messy graph with cycles into a "Directed Acyclic Graph" (DAG) of components, which is much easier to analyze.
3. **Social Networks:** Finding tightly knit groups of friends where everyone knows everyone else.

Core Concepts

To understand Tarjan's, you only need to track two numbers for every node you visit:

1. **ID (Discovery Time):** A unique number assigned to a node when you first visit it (0, 1, 2…).
2. **Low Link Value:** The lowest ID that a node can reach (including itself) using the path it is currently on.

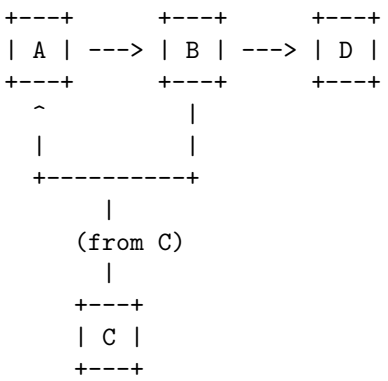
The Golden Rule: If a node's **Low Link Value** is equal to its **ID** after you have finished checking all its neighbors, that node is the "root" or beginning of a Strongly Connected Component. You then pop everything off the stack until you reach that node.

Visual Walkthrough

We will use a Depth First Search (DFS). We also use a **Stack** to keep track of the nodes currently in our recursion.

The Graph:

- Node **A** points to **B**
- Node **B** points to **C**
- Node **C** points back to **A** (creating a cycle)
- Node **B** also points to **D**



Step 1: Start at Node A We visit A. We assign it an ID of 0 and a Low Link of 0. We push it onto the stack.

```
Current Node: A
Stack: [A]

IDs:
A: 0

Low Links:
A: 0
```

```
Graph State:
(A) -> B -> D
 ^     |
 |     v
 +----(C)
```

Step 2: Move to Neighbor B From A, we go to B. Assign ID 1, Low Link 1. Push B to stack.

```
Current Node: B
Stack: [A, B]

IDs:
A: 0, B: 1

Low Links:
A: 0, B: 1

Graph State:
 A -> (B) -> D
 ^     |
 |     v
 +---- C
```

Step 3: Move to Neighbor C From B, we go to C. Assign ID 2, Low Link 2. Push C to stack.

```
Current Node: C
Stack: [A, B, C]

IDs:
A: 0, B: 1, C: 2

Low Links:
A: 0, B: 1, C: 2

Graph State:
 A ->  B -> D
 ^     |
 |     v
 +----(C)
```

Step 4: Check Neighbor of C (It is A) C points to A.

- Is A visited? **Yes.**
- Is A on the stack? **Yes.**

- **Action:** We update C's Low Link. We want the minimum of (Current Low Link, Neighbor's ID).
- Low Link (C) = min(2, 0) -> **0**.

Note: C has virtually "reached back" to the start.

```
Current Node: C (checking A)
Stack: [A, B, C]

IDs:
A: 0, B: 1, C: 2

Low Links:
A: 0, B: 1, C: 0  <-- UPDATED!

Graph State:
(A)<---(C) Connection found!
```

Step 5: Backtrack to B We finish with C. We return to B.

- **Action:** Update B's Low Link using C's Low Link.
- Low Link (B) = min(Low Link B, Low Link C)
- Low Link (B) = min(1, 0) -> **0**.

```
Current Node: B (returned from C)
Stack: [A, B, C]

IDs:
A: 0, B: 1, C: 2

Low Links:
A: 0, B: 0, C: 0  <-- UPDATED!
      ^
      |
B inherits 0 from C
```

Step 6: Visit Neighbor D (from B) B also points to D. We visit D. Assign ID 3, Low Link 3. Push D to stack.

```
Current Node: D
Stack: [A, B, C, D]

IDs:
A: 0, B: 1, C: 2, D: 3

Low Links:
A: 0, B: 0, C: 0, D: 3
```

```
Graph State:
 A ->  B -> (D)
```

Step 7: D has no neighbors D is a dead end. We check the Golden Rule:

- Does ID(D) == Low Link(D)?
- 3 == 3. **YES.**
- **Action:** D is the start of an SCC. Pop elements off the stack until we hit D.

Result: Found SCC **{D}**.

```
Current Node: D
Stack: [A, B, C]  <-- D popped

Found SCC #1: { D }
```

Step 8: Backtrack to B (Finish B) We return to B. We update B's Low Link based on D, but since D was an SCC, we generally ignore it or take the min (which doesn't change anything here because D's low link was higher). B's Low Link is 0. B's ID is 1.

- Does 1 == 0? **No.**
- B is not the root of an SCC yet. Backtrack.

Step 9: Backtrack to A (Finish A) We return to A. Update A's Low Link based on B.

- Low Link (A) = min(Low Link A, Low Link B)
- Low Link (A) = min(0, 0) -> **0**.

Now we check the Golden Rule for A:

- Does ID(A) == Low Link(A)?
- 0 == 0. **YES.**
- **Action:** A is the root of an SCC. Pop everything off the stack until we hit A.

Result: Pop C, Pop B, Pop A. Found SCC **{C, B, A}**.

```
Current Node: A
Stack: []  <-- C, B, A popped

Found SCC #2: { C, B, A }
```

Complexity Analysis

Time Complexity: O(V + E)

- V = Vertices (Nodes), E = Edges.
- We visit every node once and traverse every edge once.

```
TIME SPENT
|
| [ Node Visits (V) ] + [ Edge Checks (E) ]
|_____________________________________
     Linear Time (Very Fast)
```

Space Complexity: O(V)

- We store the recursion stack and the ID/Low arrays for every node.

```
MEMORY USED
|
| [ Recursive Stack ]
| [ IDs Array       ]
| [ Low Link Array  ]
|____________________
     Linear Space
```

Code Implementation

Python

```python
class TarjanSolver:
    def __init__(self, graph):
        self.graph = graph
        self.ids = {}
        self.low = {}
        self.on_stack = {}
        self.stack = []
        self.id_counter = 0
        self.scc_count = 0
        self.result = []

    def find_sccs(self):
        # Initialize
        for node in self.graph:
            self.ids[node] = -1
            self.low[node] = 0
            self.on_stack[node] = False

        # Run DFS for every unvisited node
        for node in self.graph:
            if self.ids[node] == -1:
                self.dfs(node)
```

```
        return self.result

    def dfs(self, at):
        self.stack.append(at)
        self.on_stack[at] = True
        self.ids[at] = self.low[at] = self.id_counter
        self.id_counter += 1

        # Visit neighbors
        for to in self.graph.get(at, []):
            if self.ids[to] == -1:
                # If neighbor is not visited, visit it
                self.dfs(to)
                # On return, propagate low link value (min logic)
                self.low[at] = min(self.low[at], self.low[to])
            elif self.on_stack[to]:
                # If neighbor is on stack, it's a back-edge
                self.low[at] = min(self.low[at], self.ids[to])

        # Golden Rule: If we are at the root of an SCC
        if self.ids[at] == self.low[at]:
            current_scc = []
            while True:
                node = self.stack.pop()
                self.on_stack[node] = False
                current_scc.append(node)
                if node == at:
                    break
            self.result.append(current_scc)
            self.scc_count += 1

# Example Usage
graph = {
    'A': ['B'],
    'B': ['C', 'D'],
    'C': ['A'],
    'D': []
}
solver = TarjanSolver(graph)
print(solver.find_sccs())
# Output: [['D'], ['C', 'B', 'A']]
```

JavaScript

```
class TarjanSolver {
    constructor(graph) {
        this.graph = graph;
        this.ids = {};
        this.low = {};
        this.onStack = {};
        this.stack = [];
        this.idCounter = 0;
        this.result = [];
    }

    findSCCs() {
        // Initialize keys
        Object.keys(this.graph).forEach(node => {
            this.ids[node] = -1;
            this.onStack[node] = false;
        });

        Object.keys(this.graph).forEach(node => {
            if (this.ids[node] === -1) {
                this.dfs(node);
            }
        });
        return this.result;
    }

    dfs(at) {
        this.stack.push(at);
        this.onStack[at] = true;
        this.ids[at] = this.low[at] = this.idCounter++;

        const neighbors = this.graph[at] || [];
        for (let to of neighbors) {
            if (this.ids[to] === -1) {
                this.dfs(to);
                this.low[at] = Math.min(this.low[at], this.low[to]);
            } else if (this.onStack[to]) {
                this.low[at] = Math.min(this.low[at], this.ids[to]);
            }
        }

        if (this.ids[at] === this.low[at]) {
            const currentSCC = [];
            let node;
            do {
                node = this.stack.pop();
```

```
                this.onStack[node] = false;
                currentSCC.push(node);
            } while (node !== at);
            this.result.push(currentSCC);
        }
    }
}
```

LeetCode Application

Problem Idea: "Critical Connections in a Network" (LeetCode 1192).

The Scenario: You have a network of servers. You need to find "critical connections" (bridges). A bridge is an edge that, if removed, makes some servers unable to reach others.

How to think about it using Tarjan's: Tarjan's algorithm is naturally built to find cycles.

1. If an edge is part of a cycle (an SCC), removing it usually doesn't break the graph because there is another way around the loop.
2. If an edge is NOT part of a cycle, it is a **Bridge**.

In the `dfs` function, right after you visit a neighbor `to` and return:

```
self.dfs(to)
self.low[at] = min(self.low[at], self.low[to])
if self.low[to] > self.ids[at]:
    # This means 'to' cannot reach back to 'at' or anything before 'at'.
    # Therefore, the edge (at -> to) is a BRIDGE.
```

Comparison

Feature	Tarjan's Algorithm	Kosaraju's Algorithm
Passes Required	**1 DFS pass** (Faster in practice)	**2 DFS passes** (Slightly slower)
Complexity	O(V + E)	O(V + E)
Space	O(V)	O(V)
Ease of Code	Harder (needs low-link logic)	Easier (standard DFS logic twice)
Stack Usage	Explicit stack handling	Implicit recursion stack

Verdict: Use Tarjan's if you need raw speed and are comfortable with the "Low Link" concept. Use Kosaraju's if you want something easier to memorize and code quickly during an interview.