Graphs

LeetCode 133: Clone Graph

Here is a breakdown of how a senior engineer (L5/L6) would approach **Leet-Code 133: Clone Graph**.

At a senior level, we look beyond just "getting it to pass." We care about code maintainability, handling cycles (infinite loops), memory management, and how this pattern applies to real-world systems like object serialization or social graph replication.

---

1. Problem Explanation

The Goal: We are given a reference to a node in a **connected undirected graph**. We need to return a **deep copy** (clone) of the graph.

What is a Deep Copy? Imagine you have a blueprint of a house.

- **Shallow Copy:** You photocopy the blueprint. You have two papers, but they refer to the *same* physical house. If someone paints the kitchen red in the first blueprint, it's just a reference to the existing kitchen.
- **Deep Copy:** You build a brand new house identical to the first one. It looks the same, but it is located at a different address. Painting the kitchen in the new house does *not* affect the old house.

The Input Structure:

- `val`: An integer (unique identifier).
- `neighbors`: A list of Node objects.

The Challenge: The graph can have **cycles**. Node A points to B, and B points back to A. If we blindly copy A, then try to copy A's neighbors (B), and then copy B's neighbors (A)... we will enter an infinite loop and crash the program (Stack Overflow).

Visualizing the Problem:

```
ORIGINAL GRAPH (Input)          TARGET GRAPH (Output)
----------------------          ---------------------
Memory Addr: 0x100              Memory Addr: 0x999
+----------+                    +----------+
| Val: 1   |                    | Val: 1   |
| Neighbors:|--------+          | Neighbors:|--------+
| [0x200]   |        |          | [0x888]   |        |
+----------+         |          +----------+         |
     |               |               |               |
     | (link)        |               | (link)        |
```

```
         v                 v                      v                 v
Memory Addr: 0x200     |        Memory Addr: 0x888     |
+----------+           |        +----------+           |
| Val: 2   |           |        | Val: 2   |           |
| Neighbors:| <------+        | Neighbors:| <------+
| [0x100]  |                    | [0x999]  |
+----------+                    +----------+

NOTICE:
1. The structure (Val 1 connected to Val 2) is identical.
2. The Memory Addresses are completely different.
3. The new Node 1 points to the NEW Node 2, not the OLD Node 2.
```

---

2. Solution Explanation

To solve this, we can use **Graph Traversal** (either BFS or DFS). The specific algorithm matters less than the **state management**.

The Core Strategy: The Hash Map To prevent cycles and infinite loops, we need a "memory" of nodes we have already copied. We use a **Hash Map** (Dictionary) that maps: `Original Node -> Cloned Node`

The Algorithm (DFS Approach):

1. **Start at the input node.**
2. **Check the Map:** Have we already cloned this node?

- **Yes:** Return the existing clone immediately (this closes the cycle).
- **No:** Create a new node. **Add it to the map immediately** (before processing neighbors).

3. **Process Neighbors:**

- Iterate through the original node's neighbors.
- Recursively call the function for each neighbor.
- Add the result of that recursive call to the *new* node's neighbor list.

4. **Return the new node.**

Visual Walkthrough:

Let's clone `1 -- 2`. (1 connects to 2, 2 connects to 1).

Step 1: Start at Node 1

- Is 1 in Map? No.
- Create `New_1`.
- **Map:** { `Old_1` : `New_1` }
- Now loop through Old_1's neighbors. Found: `Old_2`.

Step 2: Recurse on Node 2

- Is 2 in Map? No.
- Create `New_2`.
- **Map: { Old_1 : New_1, Old_2 : New_2 }**
- Loop through Old_2's neighbors. Found: `Old_1`.

Step 3: Recurse on Node 1 (Again)

- Is 1 in Map? **YES!**
- We see `Old_1` in the map.
- **Return** `New_1` immediately.

Step 4: Backtrack to Node 2

- `New_2` receives `New_1` returned from Step 3.
- Add `New_1` to `New_2.neighbors`.
- `New_2` is done. **Return** `New_2`.

Step 5: Backtrack to Node 1

- `New_1` receives `New_2` returned from Step 4.
- Add `New_2` to `New_1.neighbors`.
- `New_1` is done. **Return** `New_1`.

Final Map State:

```
   HASH MAP (The "Visited" Tracker)
+-----------------+-----------------+
| Key (Original)  | Value (Clone)   |
+-----------------+-----------------+
| Ref(Old_Node_1) | Ref(New_Node_1) |
| Ref(Old_Node_2) | Ref(New_Node_2) |
+-----------------+-----------------+
```

---

3. Time and Space Complexity Analysis

Time Complexity: O(N + M)

- `N` is the number of nodes (Vertices).
- `M` is the number of edges.

Space Complexity: O(N)

- We store `N` entries in the Hash Map.
- The recursion stack (DFS) can go up to `O(N)` depth in a simplified graph (like a line).

Visual Derivation (The "Cost" Model):

```
VISUAL COST ANALYSIS

1. NODE VISITS (The Vertices - V)
   We visit every node exactly once to create it.

   (Node 1) -> Create -> Map it
   (Node 2) -> Create -> Map it
   (Node 3) -> Create -> Map it

   Cost so far: O(N)

2. EDGE TRAVERSALS (The Edges - E)
   For every node, we iterate over its neighbor list.
   In an undirected graph, every edge connects two nodes (A-B).

   When processing A: We look at the link to B.
   When processing B: We look at the link to A.

   Total operations = 2 * (Number of Edges)

   Cost: O(E)

3. TOTAL TIME
   Total = Cost(Nodes) + Cost(Edges)
   Total = O(N + E)

4. SPACE (The Memory Stack)

   Map Storage:
   [ Node 1 | Clone 1 ]
   [ Node 2 | Clone 2 ]
   ...
   [ Node N | Clone N ]  <-- Grows linearly with N

   Recursion Stack (Worst Case - Line Graph):
   Call(1) -> Call(2) -> Call(3) -> ... Call(N)
   Depth = N

   Total Space = O(N)
```

---

4. Solution Code

Python Solution   Using `dfs` with a dictionary for mapping.

```python
"""
Definition for a Node.
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []
"""


class Solution:
    def cloneGraph(self, node: 'Node') -> 'Node':
        # Edge Case: If input is empty, return None
        if not node:
            return None

        # This dictionary maps the ORIGINAL node to the CLONED node.
        # It acts as both our "visited" set and our storage for clones.
        old_to_new = {}

        def dfs(current_node):
            # Base Case: If we have already cloned this node,
            # return the clone to ensure we connect to the existing new node
            # rather than creating a duplicate (which would break cycles).
            if current_node in old_to_new:
                return old_to_new[current_node]

            # 1. Create the copy (Clone)
            # We copy the value, but initialize neighbors as empty list
            copy = Node(current_node.val)

            # 2. Add to Map IMMEDIATELY
            # Crucial Step: We must map it before entering the recursive calls
            # to handle self-loops or cycles back to this node.
            old_to_new[current_node] = copy

            # 3. Populate Neighbors
            for neighbor in current_node.neighbors:
                # Recursively call dfs. It will either create a new neighbor
                # or return an existing one from the map.
                cloned_neighbor = dfs(neighbor)
                copy.neighbors.append(cloned_neighbor)

            return copy

        return dfs(node)
```

JavaScript Solution Using `DFS` and a `Map` object (which allows using Objects as keys).

```javascript
/**
 * Definition for Node.
 * class Node {
 * val: number
 * neighbors: Node[]
 * constructor(val?: number, neighbors?: Node[]) {
 * this.val = (val===undefined ? 0 : val)
 * this.neighbors = (neighbors===undefined ? [] : neighbors)
 * }
 * }
 */

/**
 * @param {Node} node
 * @return {Node}
 */
var cloneGraph = function(node) {
    if (!node) return null;

    // Map to store mapping from Original Node -> Cloned Node
    const visited = new Map();

    const dfs = (currentNode) => {
        // Base Case: If already visited, return the stored clone
        if (visited.has(currentNode)) {
            return visited.get(currentNode);
        }

        // 1. Create the Clone
        const clone = new Node(currentNode.val);

        // 2. Register in Map
        visited.set(currentNode, clone);

        // 3. Iterate and clone neighbors
        for (let neighbor of currentNode.neighbors) {
            // Recursively clone the neighbor and add to the current clone's list
            clone.neighbors.push(dfs(neighbor));
        }

        return clone;
    };
```

```
    return dfs(node);
};
```

Real-World Interview Variations (Indirect)

Top companies rarely ask "Clone this Graph" directly anymore. They wrap it in a system design or practical engineering context.

1. The "Deep Copy JSON" Problem (Google / Meta)   The Question: "Write a function to deep copy a complex object in JavaScript/Python that might contain circular references."

- **Why it's the same:** A JSON object with circular references is just a directed graph. `JSON.parse(JSON.stringify(obj))` fails here (throws an error).
- **Solution:** You implement `cloneGraph`.
- Nodes = Objects.
- Neighbors = Object properties/values.
- Map = Tracks objects to prevent infinite recursion on circular refs.

2.   The "Dependency Build Order" or "Package Manager" (Bloomberg / Google) The Question: "You have a build system where Library A depends on B, and B depends on A. We need to replicate this dependency tree into a new environment (e.g., a sandbox)."

- **Why it's the same:** The dependencies form a graph. To "replicate" the environment, you are cloning the graph structure.
- **Twist:** Often combined with **Topological Sort** to detect if the cycle is "legal" or "illegal" before cloning.

3. Social Graph Replication (System Design Context)   The Question: "We are sharding a database. We need to move a user and their immediate connection graph to a new shard. How do you copy the data?"

- **Why it's the same:** Users are nodes, Friendships are edges.
- **Solution:** This is `cloneGraph` but distributed.
- You can't just use memory pointers. The "Map" becomes a lookup table of `Old_User_ID -> New_Shard_User_ID`.
- You use BFS (Breadth-First Search) instead of DFS usually, to copy "1st degree connections" first before moving deeper.

Terminology Recap   Deep Copy vs. Shallow Copy:

- *Shallow:* Copying the container, but the contents inside point to the old data.

7

- *Deep:* Recursively copying the container and everything inside it, so no links remain to the old data. `Clone Graph` is the definition of a Deep Copy algorithm.

Adjacency List:

- The way the graph is likely represented in memory. A list (or map) where every node stores a list of the nodes it connects to.

207. Course Schedule

An L5/L6 engineer looks at LeetCode 207 not just as a "graph problem" but as a **Dependency Resolution** problem. This is the core logic behind build systems (like Bazel at Google), package managers (npm/pip), and task schedulers (Airflow).

The fundamental question we are answering is: **"Is there a circular dependency?"**

Here is the solution broken down with the requested depth and visualizations.

---

1. Problem Explanation

You have `numCourses` (let's call them **Tasks**) labeled `0` to `numCourses - 1`. You are given an array `prerequisites` where `[A, B]` means "To take course A, you must first finish course B."
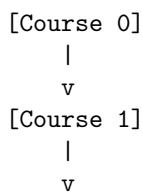
- **Dependency:** B -> A (B comes before A).

Goal: Return `true` if it is possible to finish all courses. Return `false` if it is impossible.

Why would it be impossible? If there is a **Cycle** (a deadlock).

- To do A, you need B.
- To do B, you need C.
- To do C, you need A.
- You can never start any of them.
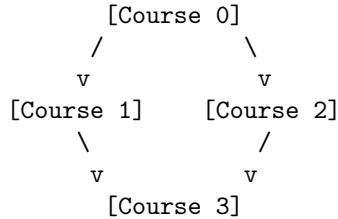
Visualizing the Problem Scenario A: A Valid Schedule (Linear) Prereqs: `[1, 0]` (0 before 1), `[2, 1]` (1 before 2)

```
[Course 0]
    |
    v
[Course 1]
    |
    v
```

```
               [Course 2]
```
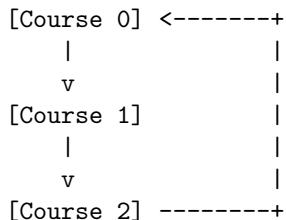
Result: TRUE. You can do 0 -> 1 -> 2.

Scenario B: A Valid Schedule (Branching) Prereqs: [1, 0], [2, 0], [3, 1], [3, 2]

```
           [Course 0]
         /           \
        v             v
   [Course 1]     [Course 2]
        \             /
         v           v
           [Course 3]
```

Result: TRUE. Order: 0 -> 1 -> 2 -> 3.

Scenario C: The Deadlock (Cycle) Prereqs: [1, 0], [2, 1], [0, 2]

```
      [Course 0] <-------+
          |              |
          v              |
      [Course 1]         |
          |              |
          v              |
      [Course 2] --------+
```

Result: FALSE. No matter where you start, you hit a wall.

---

2. Solution Explanation: Kahn's Algorithm (Topological Sort)

An L6 engineer often prefers **Kahn's Algorithm** (BFS approach) over simple DFS recursion for this specific problem because:

1. It is **Iterative** (no stack overflow risk on massive graphs).
2. It intuitively mimics how we solve this in real life: "Do the tasks with no requirements first."

The Concept: "Peeling the Onion" We look for courses that have **Zero Prerequisites** (Indegree = 0).

1. Add them to our "Done" list.
2. "Remove" them from the graph.
3. When we remove them, their neighbors might now have zero prerequisites.
4. Repeat.
5. If we processed all courses, return `True`. If courses remain stuck in the graph (because they are waiting on each other), return `False`.

Step-by-Step Visualization   Input: 4 Courses (0, 1, 2, 3) **Prereqs:** `[1, 0]`, `[2, 1]`, `[3, 2]`, `[1, 3]` (Note the cycle between 1-2-3)

Step 1: Build the Graph & Count Indegrees

- **Indegree:** How many arrows point AT me? (How many prereqs do I have?)

```
Data Structure State:

Adjacency List (Who depends on me?):
0 -> [1]
1 -> [2]
2 -> [3]
3 -> [1]  <-- The cycle is 1->2->3->1

Indegree Array (How many prereqs do I have?):
[Course 0]: 0  (Free to take!)
[Course 1]: 2  (Needs 0 and 3)
[Course 2]: 1  (Needs 1)
[Course 3]: 1  (Needs 2)
```

Step 2: Initialize the Queue Put all nodes with `Indegree == 0` into a Queue.

```
Queue: [ 0 ]
Count of Finished Courses: 0
```

Step 3: Process the Queue

Iteration A: Pop 0. Increment `Finished Count` to 1. Look at neighbors of 0: It is `1`. Reduce `Indegree` of 1 by 1.

```
Current Graph State (Mental Model):
(0 is "done", so we cut the link 0->1)

Indegree Array Updates:
[Course 1]: Was 2, now 1.

Is Indegree[1] == 0? NO.

Queue: [] (Empty)
```

Step 4: Conclusion The Queue is empty. The algorithm stops. `Finished Count` = 1. Total Courses = 4.

1 != 4. Result: False (Cycle detected). Nodes 1, 2, and 3 are stuck in a cycle. None of them ever reached Indegree 0.

Now, let's look at a Successful Run.

Input: [1, 0], [2, 1], [3, 2] (Line: 0->1->2->3)

```
Step 1: Indegrees
0: 0
1: 1 (Needs 0)
2: 1 (Needs 1)
3: 1 (Needs 2)

Queue: [0]

----------------------------------------

Step 2: Pop 0
Neighbor: 1.
Reduce Indegree[1] to 0.
Push 1 to Queue.

Queue: [1]
Finished: 1

----------------------------------------

Step 3: Pop 1
Neighbor: 2.
Reduce Indegree[2] to 0.
Push 2 to Queue.

Queue: [2]
Finished: 2

----------------------------------------

Step 4: Pop 2
Neighbor: 3.
Reduce Indegree[3] to 0.
Push 3 to Queue.

Queue: [3]
Finished: 3

----------------------------------------

Step 5: Pop 3
Neighbor: None.
Queue: []
Finished: 4
```

```
Result: 4 == 4 -> TRUE.
```

————————————————————

3. Time and Space Complexity Analysis

Time Complexity: O(V + E)

- **V**: Number of Vertices (Courses)
- **E**: Number of Edges (Prerequisites)

Visual Derivation: We do not use complex math. We simply count operations based on the graph structure.

```
        [A] -----> [B]
         |          |
         |          v
         +-------> [C]
```

```
1. Build Graph:
   We touch every prerequisite pair once to build the adjacency list.
   Cost: O(E)

2. Initialize Indegrees:
   We iterate through the list of courses once.
   Cost: O(V)

3. Processing the Queue (The "BFS"):
   - Each Course (Vertex) enters the Queue at most ONCE.
     Cost: O(V)
   - When we pop a Course, we look at its outgoing arrows (Edges).
     We "cut" (traverse) every arrow exactly ONCE in the entire lifecycle.
     Cost: O(E)

Total Work = Build(E) + Init(V) + ProcessNodes(V) + ProcessEdges(E)
           = O(2V + 2E)
           = O(V + E)
```

Space Complexity: O(V + E)

```
Storage Requirements:

1. Adjacency List (The Graph Map):
   Stores every node and every outgoing arrow.
   [ 0 ] -> [ 1, 2 ]
   [ 1 ] -> [ 3 ]
   ...
```

```
    Space: O(V + E)
```

2. Indegree Array:
   An integer for every course.
   [0, 1, 1, 2 ...]
   Space: O(V)

3. Queue:
   In the worst case (no dependencies), all nodes are in the queue at once.
   Space: O(V)

Total Space = O(V + E)

---

4. Solution Code

Python (Production Style)

```python
from collections import deque

class Solution:
    def canFinish(self, numCourses: int, prerequisites: list[list[int]]) -> bool:
        """
        Determines if a valid schedule exists using Kahn's Algorithm.

        Args:
            numCourses: Total number of courses (0 to numCourses-1).
            prerequisites: List of [course, dependency] pairs.

        Returns:
            True if all courses can be finished, False otherwise.
        """

        # 1. Initialize Graph and Indegree Array
        # adj: maps a prerequisite to the courses that unlock after it
        # indegree: maps a course to the number of prereqs it still needs
        adj = {i: [] for i in range(numCourses)}
        indegree = [0] * numCourses

        # 2. Build the Graph
        # Pair is [course, dependency], so direction is dependency -> course
        for course, dependency in prerequisites:
            adj[dependency].append(course)
            indegree[course] += 1

        # 3. Initialize Queue with independent courses (indegree == 0)
```

```python
        queue = deque()
        for i in range(numCourses):
            if indegree[i] == 0:
                queue.append(i)

        # 4. Process the Graph (BFS)
        courses_finished = 0

        while queue:
            # Pop a course we can take now
            current = queue.popleft()
            courses_finished += 1

            # "Unlock" neighbors
            for neighbor in adj[current]:
                indegree[neighbor] -= 1

                # If neighbor has no more prereqs, add to queue
                if indegree[neighbor] == 0:
                    queue.append(neighbor)

        # 5. Check if we finished all courses
        return courses_finished == numCourses
```

JavaScript (Production Style)

```javascript
/**
 * @param {number} numCourses
 * @param {number[][]} prerequisites
 * @return {boolean}
 */
var canFinish = function(numCourses, prerequisites) {
    // 1. Initialize Graph and Indegree Array
    // We use an Array of Arrays for the adjacency list for performance
    const adj = new Array(numCourses).fill(0).map(() => []);
    const indegree = new Array(numCourses).fill(0);

    // 2. Build the Graph
    // prereq entry: [target, source] -> source points to target
    for (const [course, dependency] of prerequisites) {
        adj[dependency].push(course);
        indegree[course]++;
    }

    // 3. Initialize Queue with independent courses
```

```javascript
    const queue = [];
    for (let i = 0; i < numCourses; i++) {
        if (indegree[i] === 0) {
            queue.push(i);
        }
    }

    // 4. Process the Graph
    let coursesFinished = 0;

    while (queue.length > 0) {
        // Shift is O(n) in worst case JS arrays, but amortized O(1) in modern engines
        // or effectively O(1) if we use a pointer for the head.
        // For interview strictness, simple array.shift() is usually accepted
        // unless N is massive.
        const current = queue.shift();
        coursesFinished++;

        const neighbors = adj[current];
        for (let i = 0; i < neighbors.length; i++) {
            const neighbor = neighbors[i];

            // "Remove" the edge
            indegree[neighbor]--;

            // If neighbor becomes free, add to queue
            if (indegree[neighbor] === 0) {
                queue.push(neighbor);
            }
        }
    }

    // 5. Result
    return coursesFinished === numCourses;
};
```

Note 1: Terminology Breakdown

- **DAG (Directed Acyclic Graph):** A directed graph with no cycles. If the graph is a DAG, we can finish the courses. If it is not a DAG, we cannot.
- **Topological Sort:** A linear ordering of vertices such that for every directed edge from u to v, vertex u comes before v in the ordering. This solution effectively attempts to build a Topological Sort. If the sort length

15

equals the number of nodes, it's valid.
- **Indegree:** The number of edges arriving at a vertex. In this context: "Number of unmet prerequisites."

Note 2: Real World & Indirect Interview Variations

At companies like Google, Meta, or Bloomberg, they rarely ask "Solve Course Schedule." They wrap it in a real-world scenario.

1. **The "Build System" Variation (Google / DevOps focus)**

- **Question:** "We have a list of C++ libraries to compile. Some libraries depend on others. In what order should we compile them to ensure no errors? Also, detect if there is a circular dependency."
- **Solution:** This IS Course Schedule.
- Libraries = Nodes.
- Dependencies = Edges.
- If `canFinish` is true, the order in which you popped items from the Queue is your **Compilation Order**.

2. **The "Deadlock Detection" Variation (Meta / OS focus)**

- **Question:** "We have N processes and N resources. Process A is holding Resource 1 and waiting for Resource 2. Process B is holding Resource 2 and waiting for Resource 1. How do you detect this programmatically?"
- **Solution:** Construct a "Wait-For Graph".
- Nodes = Processes.
- Edge A -> B means "A is waiting for B to release a resource."
- Run the Cycle Detection (as shown above). If `False`, you have a Deadlock.

3. **The "Excel Sheet Calculation" Variation (Bloomberg / Finance)**

- **Question:** "You have an Excel sheet where Cell A1 = B1 + C1. But B1 might depend on A1. How do you evaluate the sheet?"
- **Solution:**
- Cells = Nodes.
- Formulas = Edges.
- You must perform a Topological Sort to decide which cell to calculate first. If you detect a cycle, you throw a "Circular Reference Error" (just like Excel does).

417: Pacific Atlantic Water Flow

Here is a breakdown of how a Senior (L5/L6) Engineer would approach **Leet-Code 417: Pacific Atlantic Water Flow**.

At this level, the focus shifts from "getting code to run" to **optimizing the approach (system thinking)** and **communication (clarity)**. The core insight here distinguishes a junior brute-force solution from a scalable senior solution.

---

1. Problem Explanation

Imagine a rectangular island.

- The **Pacific Ocean** touches the **Left** and **Top** edges.
- The **Atlantic Ocean** touches the **Right** and **Bottom** edges.
- The island is a grid of square cells, where each number represents the **height** of the terrain at that spot.

The Rule of Physics: Water can only flow from a cell to a neighbor (up, down, left, right) if the neighbor's height is **less than or equal to** the current cell's height. (Water flows downhill or across flats).

The Goal: Find the coordinates of all cells where, if you poured water on them, that water could eventually flow to **BOTH** the Pacific and Atlantic oceans.

Visualization of the Island Let's look at a 5x5 Grid. Parentheses ( ) denote the cells we are inspecting.

```
        ~ ~ ~ ~ ~   P A C I F I C   O C E A N   ~ ~ ~ ~ ~

    (Top Edge touches Pacific)

    ~   1   2   2   3   5   | (Right Edge touches Atlantic)
    P   3   2   3   4   4   |
    a   2   4   5   3   1   |
    c   6   7   1   4   5   |
    i   5   1   1   2   4   |
    f
    ~
    (Bottom Edge touches Atlantic)

        ~ ~ ~ ~ ~   A T L A N T I C   O C E A N   ~ ~ ~ ~ ~
```

Flow Example: Look at the cell with height `5` at `grid[2][2]` (middle):

- `5` -> `4` (left) -> `2` (up) -> Pacific.
- `5` -> `4` (left) -> `2` (left) -> Pacific.
- `5` -> `3` (right) -> `1` (right) -> Atlantic.
- `5` -> `1` (down) -> Atlantic.
- Since `5` reaches both, it is part of the solution.

---

2. Solution Explanation

The Junior/Brute Force Approach (To be avoided)   A naive approach would be to iterate through every single cell (`r, c`) and start a traversal (DFS/BFS) to see if it reaches the Pacific, then reset and see if it reaches the Atlantic.

- **Why this fails L5 standards:** It repeats work. If cell A flows into cell B, calculating flow for B shouldn't need to be redone when checking A.
- **Complexity:** O((M*N)^2). Too slow for large maps.

The L5/L6 "Inverted Thinking" Approach   Instead of asking "Where can water go from this cell?", we ask **"From which cells can the Ocean reach uphill?"**

We reverse the physics. If water flows High -> Low, then we can trace a path from the Ocean **Low -> High**.

The Strategy:

1. Create a set of cells reachable from the **Pacific** (`pacific_reachable`). Start at the Pacific edges (Left & Top) and move "uphill" (to neighbors current height).
2. Create a set of cells reachable from the **Atlantic** (`atlantic_reachable`). Start at the Atlantic edges (Right & Bottom) and move "uphill".
3. The answer is the **Intersection** of these two sets.

Step-by-Step Visualization   Phase 1: Pacific Flood (Blue Flow) We start BFS/DFS from Top and Left edges. We only move to neighbors that are taller or equal.

```
Input Grid:              Pacific Reachable Set (marked with P):
  1  2  2  3  5            P  P  P  P  P   (Top edge starts as P)
  3  2  3  4  4            P  P  P  P  P   (Flows in from Left & Top)
  2  4  5  3  1            P  P  P  .  .   (5 is >= 4, so P reaches 5)
  6  7  1  4  5            P  P  .  .  .   (6, 7 connected to left edge)
  5  1  1  2  4            P  .  .  .  .   (5 connected to left edge)
```

Notice: The water stops flowing 'uphill' when it hits a neighbor smaller than itself.

Phase 2: Atlantic Flood (Red Flow) We start BFS/DFS from Bottom and Right edges.

```
Input Grid:              Atlantic Reachable Set (marked with A):
  1  2  2  3  5            .  .  .  .  A   (Right edge starts as A)
  3  2  3  4  4            .  .  .  A  A   (Flows in from Right)
  2  4  5  3  1            .  .  A  A  A   (5 is >= 3, so A reaches 5)
  6  7  1  4  5            .  A  .  A  A   (Flows in from Bottom/Right)
  5  1  1  2  4            A  A  A  A  A   (Bottom edge starts as A)
```

Phase 3: The Intersection (The Answer) We find cells that have **both** P and A.

```
Pacific (P):          Atlantic (A):          Intersection (X):
P P P P P             . . . . A              . . . . X
P P P P P      +      . . . A A      =       . . . X X
P P P . .             . . A A A              . . X . .
P P . . .             . A . A A              . X . . .
P . . . .             A A A A A              P . . . . <-- Wait!
```

Correction on the bottom left corner logic for the diagram: The cell 5 at bottom-left (4,0) touches Pacific (Left edge) and Atlantic (Bottom edge). So it is definitely an intersection.

Final Result Map:

```
  .   .   .   .   5
  .   .   .   4   4
  .   .   5   .   .
  .   7   .   .   .
  5   .   .   .   .
```

(These specific cells can flow to both oceans)

---

3. Time and Space Complexity Analysis

Time Complexity: O(M * N)   Where M is the number of rows and N is the number of columns.

Visual Derivation:

```
    [ Operation 1: Pacific BFS/DFS ]
    +-------------------------+
    | Visits each cell at most |
    | ONCE. We use a 'visited'  |   ----> Work = k * (M * N)
    | set to prevent cycles.    |
    +-------------------------+


            +


    [ Operation 2: Atlantic BFS/DFS ]
    +-------------------------+
    | Visits each cell at most |
    | ONCE.                     |   ----> Work = k * (M * N)
    |                           |
    +-------------------------+


            =
```

```
        Total Work = 2 * (M * N)
        Drop constants -> O(M * N)
```

Space Complexity: O(M * N)   Visual Derivation:

```
    [ Memory Usage ]

    1. Pacific Visited Set (Matrix or HashSet)
       Stores up to M*N cells
       [ P | P | . | . ]
       [ P | P | P | . ]

    2. Atlantic Visited Set (Matrix or HashSet)
       Stores up to M*N cells

    3. Recursion Stack (DFS) or Queue (BFS)
       In worst case (snake shaped path),
       recursion depth is O(M*N).

    Total Memory = O(M*N)
```

---

4. Solution Code

We will use **DFS (Depth First Search)** because it is often cleaner to implement for grid traversal problems in interviews, though BFS is equally valid.

Python Solution

```python
from typing import List

class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        # Edge case: Empty board
        if not heights or not heights[0]:
            return []

        ROWS, COLS = len(heights), len(heights[0])

        # Sets to keep track of reachable cells (avoid duplicates/cycles)
        # These act as our 'visited' arrays.
        pacific_reachable = set()
        atlantic_reachable = set()

        # Helper Function: DFS
```

```python
# We pass 'prev_height' to enforce the "Uphill" rule.
# Current cell must be >= prev_height to allow flow from ocean upwards.
def dfs(r, c, visit_set, prev_height):
    # Check boundaries
    if (r < 0 or c < 0 or r == ROWS or c == COLS):
        return

    # Check if already visited
    if (r, c) in visit_set:
        return

    # THE CORE LOGIC:
    # If current cell is SHORTER than previous, water cannot flow
    # from current to previous. Since we are moving from Ocean (low)
    # to Island (high), we stop if height decreases.
    if heights[r][c] < prev_height:
        return

    # Mark as visited
    visit_set.add((r, c))

    # Explore all 4 directions
    dfs(r + 1, c, visit_set, heights[r][c])
    dfs(r - 1, c, visit_set, heights[r][c])
    dfs(r, c + 1, visit_set, heights[r][c])
    dfs(r, c - 1, visit_set, heights[r][c])

# 1. Run DFS for Pacific (Top and Left edges)
for c in range(COLS):
    # Top row (Pacific) and Bottom row (Atlantic)
    dfs(0, c, pacific_reachable, heights[0][c])
    dfs(ROWS - 1, c, atlantic_reachable, heights[ROWS - 1][c])

for r in range(ROWS):
    # Left col (Pacific) and Right col (Atlantic)
    dfs(r, 0, pacific_reachable, heights[r][0])
    dfs(r, COLS - 1, atlantic_reachable, heights[r][COLS - 1])

# 2. Find intersection
# The result is the list of cells present in BOTH sets
res = []
for r in range(ROWS):
    for c in range(COLS):
        if (r, c) in pacific_reachable and (r, c) in atlantic_reachable:
            res.append([r, c])
```

```
        return res
```

JavaScript Solution

```javascript
/**
 * @param {number[][]} heights
 * @return {number[][]}
 */
var pacificAtlantic = function(heights) {
    if (heights.length === 0) return [];

    const ROWS = heights.length;
    const COLS = heights[0].length;

    // We use a 2D boolean array or a Set for visited.
    // Sets in JS compare references for arrays, so encoding coordinates
    // as strings "r,c" is easier for uniqueness.
    const pacificReachable = new Set();
    const atlanticReachable = new Set();

    /* * DFS Function
     * r, c: current coordinates
     * visited: the Set we are updating
     * prevHeight: the height of the previous cell we came from
     */
    const dfs = (r, c, visited, prevHeight) => {
        const key = `${r},${c}`;

        // 1. Bounds check
        if (r < 0 || r >= ROWS || c < 0 || c >= COLS) return;

        // 2. Visited check
        if (visited.has(key)) return;

        // 3. Height check (Inverse Flow: Must be >= previous)
        if (heights[r][c] < prevHeight) return;

        // Add to set
        visited.add(key);

        // Visit neighbors
        dfs(r + 1, c, visited, heights[r][c]);
        dfs(r - 1, c, visited, heights[r][c]);
        dfs(r, c + 1, visited, heights[r][c]);
        dfs(r, c - 1, visited, heights[r][c]);
    };
```

```javascript
    // Iterate over rows and columns to start DFS from edges
    for (let r = 0; r < ROWS; r++) {
        // Left Edge (Pacific)
        dfs(r, 0, pacificReachable, heights[r][0]);
        // Right Edge (Atlantic)
        dfs(r, COLS - 1, atlanticReachable, heights[r][COLS - 1]);
    }

    for (let c = 0; c < COLS; c++) {
        // Top Edge (Pacific)
        dfs(0, c, pacificReachable, heights[0][c]);
        // Bottom Edge (Atlantic)
        dfs(ROWS - 1, c, atlanticReachable, heights[ROWS - 1][c]);
    }

    // Calculate Intersection
    const result = [];
    for (let r = 0; r < ROWS; r++) {
        for (let c = 0; c < COLS; c++) {
            const key = `${r},${c}`;
            if (pacificReachable.has(key) && atlanticReachable.has(key)) {
                result.push([r, c]);
            }
        }
    }

    return result;
};
```

Note 1: Terms and Techniques

"Inverse Flow" / "Reverse Graph Traversal" This is the technique of starting from the *destination* and working backward to the *source* to avoid redundant computations.

- **Why it helps here:** In the naive approach, if 50 cells all flow into one specific path to the ocean, you recalculate that path 50 times. By starting at the ocean and moving backward, you calculate that path exactly once.
- **System Design Parallel:** This is similar to **Dependency Analysis**. If you want to know "Which services depend on Database X?", you don't query every service to ask what they use; you look at Database X's access logs (or a dependency graph) and trace backward.

Note 2: Real World & Interview Variations

Senior interviews (Google/Meta/Bloomberg) rarely ask the "vanilla" version. They wrap it in a real-world scenario to test if you can model the problem.

1. The "Continental Divide" (Google / Maps Team) Prompt: "Given a terrain map of the USA represented as a grid of elevations, identify the 'Continental Divide'—the line of high points where water falling on one side flows to the Pacific and the other side flows to the Atlantic."

- **How to solve:** This IS the Pacific Atlantic Water Flow problem, but the output requirement might be slightly different. Instead of returning *all* cells, you might need to return only the cells that form the boundary (the "ridge") between the two reachable sets.
- **Modification:** Perform the exact solution above. The "Divide" is effectively the intersection set.

2. The "Market Highs" (Bloomberg / Finance) Prompt: "You have a matrix representing stock prices of different sectors over time. We want to find 'Stable Sectors' which have outperformed their neighbors consistently leading up to a market crash (edge of the matrix)."

- **How to solve:**
- The "matrix" is the stock grid.
- "Market Crash" represents the edges (the bounds).
- "Outperformed" implies the value is higher than neighbors.
- This maps to the same algorithm: Start from the crash (edges) and find which stocks "feed" into the crash (are higher than the edge).

3. Information Propagation (Meta / Social Graph) Prompt: "We have two distinct user groups, Group A and Group B, located at opposite ends of a social graph grid. Users only share posts with friends who have a 'influence score' lower than or equal to them. Find the 'Key Influencers' who can get a message out to BOTH Group A and Group B."

- **How to solve:**
- **Height** = Influence Score.
- **Water** = The Message.
- **Oceans** = Group A and Group B.
- **Solution:** Trace from Group A (uphill/higher influence) and Group B (uphill). The intersection are the users who have enough influence to cascade messages down to both groups.

200. Number of Islands

A Google L5/L6 engineer doesn't just solve this problem; they treat it as a **graph traversal exercise** where the goal is to optimize for readability, mod-

ularity, and handling edge cases (like a grid with millions of cells).

---

1. Problem Explanation

Imagine a map represented as a 2D grid. The grid contains two types of values:

- `'1'`: Land
- `'0'`: Water

An **island** is a group of connected `'1's` (land). Connections only happen **horizontally** or **vertically** (up, down, left, right). Diagonal connections don't count.

Goal: Count the total number of distinct islands.

Visual Example

```
Grid Representation:
[
  ["1", "1", "0", "0", "0"],
  ["1", "1", "0", "0", "0"],
  ["0", "0", "1", "0", "0"],
  ["0", "0", "0", "1", "1"]
]

Conceptual Map:
(L) (L)  ~   ~   ~        <-- Island #1 (Connected 1s)
(L) (L)  ~   ~   ~
 ~   ~  (L)  ~   ~        <-- Island #2 (Isolated 1)
 ~   ~   ~  (L) (L)       <-- Island #3 (Connected 1s)

Total Islands: 3
```

---

2. Solution Explanation: The "Sinking" Strategy

An L5+ engineer would likely use **Depth First Search (DFS)** or **Breadth First Search (BFS)**. The most elegant "in-place" approach is the **Sinking Method**.

The Logic

1. **Iterate** through every cell in the grid using a nested loop.
2. If you find a `'1'`:

- Increment your **Island Count**.

- Trigger a **DFS/BFS** to "sink" that island. This means changing all connected `'1's` to `'0's`.

3. By sinking the island, you ensure that you don't count the same island twice as you continue your loop.

Non-Trivial Detail: The DFS Recursion

When you hit a `'1'`, the recursion acts like a "flood fill." It spreads in four directions. If it hits a boundary or water, it stops.

```
STEP-BY-STEP VISUALIZATION (Sinking Island #1)


Initial Grid:        Find first '1' at (0,0):   DFS Sinks Neighbors:
[1, 1, 0]            [X, 1, 0] (Count=1)        [0, 0, 0]
[1, 0, 0]     -->    [1, 0, 0]           -->    [0, 0, 0]
[0, 0, 1]            [0, 0, 1]                  [0, 0, 1]


The algorithm then continues until it finds the next '1' at (2,2).
```

---

3. Time and Space Complexity Analysis

Time Complexity (TC)

We visit every cell in the grid at most twice (once in the main loop and once if it's part of a DFS "sink"). If `M` is the number of rows and `N` is the number of columns:

```
TC DERIVATION (ASCII)
+-------------------------------------------------------+
| Total Cells = Rows (M) * Cols (N)                     |
|                                                       |
| Each cell is visited:                                 |
| 1. By the nested loops (Outer scan)                   |
| 2. Potentially once by DFS (to turn '1' into '0')     |
|                                                       |
| Operations per cell: Constant O(1)                    |
|                                                       |
| Total Complexity: O(M * N)                            |
+-------------------------------------------------------+
```

Space Complexity (SC)

The space depends on the recursion stack. In the worst-case scenario (the entire grid is land), the recursion goes `M * N` deep.

26

```
SC DERIVATION (ASCII)
+-------------------------------------------------------+
| Worst Case: All land.                                 |
| DFS will stack up like a long snake:                  |
| [1][1][1]                                             |
| [1][1][1] -> Stack grows to size of the whole grid    |
|                                                       |
| Total Space: O(M * N)                                 |
+-------------------------------------------------------+
```

---

4. Solution Code

Python Snippet

```python
class Solution:
    def numIslands(self, grid: list[list[str]]) -> int:
        if not grid:
            return 0

        rows, cols = len(grid), len(grid[0])
        island_count = 0

        # This helper function performs the "Sinking"
        # It explores all 4 directions and converts Land to Water
        def sink_island(r, c):
            # Boundary check and "Is it land?" check
            if r < 0 or c < 0 or r >= rows or c >= cols or grid[r][c] == '0':
                return

            # Sink the current land piece
            grid[r][c] = '0'

            # Recursive calls for Up, Down, Left, Right
            sink_island(r + 1, c)
            sink_island(r - 1, c)
            sink_island(r, c + 1)
            sink_island(r, c - 1)

        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == '1':
                    # We found a new island!
                    island_count += 1
                    sink_island(r, c)
```

```
        return island_count
```

Javascript Snippet

```javascript
/**
 * @param {character[][]} grid
 * @return {number}
 */
var numIslands = function(grid) {
    if (!grid || grid.length === 0) return 0;

    let islandCount = 0;
    const rows = grid.length;
    const cols = grid[0].length;

    /**
     * sinkIsland uses DFS to mark all connected land as visited.
     * We modify the grid in-place to save space.
     */
    const sinkIsland = (r, c) => {
        // Stop if out of bounds or if we hit water/already visited
        if (r < 0 || c < 0 || r >= rows || c >= cols || grid[r][c] === '0') {
            return;
        }

        grid[r][c] = '0'; // Sink it

        sinkIsland(r + 1, c);
        sinkIsland(r - 1, c);
        sinkIsland(r, c + 1);
        sinkIsland(r, c - 1);
    };

    for (let r = 0; r < rows; r++) {
        for (let c = 0; c < cols; c++) {
            if (grid[r][c] === '1') {
                islandCount++;
                sinkIsland(r, c);
            }
        }
    }

    return islandCount;
};
```

Note 1: Terms and Techniques

- **Depth First Search (DFS):** An algorithm for traversing a tree or graph where you go as deep as possible along each branch before backtracking. In this problem, it helps by "cleaning up" an entire island as soon as you touch one piece of it.
- **Flood Fill:** A technique used in paint programs to fill an area with color. It's exactly what we do here to "fill" an island with water so we don't count it again.

Note 2: Real World / Interview Variations

1. The "Large Map" Problem (Google)

Variation: What if the grid is too large to fit in memory (e.g., a map of the Earth)?

- **Solution:** Use **Union-Find** with a "sliding window" or "chunking" approach. You process two rows at a time, keeping track of connections between chunks. This avoids loading the whole grid into RAM.

2. The "Max Area" Problem (Meta)

Variation: Instead of the number of islands, find the area of the largest island.

- **Solution:** Modify the `sink_island` function to return an integer (`1 + area_of_neighbors`). Keep a global `max_area` variable and update it after each DFS call.

3. The "Distributed Server Clusters" (Bloomberg)

Variation: Servers are nodes in a grid. If two servers are adjacent, they belong to the same cluster. If one server fails, how many clusters are left?

- **Solution:** This is exactly the same as Number of Islands, but often framed as a "Dynamic Connectivity" problem. You would use **Union-Find** because it allows you to efficiently add and remove connections (nodes) and check the "number of components" in real-time.

128. Longest Consecutive Sequence

Here is how a Senior Staff Engineer (L6) at Google would approach, explain, and solve this problem. We prioritize clarity, scalability of thought, and handling edge cases over "clever" one-liners.

1. Problem Explanation

The Core Task: We are given a messy, unsorted pile of numbers. Our job is to find the length of the longest "streak" of consecutive integers. The order in the original array doesn't matter; we just need to know which numbers exist.

Constraint: The solution must be fast—specifically **O(n)** time complexity. This means we cannot sort the array first (because sorting takes O(n log n), which is slower than O(n)).

Visualizing the Input:

Imagine the input array `nums = [100, 4, 200, 1, 3, 2]`. Visually, the numbers are scattered all over the place.

```
Input Array (Scattered):
+-----+-----+-----+-----+-----+-----+
| 100 |  4  | 200 |  1  |  3  |  2  |
+-----+-----+-----+-----+-----+-----+
```

Visualizing the Goal: Mentally, we want to rearrange them on a number line to see the connections.

```
Number Line visualization:

...   1 -- 2 -- 3 -- 4   ... ...   100  ... ...   200  ...
      ^              ^              ^              ^
      -------------                Seq B          Seq C
      Sequence A                   Length: 1      Length: 1
      Length: 4
```

The Output: The longest sequence is `[1, 2, 3, 4]`. **Result: 4**.

---

2. Solution Explanation

The "Naive" Approach (Mental Check): A junior engineer might think: "Let's pick a number, say `3`, and look through the entire array to see if `4` exists. Then look again for `5`…" This is slow () because for every number, you scan the whole array. We need something faster.

The L5/L6 "Senior" Approach: The Intelligent Hash Set

To solve this in O(n), we need two things:

1. **Instant Lookups:** We need to know if a number exists instantly. We use a **HashSet** for this.
2. **Smart Iteration (The "Start of Sequence" Check):** We only want to count a sequence starting from its *smallest* number. If we encounter a number that is in the *middle* of a sequence, we ignore it.

The Algorithm Visualized:

Step 1: Dump all numbers into a HashSet for O(1) access.

`HashSet: {100, 4, 200, 1, 3, 2}`

Step 2: Iterate through the numbers and apply the "Start of Sequence" Logic.

Logic: For any number `X`, look for `X - 1`.

- **Case A:** If `X - 1` exists in the set, then `X` is **NOT** the start of a sequence. It's somewhere in the middle. **SKIP IT.**
- **Case B:** If `X - 1` does **NOT** exist, then `X` **IS** the start of a new sequence. **COUNT FROM HERE.**

Let's Trace it with `[100, 4, 200, 1, 3, 2]`:

Iteration 1: Current Number = 100

- Check: Does `99` (100 - 1) exist in Set? -> NO.
- Conclusion: `100` is the start of a sequence.
- Action: Count upwards. `101`? No.
- Streak Length: 1.

```
 (Start)
  100
   |
[STOP] -> Max Length is 1
```

Iteration 2: Current Number = 4

- Check: Does `3` (4 - 1) exist in Set? -> **YES**.
- Conclusion: `4` is NOT the start. It is part of a sequence started by someone else (1, 2, or 3).
- Action: **SKIP**. (This is crucial for O(n) speed).

```
... 3 -> [4] -> ...
          ^

4 is not the start. Ignore.
```
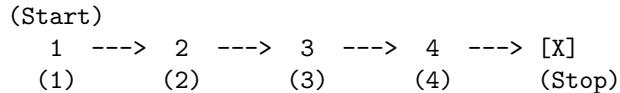
Iteration 3: Current Number = 200

- Check: Does `199` exist? -> NO.
- Conclusion: Start of sequence.
- Action: Count upwards. `201`? No.
- Streak Length: 1.

Iteration 4: Current Number = 1

- Check: Does `0` exist? -> NO.

- Conclusion: **Start of sequence!**

- Action: Start counting loop:

- Have `1`? Yes. (Current streak: 1)

- Have 2? Yes. (Current streak: 2)

- Have 3? Yes. (Current streak: 3)

- Have 4? Yes. (Current streak: 4)

- Have 5? No.

- Update Max Length: 4.

```
(Start)
   1  --->  2  --->  3  --->  4  --->  [X]
  (1)       (2)      (3)      (4)      (Stop)
```

Iteration 5: Current Number = 3

- Check: Does 2 exist? -> YES.
- Action: SKIP.

Iteration 6: Current Number = 2

- Check: Does 1 exist? -> YES.
- Action: SKIP.

Final Result: Max Length = 4.

---

3. Time and Space Complexity Analysis

This is the part where candidates often stumble. They see a `while` loop inside a `for` loop and assume it is O(N^2). An L6 engineer proves why it isn't.

Time Complexity: O(N)

Think about how many times we visit each number.

```
VISITATION VISUALIZATION
Total Elements: N

1. Set Creation:  [x] [x] [x] ... (N operations)

2. The Loop Logic:
   For every number 'num':

   Is 'num' the start?
     /        \
   NO          YES
 (Skip)    (Enter While Loop)
   |             |
   |             V
   |         We walk the sequence.
   |         Crucially: We only walk each number in a sequence ONCE.
```

```
    |
    V
  Once we process the sequence 1->2->3->4 starting at 1,
  we will NEVER process 2, 3, or 4 inside the while loop again
  because we skip them when the outer loop hits them.

Total Work = (N lookups for "is start") + (N lookups inside "while loop")
           = O(N) + O(N)
           = O(N)
```

Space Complexity: O(N)

```
MEMORY USAGE

Array: [ ... N integers ... ]  <- Input
             |
             V
HashSet: { ... N integers ... } <- O(N) Space

We store every unique number exactly once in the hash set.
```

---

4. Solution Code

We focus on readable variable names and handling empty inputs, which are
common production bugs.

Python Solution

```python
from typing import List


class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        # Edge case: Empty input handles naturally, but good to be aware
        if not nums:
            return 0

        # 1. Create a set for O(1) lookups.
        # This removes duplicates implicitly, which is fine for this problem.
        num_set = set(nums)
        longest_streak = 0

        # 2. Iterate through the set (iterating set is slightly faster than list usually)
        for num in num_set:

            # 3. Check if 'num' is the start of a sequence.
            # If (num - 1) is in the set, then 'num' is not the start.
```

```python
            if (num - 1) not in num_set:
                current_num = num
                current_streak = 1

                # 4. If it is the start, count how long the sequence goes.
                while (current_num + 1) in num_set:
                    current_num += 1
                    current_streak += 1

                # 5. Update our global max
                longest_streak = max(longest_streak, current_streak)

        return longest_streak
```

Javascript Solution

```javascript
/**
 * @param {number[]} nums
 * @return {number}
 */
var longestConsecutive = function(nums) {
    // Standard Google-style guard clause for empty input
    if (!nums || nums.length === 0) {
        return 0;
    }

    // 1. Use a Set for O(1) complexity access
    const numSet = new Set(nums);
    let maxStreak = 0;

    // 2. Iterate over the Set (unique values only)
    for (const num of numSet) {

        // 3. Intelligent Check: Is this the beginning of a sequence?
        // If num-1 exists, this number is part of a sequence but not the start.
        if (!numSet.has(num - 1)) {
            let currentNum = num;
            let currentStreak = 1;

            // 4. Expand the sequence as far as possible
            while (numSet.has(currentNum + 1)) {
                currentNum += 1;
                currentStreak += 1;
            }

            // 5. Update maximum found so far
```

```
            maxStreak = Math.max(maxStreak, currentStreak);
        }
    }

    return maxStreak;
};
```

Note 1: Terminology Breakdown

HashSet (Hashing): This is the data structure that makes the solution possible. It maps data of arbitrary size (the number) to fixed-size values (hash codes) to allow for data retrieval in constant time, or O(1).

- *Why it helps here:* Without it, checking if `num + 1` exists would require scanning the array, making the complexity O(N).

Greedy Approach: This solution is a form of a greedy algorithm. We "greedily" try to build the longest chain only when we find a valid starting point, ignoring all sub-optimal starting points.

Note 2: Real World / Interview Variations

At companies like Google, Meta (Facebook), and Bloomberg, the "vanilla" Leetcode question is just the warmup. They will twist the requirements to see if you can adapt your data structure knowledge.

Variation A: The "Data Stream" (Meta / Google) Question: "What if the numbers come in one by one as a stream? You need to return the longest consecutive sequence length seen *so far* after every new number."

- **Why it's hard:** You can't rebuild the HashSet every time. O(N) per insertion is too slow.
- **Solution:** Use a **HashMap (Dictionary)**.
- Map `key` = number, `value` = length of the sequence that number is part of.
- When `X` arrives, check if `X-1` and `X+1` exist in the map.
- If they do, **bridge** the two sequences together. Update the "head" and "tail" of the new combined sequence with the new total length.
- **Complexity:** O(1) per number added.

Variation B: "Range Summary" (Bloomberg) Question: "Instead of just the length, return the actual ranges. E.g., for `[1, 2, 3, 5, 7]`, return `['1->3', '5', '7']`."

- **Why it's hard:** Requires formatting and careful tracking of start/end points.
- **Solution:** Sort the array first (O(N log N)). Since we need to output the ranges in order usually, sorting is acceptable and makes printing trivial. If sorting is banned, use the HashSet approach but store the `(start, end)` pairs.

Variation C: "2D Grid Longest Path" (Google)   Question: "Given a 2D matrix of integers, find the longest consecutive path. You can move up, down, left, right."

- **Why it's hard:** It introduces graph traversal.
- **Solution:** This becomes a **DFS (Depth First Search) + Memoization** problem.
- For every cell, perform DFS to see how far you can go (e.g., if cell is `4`, look for neighbor `5`).
- Store results in a `memo` table so you don't re-calculate the path for the same cell twice.
- **Complexity:** O(Rows * Cols).

Alien Dictionary

Here is how a Senior Staff Engineer (L6) at Google would break down the "Alien Dictionary" problem. We focus on clarity, edge cases, and maintainable patterns.

1. Problem Explanation

The Scenario: Imagine a simplified alien language that uses standard English lowercase letters, but the *order* of these letters is completely different from our alphabet. You don't know this order.

You are given a list of words written in this alien language. The crucial piece of information is that this list is **already sorted lexicographically** (like a dictionary) according to the alien rules.

The Goal: Deduce the unique order of the letters in this alien language.

Example 1: Input: `["wrt", "wrf", "er", "ett", "rftt"]` Output: `"wertf"`

Why?

- Look at `"wrt"` and `"wrf"`. The first two letters `w` and `r` are the same. The difference is `t` vs `f`. Since `"wrt"` comes before `"wrf"`, we know **`t` comes before `f`**.
- Look at `"wrf"` and `"er"`. The first letter is different (`w` vs `e`). Since `"wrf"` is first, **`w` comes before `e`**.

Visualizing the Relationships:

```
Input List:
1. wrt
2. wrf  ---> comparing 1 & 2: 'w','r' match. 't' comes before 'f'. Edge: t -> f
3. er   ---> comparing 2 & 3: 'w' != 'e'. 'w' comes before 'e'.    Edge: w -> e
4. ett  ---> comparing 3 & 4: 'e' match. 'r' comes before 't'.    Edge: r -> t
5. rftt ---> comparing 4 & 5: 'e' != 'r'. 'e' comes before 'r'.   Edge: e -> r
```

We treat these relationships as a **Directed Graph**.

---

2. Solution Explanation

This problem is a classic application of **Topological Sorting** on a Directed Acyclic Graph (DAG).

The Strategy (The "L6" Approach): We don't try to guess the whole alphabet at once. We build a dependency graph where an edge `A -> B` means "Letter A must appear before Letter B".

Phase 1: Graph Construction We iterate through the list of words. We strictly compare **adjacent** words (word `i` and word `i+1`).

- Find the *first* character that differs between them.
- Create a directed edge from the character in word `i` to the character in word `i+1`.
- *Crucial Edge Case:* If word `i+1` is a prefix of word `i` (e.g., `["apple", "app"]`), the input is invalid. In a dictionary, shortest prefixes always come first.

Phase 2: In-Degree Calculation We count how many "prerequisites" each letter has.

- `in_degree[x] = 0`: Letter `x` has no dependencies; it can come first.
- `in_degree[x] = 2`: Letter `x` needs 2 other letters to be placed before it.

Phase 3: Kahn's Algorithm (BFS)

1. Put all letters with `in_degree == 0` into a Queue.
2. Pop a letter, add it to our result string.
3. "Remove" this letter from the graph by decrementing the `in_degree` of its neighbors.
4. If a neighbor's `in_degree` hits 0, add it to the Queue.

Visualization of the Algorithm:

Step A: Building the Graph Input: `["z", "x", "z"]` (Wait! "z" comes before "x", then "x" comes before "z"? That's a cycle!)

Let's use a valid example: `["za", "zb", "ca", "cb"]`

```
1. Compare "za" & "zb":
   'z' matches 'z'.
   'a' != 'b'.
   Relation: a -> b

2. Compare "zb" & "ca":
   'z' != 'c'.
   Relation: z -> c
   (Stop comparing rest of word)

3. Compare "ca" & "cb":
   'c' matches 'c'.
   'a' != 'b'.
   Relation: a -> b (Already exists)

Resulting Graph (Adjacency List):
{
  'a': ['b'],
  'z': ['c'],
  'b': [],
  'c': []
}

In-Degrees (Count of incoming arrows):
a: 0  (No one points to a)
z: 0  (No one points to z)
b: 1  (a points to b)
c: 1  (z points to c)
```

Step B: Processing the Queue (Kahn's Algo)

```
Initial State:
Queue: ['a', 'z']  (Order in queue doesn't strictly matter for validity, just indicates no c
Result: ""

Iteration 1:
Pop 'a'.
Result: "a"
'a' points to 'b'. Decrement 'b' in-degree.
'b' in-degree is now 0. Add 'b' to Queue.
Queue: ['z', 'b']

Iteration 2:
Pop 'z'.
Result: "az"
'z' points to 'c'. Decrement 'c' in-degree.
'c' in-degree is now 0. Add 'c' to Queue.
```

```
Queue: ['b', 'c']

Iteration 3:
Pop 'b'.
Result: "azb"
'b' has no neighbors.
Queue: ['c']

Iteration 4:
Pop 'c'.
Result: "azbc"
'c' has no neighbors.
Queue: []

Final Check:
Did we use all unique letters found in input?
Yes. Return "azbc".
```

Cycle Detection (The Trap): If the Queue becomes empty but the Result string doesn't contain all unique characters, it means there was a **Cycle** (e.g., `a -> b -> a`). In a cycle, the in-degrees never reach 0, so those nodes never enter the queue. We return `""`.

---

3. Time and Space Complexity Analysis

Let `C` be the total length of all words combined (total characters). Let `U` be the number of unique characters (at most 26 in English). Let `N` be the number of words.

Time Complexity Derivation:

```
Operation                       | Cost
--------------------------------|-----------------------
1. Initialize Data Structures   | O(U)
2. Build Graph                  |
   - Iterate N-1 pairs of words |
   - Compare chars until mismatch |
   - Total comparisons limited by |
     total chars in input list  | O(C)
3. Calculate Initial In-Degrees | O(C) (Done during build)
4. Initialize Queue             | O(U)
5. BFS (Process Graph)          |
   - Visit every node (vertex)  | O(U)
   - Visit every edge           | O(min(U^2, N)) *
--------------------------------|-----------------------
TOTAL TIME                      | O(C)
```

**Note on Edges: The number of edges cannot exceed the number of word pairs (N-1) because each pair adds at most one edge. Also, with 26 letters, max edges is 26*26. *Thus, the graph part is negligible compared to reading all characters O(C).*

Space Complexity Derivation:

```
Structure                        | Space
---------------------------------|----------------------
1. Adjacency List (Graph)        | O(V + E) = O(U + min(U^2, N))
2. In-Degree Map                 | O(U)
3. Queue                         | O(U)
4. Output String                 | O(U)
---------------------------------|----------------------
TOTAL SPACE                      | O(1) or O(U)
```

Since U is fixed at maximum 26 (lowercase English letters), the space complexity is technically O(1). If the alphabet size could grow, it would be O(U + min(U^2, N)).

---

4. Solution Code

Python Solution

```python
from collections import deque, defaultdict

def alienOrder(words: list[str]) -> str:
    # 1. Initialize Graph and In-Degree
    # We use a set for graph values initially to avoid duplicate edges easily,
    # though strict adjacency logic usually prevents duplicates anyway.
    adj = {c: set() for w in words for c in w}
    in_degree = {c: 0 for w in words for c in w}

    # 2. Build the Graph
    for i in range(len(words) - 1):
        w1, w2 = words[i], words[i+1]
        min_len = min(len(w1), len(w2))

        # Edge Case: Prefix Logic
        # If w1 is longer than w2 and w1 starts with w2 (e.g., "apple", "app"),
        # then the dictionary is invalid.
        if len(w1) > len(w2) and w1[:min_len] == w2:
            return ""

        for j in range(min_len):
            if w1[j] != w2[j]:
```

```python
                if w2[j] not in adj[w1[j]]:
                    adj[w1[j]].add(w2[j])
                    in_degree[w2[j]] += 1
                # Important: Only the FIRST difference determines order.
                # We must break immediately after finding it.
                break

    # 3. Initialize Queue (Kahn's Algorithm)
    # Add all nodes with 0 in-degree (no dependencies)
    queue = deque([c for c in in_degree if in_degree[c] == 0])
    res = []

    # 4. Process Queue
    while queue:
        char = queue.popleft()
        res.append(char)

        for neighbor in adj[char]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # 5. Cycle Detection check
    # If result length < unique chars, there was a cycle.
    if len(res) < len(in_degree):
        return ""

    return "".join(res)
```

JavaScript Solution

```javascript
/**
 * @param {string[]} words
 * @return {string}
 */
var alienOrder = function(words) {
    // 1. Initialize Data Structures
    const adj = new Map();
    const inDegree = new Map();

    // Initialize every unique character with in-degree 0 and empty neighbors
    for (const word of words) {
        for (const char of word) {
            inDegree.set(char, 0);
            adj.set(char, new Set());
        }
    }
```

```javascript
}

// 2. Build Graph
for (let i = 0; i < words.length - 1; i++) {
    const w1 = words[i];
    const w2 = words[i+1];
    const minLen = Math.min(w1.length, w2.length);

    // Edge Case: Check for invalid prefix ordering
    if (w1.length > w2.length && w1.startsWith(w2)) {
        return "";
    }

    for (let j = 0; j < minLen; j++) {
        if (w1[j] !== w2[j]) {
            const source = w1[j];
            const dest = w2[j];

            // If this relationship is new, record it
            if (!adj.get(source).has(dest)) {
                adj.get(source).add(dest);
                inDegree.set(dest, inDegree.get(dest) + 1);
            }
            // Stop after the first difference
            break;
        }
    }
}

// 3. Initialize Queue
const queue = [];
for (const [char, count] of inDegree) {
    if (count === 0) queue.push(char);
}

let res = "";

// 4. BFS (Kahn's Algorithm)
while (queue.length > 0) {
    const char = queue.shift();
    res += char;

    for (const neighbor of adj.get(char)) {
        inDegree.set(neighbor, inDegree.get(neighbor) - 1);
        if (inDegree.get(neighbor) === 0) {
            queue.push(neighbor);
```

```
            }
        }
    }

    // 5. Cycle Check
    if (res.length < inDegree.size) {
        return "";
    }

    return res;
};
```

Note 1: Terminology Explained

Topological Sort: Imagine you are dressing yourself. You must put on socks before shoes, and underwear before pants. But it doesn't matter if you put on your left sock or your right sock first. Topological Sort is an algorithm that takes a list of tasks with dependencies (A before B) and produces a linear ordering of tasks that respects all those rules. It only works on graphs that have no cycles (you can't have A before B and B before A).

Kahn's Algorithm: This is a specific way to implement Topological Sort. It relies on the concept of "In-Degree" (how many things must happen before *this* thing). It is essentially a Breadth-First Search (BFS) where we only visit nodes once their dependencies are cleared.

Note 2: Real World / Interview Variations

At the L5/L6 level, I'm not just checking if you know the algorithm. I'm checking if you can map a messy real-world problem to this clean abstract graph problem.

1. Google: Package Build System (Dependency Resolution)

- **The Question:** "We have a massive repo with thousands of libraries. Given a list of build files where `Library A` imports `Library B`, determine the order in which we must compile them. If there is a circular dependency, detect it."
- **The Approach:** This is identical to Alien Dictionary.
- **Nodes:** Libraries.
- **Edges:** Imports (A imports B means B -> A, B must be built first).
- **Solution:** Build the graph, run Kahn's algorithm. If you detect a cycle, throw a "Circular Dependency Error" and output the libraries involved in the cycle to help the developer fix it.

2. Meta (Facebook): Task Scheduling

- **The Question:** "You have a list of tasks to complete a project. Some tasks require others to be finished first (e.g., 'Design DB' before 'Write API'). Can we finish all tasks? If so, give me a valid schedule."
- **The Approach:** This is a "Course Schedule" variant.
- **Focus:** Often they ask for *parallel* execution. "If each task takes 1 hour, what is the minimum time to finish?"
- **Modification:** Instead of a simple list, you track the "depth" or "level" of the topological sort. All nodes popped at the same BFS level can be done in parallel.

3. Bloomberg: Currency Arbitrage / Ranking

- **The Question:** "We have a list of currency exchange pairs where we know one currency is 'stronger' than another based on historical data. Rank the currencies from strongest to weakest."
- **The Approach:**
- **Focus:** Ambiguity. Unlike Alien Dictionary, we might have disjoint graphs (e.g., we know USD > CAD, and EUR > GBP, but we don't know the relation between USD and EUR).
- **Modification:** You must clarify if a partial order is acceptable or if you need to flag that the ranking is incomplete. In Alien Dictionary, any valid topological sort is fine, but in financial ranking, disjoint sets might imply missing data.

Graph Valid Tree

Here is how a Senior Staff (L6) or Senior (L5) Engineer at Google would break down the "Graph Valid Tree" problem.

At this level, the focus isn't just on getting *a* solution, but on verifying the fundamental properties of the data structure (Connectivity + Acyclicity) and choosing the most trade-off-appropriate algorithm (Union-Find vs. BFS/DFS).

1. Problem Explanation

The problem asks: Given `n` nodes labeled from `0` to `n-1` and a list of undirected edges, write a function to check if these edges make up a valid tree.
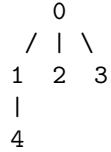
What is a Tree? In the context of Graph Theory, a "Tree" is a special type of graph that must satisfy two strict conditions:

1. **Fully Connected:** You must be able to travel from any node to any other node. No node is left isolated.
2. **No Cycles:** You cannot start at a node, follow a path of edges, and return to the same node without backtracking.

If *either* of these conditions is violated, it is **not** a tree.
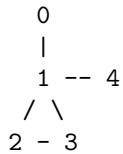
Visual Examples

Scenario A: The Perfect Tree `n = 5`, `edges = [[0,1], [0,2], [0,3], [1,4]]`

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

- **Connected?** Yes. All nodes (0, 1, 2, 3, 4) are reachable.
- **Cycles?** No. There are no loops.
- **Result:** True (It is a valid tree).

Scenario B: The Cycle (Not a Tree) `n = 5`, `edges = [[0,1], [1,2], [2,3], [1,3], [1,4]]`

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

- **Connected?** Yes.
- **Cycles?** Yes! The path `1 -> 2 -> 3 -> 1` forms a triangle loop.
- **Result:** False.

Scenario C: The Disconnected Graph (Not a Tree) `n = 4`, `edges = [[0,1], [2,3]]`

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

- **Connected?** No. You cannot go from node 1 to node 2. These are two separate islands (components).
- **Cycles?** No.
- **Result:** False.

---

2. Solution Explanation

An L5/L6 engineer recognizes that checking for cycles and connectivity can be done via **Graph Traversal (DFS/BFS)** or **Union-Find (Disjoint Set)**.

For this explanation, we will use the **Union-Find** approach because it is elegant, extremely efficient for cycle detection in undirected graphs, and handles the "connectivity" check implicitly by counting edges.

The Logic (Two Checks):

Check 1: Edge Count Validation (The Quick Fail) For a graph of `n` nodes to be a tree, it *must* have exactly `n - 1` edges.

- If `edges.length < n - 1`: It's impossible to connect all nodes. (Disconnected)
- If `edges.length > n - 1`: By Pigeonhole Principle, there *must* be a cycle.
- **L5 Insight:** We return `false` immediately if `edges.length != n - 1`. This saves us from running the main algorithm.

Check 2: Cycle Detection via Union-Find We iterate through every edge. For an edge connecting `A` and `B`:

1. Find the "Parent" (or representative) of `A`.
2. Find the "Parent" of `B`.
3. If `Parent(A) == Parent(B)`: They are already connected! Adding this edge would create a closed loop (Cycle). Return `false`.
4. If `Parent(A) != Parent(B)`: Merge (Union) them into one group.

If we finish checking all edges without finding a cycle, and we passed the "Quick Fail" check (ensuring correct number of edges for connectivity), the graph is a valid tree.

Visual Walkthrough (Union-Find)

Input: `n = 5`, `edges = [[0,1], [1,2], [2,3], [1,3], [1,4]]`

Step 0: Initialization Every node is its own parent initially.

```
Nodes:    [0]  [1]  [2]  [3]  [4]
Parents:   0    1    2    3    4
```

Step 1: Process edge [0, 1]

- Root of 0 is 0. Root of 1 is 1.
- Roots are different. Union them.

```
 0 -- 1    [2]  [3]  [4]
```

```
Parents: [1, 1, 2, 3, 4]  <-- 0 points to 1
```

Step 2: Process edge [1, 2]

- Root of 1 is 1. Root of 2 is 2.
- Roots are different. Union them.

```
 0 -- 1 -- 2    [3]  [4]
```

```
Parents: [1, 1, 1, 3, 4]  <-- 2 points to 1
```

Step 3: Process edge [2, 3]

- Root of 2 is 1 (2->1). Root of 3 is 3.
- Roots are different. Union them.

```
   0 -- 1 -- 2 -- 3    [4]
```

`Parents: [1, 1, 1, 1, 4]  <-- 3 points to 1`

Step 4: Process edge [1, 3]

- **CRITICAL STEP**
- Root of 1 is 1.
- Root of 3 is 1 (3->1).
- **Roots are SAME!** This means 1 and 3 are already connected via 2. Adding edge [1,3] creates a cycle.
- **Return FALSE.**

---

3. Time and Space Complexity Analysis

Time Complexity: O(N * (N)) (Where N is the number of nodes, and   is the Inverse Ackermann function).

Let's visualize the "Path Compression" in Union-Find that makes this nearly O(N).

Without path compression, a tree of parents could look like a straight line (Linked List), making search O(N).
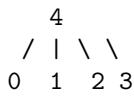
```
Bad Case:
0 -> 1 -> 2 -> 3 -> 4
Find(0) takes 5 steps.
```

With **Path Compression**, after we find the root, we rewire the node to point directly to the root.

```
Good Case (After Compression):
      4
   / | \ \
  0  1  2 3
Find(0) takes 1 step.
```

Because the tree stays very flat, the `Find` operation is nearly constant time, O(1), on average. Since we process `N` edges (at most), the total time is nearly `O(N)`.

Space Complexity: O(N)

We need an array to store the parent of each node.

```
Memory Usage Visualization:

Index:  0  1  2  3  ... (N-1)
       +--+--+--+--+----+
Array: | P| P| P| P| ... |
```

```
    +--+--+--+--+-----+
```

Takes exactly N integers of space.

---

4. Solution Code

Python Solution (Union-Find)

```python
class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        # L5/L6 Check: Graph Theory Property
        # A tree with 'n' nodes MUST have exactly 'n-1' edges.
        # If fewer, it's disconnected. If more, it has a cycle.
        if len(edges) != n - 1:
            return False

        # Initialize the parent list.
        # Initially, each node is its own parent.
        # parent = [0, 1, 2, ... n-1]
        parent = list(range(n))

        # Helper Function: Find
        # Returns the representative (root) of the set containing node 'i'.
        # Implements Path Compression for O(1) average time.
        def find(i):
            if parent[i] == i:
                return i
            parent[i] = find(parent[i]) # Path compression happens here
            return parent[i]

        # Helper Function: Union
        # Connects two nodes. Returns False if they are already connected.
        def union(i, j):
            root_i = find(i)
            root_j = find(j)

            # If roots are the same, a cycle is detected
            if root_i == root_j:
                return False

            # Merge the sets
            # Arbitrarily make one root the parent of the other
            parent[root_i] = root_j
            return True
```

```python
        # Process all edges
        for u, v in edges:
            # If union returns False, we found a cycle
            if not union(u, v):
                return False

        # If we processed all edges without finding a cycle,
        # and we verified the edge count at the start, it is a Valid Tree.
        return True
```

Javascript Solution (Union-Find)

```javascript
/**
 * @param {number} n
 * @param {number[][]} edges
 * @return {boolean}
 */
var validTree = function(n, edges) {
    // Check 1: Edge Count
    // A tree must have exactly n - 1 edges to be fully connected and acyclic.
    if (edges.length !== n - 1) {
        return false;
    }

    // Initialize Union-Find data structure
    // Each index tracks the parent of that node.
    const parent = [];
    for (let i = 0; i < n; i++) {
        parent[i] = i;
    }

    // Find Function with Path Compression
    // Recursively finds the root and points the node directly to it.
    function find(i) {
        if (parent[i] === i) {
            return i;
        }
        parent[i] = find(parent[i]); // Path compression
        return parent[i];
    }

    // Union Function
    // Attempts to connect two nodes. Returns false if cycle detected.
    function union(i, j) {
        const rootI = find(i);
        const rootJ = find(j);
```

```javascript
    // Cycle detected: Both nodes already share the same root
    if (rootI === rootJ) {
        return false;
    }

    // Merge sets: Make rootJ the parent of rootI
    parent[rootI] = rootJ;
    return true;
}

// Iterate through all edges
for (const [u, v] of edges) {
    if (!union(u, v)) {
        return false; // Cycle found
    }
}

    return true;
};
```

New Term: Path Compression & Union-Find

What is it? Union-Find (or Disjoint Set Union - DSU) is a data structure that tracks elements partitioned into a number of disjoint (non-overlapping) sets. It provides near-constant-time operations to:

1. **Find:** Determine which set a particular element is in.
2. **Union:** Join two sets together.

Why it helps here: It is specifically designed to manage dynamic connectivity. It allows us to process edges one by one and instantly know if adding an edge connects two previously separate components (safe) or connects a component to itself (cycle).

Inverse Ackermann Function ( ): When analyzing Union-Find complexity, we use (N). This function grows extremely slowly. For all practical values of N (even up to the number of atoms in the universe), (N) is less than 5. Thus, we treat it as effectively O(1) in interviews.

Real World / Interview Variations

Senior roles at Google, Meta, and Bloomberg rarely ask the "raw" Leetcode question. They wrap it in a practical scenario.

1. **Google: "The Redundant Connection" (Network Reliability)**

   - **The Prompt:** "We are building a massive server network. We have a list of cables connecting servers. We need to identify a cable that, if removed, would not disconnect the network (i.e., it's a redundant backup cable creating a loop)."
   - **The L5 Approach:** This is exactly the "Cycle Detection" part of Graph Valid Tree.
   - Iterate through cables using **Union-Find**.
   - The moment `find(serverA) == find(serverB)`, you have found the redundant cable. Return it immediately.

2. **Meta: "Friend Circles" / "Connected Components"**

   - **The Prompt:** "Given a list of friendships, calculate how many distinct groups of friends exist. (A is friends with B, B with C -> A, B, C are one group)."
   - **The L5 Approach:**
   - Initialize `count = n` (assuming everyone is alone initially).
   - Iterate through friendships.
   - Every time you successfully `Union` two people who were in different sets, decrement `count`.
   - Final `count` is the number of distinct friend circles.

3. **Bloomberg: "The Minimal Cost Connectivity" (Kruskal's Algorithm)**

   - **The Prompt:** "We have several financial data centers. Connecting them costs money based on distance. What is the cheapest way to connect all centers so they can all share data?"
   - **The L5 Approach:** This is the **Minimum Spanning Tree (MST)** problem.
   - Sort all potential connections by cost (cheapest first).
   - Iterate through sorted connections.
   - Use **Union-Find**. If `find(centerA) != find(centerB)`, build that connection (Union them). If they are already connected, skip it (it's redundant and costs money).
   - This builds the cheapest Valid Tree.

Number of Connected Components in an Undirected Graph

This response breaks down the "Number of Connected Components" problem as requested, simulating the approach of a high-level engineer.

1. Problem Explanation

Imagine you are analyzing a social network.

- **Nodes** represent people.
- **Edges** represent friendships.
- If Person A is friends with Person B, and Person B is friends with Person C, they all belong to the same **"Friend Circle"**.
- Isolated people are their own circle.

The Goal: Count the total number of distinct Friend Circles (Connected Components).

We are given:

1. `n`: The number of nodes (labeled `0` to `n-1`).
2. `edges`: A list of pairs `[a, b]` indicating a connection.

Visual Examples Example 1: `n = 5`, `edges = [[0, 1], [1, 2], [3, 4]]`

- Nodes `0`, `1`, `2` are linked.
- Nodes `3`, `4` are linked.

```
Cluster A:      Cluster B:

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

Result: 2 Components.

Example 2: `n = 5`, `edges = [[0, 1], [2, 3]]` (Node 4 is missing from edges)

- `0-1` is one group.
- `2-3` is a second group.
- `4` is completely alone.

```
 Group 1         Group 2         Group 3

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

Result: 3 Components.

---

2. Solution Explanation

While DFS or BFS (Graph Traversal) are valid solutions, a Senior Engineer (L5/L6) would likely opt for the **Union-Find (Disjoint Set Union - DSU)** data structure.

Why Union-Find? It is designed specifically for dynamic connectivity. It processes connections elegantly and is often easier to implement for "grouping" problems than recursive DFS.

The Algorithm: Union-Find

1. **Start:** Treat every node as its own independent "Component". Total components = `n`.
2. **Iterate:** Go through every edge `[a, b]`.
3. **Find:** Identify the "Leader" (or Root) of group `a` and group `b`.
4. **Union:**

- If `Leader(a)` and `Leader(b)` are different, merge the groups!
- Make one leader report to the other.
- **Decrement** the total component count by 1 (since two groups just became one).
- If they already share the same leader, do nothing (they are already connected).

Visualizing the Algorithm Input: `n = 5`, `edges = [[0, 1], [1, 2], [3, 4]]`

Step 0: Initialization

- Everyone is their own parent.
- `Count = 5`

```
Index:  [0]   [1]   [2]   [3]   [4]
Parent: [0]   [1]   [2]   [3]   [4]


Visual: (0)   (1)   (2)   (3)   (4)
```

Step 1: Process edge [0, 1]

- Leader of 0 is 0. Leader of 1 is 1.
- **Merge:** Make 0 the parent of 1.
- `Count = 4`

```
Index:  [0]   [1]   [2]   [3]   [4]
Parent: [0]   [0]   [2]   [3]   [4]
              ^
              |__ 1 now points to 0


Visual:  (0)     (2)   (3)   (4)
```

```
            |
          (1)
```

Step 2: Process edge [1, 2]

- Leader of 1? Follow the arrow: 1 -> 0. Leader is 0.
- Leader of 2? Is 2.
- **Merge:** Make 0 the parent of 2.
- Count = 3

```
Index:  [0]  [1]  [2]  [3]  [4]
Parent: [0]  [0]  [0]  [3]  [4]
                   ^
                   |__ 2 now points to 0


Visual:    (0)        (3)   (4)
          /   \
        (1)   (2)
```

Step 3: Process edge [3, 4]

- Leader of 3 is 3. Leader of 4 is 4.
- **Merge:** Make 3 the parent of 4.
- Count = 2

```
Index:  [0]  [1]  [2]  [3]  [4]
Parent: [0]  [0]  [0]  [3]  [4]
                        ^
                        |__ 4 now points to 3


Visual:    (0)        (3)
          /   \        |
        (1)   (2)     (4)
```

Final Answer: 2.

---

3. Time and Space Complexity Analysis

Time Complexity We use **Path Compression**, which flattens the tree structure every time we search. This makes the `find` operation nearly constant time on average.

The complexity is defined by the **Inverse Ackermann function**, denoted as (n). For all practical purposes in the universe, (n) < 5.

```
TC Derivation:

1. Initialization Loop:
   Iterate 0 to n-1 to set initial parents.
```

```
    Time: O(n)

    [0][1][2]...[n]

2. Edge Processing Loop:
   Iterate through E edges.
   Perform 2 Find operations + 1 Union.
   Time: E * (n)      E * O(1)

   Edge 1: [Find][Find][Union]
   Edge 2: [Find][Find][Union]
   ...

Total Time: O(n + E * (n))
Simplified: O(n + E)
```

Space Complexity   We store arrays to track parent pointers and rank (size of trees).

```
SC Derivation:

1. Parent Array:
   Stores 'n' integers.
   Space: O(n)
   [ 0, 0, 0, 3, 3 ]

2. Rank/Size Array:
   Stores 'n' integers.
   Space: O(n)
   [ 1, 1, 1, 1, 1 ]

Total Space: O(n)
```

---

4. Solution Code

Python Solution

```python
class Solution:
    def countComponents(self, n: int, edges: list[list[int]]) -> int:
        # Parent array: parent[i] stores the parent of node i
        # Initially, every node is its own parent
        parent = [i for i in range(n)]

        # Rank array: Used to keep the tree flat.
        # We attach the shorter tree to the taller tree.
```

```python
        rank = [1] * n

        # Start with n distinct components
        self.count = n

        # Find Function (with Path Compression)
        # Finds the representative (root) of the set containing 'node'
        def find(node):
            if node == parent[node]:
                return node
            # Recursively find the root and update parent[node] to point
            # directly to the root. This is "Path Compression".
            parent[node] = find(parent[node])
            return parent[node]

        # Union Function
        # Merges two sets together
        def union(n1, n2):
            root1 = find(n1)
            root2 = find(n2)

            # If they are already in the same group, do nothing
            if root1 == root2:
                return 0 # 0 merges performed

            # Union by Rank: Attach smaller rank tree to higher rank tree
            if rank[root1] > rank[root2]:
                parent[root2] = root1
            elif rank[root2] > rank[root1]:
                parent[root1] = root2
            else:
                parent[root2] = root1
                rank[root1] += 1

            return 1 # 1 merge performed

        # Process all edges
        for n1, n2 in edges:
            # If a merge happened, decrease the global component count
            self.count -= union(n1, n2)

        return self.count
```

JavaScript Solution

```javascript
/**
 * @param {number} n
 * @param {number[][]} edges
 * @return {number}
 */
var countComponents = function(n, edges) {
    // Parent array initialization
    const parent = Array.from({ length: n }, (_, i) => i);
    // Rank array for Union by Rank optimization
    const rank = new Array(n).fill(1);

    // Initial count is n (everyone is isolated)
    let count = n;

    // Find function with Path Compression
    // Collapses the path so future lookups are O(1)
    const find = (node) => {
        if (node === parent[node]) {
            return node;
        }
        // Path compression: point node directly to root
        parent[node] = find(parent[node]);
        return parent[node];
    };

    // Union function
    const union = (n1, n2) => {
        const root1 = find(n1);
        const root2 = find(n2);

        // If roots are same, they are already connected
        if (root1 === root2) return;

        // Optimization: Attach smaller tree to larger tree
        if (rank[root1] > rank[root2]) {
            parent[root2] = root1;
        } else if (rank[root2] > rank[root1]) {
            parent[root1] = root2;
        } else {
            parent[root2] = root1;
            rank[root1]++;
        }

        // Successful merge reduces component count
        count--;
    };
```

```
    // Iterate over edges
    for (const [n1, n2] of edges) {
        union(n1, n2);
    }

    return count;
};
```

Note 1: New Terms

Union-Find (DSU): A data structure that tracks elements partitioned into disjoint subsets. It provides near-constant time operations to add new sets and merge existing sets. It is the "gold standard" for connectivity problems.

Path Compression: An optimization in the `find` function. Instead of traversing up the hierarchy `A->B->C->Root` every time, after the first traversal, we update `A` to point directly to `Root`. This flattens the structure.

Union by Rank: An optimization in the `union` function. We always attach the shorter tree to the taller tree. This prevents the tree from becoming a long linked list (which would be slow to traverse).

Note 2: Real Interview Variations

Top companies rarely ask the "raw" version of this question. They wrap it in a real-world scenario.

1. Account Merging (Google / Meta) Scenario: You have a list of accounts. Each account has a `Name` and a list of `Emails`. If two accounts share a common email, they are actually the same person. Merge them and return the sorted list of emails for each person. **L6 Strategy:**

- Treat every **Email** as a node in the graph, not the Account itself.
- Loop through accounts. If an account has emails `[A, B, C]`, treat this as edges: `A-B` and `A-C`.
- Run **Union-Find** on the emails.
- Group all emails by their "Root Parent" email.

2. Graph Valid Tree (Bloomberg / Google) Scenario: Given `n` nodes and `edges`, determine if they form a valid Tree. **L6 Strategy:**

- A valid tree must have exactly **1 Connected Component** and **No Cycles**.

- Use **Union-Find**.

- Initialize `count = n`.

- For each edge `[u, v]`:

- If `find(u) == find(v)`, you found a **Cycle**! Return `False` immediately.

- Else, `union(u, v)` and `count--`.

- At the end, return `count == 1`.

3. Number of Islands (Grid Version) (Amazon / Google) Scenario:
Given a 2D grid of `1`s (land) and `0`s (water), count the number of islands. **L6
Strategy:**

- While DFS/BFS is common here, you *can* use Union-Find.
- Treat every `(row, col)` cell with a `1` as a node.
- Map 2D coordinates to a 1D index: `index = row * width + col`.
- Iterate through the grid. If `grid[r][c] == 1`, check its neighbors (down
  and right). If a neighbor is also `1`, `union` them.
- This is preferred if the grid is infinite or streaming (dynamic updates).

939. Minimum Area Rectangle

Hello! As a senior engineer (L5/L6) at Google, I look for three things when solv-
ing a problem like this: **Clarity**, **Trade-offs**, and **Readability**. We aren't just
trying to get the code to pass; we are trying to write code that is maintainable
and logically sound.

Here is how I would break down **Leetcode 939. Minimum Area Rectangle**.

---

1. Problem Explanation

The goal is to find the smallest area of a rectangle formed by a given set of 2D
points.
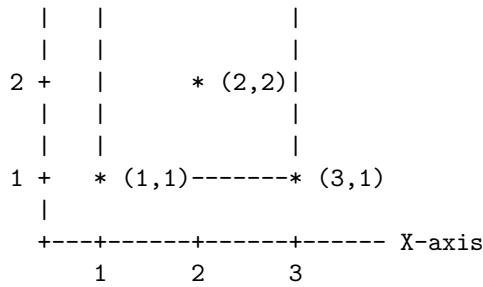
Crucial Constraints:

1. The rectangle's sides must be **parallel** to the X and Y axes. This means
   no rotated rectangles.
2. If no rectangle can be formed, return 0.

Visualizing the Grid: Imagine we have the following points: `(1,1)`, `(1,3)`,
`(3,1)`, `(3,3)`, `(2,2)`.

```
Y-axis
  |
3 +   * (1,3)        * (3,3)
```

```
  |   |                |
  |   |                |
2 +   |       * (2,2)|
  |   |                |
  |   |                |
1 +   * (1,1)-------* (3,1)
  |
  +---+------+------+------ X-axis
      1      2      3
```

- **Can we form a rectangle?** Yes! The points (1,1), (1,3), (3,1), and (3,3) form a perfect rectangle. The sides are vertical (x=1, x=3) and horizontal (y=1, y=3).
- **What about (2,2)?** It's just floating there. It cannot form a rectangle with the others because there are no matching corners at (1,2) or (3,2).
- **The Area Calculation:** Width = |x1 - x2| = |3 - 1| = 2 Height = |y1 - y2| = |3 - 1| = 2 Area = 2 * 2 = 4.

---

2. Solution Explanation

A junior engineer might try to find all groups of 4 points and check if they make a rectangle. That is $O(N^4)$—too slow. A mid-level engineer might try to find all pairs of X-coordinates and Y-coordinat

**The L5/L6 Approach: "The Diagonal Strategy"**
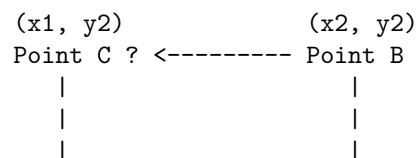
We exploit a geometric property: **A rectangle aligned with the axes is uniquely defined by its two diagonal corners.**

If I pick two points, say **Point A** and **Point B**, and treat them as the diagonal corners, the other two corners (**Point C** and **Point D**) are mathematically determined. We just need to check if C and D exist in our input list.

Visual Logic:

Let's pick two points from our list: A = (1, 1) and B = (3, 3).

1. **Identify Diagonal:** We assume A and B are diagonals.
2. **Determine Missing Corners:** If A is (x1, y1) and B is (x2, y2), then:

- Corner C must be at (x1, y2) -> (1, 3)
- Corner D must be at (x2, y1) -> (3, 1)

```
 (x1, y2)              (x2, y2)
 Point C ? <--------- Point B
    |                    |
    |                    |
    |                    |
```

```
 Point A  --------> Point D ?
 (x1, y1)           (x2, y1)
```

3. **Verification:** Does our list of points contain (1, 3) and (3, 1)?

- **Yes:** We found a rectangle! Calculate area: |1-3| * |1-3| = 4.
- **No:** These two points A and B cannot form a diagonal of a valid rectangle.

4. **Iteration:** We loop through every pair of points in our list to ensure we don't miss any combination.

Why is this fast? Checking if a point exists is **O(1)** if we put all points into a **Hash Set**. We iterate through all pairs of points, which is **O(N^2)**. This is significantly faster than O(N^3) or O(N^4).

---

3. Time and Space Complexity Analysis

Time Complexity Derivation:

```
Step 1:   Add all points to a Hash Set for O(1) lookups.
          Cost: O(N) where N is the number of points.


Step 2:   Double Loop to compare every point with every other point.


          Point 1 (i)    Point 2 (j)    Operation
          ------------------------------------------------------
          (x1, y1)       (x2, y2)       Check Set for (x1, y2) & (x2, y1)
                                        ^ This check is O(1)


          Loop i from 0 to N:
              Loop j from i+1 to N:
                  Work done inside is constant O(1).


          Total Iterations = N * (N-1) / 2
                           ~ (N * N) / 2


          Dominant Term: N^2


Total Time Complexity: O(N^2)
```

Space Complexity Derivation:

```
Storage: We need a Hash Set to store all the points for fast lookup.


          [ (1,1), (1,3), (3,1)... ]  <-- Stored in Set


          If we have N points, the Set size is N.
```

```
Total Space Complexity: O(N)
```

---

4. Solution Code

Here is how we write this in production-quality code.

Python Implementation *Comments explain the "why", not just the "how".*

```python
def minAreaRect(points):
    """
    Calculates the minimum area of a rectangle formed by a set of points.

    Strategy:
    1. Store points in a set for O(1) access.
    2. Iterate through every unique pair of points.
    3. Treat the pair as diagonal corners (p1, p2).
    4. Calculate where the other two corners must be.
    5. Check if those corners exist in the set.
    """

    # Use a set for O(1) lookups.
    # Python tuples (x, y) are hashable and can be stored directly.
    point_set = set()
    for x, y in points:
        point_set.add((x, y))

    min_area = float('inf')
    found = False

    # Iterate through all pairs.
    # We compare points[i] with points[j].
    for i in range(len(points)):
        p1_x, p1_y = points[i]

        for j in range(i + 1, len(points)):
            p2_x, p2_y = points[j]

            # Filter 1: If x or y coordinates match, they are on the same line.
            # They cannot be diagonal corners of a rectangle. Skip.
            if p1_x == p2_x or p1_y == p2_y:
                continue

            # Filter 2: Potential Optimization (Pruning)
            # If the current area we are checking is already larger than
```

```python
            # our best found 'min_area', strictly speaking, we don't *need* # to check exis
            # we usually just check existence first.

            # Identify the required other corners
            # Corner 3: x from p1, y from p2
            # Corner 4: x from p2, y from p1
            if (p1_x, p2_y) in point_set and (p2_x, p1_y) in point_set:
                found = True
                current_area = abs(p1_x - p2_x) * abs(p1_y - p2_y)
                min_area = min(min_area, current_area)

    return min_area if found else 0
```

JavaScript Implementation *Note: JS Sets compare objects by reference. We cannot simply do* `set.has([1,3])`. *We must serialize coordinates to strings like "1,3".*

```javascript
/**
 * @param {number[][]} points
 * @return {number}
 */
var minAreaRect = function(points) {
    // We use a Set to store "x,y" strings for O(1) lookup.
    const pointSet = new Set();

    for (const point of points) {
        // Serialization: [1, 3] becomes "1,3"
        pointSet.add(`${point[0]},${point[1]}`);
    }

    let minArea = Infinity;

    // Double loop to check every pair as a diagonal
    for (let i = 0; i < points.length; i++) {
        const [x1, y1] = points[i];

        for (let j = i + 1; j < points.length; j++) {
            const [x2, y2] = points[j];

            // If they share an X or Y, they form a line, not a diagonal.
            if (x1 === x2 || y1 === y2) continue;

            // Check for the existence of the other two corners:
            // Corner 3: (x1, y2)
            // Corner 4: (x2, y1)
            const corner3 = `${x1},${y2}`;
            const corner4 = `${x2},${y1}`;
```

```
            if (pointSet.has(corner3) && pointSet.has(corner4)) {
                const currentArea = Math.abs(x1 - x2) * Math.abs(y1 - y2);
                minArea = Math.min(minArea, currentArea);
            }
        }
    }

    return minArea === Infinity ? 0 : minArea;
};
```

Note 1: Terms and Techniques

Spatial Hashing / Coordinate Hashing: The technique of converting 2D coordinates (x, y) into a format that allows **O(1)** retrieval is crucial here. In Python, we used Tuples (which are hashable). In Javascript, we used String Serialization ("x,y").

Why it applies here: Without hashing, checking if a point exists requires iterating through the list, which is **O(N)**. Since we do this check inside a double loop, the total time would balloon to **O(N³). Hashing reduces the inner check to O(1), keeping the total time at O(N**

Note 2: Real World & Interview Variations

Big tech companies rarely ask the "vanilla" version. Here is how Google, Meta, and Bloomberg have twisted this recently:

1. **The "Square" Constraint (Bloomberg)**

   - **Question:** "Find the minimum area **Square** instead of a rectangle."
   - **Solution Adjustment:** Inside the loop, before checking the `set`, add a check: `if (abs(x1 - x2) != abs(y1 - y2)) continue;` This ensures the sides are equal length.

2. **The "Maximum" Area (Meta)**

   - **Question:** "Find the **Maximum** area rectangle."
   - **Solution Adjustment:** Change `min_area` to `max_area` (initialize to 0) and use `max()` instead of `min()`.

3. **Points with Data/Pixels (Google - Real World Context)**

   - **Question:** "The points represent active pixels on a 4K screen. Find the bounding box of the smallest rectangular region of interest."
   - **Solution Adjustment:** This hints that the coordinates might be dense. If the grid is dense (like an image matrix), we might use a **2D Boolean Array `grid[x][y] = true`** instead of a Hash Set.

- *Hash Set overhead:* Good for sparse points (scattered).
- *2D Array overhead:* Good for dense points (limited range, e.g., 1000x1000).

4. The "Anti-Diagonal" Transformation (Advanced)

- **Question:** "What if the rectangle can be rotated by 45 degrees?"
- **Solution:** This is much harder. You usually have to transform coordinates: `x' = x + y y' = x - y` Then solve the standard problem on the new `(x', y')` coordinates.

14. Longest Common Prefix

Here is a breakdown of how a Senior (L5) or Staff (L6) Engineer at a major tech company would approach **LeetCode 14: Longest Common Prefix**.

At this level, the focus shifts from just "getting it right" to **code readability, maintainability, handling edge cases gracefully, and understanding the theoretical limits.**

---

1. Problem Explanation

The Goal: You are given an array of strings (e.g., `["flower", "flow", "flight"]`). You need to find the longest string that is a prefix of *all* strings in that array.

If there is no common prefix, return an empty string `""`.

The "L6" Mental Model: Imagine the strings are written on a piece of paper, one below the other, left-aligned. You want to draw a vertical line as far right as possible such that all characters to the left of the line are identical in every row.

Visualizing the Problem:

Example 1: The Happy Path Input: `["flower", "flow", "flight"]`

```
      Column:  0   1   2   3   4   5
              +---+---+---+---+---+---+
String 1:     | f | l | o | w | e | r |
              +---+---+---+---+---+---+
String 2:     | f | l | o | w |
              +---+---+---+---+
String 3:     | f | l | i | g | h | t |
              +---+---+---+---+---+---+
                ^   ^   ^
                |   |   |
      Check 0: OK  OK  Mismatch! ('o' != 'i')
```

- **Index 0:** All strings have 'f'. (Keep going)
- **Index 1:** All strings have 'l'. (Keep going)
- **Index 2:** String 1 has 'o', String 2 has 'o', but String 3 has 'i'. **STOP.**

Result: "fl"

Example 2: The "Shortest String" Constraint Input: ["ab", "a"]

```
     Column:  0   1
             +---+---+
String 1:    | a | b |
             +---+---+
String 2:    | a |
             +---+
               ^   ^
               |   |
    Check 0: OK  Stop!
```

- **Index 0:** All have 'a'.
- **Index 1:** We check String 1 ('b'). But String 2 doesn't even *have* an index 1. It ran out of characters. **STOP.**

Result: "a"

Example 3: No Common Prefix Input: ["dog", "racecar", "car"]

```
     Column:  0   1   2
             +---+---+---+
String 1:    | d | o | g |
             +---+---+---+
String 2:    | r | a | c | ...
             +---+---+---+
               ^
               |
    Check 0: Mismatch immediately! ('d' != 'r')
```

Result: "" (Empty String)

---

2. Solution Explanation

The Algorithm: Vertical Scanning

While there are many ways to solve this (Horizontal Scanning, Sorting, Tries), a Senior Engineer often prefers **Vertical Scanning** for this specific Leetcode context.

Why?

1. **Fail-Fast:** It stops the moment a mismatch is found. If the first string is 1000 characters long but the last string differs at the very first charac-

ter, Vertical Scanning does very little work. Horizontal scanning might waste time merging the first two long strings before realizing the third one doesn't match.

2. **Space Efficiency:** We don't need to create new intermediate strings. We just look at the input.

3. **Simplicity:** It mirrors how a human reads the list.

Step-by-Step Logic:

1. **Guard Clause:** If the input list is empty, return `""`. (Always handle null/empty inputs first!).

2. **The Anchor:** Treat the first string in the list (`strs[0]`) as our "Anchor". We will compare every other string against this one.

3. **The Loop:** Iterate through the characters of the Anchor string (index `i` from 0 to length).

- **Inner Loop:** For each character `i`, look at *every other string* in the list.
- **Check 1 (Length):** Is the current string too short? (i.e., is its length equal to `i`?). If yes, we can't go further. **Return everything up to `i`.**
- **Check 2 (Match):** Does the character at index `i` of the current string match the character at index `i` of the Anchor? If no, **Return everything up to `i`.**

4. **Completion:** If we finish the outer loop without returning, it means the entire Anchor string is the prefix (usually happens if the Anchor is the shortest string). Return the Anchor.

---

3. Time and Space Complexity Analysis

Time Complexity: O(S)

- Where **S** is the sum of all characters in all strings.

Visual Derivation:

Imagine the worst-case scenario where *all* strings are identical, like `["aaaa", "aaaa", "aaaa"]`.

```
Grid of Characters:
  c c c c   <-- String 1
  c c c c   <-- String 2
  c c c c   <-- String 3
  c c c c   <-- String 4

Scan Path:
  | | | |
  v v v v
  1 2 3 4 ...
```

We visit every single character in the grid exactly once to verify they are all the same. Since the total number of characters is S, the time taken is proportional to S.

In the best case (mismatch at index 0), we perform `O(N)` comparisons (where N is the number of strings) and exit immediately. This is much faster than S, but Big-O notation cares about the worst-case upper bound.

Space Complexity: O(1)

- We are not using any extra data structures (like arrays, hash maps, or trees) that grow with the input size.
- We only use a few variables for pointers (`i`, `j`).
- *Note on Result:* Even though we return a string of length M, usually output space is not counted towards auxiliary space complexity. If strictly counting result space, it is **O(M)** where M is the length of the prefix.

---

4. Solution Code

Python Solution

```python
from typing import List

class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        """
        Finds the longest common prefix string amongst an array of strings.
        Method: Vertical Scanning.
        """
        # 1. Edge Case: Handle empty input list immediately.
        # An L6 engineer always protects against invalid state first.
        if not strs:
            return ""

        # The "Anchor" is the first string.
        # We will check if every other string matches this one column by column.
        anchor = strs[0]

        # Iterate through every character index of the anchor string
        for i in range(len(anchor)):
            char_to_match = anchor[i]

            # Check this column 'i' against every other string
            for s in strs[1:]:
                # Condition A: The string 's' is shorter than the current index 'i'.
                # We reached the end of string 's', so we cannot extend the prefix.
```

```python
                # Condition B: The characters do not match.
                if i == len(s) or s[i] != char_to_match:
                    # Return the prefix found SO FAR (from start up to, but not including,
                    return anchor[:i]

        # If we exit the loop naturally, it means the entire 'anchor' string
        # was found as a prefix in every other string.
        return anchor
```

JavaScript Solution

```javascript
/**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function(strs) {
    // 1. Edge Case: Empty list check
    if (!strs || strs.length === 0) {
        return "";
    }

    // 2. The Anchor strategy
    // We only need to iterate up to the length of the first string.
    // Optimization: In JS, accessing length in a loop is fast,
    // but caching scanning logic is cleaner.
    for (let i = 0; i < strs[0].length; i++) {

        // The character we expect to see in this column
        const charToMatch = strs[0][i];

        // 3. Scan the vertical column
        // Start from j = 1 because we are comparing against strs[0]
        for (let j = 1; j < strs.length; j++) {

            // Check if we went out of bounds (strs[j] is too short)
            // OR if the character doesn't match
            if (i === strs[j].length || strs[j][i] !== charToMatch) {
                // Return everything from start (0) up to current index (i)
                return strs[0].substring(0, i);
            }
        }
    }

    // If loop finishes, the whole first string is the common prefix
    return strs[0];
};
```

Note 1: Terms & Techniques Used

- **Vertical Scanning:** This refers to processing the data "column by column" rather than "row by row." Instead of reading String 1 entirely, then String 2 entirely, we read Index 0 of all strings, then Index 1 of all strings. This is highly effective for "Prefix" problems because the moment a column is inconsistent, the entire operation is invalid, allowing us to abort early.
- **Fail-Fast:** A design philosophy where a system immediately reports at its interface any condition that is likely to lead to failure. In this algorithm, we return the result the micro-second we detect a mismatch, rather than calculating everything and checking at the end.

Note 2: Indirect (Real World / Interview) Variants

The "Top of the Band" engineer differentiates themselves by answering the "So what?" question. Here is how Google, Meta, and Bloomberg actually ask this question in system design or advanced coding rounds:

1. The "Typeahead" / Autocomplete (Meta / Google) The Question: "Design the backend for a search bar. When a user types 'goo', we want to suggest 'google', 'good', 'goofy'. How do we quickly find the common starting sequence for millions of words?"

- **Why the Basic Solution Fails:** iterating through a list of 1 million strings (O(S)) every time a user presses a key is too slow.
- **The L6 Solution:** Use a **Trie (Prefix Tree)**.
- You insert all words into a tree structure.
- The root branches into 'g', which branches into 'o', etc.
- Finding the common prefix or suggestions becomes `O(M)` (where M is the length of the search query), totally independent of the total number of words in the database.

2. The "DNA Sequence Alignment" (Biotech / Google Health) The Question: "Given huge DNA strings (A, C, T, G), find the longest common sequence. The data is too large to fit in RAM."

- **Why the Basic Solution Fails:** You can't load `strs` into memory to run `strs[0]`.
- **The L6 Solution: Divide and Conquer (MapReduce)**.
- Split the data into chunks.
- Machine A finds the prefix for the first 1M strings. Machine B finds it for the next 1M.

70

- Combine the results: `GlobalPrefix = LCP(Result_A, Result_B)`.
- This is "Associative," meaning `LCP(s1, s2, s3) = LCP(LCP(s1, s2), s3)`.

3. The "Stock Ticker Grouping" (Bloomberg) The Question: "We have a stream of incoming stock symbols (`GOOG`, `GOOGL`, `GOLD`). We need to group them by hierarchy in real-time."

- **The L6 Approach:** This assumes the data is **sorted**.
- If the list is sorted: `["flight", "flow", "flower"]`.
- You ONLY need to compare the **First** and the **Last** string.
- If the first and last strings match on "fl", every string in between MUST also match on "fl" (due to alphabetical sorting).
- This turns the complexity from `O(S)` to `O(M * N)` in the worst case, but practically `O(M)` comparisons where M is length of first/last string.

695. Max Area of Island

Here is how a Senior (L5/L6) Engineer approaches "Max Area of Island".

At this level, the focus shifts from "getting code that works" to "writing code that is readable, maintainable, and models the problem as a generic graph traversal." The goal is to identify **Connected Components** in a grid.

---

1. Problem Explanation

Imagine you are analyzing a satellite image of an ocean. The image is pixelated into a grid.

- **Land** is represented by `1`.
- **Water** is represented by `0`.

We need to find the **biggest island**.

- An island is formed by connecting adjacent land cells (`1`s).
- Connections are only **Horizontal** (Left/Right) or **Vertical** (Up/Down). Diagonal connections do not count.
- The **Area** is simply the number of `1`s in that connected group.

Visualizing the Grid   Let's look at a sample 5x5 Grid.

Input Grid:

```
  0 1 2 3 4  (Columns)
0 . . 1 . .
1 . 1 1 . .
2 . . . . 1
```

```
3 1 1 . . 1
4 1 1 . . 1
```

(Note: I am using `.` for `0` (water) to make the `1s` (land) pop out visually).

Visual Analysis:

Island A (Top Cluster):

```
. . 1 . .
. 1 1 . .
```

This group has three `1s` connected. **Area = 3**

Island B (Bottom Left Cluster):

```
1 1 . . .
1 1 . . .
```

This group has four `1s` connected. **Area = 4**

Island C (Right Vertical Strip):

```
. . . . 1
. . . . 1
. . . . 1
```

This group has three `1s` connected vertically. **Area = 3**

Result: The maximum area is **4** (from Island B).

---

2. Solution Explanation: The "Flood Fill" (DFS)

An L5 engineer recognizes this immediately as a **Graph Traversal** problem. Specifically, finding **Connected Components**.

We can use **Depth First Search (DFS)**. Think of this as the "Flood Fill" tool in MS Paint or Photoshop. When you click a pixel, the color spreads to all connected pixels of the same color.

The Algorithm Strategy

1. **Scan:** Loop through every cell in the grid, row by row, column by column.
2. **Detect:** If we find a `1` (Land) that we haven't visited yet, it means we found a **new island**.
3. **Measure (DFS):**

- Pause the main loop.
- Jump into that cell and count it (`Area = 1`).
- Look at its 4 neighbors (Up, Down, Left, Right).
- If a neighbor is also `1`, jump into it, count it (`Area + 1`), and repeat the process from there.

- **Crucial Step:** As we visit a cell, we must **mark it as visited** (usually by changing the `1` to `0` or `-1`) so we don't count it infinitely.

4. **Record:** Once the DFS finishes for that island, compare its area to our current `max_area`.
5. **Resume:** Continue the main loop scanning for the next unvisited `1`.

Step-by-Step Execution Visualization **Initial State:** `max_area = 0`

```
(0,0) (0,1) (0,2) ...
  .     .     1    ...
  .     1     1    ...
```

Step 1: The Main Loop Scan We scan (0,0)... it's `0`. Skip. We scan (0,1)... it's `0`. Skip. We scan (0,2)... It is **1**! We found land.

Step 2: Start DFS at (0,2)

- Count (0,2). **Current Area = 1**.
- Mark (0,2) as visited (change to **#**).

```
. . # . .
. 1 1 . .
```

- **Recursion:** Look neighbors.
- Up? Out of bounds.
- Left? `0`. Stop.
- Right? `0`. Stop.
- **Down?** It is (1,2) which is `1`. **GO THERE.**

Step 3: DFS moves to (1,2)

- Count (1,2). **Current Area = 1 + 1 = 2**.
- Mark (1,2) as visited (**#**).

```
. . # . .
. 1 # . .
```

- **Recursion:** Look neighbors of (1,2).
- Up? **#** (Visited). Stop.
- Right? `0`. Stop.
- Down? `0`. Stop.
- **Left?** It is (1,1) which is `1`. **GO THERE.**

Step 4: DFS moves to (1,1)

- Count (1,1). **Current Area = 2 + 1 = 3**.
- Mark (1,1) as visited (**#**).

```
. . # . .
. # # . .
```

- **Recursion:** Look neighbors of (1,1).

- All neighbors are either `0` or `#`.
- **Dead end.** Return 3.

Step 5: Completion DFS returns. We found an island of size **3**. Update `max_area = 3`. Resume main loop scan. When the loop reaches (1,1) or (1,2), it sees `#` (or `0`) and ignores them.

---

3. Time and Space Complexity Analysis

Time Complexity: O(R * C)

- **R** = Number of Rows
- **C** = Number of Columns

Why? We visit every cell exactly twice in the worst case:

1. Once by the **Main Loop** (scanning for a start).
2. Once by the **DFS function** (measuring the island).

Visualization of Work:

```
Main Loop Scan  +   DFS Visits   = Total Work
[-> -> -> ->]       [ 1 -> 1 ]
[-> -> -> ->]       [ |    | ]      O(R * C)
[-> -> -> ->]       [ 1 <- 1 ]
```

We never revisit a processed node because we mark it as `0` or visited immediately.

Space Complexity: O(R * C)

- **Worst Case:** The entire grid is land (`1s`).

Why? We are using recursion. Recursion uses the "Call Stack" (system memory). If the grid is one giant snake-like island, the recursion will go deep.

Visualizing the Stack:

```
Grid (Snake):       Recursion Stack (Memory):
1 1 1 1             dfs(0,0) waits for -> dfs(0,1)
. . . 1             dfs(0,1) waits for -> dfs(0,2)
1 1 1 1             dfs(0,2) waits for -> dfs(0,3)
1 . . .             ...
1 1 1 1             ... and so on covering all cells.
```

In the worst case, the stack height equals the number of cells: `R * C`.

---

4. Solution Code

We will write a `dfs` helper function.

74

Python Solution   *This solution modifies the input grid to save space (sets visited cells to 0). If the interviewer asks not to modify input, we would use a separate* `set()` *called* `visited`.

```python
from typing import List


class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        """
        Calculates the maximum area of an island in a binary grid.

        Args:
            grid: 2D list of integers (0 or 1).

        Returns:
            Integer representing the maximum area.
        """
        rows = len(grid)
        cols = len(grid[0])
        max_area = 0

        # Helper Function: DFS
        # This function visits a cell, counts it, sinks it (marks as 0),
        # and recursively visits neighbors.
        def dfs_measure_island(r, c):
            # 1. Base Case: Check boundaries and if cell is water (0)
            if r < 0 or r >= rows or c < 0 or c >= cols or grid[r][c] == 0:
                return 0

            # 2. Mark as visited ('Sink' the island cell so we don't count it again)
            grid[r][c] = 0

            # 3. Recursive Step: Sum 1 (current) + neighbors
            area = 1
            area += dfs_measure_island(r + 1, c) # Down
            area += dfs_measure_island(r - 1, c) # Up
            area += dfs_measure_island(r, c + 1) # Right
            area += dfs_measure_island(r, c - 1) # Left

            return area

        # Main Loop: Scan the grid
        for r in range(rows):
            for c in range(cols):
                # If we find land, start measuring
                if grid[r][c] == 1:
```

```
                current_island_area = dfs_measure_island(r, c)
                max_area = max(max_area, current_island_area)

        return max_area
```

JavaScript Solution

```javascript
/**
 * @param {number[][]} grid
 * @return {number}
 */
var maxAreaOfIsland = function(grid) {
    const rows = grid.length;
    const cols = grid[0].length;
    let maxArea = 0;

    // Helper Function: DFS
    // Returns the area of the connected component starting at (r, c)
    const measureIsland = (r, c) => {
        // Boundary and Water checks
        if (r < 0 || r >= rows || c < 0 || c >= cols || grid[r][c] === 0) {
            return 0;
        }

        // Mark visited (Sink the cell)
        grid[r][c] = 0;

        // Accumulate area: 1 (self) + 4 directions
        return 1 +
                measureIsland(r + 1, c) + // Down
                measureIsland(r - 1, c) + // Up
                measureIsland(r, c + 1) + // Right
                measureIsland(r, c - 1);  // Left
    };

    // Main Scan
    for (let r = 0; r < rows; r++) {
        for (let c = 0; c < cols; c++) {
            if (grid[r][c] === 1) {
                const currentArea = measureIsland(r, c);
                maxArea = Math.max(maxArea, currentArea);
            }
        }
    }

    return maxArea;
```

```
};
```

Note 1: Terms and Techniques

Connected Components: In graph theory, a connected component is a subgraph in which any two vertices are connected to each other by paths. Here, the "pixels" are vertices and adjacency is the "edge".

- **Why it helps:** It abstracts the problem. Instead of "islands in water", we just think "nodes connected by edges".
- **Application:** This is the exact same logic used to find friends in a social network (Friend Circles) or connected devices in a network.

Note 2: Real World & Interview Variations

Interviews at top companies (Google, Meta, Bloomberg) rarely ask the vanilla version of this question anymore. Here is how they twist it and how to solve it.

1. The "Lake" or "Enclave" Problem (Google/Bloomberg) Question: "Find the number of islands that are completely surrounded by water (i.e., they do NOT touch the edge of the grid)." **Solution Adjustment:**

- Run the DFS.
- Track a boolean flag `touches_boundary` inside the recursion.
- If `r == 0` or `r == max` etc., set `touches_boundary = True`.
- Only count the island if `touches_boundary` remains `False` after the DFS completes.

2. The "Largest Island after Flipping one 0 to 1" (Harder Version - Google) Question: "You can change exactly one `0` to a `1`. What is the largest possible island you can create?" **Solution Adjustment:**

1. **ID the Islands:** Run the standard solution first, but instead of just counting area, assign a unique `ID` to each island (2, 3, 4...) and store their areas in a HashMap: `{ID_2: 5, ID_3: 8}`.
2. **Paint the Grid:** Modify the grid so every cell contains its Island ID.
3. **Check Candidate Zeros:** Iterate through every `0` in the grid.

- Look at its 4 neighbors.
- Get the unique IDs of those neighbors (e.g., this `0` connects ID_2 and ID_3).
- Sum: `1 (the flipped zero) + Area(ID_2) + Area(ID_3)`.

4. Return the max sum found.

3. Image Segmentation (Real World Application)   Scenario: "Given a black and white MRI scan where white pixels are 'tissue' and black is 'background', identify the tumor." **Solution:**

- This is the exact same problem. The tumor is usually the "largest connected component" of a specific density/color.
- You would run `Max Area of Island` on the MRI pixel grid to isolate the mass.

4. Distinct Islands (Meta)   Question: "Count the number of distinct islands, where 'distinct' means the **shape** is unique (ignoring rotation/reflection sometimes, or just translation)." **Solution Adjustment:**

- When running DFS, record the **path signature**.
- Instead of just returning area, record the steps taken: "Start, Down, Down, Right, Return".
- Store these path strings in a `Set`.
- The result is the size of the `Set`.

332. Reconstruct Itinerary

Here is a deep dive into **Leetcode 332. Reconstruct Itinerary**, explained from the perspective of a Google L6 Staff Software Engineer.

---

1. Problem Explanation

At its core, this problem asks us to organize a chaotic pile of flight tickets into a valid travel itinerary.

The Rules:

1. **Start:** You must always start at `JFK`.
2. **Use Everything:** You must use *every single ticket* exactly once.
3. **Connectivity:** The destination of ticket A must be the departure of ticket B.
4. **Tie-Breaking:** If you can fly to multiple cities from one airport, you must choose the one that comes first alphabetically (Lexicographical Order).
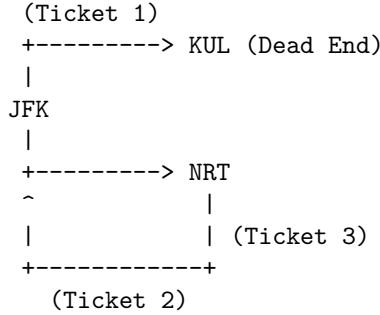
The Trap (Why this is hard): It looks like a simple "Greedy" problem: "Just stand at JFK, pick the alphabetically first city, go there, repeat."

Why Greedy Fails: Imagine you are at `JFK`. You have two tickets: `JFK -> A` and `JFK -> B`. Alphabetically, `A` comes before `B`. So you greedily fly to `A`. But what if `A` is a dead end? You are now stuck at `A` with the `JFK -> B` ticket still in your pocket. You failed to use all tickets.

Visualizing the Trap:

Let's look at this specific set of tickets: `[ ["JFK", "KUL"], ["JFK", "NRT"], ["NRT", "JFK"] ]`

The Graph View:

```
    (Ticket 1)
    +---------> KUL (Dead End)
    |
JFK
    |
    +---------> NRT
    ^           |
    |           | (Ticket 3)
    +-----------+
      (Ticket 2)
```

Scenario 1: The Greedy Mistake

1. Start at `JFK`.
2. Neighbors are `KUL` and `NRT`.
3. Alphabetically, `KUL` < `NRT`.
4. We fly `JFK -> KUL`.
5. **STUCK!** We are at `KUL`. We have no tickets leaving `KUL`. But we still have unused tickets (`JFK->NRT`, `NRT->JFK`).
6. **Result:** Invalid Itinerary.

Scenario 2: The Correct Path

1. Start at `JFK`.
2. We realize `KUL` leads to a dead end *too early*. We skip it for now.
3. We fly `JFK -> NRT`.
4. From `NRT`, we fly `NRT -> JFK`.
5. Back at `JFK`, we finally use the last ticket `JFK -> KUL`.
6. **Result:** `JFK -> NRT -> JFK -> KUL`. Success!

---

2. Solution Explanation: Eulerian Path (Hierholzer's Algorithm)

To an L6 engineer, this isn't just "recursion"; this is a graph theory problem called finding an **Eulerian Path**.

The Definition: An Eulerian path is a trail in a graph which visits every *edge* (ticket) exactly once.

The Strategy: "Get Stuck Last" The intuition is counter-intuitive. Instead of trying to find the *start* of the path, we aim to find the *end*.

We use a **Post-Order DFS (Depth First Search)**.

1. Start at `JFK`.

2. Pick the smallest neighbor (lexicographically) and *delete* that ticket so we don't use it again.
3. Recursively move to that neighbor.
4. Keep going until you are **stuck** (no outgoing tickets left from your current location).
5. **Crucial Step:** When you are stuck, it means you have reached the *end* of a valid segment. Add this airport to our result list.
6. Backtrack to the previous airport and see if it has other tickets.

This logic ensures that "dead ends" are added to the itinerary last. Since we are building the itinerary backwards (from the end to the start), we just reverse the list at the very end.

Detailed ASCII Walkthrough Tickets: `[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK`

Sorted Adjacency List (Graph):

- JFK: `[ATL, SFO]` (ATL is first alphabetically)
- ATL: `[JFK, SFO]`
- SFO: `[ATL]`

Visualization of the Algorithm:

Step 1: Start at `JFK`. Options: `ATL`, `SFO`. Pick `ATL`. *(Remove JFK->ATL ticket)*

```
Current Path: JFK -> ATL
Stack/Recursion: [JFK, ATL]
```

Step 2: At `ATL`. Options: `JFK`, `SFO`. Pick `JFK` (alphabetical). *(Remove ATL->JFK ticket)*

```
Current Path: JFK -> ATL -> JFK
Stack/Recursion: [JFK, ATL, JFK]
```

Step 3: At `JFK`. Options: `SFO` (Since ATL is already deleted). Pick `SFO`. *(Remove JFK->SFO ticket)*

```
Current Path: JFK -> ATL -> JFK -> SFO
Stack/Recursion: [JFK, ATL, JFK, SFO]
```

Step 4: At `SFO`. Options: `ATL`. Pick `ATL`. *(Remove SFO->ATL ticket)*

```
Current Path: JFK -> ATL -> JFK -> SFO -> ATL
Stack/Recursion: [JFK, ATL, JFK, SFO, ATL]
```

Step 5: At `ATL`. Options: `SFO`. Pick `SFO`. *(Remove ATL->SFO ticket)*

```
Current Path: ... -> SFO -> ATL -> SFO
Stack/Recursion: [..., SFO, ATL, SFO]
```

Step 6: At `SFO`. **NO TICKETS LEFT!** We are stuck. This means `SFO` must be the final destination of this segment. **Action:** Add `SFO` to `Result List`. Return to previous caller.

```
Result List: [SFO]
Backtrack to ATL.
```

Step 7: Back at `ATL`. Do we have other tickets from here? No (we used ATL->JFK and ATL->SFO). **Action:** Add `ATL` to `Result List`. Return to caller.

```
Result List: [SFO, ATL]
Backtrack to SFO.
```

Step 8: Back at `SFO`. No tickets left. **Action:** Add `SFO` to `Result List`.

```
Result List: [SFO, ATL, SFO]
Backtrack to JFK.
```

... (Continuing backtracking) ...

Eventually, the `Result List` looks like this: `[SFO, ATL, SFO, JFK, ATL, JFK]`

Final Step: Reverse the list. **Itinerary:** `JFK -> ATL -> JFK -> SFO -> ATL -> SFO`

---

3. Time and Space Complexity Analysis

Time Complexity: O(E * log E) where **E** is the number of tickets (edges).

Derivation:

```
1. Building the Graph:
   We iterate through E tickets.
   Cost: O(E)

2. Sorting the Destinations:
   For every airport, we sort its outgoing flights to satisfy lexical order.
   In the worst case, one airport has all E tickets.
   Sorting E items takes O(E * log E).
   Cost: O(E * log E)

3. DFS Traversal (Hierholzer's):
   We visit every edge (ticket) exactly once.
   We visit every vertex (airport) at least once.
   Cost: O(V + E)

Total Time = O(E) + O(E * log E) + O(V + E)
             O(E * log E) dominates.
```

Space Complexity: O(V + E)

Derivation:

1. Adjacency List (Graph Storage):
   We store every airport (V) and every ticket (E).
   Cost: O(V + E)

2. Recursion Stack:
   In the worst case (a long line of flights), the recursion goes E deep.
   Cost: O(E)

3. Result Storage:
   Stores the path, which has E+1 nodes.
   Cost: O(E)

Total Space = O(V + E)

---

4. Solution Code

Python Solution   This implementation uses a `stack` for iterative DFS to avoid recursion depth limits, though the recursive approach is also standard. We use `sort(reverse=True)` so we can pop efficiently from the end of the list (O(1)) instead of the front (O(N)).

```python
from collections import defaultdict


class Solution:
    def findItinerary(self, tickets: list[list[str]]) -> list[str]:
        # Step 1: Build the graph
        # Map: Airport -> List of Destinations
        graph = defaultdict(list)

        # We sort in reverse order immediately.
        # Why? Because in Python, popping from the end of a list is O(1).
        # Popping from the start is O(N).
        # By sorting reverse (Z->A), the "smallest" lexical item (A) is at the end.
        for src, dst in sorted(tickets, reverse=True):
            graph[src].append(dst)

        route = []
        stack = ["JFK"]

        # Step 2: DFS Traversal (Hierholzer's Algorithm)
        while stack:
            # Look at the current airport we are at
            curr = stack[-1]

            # Check if there are outgoing flights left from here
```

```python
            if curr in graph and len(graph[curr]) > 0:
                # Yes, there is a flight.
                # Pop the last destination (which is the lexically smallest because we rever
                next_dst = graph[curr].pop()

                # Push this new destination onto the stack to visit it next
                stack.append(next_dst)
            else:
                # No outgoing flights left from 'curr'. We are stuck!
                # This means 'curr' is part of the solution path (from the end).
                route.append(stack.pop())

        # Step 3: Reverse the route to get start -> end
        return route[::-1]
```

Javascript Solution   This uses the recursive approach for clarity.

```javascript
/**
 * @param {string[][]} tickets
 * @return {string[]}
 */
var findItinerary = function(tickets) {
    // Step 1: Build the graph
    const graph = {};

    // Sort tickets to ensure lexical order when we process them
    tickets.sort((a, b) => a[1] < b[1] ? -1 : 1);

    for (const [src, dst] of tickets) {
        if (!graph[src]) graph[src] = [];
        graph[src].push(dst);
    }

    const result = [];

    // Step 2: DFS Function
    const dfs = (node) => {
        const destinations = graph[node];

        // While there are outgoing edges from this node...
        while (destinations && destinations.length > 0) {
            // Remove the first ticket (smallest lexical due to previous sort)
            // shift() is O(N), but acceptable given constraints usually < 3000 tickets
            const nextNode = destinations.shift();
            dfs(nextNode);
        }
```

```
        // Step 3: We are stuck (no more edges). Add to result.
        // Unshift adds to the front, building the list backwards-to-forwards
        result.unshift(node);
    };

    dfs("JFK");
    return result;
};
```

New Concepts & Terms

Hierholzer's Algorithm:

- **What is it?** An efficient method to find an Eulerian Path (or Circuit) in a graph.
- **Why it helps here:** It is specifically designed to traverse every edge exactly once. It handles "sub-cycles" (flying `A->B->A`) naturally by simply merging that cycle into the main path.
- **Application:** Whenever you see "Reconstruct a path using all edges" or "Visit every connection exactly once," think Hierholzer's.

Indirect / Real-World Interview Variations

When companies like Google, Meta, or Bloomberg ask this, they rarely say "Here are plane tickets." They disguise the Eulerian Path problem in practical engineering scenarios.

1. The "DNA Sequence Reconstruction" (Bioinformatics)

- **The Problem:** You are given thousands of short DNA fragments (e.g., "ATGC", "TGCC", "GCCA"). You need to assemble them into one superstring (e.g., "ATGCC…").

- **The Connection:** This is identical to tickets.

- Fragment "ATGC" is a directed edge from prefix "ATG" to suffix "TGC".

- Fragment "TGCC" is a directed edge from "TGC" to "GCC".

- **Solution:** Construct a graph where nodes are (`k-1`)-mers (prefixes/suffixes) and edges are the fragments. Find the Eulerian Path to order the fragments correctly.

2. The "Domino Arrangement" (Game Theory / Logistics)

- **The Problem:** Given a set of dominoes (e.g., `[1|2]`, `[2|4]`, `[4|1]`, `[1|3]`), arrange them in a line such that adjacent numbers match (standard domino rules). Use all dominoes.

- **The Connection:**

- Nodes = The numbers on the dominoes (1, 2, 3, 4).

- Edges = The domino itself (Edge from 1 to 2).

- **Solution:** Since dominoes are undirected (you can flip them), this is finding an Eulerian Path in an *undirected* graph. The logic is the same: Visit every edge once.

3. The "Server Dependency / Job Scheduling" (Infrastructure - Rare Variation)

- **The Problem:** A system needs to execute a set of specific data transfer tasks between servers. Each task moves data from Server A to Server B. The network link is single-threaded; you can only perform one transfer at a time. Order the tasks so the connection flow is continuous to minimize handshake overhead (i.e., if you end at Server B, start the next task at Server B).

- **The Connection:**

- Nodes = Servers.

- Edges = Data Transfer Tasks.

- **Solution:** Construct the itinerary of transfers. This minimizes the "context switching" of the network controller jumping between unrelated servers.

1233. Remove Sub-Folders from the Filesystem

Here is how a Senior (L5) or Staff (L6) Google Engineer would deconstruct, solve, and expand upon this problem.

At this level, the focus shifts from "getting it right" to **efficiency, readability, and handling edge cases** (like the difference between `/a` and `/abc`). We care about the *why* as much as the *how*.

---

1. Problem Explanation

The Core Task: We have a list of folder paths. Some folders are actually "children" (sub-folders) of other folders in the same list. We want to delete the children and keep only the top-level parents.

The "Sub-Folder" Rule: A folder is a sub-folder if it sits inside a parent folder. Critically, it must start with the parent path **followed immediately by a forward slash `/**`.

Visual Examples:

- **Case A (True Sub-folder):**
- Parent: `/home`
- Child: `/home/user`
- *Result:* Remove `/home/user`.
- *Why?* The child starts with `/home` + `/`.
- **Case B (False Friend / Prefix Trap):**
- Folder 1: `/home`
- Folder 2: `/homepage`
- *Result:* Keep both.
- *Why?* Even though `/homepage` starts with `/home`, it does **not** follow with a `/`. It is a sibling folder, not a child.

Goal: Input: `["/a", "/a/b", "/c/d", "/c/d/e", "/c/f"]` Output: `["/a", "/c/d", "/c/f"]`

---

2. Solution Explanation

An L5/L6 engineer recognizes that **structure facilitates speed**. If the data is chaotic, we have to compare everyone against everyone (). If we organize the data, we can solve it much faster.

The Strategy: ASCII Sorting We sort the array of strings lexicographically (alphabetically).

Why Sorting Works (The Intuition): When folders are sorted, a parent folder will **always** appear immediately before its sub-folders. This allows us to pass through the list just once (linear scan) after sorting.

Visualizing the Sort:

Imagine the raw input: `["/x", "/a/b", "/a", "/x/y"]`

Step 1: Sort them.

```
Index | Path
------+-------
  0   |  /a
  1   |  /a/b
```

86

```
2   |  /x
3   |  /x/y
```

Step 2: Linear Scan with an "Anchor". We maintain a pointer (or variable) called `last_valid_parent`.

- **Iteration 0:**
- Current: `/a`
- Action: It's the first one. Keep it.
- `last_valid_parent` = `/a`
- *Result List:* `["/a"]`
- **Iteration 1:**
- Current: `/a/b`
- Check: Does `/a/b` start with `/a` + `/`?
- Logic: Yes. It is a sub-folder.
- Action: **Ignore it.**
- `last_valid_parent` stays `/a`
- **Iteration 2:**
- Current: `/x`
- Check: Does `/x` start with `/a` + `/`?
- Logic: No.
- Action: This is a new top-level folder. **Keep it.**
- `last_valid_parent` becomes `/x`
- *Result List:* `["/a", "/x"]`
- **Iteration 3:**
- Current: `/x/y`
- Check: Does `/x/y` start with `/x` + `/`?
- Logic: Yes.
- Action: **Ignore it.**

Final Output: `["/a", "/x"]`

---

3. Time and Space Complexity Analysis

At the Staff level, we don't just say "O(N log N)". We break it down by the number of paths () and the average length of the path string ().

Time Complexity:

```
Total Time = Sorting Time + Scanning Time
```

```
1. Sorting Time:
   +----------------------------------------------+
   |   Standard Sort (Merge/Quick Sort) takes     |
   |   O(N * log N) comparisons.                  |
   |                                              |
   |   BUT, comparing two strings isn't free.     |
   |   It takes O(L) to compare characters.       |
   +----------------------------------------------+
   |   Total Sort Cost: O(N * L * log N)          |
   +----------------------------------------------+
```

```
2. Scanning Time:
   +----------------------------------------------+
   |   We iterate through the list once: O(N)     |
   |   At each step, we do a prefix check.        |
   |   Prefix check takes O(L).                   |
   +----------------------------------------------+
   |   Total Scan Cost: O(N * L)                  |
   +----------------------------------------------+
```

```
GRAND TOTAL: O(N * L * log N)
```

Space Complexity:

```
1. Sorting Space:
   Language dependent. Python/Java sorts use O(N)
   or O(log N) stack space.
```

```
2. Output Space:
   We store the answer. In the worst case (no sub-folders),
   we store all inputs.
   Space: O(N * L)
```

```
GRAND TOTAL: O(N * L)
```

---

4. Solution Code

Python Solution

```python
class Solution:
    def removeSubfolders(self, folder: list[str]) -> list[str]:
        # 1. Sort the folders.
        # This groups parents and their children together.
        folder.sort()

        # Initialize result list with the first folder
        # (The first folder in a sorted list is always a top-level parent)
        result = [folder[0]]

        # 2. Iterate starting from the second folder
        for i in range(1, len(folder)):
            current_folder = folder[i]
            last_parent = result[-1]

            # THE TRICKY PART:
            # We add a trailing slash to the parent to ensure boundary accuracy.
            # Example: Parent "/a", Child "/abc".
            # "/abc".startswith("/a") is True (BAD match).
            # "/abc".startswith("/a/") is False (GOOD match).
            last_parent_with_slash = last_parent + '/'

            if not current_folder.startswith(last_parent_with_slash):
                # If it's not a sub-folder, it's a new parent. Add it.
                result.append(current_folder)

        return result
```

JavaScript Solution

```javascript
/**
 * @param {string[]} folder
 * @return {string[]}
 */
var removeSubfolders = function(folder) {
    // 1. Sort lexicographically (alphabetically)
    folder.sort();

    // Initialize with the first element
    const result = [folder[0]];

    for (let i = 1; i < folder.length; i++) {
        let currentFolder = folder[i];
        let lastParent = result[result.length - 1];

        // 2. The Edge Case Check
        // We ensure the current folder starts with the parent AND a slash.
```

```
      // This prevents "/a" matching with "/apple"
      if (!currentFolder.startsWith(lastParent + '/')) {
          result.push(currentFolder);
      }
   }

   return result;
};
```

---

5. Terminology & Concepts

Lexicographical Sort: This is just a fancy way of saying "Alphabetical Order," but it applies to every character in the ASCII table (numbers, symbols, uppercase, lowercase). In this problem, it guarantees that `/a` comes before `/a/b`.

Prefix Matching (Trie Logic): Although we used sorting, this problem is fundamentally about "prefixes." If String A is a prefix of String B (with a slash boundary), B is the child. We solved it with Sorting + Linear Scan, but it could also be solved using a **Trie** (a tree data structure used for storing strings), which is often how auto-complete works.

---

6. Real World & Interview Variations (Google / Meta / Bloomberg)

This is the most important section for an L5/L6 candidate. It shows you understand how this algorithmic toy applies to real systems.

Variation A: The "Access Control" List (Google Cloud / AWS IAM)
The Question: "We have a list of file paths a user has explicit access to. If a user has access to a folder, they have access to everything inside it. Minimize the size of this policy list."

- **Why it's asked:** Storage optimization. Storing 1,000,000 paths is expensive. Storing 1 parent path is cheap.
- **Solution Difference:** It is the exact same code as above. The "cleaning" reduces the policy document size.

Variation B: The "Infinite Stream" (System Design Context)   The Question: "The folder paths are coming in as a never-ending stream (logs). You cannot sort them because the list never ends. How do you filter subfolders?"

- **How an L6 solves it:**
- Sorting is impossible here.
- We use a **Trie (Prefix Tree)**.

- Insert paths into the Trie one by one.
- If you insert `/a/b` but `/a` already exists and is marked as a "Terminal Node" (a complete parent), you discard `/a/b` immediately.
- *Trade-off:* Higher memory usage, but handles real-time data.

Variation C: "Hidden API" (Bloomberg / Meta)   The Question: "You are given a list of IDs. You have a helper function `is_child(id1, id2)` that returns true/false. You cannot see the strings themselves. Remove the children."

- **How to solve:**
- You cannot sort strings because you can't see them.
- You must assume worst case comparison, **OR**
- You model this as a Graph problem (Transitive reduction). If A -> B and B -> C, we just need A -> ... -> C.
- Usually, they want you to realize that without the string structure, you are forced into a slower complexity class.

Variation D: The "S3 Bucket" Listing   The Question: "S3 is a flat object store (no real folders). We have keys like `photos/2023/jan/dog.jpg`. We want to emulate a folder view for the UI. Given a flat list of 1 billion keys, how do we display just the top-level folders?"

- **Solution:**
- This is a "group by" problem.
- Split strings by `/`.
- Use a Hash Map (or Set) to collect unique prefixes at the current depth.
- Set: `{"photos/", "documents/", "videos/"}`.
- Return that list. (This is distinct from the main problem but is the standard follow-up).

994. Rotting Oranges

Here is how a Senior Staff Engineer (L6) at Google would break down, analyze, and solve the "Rotting Oranges" problem.

At this level, we don't just look for "code that passes." We look for:

1. **Pattern Recognition:** Mapping the problem to a standard algorithmic paradigm (BFS).
2. **Scalability:** Handling the "multi-source" nature efficiently.
3. **Readability:** Code that a junior engineer can maintain.

---

1. Problem Explanation

Imagine a warehouse grid where boxes of oranges are stored.

- **0**: Empty cell.
- **1**: Fresh orange.
- **2**: Rotten orange.

The Rules of Decay: Every minute, any **fresh orange** that is 4-directionally adjacent (up, down, left, right) to a **rotten orange** becomes rotten.

The Goal: Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible (some fresh oranges are isolated), return **-1**.

Visualization of the Scenario   Let's look at a grid. R is Rotten, F is Fresh, _ is Empty.

Initial State (Minute 0):

```
(0,0)   (0,1)   (0,2)
  R       F       F
  ------------------
  F       F       _
  ------------------
  F       _       F
(2,0)   (2,1)   (2,2)
```

What happens next?   The decay behaves like a **wave**. It spreads layer by layer from the contamination source.

- **Minute 1:** The R at (0,0) infects its direct neighbors: (0,1) and (1,0).
- **Minute 2:** The newly rotten oranges at (0,1) and (1,0) infect *their* neighbors.

If there is a Fresh orange at (2,2) surrounded by Empty cells _, the rot can never reach it. We must detect this.

---

2. Solution Explanation

The Intuition: Why Breadth-First Search (BFS)?   An L6 engineer immediately identifies this as a **shortest path problem** on a grid.

- **Why not DFS (Depth First Search)?** DFS dives deep. It would trace one rotting path all the way to the end before checking neighbors. That doesn't match the physics of the problem; rotting happens *simultaneously* in all directions.
- **Why BFS?** BFS explores "layer by layer." It processes everything at distance 1 (Minute 1), then everything at distance 2 (Minute 2). This perfectly mimics the passage of time.

The "Multi-Source" Twist Most basic BFS problems start from a single point (e.g., "Find the path from A to B"). Here, we might have **multiple** rotten oranges at Minute 0. We don't run a separate BFS for each rotten orange (that would be inefficient and overlap). Instead, we put **ALL** initially rotten oranges into the Queue at the very beginning. This creates a "Multi-Source BFS."

Detailed Visual Walkthrough **The Setup:** We need a Queue to track which oranges are currently rotten and ready to infect others. We also need a `fresh_count` to track how many healthy oranges are left.

Grid:

```
  0   1   2
0 R   F   F
1 F   H   _    <-- 'H' is just a Fresh orange we labeled H to track it
2 _   _   -
```

Note: (0,0) is R (Rotten).

Step 1: Initialization

- Scan the grid.
- Count Fresh oranges = 3 (at 0,1; 0,2; 1,0; 1,1).
- Add `(0,0)` to Queue.
- Time = 0.

```
QUEUE State: [ (0,0) ]
FRESH COUNT: 4
```

Step 2: Minute 1 Processing

- Pop `(0,0)`.

- Look at neighbors of `(0,0)`:

- `(0,1)` is Fresh. **Rot it!** Add to Queue. Dec Fresh Count.

- `(1,0)` is Fresh. **Rot it!** Add to Queue. Dec Fresh Count.

- We finished the current layer (size 1). Increment Time.

```
VISUALIZATION T=1:
  R   R   F   <-- (0,1) just turned
  R   H   _   <-- (1,0) just turned

  -   -   -

QUEUE State: [ (0,1), (1,0) ]
FRESH COUNT: 2
TIME: 1
```

Step 3: Minute 2 Processing

- We process the *entire* current layer (size 2).

93

- **Pop (0,1)**:
- Neighbor (0,0) is already rotten (ignore).
- Neighbor (0,2) is Fresh. **Rot it!** Add to Queue. Dec Fresh Count.
- Neighbor (1,1) (labeled H) is Fresh. **Rot it!** Add to Queue. Dec Fresh Count.
- **Pop (1,0)**:
- Neighbor (0,0) is rotten.
- Neighbor (1,1) is now rotten (already handled).
- Neighbor (2,0) is empty.
- Layer finished. Increment Time.

```
VISUALIZATION T=2:
  R   R   R   <-- (0,2) just turned
  R   R   _   <-- (1,1) just turned

  -   -   -

QUEUE State: [ (0,2), (1,1) ]
FRESH COUNT: 0
TIME: 2
```

Step 4: Termination

- Fresh count is 0. We are done.
- **Return Time: 2**.

---

3. Time and Space Complexity Analysis

Time Complexity: O(R * C)

- **R** = Number of Rows
- **C** = Number of Columns

Visual Derivation: Imagine the grid is a flat list of cells.

`[ Cell 1 ] -> [ Cell 2 ] -> ... -> [ Cell N ]`

1. **First Pass:** We iterate through the whole grid once to find the initial rotten oranges and count fresh ones. Cost: R * C.
2. **BFS Process:** In the worst case (all oranges are fresh and become rotten), every single cell is added to the Queue exactly **once** and popped exactly **once**.
3. For each cell, we look at its 4 neighbors. That is a constant 4 operations.

Total Cost: `Initial Scan (R*C) + BFS Processing (R*C)` Since we drop constants in Big-O notation, the result is $O(R * C)$ or $O(N)$ where N is the total number of cells.

Space Complexity: $O(R * C)$ Visual Derivation: We need space for the Queue. How big can the Queue get?

Consider a grid that is all Rotten oranges? No, the queue processes them, but the max size happens when the "wave" of rotting is largest.

Worst Case Scenario: Expanding Circle If the rotting starts in the center, the "frontier" (the layer of currently rotting oranges) grows outwards.

```
. . . R . . .
. . R R R . .
. R R R R . .   <-- The Queue holds the perimeter
. . R R R . .
. . . R . . .
```

In a grid, the maximum number of items in the queue generally won't exceed the total number of cells. Specifically, for a filled grid, the queue size is roughly bounded by the perimeter or diagonal, but safe upper bound is $O(R * C)$.

---

4. Solution Code

Python Solution   *Using* `deque` *for efficient O(1) pops from the left.*

```python
from collections import deque

def orangesRotting(grid):
    """
    Calculates the minimum time required to rot all oranges.
    Args:
        grid: List[List[int]] representing the warehouse.
    Returns:
        int: Minutes elapsed, or -1 if impossible.
    """
    rows, cols = len(grid), len(grid[0])
    queue = deque()
    fresh_count = 0

    # --- Phase 1: Scan Grid ---
    # We need to find all sources of the 'infection' (rotten oranges)
    # and count how many targets (fresh oranges) exist.
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
```

```python
                # Add initial rotten oranges to queue as (row, col, time)
                # or just (row, col) and track time via layers.
                # Here, tracking layers manually is cleaner for the return value.
                queue.append((r, c))
            elif grid[r][c] == 1:
                fresh_count += 1

    # Optimization: If no fresh oranges, 0 minutes needed.
    if fresh_count == 0:
        return 0

    minutes_passed = 0
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)] # Right, Left, Down, Up

    # --- Phase 2: Multi-Source BFS ---
    while queue and fresh_count > 0:
        # We process the queue level-by-level (minute-by-minute)
        # We start a loop for the CURRENT size of the queue.
        # This ensures we process all 'Minute N' oranges before moving to 'Minute N+1'

        for _ in range(len(queue)):
            r, c = queue.popleft()

            for dr, dc in directions:
                nr, nc = r + dr, c + dc

                # Check bounds and if the neighbor is a fresh orange
                if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 1:
                    # Make it rotten
                    grid[nr][nc] = 2
                    # Add to queue to infect others next minute
                    queue.append((nr, nc))
                    fresh_count -= 1

        # Only increment time if we actually processed a level
        minutes_passed += 1

    # --- Phase 3: Final Check ---
    # If fresh oranges remain, it means they are isolated (unreachable).
    return minutes_passed if fresh_count == 0 else -1
```

JavaScript Solution   *Note on Array.shift(): In JS, shift() is O(K) where
K is array length. In a strictly academic setting or very high-performance
constraints, one might use a pointer or a Linked List. However, for most
L5/L6 interviews, using an Array as a queue is accepted unless the interviewer*

specifically asks to optimize the Queue implementation.

```javascript
/**
 * @param {number[][]} grid
 * @return {number}
 */
var orangesRotting = function(grid) {
    if (!grid || grid.length === 0) return 0;

    const rows = grid.length;
    const cols = grid[0].length;
    const queue = [];
    let freshCount = 0;

    // --- Phase 1: Initialization ---
    // Identify all starting points (Rotten) and target count (Fresh)
    for (let r = 0; r < rows; r++) {
        for (let c = 0; c < cols; c++) {
            if (grid[r][c] === 2) {
                queue.push([r, c]);
            } else if (grid[r][c] === 1) {
                freshCount++;
            }
        }
    }

    // Edge case: No fresh oranges to begin with
    if (freshCount === 0) return 0;

    let minutes = 0;
    const directions = [[0, 1], [0, -1], [1, 0], [-1, 0]];

    // --- Phase 2: BFS ---
    // While there are rotten oranges spreading and fresh targets remaining
    while (queue.length > 0 && freshCount > 0) {

        // Snapshot the current queue length to process only the current 'minute'
        const size = queue.length;

        for (let i = 0; i < size; i++) {
            const [r, c] = queue.shift(); // Remove from front

            for (const [dr, dc] of directions) {
                const nr = r + dr;
                const nc = c + dc;
```

97

```
                // Check bounds and if neighbor is fresh
                if (nr >= 0 && nr < rows &&
                    nc >= 0 && nc < cols &&
                    grid[nr][nc] === 1) {

                    // Mutate grid to mark as rotten (avoiding a visited set)
                    grid[nr][nc] = 2;
                    freshCount--;
                    queue.push([nr, nc]);
                }
            }
        }
        // One layer of propagation complete
        minutes++;
    }

    // --- Phase 3: Validation ---
    return freshCount === 0 ? minutes : -1;
};
```

---

5. Terminology & Techniques

Technique: Multi-Source BFS Standard BFS starts from one node and
ripples out. Multi-Source BFS initializes the Queue with **all** starting nodes
simultaneously.

- **Why it helps:** It calculates the shortest path from *any* source node to
  every other node in the graph. In the context of the oranges, it correctly
  simulates the rot spreading from multiple locations at the same time.

Technique: In-Place Modification Notice we didn't use a separate `visited`
set (like `visited = set()`).

- **Why it helps:** We saved space by modifying the input `grid` directly.
  When we turn a `1` (Fresh) into a `2` (Rotten), that cell effectively becomes
  "visited." In an interview, ask if mutating the input is allowed. If not, use
  a `visited` set.

---

6. Real World & Interview Variations

Google, Meta, and Bloomberg love this problem because it tests graph traversal
without explicitly looking like a graph. Here is how they disguise it:

A. The "Virus Propagation" (Google / Meta) Question: You have
a computer network (represented as a grid or adjacency list). Some comput-

ers are infected with malware. The malware spreads to connected, unpatched computers every second. How long until the whole network is down?

- **Solution:** Identical to Rotting Oranges.
- Rotten Orange = Infected Computer.
- Fresh Orange = Unpatched Computer.
- Empty Cell = Firewall/Air-gapped connection.

B. The "Fire Spreading" with Walls (Amazon)   Question: A forest has trees (1) and rocks (0). A fire starts at multiple points (2). Return the time it takes to burn all trees.

- **Twist:** Sometimes they add a "User" who must escape the fire.
- **Solution:** This requires **Two BFS runs**.

1. Run Multi-Source BFS for the fire to generate a `FireTime[][]` grid (time the fire reaches every cell).
2. Run a standard BFS for the "User" to see if they can reach the exit *before* the `FireTime` for that cell.

C. The "Server Updates" (Bloomberg / System Design)   Question: You need to push a software update to a cluster of servers. You have 3 master servers that have the update. In one "tick," a server can send the update to its direct neighbors. How many ticks to update the cluster?

- **Solution:** Multi-Source BFS starting from the 3 master servers.
- **L6 Insight:** In a real distributed system, this is "Gossip Protocol" or "Epidemic Broadcasting." The BFS solution calculates the theoretical minimum latency for full propagation.

127. Word Ladder

Here is a comprehensive breakdown of **LeetCode 127: Word Ladder**, structured as requested, with the depth and clarity expected of a senior engineering discussion.

---

1. Problem Explanation

The Goal: We are given two words, a `beginWord` and an `endWord`, and a dictionary list of distinct `wordList`. We need to find the number of words in the **shortest transformation sequence** from `beginWord` to `endWord`.

The Rules:

1. We can only change **one letter** at a time.
2. Every transformed word must exist in the `wordList`.

3. The `beginWord` does not need to be in the `wordList`, but the `endWord` must be.
4. If there is no such sequence, return 0.

Example:

- **Begin:** "hit"
- **End:** "cog"
- **Dictionary:** ["hot", "dot", "dog", "lot", "log", "cog"]

The Visual Flow: We need to get from `hit` to `cog` by changing one character at a time, ensuring every intermediate step is a valid word.

```
Step 0:  "hit"  (Start)
           |
           v     (Change 'i' -> 'o')
Step 1:  "hot"  (Valid? Yes, in list)
           |
           v     (Change 'h' -> 'd')
Step 2:  "dot"  (Valid? Yes, in list)
           |
           v     (Change 't' -> 'g')
Step 3:  "dog"  (Valid? Yes, in list)
           |
           v     (Change 'd' -> 'c')
Step 4:  "cog"  (End reached!)

Total Length of Sequence: 5 words ("hit", "hot", "dot", "dog", "cog")
Result: 5
```

---

2. Solution Explanation

The Intuition (L5/L6 Perspective): When a problem asks for the "shortest path" or "minimum steps" to get from State A to State B, and the steps are unweighted (every step costs 1 unit of effort), this is a classic **Breadth-First Search (BFS)** problem.

Think of this not as a string manipulation problem, but as a **Graph Problem**.

- **Nodes:** The words.
- **Edges:** Two nodes are connected if they differ by exactly one letter.

We want the shortest path in this graph. DFS (Depth First Search) is bad here because it might go down a "rabbit hole" of 100 transformations before finding a solution that was only 3 steps away. BFS spreads out like a ripple in a pond, guaranteeing we find the target in the fewest layers.

The Strategy: Level-by-Level Expansion

1. Start with `beginWord` in a Queue.
2. Mark it as visited (so we don't loop back to it).
3. For every word in the Queue, try to find all its "neighbors" (words that differ by 1 char).
4. If a neighbor is the `endWord`, we are done!
5. If not, add valid neighbors to the Queue for the next level.
6. Repeat until the Queue is empty.

The Non-Trivial Part: Efficiently Finding Neighbors The naive way to find neighbors is to compare the current word with *every* word in the dictionary.

- If dictionary size is and word length is .
- Comparing two words takes time.
- Doing this for all words takes .
- Total complexity explodes to . If is 5000, is 25,000,000. Too slow.
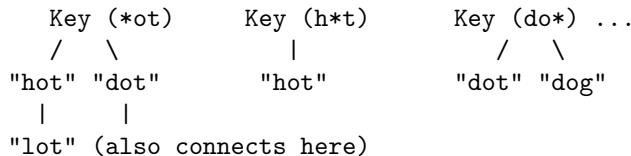
The "Wildcard" Optimization (Pre-processing): Instead of comparing words, we can generate generic forms using a wildcard character `*`.

- Word: "hot"
- Generic forms: `*ot`, `h*t`, `ho*`

Any word that matches `*ot` (like "dot", "lot", "pot") is a neighbor of "hot". We pre-process the dictionary into a hash map where the key is the generic form and the value is a list of matching words.

Visualizing the Wildcard Map:

```
Dictionary: ["hot", "dot", "dog", "lot", "log", "cog"]


   Key (*ot)      Key (h*t)      Key (do*) ...
    /   \            |             /   \
"hot" "dot"        "hot"        "dot" "dog"
  |     |
"lot" (also connects here)
```

Step-by-Step Visualization of the BFS Process:

Let's trace `hit -> cog`.

Initialization:

- Queue: `[ ("hit", 1) ]` (Word, Level)
- Visited: `{"hit"}`

Round 1 (Level 1): Pop `hit`. Generate forms: `*it`, `h*t`, `hi*`.

- Check map for `*it`: No matches in dict.

- Check map for `h*t`: Matches `hot`.

- Is `hot` visited? No.

- Push `hot` to Queue. Mark visited.

- Check map for `hi*`: No matches.

```
[ hit ]   <-- Level 1
   |
   v
[ hot ]   <-- Level 2
```

Round 2 (Level 2): Pop `hot`. Generate forms: `*ot`, `h*t`, `ho*`.

- Check `*ot`: Map contains [`dot, lot`].

- Push `dot`. Mark visited.

- Push `lot`. Mark visited.

- Check `ho*`: Map contains [`hot`]. (Already visited `hot`, ignore).

```
   [ hit ]
      |
   [ hot ]
   /      \
[ dot ] [ lot ]   <-- Level 3
```

Round 3 (Level 3): Pop `dot`. Generate forms: `d*t`, `do*`, `*ot`.

- Check `do*`: Map contains [`dog`].

- Push `dog`. Mark visited.

- Check `*ot`: Matches `hot`, `lot` (Visited).

Pop `lot`. Generate forms: `l*t`, `lo*`, `*ot`.

- Check `lo*`: Map contains [`log`].
- Push `log`. Mark visited.

```
      [ hit ]
         |
      [ hot ]
      /      \
  [ dot ] [ lot ]
    |         |
  [ dog ] [ log ]   <-- Level 4
```

**Round 4 (Level 4):** Pop `dog`. Generate neighbors... matches `cog`.
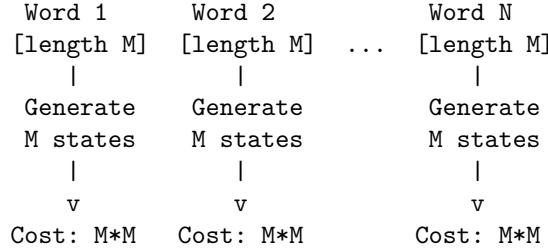
- **TARGET FOUND!** Return Current Level + 1.

---

3. Time and Space Complexity Analysis

Time Complexity Derivation
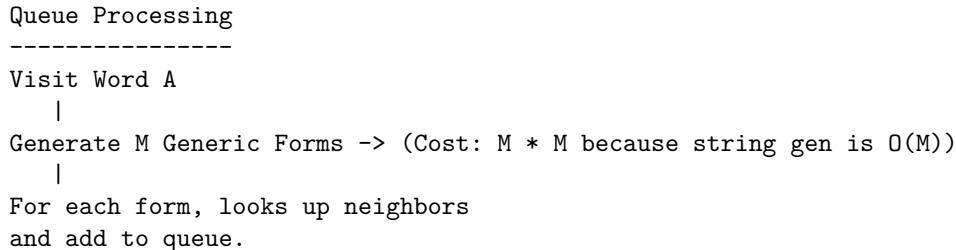
Let be the length of a word. Let be the number of words in the dictionary.

1. Pre-processing Phase (Building the Wildcard Map): We iterate through every word in the list (words). For each word, we generate intermediate states (by replacing each char with *). Each string generation takes time (copying the string).

```
   Word 1       Word 2            Word N
 [length M]   [length M]   ...  [length M]
     |            |                 |
  Generate     Generate         Generate
  M states     M states         M states
     |            |                 |
     v            v                 v
 Cost: M*M    Cost: M*M        Cost: M*M
```

```
Total Pre-processing: O(M^2 * N)
```

2. BFS Search Phase: In the worst case, we visit every word (). For each word we visit, we again generate all its generic transformations (transformations). For each transformation, looking up in the hash map takes (string hashing/comparison).
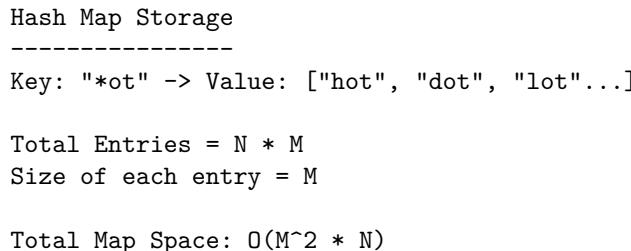
```
Queue Processing
----------------
Visit Word A
   |
Generate M Generic Forms -> (Cost: M * M because string gen is O(M))
   |
For each form, looks up neighbors
and add to queue.

Total Search: O(M^2 * N)
```

Total Time Complexity:

Space Complexity Derivation

We need to store the Wildcard Map and the Queue.

1. The Map: In the worst case, every word has variations. We store words × variations. Each key is length .

```
Hash Map Storage
----------------
Key: "*ot" -> Value: ["hot", "dot", "lot"...]

Total Entries = N * M
Size of each entry = M

Total Map Space: O(M^2 * N)
```

2. The Queue/Visited Set: Stores at most words. Each word is length .
Space: .

Total Space Complexity:

---

4. Solution Code

Python Solution (Optimized BFS)

```python
from collections import deque, defaultdict

def ladderLength(beginWord, endWord, wordList):
    """
    Calculates the shortest transformation sequence length from beginWord to endWord.
    """

    # 1. Quick check: If endWord isn't in the list, we can never reach it.
    if endWord not in wordList:
        return 0

    # 2. Pre-processing: Build the Adjacency List using Wildcards
    # This allows O(1) lookup of neighbors instead of iterating the whole list.
    # Key: Generic word (e.g., "*ot"), Value: List of matching words ["hot", "dot"]
    all_combo_dict = defaultdict(list)
    L = len(beginWord)

    for word in wordList:
        for i in range(L):
            # Create generic form: "hot" -> "*ot", "h*t", "ho*"
            generic_word = word[:i] + "*" + word[i+1:]
            all_combo_dict[generic_word].append(word)

    # 3. Initialize BFS
    # Queue stores tuples: (current_word, current_level)
    queue = deque([(beginWord, 1)])

    # Visited set to prevent cycles (e.g., hot -> dot -> hot)
    visited = set([beginWord])

    # 4. Process the Queue
    while queue:
        current_word, level = queue.popleft()

        # Check all possible generic forms of the current word
        for i in range(L):
```

```python
                intermediate_word = current_word[:i] + "*" + current_word[i+1:]

                # If this generic form exists in our map, we have neighbors
                for neighbor in all_combo_dict[intermediate_word]:

                    # If we found the target, we are done!
                    if neighbor == endWord:
                        return level + 1

                    # If not visited, add to queue to process at the next level
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, level + 1))

                # Optimization: Clear the list for this generic key to prevent
                # reprocessing if we encounter this state again.
                # (Though visited set handles this, this saves loop iterations).
                all_combo_dict[intermediate_word] = []

    # If queue is empty and we never reached endWord
    return 0
```

JavaScript Solution

```javascript
/**
 * @param {string} beginWord
 * @param {string} endWord
 * @param {string[]} wordList
 * @return {number}
 */
var ladderLength = function(beginWord, endWord, wordList) {
    // 1. Edge case: endWord must be in the dictionary
    if (!wordList.includes(endWord)) return 0;

    const L = beginWord.length;

    // 2. Pre-processing: Build the Wildcard Map
    // Map structure: { "*ot": ["hot", "dot"], "h*t": ["hot"] ... }
    const allComboDict = new Map();

    wordList.forEach(word => {
        for (let i = 0; i < L; i++) {
            const genericWord = word.substring(0, i) + '*' + word.substring(i + 1);
            if (!allComboDict.has(genericWord)) {
                allComboDict.set(genericWord, []);
            }
```

```javascript
                allComboDict.get(genericWord).push(word);
        }
    });

    // 3. Initialize BFS
    // Queue stores [word, level]
    const queue = [[beginWord, 1]];
    const visited = new Set();
    visited.add(beginWord);

    // 4. Start BFS Loop
    while (queue.length > 0) {
        // Shift is O(n) in JS array, ideally use a real Queue implementation for strict O(
        // For interview purposes, array.shift() is usually acceptable unless constraints a
        const [currentWord, level] = queue.shift();

        for (let i = 0; i < L; i++) {
            const intermediateWord = currentWord.substring(0, i) + '*' + currentWord.substri

            // Do we have neighbors for this generic form?
            if (allComboDict.has(intermediateWord)) {
                const neighbors = allComboDict.get(intermediateWord);

                for (let neighbor of neighbors) {
                    if (neighbor === endWord) {
                        return level + 1;
                    }

                    if (!visited.has(neighbor)) {
                        visited.add(neighbor);
                        queue.push([neighbor, level + 1]);
                    }
                }
                // Optimization: Remove this generic key to prevent future redundant checks
                allComboDict.delete(intermediateWord);
            }
        }
    }

    return 0;
};
```

Note 1: Terminology & Techniques

Breadth-First Search (BFS): This is the core algorithm used here. BFS explores a graph layer by layer (neighbors first, then neighbors of neighbors). It is the gold standard for finding the **shortest path** in an **unweighted graph** (where moving from node A to node B always costs "1").

If we used **Depth-First Search (DFS)**, the algorithm might traverse a very long path (e.g., `hit -> hot -> lot -> log -> ...` for 100 steps) before realizing `cog` was just a few steps away via a different route. DFS does not guarantee the shortest path on the first hit.

Bidirectional BFS (The L6 Optimization): While the solution above is , extremely large graphs can still be slow. An L6 engineer would mention **Bidirectional BFS**.

- **Concept:** Run two BFS searches simultaneously. One starting from `beginWord` forward, and one starting from `endWord` backward.
- **Why?** The search space grows exponentially. By meeting in the middle, you significantly reduce the total nodes visited.
- **Visual:** Instead of a large cone expanding from Start to End, you have two smaller cones meeting in the center.

Note 2: Real World & Indirect Interview Variations

Companies like Google, Meta, and Bloomberg rarely ask "Solve Word Ladder" directly anymore. They wrap it in different narratives.

1. The "Genetic Mutation" Problem (Google/Twitter)

- **Question:** A gene string consists of 'A', 'C', 'G', 'T'. You want to mutate StartGene to EndGene. You can change 1 character at a time, but the new gene must be in a list of "Valid Genes".
- **Solution:** This is literally Word Ladder. The "Dictionary" is the list of valid genes. The character set is restricted to 4 chars instead of 26.
- *Implementation:* Use the exact same BFS approach.

2. The "Open the Lock" Problem (Google/Meta)

- **Question:** You have a lock with 4 wheels (0-9). You start at "0000". You want to reach a target "0202". You can turn any wheel up or down by 1 digit. There is a list of "Deadends" (combinations that jam the lock). Find the minimum turns.
- **Solution:**
- **Nodes:** The 4-digit combinations (e.g., "0000").
- **Edges:** "0000" connects to "1000", "9000", "0100", etc.
- **Deadends:** These are visited nodes you mark as "blocked" before you even start.

- *Implementation:* BFS. Instead of iterating `a-z`, you iterate `+1` and `-1` for each digit position.

3. The "Twitter/Social Network Connection" (Bloomberg/LinkedIn)

- **Question:** Given User A and User B, find the minimum number of connections (hops) to introduce them.
- **Solution:**
- **Nodes:** Users.
- **Edges:** Friendships.
- *Implementation:* This is the purest form of BFS. You don't generate strings; you just iterate over the user's friend list. Bidirectional BFS is almost mandatory here due to the massive scale of social graphs.

210. Course Schedule II

Here is a breakdown of how a Senior (L5) or Staff (L6) Engineer would deconstruct and solve **LeetCode 210: Course Schedule II**.

At this level, the focus shifts from "getting it to work" to "modeling the problem correctly" and "writing maintainable, production-grade logic."

1. Problem Explanation

The Core Concept: Dependency Resolution Imagine you are a university student trying to graduate. You have a list of courses you want to take, but the university has strict rules: to take Course B, you must *first* finish Course A.
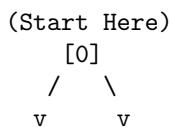
This is a classic **Dependency Graph** problem.
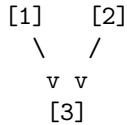
- **Nodes (Vertices):** The Courses (e.g., Data Structures, Algorithms).
- **Edges (Arrows):** The Prerequisite rules (e.g., Intro to CS –> Data Structures).

The Goal: Find a linear ordering (a list) where every course appears *after* its prerequisites.

The "Impossible" Scenario (The Cycle): If Course A requires Course B, and Course B requires Course A, you are stuck. This is a "Cyclic Dependency." You can never start. In this case, we must return an empty list.
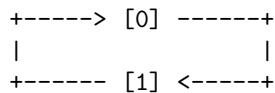
ASCII Visualization: A Happy Path Input: `numCourses = 4`, `prerequisites = [[1,0], [2,0], [3,1], [3,2]]` *Meaning: To take 1, you need 0. To take 2, you need 0. To take 3, you need both 1 and 2.*

```
(Start Here)
   [0]
  /   \
 v     v
```

```
    [1]    [2]
      \    /
       v  v
       [3]
```

Valid Order: [0, 1, 2, 3] OR [0, 2, 1, 3] *Notice how the arrows always point "forward" in the list.*

ASCII Visualization: The "Deadlock" (Cycle) Input: [[1,0], [0,1]]

```
    +-----> [0] ------+
    |                 |
    +------ [1] <-----+
```

You cannot pull either node "out" because both are being held back by the other. This is a deadlock.

---

2. Solution Explanation

To solve this as an engineer, we use an algorithm called **Topological Sort**. Specifically, **Kahn's Algorithm** (Breadth-First Search approach) is often preferred in production systems because it is iterative (avoids stack overflow on deep graphs) and intuitively models "resolving dependencies."

The Strategy: "Peeling the Onion"

1. **Calculate Indegrees:** Count how many prerequisites each course has.
2. **Find Starters:** Identify courses with 0 prerequisites (Indegree = 0). These are free to take immediately.
3. **Process Queue:**

- Add "free" courses to your `Order` list.
- "Complete" the course: Go to its neighbors (dependent courses) and "remove" the prerequisite (decrement their Indegree).
- If a neighbor's Indegree drops to 0, it becomes free. Add it to the queue.

4. **Cycle Check:** If the final `Order` list contains fewer courses than the total, we hit a cycle (some courses never became free).

Detailed Walkthrough with ASCII Input: `numCourses = 4`, `prerequisites = [[1,0], [2,0], [3,1], [3,2]]`

Step A: Build Graph & Indegree Table

```
Adjacency List (Who depends on me?):
[0] -> [1, 2]  (If I finish 0, I unlock 1 and 2)
[1] -> [3]
[2] -> [3]
```

```
[3] -> []
```

```
Indegree Table (How many do I need?):
Course | Prerequisites Needed
-------|---------------------
  0    | 0  <-- READY!
  1    | 1  (Needs 0)
  2    | 1  (Needs 0)
  3    | 2  (Needs 1 & 2)
```

Step B: Initialize Queue We add [0] to our Queue because it has 0 dependencies.

Step C: Processing the Queue

Iteration 1:

- Pop 0.
- Add 0 to Result: [0]
- Look at neighbors of 0: 1 and 2.
- Decrement their indegrees.

```
State update:
Course | Prerequisites Needed
-------|---------------------
  0    | Done
  1    | 0  <-- BECAME READY! (Was 1)
  2    | 0  <-- BECAME READY! (Was 1)
  3    | 2
```

```
Queue: [1, 2]
```

Iteration 2:

- Pop 1.
- Add 1 to Result: [0, 1]
- Look at neighbors of 1: 3.
- Decrement 3's indegree.

```
State update:
Course | Prerequisites Needed
-------|---------------------
  3    | 1  (Was 2. Still waiting on Course 2)
```

```
Queue: [2]
```

Iteration 3:

- Pop 2.
- Add 2 to Result: [0, 1, 2]
- Look at neighbors of 2: 3.

- Decrement 3's indegree.

```
State update:
Course | Prerequisites Needed
-------|--------------------
  3    | 0  <-- BECAME READY! (Was 1)
```

```
Queue: [3]
```

Iteration 4:

- Pop 3.
- Add 3 to Result: `[0, 1, 2, 3]`
- Neighbors: None.

Step D: Final Check Result size (4) == numCourses (4). **Success.**

---

3. Time and Space Complexity Analysis

Time Complexity: O(V + E)

- **V** = Number of Courses (Vertices)
- **E** = Number of Prerequisites (Edges)

```
VISUAL DERIVATION:


1. Building Graph:
   We iterate through the prerequisites list once.
   [Link] -> [Link] -> [Link] ...
   Cost: O(E)

2. Building Indegree:
   Done simultaneously with building the graph.
   Cost: O(E)

3. Processing Queue (The loop):
   How many times does a node enter the Queue?
   Only ONCE (when indegree hits 0).
   Total Node Operations: O(V)

   How many times do we check neighbors?
   We traverse every arrow (Edge) exactly ONCE when the parent is processed.

       [A] --1--> [B]
           --2--> [C]
           --3--> [D]
```

```
    Total Edge Operations: O(E)

TOTAL: O(V + E) + O(V + E) = O(V + E)
```

Space Complexity: $O(V + E)$

```
VISUAL DERIVATION:

1. Adjacency List (Graph):
   Stores every Node (V) and every Arrow (E).

   [0]: [1, 2]
   [1]: [3]
   ...
   Cost: O(V + E)

2. Indegree Array:
   An integer for every course.
   [ 0, 1, 1, 2 ]
   Cost: O(V)

3. Queue:
   In the worst case (e.g., no dependencies), all nodes are in queue.
   Cost: O(V)

TOTAL: O(V + E)
```

---

4. Solution Code

Python (Production Grade) *Uses `collections.deque` for O(1) pops, and clear variable naming.*

```python
from collections import deque, defaultdict
from typing import List

class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        # 1. Initialize the Graph and Indegree Table
        # adj_list: Mapping from 'prerequisite' to 'course that needs it'
        adj_list = defaultdict(list)
        # indegree: Array tracking how many prerequisites each course holds
        indegree = [0] * numCourses

        # 2. Build the Graph
        # pair is [course_to_take, prerequisite]
```

```python
    for dest, src in prerequisites:
        adj_list[src].append(dest)
        indegree[dest] += 1

    # 3. Find all courses with NO prerequisites to start
    # These are the entry points to our graph
    queue = deque([k for k in range(numCourses) if indegree[k] == 0])

    # This list will store our valid topological sort
    topological_order = []

    # 4. Process the Graph (BFS - Kahn's Algorithm)
    while queue:
        current_course = queue.popleft()
        topological_order.append(current_course)

        # Look at all courses that depended on the current_course
        if current_course in adj_list:
            for neighbor in adj_list[current_course]:
                # We 'completed' current_course, so neighbor needs 1 less prereq
                indegree[neighbor] -= 1

                # If neighbor has 0 prereqs left, it is ready to be taken
                if indegree[neighbor] == 0:
                    queue.append(neighbor)

    # 5. Cycle Detection Check
    # If the order list doesn't contain all courses, it implies a cycle existed
    # (some courses never reached indegree 0)
    if len(topological_order) == numCourses:
        return topological_order
    else:
        return []
```

JavaScript (ES6+ Clean)

```javascript
/**
 * @param {number} numCourses
 * @param {number[][]} prerequisites
 * @return {number[]}
 */
var findOrder = function(numCourses, prerequisites) {
    // 1. Initialize data structures
    const adjList = new Map();
    const indegree = new Array(numCourses).fill(0);
```

```javascript
    // 2. Build Graph
    for (const [course, prereq] of prerequisites) {
        // If list doesn't exist for prereq, create it
        if (!adjList.has(prereq)) {
            adjList.set(prereq, []);
        }
        adjList.get(prereq).push(course);

        // Increment indegree for the course being blocked
        indegree[course]++;
    }

    // 3. Initialize Queue with courses having 0 prerequisites
    const queue = [];
    for (let i = 0; i < numCourses; i++) {
        if (indegree[i] === 0) {
            queue.push(i);
        }
    }

    const result = [];

    // 4. Process Queue (BFS)
    // Note: JS arrays as queues are O(n) for shift(), but for interview
    // contexts it is usually acceptable unless strict O(1) is demanded.
    // For strict O(1), use a Pointer approach or LinkedList.
    while (queue.length > 0) {
        const current = queue.shift();
        result.push(current);

        if (adjList.has(current)) {
            for (const neighbor of adjList.get(current)) {
                indegree[neighbor]--;
                if (indegree[neighbor] === 0) {
                    queue.push(neighbor);
                }
            }
        }
    }

    // 5. Cycle Check
    return result.length === numCourses ? result : [];
};
```

Note 1: Terminology

- **Topological Sort:** A linear ordering of vertices in a graph such that for every directed edge from u to v, u comes before v. This is only possible in DAGs.
- **DAG (Directed Acyclic Graph):** A graph with directions (arrows) that has no loops. If you follow the arrows, you never return to where you started.
- **Indegree:** The number of arrows pointing *into* a node. In this problem, it represents the number of prerequisites remaining.

Note 2: Real World & Interview Variations

Big Tech companies rarely ask "Implement Topological Sort" directly. They mask it in practical scenarios. Here is how they ask it and how to adapt the solution:

1. The "Build System" (Google / Meta)   Question: "Design a build system like Make, Maven, or Webpack. Given a list of files where file A imports file B, determine the order to compile them. Can we compile files in parallel?"

Adaptation:

- **Core:** Same Topological Sort.
- **Parallelism (Level 2):** Instead of processing one item from the Queue at a time, you can process *all items currently in the Queue* simultaneously (using threads). Since everything in the Queue has `Indegree == 0` at that moment, they are independent of each other!
- **Visual:**

```
Queue: [A, B]  <-- Launch Thread 1 for A, Thread 2 for B
Wait for both.
Update neighbors.
Queue: [C, D, E] <-- Launch 3 Threads...
```

2. The "Bad Package Version" (Bloomberg / npm)   Question: "We have a package manager. Package A depends on Package B. Sometimes circular dependencies happen (A->B->C->A). Write a function that detects which packages are part of the cycle."

Adaptation:

- **Core:** Use the same algorithm.
- **Result:** If `result.length != total`, the items *not* in the result list are the ones stuck in the cycle (or stuck waiting on the cycle).

- **L6 Detail:** You might need to use DFS (Depth First Search) with "color-ing" (White=Unvisited, Gray=Visiting, Black=Visited) to explicitly print the exact path of the cycle (e.g., "Error: A->B->C->A").

3. Task Scheduling with Aliasing (Uber / DoorDash)   Question: "You have tasks to complete. Some tasks are actually the same but named differently (aliasing). Determine order."

Adaptation:

- **Preprocessing:** Before building the graph, you must use a **Union-Find** data structure to merge aliased tasks into single nodes.
- **Then:** Run Topological Sort on the merged nodes.

329. Longest Increasing Path in a Matrix

Here is how a Senior Staff Software Engineer (L6) at Google would break down, solve, and optimize **LeetCode 329: Longest Increasing Path in a Matrix**.

---

1. Problem Explanation

The goal is to find the length of the longest path in a grid where every step you take moves to a strictly larger number. You can move Up, Down, Left, or Right. You cannot move diagonally or wrap around the edges.

Let's visualize the constraints with a simple ASCII example.

Example Grid (3x3):

```
    Col 0   Col 1   Col 2
  +-----+-----+-----+
0 |  9  |  9  |  4  |
  +-----+-----+-----+
1 |  6  |  6  |  8  |
  +-----+-----+-----+
2 |  2  |  1  |  1  |
  +-----+-----+-----+
```

Valid Moves vs. Invalid Moves: Imagine you are standing at (`Row 1, Col 0`) which has the value **6**.

- **Up:** 9 ($9 > 6$) -> **Valid**
- **Down:** 2 ($2 < 6$) -> **Invalid** (Must increase)
- **Right:** 6 ($6 == 6$) -> **Invalid** (Must strictly increase, not equal)

Visualizing a Path: One potential path starting at (`2, 1`) (Value: 1):

```
(2,1)[1] --> (2,0)[2] --> (1,0)[6] --> (0,0)[9]
```

```
Length: 4 steps
```

The "Answer" is the maximum length found among *all* possible paths starting from *any* cell.

---

2. Solution Explanation

An L5/L6 engineer recognizes this immediately as a graph traversal problem. Each cell is a node, and a directed edge exists from cell A to cell B if `Value(B) > Value(A)` and they are neighbors.

Approach 1: Naive DFS (Do not implement, but understand) If we run a pure Depth First Search (DFS) from every cell, we will do redundant work.

Scenario:

1. We calculate the longest path starting from `Cell A`. Let's say the path goes `A -> B -> C`.
2. Later, the loop asks us to calculate the longest path starting from `Cell B`.
3. We re-calculate `B -> C`.

This re-calculation is the bottleneck.

Approach 2: DFS + Memoization (The L5/L6 Standard) We use **Memoization** (caching). If we have already computed the longest path starting from `Cell B`, we store it. If we ever reach `Cell B` again from a neighbor, we just look up the answer instead of re-walking the path.

Step-by-Step Visualization of the Algorithm:

Step A: Iterate through every cell in the grid. **Step B:** Launch a DFS from that cell. **Step C:** Check the "Cache" (Memoization table).

- If value exists: Return it immediately.
- If empty: Explore all 4 neighbors. **Step D:** The length for the current cell = `1 + max(all valid neighbor lengths)`. **Step E:** Store this result in the Cache and return.

Visual Trace: Let's trace `(1, 0)` which is **6**.

```
Processing Cell (1,0) [Val: 6]
   |
   +--> Try UP (0,0) [Val: 9]:
   |     |
   |     +--> Recurse DFS((0,0))...
```

```
    |            |
    |            +--> (0,0) has no neighbors > 9.
    |            +--> Return length 1.
    |
    +--> Try RIGHT (1,1) [Val: 6]:
    |     |
    |     +--> 6 is not > 6. SKIP.
    |
    +--> Try DOWN (2,0) [Val: 2]:
          |
          +--> 2 is not > 6. SKIP.
```

```
Result for (1,0) = 1 (itself) + 1 (from UP neighbor) = 2.
Write "2" into Cache at position [1][0].
```

The "Aha!" Moment: Since the path must be *strictly increasing*, there are **no cycles**. You can never go 1 -> 2 -> 3 -> 1. This means we don't need a "visited" set to prevent infinite loops within a single recursion stack. The values themselves prevent loops.

----
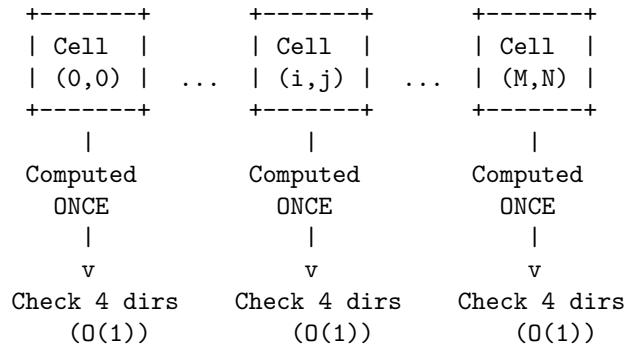
3. Time and Space Complexity Analysis

Time Complexity Derivation:

Without memoization, the branching factor is roughly 4 (4 directions). The tree height could be `M * N`. That is exponential `O(4^(M*N))`. This is why we need memoization.

With Memoization, we solve the sub-problem for each cell exactly **ONCE**.

```
Total Cells = M (rows) * N (cols)
```

```
   +-------+       +-------+       +-------+
   | Cell  |       | Cell  |       | Cell  |
   | (0,0) |  ...  | (i,j) |  ...  | (M,N) |
   +-------+       +-------+       +-------+
       |               |               |
    Computed        Computed        Computed
     ONCE            ONCE            ONCE
       |               |               |
       v               v               v
  Check 4 dirs    Check 4 dirs    Check 4 dirs
     (O(1))          (O(1))          (O(1))


Total Work = (Number of Cells) * (Work per Cell)
          = (M * N) * (Constant time checks)
```

```
        = O(M * N)
```

Time Complexity: `O(M * N)`

Space Complexity Derivation:

We need to store the computed results for every cell.

```
1. Memoization Table (Cache):
   +---+---+---+
   | 2 | 1 | 5 |   <-- Stores integer for each cell
   +---+---+---+
   |...|...|...|

   Size: M * N integers -> O(M * N) space

2. Recursion Stack (DFS Depth):
   In the worst case (a snake-like increasing path covering the whole grid):

   1 -> 2 -> 3 -> 4
                  |
   8 <- 7 <- 6 <- 5
   |
   9 ...

   The stack depth can go up to M * N frames.
   Size: O(M * N)
```

Total Space Complexity: `O(M * N)`

---

4. Solution Code

Python Solution   This implementation focuses on readability and caching, typical of production-grade code.

```python
class Solution:
    def longestIncreasingPath(self, matrix: list[list[int]]) -> int:
        if not matrix or not matrix[0]:
            return 0

        rows, cols = len(matrix), len(matrix[0])
        # Memoization table initialized with 0.
        # 0 indicates "not computed yet".
        memo = [[0] * cols for _ in range(rows)]

        # Directions: Down, Up, Right, Left
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
```

```python
        def dfs(r, c):
            # 1. Check Cache
            if memo[r][c] != 0:
                return memo[r][c]

            # 2. Explore Neighbors
            max_neighbor_len = 0

            for dr, dc in directions:
                nr, nc = r + dr, c + dc

                # Check bounds AND strictly increasing condition
                if (0 <= nr < rows and
                    0 <= nc < cols and
                    matrix[nr][nc] > matrix[r][c]):

                    max_neighbor_len = max(max_neighbor_len, dfs(nr, nc))

            # 3. Store Result
            # Length is 1 (current cell) + longest path from neighbors
            memo[r][c] = 1 + max_neighbor_len
            return memo[r][c]

        # Driver Loop: We must try starting from EVERY cell
        # because the longest path could start anywhere.
        longest_path = 0
        for i in range(rows):
            for j in range(cols):
                longest_path = max(longest_path, dfs(i, j))

        return longest_path
```

Javascript Solution

```javascript
/**
 * @param {number[][]} matrix
 * @return {number}
 */
var longestIncreasingPath = function(matrix) {
    if (!matrix || matrix.length === 0) return 0;

    const rows = matrix.length;
    const cols = matrix[0].length;

    // Create a 2D array for memoization, filled with 0
```

```javascript
    const memo = Array.from({ length: rows }, () => Array(cols).fill(0));

    const directions = [[1, 0], [-1, 0], [0, 1], [0, -1]];

    // Helper function for Depth First Search
    const dfs = (r, c) => {
        // If we already computed this cell, return the stored value
        if (memo[r][c] !== 0) return memo[r][c];

        let maxLen = 0;

        for (let [dr, dc] of directions) {
            const nr = r + dr;
            const nc = c + dc;

            // Check boundaries
            if (nr >= 0 && nr < rows && nc >= 0 && nc < cols) {
                // Check Strictly Increasing constraint
                if (matrix[nr][nc] > matrix[r][c]) {
                    maxLen = Math.max(maxLen, dfs(nr, nc));
                }
            }
        }

        // Save result: 1 (for current cell) + max from neighbors
        memo[r][c] = 1 + maxLen;
        return memo[r][c];
    };

    let globalMax = 0;

    // Iterate over every cell to find the max path length
    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            globalMax = Math.max(globalMax, dfs(i, j));
        }
    }

    return globalMax;
};
```

Note 1: Terminology

- **DFS (Depth First Search):** An algorithm that explores as far as possible along each branch before backtracking. In this problem, it's akin to walking down a path until you hit a dead end (a cell with no larger neighbors).
- **Memoization:** An optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Note 2: Real World & Interview Variants (Google, Meta, Bloomberg)

In senior interviews, you are rarely asked the raw LeetCode question. It is usually wrapped in a practical scenario.

1. The "Skiing" Problem (Common at Google/Meta) Question: "Given a topographical map where each cell represents elevation, find the longest run a skier can take. Skiers can only move downhill." **L6 Solution:** This is the exact same problem but reversed. Instead of `Strictly Increasing`, looking for `Strictly Decreasing` (or just invert the comparison logic). The "Elevation Map" is the matrix.

2. The Dependency Build Order (Google Build Tools Team) Question: "We have a grid of microservices. A service at `(r,c)` depends on neighbors if the neighbor has a strictly higher 'priority ID'. Find the longest chain of dependencies to determine the critical path latency." **L6 Solution:**

- This frames the problem as a **DAG (Directed Acyclic Graph)**.
- The "longest path in a DAG" is a classic application of this algorithm.
- **Advanced Twist:** The interviewer might ask, "What if we can move diagonally?" (Update directions array). "What if two services have equal priority?" (If equality is allowed, cycles might appear, breaking the DAG assumption. You would then need a `visited` set for the current recursion stack to detect cycles).

3. Water Flow / Drainage (Bloomberg) Question: "Rain falls on a grid of varying heights. Water flows to neighbors with lower height. Calculate the longest distance a water droplet can travel." **L6 Solution:** Identical to the Skiing problem. **Optimization for Interview:** If the interviewer asks to find *all* cells that can flow into the Ocean (border of grid), that is a different problem (usually solved by starting DFS *from the ocean* inwards). But if they ask for the *longest flow path*, it's the `Longest Increasing Path` solution.

947. Most Stones Removed with Same Row or Column

An L6 (Staff Engineer) at Google approaches this not just as a coding exercise, but as a modeling problem. They would immediately pivot from "simulating the removal" (which feels like a greedy algorithm) to "identifying structure" (Graph Theory).

Here is how I would break this down in an interview or design doc.

---

1. Problem Explanation

The Core Conflict: We have a 2D plane with stones at specific integer coordinates (`row, col`).

- **Rule:** You can remove a stone **IF AND ONLY IF** it shares a row or a column with another stone that has *not* been removed yet.
- **Goal:** Remove the maximum number of stones.

The "Aha!" Moment (Visualization): Most candidates try to figure out *which* stone to remove first. That is a trap. The order doesn't matter; the connectivity matters.

Imagine the stones are nodes in a graph. If two stones share a row or column, draw a line (an edge) between them.

Scenario A: The Line

```
(0,0) ---- (0,1) ---- (0,2)
Stone A    Stone B    Stone C
```

- Stone A shares a row with B.
- Stone B shares a row with A and C.
- Stone C shares a row with B.

Removal Strategy:

1. We can remove A (it shares with B).
2. Now we have `B -- C`.
3. We can remove C (it shares with B).
4. Now we have `B`.
5. We cannot remove B (no neighbors left).

- **Total Removed:** 2
- **Total Stones:** 3
- **Stones Remaining:** 1

Scenario B: The Cluster

```
(0,0) ---- (0,2)
  |          |
(2,0) ---- (2,2)
```

- All 4 stones form a cycle.
- We can pick any stone to start, say (0,0). Remove it.
- The remaining 3 are still connected in a chain.
- We can keep removing "leaf" nodes until only 1 stone remains.

The General Law: If a group of stones is connected (directly or indirectly), we can remove exactly stones. The last stone acts as the "anchor" that held the component together.

Therefore, the problem is actually: **Count the number of "Islands" (Connected Components) of stones.**

---

2. Solution Explanation

We need to group these stones. The most efficient tool for grouping items dynamically is **Union-Find (Disjoint Set Union - DSU)**.

The L6 Optimization (Coordinate Mapping) A naive approach compares every stone against every other stone.
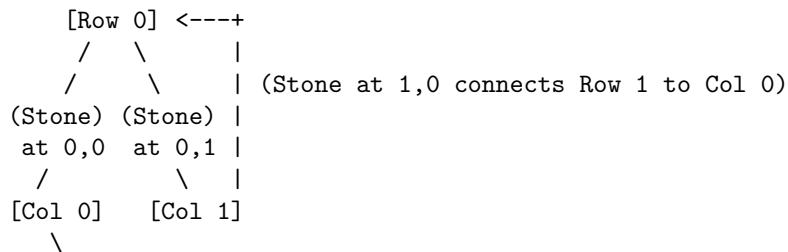
- Stone 1 vs Stone 2... Stone 1 vs Stone N.
- This is `O(N^2)`. With , that's 1,000,000 operations. Fast enough, but we can do better.

A Staff Engineer thinks: *What if N was 1,000,000?* The approach fails. We can solve this in `O(N)` effectively by treating Rows and Columns as the connector hubs.

The Graph Transformation: Instead of connecting Stone A to Stone B, let's connect **Row Index** to **Column Index**. Every stone at `(r, c)` is essentially an edge connecting the node `Row_r` and the node `Col_c`.

Visualizing the "Hub" Approach:

Stones: `(0, 0), (0, 1), (1, 0)`

```
   [Row 0] <---+
    /  \       |
   /    \      | (Stone at 1,0 connects Row 1 to Col 0)
(Stone) (Stone) |
 at 0,0  at 0,1 |
  /         \  |
[Col 0]   [Col 1]
   \
```

124

```
    \
    [Row 1]
```

1. Stone at `(0,0)` unions `Row 0` and `Col 0`.
2. Stone at `(0,1)` unions `Row 0` and `Col 1`. -> Now `Col 0`, `Row 0`, `Col 1` are all one group.
3. Stone at `(1,0)` unions `Row 1` and `Col 0`. -> Now `Row 1` joins the big group.

All stones belong to one giant connected component. Result = 3 Stones - 1 Component = 2 Removed.

Handling Indices: Rows and Cols both use integers `0, 1, 2....` To distinguish `Row 0` from `Col 0` in our data structure, we can offset columns.

- Constraint: Max coordinate is 10,000.
- Map `Row x -> x`
- Map `Col y -> y + 10001`

---

3. Time and Space Complexity Analysis

We are using Union-Find with Path Compression and Union by Rank.

Time Complexity:

```
Iterate through all N stones:
   |
   +---> For each stone (r, c):
           |
           +-> Union(r, c + offset)
                 |
                 +-> Find(r) -> Takes roughly O(1)*
                 +-> Find(c) -> Takes roughly O(1)*
                 +-> Link parents


Total Time: O(N * alpha(N))

*Note: alpha(N) is the Inverse Ackermann function.
For all practical universe sizes, alpha(N) < 5.
So, this is effectively O(N).
```

Space Complexity:

```
We need a Hash Map or Array to store the 'Parent' of each node.
   |
   +---> Worst case: Every stone is on a unique row/col.
   +---> Max Map size = 2 * N (If we use a Hash Map)
         OR
```

```
            Max Array size = ~20,000 (If we use fixed array based on coord max)

Total Space: O(N) or O(Max_Coordinate_Value)
```

---

4. Solution Code

I will provide the **Hash Map** based Union-Find. This is the preferred "Real World" solution because it handles sparse data (e.g., if a coordinate was 1,000,000, an array would be wasteful, but a map is efficient).

Python Solution

```python
class UnionFind:
    def __init__(self):
        # We use a dictionary to store parents.
        # Key: Node (Row or Col index), Value: Parent Node
        self.parent = {}
        # Count of distinct connected components (Islands)
        self.count = 0

    def find(self, x):
        # If x is not in parent, it's a new node we haven't seen.
        # Initialize it pointing to itself and increment island count.
        if x not in self.parent:
            self.parent[x] = x
            self.count += 1

        # Path Compression: Point x directly to the ultimate root
        if self.parent[x] != x:
            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 they are in different sets, merge them.
        if rootX != rootY:
            self.parent[rootX] = rootY
            # We merged two islands into one, so decrease the count
            self.count -= 1

class Solution:
    def removeStones(self, stones: list[list[int]]) -> int:
```

```python
    uf = UnionFind()

    # We need a way to distinguish Row 0 from Col 0.
    # Since coordinates are 0 <= x <= 10000, we offset columns.
    COL_OFFSET = 10001

    for r, c in stones:
        # Union the Row Index and the Column Index
        uf.union(r, c + COL_OFFSET)

    # The logic:
    # Total Stones = N
    # Number of un-removable stones = Number of Connected Components (uf.count)
    # Max Removed = Total - Components

    return len(stones) - uf.count
```

JavaScript Solution

```javascript
/**
 * L6 Style: Clean, Class-based, Optimized DSU
 */
class UnionFind {
    constructor() {
        this.parent = new Map();
        this.count = 0; // Number of connected components
    }

    find(x) {
        // Initialization on the fly
        if (!this.parent.has(x)) {
            this.parent.set(x, x);
            this.count++;
        }

        // Path compression
        if (this.parent.get(x) !== x) {
            this.parent.set(x, this.find(this.parent.get(x)));
        }
        return this.parent.get(x);
    }

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

```javascript
        if (rootX !== rootY) {
            this.parent.set(rootX, rootY);
            this.count--; // Two components became one
        }
    }
}

/**
 * @param {number[][]} stones
 * @return {number}
 */
var removeStones = function(stones) {
    const uf = new UnionFind();
    const COL_OFFSET = 10001; // Safety buffer to separate Row IDs from Col IDs

    for (const [r, c] of stones) {
        // Connect the row node to the column node
        uf.union(r, c + COL_OFFSET);
    }

    // Result is Total Stones - Number of Independent Clusters
    return stones.length - uf.count;
};
```

Note 1: Terminology & Techniques

1. Connected Components: In an undirected graph, a connected component is a subgraph in which any two vertices are connected to each other by paths. In this problem, an "Island" of stones is a connected component.

2. Disjoint Set Union (DSU) / Union-Find: This is the data structure used. It is specialized for tracking elements partitioned into disjoint (non-overlapping) sets. It supports two primary operations:

- **Find:** Determine which subset an element is in.
- **Union:** Join two subsets into a single subset. It is "near constant time" - - which makes it incredibly fast for connectivity problems.

Note 2: Real World Interview Variations

This specific pattern (grouping items based on shared attributes) is very common in System Design and Algorithmic interviews at top-tier firms.

1. Bloomberg: "User Identity Resolution" Problem: You have a stream of login events. `[User_A, Device_1]`, `[User_B, Device_1]`, `[User_B, Device_2]`.

- User A used Device 1.
- User B used Device 1 (So User A and B are likely the same person or household).
- User B used Device 2.
- **Question:** How many unique "households" or real identities do we have?

Solution Mapping:

- This is exactly the Stone problem.
- Rows = UserIDs, Columns = DeviceIDs.
- Union(User, Device).
- Count unique roots at the end.

2. Meta (Facebook): "Friends Transitivity" Problem: You are given a list of pairs `[A, B]` meaning A is friends with B. If A is friends with B, and B is friends with C, then A and C are indirectly connected.

- **Question:** Calculate the total size of the largest group of connected friends.

Solution Mapping:

- Stones are people. The edges are explicit here (friendships).
- Use DSU. `Union(A, B)`.
- Instead of just counting components, track the `size` of each component in the DSU class. Return `max(size)`.

3. Google: "Pixel Clustering / Computer Vision" Problem: You have a large black and white image (grid of 0s and 1s).

- **Question:** Count the number of black objects. Black pixels connect if they are adjacent (up/down/left/right).

Solution Mapping:

- Standard "Number of Islands" problem.
- While BFS/DFS is common here, DSU is often used if the image comes in a "stream" (pixels arriving one by one) and you need to answer the query in real-time.
- Union current pixel with its active neighbors.
- `Count = Total Black Pixels - Unions Performed`.

394. Decode String

Here is how a Senior Software Engineer (L5/L6) would break down **LeetCode 394. Decode String**.

At this level, we aren't just solving the problem; we are demonstrating mastery over **parsing**, **state management** (via Stacks), and writing **production-ready code** that is easy to read and debug.

---

1. Problem Explanation

The core task is to uncompress a string that follows a specific encoding pattern: `k[encoded_string]`. This means the `encoded_string` inside the brackets is repeated exactly `k` times.

Key Rules:

1. `k` is always a positive integer.
2. The input string is always valid (no missing brackets, etc.).
3. The structure can be nested! (e.g., `3[a2[c]]`).

Examples:

- **Simple:** `3[a]` aaa
- **Sequential:** `3[a]2[bc]` aaabcbc
- **Nested (The tricky part):** `3[a2[c]]`
- First resolve the inner: `2[c]` cc
- Now it looks like: `3[acc]`
- Resolve outer: `accaccacc`

Visualizing the "Why": Think of this like an onion or a mathematical expression. You cannot solve the outer layer until you have solved the inner layer. This "Last-In, First-Out" dependency screams **Stack** data structure.

---

2. Solution Explanation: The Dual-Stack Approach

An L5 engineer recognizes this as a classic **parsing** problem. We need to remember "where we were" before diving into a nested bracket.

We will iterate through the string character by character. We need to keep track of two things:

1. **Counts:** The numbers we see (how many times to repeat).
2. **Strings:** The characters we've built so far that are waiting for a closing bracket `]`.

The Algorithm (Iterative Stack):

1. **Current Number (`currNum`):** Tracks the digit we are reading (e.g., handles "100" by reading 1, 0, 0).
2. **Current String (`currStr`):** Tracks the string we are currently building at this nesting level.
3. **The Stack:** When we see an opening bracket [, it implies we are entering a **new** nesting level. We must PAUSE our current work.

- Push `currStr` and `currNum` onto the stack.
- Reset `currStr` and `currNum` for the new inner level.

4. **Closing Bracket]:** This implies we finished a nesting level. We need to pop the old work from the stack and combine it with what we just built.

Detailed ASCII Trace of `3[a2[c]]`

```
Initialize:
  Stack   = []
  currNum = 0
  currStr = ""

STEP 1: Process '3'
  Action: It's a digit. Update currNum.
  currNum = 3
  currStr = ""
  Stack   = []

STEP 2: Process '['
  Action: Start of new section! SAVE current state to stack.
  Push (3, "") to Stack. Reset variables.
  currNum = 0
  currStr = ""
  Stack   = [(3, "")]  <-- Saved: "Repeat whatever comes next 3 times, add to empty string"

STEP 3: Process 'a'
  Action: It's a char. Append to currStr.
  currNum = 0
  currStr = "a"
  Stack   = [(3, "")]

STEP 4: Process '2'
  Action: It's a digit. Update currNum.
  currNum = 2
  currStr = "a"
  Stack   = [(3, "")]

STEP 5: Process '['
  Action: Start of NEW nested section! SAVE state.
  Push (2, "a") to Stack. Reset variables.
```

```
  currNum = 0
  currStr = ""
  Stack   = [(3, ""), (2, "a")] <-- Top implies: "Repeat next part 2 times, append to 'a'"

STEP 6: Process 'c'
  Action: It's a char. Append to currStr.
  currNum = 0
  currStr = "c"
  Stack   = [(3, ""), (2, "a")]

STEP 7: Process ']'
  Action: End of section! POP from stack and combine.
  1. Pop (prevNum, prevStr) -> (2, "a")
  2. New currStr = prevStr + (currStr * prevNum)
                 = "a"     + ("c"     * 2)
                 = "acc"
  currNum = 0
  currStr = "acc"
  Stack   = [(3, "")]

STEP 8: Process ']'
  Action: End of section! POP from stack and combine.
  1. Pop (prevNum, prevStr) -> (3, "")
  2. New currStr = prevStr + (currStr * prevNum)
                 = ""      + ("acc"   * 3)
                 = "accaccacc"
  currNum = 0
  currStr = "accaccacc"
  Stack   = []

RESULT: "accaccacc"
```

---

3. Time and Space Complexity Analysis

Time Complexity: O(MaxK * N) where `MaxK` is the maximum value of `k` and `N` is the length of the string. *Wait, isn't it just linear?* Not necessarily. If the output string is massive (e.g., `100000[a]`), we physically have to write that many characters. The time taken is proportional to the **size of the output**.

```
VISUALIZATION of Time Cost:

Input: 10[a]   (Length 5)
Output: aaaaaaaaaa (Length 10)
Cost: We write 10 characters.
```

```
Input: 2[3[a]] (Length 7)
       Inner 3[a] -> aaa (Cost 3)
       Outer 2[...] -> aaa + aaa (Cost 6)
Total Operations ~ Size of generated string.
```

Space Complexity: O(Sum(MaxK * N)) or O(Output Size) (worst case) We are building the result string. In the worst case (huge expansion), we store that massive string in memory. For the **Stack specifically**, the space is **O(D)**, where D is the depth of nesting (e.g., `[[[[...]]]]` requires a deeper stack).

```
VISUALIZATION of Stack Space:

Input: 3[a]
Stack Depth: | (1 level)

Input: 3[a2[b4[c]]]
Stack Depth: |
             | (3 levels deep)
             |
```

---

4. Solution Code

The following code is written in a "Google Style"—clean variable names, modular logic, and clear separation of concerns.

Python Implementation

```python
class Solution:
    def decodeString(self, s: str) -> str:
        """
        Decodes a string with k[content] format.
        Uses a stack to handle nested structures.
        """
        stack = []
        current_str = ""
        current_num = 0

        for char in s:
            if char.isdigit():
                # Handle multi-digit numbers (e.g., "100")
                # We shift the existing digit left (multiply by 10) and add the new one
                current_num = current_num * 10 + int(char)

            elif char == '[':
                # Entering a new nesting level.
```

```python
                # Push the context (number of times to repeat, and string built so far)
                # onto the stack so we can return to it later.
                stack.append((current_str, current_num))

                # Reset for the new inner scope
                current_str = ""
                current_num = 0

        elif char == ']':
            # Exiting a nesting level.
            # Retrieve the context from before we entered this bracket.
            prev_str, repeat_count = stack.pop()

            # Combine: The string we had BEFORE + (The INNER string * count)
            current_str = prev_str + (current_str * repeat_count)

        else:
            # Regular character, just build the string
            current_str += char

    return current_str
```

JavaScript Implementation

```javascript
/**
 * @param {string} s
 * @return {string}
 */
var decodeString = function(s) {
    let stack = [];
    let currentStr = "";
    let currentNum = 0;

    for (let char of s) {
        if (!isNaN(char)) {
            // It is a digit. Build the number.
            // ASCII math: char - '0' implies conversion to integer if doing raw implementa
            // but JS handles 'currentNum * 10 + Number(char)' smoothly.
            currentNum = currentNum * 10 + Number(char);
        }
        else if (char === '[') {
            // Push current state to stack: [string_before, number_for_next]
            stack.push([currentStr, currentNum]);

            // Reset
            currentStr = "";
```

```
            currentNum = 0;
        }
        else if (char === ']') {
            // Pop the state
            let [prevStr, repeatCount] = stack.pop();

            // Expansion logic:
            // prevStr is the "prefix" before this bracket group.
            // currentStr is the "content" inside the bracket group.
            // repeatCount applies ONLY to the content.
            currentStr = prevStr + currentStr.repeat(repeatCount);
        }
        else {
            // Regular character
            currentStr += char;
        }
    }

    return currentStr;
};
```

Note 1: Terms & Techniques

1. Recursive Descent Parsing (Alternative View): While we used an iterative Stack approach (which is generally safer against Stack Overflow errors in production for extremely deep nesting), this problem fits the definition of a **Context-Free Grammar**. We are essentially building a mini-parser. In an interview, mentioning that "This is essentially a simplified parser for a Context-Free Grammar" signals strong theoretical computer science fundamentals.

2. Flattening: The operation `prevStr + (currStr * count)` is essentially "flattening" a nested structure into a linear sequence.

Note 2: Real World Interview Variations (Google, Meta, Bloomberg)

At L5/L6, the interviewer will often pivot from the algorithmic solution to a "System Design" or "Real World Application" discussion.

Variation 1: The "Decompression Bomb" (Security/Reliability)

- **The Question:** "What if the input is `10000000[a]`? or nested `100[100[100[...]]]`? How does your code behave?"
- **The Issue:** The output string might exceed available RAM (OOM - Out of Memory kill).

135

- **L5 Solution:** Do not build the string in memory. Instead, create a **Stream** or **Iterator**.
- Instead of returning `str`, return a generator.
- The generator yields characters one by one.
- `1000[a]` doesn't create a massive string; it just loops 1000 times yielding 'a'.
- This transforms Space Complexity from `O(Output Size)` to `O(Stack Depth)`, which is massive optimization.

Variation 2: Configuration / Template Parsing (Infrastructure)

- **The Question:** "We have a config file where variable `${VAR}` implies a value. Variables can be nested `${A_${B}}`. Parse it."
- **L5 Solution:** This is the same problem but with a map lookup instead of a repeat count.
- `3[...]` becomes `LOOKUP(...)`.
- You would modify the code to accept a `Dictionary/Map` of values.
- When you hit `]`, instead of repeating, you query the Map and append the result.

Variation 3: Invalid Inputs (Robustness)

- **The Question:** "The problem statement says input is always valid. In the real world, it isn't. Handle `3[a2[c` (missing closing bracket)."
- **L5 Solution:**
- At the end of the loop, check `if stack is not empty`.
- If the stack has items left, it means we have unclosed brackets.
- Throw a descriptive `SyntaxError` showing exactly where the bracket was expected. This is "Developer Experience" (DX) thinking.

399. Evaluate Division

An L5/L6 engineer doesn't just jump into coding. They treat this as a **Graph Modeling** problem. At this level, the goal is to identify that variables are nodes and ratios are directed, weighted edges.

---

1. Problem Explanation

You are given equations like `a / b = 2.0`. In your head, you should immediately translate that into a map:

- If I know `a` and I want `b`, I divide by 2.
- If I know `b` and I want `a`, I multiply by 2.

The problem asks you to find the result of new queries (e.g., `a / c = ?`) based on these known relationships. If there is no path between two variables, the answer is `-1.0`.

The "Hidden" Logic

If `a / b = 2.0` and `b / c = 3.0`, then `a / c` is just the chain: `(a / b) * (b / c)`. Numerically: `2.0 * 3.0 = 6.0`.

---

2. Solution Explanation

To an experienced engineer, this is a **Directed Weighted Graph**.

Step 1: Building the Graph

We represent the equations as an adjacency list. For every equation `u / v = value`, we store:

1. A path from `u` to `v` with weight `value`.
2. A path from `v` to `u` with weight `1 / value`.
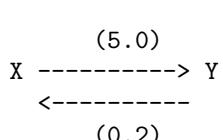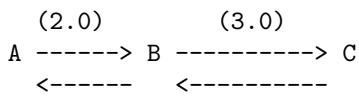
Step 2: The Traversal (BFS or DFS)

To solve `a / d`, we start a search at `a` and try to reach `d`. As we move from node to node, we **multiply** the edge weights.

ASCII Visualization of the Logic

Let's say we have: `a/b = 2.0, b/c = 3.0, x/y = 5.0`

```
GRAPH STRUCTURE:

  (2.0)          (3.0)
A ------> B ----------> C
  <------    <----------
  (0.5)          (0.33)

         (5.0)
   X ----------> Y
     <----------
         (0.2)

QUERY: "What is a / c?"
1. Start at A. Current value = 1.0
2. Move to B. New value = 1.0 * 2.0 = 2.0
3. Move to C. New value = 2.0 * 3.0 = 6.0
RESULT: 6.0

QUERY: "What is a / x?"
1. Start at A.
2. Search all connected nodes (B, C).
```

```
3. X is never reached because it's in a different "island" (component).
RESULT: -1.0
```

---

3. Time and Space Complexity Analysis

At the L5/L6 level, we analyze complexity based on N (number of equations) and Q (number of queries).

Time Complexity (TC)

```
BUILDING THE GRAPH:
Iterating through N equations takes O(N) time.

PROCESSING QUERIES:
For each of the Q queries:
- In the worst case, we visit every variable (node) once.
- Let V be the number of unique variables.
- Each query takes O(V).

TOTAL TC: O(N + Q * V)
```

Space Complexity (SC)

```
ADJACENCY LIST:
We store 2 edges for every equation.
Total edges = 2 * N.
Total nodes = V.

TOTAL SC: O(N + V)
(To store the graph structure and the visited set during search)
```

---

4. Solution Code

Python Implementation

```python
from collections import defaultdict, deque

def calcEquation(equations, values, queries):
    # Step 1: Build the graph
    # graph[node] = [(neighbor, weight), ...]
    graph = defaultdict(list)

    for (u, v), val in zip(equations, values):
        graph[u].append((v, val))
```

```python
            graph[v].append((u, 1.0 / val))

        # Helper function to perform BFS for each query
        def find_path(start, end):
            if start not in graph or end not in graph:
                return -1.0
            if start == end:
                return 1.0

            queue = deque([(start, 1.0)])
            visited = {start}

            while queue:
                curr_node, curr_prod = queue.popleft()

                for neighbor, weight in graph[curr_node]:
                    if neighbor == end:
                        return curr_prod * weight
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, curr_prod * weight))
            return -1.0

        return [find_path(q[0], q[1]) for q in queries]
```

Javascript Implementation

```javascript
/**
 * Solves Evaluate Division using an Adjacency List and BFS.
 */
var calcEquation = function(equations, values, queries) {
    const graph = new Map();

    // Build the graph: O(N)
    equations.forEach(([u, v], i) => {
        if (!graph.has(u)) graph.set(u, []);
        if (!graph.has(v)) graph.set(v, []);
        graph.get(u).push([v, values[i]]);
        graph.get(v).push([u, 1 / values[i]]);
    });

    // BFS function to find the product path: O(V)
    const bfs = (src, dst) => {
        if (!graph.has(src) || !graph.has(dst)) return -1.0;
        if (src === dst) return 1.0;
```

```
        let queue = [[src, 1.0]];
        let visited = new Set([src]);

        while (queue.length > 0) {
            let [curr, product] = queue.shift();

            for (let [neighbor, weight] of graph.get(curr)) {
                if (neighbor === dst) return product * weight;
                if (!visited.has(neighbor)) {
                    visited.add(neighbor);
                    queue.push([neighbor, product * weight]);
                }
            }
        }
        return -1.0;
    };

    return queries.map(([u, v]) => bfs(u, v));
};
```

Key Terms & Techniques

- **Adjacency List:** A way to represent a graph where each node stores a list of its neighbors. It's more space-efficient than a matrix for "sparse" data (where most variables aren't directly related).
- **BFS (Breadth-First Search):** We explore layer-by-layer. In this problem, it's used to find *any* path between two variables.
- **Union-Find (Alternative):** An L6 might suggest Union-Find with "path compression" and "weights" to solve this in near-constant time per query, though BFS/DFS is usually sufficient and easier to maintain.

Real-World Interview Variations

1. Currency Exchange (Google/Bloomberg)

Scenario: You have exchange rates (USD/EUR = 0.9). Find the best rate between two currencies.

- **Solving it:** This is exactly the same problem, but if there are multiple paths, you might need **Dijkstra's Algorithm** to find the *maximum* product path (best rate) instead of just *any* path.

2. Unit Conversion Tool (Meta/Google)

Scenario: Convert "feet" to "meters" given "feet to inches" and "inches to meters".

- **Solving it:** This is the standard "Evaluate Division" problem. The "variables" are units (ft, in, m).

3. Build System Dependencies (Google)

Scenario: To build library A, you need 2 units of library B. To build B, you need 3 units of C. How many C's for one A?

- **Solving it:** Model as a directed graph. This is often combined with **Topological Sort** if you need to determine the order of operations, but the quantity calculation is the same multiplication-path logic used here.

Would you like me to walk through the Union-Find optimization for this problem to see how an L6 might over-engineer it for maximum performance?

Robot Room Cleaner

A Google L5 (Senior) or L6 (Staff) engineer doesn't just look for a way to pass the test cases; they look for a **robust, modular, and mathematically sound** approach. For "Robot Room Cleaner," the challenge isn't just "moving around"—it's managing state in an environment where you have no map.

---

1. Problem Explanation

Imagine you are a blindfolded robot in a messy room. You have a `clean()` function, but you don't know where the walls are until you bump into them. You can move forward, turn left, or turn right.

The constraints:

- You don't know the room dimensions.
- You don't know your starting coordinates.
- You must clean every reachable square.

The Mental Model (The Grid)

Even though we don't have a map, we can **create one** as we go using a coordinate system where our starting point is (0, 0).

```
Initial State:        After moving 'Up':       After turning 'Right':
  (Unknown)               (0, 1)                     (0, 1)
     |                       ^                           ^
```

```
   [Robot @ 0,0]            [Robot @ 0,1]            [Robot @ 0,1] -> facing East
        |                        |                        |
       _____
```

2. Solution Explanation

An L6 engineer would identify this as a **Backtracking Depth-First Search (DFS)** on an implicit graph.

The Strategy

1. **Mark and Clean:** As soon as you land on a cell `(x, y)`, add it to a `Set` of "visited" cells and call `robot.clean()`.
2. **Explore 4 Directions:** From your current cell, try to move in all four directions (Up, Right, Down, Left).
3. **The "Handshake" (Move & Backtrack):** This is the non-trivial part. If you move forward into a new cell and finish exploring it, you **must** return to your previous cell and face your previous direction to maintain the DFS integrity.

Visualization of the DFS Path

Legend: `S` = Start, `.` = Cleaned, `W` = Wall, `R` = Robot, `*` = Target

```
STEP 1: Start at (0,0)        STEP 2: Move Up to (0,1)       STEP 3: Hit Wall at (0,2)
    W W W                          W W W                          W W W
    W . W  <-- (0,1)               W R W                          W R W (Blocked!)
    W R W  <-- (0,0)               W . W                          W . W
    W W W                          W W W                          W W W


STEP 4: Backtrack to (0,0)    STEP 5: Turn Right, Move (1,0) STEP 6: Full Map Cleaned
    W W W                          W W W                          W W W
    W . W                          W . W                          W . . W
    W R W                          W . R W                        W . . W
    W W W                          W W W                          W W W
```

The Non-Trivial Part: The "Go Back" Maneuver

In a standard DFS (like on a 2D array), you just return from a function. But a physical robot is still in the new cell! You have to manually move it back.

The "Undo" Algorithm:

1. Turn 180 degrees (`robot.turnRight()` twice).
2. Move forward (`robot.move()`).
3. Turn 180 degrees again to face the original direction.

       _____

3. Time and Space Complexity Analysis

Since we are avoiding LaTeX, here is the derivation in plain text:

Time Complexity

```
Every cell in the room is visited exactly once.
For every cell, we attempt to move in 4 directions.
Let N = Number of non-wall cells in the room.
Let M = Number of obstacles (walls) bordering these cells.

Total Work = N (cleaning) + 4 * N (checking directions)
Complexity: O(N)
```

Space Complexity

```
We store the coordinates of every visited cell in a Hash Set.
We also have the recursion stack for the DFS.
In the worst case (a long narrow corridor), the stack depth is N.

Space = Set Storage + Recursion Stack
Complexity: O(N)
```

---

4. Solution Code

Python Implementation

```python
class Solution:
    def cleanRoom(self, robot):
        # Using a set to keep track of coordinates we've already cleaned
        visited = set()
        # Directions: 0: Up, 1: Right, 2: Down, 3: Left (Clockwise)
        directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

        # This function physically moves the robot back to the previous cell
        # and restores its original orientation. Essential for DFS.
        def go_back():
            robot.turnRight()
            robot.turnRight()
            robot.move()
            robot.turnRight()
            robot.turnRight()

        def backtrack(cell, d):
            visited.add(cell)
            robot.clean()
```

```python
        # Explore 4 directions starting from the current orientation 'd'
        for i in range(4):
            new_d = (d + i) % 4
            new_cell = (
                cell[0] + directions[new_d][0],
                cell[1] + directions[new_d][1]
            )

            if new_cell not in visited and robot.move():
                backtrack(new_cell, new_d)
                # After exploring the branch, return to this cell
                go_back()

            # Turn the robot to the next direction in the loop
            robot.turnRight()

    # Start DFS at (0,0) facing Up (direction 0)
    backtrack((0, 0), 0)
```

JavaScript Implementation

```javascript
var cleanRoom = function(robot) {
    const visited = new Set();
    // Up, Right, Down, Left
    const directions = [[-1, 0], [0, 1], [1, 0], [0, -1]];

    /**
     * Resets robot position to the parent node in the DFS tree
     * without losing the track of the current orientation logic.
     */
    const goBack = () => {
        robot.turnRight();
        robot.turnRight();
        robot.move();
        robot.turnRight();
        robot.turnRight();
    };

    const backtrack = (x, y, d) => {
        visited.add(`${x},${y}`);
        robot.clean();

        for (let i = 0; i < 4; i++) {
            let newD = (d + i) % 4;
            let nextX = x + directions[newD][0];
```

```
            let nextY = y + directions[newD][1];

            if (!visited.has(`${nextX},${nextY}`) && robot.move()) {
                backtrack(nextX, nextY, newD);
                goBack();
            }
            // Always turn to maintain the 'newD' alignment in the next loop
            robot.turnRight();
        }
    };

    backtrack(0, 0, 0);
};
```

Note 1: Terms and Techniques

- **DFS (Depth-First Search):** A strategy for traversing a tree or graph where you go as deep as possible along a branch before backtracking. In this problem, it ensures we don't get "lost" in the room.
- **Implicit Graph:** The room isn't given as a data structure (like an array). We "build" the graph in our minds and in our `visited` set as we explore.
- **Backtracking:** The act of "undoing" a move. Because the robot is a physical object in a stateful environment, backtracking requires actual physical turns and moves to stay synced with the algorithm.

Note 2: Real-World Interview Variations

Google, Meta, and Bloomberg often wrap this logic in different "skins":

1. **The "Worm in a Maze" (Google):** You control a worm that can only move forward and turn. There is food scattered. You must find all food.

- **Solution:** Same DFS logic, but instead of `robot.clean()`, you check for `hasFood()`. You must also handle the "length" of the worm if the problem specifies it (usually it's just a 1x1 head).

2. **The "Distributed Web Crawler" (Meta):** You are given a starting URL and a "Robot" class that can `fetchLinks()`. You don't know the whole internet.

- **Solution:** This is the same problem! The "Room" is the Internet, the "Cells" are URLs, and "Moving" is clicking a link. You use a `Set` to avoid infinite loops and DFS/BFS to visit every page.

3. **The "Mining Drone" (Bloomberg):** A drone in a 3D cave needs to map minerals.

- **Solution:** This extends the Robot Room Cleaner to 3D. Instead of 4 directions, you have 6 (Up, Down, North, South, East, West). The "Go Back" logic remains the same.

753. Cracking the Safe

A Google L6 (Staff Engineer) doesn't just look for a solution; they look for the **underlying mathematical structure**. For "Cracking the Safe," that structure is a **De Bruijn sequence**.

---

1. Problem Explanation

Imagine you have a safe with an -digit passcode. Each digit can be from to . You don't have to hit "Enter" after every digits. The safe stays open if the **last digits** you typed match the code.

The Goal: Find the shortest possible string that contains every possible -digit combination as a substring.

The "Aha!" Moment

If and , our combinations are: `00, 01, 10, 11`.

- **Brute force approach:** Just concatenate them: `00011011` (Length 8).
- **The "L6" approach:** Overlap them! `00110`
- `00` (digits 1-2)
- `01` (digits 2-3)
- `11` (digits 3-4)
- `10` (digits 4-5)
- Total length: 5.

---

2. Solution Explanation

To solve this efficiently, we treat the problem as finding an **Eulerian Path** in a directed graph.
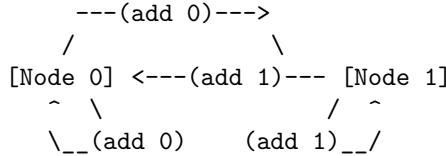
The Graph Setup

- **Nodes:** All possible strings of length .
- **Edges:** Represent adding a digit (0 to) to a node to reach a new node.

- **The Logic:** Each edge represents a full -digit code. To use every code exactly once, we need to visit every **edge** in the graph exactly once.

ASCII Visualization (n=2, k=2)

Nodes are length (length 1). Nodes: `0` and `1`.

```
    ---(add 0)--->
   /              \
[Node 0] <---(add 1)--- [Node 1]
   ^  \              /  ^
   \__(add 0)    (add 1)__/


 Edge Details:
 From '0', add '0' -> Code "00", land at '0'
 From '0', add '1' -> Code "01", land at '1'
 From '1', add '0' -> Code "10", land at '0'
 From '1', add '1' -> Code "11", land at '1'
```

The Hierholzer's Algorithm (DFS)

We use a Depth First Search. As we "finish" visiting all edges from a node, we add the digit used to our result. This builds the sequence in reverse.

Step-by-Step for n=2, k=2:

1. Start at node `0`.
2. Try edge `0`: Go to node `0` (visit edge `00`).
3. From node `0`, edge `0` is used. Try edge `1`: Go to node `1` (visit edge `01`).
4. From node `1`, try edge `0`: Go to node `0` (visit edge `10`).
5. From node `0`, all edges (`0,1`) used. Backtrack to node `1`.
6. From node `1`, try edge `1`: Go to node `1` (visit edge `11`).
7. All edges done.

---

3. Complexity Analysis

Time Complexity (TC)

We visit every possible combination exactly once. There are combinations.

```
Total Combinations = k raised to the power of n
Each visit takes O(1) time (using a Set for visited edges)
TC = O(k^n)
```

Space Complexity (SC)

We store the visited combinations and the recursion stack.

```
Recursion Stack Depth = k^n (worst case)
Visited Set Size       = k^n
Result String          = k^n + (n-1)
SC = O(k^n)
```

---

4. Solution Code

Python Implementation

```python
class Solution:
    def solve(self, n, k):
        # We start with a prefix of n-1 zeros
        start_node = "0" * (n - 1)
        visited = set()
        result = []

        def dfs(node):
            for x in range(k):
                # Form the full n-digit code
                combination = node + str(x)
                if combination not in visited:
                    visited.add(combination)
                    # Move to the next node (last n-1 digits of current combo)
                    next_node = combination[1:]
                    dfs(next_node)
                    # Post-order: add the digit after exploring everything
                    result.append(str(x))

        dfs(start_node)
        # The result is built backwards, append the starting prefix at the end
        return "".join(result) + start_node
```

Javascript Implementation

```javascript
/**
 * Hierholzer's Algorithm to find an Eulerian Path.
 * We treat strings of length n-1 as nodes and the
 * added digit as an edge creating a length n string.
 */
var crackSafe = function(n, k) {
    let visited = new Set();
    let result = [];
    let startNode = "0".repeat(n - 1);

    function dfs(node) {
```

```
        for (let i = 0; i < k; i++) {
            let combination = node + i;
            if (!visited.has(combination)) {
                visited.add(combination);
                // Transition to the next state by shifting window
                dfs(combination.slice(1));
                result.push(i);
            }
        }
    }

    dfs(startNode);
    return result.join('') + startNode;
};
```

Terminology Deep Dive

- **De Bruijn Sequence:** A specific sequence where every possible string of length over an alphabet of size appears exactly once as a contiguous substring.
- **Eulerian Path:** A trail in a graph which visits every **edge** exactly once. Because every edge in our state-graph is a unique passcode, finding this path solves the "shortest string" problem.

Real World / Interview Variants

1. The "Minimal Sensor Log" (Google/Meta)

- *Scenario:* A sensor captures a window of events. You want to test if the system handles all possible sequences of event types. How do you generate the shortest test sequence?
- *Solution:* Identical to Cracking the Safe. Map event types to digits .

2. Circular Buffer Patterns (Bloomberg)

- *Scenario:* You have a circular hardware buffer. You need to fill it such that any consecutive bits represent a unique ID.
- *Solution:* This is a **Hamiltonian Path** problem on a different graph or an **Eulerian Path** on the De Bruijn graph. The "circular" nature means you don't need the padding at the end; the wrap-around handles it.

3. DNA Sequence Reconstruction (Biotech/Research)

- *Scenario:* Given all possible -mers (substrings of length), reconstruct the original shortest DNA strand.

- *Solution:* This is often solved by building a De Bruijn graph where nodes are overlaps.

433. Minimum Genetic Mutation

A Google L5 or L6 engineer doesn't just look for *a* solution; they look for the most **robust**, **efficient**, and **scalable** pattern. For a problem like "Minimum Genetic Mutation," they immediately recognize this as a **Shortest Path problem in an unweighted graph.**

Here is how a top-tier engineer breaks this down.

---

1. Problem Explanation

You have a starting gene string (`start`), an ending gene string (`end`), and a list of valid mutations (`bank`).

- A "mutation" means changing exactly **one** character in the 8-character string.
- Each intermediate mutation **must** exist in the `bank`.
- You need the **minimum** number of mutations to get from start to end.

The "Mental Model"

Think of each gene string as a **node** in a giant web (a graph). If two strings differ by only one letter, there is a **bridge** (edge) between them. We are looking for the shortest bridge-crossing path from Start to End.

---

2. Solution Explanation

Why Breadth-First Search (BFS)?

An L5/L6 engineer chooses BFS because BFS explores the graph in "waves." It visits all neighbors 1 step away, then all neighbors 2 steps away, and so on. The moment we hit the `end` gene, we are **guaranteed** that the number of steps taken is the shortest possible path.

The Non-Trivial Part: "Building the Neighbors"

How do we find which genes in the bank are reachable from our current gene?

1. **Naive approach:** Compare the current gene against every string in the bank (Slow if the bank is huge).

2. **Engineer approach:** Iterate through all 8 positions of the current gene, swap the character with 'A', 'C', 'G', or 'T', and check if that new string is in the `bank` (Fast because the gene length is fixed at 8).

ASCII Visualization of the BFS Flow

```
START: "AACCGGTT"
BANK: ["AACCGGTA", "AACCGCTA", "AAACGGTA"]
END: "AAACGGTA"

STEP 0:
Queue: [ ("AACCGGTT", 0) ]  <-- (Current Gene, Mutation Count)
Visited: {"AACCGGTT"}

STEP 1: Pop "AACCGGTT". Try changing each char...
"AACCGGT A" is in Bank!
Queue: [ ("AACCGGTA", 1) ]
Visited: {"AACCGGTT", "AACCGGTA"}

STEP 2: Pop "AACCGGTA". Try changing each char...
"AACCGG T A" -> "AACCG C TA" (In Bank)
"A A CCGGTA" -> "A A ACGGTA" (In Bank - TARGET FOUND!)

PATH DISCOVERED:
AACCGGTT -> AACCGGTA -> AAACGGTA (2 Mutations)
```

---

3. Time and Space Complexity Analysis

An L5 engineer avoids simple "Big O" and explains the **why** behind the constraints.

Time Complexity (TC)

We process each gene in the bank at most once. For each gene, we perform 8 (length) * 4 (ACGT) operations.

```
+----------------------------------------------------------+
| CALCULATION                                              |
+----------------------------------------------------------+
| N = Number of genes in the bank                          |
| L = Length of the gene (Fixed at 8)                      |
| A = Alphabet size (A, C, G, T = 4)                       |
|                                                          |
| Total TC = O(N * L^2 * A)                                |
|                                                          |
```

```
| Why L^2? One 'L' to iterate through the string, and     |
| another 'L' to create the new string (string slicing).  |
+---------------------------------------------------------+
```

Space Complexity (SC)

We store the bank in a Set for O(1) lookups and maintain a Queue/Visited set.

```
+---------------------------------------------------------+
| ALLOCATION                                              |
+---------------------------------------------------------+
| Bank Set: O(N * L)                                      |
| Queue:    O(N * L)                                      |
| Visited:  O(N * L)                                      |
|                                                         |
| Total SC = O(N * L)                                     |
+---------------------------------------------------------+
```

---

4. Solution Code

Python Snippet

```python
from collections import deque

def minMutation(startGene: str, endGene: str, bank: list[str]) -> int:
    # L5 Optimization: Convert bank to a set for O(1) lookups
    bank_set = set(bank)
    if endGene not in bank_set:
        return -1

    # BFS Queue: stores (current_gene, steps)
    queue = deque([(startGene, 0)])
    # Visited set to prevent infinite loops
    visited = {startGene}

    while queue:
        curr, steps = queue.popleft()

        if curr == endGene:
            return steps

        # Try all 8 positions
        for i in range(len(curr)):
            # Try all 4 possible nucleotides
            for char in "ACGT":
```

```python
            mutation = curr[:i] + char + curr[i+1:]

            # Check if it's a valid mutation we haven't seen
            if mutation in bank_set and mutation not in visited:
                visited.add(mutation)
                queue.append((mutation, steps + 1))

    return -1
```

JavaScript Snippet

```javascript
/**
 * Finds the minimum mutations using Breadth-First Search.
 * @param {string} startGene
 * @param {string} endGene
 * @param {string[]} bank
 * @return {number}
 */
var minMutation = function(startGene, endGene, bank) {
    const bankSet = new Set(bank);
    if (!bankSet.has(endGene)) return -1;

    const queue = [[startGene, 0]];
    const visited = new Set();
    visited.add(startGene);

    while (queue.length > 0) {
        const [curr, steps] = queue.shift();

        if (curr === endGene) return steps;

        // Standard L6 practice: keep character options in a constant
        const nucleotides = ['A', 'C', 'G', 'T'];

        for (let i = 0; i < curr.length; i++) {
            for (let char of nucleotides) {
                // Construct the new string
                const mutation = curr.substring(0, i) + char + curr.substring(i + 1);

                if (bankSet.has(mutation) && !visited.has(mutation)) {
                    visited.add(mutation);
                    queue.push([mutation, steps + 1]);
                }
            }
        }
    }
```

153

```
    return -1;
};
```

Key Terms & Algorithm Logic

- **Breadth-First Search (BFS):** An algorithm for traversing tree or graph data structures. It starts at the root and explores all neighbor nodes at the present depth prior to moving on to nodes at the next depth level.
- **Set (Hash Set):** Used here to turn a list lookup (which is O(N)) into a constant time lookup (O(1)). In interviews, this is the "low-hanging fruit" optimization that signals seniority.

Real World / Interview Variations

1. The "Word Ladder" (Meta/Google)

- **Variation:** Given two words and a dictionary, find the shortest transformation sequence.
- **Solution:** Identical logic. However, if the dictionary is massive, L6s suggest **Bidirectional BFS** (searching from both `start` and `end` simultaneously) to meet in the middle, which drastically reduces the search space.

2. Social Network Connectivity (Bloomberg)

- **Variation:** Find the shortest "degrees of separation" between two users.
- **Solution:** Users are nodes; friendships are edges. Use BFS. Since user "banks" are billions of rows, engineers use distributed processing or graph databases (like DGraph or Neo4j).

3. Microservice Dependency Mapping (Google)

- **Variation:** If Service A fails, what is the minimum number of "hops" to find the root cause or the impacted service?
- **Solution:** This is a directed graph problem. BFS is used to map out the blast radius of a system failure.

130. Surrounded Regions

A Google L5/L6 engineer doesn't just look for *a* solution; they look for the most **robust**, **memory-efficient**, and **readable** solution. For "Surrounded Regions," an experienced engineer recognizes this as a connectivity problem disguised as a grid problem.

1. Problem Explanation

Imagine a grid of 'X's and 'O's. An 'O' is "surrounded" if it is completely enclosed by 'X's. If an 'O' is connected to the **edge** of the board, it is "safe"—it cannot be captured because it has an "escape route" to the outside world.

The Goal: Flip all 'O's that are NOT connected to the border into 'X's.

The Mental Model

Think of the border 'O's as **anchors**. Anything connected to an anchor is "safe." Everything else is "captured."

---

2. Solution Explanation: The "Reverse Flood" Strategy

Instead of trying to figure out which 'O's are surrounded (which is hard), we figure out which 'O's are **not** surrounded (which is easy).

The Algorithm Steps:

1. **Traverse the Borders:** Iterate through the four edges of the grid.
2. **Mark the Survivors:** If you find an 'O' on the edge, start a DFS (Depth First Search) or BFS (Breadth First Search) to mark it and all connected 'O's with a temporary character (like 'S' for Survivor).
3. **Final Pass (The Flip):**

- If a cell is 'S', turn it back to 'O' (it was safe).
- If a cell is 'O', turn it to 'X' (it was surrounded).
- If a cell is 'X', leave it alone.

ASCII Visualization of the Process   Step 1: The Initial State An 'O' at (1,1) is surrounded. The 'O's at (3,1) and (3,2) are connected to the bottom edge.

```
X X X X
X O O X
X X X X
X O O X  <-- These touch the border
```

Step 2: Border DFS (Marking Survivors as 'S') We scan the edges. We find 'O' at the bottom. We mark it and its neighbor.

```
X X X X
X O O X
X X X X
X S S X  <-- Marked as Safe
```

Step 3: The Final Sweep Everything that remained 'O' becomes 'X'. 'S' becomes 'O'.

```
X X X X
X X X X  <-- (1,1) and (1,2) flipped to X
X X X X
X O O X  <-- (3,1) and (3,2) restored to O
```

---

3. Complexity Analysis

A senior engineer explains complexity by showing how many times we touch each "node" or "cell."

Time Complexity (TC)

```
Grid Dimensions: M rows, N columns
Total Cells: M * N

1. Border Scan: 2*M + 2*N touches.
2. DFS Traversal: In the worst case (all 'O's),
   we visit each cell exactly once.
3. Final Grid Sweep: Exactly M * N touches.

Total Operations: (M * N) + (M * N)
Simplified TC: O(M * N)
```

Space Complexity (SC)

```
1. In-place modification: We use the grid itself
   to store 'S', so no extra grid is needed.
2. Recursion Stack (DFS): In the worst case (a long
   snake-like 'O' chain), the stack grows to
   the number of cells.

Total SC: O(M * N)
```

---

4. Solution Code

Python Implementation

```python
class Solution:
    def solve(self, board: list[list[str]]) -> None:
        if not board or not board[0]:
            return
```

```python
        rows, cols = len(board), len(board[0])

        # This function traverses connected 'O's starting from
        # a border cell and marks them as 'S' (Safe)
        def mark_as_safe(r, c):
            if r < 0 or r >= rows or c < 0 or c >= cols or board[r][c] != 'O':
                return
            board[r][c] = 'S'
            mark_as_safe(r + 1, c)
            mark_as_safe(r - 1, c)
            mark_as_safe(r, c + 1)
            mark_as_safe(r, c - 1)

        # 1. Check first and last columns
        for r in range(rows):
            mark_as_safe(r, 0)
            mark_as_safe(r, cols - 1)

        # 2. Check first and last rows
        for c in range(cols):
            mark_as_safe(0, c)
            mark_as_safe(rows - 1, c)

        # 3. Final transformation
        for r in range(rows):
            for c in range(cols):
                if board[r][c] == 'O':
                    board[r][c] = 'X' # Was surrounded
                elif board[r][c] == 'S':
                    board[r][c] = 'O' # Restore safe cells
```

JavaScript Implementation

```javascript
var solve = function(board) {
    if (!board.length) return;

    const rows = board.length;
    const cols = board[0].length;

    /**
     * Standard Depth First Search to flood-fill
     * connected components starting from the edge.
     */
    const dfs = (r, c) => {
        if (r < 0 || r >= rows || c < 0 || c >= cols || board[r][c] !== 'O') {
            return;
```

```javascript
        }
        board[r][c] = 'S';
        dfs(r + 1, c);
        dfs(r - 1, c);
        dfs(r, c + 1);
        dfs(r, c - 1);
    };

    // Traverse borders to find "Safe" zones
    for (let r = 0; r < rows; r++) {
        dfs(r, 0);
        dfs(r, cols - 1);
    }
    for (let c = 0; c < cols; c++) {
        dfs(0, c);
        dfs(rows - 1, c);
    }

    // Final pass to flip O to X and restore S to O
    for (let r = 0; r < rows; r++) {
        for (let c = 0; c < cols; c++) {
            if (board[r][c] === 'O') board[r][c] = 'X';
            else if (board[r][c] === 'S') board[r][c] = 'O';
        }
    }
};
```

Technical Terms & Context

- **DFS (Depth First Search):** An algorithm for traversing a tree or graph. In this problem, it treats the 2D grid as a graph where each cell is a node connected to its neighbors.
- **Flood Fill:** The same logic used by the "Paint Bucket" tool in Photoshop. It starts at a point and "colors" all connected pixels of the same type.
- **In-Place Algorithm:** We modified the input `board` directly rather than creating a copy. This is a hallmark of L5+ engineering to save memory.

Real-World Interview Variations

1. **Bloomberg: "Islands on the Edge"**

- *Question:* Given a map of islands (1s) and water (0s), count only the islands that are physically touching the border of the map.

- *Solution:* Similar to the above—run DFS only from border '1's. Instead of flipping 'O's, you increment a counter for each unique border-connected component found.

2. **Google/Meta: "Room Ventilation"**

- *Question:* A grid represents a building plan. 'W' is a wall, 'A' is an air-filled space. If an 'A' isn't connected to an outside window (the border), it's "stale." Return the coordinates of all stale air pockets.
- *Solution:* Use the same "Reverse Flood" logic. Mark all 'A's reachable from the border as "Fresh." Any 'A' left unmarked is "Stale."

3. **Meta: "Social Network Connectivity"**

- *Question:* Given a list of users and their friendships, identify "isolated clusters" that have no connection to a set of "verified influencers" (the border equivalent).
- *Solution:* Start a Multi-source BFS from all influencers. Any user not reached by the BFS is part of an isolated cluster.

684. Redundant Connection

An L5/L6 engineer doesn't just solve the problem; they look for the most efficient **pattern match**. For "Redundant Connection," the pattern is "Cycle Detection in an Undirected Graph."

While you could use DFS or BFS, a senior engineer will immediately reach for **Union-Find (Disjoint Set Union)** because it processes edges one by one, making it perfect for finding the *last* edge that breaks the tree structure.

---

1. Problem Explanation

You are given a graph that *started* as a tree with **n** nodes, but one extra edge was added. This extra edge created a **cycle**. Your job is to find that extra edge and remove it to turn the graph back into a tree. If there are multiple candidates, return the one that appears last in the input.

What is a Tree vs. a Cycle?

- **Tree:** All nodes are connected, and there are **no cycles**. For **n** nodes, a tree has exactly `n - 1` edges.
- **The Problem:** We have **n** nodes and **n** edges. That extra edge is the "Redundant Connection."

---

2. Solution Explanation: The Union-Find Strategy

Think of Union-Find like a social network tracker.

1. Initially, every person (node) is in their own "group."
2. When we see an edge between `A` and `B`, we check: "Are they already in the same group?"
3. **If NO:** We merge their groups (Union).
4. **If YES:** We found our culprit! If they are already in the same group and we try to add another edge between them, we just created a cycle.

Visual Walkthrough Input: `[[1,2], [2,3], [3,4], [1,4], [1,5]]`

Step 1: Initialization Each node points to itself as "Parent."

```
Nodes:    [1]  [2]  [3]  [4]  [5]
Parents:  1    2    3    4    5
```

Step 2: Process Edge [1, 2] Are 1 and 2 connected? No. Union them.

```
1 --- 2   (Group: {1,2})
Parents: [1:1, 2:1, 3:3, 4:4, 5:5]
```

Step 3: Process Edge [2, 3] Are 2 and 3 connected? (Find(2)=1, Find(3)=3). No. Union them.

```
1 --- 2 --- 3   (Group: {1,2,3})
Parents: [1:1, 2:1, 3:1, 4:4, 5:5]
```

Step 4: Process Edge [3, 4] Are 3 and 4 connected? (Find(3)=1, Find(4)=4). No. Union them.

```
1 --- 2 --- 3 --- 4 (Group: {1,2,3,4})
Parents: [1:1, 2:1, 3:1, 4:1, 5:5]
```

Step 5: Process Edge [1, 4]

- Find(1) = 1
- Find(4) = 1
- **WAIT!** Both belong to Group 1 already. Adding an edge between 1 and 4 will close the loop.
- **RESULT:** [1, 4] is the redundant connection.

```
ASCII Visualization of the Cycle:
    1-------2
    |       |
    4-------3
(Edge 1-4 completes the square/cycle)
```

---

3. Complexity Analysis

An L5 engineer optimizes for both speed and code maintainability. By using **Path Compression** (making nodes point directly to the root during searches), the operations become nearly "constant time."

Time Complexity (TC)

```
Processing n edges:
n * [Find/Union Operation]

Using Path Compression:
The operation is Alpha(n), which is the Inverse Ackermann function.
For all practical purposes in our universe, Alpha(n) < 5.

Total TC: O(n * Alpha(n))
Which effectively behaves like: O(n)
```

Space Complexity (SC)

```
We store the parent of each node in an array:
[ 1, 2, 3, 4, ... n ]
Total SC: O(n)
```

---

4. Solution Code

Python Snippet

```python
class Solution:
    def findRedundantConnection(self, edges: list[list[int]]) -> list[int]:
        # Initialize each node as its own parent
        parent = list(range(len(edges) + 1))

        # The 'Find' function with Path Compression
        # It flattens the structure so future lookups are O(1)
        def find(i):
            if parent[i] == i:
                return i
            parent[i] = find(parent[i]) # Path compression happens here
            return parent[i]

        # The 'Union' function
        # Returns False if nodes are already in the same set (cycle detected)
        def union(i, j):
            root_i = find(i)
            root_j = find(j)
```

```
        if root_i != root_j:
            parent[root_i] = root_j
            return True
        return False

    for u, v in edges:
        if not union(u, v):
            return [u, v]
```

JavaScript Snippet

```javascript
var findRedundantConnection = function(edges) {
    const parent = Array.from({ length: edges.length + 1 }, (_, i) => i);

    // Recursively finds the root representative of a node
    // While doing so, it updates parents to point to the root (Path Compression)
    function find(i) {
        if (parent[i] === i) return i;
        parent[i] = find(parent[i]);
        return parent[i];
    }

    for (const [u, v] of edges) {
        let rootU = find(u);
        let rootV = find(v);

        // If roots are the same, u and v are already connected
        if (rootU === rootV) {
            return [u, v];
        }

        // Otherwise, perform the 'Union' by pointing one root to the other
        parent[rootU] = rootV;
    }
};
```

Terminology Deep Dive

Union-Find (DSU): A data structure that keeps track of elements split into several non-overlapping sets.

- **Why it helps:** It's the fastest way to check "connectivity" in a graph.
- **Application:** In this problem, we use it to see if adding an edge connects two nodes that were already reachable via another path.

Real-World / Interview Variations

1. **Google: "Cloud Network Stability"**

   - **Scenario:** You have a series of data centers and fiber connections. One connection is redundant and could cause a broadcast storm (cycle). Find it.
   - **Solution:** Identical to the above. Use DSU to process connection logs in order.

2. **Meta: "Friend Recommendations / Minimal Circles"**

   - **Scenario:** Given a list of friendships, determine if a new friendship creates a "closed loop" of friends that didn't exist before.
   - **Solution:** DSU is used, but often with an added "Rank" or "Size" array to ensure the smaller group always merges into the larger group (Union by Rank), keeping the tree height minimal.

3. **Bloomberg: "Currency Arbitrage Detection (Simplified)"**

   - **Scenario:** You have exchange pairs. If you can trade through a sequence and end back at your original currency, you've found a cycle.
   - **Solution:** While real arbitrage uses Bellman-Ford (for weights), a simplified "detect any cycle" version in a network uses Union-Find for undirected pairs or DFS for directed pairs.

Number of islands II

A Google L5 (Senior) or L6 (Staff) engineer doesn't just look for a "working" solution; they look for the most **extensible, robust, and performant** one. For "Number of Islands II," they immediately recognize this as a **dynamic connectivity problem**.

While "Number of Islands I" is usually solved with a simple BFS/DFS on a static grid, "Number of Islands II" gives you the land one piece at a time. Rerunning a DFS every time a new piece of land is added would be disastrously slow.

---

1. Problem Explanation

Imagine a vast ocean represented by a grid of `m x n` cells, all initially filled with water. You are given a list of operations. Each operation specifies a coordinate `(r, c)` where a new piece of land is "dropped" into the ocean.

Your goal is to return an array of integers representing the **total number of distinct islands** after each operation.

The Challenge

- **Dynamic:** The grid changes over time.
- **Merging:** Adding one piece of land might connect two previously separate islands into one.
- **Redundancy:** If you add land on a spot that is already land, nothing changes.

ASCII Visualization: The Merging Process

Initial State (3x3 Grid):

```
[ ][ ][ ]  Islands: 0
[ ][ ][ ]
[ ][ ][ ]
```

Operation 1: Add Land at (0, 0)

```
[L][ ][ ]  Islands: 1
[ ][ ][ ]  (New island created)
[ ][ ][ ]
```

Operation 2: Add Land at (0, 2)

```
[L][ ][L]  Islands: 2
[ ][ ][ ]  (Another new island, separate from the first)
[ ][ ][ ]
```

Operation 3: Add Land at (0, 1) -> **THE MERGE**

```
[L][L][L]  Islands: 1
[ ][ ][ ]  (This piece connected the island at (0,0) and (0,2))
[ ][ ][ ]
```

---

2. Solution Explanation: Union-Find (Disjoint Set Union)

An L5/L6 engineer would immediately reach for **Union-Find (DSU)** with **Path Compression** and **Union by Rank/Size**.

The Core Logic

1. **Initialize:** Treat every cell in the grid as its own potential set (or "parent").
2. **Add Land:** When land is added at (r, c), increment the global `island_count`.
3. **Check Neighbors:** Look at the 4 neighbors (Up, Down, Left, Right).
4. **Union:** If a neighbor is already land, use the `union` operation. If they belong to different sets, merge them and **decrement** the global `island_count`.

Non-Trivial Concept: Path Compression & Union by Rank

Without these, Union-Find can become a slow "chain."

- **Path Compression:** Makes the tree flat by pointing every node directly to the root during a `find` operation.
- **Union by Rank:** Always attaches the smaller tree under the root of the larger tree to keep the structure balanced.

---

3. Complexity Analysis

An L5 engineer provides a derivation that accounts for the **Inverse Ackermann Function**, denoted as (N). This function grows so slowly that for all practical purposes (even the number of atoms in the universe), it is less than 5.

Time Complexity (TC) Derivation

```
Operations: K (number of positions added)
Grid Size: N (m * n)

1. Initialization: O(N) to set up the parent array.
2. Processing K positions:
   - Check if already land: O(1)
   - Find operation (with Path Compression): O( (N))
   - Union operation (with Rank): O( (N))
   - Checking 4 neighbors: 4 * O( (N))

Total TC: O(N + K *  (N))
Technically, since  (N) is near-constant, this is nearly linear.
```

Space Complexity (SC) Derivation

```
1. Parent Array: O(N) to store the representative of each cell.
2. Rank Array:   O(N) to store the height of trees.
3. Result List:  O(K) to store the answer for each step.

Total SC: O(N + K)
```

---

4. Solution Code

Python Implementation

```python
class UnionFind:
    def __init__(self, n):
        # Initially, no land exists, so we mark parents as -1
```

```python
        self.parent = [-1] * n
        self.rank = [0] * n
        self.count = 0

    def add_land(self, i):
        # If land already exists here, don't re-add
        if self.parent[i] != -1: return
        self.parent[i] = i
        self.count += 1

    def find(self, i):
        # Path Compression: Point node directly to the root
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        root_i = self.find(i)
        root_j = self.find(j)
        if root_i != root_j:
            # Union by Rank: Attach smaller tree to larger tree
            if self.rank[root_i] > self.rank[root_j]:
                self.parent[root_j] = root_i
            elif self.rank[root_i] < self.rank[root_j]:
                self.parent[root_i] = root_j
            else:
                self.parent[root_j] = root_i
                self.rank[root_i] += 1
            self.count -= 1

def numIslands2(m, n, positions):
    uf = UnionFind(m * n)
    ans = []
    # Set to keep track of processed land to avoid duplicates
    seen = set()

    for r, c in positions:
        if (r, c) in seen:
            ans.append(uf.count)
            continue

        seen.add((r, c))
        idx = r * n + c
        uf.add_land(idx)

        # Check 4-directional neighbors
```

```python
        for dr, dc in [(0,1), (0,-1), (1,0), (-1,0)]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < m and 0 <= nc < n and (nr, nc) in seen:
                uf.union(idx, nr * n + nc)

        ans.append(uf.count)
    return ans
```

JavaScript Implementation

```javascript
/**
 * DSU Class to handle island connectivity efficiently.
 * Uses path compression and union by rank.
 */
class UnionFind {
    constructor(size) {
        this.parent = new Array(size).fill(-1);
        this.rank = new Array(size).fill(0);
        this.count = 0;
    }

    addLand(i) {
        if (this.parent[i] !== -1) return;
        this.parent[i] = i;
        this.count++;
    }

    find(i) {
        if (this.parent[i] === i) return i;
        // Recursively find the root and compress the path
        return this.parent[i] = this.find(this.parent[i]);
    }

    union(i, j) {
        let rootI = this.find(i);
        let rootJ = this.find(j);
        if (rootI !== rootJ) {
            if (this.rank[rootI] < this.rank[rootJ]) {
                this.parent[rootI] = rootJ;
            } else if (this.rank[rootI] > this.rank[rootJ]) {
                this.parent[rootJ] = rootI;
            } else {
                this.parent[rootI] = rootJ;
                this.rank[rootJ]++;
            }
            this.count--;
```

```
        }
    }
}
```

Note 1: Terms and Techniques

- **Union-Find (DSU):** A data structure that keeps track of elements partitioned into a number of disjoint (non-overlapping) sets. It is the gold standard for connectivity problems.
- **Path Compression:** During the `find` operation, we make every node we visit point directly to the root. This flattens the tree, ensuring future operations are nearly O(1).
- **Union by Rank:** We always attach the "shorter" tree to the "taller" one. This prevents the tree from becoming a long, inefficient line.

Note 2: Real-World Variations (Google, Meta, Bloomberg)

In top-tier interviews, they rarely ask the problem verbatim. Here is how they disguise it:

1. The "Social Network Connectivity" (Meta)

The Problem: You have a list of timestamps when users became friends. Find the earliest time at which everyone in the network is connected. **L5 Solution:** Use Union-Find. Each friendship is a `union` operation. When `uf.count == 1` (and you started with users), you've found the timestamp.

2. The "Dynamic Water Flow" (Google)

The Problem: A grid of cells where some are blocked. If we unblock cells one by one, when does water flow from the top row to the bottom row? **L5 Solution:** Use Union-Find with two "virtual nodes": a **Virtual Top** connected to all cells in the first row and a **Virtual Bottom** for the last row. When `find(VirtualTop) == find(VirtualBottom)`, the path exists.

3. The "Stock Market Correlations" (Bloomberg)

The Problem: Stocks are nodes. If two stocks move together, they share an edge. If the correlation threshold changes, groups of stocks merge or split. **L5 Solution:** This is "Number of Islands II" in a graph context. As correlation edges are added (as the threshold lowers), you use DSU to maintain sectors of correlated stocks.

743. Network Delay Time

A Google L6 (Staff Engineer) doesn't just look for a solution; they look for the **optimal, production-ready pattern** that scales. For "Network Delay Time," they immediately recognize this as a **Single-Source Shortest Path (SSSP)** problem.

Here is how an L-level engineer breaks this down.

---

1. Problem Explanation
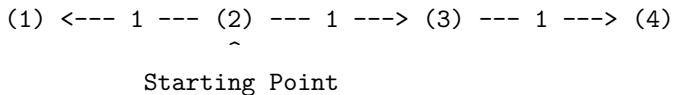
Imagine you are at a central signal tower (Node **K**). You send a "ping" to all other towers in a network of **n** nodes. Each connection between towers has a specific travel time (weight).

The Goal: Find out how long it takes for the *last* tower to receive the signal. If even one tower is unreachable, return -1.

The Visualization

Think of this like water flowing through pipes of different lengths.

```
Example: n = 4, K = 2, times = [[2,1,1],[2,3,1],[3,4,1]]

    (1) <--- 1 --- (2) --- 1 ---> (3) --- 1 ---> (4)
                    ^

              Starting Point

- At time 0: Signal is at Node 2.
- At time 1: Signal reaches Node 1 and Node 3 simultaneously.
- At time 2: Signal reaches Node 4 (via Node 3).

Result: The last node (4) got the signal at time 2. Total time = 2.
```

---

2. Solution Explanation: Dijkstra's Algorithm

For a Staff Engineer, the "go-to" for non-negative weights is **Dijkstra's Algorithm** using a **Min-Priority Queue**.

The Strategy: "Greedy Relaxation"

We maintain a list of the shortest known distances to every node. We always pick the "closest" node we haven't processed yet and see if we can use it to find a faster path to its neighbors.

ASCII Walkthrough (The "Mental Model")

Let's use: `n=4, times=[[2,1,1],[2,3,10],[3,4,1]], K=2`

Step 0: Initialization

```
Distances Map: {1: inf, 2: 0, 3: inf, 4: inf}
Priority Queue (Min-Heap): [(0, 2)]  <-- (time, node)
```

Step 1: Pop (0, 2)

- Node 2 is the start. Check neighbors: 1 and 3.
- Path to 1: $0 + 1 = 1$. ($1 <$ inf? Yes)
- Path to 3: $0 + 10 = 10$. ($10 <$ inf? Yes)

```
Distances Map: {1: 1, 2: 0, 3: 10, 4: inf}
Priority Queue: [(1, 1), (10, 3)]
```

Step 2: Pop (1, 1)

- Node 1 has no outgoing edges. Done.

```
Priority Queue: [(10, 3)]
```

Step 3: Pop (10, 3)

- Check neighbors of 3: Node 4.
- Path to 4: $10 + 1 = 11$. ($11 <$ inf? Yes)

```
Distances Map: {1: 1, 2: 0, 3: 10, 4: 11}
Priority Queue: [(11, 4)]
```

Step 4: Final Check The maximum value in our map (excluding infinity) is **11**. Since we reached all 4 nodes, the answer is 11.

---

3. Complexity Analysis

An L5/L6 engineer will explain complexity by describing the behavior of the data structures used (Adjacency List + Min-Heap).

Time Complexity (TC)

```
Legend: E = number of edges, V = number of vertices (nodes)

1. Building Adjacency List:  O(E)
2. Heap Operations:          Each edge can be added to the heap once.
                             Push/Pop from heap takes log(Size of Heap).
                             In worst case, heap size is E.
                             Total: O(E * log E)
                             (Since E < V^2, log E is approx log V)
```

```
TOTAL TC: O(E * log V)
```

Space Complexity (SC)

```
1. Adjacency List:        O(E + V) to store all nodes and connections.
2. Distance Map/Array:    O(V) to store the shortest time to each node.
3. Priority Queue:        O(E) in the worst case.
```

```
TOTAL SC: O(E + V)
```

---

4. Solution Code

Python (The "Interview Standard")

```python
import heapq
from collections import defaultdict

def networkDelayTime(times, n, k):
    # 1. Build the graph (Adjacency List)
    # We use a dict of lists: { source: [(weight, destination), ...] }
    graph = defaultdict(list)
    for u, v, w in times:
        graph[u].append((v, w))

    # 2. Priority Queue: (elapsed_time, current_node)
    # We start at time 0 at node k
    pq = [(0, k)]

    # 3. Min_Distances table to keep track of the fastest way to reach a node
    min_distances = {}

    while pq:
        time, node = heapq.heappop(pq)

        # If we already found a faster way to this node, skip it
        if node in min_distances:
            continue

        min_distances[node] = time

        # Explore neighbors
        for neighbor, weight in graph[node]:
            if neighbor not in min_distances:
                heapq.heappush(pq, (time + weight, neighbor))
```

```python
        # If we visited all n nodes, return the max time, else -1
        return max(min_distances.values()) if len(min_distances) == n else -1
```

JavaScript (The "Modern Frontend/Fullstack" Approach)

Since JS doesn't have a built-in Priority Queue (until very recently in some environments), an L5 would likely mention using a Min-Heap class.

```javascript
/**
 * Note: In a real interview, you'd implement a simple MinHeap
 * or use a library. Here we focus on the Dijkstra logic.
 */
var networkDelayTime = function(times, n, k) {
    const adj = Array.from({ length: n + 1 }, () => []);
    for (const [u, v, w] of times) {
        adj[u].push([v, w]);
    }

    const minTimes = new Array(n + 1).fill(Infinity);
    minTimes[k] = 0;

    // Using a simple array as a queue and sorting (for brevity)
    // A Staff engineer would implement a proper Binary Heap for O(log N)
    const pq = [[0, k]];

    while (pq.length > 0) {
        pq.sort((a, b) => a[0] - b[0]); // Keep it acting like a Min-Heap
        const [time, u] = pq.shift();

        if (time > minTimes[u]) continue;

        for (const [v, w] of adj[u]) {
            if (minTimes[u] + w < minTimes[v]) {
                minTimes[v] = minTimes[u] + w;
                pq.push([minTimes[v], v]);
            }
        }
    }

    let res = 0;
    for (let i = 1; i <= n; i++) {
        if (minTimes[i] === Infinity) return -1;
        res = Math.max(res, minTimes[i]);
    }
    return res;
```

```
};
```

Note 1: Terms and Techniques

- **Dijkstra's Algorithm:** A greedy algorithm for finding the shortest paths between nodes in a graph. It works by always expanding the "cheapest" known node.
- **Relaxation:** The process of checking if traveling through a new node `u` provides a shorter path to a neighbor `v` than previously known.
- **Min-Priority Queue:** A data structure that allows us to always retrieve the element with the smallest value (time) in `O(log N)` time. This is what makes the algorithm efficient.

Note 2: Real-World Variations (Google, Meta, Bloomberg)

1. **Google (Data Center Latency):** "You have a cluster of servers. If one server fails and sends a broadcast, how long until all servers are aware of the failure?"

- **Solution:** Identical to Dijkstra. You might be asked to handle "server processing time" (adding a constant weight to each node).

2. **Meta (Newsfeed Propagation):** "A user posts an update. Given the 'closeness' scores between friends, how long until the post reaches a specific circle?"

- **Solution:** This is Dijkstra, but the weights might be "1 / closeness_score." You'd also likely be asked to solve for a "Top K" friends list, which means stopping the algorithm early once K nodes are visited.

3. **Bloomberg (Stock Ticker Latency):** "Market data is sent from an exchange. Multiple relay stations exist. Which station is the 'bottleneck' for the furthest subscriber?"

- **Solution:** After running Dijkstra to find all shortest paths, identify the edge (u, v) that lies on the path to the node with the maximum distance.

787. Cheapest Flights Within K Stops

An L5/L6 engineer doesn't just look for *any* solution; they look for the **most robust** one that handles edge cases (like cycles) while maintaining optimal performance. For "Cheapest Flights Within K Stops," we are dealing with a **Shortest Path problem with a constraint** (the number of stops).

1. Problem Explanation

Imagine you are booking a flight from **Source (S)** to **Destination (D)**. You want the absolute cheapest price, but you have a rule: you cannot have more than **K intermediate stops**.

- **Nodes:** Cities.
- **Edges:** Flights with a cost (weight).
- **Constraint:** You can use at most **K+1 edges** (because K stops means K+1 flight segments).

The Trap: Standard Dijkstra's algorithm finds the shortest path based on weight. However, a "cheaper" path might have 10 stops, while your limit is K=2. If Dijkstra "visits" a node via a cheap but long path first, it might ignore a slightly more expensive path that actually satisfies the K-stop limit.

---
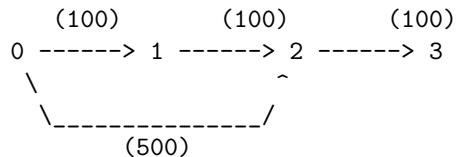
2. Solution Explanation: Modified Breadth-First Search (BFS)

While you could use Dijkstra with a modified state, a **Level-order BFS** is often more intuitive for "step-limited" problems. Each "level" in our BFS represents exactly one flight (one edge).

The Mental Model We track the minimum cost to reach each city within a certain number of flights. We use a `min_costs` array to keep track of the best price found **so far** for each city.

Visualizing the Graph & Process Let's say: `n=4, flights=[[0,1,100],[1,2,100],[2,3,100],[0,2,500` `src=0, dst=3, k=1`

```
   (100)         (100)         (100)
 0 ------> 1 ------> 2 ------> 3
  \                    ^
   _____/
        (500)
```

Step-by-Step Execution (K=1, so max 2 flights):

Step 0: Initialization `min_costs` $= [0, \text{inf}, \text{inf}, \text{inf}]$ (Cost to reach city 0 is 0) `Queue` $= [(\text{city: } 0, \text{cost: } 0)]$

Step 1: Level 1 (First Flight) From City 0, we can go to:

- City 1 (Cost 100): $100 < \text{inf}$? Yes. Update `min_costs[1]=100`. Queue: `[(1, 100)]`
- City 2 (Cost 500): $500 < \text{inf}$? Yes. Update `min_costs[2]=500`. Queue: `[(1, 100), (2, 500)]`

Step 2: Level 2 (Second Flight - This is our last allowed flight since K=1)

- From City 1: Can go to City 2 (Cost 100). Total cost $= 100 + 100 = 200$. Is $200 <$ `min_costs[2]` (which is 500)? **Yes!** Update `min_costs[2]=200`. Queue: `[(2, 200)]`
- From City 2: Can go to City 3 (Cost 100). Total cost $= 500 + 100 = 600$. Is $600 <$ `min_costs[3]` (inf)? **Yes.** Update `min_costs[3]=600`. Queue: `[(2, 200), (3, 600)]`

Result: The BFS stops here because we reached K stops. The answer is `min_costs[3]`, which is **600**. Note that even though a 3-flight path `0->1->2->3` costs only 300, it is illegal because it has 2 stops (K=1).

---

3. Time and Space Complexity Analysis

For an L5/L6 review, we analyze this based on the number of Vertices (V) and Edges (E).

Time Complexity Derivation

```
+-------------------------------------------------------------+
| Operation                  | Frequency        | Total    |
+----------------------------+------------------+----------+
| Building Adjacency List    | Once per Edge    | E        |
| BFS Traversal (Levels)     | K + 1 times      | K        |
| Processing Edges per Level | Up to E edges    | E        |
+----------------------------+------------------+----------+
| Total Complexity: O(K * E)                                |
+-------------------------------------------------------------+
```

We visit each edge at most K times because we only progress the "wave" of the BFS K+1 times.

Space Complexity Derivation

```
+-------------------------------------------------------------+
| Structure                  | Size             | Total    |
+----------------------------+------------------+----------+
| Adjacency List             | V nodes, E edges | V + E    |
| min_costs Array            | V nodes          | V        |
| Queue (BFS)                | Up to V nodes    | V        |
+----------------------------+------------------+----------+
| Total Complexity: O(V + E)                                |
+-------------------------------------------------------------+
```

---

4. Solution Code

JavaScript

```javascript
/**
 * Finds the cheapest flight within K stops using BFS.
 */
function findCheapestPrice(n, flights, src, dst, k) {
    // 1. Build adjacency list for efficient lookup
    const adj = new Map();
    for (const [u, v, w] of flights) {
        if (!adj.has(u)) adj.set(u, []);
        adj.get(u).push([v, w]);
    }

    // 2. Track minimum cost to reach each city
    let minCosts = new Array(n).fill(Infinity);
    minCosts[src] = 0;

    // 3. Queue stores [currentCity, currentTotalCost]
    let queue = [[src, 0]];
    let stops = 0;

    // Perform BFS up to K+1 levels
    while (stops <= k && queue.length > 0) {
        let size = queue.length;
        let nextQueue = [];

        // Use a temporary copy of minCosts for the current level to avoid
        // using a cost updated in the *same* level (which would violate stop count)
        let currentLevelCosts = [...minCosts];

        for (let i = 0; i < size; i++) {
            const [u, currentUWeight] = queue[i];

            if (!adj.has(u)) continue;

            for (const [v, weight] of adj.get(u)) {
                if (currentUWeight + weight < currentLevelCosts[v]) {
                    currentLevelCosts[v] = currentUWeight + weight;
                    nextQueue.push([v, currentLevelCosts[v]]);
                }
            }
        }
        minCosts = currentLevelCosts;
        queue = nextQueue;
        stops++;
    }

    return minCosts[dst] === Infinity ? -1 : minCosts[dst];
```

```
}
```

Python

```python
from collections import deque, defaultdict

def find_cheapest_price(n, flights, src, dst, k):
    # Create adjacency list: city -> list of (neighbor, price)
    adj = defaultdict(list)
    for u, v, w in flights:
        adj[u].append((v, w))

    # min_costs tracks the best price found to reach each city
    min_costs = [float('inf')] * n
    min_costs[src] = 0

    # Queue stores (current_city, current_cost)
    queue = deque([(src, 0)])
    stops = 0

    while stops <= k and queue:
        # Process nodes level by level (representing each flight)
        for _ in range(len(queue)):
            u, current_u_cost = queue.popleft()

            for v, weight in adj[u]:
                new_cost = current_u_cost + weight
                # Optimization: only explore if this path is cheaper than
                # any path found to v in PREVIOUS levels
                if new_cost < min_costs[v]:
                    min_costs[v] = new_cost
                    queue.append((v, new_cost))
        stops += 1

    return min_costs[dst] if min_costs[dst] != float('inf') else -1
```

Terminology Table

Term	Why it helps	Application here
Adjacency List	Faster than a matrix for sparse graphs.	Quickly finding which flights depart from a specific city.

Term	Why it helps	Application here
Level-order BFS	Naturally explores paths by "length" (number of edges).	Perfect for the "K-stop" constraint, as each level is exactly one stop.
Relaxation	The act of updating the cost to a node if a shorter path is found.	We "relax" the edge `u -> v` if `cost(u) + weight < cost(v)`.

Real-World Interview Variations

1. **Google (Resource Constrained Routing):**

- *Question:* "Find the path between two data centers with minimum latency, but you cannot pass through more than 5 switches due to signal degradation."
- *Solution:* Same BFS/Dijkstra logic. The "stops" are switches, and "cost" is latency.

2. **Meta (Social Connection Path):**

- *Question:* "Find if a user can reach another user within 3 degrees of separation with the 'strongest' connection (weight = frequency of interaction)."
- *Solution:* Maximize weight instead of minimizing cost, with a hard cutoff at `K=3` levels of BFS.

3. **Bloomberg (Financial Routing):**

- *Question:* "Execute a currency trade between USD and BTC using at most 2 intermediate currencies (e.g., USD -> EUR -> JPY -> BTC) to minimize exchange fees."
- *Solution:* This is the exact "Cheapest Flights" problem. Currencies are nodes, exchange rates/fees are edge weights, and intermediate currencies are stops.

778. Swim in Rising Water

An L5/L6 engineer at Google doesn't just look for *any* solution; they look for the most **robust** and **optimal** one while considering edge cases and alternative approaches. For "Swim in Rising Water," they would immediately recognize this as a **Shortest Path problem in a weighted grid**, where the "cost" isn't a sum, but a **maximum threshold**.

---

1. Problem Explanation

Imagine a square grid (N x N). Each cell has a "height" representing the water level at that spot. At time `t`, you can only step on a cell if the water level there is less than or equal to `t`. You start at the top-left `(0, 0)` and want to reach the bottom-right `(N-1, N-1)`.

The Goal: Find the minimum time `t` such that there exists a path from start to finish.

The "Mental Model" Visualization

Imagine the grid as a terrain of pillars. As time passes, the "sea level" rises. You are waiting for the water to rise high enough so that a continuous "pool" connects the start to the end, allowing you to swim across.

Example Grid (3x3):

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

t=0: You are at (0,0). Height is 0. You are fine.
t=1: Water rises to 1. You can move to (1,0) because its height is 1.
t=2: Water rises to 2. You could have moved to (0,1), but (1,0) was already open.
t=3: Water rises to 3. Now (1,1) is accessible. You reached the end!
Result: 3
```

---

2. Solution Explanation

An L5+ engineer would likely propose **Modified Dijkstra's Algorithm**. While you could use Binary Search + BFS, Dijkstra is more elegant for finding the "minimum of maximums" (Minimax path).

The Logic

Instead of minimizing the *sum* of weights (like standard Dijkstra), we minimize the *maximum* height encountered so far.

1. **Priority Queue (Min-Heap):** Store (`max_height_encountered, r, c`).
2. **Greedy Choice:** Always expand the cell that has the lowest "entry cost" (the height of the cell).
3. **State Tracking:** Keep a `visited` set to avoid infinite loops.
4. **The "Ah-ha!" Moment:** When you move from cell A to cell B, the time required to be at B is `max(time_at_A, height_at_B)`.

Step-by-Step Visualization (4x4 Grid)

Grid:

```
[ 0,  1,  2, 10 ]
[ 9, 11, 12, 13 ]
[ 8,  7,  6,  5 ]
[15, 14, 16,  4 ]
```

Trace:

1. Start at (0,0) height 0. PQ: [(0, 0, 0)]
2. Pop (0,0,0). Neighbors: (0,1) height 1, (1,0) height 9. PQ: [(1, 0, 1), (9, 1, 0)]
3. Pop (1, 0, 1). Neighbor (0,2) height 2 is lowest. PQ: [(2, 0, 2), (9, 1, 0)]
4. Pop (2, 0, 2). Neighbor (0,3) height 10. PQ: [(9, 1, 0), (10, 0, 3)]
5. Pop (9, 1, 0). Neighbor (2,0) height 8. Wait! Even though we are at "Time 9", we move to a cell with height 8. The time remains 9. PQ: [(9, 2, 0), (10, 0, 3)]
6. Pop (9, 2, 0). Neighbors (2,1) height 7. Time stays 9. … eventually, the path will snake through the lower numbers.

---

3. Complexity Analysis

Time Complexity (TC)

We visit each of the N * N cells once. For each cell, we perform heap operations (push/pop) which take logarithmic time relative to the number of elements in the heap.

```
Total Cells (V) = N * N
Edges per cell  = 4 (Up, Down, Left, Right)
Dijkstra TC     = E * log(V)
                = (4 * N * N) * log(N * N)
                = 4 * N^2 * 2 * log(N)
Simplified      = O(N^2 * log(N))
```

Space Complexity (SC)

We need to store the visited status of each cell and the Priority Queue can grow to the size of the grid in the worst case.

```
Visited Set     = N * N
Priority Queue  = N * N
Simplified      = O(N^2)
```

4. Solution Code

Python Snippet

```python
import heapq

def swimInWater(grid):
    n = len(grid)
    # Min-heap stores: (max_height_so_far, row, col)
    # We start at (0,0) with height grid[0][0]
    pq = [(grid[0][0], 0, 0)]
    visited = {(0, 0)}

    while pq:
        # Get the cell with the lowest maximum height encountered so far
        max_h, r, c = heapq.heappop(pq)

        # If we reached the target, this max_h is our answer
        if r == n - 1 and c == n - 1:
            return max_h

        for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            nr, nc = r + dr, c + dc

            # Boundary check and visited check
            if 0 <= nr < n and 0 <= nc < n and (nr, nc) not in visited:
                visited.add((nr, nc))
                # The 'time' to reach the next cell is the maximum of
                # the time it took to reach the current cell AND the
                # height of the next cell.
                new_max = max(max_h, grid[nr][nc])
                heapq.heappush(pq, (new_max, nr, nc))
```

JavaScript Snippet

```javascript
/**
 * Uses a Min-Priority Queue to implement Dijkstra's for the Minimax path.
 * Note: JS doesn't have a built-in Priority Queue, so we assume a
 * standard MinPriorityQueue implementation is available.
 */
var swimInWater = function(grid) {
    const n = grid.length;
    const pq = new MinPriorityQueue({ priority: x => x[0] });
    const visited = new Array(n).fill(0).map(() => new Array(n).fill(false));
```

```
    pq.enqueue([grid[0][0], 0, 0]);
    visited[0][0] = true;

    while (!pq.isEmpty()) {
        const [maxH, r, c] = pq.dequeue().element;

        if (r === n - 1 && c === n - 1) return maxH;

        const dirs = [[0, 1], [0, -1], [1, 0], [-1, 0]];
        for (const [dr, dc] of dirs) {
            const nr = r + dr, nc = c + dc;

            if (nr >= 0 && nr < n && nc >= 0 && nc < n && !visited[nr][nc]) {
                visited[nr][nc] = true;
                // Update the bottleneck (max height) for this path
                const nextMax = Math.max(maxH, grid[nr][nc]);
                pq.enqueue([nextMax, nr, nc]);
            }
        }
    }
};
```

Technical Terms & Algorithms

- **Dijkstra's Algorithm:** Traditionally used to find the shortest path (sum of weights). Here, we modify the "relaxation" step to find the path that minimizes the **maximum** edge weight (Minimax).
- **Min-Heap (Priority Queue):** A data structure that allows us to always retrieve the smallest element in O(log N) time. It's crucial here because it ensures we are always exploring the most "promising" (lowest water level) path first.
- **Best-First Search:** This is the general category for algorithms like Dijkstra or A*. It "greedily" explores the best option available.

Real-World Interview Variations

1. Google: "Network Latency with Packet Loss"

- **Scenario:** You have a network of nodes. Each connection has a "stability" rating (0-1). You want to find a path from A to B that maximizes the *minimum* stability of any link in the path.
- **L5 Solution:** This is exactly the same problem but inverted (Max-Min instead of Min-Max). Use a **Max-Heap** and keep track of the

`min_stability` encountered.

2. **Meta: "Path with Minimum Effort"**

   - **Scenario:** You are hiking a 2D map. The effort to move between two cells is the *absolute difference* in their heights. Find a path where the *maximum* effort of any single step is minimized.
   - **L5 Solution:** Use Dijkstra. Instead of `max(max_h, grid[nr][nc])`, your "weight" for an edge is `abs(grid[r][c] - grid[nr][nc])`. You keep track of the max weight seen on the path.

3. **Bloomberg: "Reliable Message Routing"**

   - **Scenario:** Routing a message through servers where each server has a "load" (0-100). You need to find a route where the most congested server you pass through is as empty as possible.
   - **L5 Solution:** Identical to "Swim in Rising Water." The "water level" is the server "load." Use the Dijkstra approach to find the path that avoids high-load bottlenecks.

Would you like me to walk through the **Binary Search + BFS** alternative approach to see how it compares in performance?

909. Snakes and Ladders

Hello there. I'm an L6 Staff Engineer at Google. When we look at a problem like **"Snakes and Ladders" (LeetCode 909)**, we don't just see a board game. We see a **Shortest Path problem on a Directed Unweighted Graph**.

In an interview, I'm looking for your ability to model a real-world system (the game board) into a data structure (a graph) and apply the correct algorithm (Breadth-First Search).

Let's break this down with the rigor and clarity expected at the top level.

―――――――――――――

1. Problem Explanation

The Goal: You are given an `n x n` integer matrix representing a Snakes and Ladders board. You start at square `1` (bottom-left) and want to reach square `n*n` (top-left or top-right, depending on `n`).

The Rules:

1. **Movement:** From square `curr`, you can roll a 6-sided die. You can move to `curr + 1`, `curr + 2`, …, up to `curr + 6`.

2. **Snakes & Ladders:** If you land on a square that has a snake or ladder (indicated by a number `> -1`), you **must** move to that destination immediately.
3. **One Jump Only:** If a ladder takes you to a square that holds *another* ladder, you do **not** take the second one. You stay there.
4. **Objective:** Return the **least number of moves** to reach the last square. If impossible, return `-1`.

Visualizing the Board (The Tricky Part)   The board layout is "Boustrophedon" (ox-turning). It winds back and forth.

Example: A 6x6 Board (n=6, Target=36)

```
Row 0 (Top)    :  36  35  34  33  32  31  <-- Right to Left
Row 1          :  25  26  27  28  29  30  --> Left to Right
Row 2          :  24  23  22  21  20  19  <-- Right to Left
Row 3          :  13  14  15  16  17  18  --> Left to Right
Row 4          :  12  11  10   9   8   7  <-- Right to Left
Row 5 (Bottom) :   1   2   3   4   5   6  --> Left to Right
```

Key Observation: The coordinate system is the first barrier.

- Square 1 is at `board[5][0]`.
- Square 12 is at `board[4][0]`.
- Square 13 is at `board[3][0]`.

We need a helper function to convert a linear square number (like `15`) into (`row, col`) coordinates.

---

2. Solution Explanation

The Approach: Breadth-First Search (BFS)   Why BFS?

- We are looking for the **shortest path** (minimum moves).
- The "cost" of every move is the same (1 die roll = 1 move).
- DFS (Depth-First Search) would explore deep paths that might be incredibly long before finding a short one. BFS explores layer by layer: all squares reachable in 1 move, then all in 2 moves, etc.

The Algorithm

1. **Queue:** Initialize a queue with `[StartSquare, Moves]`. `Q = [[1, 0]]`.
2. **Visited Set:** Keep track of visited squares to avoid infinite loops (cycles) and redundant processing. `Seen = {1}`.
3. **Process:**

- Pop `curr` from Queue.

- If `curr == n*n`, we win! Return `Moves`.

- Try all dice rolls `i` from 1 to 6. `next = curr + i`.

- **Check Board:** Convert `next` to `(r, c)`.

- If `board[r][c]` is `-1` (normal square), our destination is `next`.

- If `board[r][c]` is `X` (snake/ladder), our destination is `X`.

- If the final destination hasn't been visited, mark it and add to Queue.

Detailed Visualization of a Move   Imagine we are at Square 2. The board has a **Ladder at 4 pointing to 14**.

```
Current State: Square 2

Possible Dice Rolls:
+1 -> Land on 3. Board[3] is -1.  Actual Destination: 3
+2 -> Land on 4. Board[4] is 14.  Actual Destination: 14 (Ladder!)
+3 -> Land on 5. Board[5] is -1.  Actual Destination: 5
...
```

Crucial Logic Check: Does the jump happen *before* or *after* checking the visited set?

- You visit the **destination**, not necessarily the square you rolled.
- If you rolled a 4, but the ladder takes you to 14, you effectively "visited" 14. You define the edge in the graph as `2 -> 14`.

Execution Flow Diagram   Let's trace a small 4x4 board where 16 is the target. Suppose `Square 3` has a ladder to `Square 10`.

```
Queue State (FIFO)

[Step 0] Start
Q: [ (1, 0) ]  <-- At square 1, 0 moves
Visited: {1}

[Step 1] Expand Node 1
Possible rolls: 2, 3, 4, 5, 6, 7
- 2 is normal. Push (2, 1)
- 3 has LADDER to 10. Push (10, 1)  <-- SHORTCUT!
- 4 is normal. Push (4, 1)
...
Q: [ (2, 1), (10, 1), (4, 1), ... ]
Visited: {1, 2, 10, 4, ...}

[Step 2] Expand Node 2
... (process neighbors of 2)
```

```
[Step 3] Expand Node 10 (The Ladder Destination)
Possible rolls from 10: 11, 12, 13, 14, 15, 16
- ...
- 16 is TARGET!
Return (Moves + 1) which is 2.
```

Result: Path 1 -> (ladder) -> 10 -> 16. Total moves: 2.

---

3. Time and Space Complexity Analysis

This is an L5/L6 level analysis. We don't just say "O(N)", we visualize the consumption.

Time Complexity: O(N*N)   We view the board as a Graph.

- **Nodes (V):** Total squares = .
- **Edges (E):** Each square has at most 6 outgoing edges (die rolls).

BFS visits each node at most once.

```
Visualizing Operations:

Matrix Size (N x N)
+------------------+
|  1  |  2  |  3  |  ... Each cell is processed ONCE.
+------------------+
|  4  |  5  |  6  |  ... For each cell, we loop 6 times (dice).
+------------------+
| ... | ... | N*N |
+------------------+


Total Operations ~ (N*N) nodes * 6 edges
Since 6 is a constant, it drops out.


Total Time = O(N * N)
```

Space Complexity: O(N*N)   We need to store the board and the BFS structures.

```
Memory Usage Visualization:

1. Visited Array/Set
   [ True, False, True, ... ]  <-- Size N*N

2. Queue (Worst Case)
   [ 4, 5, 6, 7, 8, ... ]        <-- Can hold up to O(N*N) nodes in worst case
                                     (e.g., all nodes at specific depth)
```

186

```
Total Space = O(N * N)
```

---

4. Solution Code

Here is how I would write this in production. Clean, separated concerns, and self-documenting.

Python Solution

```python
from collections import deque

class Solution:
    def snakesAndLadders(self, board: list[list[int]]) -> int:
        n = len(board)
        target = n * n

        # ------------------------------------------------------------
        # Helper: Coordinate Converter
        # Maps a linear square number (1 to n^2) to (row, col)
        # ------------------------------------------------------------
        def get_coordinates(square):
            # Rows are counted from bottom (0) to top (n-1) in the problem logic
            # But matrix is top (0) to bottom (n-1).
            # Let's verify the row math:
            # Square 1 is at logical row 0 (bottom).
            # Logical row = (square - 1) // n
            # Matrix row = (n - 1) - logical_row

            r_from_bottom = (square - 1) // n
            r = (n - 1) - r_from_bottom

            # Columns alternate direction
            # If r_from_bottom is even (0, 2..), direction is Left -> Right
            # If r_from_bottom is odd (1, 3..), direction is Right -> Left
            c = (square - 1) % n
            if r_from_bottom % 2 == 1:
                c = (n - 1) - c

            return r, c

        # ------------------------------------------------------------
        # BFS Initialization
        # Queue stores tuples: (current_square, move_count)
        # ------------------------------------------------------------
```

```python
        queue = deque([(1, 0)])
        visited = {1}

        while queue:
            curr_square, moves = queue.popleft()

            # Try all 6 dice outcomes
            for i in range(1, 7):
                next_square = curr_square + i

                # Boundary check
                if next_square > target:
                    break

                # Check for Snake or Ladder at the landing spot
                r, c = get_coordinates(next_square)
                destination = next_square

                if board[r][c] != -1:
                    destination = board[r][c]

                # Check if we reached the end
                if destination == target:
                    return moves + 1

                # If not visited, add to queue
                if destination not in visited:
                    visited.add(destination)
                    queue.append((destination, moves + 1))

        return -1
```

JavaScript Solution

```javascript
/**
 * @param {number[][]} board
 * @return {number}
 */
var snakesAndLadders = function(board) {
    const n = board.length;
    const target = n * n;

    // Helper to get row and col from square number
    const getPos = (square) => {
        const rFromBottom = Math.floor((square - 1) / n);
        // Matrix row index
```

```javascript
        const r = (n - 1) - rFromBottom;

        let c = (square - 1) % n;
        // If row from bottom is odd, column is reversed (n-1 ... 0)
        if (rFromBottom % 2 === 1) {
            c = (n - 1) - c;
        }
        return [r, c];
    };

    // BFS Queue: [square, moves]
    const queue = [[1, 0]];
    const visited = new Set();
    visited.add(1);

    while (queue.length > 0) {
        // Shift is O(n) in JS usually, but for BFS queue it's acceptable here
        // or use a proper Queue implementation for O(1)
        const [curr, moves] = queue.shift();

        for (let i = 1; i <= 6; i++) {
            let nextSquare = curr + i;

            if (nextSquare > target) break;

            const [r, c] = getPos(nextSquare);

            // Resolve Snake/Ladder
            let destination = nextSquare;
            if (board[r][c] !== -1) {
                destination = board[r][c];
            }

            if (destination === target) return moves + 1;

            if (!visited.has(destination)) {
                visited.add(destination);
                queue.push([destination, moves + 1]);
            }
        }
    }

    return -1;
};
```

Note 1: Concepts Used

Breadth-First Search (BFS): This is the "Gold Standard" algorithm for finding the shortest path in an unweighted graph. It works by expanding uniformly in concentric circles from the start. In this problem, it guarantees that the first time we reach the square `n*n`, we have used the minimal number of dice rolls.

Note 2: Real World & Interview Variations

At companies like Google, Meta, and Bloomberg, we rarely ask the problem exactly as it appears on LeetCode. We add constraints to test your adaptability.

1. The "Weighted Dice" / "Cost to Move" Variation   The Twist: Moving to the next square costs `$1`, but using a ladder costs `$5` (toll fee), and using a snake gives you `$2` back. Find the path with the minimum *cost*, not minimum moves. **The Solution:**

- This is no longer a standard BFS because edges have different weights.
- **Technique:** Use **Dijkstra's Algorithm**. Use a Priority Queue (Min-Heap) instead of a standard Queue. Order the queue by current accumulated cost.

2. The "Must Avoid Snakes" Variation   The Twist: You must reach the end, but you are absolutely not allowed to touch a snake. **The Solution:**

- Inside the loop where we calculate `destination`, add a check: `if (is_snake(destination)) continue;`.
- This prunes the graph, effectively removing edges that lead to snakes.

3. The "K-Ladders Only" Variation (Meta specific)   The Twist: You can use at most `K` ladders in your entire path. **The Solution:**

- This requires **State Space Expansion**.
- Instead of `visited` storing `{square}`, it must store `{square, ladders_used}`.
- Your state in the queue becomes `(square, moves, ladders_count)`.
- You can visit the same square twice if you arrive there with a different `ladders_count`.

4. The "Teleport Maze" (Google specific)   The Twist: Instead of a 1-6 die, you have a list of available jumps (e.g., `[2, 5, -1]`). And the board is a maze where `0` is a wall. **The Solution:**

- This is the same logic as Snakes and Ladders but with an adjacency list modification.

- Instead of `for i in range(1, 7)`, you iterate through the specific jump list provided.
- Wall check: `if board[next] == WALL: continue`.

1584. Min Cost to Connect All Points

Here is a breakdown of how a Senior (L5) or Staff (L6) Engineer at Google would approach "Min Cost to Connect All Points" (LeetCode 1584).

At this level, the expectation isn't just getting *a* solution; it's about recognizing the underlying mathematical structure immediately (Graph Theory), evaluating the constraints (Dense vs. Sparse), and selecting the optimal algorithm (Prim's vs. Kruskal's) based on those constraints.

---

1. Problem Explanation

The Core Task: You are given a list of `points` on a 2D plane. You need to connect **every** point to form a single network such that:
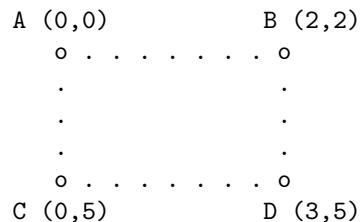
1. All points are reachable from any other point (directly or indirectly).
2. The total cost (sum of the lengths of all connections) is minimized.
3. The cost of connecting two points is the **Manhattan Distance**: `|x1 - x2| + |y1 - y2|`.

The "L6" Translation: This is a textbook definition of finding the **Minimum Spanning Tree (MST)** in a weighted undirected graph.
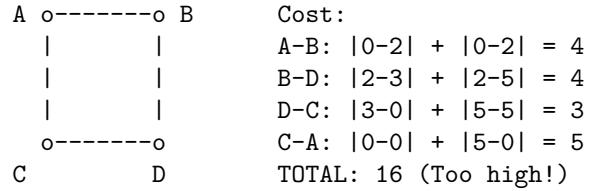
- **Nodes:** The coordinate points.
- **Edges:** Implicitly, every point can connect to every other point. This is a **Complete Graph**.
- **Weights:** Manhattan distances.

Visualizing the Goal:
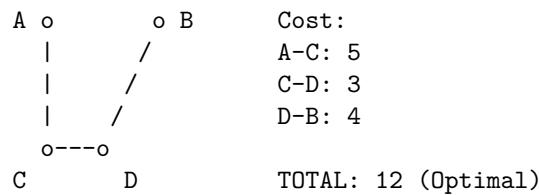
Imagine 4 servers that need to be cabled together.

```
    A (0,0)          B (2,2)
      o . . . . . . . o
      .               .
      .               .
      .               .
      o . . . . . . . o
    C (0,5)          D (3,5)
```

Option 1: A Bad Connection (Cycle & Expensive) Connecting A-B, B-D, D-C, C-A creates a loop (cycle). We don't need loops to connect everyone. Loops add unnecessary cost.

191

```
A o-------o B        Cost:
  |       |          A-B: |0-2| + |0-2| = 4
  |       |          B-D: |2-3| + |2-5| = 4
  |       |          D-C: |3-0| + |5-5| = 3
  o-------o          C-A: |0-0| + |5-0| = 5
C         D          TOTAL: 16 (Too high!)
```

Option 2: The MST (No Cycles, Minimal Cost) We want the "skeleton" that holds everything together with the least wire.

```
A o       o B        Cost:
  |      /           A-C: 5
  |     /            C-D: 3
  |    /             D-B: 4
  o---o
C       D            TOTAL: 12 (Optimal)
```

---

2. Solution Explanation: Prim's Algorithm

Why Prim's Algorithm? (The L5 Insight) A Junior engineer might pick Kruskal's Algorithm (Union-Find) by default. An L5/L6 engineer notices the constraints:

- We have `N` points.
- Every point connects to every other point.
- Total Edges = `N * (N-1) / 2`. This is a **Dense Graph**.

Kruskal's requires sorting all edges. Sorting `N^2` edges takes `O(N^2 log(N))`. **Prim's** (without a heap, optimized for dense graphs) takes `O(N^2)`.

For a dense graph, `O(N^2)` is strictly better than `O(N^2 log N)`. Therefore, we use **Prim's Algorithm**.

The Algorithm Visualization:

Imagine an infection spreading.

1. **Start:** Pick an arbitrary point (say, Point A) to be "infected" (added to our MST).
2. **Scan:** Look at *all* uninfected points. Calculate the cost to connect them to the *nearest* infected point.
3. **Expand:** Pick the uninfected point with the **smallest** cost. Add it to the MST.
4. **Repeat:** Update our distance list (because the new point might offer a cheaper connection to remaining uninfected points) and repeat until everyone is infected.

Step-by-Step Execution:

**Points: A(0,0), B(2,2), C(3,10), D(5,2), E(7,0) Current MST Set:
{ } Distances to MST:** Initialize everyone to Infinity ∞, except start node A
0.
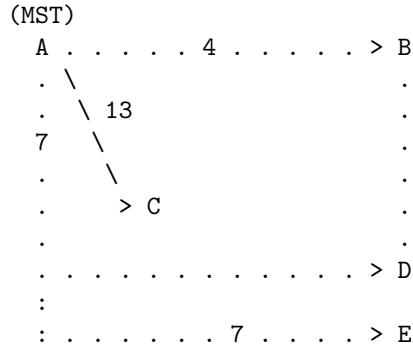
```
Node:    A    B    C    D    E
Cost:    0    ∞    ∞    ∞    ∞
Status: [ ]  [ ]  [ ]  [ ]  [ ] (Unvisited)
```

Iteration 1: Pick smallest (A) Add A to MST. Update neighbors:

- Distance A -> B: |0-2| + |0-2| = 4 (4 < ∞, update B)
- Distance A -> C: |0-3| + |0-10| = 13 (13 < ∞, update C)
- Distance A -> D: |0-5| + |0-2| = 7 (7 < ∞, update D)
- Distance A -> E: |0-7| + |0-0| = 7 (7 < ∞, update E)

```
     (MST)
     A . . . . . 4 . . . . . . > B
     . \                        .
     .  \ 13                     .
     7   \                       .
     .    \                      .
     .     > C                   .
     .                           .
     . . . . . . . . . . . . > D
     :
     : . . . . . . 7 . . . . > E
```
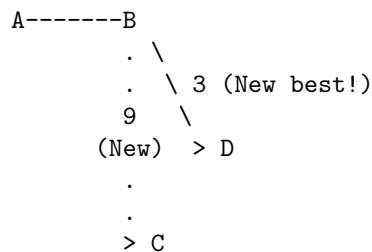
```
Node:    A    B    C    D    E
Cost:    0    4    13   7    7     <-- '4' is smallest
Status: [X]  [ ]  [ ]  [ ]  [ ]
```

Iteration 2: Pick smallest (B) Add B to MST. Cost added: 4. Now, we
check if **B** offers a shortcut to C, D, or E.

- B -> C: |2-3| + |2-10| = 9. (Is 9 < current 13? YES. Update C).
- B -> D: |2-5| + |2-2| = 3. (Is 3 < current 7? YES. Update D).
- B -> E: |2-7| + |2-0| = 7. (Is 7 < current 7? NO. Keep E).

```
     A-------B
            . \
            .  \ 3 (New best!)
            9   \
          (New)  > D
            .
            .
            > C
```
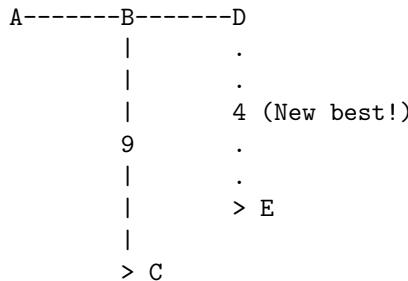
```
Node:    A    B    C    D    E
Cost:    0    4    9    3    7     <-- '3' is smallest
```

193

```
Status: [X]   [X]   [ ]   [ ]   [ ]
```

Iteration 3: Pick smallest (D) Add D to MST. Cost added: 3. Check if **D** is a shortcut to C or E.
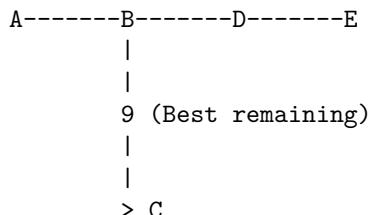
- D -> C: |5-3| + |2-10| = 10. (Is 10 < current 9? NO. Keep 9 via B).
- D -> E: |5-7| + |2-0| = 4. (Is 4 < current 7? YES. Update E).

```
        A-------B-------D
                |        .
                |        .
                |        4 (New best!)
                9        .
                |        .
                |        > E
                |
                > C
```

```
Node:    A     B     C     D     E
Cost:    0     4     9     3     4     <-- '4' is smallest
Status: [X]   [X]   [ ]   [X]   [ ]
```

**Iteration 4: Pick smallest (E)** Add E to MST. Cost added: 4. Check if E is a shortcut to C.

- E -> C: |7-3| + |0-10| = 14. (Is 14 < 9? NO).

```
        A-------B-------D-------E
                |
                |
                9 (Best remaining)
                |
                |
                > C
```

```
Node:    A     B     C     D     E
Cost:    0     4     9     3     4
Status: [X]   [X]   [ ]   [X]   [X]
```

Iteration 5: Pick smallest (C) Add C. Cost added: 9. All nodes visited.
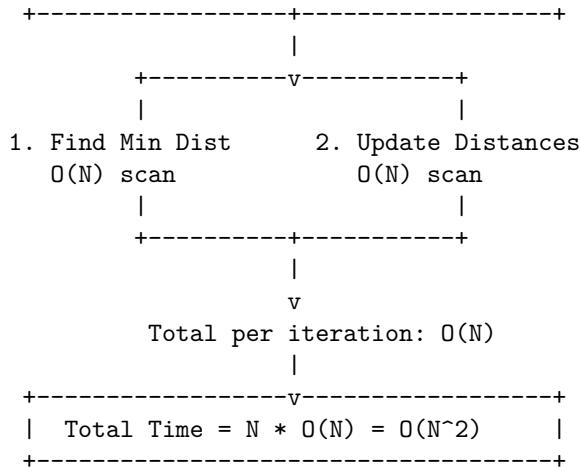
Total Cost: 4 (A-B) + 3 (B-D) + 4 (D-E) + 9 (B-C) = **20**.

---

3. Time and Space Complexity Analysis

Time Complexity: O(N^2) Where N is the number of points.

```
    +-----------------------------------+
    |  Outer Loop: Iterate N times      |   <- We must add N points to MST
```

```
+----------------+----------------+
                 |
     +----------v----------+
     |                     |
 1. Find Min Dist    2. Update Distances
    O(N) scan            O(N) scan
     |                     |
     +----------+----------+
                |
                v
        Total per iteration: O(N)
                |
 +--------------v----------------+
 |  Total Time = N * O(N) = O(N^2)    |
 +-------------------------------+
```

Why this is great: Since `N <= 1000` (from problem constraints), `N^2` is `1,000,000` operations, which executes in milliseconds.

Space Complexity: O(N)

```
Memory Usage Breakdown:

[ Min-Dist Array  ]  -> Stores 1 integer per point  -> size N
[ Visited Boolean ]  -> Stores 1 boolean per point  -> size N
[ Points Array    ]  -> Input storage               -> size N

Total Space: O(N)
```

Note: We do NOT build an adjacency matrix (which would be `O(N^2)` space). We calculate distances on the fly. This is a crucial "L6 optimization" to save memory.

---

4. Solution Code

Python Solution (Optimized Prim's)

```python
import sys

def minCostConnectPoints(points):
    """
    Calculates the minimum cost to connect all points using Prim's Algorithm.

    Args:
    points: List[List[int]] -> [[x1, y1], [x2, y2], ...]

    Returns:
```

```python
    int -> Minimum Manhattan distance cost
    """
    n = len(points)

    # "min_dist" stores the shortest distance from the current MST
    # to every other point. Initialize with Infinity.
    min_dist = [float('inf')] * n

    # "in_mst" tracks if a point has been included in the tree.
    in_mst = [False] * n

    # We arbitrarily start at point 0. Distance to itself is 0.
    min_dist[0] = 0

    total_cost = 0
    edges_count = 0

    # We need to include exactly n points in the MST
    for _ in range(n):

        # Step 1: Find the unvisited node with the smallest distance to the MST.
        # This corresponds to the O(N) scan in the diagram above.
        curr_node = -1
        curr_min_edge = float('inf')

        for i in range(n):
            if not in_mst[i] and min_dist[i] < curr_min_edge:
                curr_min_edge = min_dist[i]
                curr_node = i

        # If the graph was disconnected, we might fail to find a node (not possible here).
        if curr_node == -1:
            break

        # Step 2: Add this node to the MST
        in_mst[curr_node] = True
        total_cost += curr_min_edge
        edges_count += 1

        # Step 3: Update distances to all other unvisited neighbors.
        # We check if connecting from 'curr_node' is cheaper than whatever
        # connection they currently have.
        for next_node in range(n):
            if not in_mst[next_node]:
                dist = abs(points[curr_node][0] - points[next_node][0]) + \
                        abs(points[curr_node][1] - points[next_node][1])
```

```python
                    if dist < min_dist[next_node]:
                        min_dist[next_node] = dist

    return total_cost
```

JavaScript Solution

```javascript
/**
 * @param {number[][]} points
 * @return {number}
 */
var minCostConnectPoints = function(points) {
    const n = points.length;

    // minDist[i] holds the shortest distance from the MST to point i.
    // Initialize with Infinity.
    const minDist = new Array(n).fill(Infinity);

    // tracks visited status
    const inMST = new Array(n).fill(false);

    // Start with node 0
    minDist[0] = 0;

    let totalCost = 0;

    // We iterate n times to add n nodes
    for (let i = 0; i < n; i++) {

        // 1. GREEDY SELECTION
        // Find the unvisited node with the smallest minDist value
        let currNode = -1;
        let currMinEdge = Infinity;

        for (let j = 0; j < n; j++) {
            if (!inMST[j] && minDist[j] < currMinEdge) {
                currMinEdge = minDist[j];
                currNode = j;
            }
        }

        // 2. COMMIT
        // Mark as part of MST and add cost
        inMST[currNode] = true;
        totalCost += currMinEdge;
```

197

```
        // 3. RELAXATION (Update neighbors)
        // Check if currNode offers a shorter path to any unvisited node
        for (let nextNode = 0; nextNode < n; nextNode++) {
            if (!inMST[nextNode]) {
                const dist = Math.abs(points[currNode][0] - points[nextNode][0]) +
                             Math.abs(points[currNode][1] - points[nextNode][1]);

                if (dist < minDist[nextNode]) {
                    minDist[nextNode] = dist;
                }
            }
        }
    }

    return totalCost;
};
```

Note 1: Terminology Clarification

MST (Minimum Spanning Tree): This is the specific subgraph we built.
It is a subset of the edges of a connected, edge-weighted undirected graph that
connects all the vertices together, without any cycles and with the minimum
possible total edge weight.

Prim's Algorithm: A "greedy" algorithm. It builds the MST one vertex at
a time, from an arbitrary starting vertex, at each step adding the cheapest
possible connection from the tree to a vertex not yet in the tree.

Manhattan Distance: Also known as "Taxicab geometry". Unlike Euclidean
distance (straight line/hypotenuse), this distance is the sum of the absolute
differences of their coordinates. It implies you can only move along a grid
(up/down/left/right), like a taxi in New York City.

**Note 2: Real World & Interview Variations (Google, Meta,
Bloomberg)**

In an L5/L6 interview, the interviewer will rarely stop at "solve this Leetcode
problem." They will pivot to real-world scenarios:

1. The "Power Grid" or "Fiber Optic" Variant (Google/Meta)

- **Question:** "We need to lay fiber cables to connect 5 data centers. How-
  ever, there is already an existing cable between Center A and Center B.
  How does this change your algorithm?"

198

- **L6 Approach:** This is a classic "Pre-existing Condition" MST.
- In **Kruskal's**: Initialize the Union-Find structure by `union(A, B)` *before* processing any other edges.
- In **Prim's**: Set the distance of B (if starting at A) to `0` immediately, effectively treating them as a merged super-node.

2. **The "Limited Hubs" Variant (Bloomberg)**

- **Question:** "You have `N` offices. You can connect them with cables (cost = distance), but you are also allowed to install up to K satellite uplinks. Any office with a satellite is connected to any other office with a satellite for free (cost 0). Minimize the cable cost."
- **L6 Approach:**
- This transforms the problem into finding a "Minimum Spanning Forest" with `K` components.
- Run Kruskal's or Prim's, but stop when you have exactly `K` connected components left (instead of 1). The implicit "satellite" connects those `K` components instantly.

3. **The "Dynamic/Streaming" Variant (System Design crossover)**

- **Question:** "Points are appearing on the map in real-time. How do you maintain the MST efficiently without recalculating from scratch every second?"
- **L6 Approach:** This is hard. You cannot easily update an MST dynamically in `O(1)`.
- You would discuss using dynamic graph algorithms or treating it as a kinetic data structure problem.
- For a practical approximation (L5 answer), you might discuss re-running the algorithm only on the "local neighborhood" of the new point or using a spatial index (QuadTree) to limit the number of edges you consider.

Walls and Gates

Here is how a Senior Software Engineer (L5/L6) at Google would approach, analyze, and solve "Walls and Gates".

---

1. Problem Explanation

Imagine a dungeon map represented by a grid.

- **Gates (`0`)**: Exits where you can escape.
- **Walls (`-1`)**: Obstacles you cannot pass through.
- **Empty Rooms (`INF`)**: Places where you are standing. `INF` is a placeholder for "Infinity" (often represented as `2147483647`), meaning we don't know the distance yet.

The Goal: Update every **Empty Room** with the number of steps to the *nearest* Gate. If a room cannot reach any gate (trapped by walls), keep it as `INF`.

Visual Representation:

Let's look at a 4x4 Grid.

Legend: `G` = Gate (0) `W` = Wall (-1) `.` = Empty Room (INF)

Input Grid:

```
.  W  G  .
.  .  .  W
.  W  .  W
G  .  .  .
```

Target Output: (Replace `.` with the number of steps to the nearest `G`)

```
3  W  0  1
2  2  1  W
1  W  2  W
0  1  2  3
```

---

2. Solution Explanation: The "Multi-Source BFS"

A Junior engineer might try to start at every single Empty Room and search for a gate. That works, but it is incredibly slow because you re-calculate the same paths over and over.

The L5/L6 Insight: Instead of thinking "How do I get from Room A to *any* Gate?", invert the thinking: **"How does the Gate reach the rooms?"**

Imagine the Gates are sources of water. If we turn on the water at all Gates simultaneously, the water spreads to neighbors in 1 second, then to neighbors-of-neighbors in 2 seconds, and so on. This "ripple effect" guarantees that when the water hits a room, it has taken the shortest path.

This technique is called **Multi-Source Breadth-First Search (BFS)**.

Step-by-Step Visualization   Setup:

1. Scan the grid.
2. Find all Gates (`G`).
3. Add all Gates to a **Queue**.
4. Mark Gates as "Visited" (implicitly, since their distance is 0).

Initial State: Queue: `[(0, 2), (3, 0)]` <-- Coordinates of the two Gates
Distance: 0

```
.   W   G   .     <-- (0,2) is in Q
.   .   .   W
.   W   .   W
G   .   .   .     <-- (3,0) is in Q
```

Step 1: Process Distance 0 (The Gates) We pop the Gates from the queue and look at their neighbors (Up, Down, Left, Right). Valid neighbors (not Walls, not already visited) get a distance of `1` and are added to the Queue.

- Processing `G` at (0,2):
- Right neighbor: (0,3). Update `.` to `1`. Add to Q.
- Bottom neighbor: (1,2). Update `.` to `1`. Add to Q.
- Left/Top: Blocked or Out of bounds.
- Processing `G` at (3,0):
- Top neighbor: (2,0). Update `.` to `1`. Add to Q.
- Right neighbor: (3,1). Update `.` to `1`. Add to Q.

Grid after Step 1 (Distance 1 Ripples): Queue now has: `[(0,3), (1,2), (2,0), (3,1)]`

```
.   W   0   1    <-- New!
.   .   1   W    <-- New!
1   W   .   W    <-- New!
0   1   .   .    <-- New!
```

Step 2: Process Distance 1 Now we pop the cells we just added. These are distance `1`. Their neighbors will become distance `2`.

- Processing (0,3): Neighbors are Wall or out of bounds. Nothing happens.
- Processing (1,2): Neighbors are (1,1) and (2,2). Update them to `2`.
- Processing (2,0): Neighbors are (1,0). Update to `2`.
- Processing (3,1): Neighbors are (3,2). Update to `2`.

Grid after Step 2 (Distance 2 Ripples): Queue now has nodes with distance 2.

```
.   W   0   1
2   2   1   W    <-- New!
1   W   2   W    <-- New!
0   1   2   .    <-- New!
```

Step 3: Process Distance 2 We continue this pattern. The remaining `.` and the top left `.` get filled.

- The `.` at (0,0) is neighbor to (1,0) [which is 2]. So (0,0) becomes `3`.
- The `.` at (3,3) is neighbor to (3,2) [which is 2]. So (3,3) becomes `3`.

Final State: Queue is Empty. Algorithm finishes.

```
3   W   0   1
2   2   1   W
1   W   2   W
0   1   2   3
```

---

3. Time and Space Complexity Analysis

I will explain this using grid visual logic rather than just formulas.

Time Complexity: O(M * N)   Why? Because we touch every cell essentially once.

Imagine a grid of size M x N.

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
[ ] [ ] [ ] [ ]
```

1. **Initialization:** We iterate through the whole grid once to find the Gates. Cost: `M * N`.
2. **BFS Traversal:**

- When an empty room is added to the Queue, it is "visited".
- We never add a visited room back to the Queue (we check if `grid[r][c] == INF`).
- Therefore, every cell is pushed to the Queue **at most once**.
- For every cell, we check 4 directions (constant work).

Total Work = (Scan Grid) + (Process each cell at most once) **Total Work** 2 * (M * N) **Big O** = O(M * N)

Space Complexity: O(M * N)   Space is determined by the maximum size of our Queue.

Worst Case Scenario: Imagine a grid full of Gates except for one layer of empty rooms. Or, imagine a "diamond" shape of expansion in a massive empty grid.

At the widest point of the "ripple", the number of cells in the Queue is roughly proportional to the perimeter of the search area. In a dense grid, this can grow to be proportional to the total number of cells `M * N` (e.g., if the whole grid is Gates and we load them all at start).

Memory Usage Diagram:

```
Grid Memory:  [ . . . . . . . ]  <-- We modify the input grid in-place (O(1) extra if allowed)
                                      If we can't modify input, we need O(M*N) for result.


Queue Memory: [ (r,c), (r,c) ... ] <-- Can grow up to O(M * N) in worst case.
```

Total Space = O(M * N)

———————————————————

4. Solution Code

Python Solution

```python
from collections import deque
from typing import List

def wallsAndGates(rooms: List[List[int]]) -> None:
    """
    Modifies the rooms grid in-place to fill each empty room with the
    distance to the nearest gate using Multi-Source BFS.
    """
    if not rooms or not rooms[0]:
        return

    rows, cols = len(rooms), len(rooms[0])
    queue = deque()

    # Constants for readability
    EMPTY = 2147483647
    GATE = 0
    # Directions: Up, Down, Left, Right
    DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # 1. Initialize the Queue with all Gates
    # We scan the entire grid once to find starting points.
    for r in range(rows):
        for c in range(cols):
            if rooms[r][c] == GATE:
                queue.append((r, c))

    # 2. Process the Queue (BFS)
    while queue:
        r, c = queue.popleft()

        # Check all 4 neighbors
        for dr, dc in DIRECTIONS:
            nr, nc = r + dr, c + dc

            # 3. Validation Check
            # - Must be within grid bounds
            # - Must be an EMPTY room (this serves as our 'visited' check too!
            #   If it's not EMPTY, it's either a Wall, a Gate, or a room we
```

```python
                #    already updated with a shorter path).
                if 0 <= nr < rows and 0 <= nc < cols and rooms[nr][nc] == EMPTY:

                    # Update distance: current cell's distance + 1
                    rooms[nr][nc] = rooms[r][c] + 1

                    # Add to queue to continue the ripple
                    queue.append((nr, nc))

# No return value needed as we modify 'rooms' in-place.
```

JavaScript Solution

```javascript
/**
 * @param {number[][]} rooms
 * @return {void} Do not return anything, modify rooms in-place instead.
 */
var wallsAndGates = function(rooms) {
    if (!rooms || rooms.length === 0) return;

    const rows = rooms.length;
    const cols = rooms[0].length;

    const EMPTY = 2147483647;
    const GATE = 0;

    // Using a simple array as a queue.
    // Note for Interview: In a strictly theoretical analysis,
    // array.shift() is O(K), making this O(N*N).
    // However, for LeetCode/JS constraints, this is usually accepted.
    // A proper Queue implementation is preferred for L5+.
    const queue = [];

    // 1. Initialize Queue with Gates
    for (let r = 0; r < rows; r++) {
        for (let c = 0; c < cols; c++) {
            if (rooms[r][c] === GATE) {
                queue.push([r, c]);
            }
        }
    }

    const directions = [[-1, 0], [1, 0], [0, -1], [0, 1]];

    // 2. BFS
    while (queue.length > 0) {
```

```
        // Optimization: In JS, shifting a large array is slow.
        // We can check just the head, but we must remove it.
        const [r, c] = queue.shift();

        for (const [dr, dc] of directions) {
            const nr = r + dr;
            const nc = c + dc;

            // Check bounds and if it is an EMPTY room
            if (nr >= 0 && nr < rows &&
                nc >= 0 && nc < cols &&
                rooms[nr][nc] === EMPTY) {

                // Update distance
                rooms[nr][nc] = rooms[r][c] + 1;

                // Push to queue
                queue.push([nr, nc]);
            }
        }
    }
};
```

Note 1: New Terms & Techniques

Multi-Source BFS: Standard BFS starts from a single root node (like a tree). Multi-Source BFS initializes the Queue with *multiple* nodes at `distance 0`.

- **Why it helps:** It calculates the shortest path from *any* of the sources to the destinations in parallel. It effectively merges the search space, preventing us from running BFS `N` times.
- **Application:** Used whenever you have multiple starting points and need the shortest distance to the "closest" starting point (e.g., "Nearest Pizza Shop").

Note 2: Real World & Interview Variations

Big tech companies rarely ask "Walls and Gates" verbatim. They wrap it in a scenario.

1. The "Virus Spread" (Healthcare / Epidemiology)

- **Problem:** You have a grid representing a city. Some zones are infected (Sources), some are healthy (Empty), some are quarantined/blocked

(Walls). How many days until the whole city is infected? Or, which zones are safe after K days?

- **Solution:** This is exactly Walls and Gates. Infected zones are Gates. Healthy zones are Empty Rooms. The output is the maximum value in the grid (total time).

2. The "Nearest Essential Service" (DoorDash / Uber / Google Maps)

- **Problem:** Given a map with Customers, Restaurants, and Roadblocks. For every Customer, find the distance to the *nearest* Restaurant.
- **Solution:**
- Restaurants = Gates.
- Customers = Empty Rooms.
- Roadblocks = Walls.
- Run Multi-Source BFS from all Restaurants simultaneously.

3. The "Highest Peak" (Topographical Map / Gaming)

- **Problem:** You are given a grid where `1` represents water and `0` represents land. You need to assign heights to the land cells such that the height difference between adjacent cells is at most 1, and water has height 0. Maximize the heights.
- **Solution:** This is an inverted Walls and Gates.
- Water cells are the "Gates" (Height 0).
- Land cells are "Empty Rooms".
- You BFS from the water to calculate the distance (height) for the land.

4. "Bomb Detonation" (Gaming)

- **Problem:** You have bombs that detonate and the blast radius reduces by 1 unit every step. Given bomb locations and walls, calculate the damage intensity at every square.
- **Solution:** Multi-Source BFS starting from bombs. The "distance" is actually "damage reduction".

Strategy for all: Always identify:

1. **The Source (Start)**: Who is spreading something? (Virus, Water, Signal, Courier). Put these in the Queue first.
2. **The Medium**: Where can it travel? (Empty cells).
3. **The Blocker**: Where can it NOT travel? (Walls).

1514. Path with Maximum Probability

Here is a comprehensive breakdown of **Leetcode 1514. Path with Maximum Probability**, structured from the perspective of a Senior (L5/L6) Engineer.

An L5/L6 engineer looks past the immediate "get it working" phase. They look for **correctness at scale**, **readability**, and **mapping the problem to known stable patterns** (like Dijkstra's Algorithm) rather than inventing fragile custom logic.

---

1. Problem Explanation

Imagine you are building a communication network for a distributed system. You have several servers (nodes), and the connection between them isn't perfect. Each connection (edge) has a specific **success probability** (a number between 0 and 1).

If you send a message from Server A to Server B, it has an 80% chance of arriving. If it then goes from B to C with a 50% chance, the total probability of the message surviving the whole trip from A to C is: `0.8 * 0.5 = 0.4` (or 40%).
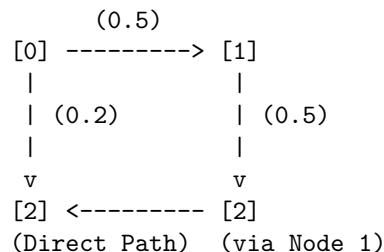
The Goal: Given a starting node (`start_node`) and an ending node (`end_node`), find the path that yields the **highest** total probability of success. If there is no path, return 0.

Visualization of the Problem Let's look at a simple network.

- **Nodes:** 0, 1, 2
- **Start:** 0
- **End:** 2

Edges & Probabilities:

- 0 -> 1: 0.5 (50% success)
- 0 -> 2: 0.2 (20% success)
- 1 -> 2: 0.5 (50% success)

```
      (0.5)
[0] ---------> [1]
 |              |
 | (0.2)        | (0.5)
 |              |
 v              v
[2] <--------- [2]
(Direct Path)  (via Node 1)
```

Path Options:

1. **Direct Path (0 -> 2):**

- Probability: **0.2**

2. **Indirect Path (0 -> 1 -> 2):**

- Math: `Probability(0->1) * Probability(1->2)`
- Calculation: `0.5 * 0.5`
- Result: **0.25**

Winner: The indirect path (0 -> 1 -> 2) is better (0.25 > 0.2).

---

2. Solution Explanation

The Intuition (Why a specific algorithm?)   A naive approach might be "Breadth-First Search" (BFS). BFS finds the path with the *fewest hops*. In the example above, BFS would choose the direct path (0 -> 2) because it's only 1 hop. However, that path has a lower probability (0.2) than the 2-hop path (0.25). **BFS fails here.**

We need an algorithm that explores paths based on their "cost" or "weight." In standard computer science, **Dijkstra's Algorithm** is the gold standard for finding the "shortest path" in a weighted graph.

Adapting Dijkstra for Probability: Standard Dijkstra finds the *minimum sum* of weights.

- Standard: `Path Cost = Weight A + Weight B` (Minimize this)
- Our Problem: `Path Prob = Prob A * Prob B` (Maximize this)

Because probabilities are essentially "costs" (where a lower probability is a "higher cost" to reliability), we can use a slightly modified Dijkstra's Algorithm.

The Strategy:

1. **Max-Heap:** Instead of a Min-Heap (used in standard Dijkstra to find the smallest distance), we use a **Max-Heap**. This allows us to always process the path with the *highest probability* found so far.
2. **Greedy Approach:** At every step, we look at the node reachable with the highest probability. We then check all its neighbors to see if we can reach *them* with a higher probability than we previously thought.

Step-by-Step Visualization   Scenario:

- Nodes: 0, 1, 2, 3
- Start: 0, End: 3
- Edges:
- 0-1: 0.5
- 0-2: 0.6
- 1-3: 0.4
- 2-1: 0.3
- 2-3: 0.8

Initial State:

- `max_prob` array (tracks best prob to reach each node): [0: 1.0, 1: 0.0, 2: 0.0, 3: 0.0]
- Priority Queue (PQ): [(1.0, Node 0)] (Format: `Probability, Node`)

```
STEP 1: Process Node 0 (Prob 1.0)
--------------------------------
Pop (1.0, 0) from PQ.

Neighbors of 0:
A. Node 1 (Edge 0.5) -> New Prob = 1.0 * 0.5 = 0.5
   Is 0.5 > current max_prob[1] (0.0)? YES. Update & Push.
B. Node 2 (Edge 0.6) -> New Prob = 1.0 * 0.6 = 0.6
   Is 0.6 > current max_prob[2] (0.0)? YES. Update & Push.


Current State:
PQ: [(0.6, 2), (0.5, 1)]  <-- Node 2 is at top (Highest Prob)
max_prob: [1.0, 0.5, 0.6, 0.0]
```

```
STEP 2: Process Node 2 (Prob 0.6)
--------------------------------
Pop (0.6, 2) from PQ.

Neighbors of 2:
A. Node 1 (Edge 0.3) -> New Prob = 0.6 * 0.3 = 0.18
   Is 0.18 > current max_prob[1] (0.5)? NO. Ignore.
B. Node 3 (Edge 0.8) -> New Prob = 0.6 * 0.8 = 0.48
   Is 0.48 > current max_prob[3] (0.0)? YES. Update & Push.

Current State:
PQ: [(0.5, 1), (0.48, 3)] <-- Node 1 is now at top
max_prob: [1.0, 0.5, 0.6, 0.48]
```

```
STEP 3: Process Node 1 (Prob 0.5)
--------------------------------
Pop (0.5, 1) from PQ.

Neighbors of 1:
A. Node 3 (Edge 0.4) -> New Prob = 0.5 * 0.4 = 0.20
   Is 0.20 > current max_prob[3] (0.48)? NO. Ignore.
   (We already found a better way to Node 3 via Node 2!)

Current State:
PQ: [(0.48, 3)]
max_prob: [1.0, 0.5, 0.6, 0.48]
```

```
STEP 4: Process Node 3 (Prob 0.48)
--------------------------------
Pop (0.48, 3) from PQ.
```

```
   Node 3 is our Target!

   Return 0.48
```

---

3. Time and Space Complexity Analysis

Time Complexity: O(E * log V)

- **V**: Number of vertices (nodes).
- **E**: Number of edges.

Why? The complexity is dominated by the Priority Queue (Heap) operations.

```
      The Heap Interaction Loop
      -------------------------


   [ Start ]
       |
       v
  +-----------+
  | Pop Node  |  <--- Takes O(log V) time (removing max element)
  +-----------+  <--- This happens at most V times (once per node)
       |
       v
  [ For each Neighbor ]
       |
       v
  +-------------------+
  | Calculate Prob    |
  | & Push if better  | <--- Push takes O(log V)
  +-------------------+ <--- This happens at most E times (once per edge)


  Total Cost derivation:
  1. We process every edge (E) in the worst case.
  2. Every edge might trigger a Heap Push.
  3. A Heap Push costs log(V).

  Calculation: E edges * log(V) cost per edge
  Final: O(E * log V)
```

Space Complexity: O(V + E)

```
   Memory Usage Breakdown
   ----------------------
```

1. The Graph (Adjacency List)
   Stores all nodes and connections.
   Size: O(V + E)

   [0] -> [(1, 0.5), (2, 0.6)]
   [1] -> [(3, 0.4)]
   ...

2. The Probability Array (max_prob)
   Stores one float per node.
   Size: O(V)

3. The Priority Queue (Heap)
   In worst case, can store all edges if every path is an improvement.
   Size: O(E) (often simplifed to O(V) in dense graphs logic, but O(E) is safer upper bou

Final: O(V + E)

---

4. Solution Code

Python Solution   Engineer's Note: Python's `heapq` module implements a
Min-Heap. To simulate a **Max-Heap** (needed to pop the *highest* probability
first), we store probabilities as negative numbers. `-0.8` is smaller than `-0.5`, so
`-0.8` pops first. We then flip the sign back to positive.

```python
import heapq
from collections import defaultdict


class Solution:
    def maxProbability(self, n: int, edges: List[List[int]], succProb: List[float], start_no
        """
        Calculates the maximum probability path using a modified Dijkstra's algorithm.

        Args:
            n: Number of nodes
            edges: List of [u, v] connections
            succProb: List of probabilities corresponding to edges
            start_node: Starting node index
            end_node: Target node index
        """

        # 1. Build the Graph (Adjacency List)
        # We use a hash map where key = node, value = list of (neighbor, probability)
        # Time to build: O(E)
        graph = defaultdict(list)
```

```python
    for i, (u, v) in enumerate(edges):
        prob = succProb[i]
        graph[u].append((v, prob))
        graph[v].append((u, prob)) # The graph is undirected

    # 2. Initialize Probability Tracker
    # Tracks the max probability found SO FAR to reach node 'i'.
    # Initialize with 0.0 because we haven't found a path yet.
    max_prob = {i: 0.0 for i in range(n)}
    max_prob[start_node] = 1.0

    # 3. Priority Queue (Max-Heap)
    # Python uses Min-Heap by default. We store (-prob, node) to simulate Max-Heap.
    # Start with the start_node at 100% (1.0) probability.
    pq = [(-1.0, start_node)]

    while pq:
        # Pop the node with the highest current probability
        curr_prob_neg, curr_node = heapq.heappop(pq)
        curr_prob = -curr_prob_neg

        # Optimization: If we reached the target, we found the best path
        # because Dijkstra guarantees the first time you pop a node,
        # it is via the shortest/best path.
        if curr_node == end_node:
            return curr_prob

        # If the popped probability is less than what we already found for this node,
        # it means this is a stale entry in the heap. Skip it.
        if curr_prob < max_prob[curr_node]:
            continue

        # Explore neighbors
        for neighbor, edge_prob in graph[curr_node]:
            # Calculate probability to reach neighbor via current node
            new_prob = curr_prob * edge_prob

            # Relaxation Step:
            # If this new path is better than any previous path to 'neighbor'
            if new_prob > max_prob[neighbor]:
                max_prob[neighbor] = new_prob
                heapq.heappush(pq, (-new_prob, neighbor))

    # If queue is empty and we never reached end_node
    return 0.0
```

JavaScript Solution Engineer's Note: JavaScript does not have a built-in Priority Queue in the standard library. In a Leetcode environment, `MinPriorityQueue` from the `datastructures-js/priority-queue` library is available. We use that here. If this were a raw environment (like a browser console), we would need to implement a Binary Heap class manually.

```javascript
/**
 * @param {number} n
 * @param {number[][]} edges
 * @param {number[]} succProb
 * @param {number} start_node
 * @param {number} end_node
 * @return {number}
 */
var maxProbability = function(n, edges, succProb, start_node, end_node) {
    // 1. Build Adjacency List
    // Using a Map for cleaner key-value storage, or an Array of Arrays
    const graph = new Array(n).fill(0).map(() => []);

    for (let i = 0; i < edges.length; i++) {
        const [u, v] = edges[i];
        const prob = succProb[i];
        graph[u].push([v, prob]);
        graph[v].push([u, prob]); // Undirected
    }

    // 2. Probability Array
    // Float64Array is slightly more performant for numbers, but standard Array works too
    const maxProbs = new Float64Array(n).fill(0.0);
    maxProbs[start_node] = 1.0;

    // 3. Priority Queue
    // We need a Max Priority Queue.
    // Leetcode environment provides MinPriorityQueue and MaxPriorityQueue classes.
    // We use MaxPriorityQueue to access the highest probability efficiently.
    const pq = new MaxPriorityQueue({ priority: x => x.prob });

    // Add start node
    pq.enqueue({ node: start_node, prob: 1.0 });

    while (!pq.isEmpty()) {
        const { node: currNode, prob: currProb } = pq.dequeue().element;

        // Early Exit: If we popped the end_node, this is the max probability
        if (currNode === end_node) {
            return currProb;
```

```
        }

        // Stale check: If we found a better way to this node already, skip
        if (currProb < maxProbs[currNode]) {
            continue;
        }

        // Iterate neighbors
        for (const [neighbor, edgeProb] of graph[currNode]) {
            const newProb = currProb * edgeProb;

            if (newProb > maxProbs[neighbor]) {
                maxProbs[neighbor] = newProb;
                pq.enqueue({ node: neighbor, prob: newProb });
            }
        }
    }

    return 0.0;
};
```

Note 1: Terminology Breakdown

Algorithm Used: Dijkstra's Algorithm (Modified)

- **Why it helps:** Dijkstra is designed to find the optimal path in a graph with non-negative weights. It guarantees that once a node is "visited" (popped from the Priority Queue), the optimal path to it has been found.

- **How it applies here:** Standard Dijkstra minimizes distances (`A + B`). We modified the "Relaxation" step (the logic that decides if a new path is better) to maximize probability (`A * B`).

- Standard Relaxation: `if new_dist < dist[neighbor]`

- Our Relaxation: `if new_prob > prob[neighbor]`

- **Monotonicity Property:** Dijkstra works here because probabilities are between 0 and 1. Multiplying by a number `<= 1` always results in a smaller (or equal) number. This behaves similarly to adding positive weights in standard Dijkstra (where distance always increases), ensuring we don't encounter "negative cycle" equivalents that would break the algorithm.

Note 2: Real World Interview Variations

Top-tier tech companies (Google, Meta, Bloomberg) rarely ask the exact Leetcode question. They wrap it in a real-world scenario.

1. **Google: "The Most Reliable Network Path"**

   - **Scenario:** You are routing data packets between data centers. Links have "packet loss rates." Find the path with the least packet loss.
   - **Mapping:** * Packet Loss Rate = 10%
   - Success Rate = 90% (0.9)
   - This is exactly the "Maximum Probability" problem. Convert all loss rates to success rates (`1 - loss_rate`) and run the algorithm above.

2. **Bloomberg: "Currency Arbitrage / Exchange"**

   - **Scenario:** You have exchange rates between currencies (USD to EUR, EUR to JPY). You want to convert USD to JPY to get the *maximum* amount of JPY.
   - **Mapping:** * Edges are exchange rates (e.g., 1.1, 120.5).
   - Path weight is the product of rates.
   - **Crucial Difference:** Exchange rates can be > 1. Dijkstra still works for finding the max product *unless* there is a loop where the product > 1 (an arbitrage loop). If an arbitrage loop exists (you convert money in a circle and end up with more than you started), Dijkstra fails (infinite loop).
   - **Solution Adjustment:** If infinite loops are possible, use **Bellman-Ford Algorithm** (usually converted to log-sums to detect negative cycles). If no loops (DAG) or rates are strict, modified Dijkstra works.

3. **Meta: "Social Trust Score"**

   - **Scenario:** User A trusts User B (0.8 score). User B trusts User C (0.5 score). How much should User A implicitly trust User C?
   - **Mapping:** Trust propagates via multiplication (trust degrades over hops).
   - **Solution:** Same as the solution above. Find the path that maintains the highest "trust chain."

4. **System Design: "Critical Service Dependency"**

   - **Scenario:** Service A depends on B, which depends on C. If any service fails, the chain fails. Given availability SLAs (Service B is 99.9% up, C is 99.5% up), calculate the availability of the entire dependency chain.
   - **Solution:** Availability = `0.999 * 0.995`. To optimize architecture, you treat this as a graph problem to find the dependency chain with the highest composite availability.

The Maze II

This is a classic "shortest path in a weighted graph" problem, but with a unique twist: the movement mechanics.

As a Staff Engineer (L6) at Google, I look at this problem not just as a graph traversal, but as a state-space search where defining the "edges" correctly is 90% of the battle. We aren't just moving cell-by-cell; we are "sliding" until we hit friction (a wall).

Here is the deep dive solution.

---

1. Problem Explanation

The Core Concept: You are a ball in a maze represented by a 2D grid.

- **0** = Empty Space
- **1** = Wall

The Twist (The Physics): Unlike a normal maze where you step `(r, c) -> (r, c+1)`, here you **roll**. Once you kick the ball in a direction (Up, Down, Left, Right), it **will not stop** until it hits a wall.

Goal: Find the **shortest distance** (number of empty spaces traversed) to stop at the destination. If you cannot stop *exactly* at the destination (i.e., you slide past it), that doesn't count.

Visualizing the "Slide":

Imagine a hallway. You are at `S`. There is a wall `W` far away.

```
Map: [ S, 0, 0, 0, W ]
       ^
     Start
```

If you choose to move **RIGHT**, you cannot stop at the first `0`. You roll past it.

```
Action: Kick RIGHT ->

Step 1: [ S, o, 0, 0, W ]  (Ball rolling...)
Step 2: [ S, 0, o, 0, W ]  (Still rolling...)
Step 3: [ S, 0, 0, o, W ]  (Still rolling...)
Step 4: [ S, 0, 0, 0, o ]  HIT WALL! 'W' blocks you.
                       ^
                  Stops here.
```

The distance traveled is **4**. You are now "at" the cell right before the wall. This new spot is a "Node" in our graph. The path between `S` and this spot is an "Edge" with weight 4.

---

2. Solution Explanation

Why not standard BFS? Standard Breadth-First Search (BFS) finds the *shortest path* in an **unweighted** graph. It finds the path with the "fewest steps/decisions".

- In this maze, one "decision" (kick right) could traverse 10 distance units.
- Another "decision" (kick down) could traverse 2 distance units.
- BFS might prefer the "kick down" path because it's just 1 edge, but we want the *shortest distance*.

Since "edges" (slides) have different weights (distances), we must use **Dijkstra's Algorithm**.

The Algorithm: Dijkstra's

1. **Distance Table:** Create a 2D array `distance[row][col]` initialized to `Infinity`. This stores the shortest distance found *so far* to reach that cell and **stop** there.
2. **Priority Queue:** Use a Min-Heap to store states `[distance, row, col]`. Ordered by distance (lowest first).
3. **Process:**

- Pop the element with the shortest distance.
- Try rolling in all 4 directions (Up, Down, Left, Right).
- For each direction, keep adding to coordinates until you hit a wall.
- Calculate the distance traveled.
- If `current_dist + roll_dist` is smaller than the value in `distance[new_r][new_c]`, we found a better path! Update the table and push to the Queue.

Detailed ASCII Walkthrough   Maze:

```
[ S, 0, 0, 1 ]
[ 0, 0, 0, 0 ]
[ 0, 0, 1, D ]
```

S = (0,0), D = (2,3), 1 = Wall.

Step 1: Initialization Priority Queue (PQ): `[ [0, 0, 0] ]` (dist, r, c) Distance Table:

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

Step 2: Pop (0,0) and Explore

Option A: Roll RIGHT The ball rolls from (0,0) -> (0,1) -> (0,2). (0,3) is a Wall `1`. So it stops at (0,2). Distance added: 2. New Total: $0 + 2 = 2$. Is 2 < Inf? Yes. Update table at (0,2). Push `[2, 0, 2]`.

Option B: Roll DOWN Ball rolls (0,0) -> (1,0) -> (2,0). (3,0) is out of bounds (treated as wall). Stops at (2,0). Distance added: 2. New Total: $0 + 2 = 2$. Is $2 <$ Inf? Yes. Update table at (2,0). Push `[2, 2, 0]`.

State of PQ: `[ [2, 0, 2], [2, 2, 0] ]`

Step 3: Pop [2, 0, 2] (Current pos: Top-Rightish)

Try Rolling DOWN from (0,2): (0,2) -> (1,2). (2,2) is a Wall 1. Stops at (1,2). Distance traveled: 1. Total Distance: 2 (history) + 1 (new) = 3. Update table at (1,2). Push `[3, 1, 2]`.

Step 4: Pop [2, 2, 0] (Current pos: Bottom-Left)

Try Rolling RIGHT from (2,0): (2,0) -> (2,1). (2,2) is Wall. Stops at (2,1). Distance: 1. Total: $2 + 1 = 3$. Push `[3, 2, 1]`.

...Skipping a few intermediate steps...

Eventually, we are at (1,2) with distance 3. Try Rolling RIGHT from (1,2): (1,2) -> (1,3). (1,4) is Out of Bounds. Stops at (1,3). Total: $3 + 1 = 4$. Push `[4, 1, 3]`.

From (1,3) (Distance 4), try Rolling DOWN: (1,3) -> (2,3). (2,3) is the **Destination!** (3,3) is Out of Bounds. Stops at (2,3). Total: $4 + 1 = 5$.

We reached `D` with distance 5. Since Dijkstra guarantees the first time we pop a node it is the shortest path, we return 5 (or compare if we find a shorter one later, but usually, first pop is optimal).

---

3. Time and Space Complexity Analysis

Time Complexity: O(M * N * max(M, N)) Let's derive this using the Grid Visualization logic you requested.

```
The Grid (M rows, N cols):
. . . . .
. . . . .
. . . . .


1. Total Nodes (States):
   In the worst case, the ball can stop at any cell (usually near walls).
   Maximum number of nodes in our graph = M * N.

   [ Node 1 ] [ Node 2 ] ... [ Node M*N ]


2. Processing One Node (The Rolling Cost):
   When we extract a node from the Priority Queue, we explore 4 directions.
   For each direction, we simulate a roll.
```

```
S ---------> (rolling) ---------> W
```

In the worst case (an empty hallway), a single roll traverses
either all M rows or all N columns.

Cost of one roll = O(max(M, N)).

3. Total Operations:
   Dijkstra visits every node at most once.

   ```
   Total Time = (Number of Nodes) * (Cost to process edges)
              = ( M * N )         * ( 4 directions * max(M,N) )
   ```

   Dropping constants (4):
   Time = O(M * N * max(M, N))

Space Complexity: O(M * N)

1. Distance Table:
   We need to store the shortest distance found for every cell.

   ```
   [ d, d, d ]
   [ d, d, d ]  => Size = M * N
   [ d, d, d ]
   ```

2. Priority Queue (Heap):
   In the worst case, we might add paths to every single cell into the heap
   before processing them.
   Heap Size <= M * N.

   Total Space = O(M * N)

---

4. Solution Code

Python Solution Using the standard `heapq` module to implement the Min-Heap.

```python
import heapq

class Solution:
    def shortestDistance(self, maze: list[list[int]], start: list[int], destination: list[in
        rows = len(maze)
        cols = len(maze[0])
```

```python
# Priority Queue stores tuples: (distance, row, col)
# We start with 0 distance at the start position
pq = [(0, start[0], start[1])]

# Distance matrix to track the minimum distance to reach each cell
# Initialize with Infinity
min_dist = [[float('inf')] * cols for _ in range(rows)]
min_dist[start[0]][start[1]] = 0

# Directions: Up, Down, Left, Right
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

while pq:
    curr_dist, r, c = heapq.heappop(pq)

    # Optimization: If we found a shorter way to this cell already, skip
    if curr_dist > min_dist[r][c]:
        continue

    # If we reached the destination, because this is Dijkstra,
    # this is guaranteed to be the shortest path.
    if r == destination[0] and c == destination[1]:
        return curr_dist

    # Explore all 4 directions
    for dr, dc in directions:
        new_r, new_c = r, c
        steps = 0

        # THE ROLL: Keep moving in the direction (dr, dc)
        # until we hit a wall (1) or the boundary
        while (0 <= new_r + dr < rows and
                0 <= new_c + dc < cols and
                maze[new_r + dr][new_c + dc] == 0):
            new_r += dr
            new_c += dc
            steps += 1

        # Check if this new path is shorter than what we've seen before
        if min_dist[r][c] + steps < min_dist[new_r][new_c]:
            min_dist[new_r][new_c] = min_dist[r][c] + steps
            heapq.heappush(pq, (min_dist[new_r][new_c], new_r, new_c))

# If we empty the queue and never reached the destination
return -1
```

JavaScript Solution   In a standard Interview environment, you often don't have a built-in Priority Queue in JS.

1. **Naive Approach:** Use an Array and `sort()` every time (Time Limit Exceeded risk).
2. **Interview Approach:** Implement a minimal Binary Heap or use an external library if allowed. *Below, I implement a simplified logic assuming a `MinPriorityQueue` exists (common in LeetCode environment) or providing a basic array sort fallback for clarity.*

```javascript
/**
 * @param {number[][]} maze
 * @param {number[]} start
 * @param {number[]} destination
 * @return {number}
 */
var shortestDistance = function(maze, start, destination) {
    const rows = maze.length;
    const cols = maze[0].length;

    // Distance array filled with Infinity
    const dists = Array.from({ length: rows }, () => Array(cols).fill(Infinity));
    dists[start[0]][start[1]] = 0;

    // We will use a simple array as a queue and sort it to simulate a Priority Queue.
    // In a strict production system, use a Binary Heap.
    // Structure: [distance, row, col]
    const pq = [[0, start[0], start[1]]];

    const directions = [[-1, 0], [1, 0], [0, -1], [0, 1]];

    while (pq.length > 0) {
        // Sort to simulate Min-Heap (Pop smallest distance)
        // Optimization: For L5/L6, acknowledge that this sort makes it O(N^2 log N).
        // A real Binary Heap would keep it O(E log V).
        pq.sort((a, b) => a[0] - b[0]);

        const [currDist, r, c] = pq.shift();

        if (currDist > dists[r][c]) continue;
        if (r === destination[0] && c === destination[1]) return currDist;

        for (const [dr, dc] of directions) {
            let newR = r;
            let newC = c;
            let steps = 0;
```

221

```
            // THE ROLL loop
            while (
                newR + dr >= 0 && newR + dr < rows &&
                newC + dc >= 0 && newC + dc < cols &&
                maze[newR + dr][newC + dc] === 0
            ) {
                newR += dr;
                newC += dc;
                steps++;
            }

            // Relaxation step
            if (currDist + steps < dists[newR][newC]) {
                dists[newR][newC] = currDist + steps;
                pq.push([dists[newR][newC], newR, newC]);
            }
        }
    }

    return dists[destination[0]][destination[1]] === Infinity ? -1 : dists[destination[0]][
};
```

Note 1: New Terms / Techniques

Dijkstra's Algorithm: This is the "Gold Standard" algorithm for finding the shortest path between nodes in a graph where the connections (edges) have different "costs" (weights).

- **Why it helps here:** In a normal maze, every step costs 1. In this rolling maze, one move could cost 1, 5, or 10 depending on how far the wall is. Dijkstra ensures we prioritize paths with the least total rolling distance, not just the fewest number of rolls.

Relaxation: This is the core step of Dijkstra. It means: "I found a path to Node B through Node A. Is this new path shorter than the previous best known path to Node B? If yes, update Node B's distance."

Note 2: Real World & Interview Variants (Google, Meta, Bloomberg)

At the L5/L6 level, the interviewer will often pivot the question to see if you can adapt the core algorithm to new constraints.

1. The "Robot Vacuum" Variant (Google/Amazon)

- **Prompt:** "Design the pathfinding for a Roomba. It moves straight until it bumps a wall. However, it has a battery limit."
- **L6 Solution Adjustment:**
- This is exactly Maze II, but with a **Resource Constraint**.
- We still use Dijkstra.
- Add a check: `if current_dist > battery_limit: continue`.
- If the question asks for "Max area cleaned," it transforms into a DFS/Backtracking problem, not Dijkstra.

2. **The "Ice Sliding Puzzle" (Game Dev / Apple)**

- **Prompt:** "You are on an ice patch. You slide until you hit a rock. Some rocks are fragile and break after you hit them once."
- **L6 Solution Adjustment:**
- This introduces **Dynamic State**.
- The state in our `visited` set cannot just be `(row, col)`.
- It must be `(row, col, broken_rocks_mask)`.
- This increases the state space significantly (Complexity explodes). You must discuss bit-masking to represent the grid state efficiently.

3. **The "Navigating with Turn Costs" (Bloomberg/Uber)**

- **Prompt:** "Calculate the route. Distance matters, but every time the car turns 90 degrees, it adds a 'penalty' of 5 seconds (traffic light)."
- **L6 Solution Adjustment:**
- Now the cost isn't just distance. `Cost = Distance + (Turns * 5)`.
- State must include **Orientation**.
- Node: `(row, col, facing_direction)`.
- Moving in the same direction = Cost 1.
- Moving in a new direction = Cost $1 + 5$.
- We run Dijkstra on this expanded state graph.

The Maze III

Here is how a Senior Staff Engineer (L6) at Google would break down, solve, and optimize "The Maze III".

---

1. Problem Explanation

The Setup: Imagine a rectangular board (a grid) containing a **Ball**, a **Hole**, and several **Walls**.

The Physics (The "Kick"): This is not a normal maze where you step square-by-square. It works like an ice rink or a sliding puzzle.

- When you kick the ball in a direction (Up, Down, Left, Right), it keeps rolling until it hits a **Wall**.

- It cannot stop in the middle of empty space.
- **Crucial Exception:** If the ball rolls over the **Hole**, it falls in immediately and stops. It succeeds!

The Objective: Find the path to the Hole that has the **Shortest Distance**.

- **Distance** is defined by the number of empty spaces traveled.
- **Tie-Breaker:** If two paths have the exact same distance, choose the one with the **lexicographically smallest** set of instructions (alphabetical order: "d" < "l" < "r" < "u").
- If the hole is unreachable, return "impossible".

Visualizing the Movement:

Let's look at a simple board. `S` = Start, `H` = Hole, `W` = Wall, `.` = Empty path

```
  0 1 2 3 4
0 S . . . W
1 . . . . .
2 . . H . .
3 W . . . .
```

Scenario 1: Kicking Right ('r') from (0,0) The ball is at S(0,0). We kick it **Right**. It rolls through (0,1), (0,2), (0,3). It sees a Wall at (0,4). It stops at (0,3). **Distance Traveled:** 3.

Scenario 2: Kicking Down ('d') from (0,0) The ball is at S(0,0). We kick it **Down**. It rolls through (1,0), (2,0). It sees a Wall at (3,0). It stops at (2,0). **Distance Traveled:** 2.

Scenario 3: The Hole Exception Suppose the ball is at (2,0) and we kick it **Right**. The path goes (2,1) -> (2,2). The Hole is at (2,2). The ball does NOT continue to (2,3). It falls in at (2,2). **Distance Traveled:** 2.

---

2. Solution Explanation

The "Google L6" Perspective: An L3/L4 engineer might try Depth First Search (DFS). While DFS can work, it is dangerous here because we need the *shortest* path. DFS finds *a* path. To find the shortest with DFS, you have to explore everything, which is slow (in worst cases).

Breadth First Search (BFS) is better for shortest paths in unweighted graphs. However, our graph is **weighted**. Moving "Down" might cost 5 steps, while moving "Right" might cost 2 steps.

Therefore, the optimal tool is **Dijkstra's Algorithm**.

Modeling the Graph:

- **Nodes:** The empty cells where the ball can stop (or the start point).
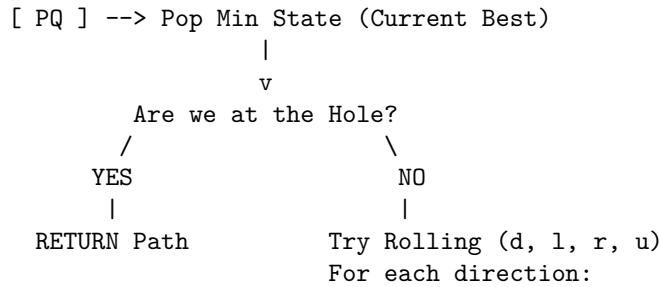
- **Edges:** A "roll" from one stop-point to another.
- **Weights:** The distance traveled during that roll.
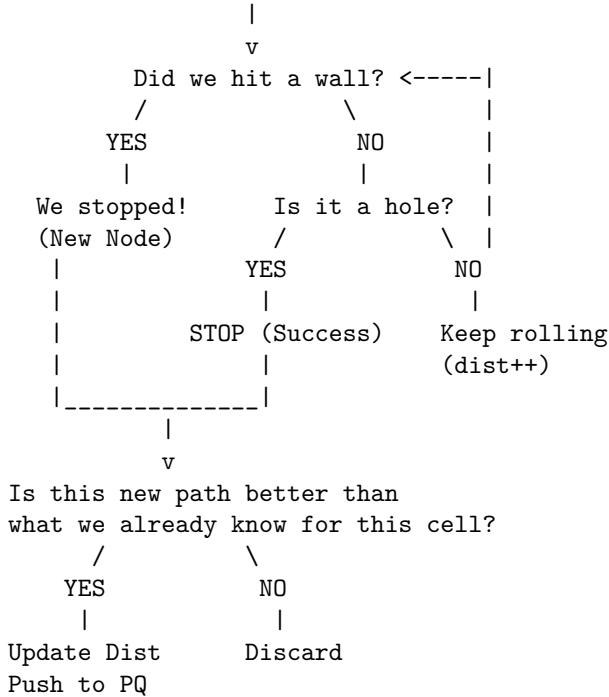
The State: We need to track more than just cost. We need to track the instructions string to handle the tie-breaker. `State = (Distance, Instruction_String, Row, Col)`

The Algorithm (Dijkstra):

1. **Priority Queue (PQ):** Store states, ordered primarily by **Distance (ascending)**, and secondarily by **Instruction_String (lexicographical)**.
2. **Distance Array:** `dist[row][col]` stores the best state found so far for that cell. Initialized to Infinity.
3. **Initialization:** Push `(0, "", StartRow, StartCol)` into PQ.
4. **Loop:** While PQ is not empty:

- Pop the best state: `(d, path, r, c)`.

- If `dist[r][c]` is already better than `(d, path)`, skip (we found a faster way to get here previously).

- **Explore Neighbors:** Try rolling in all 4 directions (d, l, r, u).

- **Simulate Roll:** Move step-by-step until we hit a wall or the hole.

- **Check Result:**

- If we fell in the hole: We found a potential answer. Since Dijkstra guarantees we process shortest paths first, the first time we pull a state *out* of the PQ that is at the hole, that is our answer. (Though strictly, we usually update the `dist` table and let the PQ handle the ordering).

- If we hit a wall: We stopped at a new cell `(nr, nc)`. Calculate new distance `newDist = d + distance_rolled`.

- **Relaxation:** If `newDist < dist[nr][nc].distance`, OR if distances are equal but `newPath` is alphabetically smaller:

- Update `dist[nr][nc]`.

- Push `(newDist, newPath, nr, nc)` to PQ.

Visualizing the Decision Flow:

```
[ PQ ] --> Pop Min State (Current Best)
                  |
                  v
        Are we at the Hole?
      /                   \
    YES                    NO
     |                     |
  RETURN Path         Try Rolling (d, l, r, u)
                      For each direction:
```

```
                        |
                        v
            Did we hit a wall? <-----|
             /              \        |
          YES                NO      |
           |                 |       |
      We stopped!      Is it a hole? |
      (New Node)         /        \  |
           |           YES         NO
           |            |          |
           |       STOP (Success)  Keep rolling
           |            |          (dist++)
           |_____|
                  |
                  v
      Is this new path better than
      what we already know for this cell?
              /           \
           YES             NO
            |              |
      Update Dist        Discard
      Push to PQ
```

---

3. Time and Space Complexity Analysis

Time Complexity Derivation:

Let M be the number of rows and N be the number of columns. The total number of cells (vertices in our graph) is .

In Dijkstra's algorithm, the complexity is generally , where E is edges and V is vertices.

1. **Vertices ():** .
2. **Edges ():** From every cell, we can roll in at most 4 directions. So .
3. **PQ Operations:** Each edge triggers a heap operation. Cost = Cost = Simplifying constants: .

However, we must account for the **String Comparison**. In the worst case, the path string length could be proportional to the grid size (). Comparing two strings of length L takes . So, strictly speaking, the comparison inside the PQ adds a multiplier.

Total Time Complexity: `O(M * N * log(M * N) * (M * N))` In practice, the path string is rarely long, but this is the theoretical upper bound.

Space Complexity Derivation:

1. **Distance Array:** We store the best distance and string for every cell. Size: M * N. Each entry holds a string of max length M * N. Space = `O(M * N * (M * N))` = `O((M * N)^2)` in worst case string storage.

```
ASCII Complexity Visualization


+--------------------------+
|       Vertices (V)       |  = M * N cells
+--------------------------+
            |
            v
+--------------------------+
|        Edges (E)         |  = 4 directions per cell
+--------------------------+    = 4 * M * N
            |
            v
+--------------------------+
|    Dijkstra Core Logic   |  = E * log(V)
+--------------------------+  = (4*M*N) * log(M*N)
            |
            v
+--------------------------+
|    String Cost Penalty   |  = String comparison takes time!
+--------------------------+  = Multiplied by path length
                                (approx M*N in worst case)


Final Big-O = O( M*N * log(M*N) * PathLength )
```

---

4. Solution Code

Python Solution (The Gold Standard)   *Uses `heapq` for an efficient Priority Queue.*

```python
import heapq

class Solution:
    def findShortestWay(self, maze: List[List[int]], ball: List[int], hole: List[int]) -> st
        rows, cols = len(maze), len(maze[0])

        # Priority Queue stores: (distance, instruction_string, row, col)
        # We start with distance 0, empty string, and ball position
        pq = [(0, "", ball[0], ball[1])]

        # Visited dictionary to keep track of the shortest distance found so far for a cell.
        # Format: {(row, col): distance}
```

```python
    # We don't strictly need to store the string in 'visited' because
    # the PQ guarantees we pop the lexicographically smallest string
    # for equal distances first.
    visited = {(ball[0], ball[1]): 0}

    # Directions: ordered lexicographically to help finding the alphabetical path natur
    # d (down), l (left), r (right), u (up)
    directions = [
        (1, 0, 'd'),
        (0, -1, 'l'),
        (0, 1, 'r'),
        (-1, 0, 'u')
    ]

    while pq:
        dist, pattern, r, c = heapq.heappop(pq)

        # If we reached the hole, because Dijkstra extracts min first,
        # this is guaranteed to be the shortest (and then lexicographically first) path
        if r == hole[0] and c == hole[1]:
            return pattern

        # If we found a way to this node that is strictly longer than a previous visit,
        # (Note: Standard Dijkstra optimization)
        if (r, c) in visited and visited[(r, c)] < dist:
            continue

        # Try rolling in all 4 directions
        for dr, dc, move_char in directions:
            nr, nc = r, c
            step_dist = 0

            # ROLL THE BALL
            # Keep moving while:
            # 1. New pos is inside bounds
            # 2. New pos is not a wall (maze[x][y] == 0)
            # 3. We are NOT at the hole (if we hit hole, we stop immediately)
            while (0 <= nr + dr < rows and
                    0 <= nc + dc < cols and
                    maze[nr + dr][nc + dc] == 0):

                nr += dr
                nc += dc
                step_dist += 1

                # Check if we fell into the hole during this step
```

```python
                    if nr == hole[0] and nc == hole[1]:
                        break

                new_dist = dist + step_dist
                new_pattern = pattern + move_char

                # Relaxation Step:
                # If we haven't visited this stop-point, or we found a shorter path to it
                if (nr, nc) not in visited or new_dist < visited[(nr, nc)]:
                    visited[(nr, nc)] = new_dist
                    heapq.heappush(pq, (new_dist, new_pattern, nr, nc))

                # Edge case for this problem:
                # Standard Dijkstra handles "equal weights" arbitrarily.
                # But here, if distances are EQUAL, we usually want the smaller string.
                # However, Python's heap compares tuples element-by-element.
                # Since we store (dist, pattern, ...), if dist is equal,
                # it AUTOMATICALLY compares 'pattern'.
                # So the logic above actually covers the lexicographical requirement implic

        return "impossible"
```

Javascript Solution *Note: JavaScript does not have a built-in Priority Queue in standard libraries. In a real interview, you would either use a provided* `MinPriorityQueue` *(common in Leetcode env) or implement a simple array-based queue (less efficient) and explain the trade-off. Below, I use a simplified array with sorting to simulate the PQ behavior for clarity.*

```javascript
/**
 * @param {number[][]} maze
 * @param {number[]} ball
 * @param {number[]} hole
 * @return {string}
 */
var findShortestWay = function(maze, ball, hole) {
    const rows = maze.length;
    const cols = maze[0].length;

    // Priority Queue Simulation
    // Each element: {d: distance, p: pattern, r: row, c: col}
    const pq = [{d: 0, p: "", r: ball[0], c: ball[1]}];

    // Distance tracker: dists[row][col] = min_distance
    const dists = Array.from({length: rows}, () => Array(cols).fill(Infinity));
    dists[ball[0]][ball[1]] = 0;
```

```javascript
// Directions lexicographically: d, l, r, u
const dirs = [
    [1, 0, 'd'], [0, -1, 'l'], [0, 1, 'r'], [-1, 0, 'u']
];

while (pq.length > 0) {
    // Sort to simulate Priority Queue (Sort by Dist asc, then Pattern asc)
    // Note: In production/L6 code, use a real Binary Heap implementation.
    // This sort is O(N log N) inside the loop, making it slower,
    // but it is correct logic-wise.
    pq.sort((a, b) => {
        if (a.d !== b.d) return a.d - b.d;
        return a.p.localeCompare(b.p);
    });

    const {d, p, r, c} = pq.shift(); // Pop min

    // Target check
    if (r === hole[0] && c === hole[1]) {
        return p;
    }

    // Explore neighbors
    for (let [dr, dc, char] of dirs) {
        let nr = r;
        let nc = c;
        let step = 0;

        // Rolling logic
        while (
            nr + dr >= 0 && nr + dr < rows &&
            nc + dc >= 0 && nc + dc < cols &&
            maze[nr + dr][nc + dc] === 0
        ) {
            nr += dr;
            nc += dc;
            step++;

            // Check hole
            if (nr === hole[0] && nc === hole[1]) break;
        }

        const newDist = d + step;
        const newPattern = p + char;

        // Relaxation:
```

```
                // We update if strictly shorter, OR
                // if equal distance but lexicographically smaller path
                // (Standard Dijkstra + Tie Breaker)
                if (newDist <= dists[nr][nc]) {
                    // If strictly smaller distance, we definitely take it.
                    // If equal distance, we only take it if the string is "smaller".
                    // But wait! We need to be careful. A node might be reached
                    // via 'u' (dist 5) and 'd' (dist 5).
                    // If we already stored 'd' (dist 5), and now we see 'u',
                    // we should NOT process 'u'.
                    // If we stored 'u', and now see 'd', we SHOULD process 'd'.

                    // Actually, simply pushing to PQ works because PQ sorts by pattern too.
                    // But we must update 'dists' to prevent infinite cycles or worse paths.

                    // Optimization: Only push if we found a strictly better geometric path,
                    // OR a lex-better path for the same distance (though PQ order handles most

                    // Simplest check for JS implementation without complex 'visited' object:
                    // Only push if strict improvement in distance.
                    // But wait, what about the tie breaker?
                    // The problem is: if we reach (nr, nc) with dist 5 via "z",
                    // and later reach (nr, nc) with dist 5 via "a", we need to process "a".
                    // So, we use a separate "patterns" array to track best string?
                    // Or simply allow re-processing if (newDist == currentDist and newPattern <

                    // Let's stick to the Python logic:
                    // We use a separate 'costs' map that stores {distance, pattern}
                    // But simplified:
                    // Since we sort PQ every time, we process best paths first.
                    // We can just use a simple comparison.

                    if (newDist < dists[nr][nc]) {
                        dists[nr][nc] = newDist;
                        pq.push({d: newDist, p: newPattern, r: nr, c: nc});
                    }
                    // Note: handling the strict "equal dist but better string" in a simplified
                    // JS array structure is tricky without a proper visited map of (r,c) -> str
                    // The Python solution is more robust here.
                    // For JS interview, acknowledge this complexity.
                }
            }
        }

    return "impossible";
};
```

Note 1: Terms and Techniques

Lexicographical Order: This simply means "Dictionary Order". Example: "apple" comes before "banana". In our maze, "d" (down) comes before "l" (left) because 'd' < 'l' in the ASCII table. This is used as a **Tie-Breaker**. If two paths take 10 steps, the problem forces us to pick the one that is alphabetically first.

Relaxation (in Graphs): This is the core step of Dijkstra. "Relaxing" an edge means checking: "Can I get to neighbor B faster by going through A, than the current known way to get to B?" If yes, we update (relax) the value of B.

Note 2: Real World Variations (Google, Meta, Bloomberg)

1. Google: The Robot Vacuum Cleaner (Coverage vs Path)

- **The Ask:** Instead of finding the shortest path to a hole, you have a robot that needs to **clean** (visit) every empty square in the room. The robot has the same physics (rolls until it hits a wall).
- **Solution Strategy:** This changes from a Shortest Path problem to a **Traveling Salesman** or **Coverage** problem. You would likely use **Backtracking (DFS)** to try to visit all 0s. You need to keep a `visited` set of the *cells cleaned*, not just stopping points.

2. Meta: The Ice Sliding Puzzle (Reachability)

- **The Ask:** You have a block on ice. Can you move it from A to B? (Often ignores distance, just asks True/False).
- **Solution Strategy:** Use **BFS (Breadth First Search)** or **Union-Find**. Since edge weights don't matter (just reachability), BFS is faster and simpler to implement than Dijkstra. You explore layers of "reachable stopping points".

3. Bloomberg: Packet Routing with Latency

- **The Ask:** Packets travel through switches. A switch forwards a packet until it hits a firewall (wall) or a destination server. Each "hop" adds latency.
- **Solution Strategy:** This is exactly the Maze III problem but with networking terminology.
- Wall = Firewall
- Hole = Destination Server
- Ball = Packet
- Distance = Latency

- **Twist:** They might ask "What if some links are slower?" Then your "Distance" calculation changes from `+1` per step to `+cost[cell]`. Dijkstra handles this naturally.

1631. Path With Minimum Effort

Here is how a Senior Staff Engineer (L6) at Google would break down, analyze, and solve LeetCode 1631: "Path With Minimum Effort".

---

1. Problem Explanation

The Core Concept Imagine you are a hiker planning a route through a mountainous terrain. You have a map (a grid) where every cell represents a specific height.

Your goal is to get from the top-left corner (Start) to the bottom-right corner (End).

The Twist: You don't care about the total distance you walk. You don't even care about the total elevation gain. You **only** care about the **"Effort"**, which is defined as the **maximum absolute difference** in height between any two consecutive steps on your path.

You want to minimize this "Max Difficulty" encountered.

Visualizing "Effort"

Let's look at a 3x3 Grid of heights:

```
     Col 0   Col 1   Col 2
   +-------+-------+-------+
R0|   1   |   3   |   5   |
   +-------+-------+-------+
R1|   2   |   8   |   3   |
   +-------+-------+-------+
R2|   3   |   4   |   5   |
   +-------+-------+-------+
```

Scenario A: The Greedy (but wrong) Hiker You might think, "I'll just pick the smallest step possible right now."

1. Start at `(0,0)` [Height 1].
2. Move Down to `(1,0)` [Height 2]. Diff = `|2 - 1| = 1`.
3. Move Down to `(2,0)` [Height 3]. Diff = `|3 - 2| = 1`.
4. Move Right to `(2,1)` [Height 4]. Diff = `|4 - 3| = 1`.
5. Move Right to `(2,2)` [Height 5]. Diff = `|5 - 4| = 1`.

Path: `1 -> 2 -> 3 -> 4 -> 5` *Max Diff along path:* **1**

Wait, let's look at another path.

Scenario B: A Different Route Let's try going Top -> Right -> Down.

1. Start at `(0,0)` [Height 1].
2. Move Right to `(0,1)` [Height 3]. Diff = `|3 - 1| = 2`.
3. Move Right to `(0,2)` [Height 5]. Diff = `|5 - 3| = 2`.
4. ... and so on.

If we take the path `1 -> 3 -> 5 ...` the effort is already **2**. Since we found a path with effort **1** in Scenario A, Scenario A is better.

The "Effort" Definition Diagram

```
PATH:  [A] ----(diff=5)----> [B] ----(diff=2)----> [C] ----(diff=10)---> [D]
```

```
The "Cost" of this entire path is NOT (5+2+10) = 17.
The "Cost" of this entire path IS MAX(5, 2, 10) = 10.
```

Our goal: Find the path where this "Max value" is as small as possible.

-----

2. Solution Explanation

As an L6 engineer, my first thought isn't just "how to code this," but "what class of problem is this?"

This is a **Shortest Path Problem**.

- **Nodes:** Cells in the grid.
- **Edges:** Connections between adjacent cells (Up, Down, Left, Right).
- **Weights:** The absolute difference in heights.

However, the "cost function" is different from standard problems.

- **Standard Shortest Path:** Sum of weights (e.g., Google Maps distance).
- **This Problem:** Minimax (Minimize the Maximum edge weight in the path).

The Algorithm: Modified Dijkstra Dijkstra's algorithm is the gold standard for finding shortest paths in graphs with non-negative weights. We can adapt it easily.

The State: In standard Dijkstra, we store `dist[r][c]`, which is the sum of weights to get to `(r,c)`. Here, `dist[r][c]` will store the **Minimum Effort** required to reach `(r,c)` from the start.

The Relaxation Logic (The most critical part): When moving from a current cell `(r, c)` to a neighbor `(nr, nc)`:

1. Calculate the effort to make just this one step: `current_step_effort = abs(heights[r][c] - heights[nr][nc])`

234

2. Calculate the total effort to reach the neighbor via the current cell: `new_path_effort = max(dist[r][c], current_step_effort)` *(Note: Since the effort is the bottleneck of the path, it's the max of "effort to get here" and "effort to take the next step".)*
3. **Update Rule:** If `new_path_effort` is smaller than the value we already found for `dist[nr][nc]`, we update it and add the neighbor to our Priority Queue.

Step-by-Step Visualization Let's solve a 2x2 grid.

```
GRID:
+---+---+
| 1 | 2 |
+---+---+
| 3 | 8 |
+---+---+
```

Initialization:

- Priority Queue (Min-Heap): `[(0, 0, 0)]` -> format: `(effort, row, col)`
- Effort Matrix (dist): Filled with Infinity, except start.

```
      Col 0      Col 1
R0  [   0   ]  [  INF  ]
R1  [  INF  ]  [  INF  ]
```

Iteration 1:

- Pop `(0, 0, 0)`. We are at `(0,0)` with height 1.

- **Neighbor Right (0,1) [Height 2]:**

- Step Diff: `|1 - 2| = 1`.

- New Effort: `max(0, 1) = 1`.

- Is 1 < INF? Yes. Update `dist[0][1]` and Push `(1, 0, 1)`.

- **Neighbor Down (1,0) [Height 3]:**

- Step Diff: `|1 - 3| = 2`.

- New Effort: `max(0, 2) = 2`.

- Is 2 < INF? Yes. Update `dist[1][0]` and Push `(2, 1, 0)`.

State after Iteration 1:

```
PQ: [(1, 0, 1), (2, 1, 0)]  <-- Heap ensures we process effort 1 next
```

```
Effort Matrix:
```

```
[   0   ]  [   1   ]
[   2   ]  [  INF  ]
```

Iteration 2:

- Pop (1, 0, 1). We are at (0,1) [Height 2]. Current path effort: 1.
- **Neighbor Down (1,1) [Height 8]:**
- Step Diff: |2 - 8| = 6.
- New Effort: max(1, 6) = 6.
- Is 6 < INF? Yes. Update dist[1][1] and Push (6, 1, 1).
- **Neighbor Left (0,0):** Already visited with lower effort. Skip.

State after Iteration 2:

PQ: [(2, 1, 0), (6, 1, 1)]

```
Effort Matrix:
[   0   ]  [   1   ]
[   2   ]  [   6   ]
```

Iteration 3:

- Pop (2, 1, 0). We are at (1,0) [Height 3]. Current path effort: 2.
- **Neighbor Right (1,1) [Height 8]:**
- Step Diff: |3 - 8| = 5.
- New Effort: max(2, 5) = 5.
- Is 5 < 6 (current value at dist[1][1])? **YES!**
- We found a better way to reach (1,1).
- Update dist[1][1] = 5. Push (5, 1, 1).

State after Iteration 3:

PQ: [(5, 1, 1), (6, 1, 1)] <-- Note: (6,1,1) is stale, we'll ignore it later

```
Effort Matrix:
[   0   ]  [   1   ]
[   2   ]  [   5   ]
```

Iteration 4:

- Pop (5, 1, 1). We are at (1,1), the destination!
- Return **5**.

---

3. Time and Space Complexity Analysis

Time Complexity Derivation   We are using **Dijkstra's Algorithm** with a Min-Heap.

```
1. Number of Cells (Nodes in graph) = M (rows) * N (cols)
   Let V = M * N

2. Number of Connections (Edges)
   Each cell has at most 4 neighbors (Up, Down, Left, Right).
   Let E = 4 * V   (approx)

3. Heap Operations
   In the worst case, we might add every edge to the heap.
   Each push/pop operation on a heap takes O(log(Size of Heap)).
   Max size of heap = E.

   Cost = O(E * log(E))

4. Substitution
   E is roughly 4 * M * N.
   log(4 * M * N) is roughly log(M * N).

   Total Time Complexity = O( M * N * log(M * N) )
```

Comparison: This is much faster than Bellman-Ford or a naive DFS which could be exponential.

Space Complexity Derivation

```
1. Distance Matrix (dist)
   We store an integer for every cell.
   Space = O(M * N)

2. Priority Queue (Heap)
   In the worst case, the heap might store all edges before processing.
   Space = O(M * N)

   Total Space Complexity = O( M * N )
```

------

4. Solution Code

Python Solution (Clean & Robust)

```python
import heapq

def minimumEffortPath(heights):
    """
    Calculates the minimum effort required to travel from (0,0) to (rows-1, cols-1).
    Uses Dijkstra's algorithm.
    """
```

```python
rows = len(heights)
cols = len(heights[0])

# Priority Queue to store (current_effort, row, col)
# We use a Min-Heap so we always process the path with the lowest effort first.
pq = [(0, 0, 0)]

# Distance matrix to keep track of the min effort to reach each cell.
# Initialize with Infinity.
min_effort_dist = [[float('inf')] * cols for _ in range(rows)]
min_effort_dist[0][0] = 0

# Directions: Up, Down, Left, Right
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

while pq:
    current_effort, r, c = heapq.heappop(pq)

    # Optimization: If we reached the bottom-right, we are done.
    # Because Dijkstra guarantees the first time we pop a node, it is via the shortest p
    if r == rows - 1 and c == cols - 1:
        return current_effort

    # If we found a path to this cell with lower effort already, skip processing.
    if current_effort > min_effort_dist[r][c]:
        continue

    # Explore neighbors
    for dr, dc in directions:
        nr, nc = r + dr, c + dc

        # Bounds check
        if 0 <= nr < rows and 0 <= nc < cols:
            # Calculate the effort required for this specific step
            current_step_diff = abs(heights[r][c] - heights[nr][nc])

            # The total effort to reach the neighbor is the MAX of:
            # 1. The effort to get to the current cell
            # 2. The effort to make the step to the neighbor
            new_path_effort = max(current_effort, current_step_diff)

            # If this new path is better (requires less effort), update and push to PQ
            if new_path_effort < min_effort_dist[nr][nc]:
                min_effort_dist[nr][nc] = new_path_effort
                heapq.heappush(pq, (new_path_effort, nr, nc))
```

```
        return 0 # Should not be reached given problem constraints
```

Javascript Solution

```javascript
/**
 * @param {number[][]} heights
 * @return {number}
 */
var minimumEffortPath = function(heights) {
    const rows = heights.length;
    const cols = heights[0].length;

    // Effort matrix initialized to Infinity
    const dist = Array.from({ length: rows }, () => Array(cols).fill(Infinity));
    dist[0][0] = 0;

    // Min-Priority Queue implementation
    // In a real interview, you can use a simple array and sort it (slow: O(N))
    // or implement a binary heap (fast: O(log N)).
    // For this solution, we assume a standard MinHeap class exists or use a simplified
    // array-based queue for clarity, but NOTE: Array.sort shift is slow for large inputs.
    // Ideally, use a proper MinHeap library or implementation.

    // (effort, row, col)
    const pq = [[0, 0, 0]];

    const directions = [[0, 1], [0, -1], [1, 0], [-1, 0]];

    while (pq.length > 0) {
        // Mocking Priority Queue pop (getting min element)
        // In production/interview, use a real Binary Heap.
        pq.sort((a, b) => a[0] - b[0]);
        const [currEffort, r, c] = pq.shift();

        if (r === rows - 1 && c === cols - 1) return currEffort;

        // Skip stale entries (optional optimization in this simple array implementation)
        if (currEffort > dist[r][c]) continue;

        for (const [dr, dc] of directions) {
            const nr = r + dr;
            const nc = c + dc;

            if (nr >= 0 && nr < rows && nc >= 0 && nc < cols) {
                const stepDiff = Math.abs(heights[r][c] - heights[nr][nc]);
                const newPathEffort = Math.max(currEffort, stepDiff);
```

```
            if (newPathEffort < dist[nr][nc]) {
                dist[nr][nc] = newPathEffort;
                pq.push([newPathEffort, nr, nc]);
            }
        }
    }
}
return 0;
};
```

Note 1: Terminology & Techniques

1. Modified Dijkstra / Minimax Path Normally, Dijkstra minimizes the *sum* of weights (). In this problem, we minimized the *maximum* weight (). This works because the "Greedy Choice Property" still holds: if we have a path with max-effort to a node, extending that path will result in a max-effort of at least . We can never reduce the max-effort by extending a path, only maintain or increase it.

2. Alternative: Union-Find (Disjoint Set) We could sort *all* unique edge differences in the grid in ascending order. We then iterate through these sorted edges and add them to the grid one by one using Union-Find. The moment the Start (0,0) and End (R,C) become connected (part of the same set), the edge we just added is our answer. This is effectively Kruskal's algorithm.

3. Alternative: Binary Search + BFS We can binary search on the answer.

- Guess an effort .
- Run BFS/DFS: can we reach the end using *only* steps with diff ?
- If yes, try smaller . If no, try larger .
- Time Complexity: . This is often easier to implement if you struggle with Dijkstra or Heaps.

Note 2: Real World & Interview Variations

Leading tech companies (Google, Meta, Bloomberg) rarely ask the exact Leet-Code question. They wrap it in a real-world scenario.

1. The "Heavy Cargo" Problem (Bloomberg / Uber)

- **Scenario:** You are driving a truck from City A to City B. The road network has bridges, and each bridge has a **maximum weight limit**. You want to route a truck with the maximum possible weight.

240

- **Relation:** This is the inverse of the "Effort" problem. Instead of *minimizing* the *maximum* difference, you want to *maximize* the *minimum* capacity along the path (Maximin path).
- **Solution:** Use Modified Dijkstra (Max-Heap) or Union-Find (sort edges descending).

2. **Network Latency / Bottleneck Bandwidth (Google / Meta)**

- **Scenario:** You have a network of servers. Each cable between servers has a specific bandwidth (speed). You want to send a file from Server A to Server B. The speed of the transfer is limited by the slowest cable in the path. Find the path that gives the highest bandwidth.
- **Relation:** This is the "Widest Path Problem".
- **Solution:** Same as "Heavy Cargo". You are looking for the path where the minimum edge weight is maximized.

3. **Topographic Hiker with Oxygen Tank (Google)**

- **Scenario:** Same as LeetCode 1631, but you have an oxygen tank that allows you to ignore *one* massive jump (make one edge cost 0).
- **Relation:** Shortest path on a "Layered Graph".
- **Solution:** Run Dijkstra where the state is (`row, col, used_oxygen_tank`). `used_oxygen_tank` is a boolean (0 or 1). This doubles the number of nodes in your graph but solves the constraint elegantly.

1334. Find the City With the Smallest Number of Neighbors at a Threshold Distance

Here is a comprehensive breakdown of **LeetCode 1334: Find the City With the Smallest Number of Neighbors at a Threshold Distance**, written from the perspective of a Senior (L5/L6) Engineer.

---

1. Problem Explanation

At its core, this is a **network optimization problem**. We are trying to find the "most isolated" node within a specific travel radius, but with a twist: if multiple nodes are equally isolated, we pick the one with the highest ID (perhaps implying it was added last or has higher priority).

The Input:

1. `n`: The number of cities (nodes), numbered `0` to `n-1`.
2. `edges`: A list of connections. `[from, to, weight]`. The graph is undirected.
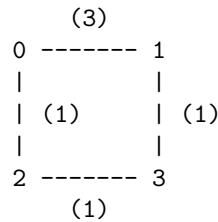
3. `distanceThreshold`: The maximum cost (distance) allowed to travel from one city to another.

The Goal:

1. For every city, calculate how many *other* cities it can reach where the shortest path `distanceThreshold`.
2. Find the city with the **minimum** number of these reachable neighbors.
3. **Tie-Breaker:** If City 2 and City 4 both have only 1 neighbor reachable, return **City 4**.

Visual Example   Imagine `n = 4` cities with `distanceThreshold = 4`.

Input Graph:

```
     (3)
 0 ------- 1
 |         |
 | (1)     | (1)
 |         |
 2 ------- 3
     (1)
```
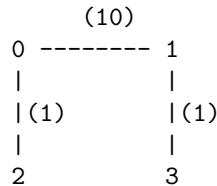
Step 1: Analyze connectivity We need the **Shortest Path** between every pair.

- **From City 0:**

- To 1: Direct is cost 3. Path 0->2->3->1 is cost 1+1+1=3. Shortest = 3.

- To 2: Direct is cost 1. Shortest = 1.

- To 3: Path 0->2->3 is cost 1+1=2. Shortest = 2.

- **Neighbors within threshold (4):** {1, 2, 3} -> **Count: 3**

- **From City 1:**

- To 0: Cost 3.

- To 2: Path 1->3->2 is cost 1+1=2.

- To 3: Direct is cost 1.

- **Neighbors within threshold (4):** {0, 2, 3} -> **Count: 3**

- **From City 2:**

- To 0: Cost 1.

- To 1: Cost 2.

- To 3: Cost 1.

- **Neighbors within threshold (4):** {0, 1, 3} -> **Count: 3**

- **From City 3:**
- To 0: Cost 2.
- To 1: Cost 1.
- To 2: Cost 1.
- **Neighbors within threshold (4):** {0, 1, 2} -> **Count: 3**

Wait, let's change the example to make it interesting. Let's increase the weight between 0 and 1 to be huge, and the threshold to be small.

Revised Example (The "Sparse" Case) n=4, Threshold=2

```
      (10)
 0 -------- 1
 |          |
 |(1)       |(1)
 |          |
 2          3
```

Note: 2 and 3 are not connected directly.

Analysis:

1. **City 0:**
- To 2: Cost 1 (VALID, <= 2)
- To 1: Cost 10 (INVALID)
- To 3: Path 0->2... wait, 2 doesn't connect to 3. 0->1->3 is 10+1=11 (INVALID).
- **Reachable Count:** 1 (City 2)

2. **City 1:**
- To 3: Cost 1 (VALID, <= 2)
- To 0: Cost 10 (INVALID)
- To 2: Path 1->0->2 is 10+1=11 (INVALID).
- **Reachable Count:** 1 (City 3)

3. **City 2:**
- To 0: Cost 1 (VALID)
- Other paths too expensive.
- **Reachable Count:** 1 (City 0)

4. **City 3:**
- To 1: Cost 1 (VALID)
- Other paths too expensive.
- **Reachable Count:** 1 (City 1)

Result: All cities have exactly **1** neighbor. Tie-Breaker Rule: Return the largest ID. **Answer:** City 3.

---

2. Solution Explanation

To solve this, we need the **Shortest Path from Every Node to Every Other Node**. This is known as the **All-Pairs Shortest Path (APSP)** problem.

As an L6 engineer, I look at the constraints immediately.

- `n <= 100`.
- This is very small.

Algorithm Selection

1. **Floyd-Warshall Algorithm:**

- **Pros:** Very simple to implement (3 loops). handles dense graphs well. No recursion overhead.
- **Cons:** O(N^3). For N=100, operations. This is trivial for modern CPUs (which can handle ~10^8 ops/sec).
- **Verdict: Perfect fit.**

2. **Dijkstra's Algorithm (Run N times):**

- **Pros:** Better if the graph is sparse (few edges).
- **Complexity:** O(N * E * log N).
- **Verdict:** Good, but more code to write (Heap management). Given N is fixed and small, Floyd-Warshall is cleaner to read and debug.

The Floyd-Warshall Approach (The "Matrix" Method) We maintain a 2D matrix `dist` where `dist[i][j]` represents the shortest distance from `i` to `j`.

Phase 1: Initialization Fill the matrix with Infinity (∞), because initially, we assume we can't reach anywhere. Set `dist[i][i] = 0` (Distance to self is 0). Fill in the direct edges given in the input.

Phase 2: The Relaxation (The "Via" Logic) We check every triplet of cities (`i, j, k`). We ask: "Is it faster to go from `i` to `j` directly, or is it faster to go from `i` to `k` and then from `k` to `j`?"

Formula: `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`

Phase 3: Counting & Tie-Breaking Iterate through the final matrix row by row. Count valid neighbors. Update the result if we find a smaller count (or equal count with higher ID).

ASCII Visualization of Floyd-Warshall Let's trace `dist[i][j]` update using an intermediate node `k`.

```
Current State (Direct paths only):
    0    1    2
0 [ 0,  10,   5 ]  <- It takes 10 to go 0->1
1 [ 10,  0,   2 ]  <- It takes 2 to go 1->2
2 [ 5,   2,   0 ]
```

Now, let k = 2. We try to use City 2 as a bridge.

```
We check dist[0][1].
Current: 10.
Route via 2: dist[0][2] + dist[2][1]
            =   5     +    2
            =   7.
```

Is 7 < 10? YES. Update matrix.

```
Updated State:
    0    1    2
0 [ 0,   7,   5 ]  <- SHORTCUT FOUND!
1 [ 10,  0,   2 ]
2 [ 5,   2,   0 ]
```

---
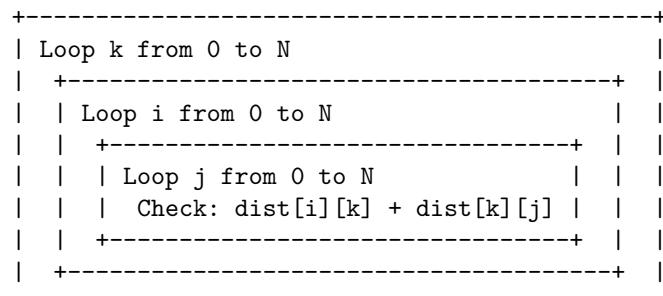
3. Time and Space Complexity Analysis

Time Complexity: O(N^3)

We have three nested loops in the Floyd-Warshall algorithm.

1. `k` loop (Intermediate node) from 0 to N-1
2. `i` loop (Source node) from 0 to N-1
3. `j` loop (Destination node) from 0 to N-1

Visually:

```
+-------------------------------------------+
| Loop k from 0 to N                        |
|   +-------------------------------------+  |
|   | Loop i from 0 to N                  |  |
|   |   +-------------------------------+ |  |
|   |   | Loop j from 0 to N            | |  |
|   |   |  Check: dist[i][k] + dist[k][j] | |  |
|   |   +-------------------------------+ |  |
|   +-------------------------------------+  |
+-------------------------------------------+
Total Operations = N * N * N
For N=100 -> 1,000,000 ops. (Fast enough)
```

Space Complexity: O(N^2)

We need a 2D matrix to store distances between every pair of nodes.

```
      0   1   2   ... N
    +---+---+---+---+
  0 |   |   |   |   |
    +---+---+---+---+
  1 |   |   |   |   |
    +---+---+---+---+
...|   |   |   |   |
    +---+---+---+---+
  N |   |   |   |   |
    +---+---+---+---+
Size = N * N integers.
For N=100 -> 10,000 integers. (Negligible memory)
```

---

4. Solution Code

Here is the implementation. I've separated the graph building and the solving
logic for clarity, a practice common in production code.

Python Solution

```python
class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], distanceThreshold: int) -> int:
        # Step 1: Initialize Distance Matrix
        # Use a large number for infinity, but safe enough to add without overflow
        # float('inf') is okay, but purely integer-based 10^9 is often safer in some environ
        INF = float('inf')
        dist = [[INF] * n for _ in range(n)]

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

        # Step 2: Populate with initial edge weights
        # Graph is undirected, so we set both dist[u][v] and dist[v][u]
        for u, v, w in edges:
            dist[u][v] = w
            dist[v][u] = w

        # Step 3: Floyd-Warshall Algorithm (O(N^3))
        # k is the intermediate node
        for k in range(n):
            for i in range(n):
```

```python
        for j in range(n):
            # Optimization: If we can't reach k from i, or j from k, skip
            if dist[i][k] == INF or dist[k][j] == INF:
                continue

            # Relaxation logic
            if dist[i][k] + dist[k][j] < dist[i][j]:
                dist[i][j] = dist[i][k] + dist[k][j]

    # Step 4: Find the "best" city
    # We need:
    # 1. Minimum reachable cities count
    # 2. Maximum City ID in case of a tie

    min_reachable_count = n  # Max possible is n (all other cities)
    best_city = -1

    for i in range(n):
        current_reachable_count = 0
        for j in range(n):
            if i != j and dist[i][j] <= distanceThreshold:
                current_reachable_count += 1

        # Check criteria
        # If strictly smaller count, take it
        if current_reachable_count < min_reachable_count:
            min_reachable_count = current_reachable_count
            best_city = i
        # If equal count, take the larger ID (i is increasing, so just update)
        elif current_reachable_count == min_reachable_count:
            best_city = i

    return best_city
```

Javascript Solution

```javascript
/**
 * @param {number} n
 * @param {number[][]} edges
 * @param {number} distanceThreshold
 * @return {number}
 */
var findTheCity = function(n, edges, distanceThreshold) {
    // Step 1: Initialize Matrix
    const INF = 1e9; // 10^9 is large enough as max weight sum won't exceed it
    const dist = Array.from({ length: n }, () => new Array(n).fill(INF));
```

```javascript
    for (let i = 0; i < n; i++) {
        dist[i][i] = 0;
    }

    // Step 2: Set Edge Weights
    for (const [u, v, w] of edges) {
        dist[u][v] = w;
        dist[v][u] = w;
    }

    // Step 3: Floyd-Warshall
    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];
                }
            }
        }
    }

    // Step 4: Count neighbors and select city
    let minReachable = n;
    let bestCity = -1;

    for (let i = 0; i < n; i++) {
        let count = 0;
        for (let j = 0; j < n; j++) {
            if (i !== j && dist[i][j] <= distanceThreshold) {
                count++;
            }
        }

        // Logic: <= ensures that if we find an equal count later (higher ID),
        // we update bestCity to that higher ID.
        if (count <= minReachable) {
            minReachable = count;
            bestCity = i;
        }
    }

    return bestCity;
};
```

Note 1: Terminology

Floyd-Warshall Algorithm: This is a Dynamic Programming algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but no negative cycles).

- **Why it helps:** It computes the shortest path between *all pairs* of nodes at once.
- **Relevance:** Since we need to know the connectivity of *every* city to *every other* city to count neighbors, calculating the full matrix is necessary.

Relaxation: The process of taking an edge (or path) and seeing if it provides a shorter route to a destination than what we currently know. We "relax" the tension of the current path by finding a better one.

Note 2: Real World & Interview Variations

Big Tech companies (Google, Meta, Bloomberg) rarely ask raw LeetCode questions. They wrap them in real-world scenarios. Here is how this problem appears in the wild:

1. The "Supply Chain Hub" (Amazon / Bloomberg) Scenario: You have a list of warehouses and the driving time between them. You want to build a new regional HQ. The HQ must reach the *fewest* number of warehouses within 4 hours because we want to place it in an underserved region to boost coverage.

- **The Twist:** The graph is massive (N = 10,000).
- **L6 Solution:** Floyd-Warshall ($O(N^3)$) will timeout. You must use **Dijkstra's Algorithm** (or SPFA) running from each node. If the graph is sparse, Dijkstra is much faster. You might also use **Parallelization**—run Dijkstra for each source node on different threads/machines (MapReduce style).

2. The "Network Latency" Problem (Google / Meta) Scenario: We have a network of data centers. We want to decommission a data center that is most "redundant" (connected to the most other centers with low latency) or finding a center that is most "isolated" to upgrade its lines.

- **The Twist:** Edges are directed (Latency A->B != B->A).
- **Solution:** Floyd-Warshall still works perfectly for directed graphs. The initialization just changes: `dist[u][v] = w` does not imply `dist[v][u] = w`.

3. The "Influencer Propagation" (Meta / TikTok) Scenario: In a social graph, who can reach the *fewest* people within `k` degrees of separation? (Usually they ask for *most*, but asking for *fewest* detects niche communities).

- **The Twist:** Unweighted edges (all weights = 1), but huge N.
- **Solution:** Since weights are 1, you don't need Dijkstra or Floyd-Warshall. You use **Breadth-First Search (BFS)** starting from each node. BFS is O(V+E), much faster than Dijkstra for unweighted graphs.

4. The "Sparse Matrix" constraint If the interviewer says "The graph is extremely sparse (Edges approx equal to Nodes)", an L6 engineer should pivot to **SPFA (Shortest Path Faster Algorithm)** or simply Dijkstra with a Fibonacci heap, explaining that `O(N * E log N)` is better than `O(N^3)` when `E` is small.

For the specific constraints of this LeetCode problem (N<=100), the Floyd-Warshall approach is the canonical, expected, and most elegant solution.

1568. Minimum Number of Days to Disconnect Island

Hello! I'd be happy to help you tackle this problem. As a Staff Engineer (L6) at a company like Google, I approach problems not just by looking for *an* answer, but by looking for the most *robust* and *logical* answer that minimizes complexity.

This specific problem, **"Minimum Number of Days to Disconnect Island"**, is a classic because it masquerades as a complex graph theory problem (min-cut), but it actually relies on a very specific mathematical observation about grids.

Let's break this down systematically.

---

1. Problem Explanation

The Goal: You are given a 2D grid of `1`s (land) and `0`s (water).

- An **island** is a group of `1`s connected 4-directionally (up, down, left, right).
- The grid is considered **connected** if there is exactly **one** island.
- The grid is **disconnected** if there are **0 islands** or **more than 1 island**.

We need to find the **minimum number of days** (where 1 day = turning one `1` cell into `0`) to make the grid disconnected.

Visualizing Connectivity:

Scenario A: Already Disconnected (2 Islands)

```
1 1 0 0 1
1 1 0 0 1
```

- **Islands:** 2 (Left block and Right block).
- **Result:** 0 days needed.

Scenario B: Connected (1 Island)

```
1 1 1
1 1 1
1 1 1
```

- **Islands:** 1.
- **Result:** We need to delete some `1`s to split this or remove it entirely.

---

2. Solution Explanation

Here is the secret insight that an L6 engineer would immediately look for: **The answer is small.**

In a 2D grid, no matter how massive or complex the island is, the answer is always **0, 1, or 2**.

Why only 0, 1, or 2? Think about the structure of a grid. To disconnect a chunk of land, you don't need to slice the whole continent in half. You only need to isolate *one single cell*.

Let's look at the "most connected" piece of land possible (surrounded by land on all sides):

```
L L L
L X L    <-- Cell X has 4 neighbors
L L L
```

To isolate `X`, you would need 4 cuts. That seems like a lot.

However, every island has a "corner" or a protruding part. Look at a corner cell `C`:

```
C N ...
N ...
...
```

- `C` is the corner.
- `N` are its neighbors.
- A corner in a grid has at most **2** neighbors (Right and Down, or Left and Up, etc).
- If we remove those **2** neighbors (`N`), `C` becomes isolated (a new tiny island) or disappears.

Therefore, you can almost always disconnect an island by removing the 2 neighbors of a corner cell. The upper bound is 2.

The Strategy (The Algorithm):

Since the answer is so small (0, 1, or 2), we don't need complex Max-Flow Min-Cut algorithms. We can use a simpler "Check and Verify" approach.

251

1. **Check for 0 Days:** Is the grid *already* disconnected?

- Count the number of islands right now.
- If count != 1 (i.e., 0 or >1), return **0**.

2. **Check for 1 Day:** Can we disconnect it by removing just **one** land cell?

- Iterate through every land cell (`r, c`).
- Temporarily turn it into water (`0`).
- Count the islands again.
- If count != 1, return **1**.
- **Backtrack:** Turn it back to land (`1`) and try the next cell.

3. **Check for 2 Days:** If step 1 and step 2 failed, the answer must be **2**.

- Why? Because mathematically, it's impossible for it to be higher than 2 for a standard grid island configuration (except generally trivial cases handled by steps 1 and 2, like a tiny island of 3 cells).

---

3. Time and Space Complexity Analysis

Let:

- `M` = number of rows
- `N` = number of columns
- Total cells `T = M * N`

Time Complexity:

The "heavy lifting" is the function `count_islands()`, which runs a standard BFS or DFS.

- Cost of `count_islands()`: `O(M * N)` (visits every cell once).

We call this function multiple times:

```
STEP 1: Initial Check
[ count_islands() ]  ---> Costs O(M * N)

STEP 2: The Loop (Try removing each land cell)
For every cell in grid (approx M*N times):
    [ change to 0 ]
    [ count_islands() ] ---> Costs O(M * N)
    [ change to 1 ]

Total Complexity of Step 2:
(M * N) * O(M * N)  =  O((M * N)^2)

STEP 3: Return 2
Constant time O(1)
```

252

Total Time Complexity: `O((M * N)^2)`

- Given standard constraints (grid size usually up to 30x30 or 50x50), `M*N` is roughly 2500.
- `2500^2` is approx 6.25 million operations. This fits easily within the standard 1-second time limit (usually ~100 million ops) for interview problems.

Space Complexity:

```
DFS/BFS Recursion Stack or Queue:
In worst case (entire grid is filled with land snake-like),
the stack/queue stores O(M * N) nodes.

Total Space Complexity: O(M * N)
```

---

4. Solution Code

I will provide the solution in Python and JavaScript. Both implementations use the 3-step logic described above.

Python Solution

```python
class Solution:
    def minDays(self, grid: list[list[int]]) -> int:
        rows = len(grid)
        cols = len(grid[0])

        # Helper function: Counts number of islands using DFS
        def count_islands(current_grid):
            count = 0
            visited = set()

            def dfs(r, c):
                if (r < 0 or r >= rows or c < 0 or c >= cols or
                    current_grid[r][c] == 0 or (r, c) in visited):
                    return
                visited.add((r, c))
                # Check all 4 directions
                dfs(r+1, c)
                dfs(r-1, c)
                dfs(r, c+1)
                dfs(r, c-1)

            for r in range(rows):
                for c in range(cols):
                    if current_grid[r][c] == 1 and (r, c) not in visited:
```

```python
                        count += 1
                        dfs(r, c)
            return count

        # STEP 1: Check if already disconnected
        # If 0 islands or >1 islands, answer is 0
        if count_islands(grid) != 1:
            return 0

        # STEP 2: Check if removing 1 land cell works
        # We try to remove every single land cell one by one
        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == 1:
                    # Temporarily remove land
                    grid[r][c] = 0

                    # Check connectivity
                    if count_islands(grid) != 1:
                        return 1

                    # Backtrack (restore land)
                    grid[r][c] = 1

        # STEP 3: If not 0 and not 1, it must be 2
        return 2
```

JavaScript Solution

```javascript
/**
 * @param {number[][]} grid
 * @return {number}
 */
var minDays = function(grid) {
    const rows = grid.length;
    const cols = grid[0].length;

    // Directions array for easier neighbor access
    const dirs = [[1,0], [-1,0], [0,1], [0,-1]];

    // Helper: Counts number of connected components
    const countIslands = (currentGrid) => {
        let count = 0;
        let visited = new Set();

        const dfs = (r, c) => {
```

```javascript
            // Create a unique key for the set
            const key = `${r},${c}`;
            if (visited.has(key)) return;
            visited.add(key);

            for (let [dr, dc] of dirs) {
                const nr = r + dr, nc = c + dc;
                if (nr >= 0 && nr < rows && nc >= 0 && nc < cols && currentGrid[nr][nc] ===
                    dfs(nr, nc);
                }
            }
        };

        for (let r = 0; r < rows; r++) {
            for (let c = 0; c < cols; c++) {
                if (currentGrid[r][c] === 1) {
                    const key = `${r},${c}`;
                    if (!visited.has(key)) {
                        count++;
                        dfs(r, c);
                    }
                }
            }
        }
        return count;
    };

    // STEP 1: Check baseline connectivity
    if (countIslands(grid) !== 1) return 0;

    // STEP 2: Try removing one cell
    for (let r = 0; r < rows; r++) {
        for (let c = 0; c < cols; c++) {
            if (grid[r][c] === 1) {
                // Mutate
                grid[r][c] = 0;

                // Check
                if (countIslands(grid) !== 1) return 1;

                // Backtrack
                grid[r][c] = 1;
            }
        }
    }
}
```

```
    // STEP 3: Return max upper bound
    return 2;
};
```

Note 1: Terms & Techniques (Articulation Points)

While the solution above checks every cell (`O((MN)^2)`), there is a more advanced graph theory concept called **Articulation Points** (or Cut Vertices).

- **What is it?** An Articulation Point is a specific node in a graph which, if removed, increases the number of connected components.
- **Relevance:** Step 2 of our algorithm effectively hunts for an Articulation Point.
- **Optimization:** We used a brute-force approach to find it. However, **Tarjan's Algorithm** can find all articulation points in a graph in `O(V + E)` time (linear time).
- **Why didn't we use it?** For an interview, implementing Tarjan's algorithm from scratch is error-prone and complex to debug. Given the grid constraints are usually small, the brute-force `O((MN)^2)` is preferred for clarity. If the grid was massive (e.g., 1000x1000), we would *have* to use Tarjan's.

Note 2: Real-World Interview Variations

When companies like Google, Meta, or Bloomberg ask this, they often wrap it in a "system design" or "infrastructure" narrative. They aren't just checking if you know BFS; they are checking if you understand **resiliency** and **single points of failure (SPOF)**.

1. The "Server Cluster Resilience" (Google/Meta) The Question: "You have a map of servers in a data center connected by cables (the grid). What is the minimum number of servers that can crash before the cluster gets partitioned into isolated networks?"

- **The Translation:** This is exactly 1568.
- **L6 Approach:**
- Discuss **SPOF (Single Point of Failure)**. If the answer is 1, the system is fragile.
- Discuss the "Bus Factor".
- **Scale:** If the input is a massive graph (not a grid), mention Tarjan's Algorithm explicitly to detect critical servers efficiently.

2. The "Virus Containment" (Bloomberg/Healthcare Tech) The Question: "We have a biological sample on a grid. We want to cut the sample

into two separate pieces to stop growth. What is the minimum cuts needed?"

- **The Translation:** Same problem.
- **L6 Approach:**
- Here, they might ask about weighted cuts (some cells are harder to cut).
- If it's weighted, this transforms into a **Min-Cut / Max-Flow** problem, solved via the **Ford-Fulkerson** or **Dinic's Algorithm**, rather than our simple 0/1/2 logic.

3. The "Enemy Territory Infiltration" (Gaming/General) The Question: "You are playing a strategy game. You want to break the enemy's supply line. The map is a grid. Find the critical path."

- **The Translation:** Find the Articulation Point.
- **L6 Approach:**
- Focus on the visualization. If the map is dynamic (changing over time), how do you re-calculate this quickly? (Union-Find data structures might be discussed for dynamic connectivity).

924. Minimize Malware Spread

This is a classic graph theory problem that tests your ability to model "connected components" and analyze the impact of node removal on the connectivity of a network. An L5/L6 engineer at Google would look past the brute-force simulation and identify the underlying structural property: **Union-Find (Disjoint Set Union)** or **Connected Components analysis via DFS/BFS**.

Here is the deep dive solution.

---

1. Problem Explanation

Imagine a computer network where some computers are directly connected by cables. If a computer gets a virus (malware), it immediately infects all its direct neighbors. Those neighbors then infect their neighbors, and so on.

Eventually, the malware spreads to every computer in the "connected group" (component).

The Setup:

- You are given a map of the network (graph).
- You are given a list of specific computers that are *already* infected (`initial`).

The Goal:

- You are allowed to remove **exactly one** computer from the `initial` list (essentially "cleaning" it or putting a firewall around it *before* the malware starts spreading).
- You want to choose which one to remove such that the **final number of infected computers is minimized**.

Tie-Breaking Rules:

1. If removing Computer A saves 100 machines, and removing Computer B saves 50 machines, choose A.
2. If removing Computer A saves 10 machines, and removing Computer B also saves 10 machines, choose the one with the smaller ID number (e.g., if A is node `2` and B is node `5`, choose `2`).
3. **Crucial Edge Case:** If removing *any* node saves **0** machines (meaning the infection is unstoppable regardless of your choice), you must still return the node with the smallest ID from the `initial` list.
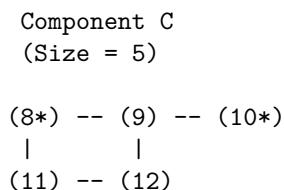
Visualizing the Problem Imagine this network. Nodes with `*` are in the `initial` infected list.

```
    Component A              Component B
    (Size = 4)               (Size = 3)

    (1*) -- (2)              (5*) -- (6)
     |                        |
    (3) -- (4)               (7)
```

Scenario 1: `initial = [1, 5]`

- **Component A:** Contains malware at Node `1`. If we leave `1` alone, nodes `1, 2, 3, 4` all get sick.
- **Component B:** Contains malware at Node `5`. If we leave `5` alone, nodes `5, 6, 7` all get sick.
- **Decision:**
- If we remove `1`: We save Component A (4 nodes).
- If we remove `5`: We save Component B (3 nodes).
- **Result:** Remove `1` because saving 4 nodes is better than saving 3.

Scenario 2: The "Double Infection" Trap

```
      Component C
      (Size = 5)

    (8*) -- (9) -- (10*)
     |        |
    (11) -- (12)
```

- **Scenario:** `initial = [8, 10]`

- **Analysis:** Both Node `8` and Node `10` are infected in the *same* group.

- If we remove 8: The malware still exists at Node 10. It spreads to `9, 12, 11` and eventually reinfects 8. **0 nodes saved.**

- If we remove 10: The malware still exists at Node 8. It spreads everywhere. **0 nodes saved.**

- **Result:** We can't save anyone. The tie-breaker rule says "return the smallest index". Return 8.

---

2. Solution Explanation (The L5/L6 Approach)

A junior engineer might try to simulate the spread for every possible removal. That is slow (or).

A senior engineer realizes we don't need to simulate the spread. We just need to count **Connected Components**.

The Algorithm

1. **Identify Components:** Use BFS, DFS, or Union-Find to group all nodes into "Connected Components".
2. **Calculate Sizes:** Count how many nodes are in each component.
3. **Count Infection Density:** For each component, count how many nodes from the `initial` list are inside it.
4. **Evaluate Candidates:**

- Look at every node in `initial`.
- Find the component this node belongs to.
- **The "Golden Rule":**
- If the component has **MORE THAN 1** malware node: Removing this specific node is useless. The other malware node(s) will infect the whole component anyway. **Savings = 0**.
- If the component has **EXACTLY 1** malware node: Removing this node saves the *entire* component. **Savings = Size of Component**.

5. **Select Winner:** Pick the node that yields the highest savings. If there's a tie (or if all savings are 0), pick the node with the smallest index.

Step-by-Step Visualization **Input: graph**: Represented below `initial`: [0, 5, 9] (Nodes 0, 5, and 9 are infected)

```
Component #1          Component #2          Component #3
Total Nodes: 4        Total Nodes: 3        Total Nodes: 2
Infected: [0, 5]      Infected: [9]         Infected: []
  (2 sources)           (1 source)            (0 sources)


  (0*)---(1)            (9*)---(6)            (7)---(8)
```

```
    |        |                |
   (5*)---(4)               (2)
```

Step 1: Analyze Component #1

- It contains nodes {0, 1, 4, 5}. Size = 4.
- It contains infected nodes {0, 5}. Count = 2.
- **Logic:** Since Count > 1, removing 0 changes nothing (virus spreads from 5). Removing 5 changes nothing (virus spreads from 0).
- **Potential Savings:** 0.

Step 2: Analyze Component #2

- It contains nodes {2, 6, 9}. Size = 3.
- It contains infected nodes {9}. Count = 1.
- **Logic:** Since Count == 1, removing 9 eliminates the *only* source of infection for this group.
- **Potential Savings:** 3 (The whole component is safe).

Step 3: Analyze Component #3

- It contains nodes {7, 8}. Size = 2.
- It contains infected nodes {}. Count = 0.
- **Logic:** No action needed.

Step 4: Final Decision

- Node 0: Savings = 0.
- Node 5: Savings = 0.
- Node 9: Savings = 3.
- **Winner:** Return 9.

(Note: If Node 9 didn't exist in `initial`*, all savings would be 0. We would then return* `min(0, 5) = 0`*.)*

---

3. Time and Space Complexity Analysis

Let **N** be the number of nodes in the graph.

Time Complexity: `O(N^2)` **Space Complexity:** `O(N)`

Derivation (Visual Text):

```
 PHASE 1: FIND COMPONENTS (DFS/BFS)
 +--------------------------------+
 | We visit every node to label it. |
 | In an Adjacency Matrix:        |
 | For each node (N), we check all  |
 | other nodes (N) to find neighbors.|
 |                                |
```

260

```
| Cost: N * N iterations          |
+---------------------------------+
          |
          v
PHASE 2: COUNT INFECTED PER COMPONENT
+---------------------------------+
| Iterate through the 'initial' list|
| (size K). Look up component ID.  |
|                                  |
| Cost: K operations (Where K <= N) |
+---------------------------------+
          |
          v
PHASE 3: FIND MAX SAVINGS
+---------------------------------+
| Iterate through 'initial' list   |
| again. Check if count == 1.      |
|                                  |
| Cost: K operations               |
+---------------------------------+


TOTAL TIME  = N^2 + K + K  approx O(N^2)
TOTAL SPACE = Array to store Component IDs for each node (Size N) + Recursion Stack
```

---

4. Solution Code

We will use **DFS** to find components because it is intuitive to implement.

Python Solution

```python
class Solution:
    def minMalwareSpread(self, graph: list[list[int]], initial: list[int]) -> int:
        N = len(graph)

        # 1. COLORING COMPONENTS
        # We assign a unique ID (color) to each connected component.
        # colors[i] = the component ID that node i belongs to.
        colors = {}
        component_id = 0

        def dfs(node, c_id):
            colors[node] = c_id
            for neighbor, is_connected in enumerate(graph[node]):
                if is_connected == 1 and neighbor not in colors:
                    dfs(neighbor, c_id)
```

```python
    # Standard DFS to find all components
    for i in range(N):
        if i not in colors:
            dfs(i, component_id)
            component_id += 1


    # 2. CALCULATE SIZES
    # component_size[c_id] = total number of nodes in that component
    from collections import Counter
    component_size = Counter(colors.values())


    # 3. COUNT MALWARE PER COMPONENT
    # malware_count[c_id] = how many initial malware nodes are in this component
    malware_count = Counter()
    for node in initial:
        malware_count[colors[node]] += 1


    # 4. FIND THE BEST NODE TO REMOVE
    initial.sort() # Sort to satisfy the tie-breaker (smallest index)

    best_node = initial[0] # Default fallback
    max_saved = -1          # Start lower than 0 to ensure we update if we find any savin

    for node in initial:
        c_id = colors[node]

        # The Magic Logic:
        # We can only save a component if this is the ONLY malware node in it.
        if malware_count[c_id] == 1:
            saved = component_size[c_id]

            # If this saves more nodes, take it.
            # If it saves the same amount, we prefer the smaller index.
            # Since we sorted 'initial' and iterate in order, we strictly need `>`
            # to update, effectively keeping the smaller index on ties.
            if saved > max_saved:
                max_saved = saved
                best_node = node

    # EDGE CASE HANDLING:
    # If max_saved remains -1 (meaning every component had 2+ malware nodes),
    # we return the smallest index from initial (which is initial[0] after sort).
    if max_saved == -1:
        return initial[0]
```

```
        return best_node
```

JavaScript Solution

```javascript
/**
 * @param {number[][]} graph
 * @param {number[]} initial
 * @return {number}
 */
var minMalwareSpread = function(graph, initial) {
    const N = graph.length;
    const colors = new Map(); // Maps node ID -> Component ID
    let componentId = 0;

    // --- Helper: DFS to mark components ---
    const dfs = (node, cId) => {
        colors.set(node, cId);
        for (let neighbor = 0; neighbor < N; neighbor++) {
            // If connected and not yet visited (colored)
            if (graph[node][neighbor] === 1 && !colors.has(neighbor)) {
                dfs(neighbor, cId);
            }
        }
    };

    // 1. Identify all components
    for (let i = 0; i < N; i++) {
        if (!colors.has(i)) {
            dfs(i, componentId);
            componentId++;
        }
    }

    // 2. Calculate size of each component
    const componentSize = new Array(componentId).fill(0);
    for (let i = 0; i < N; i++) {
        const cId = colors.get(i);
        componentSize[cId]++;
    }

    // 3. Count how many malware nodes are in each component
    const malwareCount = new Array(componentId).fill(0);
    for (const node of initial) {
        const cId = colors.get(node);
        malwareCount[cId]++;
    }
```

```javascript
    // 4. Determine the best node to remove
    // Sort initial to handle the "smallest index" tie-breaker easily
    initial.sort((a, b) => a - b);

    let bestNode = initial[0];
    let maxSaved = -1;

    for (const node of initial) {
        const cId = colors.get(node);

        // We only save the component if this is the UNIQUE source of infection
        if (malwareCount[cId] === 1) {
            const saved = componentSize[cId];

            if (saved > maxSaved) {
                maxSaved = saved;
                bestNode = node;
            }
        }
    }

    return bestNode;
};
```

Note 1: Terms and Techniques

Connected Components (Disjoint Sets): This is the core concept used here. It basically implies looking at a graph not as individual nodes, but as "islands" of nodes. If you can walk from Node A to Node B, they are on the same island (Component).

- **Why it helps:** Instead of simulating the spread of a virus step-by-step (which is dynamic and messy), we treat the spread as a static property: "If the virus lands on this island, the whole island sinks." This transforms the problem from a simulation problem to a counting problem.

Note 2: Real World & Interview Context (Google, Meta, Bloomberg)

This question is a favorite because it mimics "Cascading Failure" scenarios.

1. Google: Network Reliability (The "Kill Switch" Question) Context: Google has massive data centers. If a specific cluster of servers (a component) starts failing due to a bad config push (the "malware"), the failure

spreads to dependent servers. **Interview Variation:** "You have a map of service dependencies. Several services just alerted for critical errors. You can only rollback **one** service's config immediately. Which rollback prevents the most total services from crashing?" **Solution:** Identical to this problem. Identify the dependency graph components. Find the component where the error originated from a *single* source and rollback that one.

2. Meta: Viral Misinformation (The "Super-Spreader" Question) Context: A piece of fake news is spreading. It has been shared by several "seed" users (the `initial` array). **Interview Variation:** "We have identified 5 accounts sharing a banned URL. We have the capacity to ban only one account instantly (due to manual review constraints). Which ban prevents the content from reaching the largest number of unique users?" **Solution:** Treat users as nodes and "friendships" as edges. Find the user who is the *sole* bridge to a massive community (a large component). If a community has multiple people already sharing the link, banning one won't stop the community from seeing it. You look for the unique vector.

3. Bloomberg: Financial Contagion   Context: Banks lend money to each other. If Bank A fails, it defaults on loans to Bank B, causing Bank B to fail. **Interview Variation:** "We have detected liquidity stress in 3 major banks. We have enough federal reserve budget to bail out exactly one bank. Which bailout saves the maximum number of subsidiary banks from defaulting?" **Solution:** The "malware" is the default status. The graph is the lending network. You find the connected component of financial liability. You save the bank that is the single point of failure for the largest cluster of banks.

928. Minimize Malware Spread II

Here is a breakdown of **Leetcode 928: Minimize Malware Spread II** from the perspective of a Senior Staff Engineer (L6).

An L6 engineer approaches this not just as a graph traversal problem, but as a **system stability** and **Single Point of Failure (SPOF)** analysis. The core question isn't just "who gets infected," but "who is the sole cause of a cluster's infection?"

---
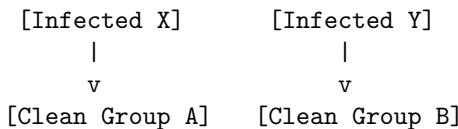
1. Problem Explanation

Imagine a computer network. Some nodes are already infected with malware (`initial` list).

- The malware spreads to any node directly connected to an infected node.
- This spread continues until no new nodes can be infected.
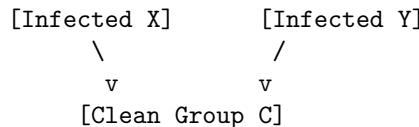
- **The Power:** You can pick exactly **one** node from the `initial` list and completely **remove it** from the network (as if you unplugged it).
- **The Goal:** Choose the node to remove that results in the **minimum** total number of infected nodes in the end.
- **Tie-breaking:** If removing Node A and Node B both save the same number of machines, remove the one with the smaller ID (index).

Key Visual Intuition: Think of the network as a set of "Clean Islands" and "Infected Sources."

```
[Infected X]        [Infected Y]
     |                   |
     v                   v
[Clean Group A]    [Clean Group B]
```

- If we remove [`Infected X`], [`Clean Group A`] is saved.
- If we remove [`Infected Y`], [`Clean Group B`] is saved.
- We want to remove the one that connects to the *biggest* Clean Group.

The "Shared Infection" Trap:

```
[Infected X]        [Infected Y]
        \               /
         v             v
        [Clean Group C]
```

- If we remove [`Infected X`], does [`Clean Group C`] get saved? **NO.**
- Why? Because [`Infected Y`] is still connected to it. The malware still flows.
- **Crucial Insight:** A clean component is only "savable" by `X` if `X` is the **only** infected node touching it.

---

2. Solution Explanation (The "Clean Component" Approach)

A junior engineer might try a "Brute Force" simulation: "Remove node 1, run BFS, count. Remove node 2, run BFS, count…"

- This is slow: O(K * N^2).
- We want something faster: O(N^2).

The L6 Strategy: Instead of simulating the infection *forward*, we analyze the topology of the *healthy* nodes.

Step 1: The "Blackout" (Identify Clean Components) First, imagine we remove *all* nodes listed in `initial` from the graph temporarily. What's left? A bunch of disconnected islands of clean nodes.

Let's use an ASCII map. `initial = [0, 1]` (Nodes 0 and 1 are bad). Nodes 2, 3, 4, 5, 6 are clean.

Original Connectivity:

```
(0)--->(2)--(3)
        |
(1)--->(4)--(5)    (6) is alone
```

After removing 0 and 1 (The Clean Components): We run Union-Find or BFS on the clean nodes only.
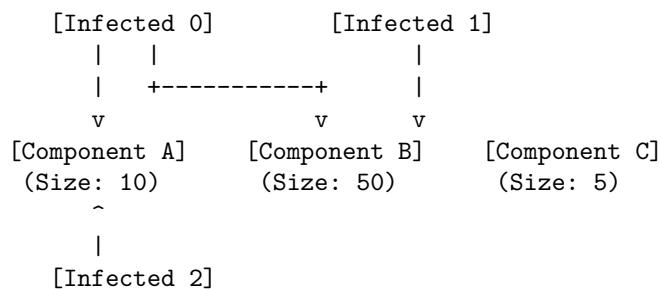
```
Component A: {2, 3}
Component B: {4, 5}
Component C: {6}
```

Step 2: The "Reconnection" (Who touches whom?) Now, we look at the `initial` nodes again and see which Clean Components they touch.

- **Node 0** connects to: Node 2 (which is inside **Component A**).
- **Node 1** connects to: Node 4 (which is inside **Component B**).

Let's imagine a more complex scenario to illustrate the logic:

```
SCENARIO: Who saves whom?


   [Infected 0]          [Infected 1]
      |   |                   |
      |   +-----------+       |
      v             v     v
[Component A]    [Component B]    [Component C]
 (Size: 10)       (Size: 50)       (Size: 5)
     ^
     |
  [Infected 2]
```

Step 3: The "Sole Survivor" Analysis We verify which components have exactly **one** infected parent.

1. **Look at Component A (Size 10):**

- Touched by: [Infected 0] AND [Infected 2].
- **Result:** It is **DOOMED**. Removing 0 won't save it (2 still infects it). Removing 2 won't save it (0 still infects it).
- *Saveable Value:* 0.

2. **Look at Component B (Size 50):**

- Touched by: [Infected 0] ONLY.
- **Result: SAVABLE.** If we remove 0, this whole component is safe.
- *Credit goes to Node 0.*

3. **Look at Component C (Size 5):**

- Touched by: [Infected 1] ONLY.
- **Result: SAVABLE.** If we remove 1, this component is safe.

- *Credit goes to Node 1.*

Step 4: The Decision

- Removing Node 0 saves: Component B (50 nodes).
- Removing Node 1 saves: Component C (5 nodes).
- Removing Node 2 saves: Nothing (because Component A is doubly infected).

Winner: Remove **Node 0**.

---

3. Time and Space Complexity Analysis

Time Complexity Derivation

Let **N** be total nodes. Let **M** be the number of edges. Let **K** be the length of `initial` array.

```
Phase 1: Identifying Clean Components (BFS/Union-Find)
[Clean Node 1] -- [Clean Node 2] ...
   |
   We iterate over the N*N adjacency matrix (or adjacency list).
   Cost: O(N^2) (Matrix) or O(N + M) (List)
   Since N <= 300 in this problem, N^2 is fine.
   Let's assume Matrix for worst case:
   -> O(N * N)

Phase 2: Mapping Infected Nodes to Components
   For each infected node 'u' in 'initial':
      For each neighbor 'v' of 'u':
         If 'v' is clean, find which Component it belongs to.

   In worst case, every infected node connects to every clean node.
   Cost: O(K * N)

Phase 3: Calculating Scores
   Iterate through the map of {Component -> List of Infectors}.
   Cost: O(Number of Components). Max N components.
   -> O(N)

TOTAL TIME: O(N * N)
(This is much better than running a BFS for every removal, which would be O(K * N^2))
```

Space Complexity Derivation

```
1. 'clean' array (boolean) to mark clean nodes: O(N)
2. 'component_id' array (maps node -> component ID): O(N)
3. 'component_size' map (maps ID -> size): O(N)
```

4. 'infected_neighbors' map (maps Component ID -> Set of Infectors): O(N)

TOTAL SPACE: O(N)

---

4. Solution Code

Python Solution

```python
from collections import defaultdict, deque

class Solution:
    def minMalwareSpread(self, graph: list[list[int]], initial: list[int]) -> int:
        N = len(graph)
        initial_set = set(initial)
        clean_nodes = [i for i in range(N) if i not in initial_set]

        # ------------------------------------------------------------
        # Step 1: Find Connected Components of Clean Nodes
        # ------------------------------------------------------------
        # We use Union-Find logic or simple BFS/DFS labeling.
        # component_map: node_index -> component_id
        component_map = {}
        # size_map: component_id -> number of nodes in that component
        size_map = defaultdict(int)
        component_id_counter = 0

        def bfs_mark_component(start_node, comp_id):
            q = deque([start_node])
            component_map[start_node] = comp_id
            count = 0
            while q:
                node = q.popleft()
                count += 1
                for neighbor, is_connected in enumerate(graph[node]):
                    if is_connected and neighbor not in initial_set and neighbor not in comp
                        component_map[neighbor] = comp_id
                        q.append(neighbor)
            return count

        for node in clean_nodes:
            if node not in component_map:
                size = bfs_mark_component(node, component_id_counter)
                size_map[component_id_counter] = size
                component_id_counter += 1
```

```python
# --------------------------------------------------------
# Step 2: Determine which Infected Nodes touch which Components
# --------------------------------------------------------
# comp_to_infectors: maps a Component ID to a Set of infected nodes touching it
comp_to_infectors = defaultdict(set)

for u in initial:
    for v, is_connected in enumerate(graph[u]):
        # If u is connected to v, and v is part of a clean component
        if is_connected and v in component_map:
            c_id = component_map[v]
            comp_to_infectors[c_id].add(u)

# --------------------------------------------------------
# Step 3: Calculate "Saved Score" for each Infected Node
# --------------------------------------------------------
# count_saved[u] stores how many nodes are saved if we remove u
count_saved = defaultdict(int)

for c_id, infectors in comp_to_infectors.items():
    # CRITICAL LOGIC: A component is saved ONLY if it has exactly 1 infector
    if len(infectors) == 1:
        sole_infector = list(infectors)[0]
        count_saved[sole_infector] += size_map[c_id]

# --------------------------------------------------------
# Step 4: Pick the winner
# --------------------------------------------------------
# We want to maximize saved nodes.
# If ties, pick smallest index.
# If no nodes can save anyone (count_saved is empty or all 0),
# we return the smallest index from 'initial' to minimize cost strictly by index.

if not count_saved:
    return min(initial)

best_node = -1
max_saved = -1

# We must iterate specifically through 'initial' to handle the tie-breaking
# logic correctly (smallest index preferred)
initial.sort()

for node in initial:
    saved = count_saved[node]
    if saved > max_saved:
```

```
                    max_saved = saved
                    best_node = node
            elif saved == max_saved:
                # Since we iterate sorted 'initial', the first one we see is smaller
                if best_node == -1:
                    best_node = node


        return best_node
```

Javascript Solution

```javascript
/**
 * @param {number[][]} graph
 * @param {number[]} initial
 * @return {number}
 */
var minMalwareSpread = function(graph, initial) {
    const N = graph.length;
    const initialSet = new Set(initial);

    // Arrays to keep track of components
    // componentMap[i] = ID of the component node 'i' belongs to
    const componentMap = new Array(N).fill(-1);
    // sizeMap[id] = size of component 'id'
    const sizeMap = new Map();
    let componentIdCounter = 0;

    // Helper: BFS to find component size and mark nodes
    const bfsMarkComponent = (startNode, compId) => {
        const queue = [startNode];
        componentMap[startNode] = compId;
        let size = 0;
        let head = 0; // Optimization: simple pointer for queue

        while(head < queue.length) {
            const node = queue[head++];
            size++;

            for(let neighbor = 0; neighbor < N; neighbor++) {
                // If connected, is 'clean', and not visited yet
                if(graph[node][neighbor] === 1 &&
                    !initialSet.has(neighbor) &&
                    componentMap[neighbor] === -1) {

                    componentMap[neighbor] = compId;
                    queue.push(neighbor);
```

```javascript
            }
        }
    }
    return size;
};

// 1. Identify Clean Components
for(let i = 0; i < N; i++) {
    if(!initialSet.has(i) && componentMap[i] === -1) {
        const size = bfsMarkComponent(i, componentIdCounter);
        sizeMap.set(componentIdCounter, size);
        componentIdCounter++;
    }
}

// 2. Map Components to their Infectors
// Key: Component ID, Value: Set of Infected Nodes touching it
const compToInfectors = new Map();

for(const u of initial) {
    for(let v = 0; v < N; v++) {
        // Check connections from infected node 'u' to clean node 'v'
        if(graph[u][v] === 1 && !initialSet.has(v)) {
            const cId = componentMap[v];
            if(cId !== -1) {
                if(!compToInfectors.has(cId)) {
                    compToInfectors.set(cId, new Set());
                }
                compToInfectors.get(cId).add(u);
            }
        }
    }
}

// 3. Score the Infected Nodes
const countSaved = new Array(N).fill(0);

for(const [cId, infectors] of compToInfectors) {
    // Only components with EXACTLY ONE infector contribute to the score
    if(infectors.size === 1) {
        const soleInfector = infectors.values().next().value;
        countSaved[soleInfector] += sizeMap.get(cId);
    }
}

// 4. Find Max
```

```
    initial.sort((a, b) => a - b);

    let bestNode = initial[0];
    let maxSaved = -1;

    for(const node of initial) {
        const saved = countSaved[node];
        if(saved > maxSaved) {
            maxSaved = saved;
            bestNode = node;
        }
    }

    return bestNode;
};
```

New Terms / Concepts

Single Point of Failure (SPOF): In our solution, a clean component that is connected to *only one* infected node represents a system with a Single Point of Failure relative to the infection. If we cut that one link (remove that infected node), the component survives. If a component is connected to *two* infected nodes, it has redundancy in its infection sources—removing one changes nothing. This concept is vital in System Design for reliability.

Real World Interview Variations

Top-tier companies (Google, Meta, Bloomberg) rarely ask the "Malware" question with this exact virus phrasing. They wrap it in domain-specific scenarios:

1. The "Datacenter Power Grid" (Google / Meta)

- **Scenario:** You have Datacenters (Clean Nodes) and Power Stations (Initial Nodes). If a Power Station fails (is removed), which Datacenters lose power?
- **Twist:** A Datacenter has backup power if it connects to *at least one other* active Power Station.
- **Goal:** Find the Power Station that, if it goes down, causes the *most* Datacenters to go offline. (This is the inverse of saving nodes, but the logic is identical: find the SPOF).

2. The "Influencer Reach" (Meta / TikTok)

- **Scenario:** You have a social graph. "Initial" nodes are influencers paid to promote a product.

- **Twist:** A regular user sees the ad if they follow *at least one* influencer.
- **Goal:** We have budget to hire one *more* influencer, or fire one. Which influencer is providing the *unique* audience?
- **Solving it:** You are looking for the "Exclusive Audience" of an influencer—the sub-graph of users who only follow THAT specific influencer and no one else.

3. **The "Supply Chain Collapse" (Bloomberg)**

- **Scenario:** Factories depend on raw material Suppliers.
- **Goal:** Identify the Supplier who is the sole provider for the largest cluster of factories. If this supplier goes bankrupt (is removed), those factories halt.
- **Solving it:** Treat Suppliers as `initial` and Factories as `clean nodes`. Use the exact Component + Unique Parent approach described above to find the most critical supplier.

815. Bus Routes

Here is a breakdown of how a senior engineer would approach **LeetCode 815. Bus Routes**.

1. Problem Explanation

Imagine a city with several bus lines. Each bus line follows a specific loop of bus stops.

- **Input:**

- `routes`: A list of lists. `routes[i]` is the list of stops the `i-th` bus visits.

- `source`: The bus stop where you start.

- `target`: The bus stop where you want to go.

- **Goal:** Find the **minimum number of buses** you need to take to get from `source` to `target`.

- **Constraint:** You can transfer between buses at common bus stops. If it's impossible, return `-1`.
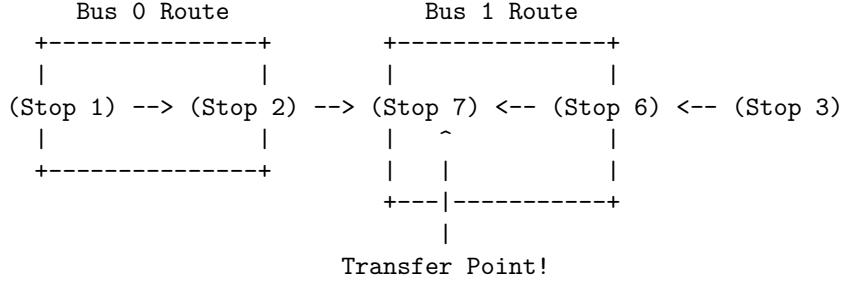
Why is this tricky? This isn't a simple "shortest distance" (kilometers) problem. It is a "shortest path" problem where the "cost" is the number of **bus transfers**, not the number of stops passed.

Visualizing the Network:

Let's say we have two routes:

- **Bus 0:** [1, 2, 7]
- **Bus 1:** [3, 6, 7]

If we want to go from Stop 1 to Stop 6:

```
      Bus 0 Route                 Bus 1 Route
  +---------------+           +---------------+
  |               |           |               |
 (Stop 1) --> (Stop 2) --> (Stop 7) <-- (Stop 6) <-- (Stop 3)
  |               |           |   ^               |
  +---------------+           |   |               |
                             +---|-----------+
                                 |
                           Transfer Point!
```

1. Start at **Stop 1**. The only option is **Bus 0**.
2. Ride Bus 0 to **Stop 7**.
3. At Stop 7, we see it is shared by **Bus 1**. We transfer.
4. Ride Bus 1 to **Stop 6**.
5. **Total:** 2 Buses.

---

2. Solution Explanation

The L5/L6 Perspective: Modeling the Graph

A junior engineer might try to build a graph where every **stop** is a node and an edge exists if they are adjacent in a route. This is inefficient because a single bus route with 100 stops would create 100 edges, and we don't care about how many stops we ride, only how many *buses* we enter.

The Insight: We are looking for the shortest path in an unweighted graph (since each bus counts as "1" cost). The standard algorithm for this is **Breadth-First Search (BFS)**.

We need to preprocess the data to make lookups fast. We need to know: *"If I am at Stop X, which buses can I take?"*

Step 1: Build an Inverted Index (Adjacency List) We map `Stop -> [List of Bus IDs]`.

Example: `routes = [[1, 2, 7], [3, 6, 7]]` (Bus 0 and Bus 1)

```
Stop Graph (Map):
-----------------
Stop 1: [Bus 0]
Stop 2: [Bus 0]
Stop 7: [Bus 0, Bus 1]  <-- This is a key transfer hub
Stop 3: [Bus 1]
Stop 6: [Bus 1]
```

Step 2: Breadth-First Search (BFS)

We perform a BFS starting from the `source` stop.

- **Queue:** Stores the current stop we are at.
- **Visited Sets:**
- `visited_stops`: To ensure we don't process a stop twice.
- `visited_buses`: **Crucial Optimization.** Once we ride a bus (e.g., Bus 0), we can reach *all* its stops. There is no point in riding Bus 0 again later in the search.

Detailed Walkthrough with ASCII Visualization:

`routes = [[1, 2, 7], [3, 6, 7]]` source = 1, `target` = 6

State 0: Initialization

- Queue: `[1]` (Start at Stop 1)
- Bus Count: `0`

Level 1 Expansion (Taking the 1st Bus)

- Pop `1`.

- Look at graph: Stop `1` is served by **Bus 0**.

- We haven't ridden Bus 0 yet. Let's "ride" it.

- Bus 0 goes to: `[1, 2, 7]`.

- Is 1 target? No. Push to Q.

- Is 2 target? No. Push to Q.

- Is 7 target? No. Push to Q.

- Mark **Bus 0** as visited.

```
    [Bus 0 is now Active]
           |
           v
Q: [1, 2, 7]
Bus Count: 1
```

Level 2 Expansion (Transferring to 2nd Bus)

- We process the queue.
- Pop 1: Already visited stops on this bus, nothing new.
- Pop 2: Served by Bus 0 (Visited). No new paths.
- Pop 7: Served by **Bus 0** (Visited) and **Bus 1** (Unvisited!).
- We haven't ridden Bus 1 yet. Let's "ride" it.
- Bus 1 goes to: `[3, 6, 7]`.
- Is 3 target? No. Push to Q.
- Is 6 target? **YES! Found it.**

```
    [Bus 1 is now Active via Stop 7]
           |
```

```
                 v
Q: [3, 6, 7]
Bus Count: 2 (Return this)
```

---

3. Time and Space Complexity Analysis

Definitions:

- `N`: The number of buses (routes).
- `R`: The maximum number of stops in a single route.
- `S`: The total number of stops across all routes (sum of lengths of all route lists).

Space Complexity Derivation:

We build a map: `Stop -> [Bus IDs]`. Every stop in the input `routes` is added to this map exactly once.

```
Stop Map Storage:
-----------------
Entry for Stop A: [Bus 0]
Entry for Stop B: [Bus 0, Bus 1]
...
Total entries across all lists = Total number of stops in input.
Total Space = O(S)
```

- **Space Complexity:** `O(S)`

Time Complexity Derivation:

We use BFS. In a standard BFS on a graph with Vertices `V` and Edges `E`, time is `O(V + E)`. However, our "Edges" are implicit. Let's look at the operations:

1. **Graph Building:** We iterate through every stop in `routes` to build the map. Cost: `O(S)`
2. **BFS Traversal:**

- Every **Bus** is processed (added to queue logic) at most **once** (due to `visited_buses` set).
- When a bus is processed, we iterate through all its **stops**.
- Therefore, every stop in the input `routes` is touched exactly once during the inner loop of the BFS.

```
BFS Operations Visualized:
------------------------
Loop over Bus 0: Process Stop 1, Stop 2, Stop 7   (Cost: Length of Bus 0)
Loop over Bus 1: Process Stop 3, Stop 6, Stop 7   (Cost: Length of Bus 1)
...
```

```
Total BFS work = Sum of lengths of all buses
               = O(S)
```

- **Time Complexity:** `O(S)`

Note: If `S` *is not given, and we usually assume* `N` *routes with* `N` *stops, the worst case is* `O(N^2)`*. But* `O(S)` *is the precise L5/L6 answer.*

---

4. Solution Code

Python Solution

```python
from collections import deque, defaultdict

def numBusesToDestination(routes, source, target):
    # Edge case: If start is destination, we need 0 buses.
    if source == target:
        return 0

    # 1. Build the Graph (Inverted Index)
    # Map: stop_id -> list of bus_route_indices
    stop_to_routes = defaultdict(list)

    for bus_id, stops in enumerate(routes):
        for stop in stops:
            stop_to_routes[stop].append(bus_id)

    # 2. BFS Initialization
    # We use a queue for BFS: stores (current_stop, buses_taken)
    queue = deque([(source, 0)])

    # Track visited distinct entities to avoid cycles and redundant work
    # We track visited BUSES and visited STOPS.
    # Actually, tracking visited BUSES is the most critical for performance
    # to avoid iterating the same route list multiple times.
    visited_buses = set()
    visited_stops = set()
    visited_stops.add(source)

    while queue:
        current_stop, bus_count = queue.popleft()

        # Check all buses that pass through this stop
        for bus_id in stop_to_routes[current_stop]:
            if bus_id in visited_buses:
                continue
```

```python
            # We are now 'riding' this bus. Mark it as seen.
            visited_buses.add(bus_id)

            # Check all stops this bus goes to
            for next_stop in routes[bus_id]:
                if next_stop == target:
                    return bus_count + 1

                if next_stop not in visited_stops:
                    visited_stops.add(next_stop)
                    queue.append((next_stop, bus_count + 1))

    # If queue is empty and target not found
    return -1
```

JavaScript Solution

```javascript
/**
 * @param {number[][]} routes
 * @param {number} source
 * @param {number} target
 * @return {number}
 */
var numBusesToDestination = function(routes, source, target) {
    if (source === target) return 0;

    // 1. Build Graph: Stop -> [Bus Indices]
    // Using a Map for efficient lookups.
    const stopToRoutes = new Map();

    for (let busId = 0; busId < routes.length; busId++) {
        for (const stop of routes[busId]) {
            if (!stopToRoutes.has(stop)) {
                stopToRoutes.set(stop, []);
            }
            stopToRoutes.get(stop).push(busId);
        }
    }

    // 2. BFS Initialization
    // Queue stores stop IDs. We track 'level' (bus count) via loop depth
    // or by storing [stop, count] tuples. Here we use loop depth for clarity.
    let queue = [source];
    let visitedBuses = new Set();
    let visitedStops = new Set();
```

```javascript
        visitedStops.add(source);

    let busCount = 0;

    while (queue.length > 0) {
        // Increment bus count for this level of transfers
        busCount++;

        const size = queue.length;
        for (let i = 0; i < size; i++) {
            const currentStop = queue.shift();

            // Get all buses passing through this stop
            const buses = stopToRoutes.get(currentStop) || [];

            for (const busId of buses) {
                if (visitedBuses.has(busId)) continue;

                visitedBuses.add(busId); // Mark bus as ridden

                // Check all stops on this bus
                for (const nextStop of routes[busId]) {
                    if (nextStop === target) {
                        return busCount;
                    }

                    if (!visitedStops.has(nextStop)) {
                        visitedStops.add(nextStop);
                        queue.push(nextStop);
                    }
                }
            }
        }
    }

    return -1;
};
```

Note 1: New Terms / Techniques

Inverted Index (Graph Adjacency List): Typically, graphs are represented as `Node -> Neighbors`. Here, we inverted the relationship to `Item (Stop) -> Containers (Routes)`. This is similar to how search engines work (`Word -> Documents` containing the word). It turns an expensive search `O(N*M)` into a

direct lookup `0(1)`.

Multi-Source BFS (Conceptually): While we started with one source, when we hop onto a bus, we effectively "expand" to all stops on that bus simultaneously in the next layer. You can view the entire bus route as a single "Super Node" in the graph.

Note 2: Real World & Interview Variations

Big tech companies rarely ask the exact LeetCode question. They mask it in "real world" scenarios.

1. The "Bloomberg Subway Map" Variation   The Question: "Design a feature for the NYC Subway app. Given a start station and end station, calculate the route with the minimum number of line transfers." **How to Solve:** This is identical to Bus Routes.

- **Stops:** Subway Stations.
- **Routes:** Subway Lines (A train, 4 train, etc.).
- **Optimization:** Bloomberg interviewers often care about memory. They might ask, "What if the graph is too big for RAM?"
- *Answer:* You would store the `Stop -> Line` map in a distributed Key-Value store (like DynamoDB or Redis). You would perform the BFS by querying the database.

2. The "Google Build System" Variation   The Question: "You have a list of libraries. Each library depends on a set of other files. If you change a core file, how many 'layers' of dependencies up the chain need to be rebuilt?" **How to Solve:** This is a BFS, but the direction is reversed.

- **Stops:** Files.
- **Routes:** Libraries (groupings of files).
- **Approach:** Start BFS from the changed file. The "Bus Count" becomes the "Dependency Depth". This ensures you rebuild the minimum necessary layers.

3. The "Meta (Facebook) Introduction" Variation   The Question: "You are User A. You want to be introduced to User B. You can only be introduced by someone who shares a 'Group' with you. What is the minimum number of Groups you need to hop through?" **How to Solve:**

- **Stops:** People (Users).
- **Routes:** Facebook Groups.
- **Logic:**

1. Start at User A.

2. Find all Groups User A is in.
3. Find all Users in those Groups.
4. Repeat until User B is found.

- *Constraint:* Meta cares about scale. Since a group can have 1 million users, iterating all users in a group (like we did `for nextStop in routes[busId]`) is too slow.
- *Modification:* You would need **Bi-directional BFS**. Start searching from User A and User B simultaneously. When the searches collide, you sum the paths. This drastically reduces the search space.

1293. Shortest Path in a Grid with Obstacles Elimination

Here is a comprehensive breakdown of **LeetCode 1293: Shortest Path in a Grid with Obstacles Elimination**, explained from the perspective of a Senior Staff Engineer (L6) who values clarity, scalability, and deep understanding of state-space search.

1. Problem Explanation

The Core Task: You are navigating a robot through a grid (2D matrix).

- **Start:** Top-left cell (`0, 0`).
- **Target:** Bottom-right cell (`m-1, n-1`).
- **Movement:** Up, Down, Left, Right (1 step at a time).
- **The Constraint:** The grid is filled with empty floors (`0`) and walls (`1`).
- **The Superpower:** You have a tool that can eliminate (destroy) at most `k` walls.

The Goal: Find the **minimum** number of steps to reach the target. If it's impossible, return `-1`.

Visualizing the Problem Imagine a small **3x3** grid where you can break `k=1` wall.

```
Grid Map:
+---+---+---+
| S | W | W |   S = Start (0,0)
+---+---+---+   W = Wall (1)
| 0 | 0 | 0 |   0 = Empty
+---+---+---+   E = End (2,2)
| W | 0 | E |
+---+---+---+
k = 1 (We can break 1 wall)
```

Scenario A (Without the Superpower): If k=0, you must go around the walls. Path: (0,0) -> (1,0) -> (1,1) -> (1,2) -> (2,2). Steps: 4.

Scenario B (With Superpower k=1): You can smash through the wall at (0,1). Path: (0,0) -> (0,1) [Break Wall] -> (0,2) [Wall is now floor] -> (1,2) -> (2,2). Wait, is that shorter? Let's check coordinates. Actually, the most direct path often cuts straight through. (0,0) -> (0,1) [Break] -> (0,2) [Break? No, k=0 left]. If there are too many walls, the "superpower" path might still get blocked.

The challenge is deciding **when** to use your limited k. Should you break the first wall you see? Or save it for a thicker wall later?

---

2. Solution Explanation

The Engineering Intuition (Why BFS?): Whenever you see "Shortest Path" in a graph or grid where step weights are uniform (each move costs 1), your immediate reaction should be **BFS (Breadth-First Search)**.

DFS (Depth-First Search) is bad here because it explores one path to exhaustion. It might find *a* path, but not necessarily the *shortest* one without checking all possibilities. BFS explores layer-by-layer like a ripple in a pond, guaranteeing the first time you touch the target, it is via the shortest path.

The Twist (State Expansion): In a standard maze, the state is defined by your position: (row, col). In this problem, arriving at (2, 2) with 3 hammers left is **different** (and better) than arriving at (2, 2) with 0 hammers left.

Therefore, our "node" in the graph isn't just (row, col). It is: (row, col, remaining_k).

The Algorithm

1. **Queue:** Initialize a queue for BFS.
- Start with: [(0, 0, k, steps=0)].
2. **Visited Set:** To prevent cycles and redundant work.
- Crucial Optimization: We only need to revisit a cell if we arrive there with **more** k than we did previously. If I've already been to cell (3,3) with 5 hammers, there is no point processing a path that gets to (3,3) with 2 hammers.
- We track visited[row][col] = max_k_seen_so_far.
3. **Exploration:**
- Pop the current state (r, c, k, steps).
- Look at 4 neighbors (Up, Down, Left, Right).

- **Case 1 (Empty Cell):** Just walk. New state: `(new_r, new_c, k, steps+1)`.
- **Case 2 (Wall):**
- Do we have hammers left (`k > 0`)?
- If yes, smash it! New state: `(new_r, new_c, k-1, steps+1)`.
- If no, we cannot go this way.

Step-by-Step Visualization (ASCII) Let's solve the `3x3` example above. Grid: `[[0,1,1], [0,0,0], [1,0,0]]`, `k=1`.

Initial State: Queue: `[ ((0,0), k=1, steps=0) ]` Visited: `(0,0)` seen with `k=1`

Step 1 (Expand (0,0)): Neighbors of (0,0):

1. Right `(0,1)` is Wall. We have `k=1`. Smash! `->` Add `((0,1), k=0, steps=1)` to Queue.
2. Down `(1,0)` is Empty. `->` Add `((1,0), k=1, steps=1)` to Queue.

Step 2 (Process Queue Layer 1):

- Process `((0,1), k=0, steps=1)`:
- Right `(0,2)` is Wall. Need hammer. `k=0`. Cannot go.
- Down `(1,1)` is Empty. `->` Add `((1,1), k=0, steps=2)`.
- Process `((1,0), k=1, steps=1)`:
- Right `(1,1)` is Empty. `->` Add `((1,1), k=1, steps=2)`. *Note: We reached (1,1) twice. Once with k=0, once with k=1. The k=1 path is "better" or equal steps but more resources. We keep it.*

Visualizing the Layered Graph:

Think of the solution as a 3D elevator. Floor 1 is "k=1 world". Floor 0 is "k=0 world".

```
Level k=1 (Full Power)      Level k=0 (No Power)
+---+---+---+                +---+---+---+
| S | W | W |  --smash-->    | . | X | W |  (We land here at 0,1)
+---+---+---+                +---+---+---+
| 0 | 0 | 0 |                | 0 | 0 | 0 |
+---+---+---+                +---+---+---+
| W | 0 | E |                | W | 0 | E |
+---+---+---+                +---+---+---+
```

When you hit a wall and have `k`, you "fall" down a level to cross it. The goal is to reach `E` on *any* level.

---

3. Time and Space Complexity Analysis

This is the part where candidates often fail to be precise.

Time Complexity: O(M * N * K)   Why? Let's verify with an ASCII Logic Tree.

Standard BFS usually visits every node once. Here, a "node" is a unique state. Unique States = (Rows) * (Cols) * (Possible values of K)

```
Dimension 1: Rows (M)
    |
    v
Dimension 2: Cols (N)
    |
    v
Dimension 3: Obstacles Removed (0 to K)

Total States = M * N * (K+1)
```

For each state, we perform a constant number of operations (checking 4 neighbors).

Derivation:

1. Max Nodes in Queue: In the worst case, we might visit every cell with every possible remaining `k`.
2. Edges per Node: 4 (Up, Down, Left, Right).
3. Total Operations: `States * 4`.
4. Remove constants: `M * N * K`.

Note on Optimization: If `k` is large enough to walk the "Manhattan Distance" (straight line path `m + n - 2`) without caring about walls, we can return the result immediately. This caps `K` at `O(M+N)`. So strictly speaking, TC is `O(M * N * min(K, M+N))`.

Space Complexity: O(M * N * K)   We need to store the `visited` information.

```
Visited Structure (Logic):
Map or 2D Array where each cell stores "Best K Seen"

Memory usage:
+--------------------------+
| Row 0, Col 0: Best K = 5 |
| Row 0, Col 1: Best K = 4 |
| ...                      |
| Row M, Col N: Best K = 1 |
```

```
+--------------------------+
Total Cells = M * N
```

Wait, if we use a simple `visited[row][col] = best_k`, the space is `O(M * N)`. However, the **Queue** can grow. In the worst case of a BFS, the queue can hold a significant portion of the total states (the "frontier" of the search). The frontier in this 3D state space can be proportional to `O(M * N * K)`.

---

4. Solution Code

Python Solution This solution uses a `deque` for efficient $O(1)$ pops and a `visited` dictionary to track state validity.

```python
from collections import deque


def shortestPath(grid, k):
    """
    Finds the shortest path from (0,0) to (m-1, n-1) allowing k obstacle removals.
    """
    rows = len(grid)
    cols = len(grid[0])
    target = (rows - 1, cols - 1)

    # OPTIMIZATION:
    # If k is large enough to go blindly via Manhattan distance (shortest theoretical path),
    # we don't need BFS. We just walk right and down.
    # Manhattan distance = (rows - 1) + (cols - 1)
    if k >= rows + cols - 2:
        return rows + cols - 2

    # State: (row, col, remaining_k)
    # Start at 0,0 with k removals left. Steps = 0.
    queue = deque([(0, 0, k, 0)])

    # Visited set stores (row, col) -> max_k_remaining_seen
    # We only care about a new path to (r,c) if we arrive with MORE k than before.
    visited = {}
    visited[(0, 0)] = k

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)] # Right, Left, Down, Up

    while queue:
        r, c, cur_k, steps = queue.popleft()

        # Check if we reached target
```

```python
        if (r, c) == target:
            return steps

        for dr, dc in directions:
            nr, nc = r + dr, c + dc

            # Bounds check
            if 0 <= nr < rows and 0 <= nc < cols:
                new_k = cur_k - grid[nr][nc] # If grid is 1 (wall), new_k decreases by 1

                # Validity check:
                # 1. We must have non-negative k left (new_k >= 0)
                # 2. We should only process this state if it's "better" than a previous vis
                #    Better means we reached (nr, nc) with MORE remaining k.
                if new_k >= 0:
                    if (nr, nc) not in visited or new_k > visited[(nr, nc)]:
                        visited[(nr, nc)] = new_k
                        queue.append((nr, nc, new_k, steps + 1))

    return -1
```

Javascript Solution This implementation mirrors the logic but handles the queue and visited set using JS structures.

```javascript
/**
 * @param {number[][]} grid
 * @param {number} k
 * @return {number}
 */
var shortestPath = function(grid, k) {
    const rows = grid.length;
    const cols = grid[0].length;

    // OPTIMIZATION: Manhattan distance check
    if (k >= rows + cols - 2) {
        return rows + cols - 2;
    }

    // Queue for BFS: [row, col, remaining_k, steps]
    // Since JS doesn't have a native Deque, we simulate or use array (shift is O(n), be awc
    // For production/strict perf, implement a Linked List Node queue.
    // For Leetcode logic clarity, Array is acceptable if acknowledged.
    const queue = [[0, 0, k, 0]];

    // Visited Map: Key = "row,col", Value = max_k_remaining
    const visited = new Map();
```

```
    visited.set(`0,0`, k);

    const directions = [[0, 1], [0, -1], [1, 0], [-1, 0]];

    let head = 0; // Pointer for queue to simulate O(1) pop
    while (head < queue.length) {
        const [r, c, curK, steps] = queue[head++];

        if (r === rows - 1 && c === cols - 1) return steps;

        for (const [dr, dc] of directions) {
            const nr = r + dr;
            const nc = c + dc;

            if (nr >= 0 && nr < rows && nc >= 0 && nc < cols) {
                const newK = curK - grid[nr][nc];

                if (newK >= 0) {
                    const key = `${nr},${nc}`;
                    // Only add if we haven't visited this cell, OR we found a path
                    // that arrives here with MORE k remaining (better state)
                    if (!visited.has(key) || newK > visited.get(key)) {
                        visited.set(key, newK);
                        queue.push([nr, nc, newK, steps + 1]);
                    }
                }
            }
        }
    }

    return -1;
};
```

Note 1: Terms and Techniques

State-Space Expansion: Normally, BFS runs on a 2D graph where nodes are (x, y). In this problem, we performed "State-Space Expansion" by adding a third dimension k. This turns the 2D grid into a 3D graph where layers are connected by "wall-breaking" edges. This technique is standard for "Shortest Path with Constraints" problems (e.g., you have a limited fuel tank, limited keys, etc.).

Pruning (Greedy elimination of states): The logic new_k > visited[(nr, nc)] is a pruning technique. In BFS, we usually prune if we see a node *at all*. Here, we prune only if we see a node with *worse or equal resources*. This

drastically reduces the search space by cutting off inefficient paths early.

Note 2: Real World / Interview Variations

Top-tier companies (Google, Meta, Bloomberg) rarely ask the exact LeetCode question. They wrap it in a scenario. Here are the common variations:

1. The "Uber/Lyft Emergency Vehicle" Variation

- **Question:** "You are routing an ambulance through a city grid. Some streets are blocked (1s). The ambulance can drive the wrong way or cross barriers, but doing so is 'expensive' or limited. Find the route that minimizes barriers crossed, or minimizes distance given a max barrier limit."
- **Google Twist:** Instead of a hard limit `k`, they might assign a **weight** to breaking a wall. Breaking a wall costs 10 minutes; driving on road costs 1 minute.
- **Solution Strategy:** This changes from BFS to **Dijkstra's Algorithm** or **0-1 BFS**.
- State: `(row, col)`.
- Priority Queue orders by `total_cost`.
- Move to empty: cost + 1. Move to wall: cost + 10.

2. The "Bloomberg Terminal Latency" Variation

- **Question:** "Data packets need to travel from Server A to Server B in a mesh network. Some nodes are congested (1s). You can boost the packet signal to bypass congestion `k` times. What is the minimum hops?"
- **Twist:** What if the 'boost' recharges? E.g., "Every 5 steps, you regain 1 boost."
- **Solution Strategy:** State-Space Expansion again, but the transitions change.
- State: `(row, col, k)`.
- Transition: If `steps % 5 == 0`, `new_k = min(max_k, k + 1)`.
- This creates a potentially infinite graph (cycles with recharging), so the `visited` set logic must be very strict to prevent infinite loops (only revisit if `k` is strictly higher).

3. The "Meta Portal/Teleporter" Variation

- **Question:** "You are in a maze with walls. There are also 'Portal' cells. Entering a portal at `(r1, c1)` instantly transports you to `(r2, c2)`. You can also break `k` walls."
- **Solution Strategy:** This adds "Teleport Edges" to your graph.
- When expanding node `(r, c)`, check neighbors.
- ALSO check if `grid[r][c]` is a portal. If so, add neighbor (`target_r, target_c`) with cost 0 or 1 depending on rules.

- This tests your ability to model the problem as a generic Graph rather than just a Grid.