Graph

1. Number of Islands

Pattern: Graph Traversal (DFS/BFS on Grid)

Problem Statement

Given an `m x n` 2D binary grid `grid` which represents a map of `'1'`s (land) and `'0'`s (water), return the number of islands.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Sample Input & Output

```
Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 3
Explanation: One contiguous island in top-left (9 cells),
one single-cell island at (0,3), rest is water.
```

```
Input: grid = [["1"]]
Output: 1
Explanation: Single land cell → one island.
```

```
Input: grid = [
  ["0","0","0"],
  ["0","0","0"]
]
Output: 0
Explanation: No land → zero islands.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        # STEP 1: Initialize structures
        # - Track visited land via in-place mutation or visited set.
        # - Here, we mutate grid: mark visited '1' as '0' to avoid revisiting.
        if not grid or not grid[0]:
            return 0

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

        # Helper: DFS to sink the entire island
        def dfs(r, c):
            # Boundary check + water/visited check
            if (r < 0 or r >= rows or c < 0 or c >= cols
                or grid[r][c] == '0'):
                return
            # Mark as visited by turning into water
            grid[r][c] = '0'
            # Visit all 4 neighbors
            dfs(r + 1, c)
            dfs(r - 1, c)
```

```
            dfs(r, c + 1)
            dfs(r, c - 1)

        # STEP 2: Main loop / recursion
        #   - Scan every cell. When we find unvisited land ('1'),
        #     trigger DFS to "sink" the whole island and increment count.
        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == '1':
                    dfs(r, c)
                    island_count += 1

        # STEP 3: Update state / bookkeeping
        #   - island_count is updated per DFS call.
        #   - Grid is mutated in place (allowed per problem constraints).

        # STEP 4: Return result
        #   - Handles empty grid above; otherwise returns count.
        return island_count

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    grid1 = [
        ["1","1","1","1","0"],
        ["1","1","0","1","0"],
        ["1","1","0","0","0"],
        ["0","0","0","0","0"]
    ]
    print(sol.numIslands(grid1))  # Expected: 1 (not 3! - all connected)
    # Actually one contiguous island. Earlier explanation was wrong.

    #  Test 2: Edge case - single cell
    grid2 = [["1"]]
    print(sol.numIslands(grid2))  # Expected: 1

    #  Test 3: Tricky/negative - all water
    grid3 = [
        ["0","0","0"],
        ["0","0","0"]
```

```
    ]
    print(sol.numIslands(grid3))  # Expected: 0
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1** step by step:

Initial grid:

```
[
  ['1','1','1','1','0'],
  ['1','1','0','1','0'],
  ['1','1','0','0','0'],
  ['0','0','0','0','0']
]
```

Step 1: `r=0, c=0` → `grid[0][0] == '1'` → start DFS.
- Mark `grid[0][0] = '0'`.
- Recursively visit neighbors: (1,0), (-1,0) invalid, (0,1), (0,-1) invalid.

Step 2: From (0,1): mark as '0', visit (0,2), (1,1), etc.
- DFS continues, turning all connected '1's into '0's.
- Eventually, the entire top-left block (including (1,3)) becomes '0' because: - (0,3) connects down to (1,3), which connects left to (1,2)? No — (1,2) is '0'.
- But (0,3) is connected to (0,2) → yes! So it's all one island.

Result: Only **one** DFS is triggered → `island_count = 1`.

> **Correction**: The initial sample explanation mistakenly said "3 islands", but the grid actually forms **one** connected island. The correct LeetCode example with output `3` uses a different grid (e.g., with disconnected land at (0,3) and (1,3) not connected). Our test uses a fully connected top region.

Final outputs:

```
1
1
0
```

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  Each cell is visited at most once. DFS visits each land cell once, and outer loops scan all cells. Total operations scale linearly with grid size.

- **Space Complexity**: `O(m * n)`

  In worst case (all '1's), recursion depth = `m * n` (e.g., snake-like DFS path). This is the call stack size. No extra visited set used — we mutate input.

2. Flood Fill

Pattern: Graph Traversal (DFS/BFS)

Problem Statement

An image is represented by an `m x n` integer grid `image` where `image[i][j]` represents the pixel value of the image.

You are also given three integers `sr`, `sc`, and `color`. You should perform a **flood fill** on the image starting from the pixel `image[sr][sc]`.

To perform a flood fill, consider the starting pixel, plus any pixels connected **4-directionally** to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (and so on), and change their color to `color`.

Return the modified image after performing the flood fill.

Sample Input & Output

```
Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
Explanation: From the center (1,1), all connected 1s are changed to 2.
```

```
Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0
Output: [[0,0,0],[0,0,0]]
Explanation: No change - new color is same as original.
```

```
Input: image = [[1]], sr = 0, sc = 0, color = 3
Output: [[3]]
Explanation: Single pixel updated.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def floodFill(
        self, image: List[List[int]], sr: int, sc: int, color: int
    ) -> List[List[int]]:
        # STEP 1: Initialize structures
        #   - Store original color to avoid infinite loops
        #   - If new color == original, return early (no work needed)
        original_color = image[sr][sc]
        if original_color == color:
            return image

        rows, cols = len(image), len(image[0])

        # STEP 2: Main loop / recursion
        #   - Use DFS via recursion to visit 4-directional neighbors
        #   - Invariant: only pixels with original_color are processed
        def dfs(r, c):
            # Base case: out of bounds or wrong color
            if (
                r < 0 or r >= rows or
                c < 0 or c >= cols or
```

```python
                image[r][c] != original_color
            ):
                return

            # STEP 3: Update state / bookkeeping
            #   - Paint current pixel
            #   - Recurse to neighbors (up, down, left, right)
            image[r][c] = color
            dfs(r - 1, c)  # up
            dfs(r + 1, c)  # down
            dfs(r, c - 1)  # left
            dfs(r, c + 1)  # right

        dfs(sr, sc)

        # STEP 4: Return result
        #   - Image is modified in-place; return it
        return image

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    img1 = [[1,1,1],[1,1,0],[1,0,1]]
    result1 = sol.floodFill(img1, 1, 1, 2)
    expected1 = [[2,2,2],[2,2,0],[2,0,1]]
    assert result1 == expected1, f"Test 1 failed: got {result1}"
    print("  Test 1 passed")

    #  Test 2: Edge case - same color
    img2 = [[0,0,0],[0,0,0]]
    result2 = sol.floodFill(img2, 0, 0, 0)
    expected2 = [[0,0,0],[0,0,0]]
    assert result2 == expected2, f"Test 2 failed: got {result2}"
    print("  Test 2 passed")

    #  Test 3: Tricky/negative - single pixel
    img3 = [[1]]
    result3 = sol.floodFill(img3, 0, 0, 3)
    expected3 = [[3]]
    assert result3 == expected3, f"Test 3 failed: got {result3}"
```

```
    print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**:
`image = [[1,1,1],[1,1,0],[1,0,1]]`, `sr=1`, `sc=1`, `color=2`

Initial state:
- `original_color = image[1][1] = 1`
- `color = 2` → different → proceed
- `rows = 3`, `cols = 3`

Call `dfs(1, 1)`:
- `(1,1)` is in bounds and `image[1][1] == 1` → valid
- Set `image[1][1] = 2` → image now:
`[[1,1,1],     [1,2,0],     [1,0,1]]` - Recurse to 4 neighbors:

1. **`dfs(0,1)`** (up):

   - Valid (value = 1) → set to 2

   - Image: row 0 becomes `[1,2,1]`

   - Recurse from (0,1):
     - Up: (-1,1) → invalid

     - Down: (1,1) → now 2   1 → skip

     - Left: (0,0) → value=1 → set to 2
       * From (0,0): up/down invalid; right=(0,1)=2 (skip); left invalid

     - Right: (0,2) → value=1 → set to 2
       * From (0,2): neighbors checked → no new changes

   - Row 0 now: `[2,2,2]`

2. **`dfs(2,1)`** (down):

   - `image[2][1] = 0` 1 → return immediately

3. **`dfs(1,0)`** (left):

   - `image[1][0] = 1` → set to 2

   - Image row 1: `[2,2,0]`

   - From (1,0):
     - Up: (0,0) = 2 → skip

     - Down: (2,0) = 1 → valid → set to 2
       * From (2,0): neighbors → (1,0)=2, (2,1)=0 → stop

   - Row 2 becomes: `[2,0,1]`

4. **`dfs(1,2)`** (right):

   - `image[1][2] = 0` → skip

Final image:

```
[[2,2,2],
 [2,2,0],
 [2,0,1]]
```

Matches expected output

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  In worst case, we visit every pixel once (e.g., entire grid same color). Each pixel processed once via DFS.

- **Space Complexity**: `O(m * n)`

  Recursion stack depth can be up to `m * n` in worst case (e.g., snake-like path). No extra data structures beyond input.

3. 01 Matrix

Pattern: Multi-source BFS (Breadth-First Search)

Problem Statement

> Given an `m x n` binary matrix `mat`, return the distance of the nearest `0` for each cell.
> The distance between two adjacent cells is `1`.
>
> **Clarification**:
> - "Adjacent" means up/down/left/right (4-directional).
> - If a cell is `0`, its distance is `0`.
> - All other cells must compute shortest distance to any `0`.

Sample Input & Output

```
Input: [[0,0,0],[0,1,0],[0,0,0]]
Output: [[0,0,0],[0,1,0],[0,0,0]]
Explanation: All 1s are already adjacent to 0s; distance = 1.
```

```
Input: [[0,0,0],[0,1,0],[1,1,1]]
Output: [[0,0,0],[0,1,0],[1,2,1]]
Explanation: Bottom row: middle cell is 2 steps from nearest 0.
```

```
Input: [[1]]
Output: [[-1]] →  Wait! Actually: [[0]] if input were [[0]],
but [[1]] has no 0!
Correction: Input must contain at least one 0 per problem constraints.
So edge case: [[1,0]] → Output: [[1,0]]
```

Final test cases: - Normal: `[[0,1,1],[1,1,1],[1,1,0]]` → `[[0,1,2],[1,2,1],[2,1,0]]` - Edge: `[[0]]` → `[[0]]` - Tricky: `[[1,0,1],[1,1,1],[1,1,1]]` → `[[1,0,1],[2,1,2],[3,2,3]]`

LeetCode Editorial Solution + Inline Tests

```
from collections import deque
from typing import List

class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        # STEP 1: Initialize structures
        #   - Use multi-source BFS: start from all 0s simultaneously.
        #   - dist matrix tracks shortest distance; init with 0s at 0-cells,
        #     and -1 (or large) for unvisited 1s.
        m, n = len(mat), len(mat[0])
        dist = [[-1] * n for _ in range(m)]
        q = deque()

        # Enqueue all 0s as starting points (distance = 0)
        for i in range(m):
            for j in range(n):
                if mat[i][j] == 0:
                    dist[i][j] = 0
                    q.append((i, j))

        # STEP 2: Main loop / recursion
        #   - BFS guarantees shortest path in unweighted grid.
        #   - Process neighbors level-by-level (distance increases by 1).
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        while q:
            x, y = q.popleft()

            # STEP 3: Update state / bookkeeping
            #   - For each neighbor, if unvisited, set distance = current + 1
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if (0 <= nx < m and 0 <= ny < n and
                    dist[nx][ny] == -1):  # not visited
                    dist[nx][ny] = dist[x][y] + 1
                    q.append((nx, ny))

        # STEP 4: Return result
        #   - All cells guaranteed reachable (problem ensures  1 zero)
        return dist
```

```python
# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    mat1 = [[0,1,1],[1,1,1],[1,1,0]]
    out1 = sol.updateMatrix(mat1)
    expected1 = [[0,1,2],[1,2,1],[2,1,0]]
    assert out1 == expected1, f"Test 1 failed: got {out1}"
    print("  Test 1 passed")

    #  Test 2: Edge case - single zero
    mat2 = [[0]]
    out2 = sol.updateMatrix(mat2)
    expected2 = [[0]]
    assert out2 == expected2, f"Test 2 failed: got {out2}"
    print("  Test 2 passed")

    #  Test 3: Tricky/negative - zero not at corner
    mat3 = [[1,0,1],[1,1,1],[1,1,1]]
    out3 = sol.updateMatrix(mat3)
    expected3 = [[1,0,1],[2,1,2],[3,2,3]]
    assert out3 == expected3, f"Test 3 failed: got {out3}"
    print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3**:
Input: `[[1,0,1],[1,1,1],[1,1,1]]`

Initial state: - `mat = [[1,0,1], [1,1,1], [1,1,1]]` - `dist` initialized as `[[-1,0,-1], [-1,-1,-1], [-1,-1,-1]]` - Queue `q = deque([(0,1)])` ← only the `0` at (0,1)

Step 1: Pop `(0,1)` from queue.
Check 4 neighbors: - (1,1): valid, unvisited → set `dist[1][1] = 0+1 = 1`, enqueue `(1,1)` - (0,0): valid, unvisited → `dist[0][0] = 1`, enqueue `(0,0)` - (0,2): valid, unvisited → `dist[0][2] = 1`, enqueue `(0,2)` - (-1,1): invalid → skip

Now:
`dist = [[1,0,1], [-1,1,-1], [-1,-1,-1]]`
`q = [(1,1), (0,0), (0,2)]`

Step 2: Pop `(1,1)`
Neighbors: - (2,1): unvisited → `dist[2][1] = 1+1 = 2`, enqueue - (1,0): unvisited → `dist[1][0] = 2`, enqueue - (1,2): unvisited → `dist[1][2] = 2`, enqueue - (0,1): already visited → skip

`dist = [[1,0,1], [2,1,2], [-1,2,-1]]`
`q = [(0,0), (0,2), (2,1), (1,0), (1,2)]`

Step 3: Pop `(0,0)`
Neighbors: (1,0) already set to 2 → skip; (-1,0) invalid; (0,-1) invalid; (0,1) visited.
→ No new updates.

Step 4: Pop `(0,2)`
Neighbors: (1,2) already 2 → skip; others invalid/visited.

Step 5: Pop `(2,1)`
Neighbors: - (2,0): unvisited → `dist[2][0] = 2+1 = 3`, enqueue - (2,2): unvisited → `dist[2][2] = 3`, enqueue

Now `dist = [[1,0,1], [2,1,2], [3,2,3]]`

Step 6: Pop remaining `(1,0)`, `(1,2)`, `(2,0)`, `(2,2)`
→ All their neighbors already visited or out of bounds.

Final `dist`: `[[1,0,1],[2,1,2],[3,2,3]]`

Key insight: BFS from **all zeros at once** ensures first time we reach a `1`, it's via shortest path.

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  Each cell is enqueued and dequeued at most once. We visit every cell a constant number of times (once for initialization, once in BFS). Total operations scale linearly with number of cells.

- **Space Complexity**: `O(m * n)`

  The `dist` matrix uses `O(mn)` space. The queue in worst case (e.g., half the grid is 0) may store up to `O(mn)` cells. Thus total space is `O(mn)`.

4. Rotting Oranges

Pattern: Multi-source BFS (Breadth-First Search)

Problem Statement

You are given an `m x n` grid where each cell can have one of three values:
- `0` representing an empty cell,
- `1` representing a fresh orange,
- `2` representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return the **minimum number of minutes** that must elapse until no cell has a fresh orange. If this is impossible, return `-1`.

Sample Input & Output

```
Input: [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
Explanation: All oranges rot in 4 minutes.
```

```
Input: [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange at bottom-left (2,0) never rots.
```

```
Input: [[0,2]]
Output: 0
Explanation: No fresh oranges exist initially.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import deque

class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        # STEP 1: Initialize structures
        # Use a queue to store all initially rotten oranges (multi-source BFS)
        # Track fresh count to detect if all oranges rotted
        m, n = len(grid), len(grid[0])
        queue = deque()
        fresh = 0

        # Scan grid to find rotten oranges and count fresh ones
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 2:
                    queue.append((i, j))
                elif grid[i][j] == 1:
                    fresh += 1

        # If no fresh oranges, time = 0
        if fresh == 0:
            return 0

        # STEP 2: Main loop / recursion
        # Process all currently rotten oranges level-by-level(minute-by-minute)
        # Each BFS level = 1 minute
        minutes = 0
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

        while queue and fresh > 0:
            # Process all oranges that rot at current minute
            for _ in range(len(queue)):
                x, y = queue.popleft()

                # Check 4 neighbors
                for dx, dy in directions:
                    nx, ny = x + dx, y + dy
                    # STEP 3: Update state / bookkeeping
                    #   - Only rot fresh oranges; reduce fresh count
```

```python
                    if (0 <= nx < m and 0 <= ny < n and
                        grid[nx][ny] == 1):
                        grid[nx][ny] = 2
                        fresh -= 1
                        queue.append((nx, ny))
            minutes += 1

        # STEP 4: Return result
        #   - If any fresh oranges remain, return -1
        return minutes if fresh == 0 else -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    grid1 = [[2,1,1],[1,1,0],[0,1,1]]
    print(sol.orangesRotting(grid1))  # Expected: 4

    #   Test 2: Edge case - impossible to rot all
    grid2 = [[2,1,1],[0,1,1],[1,0,1]]
    print(sol.orangesRotting(grid2))  # Expected: -1

    #   Test 3: Tricky/negative - no fresh oranges
    grid3 = [[0,2]]
    print(sol.orangesRotting(grid3))  # Expected: 0
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `grid = [[2,1,1],[1,1,0],[0,1,1]]`.

Initial state:
- Grid: `[2,1,1]` `[1,1,0]` `[0,1,1]` - `fresh = 6` (six 1s) - Queue starts with `(0,0)` (only rotten orange)

Minute 0 → Minute 1:
- Process `(0,0)`: check neighbors → `(0,1)` and `(1,0)` are fresh.

→ Rot them: set to `2`, `fresh = 4`, add to queue.
- Queue now: `[(0,1), (1,0)]`
- `minutes = 1`

Minute 1 → Minute 2:
- Process `(0,1)`: neighbors → `(0,2)` and `(1,1)` are fresh.
→ Rot them: `fresh = 2`, queue adds `(0,2), (1,1)`
- Process `(1,0)`: neighbor `(2,0)` is `0` (empty); `(1,1)` already handled.
- Queue now: `[(0,2), (1,1)]`
- `minutes = 2`

Minute 2 → Minute 3:
- Process `(0,2)`: neighbor `(1,2)` is `0` → skip.
- Process `(1,1)`: neighbor `(2,1)` is fresh → rot it.
→ `fresh = 1`, add `(2,1)` to queue.
- Queue now: `[(2,1)]`
- `minutes = 3`

Minute 3 → Minute 4:
- Process `(2,1)`: neighbor `(2,2)` is fresh → rot it.
→ `fresh = 0`, add `(2,2)`
- Queue now: `[(2,2)]`
- `minutes = 4`

Loop ends (`fresh == 0`). Return `4`.

Final output: `4`

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  We visit each cell at most once during initialization and once during BFS. Total operations scale linearly with grid size.

- **Space Complexity**: `O(m * n)`

  In worst case (all oranges rotten), the queue holds all cells. Also, we modify grid in-place (no extra space beyond queue).

5. Shortest Path to Get Food

Pattern: BFS (Breadth-First Search)

Problem Statement

You are given an `m x n` character grid `grid` representing a kitchen layout.
- `'O'` denotes an empty cell you can walk through.
- `'#'` denotes a food cell (your target).
- `'X'` denotes an obstacle (blocked).
- `'*'` denotes your starting position.

Return the **length of the shortest path** from your starting position to any food cell.
If no path exists, return `-1`.
You can move up, down, left, or right. Each move counts as 1 step.

Sample Input & Output

```
Input: grid = [["X","X","X","X","X","X"],
               ["X","*","O","O","O","X"],
               ["X","O","O","#","O","X"],
               ["X","X","X","X","X","X"]]
Output: 3
Explanation: Start at (1,1) → (1,2) → (1,3) → (2,3) (food). 3 steps.
```

```
Input: grid = [["X","X","X","X","X"],
               ["X","*","X","O","X"],
               ["X","O","X","#","X"],
               ["X","X","X","X","X"]]
Output: -1
Explanation: Food is unreachable due to surrounding walls.
```

```
Input: grid = [["*","#"]]
Output: 1
Explanation: Start adjacent to food - 1 step.
```

LeetCode Editorial Solution + Inline Tests

```
from collections import deque
from typing import List

class Solution:
    def getFood(self, grid: List[List[str]]) -> int:
        # STEP 1: Initialize structures
        #   - Find start position ('*')
        #   - Use queue for BFS: stores (row, col, steps)
        #   - Track visited cells to avoid cycles
        m, n = len(grid), len(grid[0])
        start = None
        for i in range(m):
            for j in range(n):
                if grid[i][j] == '*':
                    start = (i, j)
                    break
            if start:
                break

        if not start:
            return -1  # Should not happen per problem constraints

        queue = deque([(start[0], start[1], 0)])
        visited = [[False] * n for _ in range(m)]
        visited[start[0]][start[1]] = True

        # Directions: up, down, left, right
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        # STEP 2: Main loop / recursion
        #   - BFS explores level-by-level → guarantees shortest path
        #   - Stop when we hit a food cell ('#')
        while queue:
            r, c, steps = queue.popleft()

            # STEP 3: Update state / bookkeeping
            #   - Check if current cell is food
```

```
            if grid[r][c] == '#':
                return steps

            # Explore neighbors
            for dr, dc in directions:
                nr, nc = r + dr, c + dc

                # Validate bounds and accessibility
                if (0 <= nr < m and 0 <= nc < n and
                    not visited[nr][nc] and
                    grid[nr][nc] != 'X'):

                    visited[nr][nc] = True
                    queue.append((nr, nc, steps + 1))

        # STEP 4: Return result
        #   - If queue empties without finding food → unreachable
        return -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    grid1 = [
        ["X","X","X","X","X","X"],
        ["X","*","O","O","O","X"],
        ["X","O","O","#","O","X"],
        ["X","X","X","X","X","X"]
    ]
    assert sol.getFood(grid1) == 3, f"Expected 3, got {sol.getFood(grid1)}"

    #  Test 2: Edge case - unreachable food
    grid2 = [
        ["X","X","X","X","X"],
        ["X","*","X","O","X"],
        ["X","O","X","#","X"],
        ["X","X","X","X","X"]
    ]
    assert sol.getFood(grid2) == -1, f"Expected -1, got {sol.getFood(grid2)}"

    #  Test 3: Tricky/negative - adjacent food
```

```
grid3 = [["*","#"]]
assert sol.getFood(grid3) == 1, f"Expected 1, got {sol.getFood(grid3)}"

print("  All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1** step-by-step:

Initial grid:

```
Row 0: X X X X X X
Row 1: X * O O O X
Row 2: X O O # O X
Row 3: X X X X X X
```

Step 1: Find start → at `(1,1)`.
- Initialize `queue = [(1,1,0)]`
- `visited[1][1] = True`

Step 2: Dequeue `(1,1,0)`
- Not food (`'*'  '#'`)
- Check 4 neighbors: - `(0,1)`: 'X' → skip
- `(2,1)`: 'O' → valid → mark visited, enqueue `(2,1,1)`
- `(1,0)`: 'X' → skip
- `(1,2)`: 'O' → valid → enqueue `(1,2,1)`
- Queue now: `[(2,1,1), (1,2,1)]`

Step 3: Dequeue `(2,1,1)`
- Cell = 'O' → not food
- Neighbors: - `(1,1)`: visited → skip
- `(3,1)`: 'X' → skip
- `(2,0)`: 'X' → skip
- `(2,2)`: 'O' → enqueue `(2,2,2)`
- Queue: `[(1,2,1), (2,2,2)]`

Step 4: Dequeue `(1,2,1)`
- Cell = 'O'

- Neighbors: - `(0,2)`: ‘X’ → skip
- `(2,2)`: already enqueued (but not visited yet? Actually, it will be marked when enqueued — so skip if already visited)
→ In our code, we mark **when enqueuing**, so `(2,2)` is already visited → skip
- `(1,1)`: visited
- `(1,3)`: ‘O’ → enqueue `(1,3,2)`
- Queue: `[(2,2,2), (1,3,2)]`

Step 5: Dequeue `(2,2,2)`
- Cell = ‘O’
- Neighbors: - `(1,2)`: visited
- `(3,2)`: ‘X’
- `(2,1)`: visited
- `(2,3)`: ‘#’ → **FOOD!** → return `steps + 1 = 2 + 1 = 3`

Final output: `3`

Key insight: BFS guarantees the **first time** we reach food is via the **shortest path**.

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  In worst case, we visit every cell once. Each cell is enqueued and dequeued at most once. Grid has `m * n` cells.

- **Space Complexity**: `O(m * n)`

  The `visited` matrix uses `O(m * n)`. The BFS queue can hold up to `O(m * n)` cells in worst case (e.g., all cells are open).

6. Word Search

Pattern: Backtracking (DFS on Grid)

Problem Statement

Given an `m x n` grid of characters `board` and a string `word`, return `true` if `word` exists in the grid.
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Sample Input & Output

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],
word = "ABCCED"
Output: true
Explanation: Path: A → B → C → C → E → D (valid adjacent path without reuse)
```

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],
word = "SEE"
Output: true
Explanation: S → E → E (down then right)
```

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],
word = "ABCB"
Output: false
Explanation: After A→B→C, next B is not reachable without reusing 'B'
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        # STEP 1: Initialize structures
        #   - Store board dimensions and visited state
```

```
        rows, cols = len(board), len(board[0])
        visited = [[False] * cols for _ in range(rows)]

        # STEP 2: Main loop / recursion
        #   - Try starting DFS from every cell
        #   - If any path matches word, return True
        for r in range(rows):
            for c in range(cols):
                if self._dfs(board, word, 0, r, c, visited):
                    return True
        return False

    def _dfs(
        self,
        board: List[List[str]],
        word: str,
        idx: int,
        r: int,
        c: int,
        visited: List[List[bool]]
    ) -> bool:
        # Base case: full word matched
        if idx == len(word):
            return True

        # Boundary & validity checks
        if (
            r < 0 or r >= len(board) or
            c < 0 or c >= len(board[0]) or
            visited[r][c] or
            board[r][c] != word[idx]
        ):
            return False

        # STEP 3: Update state / bookkeeping
        #   - Mark current cell as used
        visited[r][c] = True

        # Explore 4 directions
        found = (
            self._dfs(board, word, idx + 1, r + 1, c, visited) or
            self._dfs(board, word, idx + 1, r - 1, c, visited) or
```

```python
            self._dfs(board, word, idx + 1, r, c + 1, visited) or
            self._dfs(board, word, idx + 1, r, c - 1, visited)
        )

        # Backtrack: unmark current cell
        visited[r][c] = False

        # STEP 4: Return result
        #   - Propagate match status up recursion stack
        return found

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    board1 = [
        ["A","B","C","E"],
        ["S","F","C","S"],
        ["A","D","E","E"]
    ]
    assert sol.exist(board1, "ABCCED") == True

    #   Test 2: Edge case - single letter
    board2 = [["A"]]
    assert sol.exist(board2, "A") == True

    #   Test 3: Tricky/negative - reuse not allowed
    board3 = [
        ["A","B","C","E"],
        ["S","F","C","S"],
        ["A","D","E","E"]
    ]
    assert sol.exist(board3, "ABCB") == False

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `exist(board, "SEE")` on the standard board:

Initial State:
- `board` = 3×4 grid as above
- `word = "SEE"` → length = 3
- `visited` = 3×4 grid of `False`

Step 1: Outer loops try every cell as start.
- At `(0,0)`: 'A'   'S' → skip
- …
- At `(1,0)`: 'S' == 'S' → call `_dfs(..., idx=0, r=1, c=0)`

Step 2: Inside `_dfs` at `(1,0)`, `idx=0`
- Not out of bounds
- `visited[1][0]` is `False`
- `board[1][0] == 'S' == word[0]` → OK
- Mark `visited[1][0] = True`
- Now try 4 neighbors for next char `'E'` (`idx=1`)

Step 3: Try down → `(2,0)` = 'A'   'E' → fail
Try up → `(0,0)` = 'A'   'E' → fail
Try right → `(1,1)` = 'F'   'E' → fail
Try left → invalid (c=-1) → fail
→ All fail → backtrack: set `visited[1][0] = False` → return `False`

Step 4: Continue outer loop…
At `(1,3)`: 'S' == 'S' → call `_dfs(..., idx=0, r=1, c=3)`

Step 5: In `_dfs(1,3)`, mark visited → try neighbors:
- Down: `(2,3)` = 'E' == word[1] → recurse to `idx=1`

Step 6: In `_dfs(2,3)`, mark visited → now look for `'E'` (`idx=2`)
- Up: `(1,3)` → visited → skip
- Down: invalid
- Left: `(2,2)` = 'E' == word[2] → recurse to `idx=2`

Step 7: In `_dfs(2,2)`, mark visited → now `idx=2`, next call with `idx=3`
- Base case: `idx == len(word) (3)` → return `True`

Step 8: `True` propagates up → outer function returns `True`

Final output: `True`

Key takeaway: Backtracking explores all paths but **undoes choices** (via `visited[r][c] = False`) so other paths can reuse cells.

Complexity Analysis

- **Time Complexity**: `O(m * n * 4^L)`

  For each of the `m*n` starting cells, we may explore up to 4 directions per character, and the word has length `L`. Worst-case exponential due to backtracking.

- **Space Complexity**: `O(L)`

  The recursion stack depth is at most `L` (length of word). The `visited` matrix is `O(m*n)`, but since we reuse it and it's input-sized, some consider auxiliary space as `O(L)`. However, strictly: **`O(m * n)`** due to `visited` array.
  Clarification: LeetCode typically counts **extra** space beyond input. Since `visited` is extra, space = `O(m * n)`. But in interviews, clarify assumptions.

7. Number of Connected Components in an Undirected Graph

Pattern: Graph Traversal (DFS/BFS) + Union-Find (Disjoint Set Union)

Problem Statement

> You are given an undirected graph with `n` nodes labeled from `0` to `n - 1`. The graph is represented as an integer `n` and a list of edges `edges`, where each `edges[i] = [a, b]` indicates an undirected edge between nodes `a` and `b`.
>
> Return the number of **connected components** in the graph.

Sample Input & Output

```
Input: n = 5, edges = [[0,1],[1,2],[3,4]]
Output: 2
Explanation: Nodes 0-1-2 form one component; nodes 3-4 form another.
```

```
Input: n = 5, edges = []
Output: 5
Explanation: No edges → each node is its own component.
```

```
Input: n = 1, edges = []
Output: 1
Explanation: Single node with no edges → one component.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        # STEP 1: Build adjacency list
        #   - Why? To enable efficient graph traversal.
        #   - Undirected → add both directions.
        graph = [[] for _ in range(n)]
        for a, b in edges:
            graph[a].append(b)
            graph[b].append(a)

        # STEP 2: Track visited nodes
        #   - Prevent revisiting and infinite loops.
        visited = [False] * n
        components = 0

        # STEP 3: DFS helper to mark all nodes in a component
        def dfs(node):
            visited[node] = True
            for neighbor in graph[node]:
                if not visited[neighbor]:
                    dfs(neighbor)

        # STEP 4: Iterate through all nodes
        #   - Each unvisited node starts a new component.
        for i in range(n):
```

```python
            if not visited[i]:
                dfs(i)
                components += 1

        # STEP 5: Return total count
        return components

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.countComponents(5, [[0,1],[1,2],[3,4]]) == 2

    #  Test 2: Edge case - no edges
    assert sol.countComponents(5, []) == 5

    #  Test 3: Tricky/negative - single node
    assert sol.countComponents(1, []) == 1

    print(" All tests passed!")
```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

```
We'll walk through **Test 1**: `n = 5`, `edges = [[0,1],[1,2],[3,4]]`.

1. **Build graph**:
   - Initialize `graph = [[], [], [], [], []]`
   - Process `[0,1]` → `graph[0] = [1]`, `graph[1] = [0]`
   - Process `[1,2]` → `graph[1] = [0,2]`, `graph[2] = [1]`
   - Process `[3,4]` → `graph[3] = [4]`, `graph[4] = [3]`
   - Final `graph = [[1], [0,2], [1], [4], [3]]`

2. **Initialize**:
   - `visited = [False, False, False, False, False]`
   - `components = 0`
```

```
3. **Loop over nodes**:
   - `i = 0`: not visited → start DFS
     - `dfs(0)`:
       - Mark `visited[0] = True`
       - Visit neighbor `1` → not visited → `dfs(1)`
         - Mark `visited[1] = True`
         - Neighbors: `0` (visited), `2` → `dfs(2)`
           - Mark `visited[2] = True`
           - Neighbor `1` already visited → return
       - Backtrack → DFS ends
     - `components = 1`
   - `i = 1`: already visited → skip
   - `i = 2`: already visited → skip
   - `i = 3`: not visited → start DFS
     - `dfs(3)`:
       - Mark `visited[3] = True`
       - Visit `4` → not visited → `dfs(4)`
         - Mark `visited[4] = True`
         - Neighbor `3` visited → return
     - `components = 2`
   - `i = 4`: visited → skip

4. **Return** `2`

Final `visited = [True, True, True, True, True]`
Output: `2`

---
```

Complexity Analysis

- **Time Complexity**: `O(n + e)`

  We visit each node once (`n`) and each edge twice (once per direction, but still `O(e)` total). DFS visits every reachable node/edge exactly once.

- **Space Complexity**: `O(n + e)`

  Adjacency list uses `O(n + e)` space. Recursion stack in worst case (e.g., a line graph) uses `O(n)` space. So total is `O(n + e)`.

8. Graph Valid Tree

Pattern: Graph — Union-Find / DFS Cycle Detection

Problem Statement

> You are given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges (each edge is a pair of nodes). Write a function to check whether these edges make up a valid tree.

A **valid tree** must satisfy **two conditions**:
1. There are **exactly `n - 1` edges**.
2. The graph is **fully connected and acyclic** (i.e., one connected component with no cycles).

Sample Input & Output

```
Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
Output: True
Explanation: 5 nodes, 4 edges, connected and no cycles → valid tree.
```

```
Input: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
Output: False
Explanation: Contains a cycle (1-2-3-1), so not a tree.
```

```
Input: n = 1, edges = []
Output: True
Explanation: Single node with no edges is a valid tree.
```

LeetCode Editorial Solution + Inline Tests

We'll use **Union-Find (Disjoint Set Union)** — a classic pattern for cycle detection in undirected graphs.
- If we ever try to union two nodes already in the same set → **cycle detected**.
- Also verify edge count = `n - 1`.

```
from typing import List

class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        # STEP 1: Quick edge count check
        #   - A tree must have exactly n - 1 edges
        if len(edges) != n - 1:
            return False

        # STEP 2: Initialize Union-Find parent array
        #   - Each node starts as its own parent
        parent = list(range(n))

        # Helper: Find root with path compression
        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x])  # Path compression
            return parent[x]

        # STEP 3: Process each edge
        #   - If two nodes share root → cycle → invalid
        for a, b in edges:
            root_a = find(a)
            root_b = find(b)
            if root_a == root_b:
                return False  # Cycle detected!
            parent[root_a] = root_b  # Union

        # STEP 4: Return True
        #   - Passed edge count + no cycles → valid tree
        return True

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()
```

```python
#   Test 1: Normal case
assert sol.validTree(5, [[0,1],[0,2],[0,3],[1,4]]) == True

#   Test 2: Edge case - single node
assert sol.validTree(1, []) == True

#   Test 3: Tricky/negative - cycle present
assert sol.validTree(5, [[0,1],[1,2],[2,3],[1,3],[1,4]]) == False

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `n = 5`, `edges = [[0,1],[0,2],[0,3],[1,4]]`

Initial state:
- `parent = [0, 1, 2, 3, 4]`
- Edge count = 4 → equals `5 - 1` → proceed.

Edge [0,1]:
- `find(0)` → 0, `find(1)` → 1 → different roots
- Union: set `parent[0] = 1` → `parent = [1, 1, 2, 3, 4]`

Edge [0,2]:
- `find(0)` → `find(1)` → 1; `find(2)` → 2
- Union: `parent[1] = 2` → `parent = [1, 2, 2, 3, 4]`

Edge [0,3]:
- `find(0)` → `find(1)` → `find(2)` → 2; `find(3)` → 3
- Union: `parent[2] = 3` → `parent = [1, 2, 3, 3, 4]`

Edge [1,4]:
- `find(1)` → `find(2)` → `find(3)` → 3; `find(4)` → 4
- Union: `parent[3] = 4` → `parent = [1, 2, 3, 4, 4]`

No cycles found → return `True`.

Now **Test 3**: `edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]`
- First 3 edges connect 0–1–2–3 into one component.
- When processing `[1,3]`:

- `find(1)` → root = 3 (after unions)
- `find(3)` → root = 3
- Same root → **cycle detected** → return `False` immediately.

Complexity Analysis

- **Time Complexity**: `O(n · (n))  O(n)`

  We process `n - 1` edges. Each `find` uses path compression, making amortized cost nearly constant (= inverse Ackermann function).

- **Space Complexity**: `O(n)`

  The `parent` array stores one entry per node. No recursion stack (iterative union).

9. Course Schedule

Pattern: Topological Sort (Graph — Directed Acyclic Graph / Cycle Detection)

Problem Statement

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [a_i, b_i]` indicates that you must take course `b_i` first if you want to take course `a_i`.

Return `true` if you can finish all courses. Otherwise, return `false`.

Sample Input & Output

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: To take course 1, you must first take course 0. This is valid.
```

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: Course 0 requires course 1,
and course 1 requires course 0 → cycle.
```

```
Input: numCourses = 1, prerequisites = []
Output: true
Explanation: Only one course with no prerequisites → always finishable.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]):
        # STEP 1: Build adjacency list and in-degree array
        #   - adj[i] = list of courses that depend on course i
        #   - indegree[i] = number of prerequisites for course i
        adj = [[] for _ in range(numCourses)]
        indegree = [0] * numCourses

        for course, prereq in prerequisites:
            adj[prereq].append(course)
            indegree[course] += 1

        # STEP 2: Initialize queue with all courses having no prerequisites
        #   - These can be taken immediately (indegree == 0)
        from collections import deque
        queue = deque()
        for i in range(numCourses):
```

```python
            if indegree[i] == 0:
                queue.append(i)

        # STEP 3: Process courses in topological order
        #   - For each course taken, reduce indegree of its dependents
        #   - If a dependent's indegree becomes 0, add to queue
        taken = 0
        while queue:
            curr = queue.popleft()
            taken += 1
            for neighbor in adj[curr]:
                indegree[neighbor] -= 1
                if indegree[neighbor] == 0:
                    queue.append(neighbor)

        # STEP 4: Return result
        #   - If we took all courses, no cycle exists → return True
        return taken == numCourses

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - linear dependency
    assert sol.canFinish(2, [[1,0]]) == True

    #  Test 2: Edge case - single course, no prereqs
    assert sol.canFinish(1, []) == True

    #  Test 3: Tricky/negative - cycle in graph
    assert sol.canFinish(2, [[1,0],[0,1]]) == False

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3**: `numCourses = 2`, `prerequisites = [[1,0],[0,1]]`.

1. **Initialize structures**

   - `adj = [[], []]` → two empty lists for courses 0 and 1

   - `indegree = [0, 0]`

2. **Build graph from prerequisites**

   - Process `[1,0]`: course 1 depends on 0
     → `adj[0].append(1)` → `adj = [[1], []]`
     → `indegree[1] += 1` → `indegree = [0, 1]`

   - Process `[0,1]`: course 0 depends on 1
     → `adj[1].append(0)` → `adj = [[1], [0]]`
     → `indegree[0] += 1` → `indegree = [1, 1]`

3. **Initialize queue**

   - Check `indegree[0] = 1` → skip

   - Check `indegree[1] = 1` → skip
     → `queue = deque()` (empty)

4. **Process queue**

   - Queue is empty → `while` loop never runs
     → `taken = 0`

5. **Return result**

   - `taken == numCourses` → `0 == 2` → `False`

Final output: `False` — correctly detects cycle.

Complexity Analysis

- **Time Complexity**: `O(V + E)`

  We visit each course (`V = numCourses`) once and each prerequisite edge (`E = len(prerequisites)`) once during graph building and BFS traversal.

- **Space Complexity**: `O(V + E)`

  The adjacency list stores `E` edges, and `indegree` + `queue` use `O(V)` space.

10. Course Schedule II

Pattern: Topological Sort (Graph – DAG Detection + Ordering)

Problem Statement

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you must take course `bi` first if you want to take course `ai`.

Return the ordering of courses you should take to finish all courses. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return an **empty array**.

Sample Input & Output

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: To take course 1, you must first take course 0.
```

```
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]
Explanation: Courses 1 and 2 depend on 0; course 3 depends on both 1 and 2.
```

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: []
Explanation: Circular dependency → impossible to finish.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import deque, defaultdict

class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]):
        # STEP 1: Initialize structures
        #   - Build adjacency list (graph) and in-degree array
        graph = defaultdict(list)
        in_degree = [0] * numCourses

        for course, prereq in prerequisites:
            graph[prereq].append(course)
            in_degree[course] += 1

        # STEP 2: Main loop / recursion
        #   - Use Kahn's algorithm: start with zero in-degree nodes
        queue = deque()
        for i in range(numCourses):
            if in_degree[i] == 0:
                queue.append(i)

        topo_order = []

        # STEP 3: Update state / bookkeeping
        #   - Process nodes level by level; reduce in-degree of neighbors
        while queue:
            current = queue.popleft()
            topo_order.append(current)

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

        # STEP 4: Return result
        #   - If topo_order doesn't include all courses → cycle exists
        return topo_order if len(topo_order) == numCourses else []

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()
```

```python
#  Test 1: Normal case
result1 = sol.findOrder(2, [[1,0]])
print("Test 1:", result1)  # Expected: [0, 1]

#  Test 2: Edge case - no prerequisites
result2 = sol.findOrder(3, [])
print("Test 2:", result2)  # Expected: [0, 1, 2] (any order)

#  Test 3: Tricky/negative - cycle
result3 = sol.findOrder(2, [[1,0],[0,1]])
print("Test 3:", result3)  # Expected: []
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `numCourses = 2`, `prerequisites = [[1,0]]`.

1. **Initialize `graph` and `in_degree`**:
   - `graph` starts empty (defaultdict of lists).
   - `in_degree = [0, 0]` (for courses 0 and 1).
2. **Process prerequisites**:
   - For `[1,0]`: course `1` depends on `0`.
     - Add `1` to `graph[0]` → `graph = {0: [1]}`
     - Increment `in_degree[1]` → `in_degree = [0, 1]`
3. **Find zero in-degree nodes**:
   - Course `0`: `in_degree[0] == 0` → add to queue.
   - Course `1`: `in_degree[1] == 1` → skip.
   - `queue = deque([0])`
4. **Begin BFS (Kahn's algorithm)**:
   - Pop `0` from queue → add to `topo_order = [0]`
   - Look at neighbors of `0`: `[1]`
     - Reduce `in_degree[1]` from `1` to `0`
     - Now `in_degree[1] == 0` → add `1` to queue

5. **Next iteration**:
   - Pop `1` → `topo_order = [0, 1]`
   - `graph[1]` has no neighbors → nothing to update
6. **Queue empty** → exit loop.
   - `len(topo_order) = 2 == numCourses` → return `[0,1]`

Final Output: `[0, 1]` — correct topological order.

Key takeaway: **Topological sort only works on DAGs**. If a cycle exists, some nodes never reach in-degree 0, so the result list will be shorter than `numCourses`.

Complexity Analysis

- **Time Complexity**: `O(V + E)`

  We visit each course (`V = numCourses`) once and each prerequisite edge (`E = len(prerequisites)`) once during graph building and BFS traversal.

- **Space Complexity**: `O(V + E)`

  The adjacency list (`graph`) stores `E` edges. The `in_degree` array and `queue` use `O(V)` space. Total: `O(V + E)`.

11. Alien Dictionary

Pattern: Topological Sort (Graph + DFS/Kahn's Algorithm)

Problem Statement

There is a new alien language that uses the English alphabet. However, the order among letters is unknown to you.
You are given a list of strings `words` from the alien dictionary, where the strings are sorted lexicographically by the rules of this new language.
Return a string of the unique letters in the new alien language sorted in **lexicographically increasing order** by the new rules. If there is no solution, return `""`. If there are multiple solutions, return **any** of them.

A string `s` is lexicographically smaller than a string `t` if at the first letter where they differ, the letter in `s` comes before the letter in `t` in the alien language.

Constraints:
- `1 <= words.length <= 100`
- `1 <= words[i].length <= 100`
- `words[i]` consists of only lowercase English letters.

Sample Input & Output

```
Input: words = ["wrt", "wrf", "er", "ett", "rftt"]
Output: "wertf"
Explanation: From word pairs, we infer: t < f, w < e, r < t, e < r
→ valid topological order exists.
```

```
Input: words = ["z", "x", "z"]
Output: ""
Explanation: Contradiction: z < x and x < z → cycle → invalid.
```

```
Input: words = ["abc", "ab"]
Output: ""
Explanation: Prefix "ab" comes after "abc" - invalid lexicographic order.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import defaultdict, deque

class Solution:
    def alienOrder(self, words: List[str]) -> str:
        # STEP 1: Initialize structures
        #   - Build graph: char -> set of chars that come after it
        #   - Track in-degree for each unique char
        graph = defaultdict(set)
        in_degree = {}

        # Add all unique chars with in-degree 0 initially
        for word in words:
            for char in word:
                in_degree[char] = 0

        # STEP 2: Main loop / recursion
        #   - Compare adjacent words to infer ordering
        for i in range(len(words) - 1):
            word1, word2 = words[i], words[i + 1]

            # Edge case: prefix violation (e.g., ["abc", "ab"])
            if len(word1) > len(word2) and word1.startswith(word2):
                return ""

            # Find first differing character
            min_len = min(len(word1), len(word2))
            for j in range(min_len):
                c1, c2 = word1[j], word2[j]
                if c1 != c2:
                    # c1 must come before c2
                    if c2 not in graph[c1]:
                        graph[c1].add(c2)
                        in_degree[c2] += 1
                    break  # Only first difference matters

        # STEP 3: Update state / bookkeeping
        #   - Use Kahn's algorithm (BFS) for topological sort
        queue = deque([char for char in in_degree if in_degree[char] == 0])
        result = []

        while queue:
```

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

        # STEP 4: Return result
        #   - If result doesn't include all chars → cycle exists
        if len(result) != len(in_degree):
            return ""
        return "".join(result)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert sol.alienOrder(["wrt", "wrf", "er", "ett", "rftt"]) == "wertf"

    #   Test 2: Edge case - cycle
    assert sol.alienOrder(["z", "x", "z"]) == ""

    #   Test 3: Tricky/negative - invalid prefix order
    assert sol.alienOrder(["abc", "ab"]) == ""

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**: `["wrt", "wrf", "er", "ett", "rftt"]`.

1. **Initialize `in_degree`** with all unique chars:
   Chars: `w, r, t, f, e` → `in_degree = {'w':0, 'r':0, 't':0, 'f':0, 'e':0}`
2. **Compare adjacent words**:

   - "wrt" vs "wrf" → first diff at index 2: `t` vs `f` → add edge `t → f`
     → `graph['t'] = {'f'}`, `in_degree['f'] = 1`
   - "wrf" vs "er" → first diff: `w` vs `e` → edge `w → e`
     → `graph['w'] = {'e'}`, `in_degree['e'] = 1`
   - "er" vs "ett" → first diff: `r` vs `t` → edge `r → t`
     → `graph['r'] = {'t'}`, `in_degree['t'] = 1`
   - "ett" vs "rftt" → first diff: `e` vs `r` → edge `e → r`
     → `graph['e'] = {'r'}`, `in_degree['r'] = 1`

3. **Final `in_degree`**:
   `{'w':0, 'r':1, 't':1, 'f':1, 'e':1}`

4. **Kahn's BFS**:

   - Start with `queue = ['w']` (only char with in-degree 0)
   - Pop `'w'` → add to result → process neighbors: `'e'` → decrement `in_degree['e']` to 0 → enqueue `'e'`
   - Pop `'e'` → result = `['w','e']` → process `'r'` → `in_degree['r'] = 0` → enqueue `'r'`
   - Pop `'r'` → result = `['w','e','r']` → process `'t'` → `in_degree['t'] = 0` → enqueue `'t'`
   - Pop `'t'` → result = `['w','e','r','t']` → process `'f'` → `in_degree['f'] = 0` → enqueue `'f'`
   - Pop `'f'` → result = `['w','e','r','t','f']`

5. **Result length = 5 = total chars** → return "wertf"

Final Output: "wertf"
Key Insight: Lexicographic order gives pairwise constraints → model as DAG → topological sort.

Complexity Analysis

- **Time Complexity**: `O(C)`

  Where `C` = total number of characters in all words.
  We scan each character once to build the graph and once during BFS.
  Each edge is processed once. Number of edges number of unique char pairs 26×26 = O(1), but dominated by input size `C`.

- **Space Complexity**: `O(1)` (or `O(U + E)`)

`U` = number of unique characters (26), `E` = edges (26^2).
So technically `O(1)` since alphabet is fixed, but more precisely `O(U + E)`.

12. Clone Graph

Pattern: Graph Traversal (BFS/DFS) + Hash Map (Node Mapping)

Problem Statement

Given a reference of a node in a **connected undirected graph**, return a **deep copy (clone)** of the graph.
Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

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

Note: The graph is represented using an adjacency list.
You must return the copy of the given node as a reference to the cloned graph.

Sample Input & Output

```
Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
Explanation: There are 4 nodes. Node 1 connects to 2 and 4, etc.
The cloned graph has same structure but new node objects.
```

```
Input: adjList = [[]]
Output: [[]]
Explanation: Single node with no neighbors.
```

```
Input: adjList = [[2],[1]]
Output: [[2],[1]]
Explanation: Two nodes connected bidirectionally.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# Definition for a Node (as provided by LeetCode)
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: Optional['Node']) -> Optional['Node']:
        # STEP 1: Initialize structures
        #   - Use a hash map to track original -> cloned node mapping
        #   - Prevents cycles and duplicate cloning
        if not node:
            return None

        visited = {}

        # STEP 2: Main loop / recursion
        #   - DFS via helper function
        #   - Clone current node, then recursively clone neighbors
        def dfs(n: 'Node') -> 'Node':
            if n in visited:
                return visited[n]  # Return existing clone

            # Create clone of current node (without neighbors yet)
            clone = Node(n.val)
            visited[n] = clone  # Record mapping immediately

            # STEP 3: Update state / bookkeeping
            #   - Recursively clone all neighbors and link them
            for neighbor in n.neighbors:
```

```python
                clone.neighbors.append(dfs(neighbor))

            return clone

        # STEP 4: Return result
        #   - Start DFS from input node
        return dfs(node)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case - 4-node cycle
    n1 = Node(1)
    n2 = Node(2)
    n3 = Node(3)
    n4 = Node(4)
    n1.neighbors = [n2, n4]
    n2.neighbors = [n1, n3]
    n3.neighbors = [n2, n4]
    n4.neighbors = [n1, n3]
    cloned = sol.cloneGraph(n1)
    # Verify structure by values
    assert cloned.val == 1
    assert set(n.val for n in cloned.neighbors) == {2, 4}
    assert set(n.val for n in cloned.neighbors[0].neighbors) == {1, 3}
    print("  Test 1 passed: 4-node graph cloned correctly.")

    #   Test 2: Edge case - single node with no neighbors
    single = Node(1)
    cloned_single = sol.cloneGraph(single)
    assert cloned_single.val == 1
    assert len(cloned_single.neighbors) == 0
    print("  Test 2 passed: Single node cloned correctly.")

    #   Test 3: Tricky case - two nodes connected bidirectionally
    a = Node(1)
    b = Node(2)
    a.neighbors = [b]
    b.neighbors = [a]
    cloned_ab = sol.cloneGraph(a)
    assert cloned_ab.val == 1
```

```
    assert len(cloned_ab.neighbors) == 1
    assert cloned_ab.neighbors[0].val == 2
    assert cloned_ab.neighbors[0].neighbors[0] is cloned_ab  # cycle preserved
    print("  Test 3 passed: 2-node cycle cloned correctly.")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3**: two nodes `a(1)` and `b(2)` connected to each other.

1. **Call `cloneGraph(a)`**
    - `node = a` (not None), so proceed.
    - `visited = {}` (empty dict).
2. **Enter `dfs(a)`**
    - `a` not in `visited`.
    - Create `clone_a = Node(1)`.
    - `visited = {a: clone_a}`.
3. **Loop over `a.neighbors` → `[b]`**
    - Call `dfs(b)`.
4. **Enter `dfs(b)`**
    - `b` not in `visited`.
    - Create `clone_b = Node(2)`.
    - `visited = {a: clone_a, b: clone_b}`.
5. **Loop over `b.neighbors` → `[a]`**
    - Call `dfs(a)` again.
6. **Re-enter `dfs(a)`**
    - Now `a` **is** in `visited` → return `clone_a`.

7. **Back in `dfs(b)`**:
    - Append `clone_a` to `clone_b.neighbors`.
    - Return `clone_b`.
8. **Back in `dfs(a)`**:
    - Append `clone_b` to `clone_a.neighbors`.
    - Return `clone_a`.
9. **Final state**:
    - `clone_a.val = 1`, `clone_a.neighbors = [clone_b]`
    - `clone_b.val = 2`, `clone_b.neighbors = [clone_a]`
    - The cycle is perfectly replicated with new objects.

Key insight: The `visited` map ensures we never clone the same node twice and correctly rebuilds cycles.

Complexity Analysis

- **Time Complexity**: `O(N + M)`

  Where `N` = number of nodes, `M` = number of edges.
  Each node is visited once, and each edge is traversed once during neighbor iteration.

- **Space Complexity**: `O(N)`

  The `visited` hash map stores one entry per node.
  Recursion stack depth is `O(N)` in worst case (e.g., linear chain).

13. Accounts Merge

Pattern: Union-Find (Disjoint Set Union) + Hashing

Problem Statement

Given a list of `accounts` where each element `accounts[i]` is a list of strings, where the first element `accounts[i][0]` is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in **sorted order**. The accounts themselves can be returned in any order.

Sample Input & Output

```
Input: accounts = [
  ["John","johnsmith@mail.com","john_newyork@mail.com"],
  ["John","johnsmith@mail.com","john00@mail.com"],
  ["Mary","mary@mail.com"],
  ["John","johnnybravo@mail.com"]
]
Output: [
  ["John","john00@mail.com","john_newyork@mail.com","johnsmith@mail.com"],
  ["John","johnnybravo@mail.com"],
  ["Mary","mary@mail.com"]
]
Explanation: The first two accounts share "johnsmith@mail.com",
so they are merged.
```

```
Input: accounts = [["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],
                   ["Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"],
                   ["Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"],
                   ["Hanzo","Hanzo3@m.co","Hanzo1@m.co","Hanzo0@m.co"],
                   ["Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"]]
Output: Same as input (no shared emails → no merging)
```

```
Input: [["David","David0@m.co","David1@m.co"],
        ["David","David3@m.co","David4@m.co"],
        ["David","David4@m.co","David5@m.co"],
        ["David","David2@m.co","David3@m.co"],
        ["David","David1@m.co","David2@m.co"]]
Output: One merged account with all 6 emails sorted.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import defaultdict

class Solution:
    def accountsMerge(self, accounts: List[List[str]]) -> List[List[str]]:
        # STEP 1: Initialize Union-Find structures
        #   - email_to_id: map each email to a unique integer ID
        #   - email_to_name: map email to its owner's name
        #   - parent: list for DSU (indexed by email ID)
        email_to_id = {}
        email_to_name = {}
        id_counter = 0

        # Assign unique IDs to all emails and record names
        for account in accounts:
            name = account[0]
            for email in account[1:]:
                if email not in email_to_id:
                    email_to_id[email] = id_counter
                    email_to_name[email] = name
                    id_counter += 1

        # Initialize parent array for DSU
        parent = list(range(id_counter))

        # Helper: find root with path compression
        def find(x: int) -> int:
            if parent[x] != x:
                parent[x] = find(parent[x])
```

```
            return parent[x]

        # Helper: union two email IDs
        def union(x: int, y: int):
            rx, ry = find(x), find(y)
            if rx != ry:
                parent[ry] = rx

        # STEP 2: Union emails within the same account
        #   - All emails in one account belong to same person
        #   - Union first email with every other in the list
        for account in accounts:
            first_email = account[1]
            first_id = email_to_id[first_email]
            for email in account[2:]:
                union(first_id, email_to_id[email])

        # STEP 3: Group emails by root parent
        #   - Use root ID as key to collect all connected emails
        root_to_emails = defaultdict(list)
        for email in email_to_id:
            root = find(email_to_id[email])
            root_to_emails[root].append(email)

        # STEP 4: Build final result
        #   - For each group: get name from any email, sort emails
        result = []
        for emails in root_to_emails.values():
            name = email_to_name[emails[0]]
            result.append([name] + sorted(emails))

        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - two accounts share an email
    accounts1 = [
        ["John","johnsmith@mail.com","john_newyork@mail.com"],
        ["John","johnsmith@mail.com","john00@mail.com"],
        ["Mary","mary@mail.com"],
```

```
        ["John","johnnybravo@mail.com"]
    ]
    expected1 = [
      ["John","john00@mail.com","john_newyork@mail.com","johnsmith@mail.com"],
      ["John","johnnybravo@mail.com"],
      ["Mary","mary@mail.com"]
    ]
    output1 = sol.accountsMerge(accounts1)
    assert sorted(
      [sorted(x) for x in output1]) == sorted([sorted(x) for x in expected1])
    print("  Test 1 passed")

    #   Test 2: Edge case - no shared emails
    accounts2 = [
        ["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],
        ["Kevin","Kevin3@m.co","Kevin5@m.co","Kevin0@m.co"]
    ]
    output2 = sol.accountsMerge(accounts2)
    # Should return same structure (emails sorted per account)
    expected2 = [
        ["Gabe","Gabe0@m.co","Gabe1@m.co","Gabe3@m.co"],
        ["Kevin","Kevin0@m.co","Kevin3@m.co","Kevin5@m.co"]
    ]
    assert sorted(
        [sorted(x) for x in output2]) == sorted([sorted(x) for x in expected2])
    print("  Test 2 passed")

    #   Test 3: Tricky case - chain of shared emails
    accounts3 = [
        ["David","David0@m.co","David1@m.co"],
        ["David","David3@m.co","David4@m.co"],
        ["David","David4@m.co","David5@m.co"],
        ["David","David2@m.co","David3@m.co"],
        ["David","David1@m.co","David2@m.co"]
    ]
    output3 = sol.accountsMerge(accounts3)
    expected3 = [["David","David0@m.co","David1@m.co","David2@m.co",
                  "David3@m.co","David4@m.co","David5@m.co"]]
    assert len(output3) == 1
    assert sorted(output3[0]) == sorted(expected3[0])
    print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1** step-by-step:

1. **Email ID Assignment**
   - Process each account:
     - Account 0: "John" → emails: `johnsmith@mail.com` → ID 0, `john_newyork@mail.com` → ID 1
     - Account 1: "John" → `johnsmith@mail.com` (already ID 0), `john00@mail.com` → ID 2
     - Account 2: "Mary" → `mary@mail.com` → ID 3
     - Account 3: "John" → `johnnybravo@mail.com` → ID 4
   - `email_to_id = {johnsmith:0, john_newyork:1, john00:2, mary:3, johnnybravo:4}`
   - `parent = [0,1,2,3,4]` (each email its own root)
2. **Union Within Accounts**
   - Account 0: union(0,1) → parent[1] = 0 → `parent = [0,0,2,3,4]`
   - Account 1: union(0,2) → find(0)=0, find(2)=2 → parent[2]=0 → `parent = [0,0,0,3,4]`
   - Account 2: only one email → no union
   - Account 3: only one email → no union
3. **Path Compression on Find**
   - When we later call `find(1)`, it becomes 0 and updates `parent[1]=0` (already done)
   - `find(2)` → parent[2]=0, so returns 0 and sets `parent[2]=0`

4. **Group by Root**
   - For each email:
     - `johnsmith` (ID 0): root = 0 → group[0] += [johnsmith]
     - `john_newyork` (ID 1): find(1)=0 → group[0] += [john_newyork]
     - `john00` (ID 2): find(2)=0 → group[0] += [john00]
     - `mary` (ID 3): root=3 → group[3] = [mary]
     - `johnnybravo` (ID 4): root=4 → group[4] = [johnnybravo]
5. **Build Result**
   - Group 0: name = "John", emails sorted → `["John", "john00...", "john_newyork...", "johnsmith..."]`
   - Group 3: `["Mary", "mary@mail.com"]`
   - Group 4: `["John", "johnnybravo@mail.com"]`

Final output matches expected.

Complexity Analysis

- **Time Complexity**: `O(N * (N))`
  > Where `N` is total number of emails. Each `find`/`union` is nearly O(1) due to path compression and union by rank (implicit here). Sorting emails per group adds `O(K log K)` per group, but total over all groups is `O(N log N)` in worst case. However, since DSU dominates the merging logic and sorting is unavoidable for output, overall is often cited as `O(N log N)` due to sorting. But **core union-find is `O(N (N))`**, and is inverse Ackermann (~constant).
- **Space Complexity**: `O(N)`
  > We store `email_to_id`, `email_to_name`, `parent`, and `root_to_emails` — all proportional to number of unique emails `N`.

14. Word Ladder

Pattern: BFS (Breadth-First Search) on Implicit Graph

Problem Statement

A transformation sequence from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words such that:
- Every adjacent pair of words differs by exactly one letter.
- Every word in the sequence is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `endWord` must be in `wordList`.

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return the **number of words** in the shortest transformation sequence from `beginWord` to `endWord`, or `0` if no such sequence exists.

Sample Input & Output

```
Input: beginWord = "hit", endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation is "hit" →
"hot" → "dot" → "dog" → "cog" (5 words).
```

```
Input: beginWord = "hit", endWord = "cog",
wordList = ["hot","dot","dog","lot","log"]
Output: 0
Explanation: "cog" is not in wordList, so no valid path exists.
```

```
Input: beginWord = "a", endWord = "c", wordList = ["a","b","c"]
Output: 2
Explanation: "a" → "c" (differs by one letter), so sequence length = 2.
```

LeetCode Editorial Solution + Inline Tests

```python
from collections import deque
from typing import List

class Solution:
    def ladderLength(
        self, beginWord: str, endWord: str, wordList: List[str]
    ) -> int:
        # STEP 1: Initialize structures
        #   - Convert wordList to a set for O(1) lookups
        #   - Use a queue for BFS: stores (current_word, level)
        word_set = set(wordList)
        if endWord not in word_set:
            return 0  # Early exit if endWord missing

        queue = deque([(beginWord, 1)])
        visited = {beginWord}

        # STEP 2: Main loop / recursion
        #   - BFS explores all words at current level before moving deeper
        #   - For each word, try changing every char to 'a'-'z'
        while queue:
            word, level = queue.popleft()

            # STEP 3: Update state / bookkeeping
            #   - If we reach endWord, return level (number of words)
            if word == endWord:
                return level

            # Generate all possible one-letter mutations
            for i in range(len(word)):
                for c in 'abcdefghijklmnopqrstuvwxyz':
                    next_word = word[:i] + c + word[i+1:]
                    if next_word in word_set and next_word not in visited:
                        visited.add(next_word)
                        queue.append((next_word, level + 1))

        # STEP 4: Return result
        #   - If BFS completes without finding endWord, return 0
        return 0
```

```python
# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.ladderLength(
        "hit", "cog", ["hot","dot","dog","lot","log","cog"]
    ) == 5

    #  Test 2: Edge case - endWord missing
    assert sol.ladderLength(
        "hit", "cog", ["hot","dot","dog","lot","log"]
    ) == 0

    #  Test 3: Tricky/negative - single-letter words
    assert sol.ladderLength("a", "c", ["a","b","c"]) == 2

    print(" All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**:
`beginWord = "hit"`, `endWord = "cog"`,
`wordList = ["hot","dot","dog","lot","log","cog"]`

Initial State:
- `word_set = {"hot","dot","dog","lot","log","cog"}`
- `queue = deque([("hit", 1)])`
- `visited = {"hit"}`

Step 1: Dequeue `("hit", 1)`
- Not `"cog"` → continue
- Try all 1-letter changes of `"hit"`:

- Change index 0: `"ait"`, `"bit"`, …, `"hot"` → only `"hot"` in `word_set`
- Add `"hot"` to `visited`, enqueue `("hot", 2)`

State:
- `queue = [("hot", 2)]`
- `visited = {"hit", "hot"}`

Step 2: Dequeue `("hot", 2)`
- Not `"cog"`
- Mutations:
- Index 1: `"hat"`, `"hbt"`, …, `"dot"`, `"lot"` → both in set
- Enqueue `("dot", 3)` and `("lot", 3)`, mark visited

State:
- `queue = [("dot", 3), ("lot", 3)]`
- `visited = {"hit", "hot", "dot", "lot"}`

Step 3: Dequeue `("dot", 3)`
- Mutations:
- Index 2: `"doa"`, …, `"dog"` → `"dog"` valid
- Enqueue `("dog", 4)`

State:
- `queue = [("lot", 3), ("dog", 4)]`
- `visited` adds `"dog"`

Step 4: Dequeue `("lot", 3)`
- Mutations → `"log"` valid
- Enqueue `("log", 4)`

State:
- `queue = [("dog", 4), ("log", 4)]`

Step 5: Dequeue `("dog", 4)`
- Mutations → `"cog"` found!
- Enqueue `("cog", 5)`

Step 6: Dequeue `("log", 4)`
- Also finds `"cog"`, but already visited → skip

Step 7: Dequeue `("cog", 5)`
- Matches `endWord` → return `5`

Final Output: `5`

Key Takeaway: BFS guarantees shortest path in unweighted graphs. Each level = one more word in sequence.

Complexity Analysis

- **Time Complexity**: `O(M² × N)`
  - `N` = number of words in `wordList`
  - `M` = length of each word
  - For each word (up to `N`), we generate `M` positions × 26 letters = `O(26M)` `O(M)`
  - But slicing `word[:i] + c + word[i+1:]` takes `O(M)`, so total per word: `O(M²)`
  - Overall: `O(N × M²)`
- **Space Complexity**: `O(N × M)`
  - `word_set` stores `N` words of length `M` → `O(NM)`
  - `visited` and `queue` may store up to `N` words → `O(NM)`
  - Total: `O(NM)`

15. Minimum Knight Moves

Pattern: BFS (Breadth-First Search) on Implicit Graph

Problem Statement

In an infinite chessboard, a knight starts at `[0, 0]` and wants to reach a target position `[x, y]`.
Return the **minimum number of moves** required for the knight to reach the target.

A knight moves in an L-shape: 2 squares in one direction and 1 square perpendicular, or vice versa.

Note: Due to symmetry, you can assume `x  0`, `y  0`. The answer is the same for all quadrants.

Sample Input & Output

```
Input: x = 2, y = 1
Output: 1
Explanation: Knight moves from (0,0) → (2,1) in one move.
```

```
Input: x = 5, y = 5
Output: 4
Explanation: One optimal path: (0,0) → (2,1) → (3,3) → (4,5) → (5,5)
```

```
Input: x = 1, y = 1
Output: 2
Explanation: Cannot reach (1,1) in 1 move. Minimum: (0,0) → (2,-1) → (1,1)
```

LeetCode Editorial Solution + Inline Tests

```python
from collections import deque
from typing import Tuple

class Solution:
    def minKnightMoves(self, x: int, y: int) -> int:
        # Normalize to first quadrant using symmetry
        x, y = abs(x), abs(y)

        # STEP 1: Initialize BFS structures
        #   - Queue holds (current_x, current_y, move_count)
        #   - Visited set prevents reprocessing same cell
        queue = deque([(0, 0, 0)])
        visited = {(0, 0)}

        # All 8 possible knight moves
        directions = [
            (2, 1), (2, -1), (-2, 1), (-2, -1),
            (1, 2), (1, -2), (-1, 2), (-1, -2)
        ]

        # STEP 2: Main BFS loop
        #   - Process level by level (guarantees minimum moves)
        #   - Stop when target is reached
        while queue:
            curr_x, curr_y, moves = queue.popleft()

            # STEP 3: Check if target reached
            if curr_x == x and curr_y == y:
                return moves

            # STEP 4: Explore all valid next positions
            for dx, dy in directions:
                nx, ny = curr_x + dx, curr_y + dy

                # Optimization: only explore near target
                # Since knight can't go too far negatively,
                # we bound search to x+2, y+2 (empirically safe)
                if (nx, ny) not in visited and nx >= -2 and ny >= -2:
                    visited.add((nx, ny))
                    queue.append((nx, ny, moves + 1))

        # Should never reach here for valid input
```

```python
        return -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    result1 = sol.minKnightMoves(2, 1)
    print(f"Test 1 - Input: (2,1) → Output: {result1}")  # Expected: 1

    #  Test 2: Edge case (diagonal close)
    result2 = sol.minKnightMoves(1, 1)
    print(f"Test 2 - Input: (1,1) → Output: {result2}")  # Expected: 2

    #  Test 3: Tricky/negative (symmetry test)
    result3 = sol.minKnightMoves(-5, -5)
    print(f"Test 3 - Input: (-5,-5) → Output: {result3}")  # Expected: 4
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `minKnightMoves(1, 1)` step by step.

Initial state:
- `x = 1`, `y = 1` → normalized to `(1, 1)`
- `queue = deque([(0, 0, 0)])`
- `visited = {(0, 0)}`

Step 1: Dequeue `(0, 0, 0)`
- Not target → explore neighbors.
- Generate 8 moves:
`(2,1)`, `(2,-1)`, `(-2,1)`, `(-2,-1)`,
`(1,2)`, `(1,-2)`, `(-1,2)`, `(-1,-2)`
- Filter by `nx >= -2 and ny >= -2` → all pass.
- Add all 8 to `visited` and `queue` with `moves = 1`.

State after Step 1:
- `queue` has 8 entries, e.g., `(2,1,1)`, `(2,-1,1)`, …, `(-1,-2,1)`
- `visited` has 9 points.

Step 2: Process each of the 8 positions (BFS level 1).
None equal `(1,1)`. For each, generate next moves.
Many are duplicates → skipped by `visited`.
But from `(2,-1)`, one move is `(2-1, -1+2) = (1,1)` → **target!**

Step 3: When processing `(2, -1, 1)`:
- Generate `(1, 1)` → not visited → add to queue as `(1,1,2)`.

Step 4: Later, dequeue `(1,1,2)` → matches target → return `2`.

Final output: `2`

Complexity Analysis

- **Time Complexity**: `O(max(|x|, |y|)²)`

  BFS explores a grid roughly proportional to the distance from origin to target. The pruning (`nx >= -2`, `ny >= -2`) bounds the search space to a constant factor around the target.

- **Space Complexity**: `O(max(|x|, |y|)²)`

  The `visited` set and queue store positions in the explored region, which scales quadratically with distance.

16. Bus Routes

Pattern: Graph BFS (Shortest Path in Unweighted Graph)

Problem Statement

You are given an array `routes` representing bus routes where `routes[i]` is a bus route that the `ith` bus repeats forever.
For example, if `routes[0] = [1, 5, 7]`, this means the 0th bus travels in a circular route: `1 → 5 → 7 → 1 → 5 → 7 → ...`
You start at bus stop `source` and want to go to bus stop `target`. Return the **least number of buses you must take** to travel from `source` to `target`. Return `-1` if it is not possible.

Constraints:
- `1 <= routes.length <= 500`
- `1 <= routes[i].length <= 10`
- All values in `routes[i]` are unique.
- `sum(routes[i].length) <= 10`
- `0 <= source, target < 10`

Sample Input & Output

```
Input: routes = [[1,2,7],[3,6,7]], source = 1, target = 6
Output: 2
Explanation: Take bus 0 from stop 1 → 7, then bus 1 from 7 → 6.
```

```
Input: routes = [[7,12],[4,5,15],[6],[15,19],[9,12,13]],
       source = 15, target = 12
Output: -1
Explanation: No path connects stop 15 to stop 12.
```

```
Input: routes = [[1,2,7],[3,6,7]], source = 7, target = 7
Output: 0
Explanation: Already at target - no bus needed.
```

LeetCode Editorial Solution + Inline Tests

```python
from collections import defaultdict, deque
from typing import List

class Solution:
    def numBusesToDestination(
        self, routes: List[List[int]], source: int, target: int
    ) -> int:
        # STEP 1: Initialize structures
        #   - stop_to_buses: map each stop to list of bus indices
        #   - queue: BFS queue of (stop, num_buses)
        #   - visited_stops: avoid revisiting stops
        #   - visited_buses: avoid re-boarding same bus

        if source == target:
            return 0

        stop_to_buses = defaultdict(list)
        for bus_id, stops in enumerate(routes):
            for stop in stops:
                stop_to_buses[stop].append(bus_id)

        queue = deque([(source, 0)])
        visited_stops = set([source])
        visited_buses = set()

        # STEP 2: Main loop / recursion
        #   - BFS over stops; when we board a bus, we mark all its
        #     stops as reachable with +1 bus count
        while queue:
            current_stop, num_buses = queue.popleft()

            # STEP 3: Update state / bookkeeping
            #   - For each bus that serves current_stop,
            #     if not already taken, add all its stops to queue
            for bus_id in stop_to_buses[current_stop]:
                if bus_id in visited_buses:
                    continue
                visited_buses.add(bus_id)

                for next_stop in routes[bus_id]:
                    if next_stop == target:
                        return num_buses + 1
```

```python
                    if next_stop not in visited_stops:
                        visited_stops.add(next_stop)
                        queue.append((next_stop, num_buses + 1))

        # STEP 4: Return result
        #   - If BFS ends without finding target, impossible
        return -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.numBusesToDestination(
        [[1,2,7],[3,6,7]], 1, 6
    ) == 2, "Test 1 Failed"

    #  Test 2: Edge case - already at target
    assert sol.numBusesToDestination(
        [[1,2,7],[3,6,7]], 7, 7
    ) == 0, "Test 2 Failed"

    #  Test 3: Tricky/negative - unreachable
    assert sol.numBusesToDestination(
        [[7,12],[4,5,15],[6],[15,19],[9,12,13]], 15, 12
    ) == -1, "Test 3 Failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**:
`routes = [[1,2,7],[3,6,7]]`, `source = 1`, `target = 6`.

Initial Setup: - `stop_to_buses`: - `1 → [0]` - `2 → [0]` - `7 → [0,1]` - `3 → [1]` - `6 → [1]` - `queue = deque([(1, 0)])` - `visited_stops = {1}` - `visited_buses = set()`

Step 1: Dequeue `(1, 0)`
- Current stop = `1`, buses taken = `0`
- Buses at stop `1`: `[0]`
- Bus `0` not visited → mark `visited_buses = {0}`
- Explore all stops on bus `0`: `[1, 2, 7]`
- `1`: already visited → skip
- `2`: not visited → add `(2, 1)` to queue, `visited_stops = {1,2}`
- `7`: not visited → add `(7, 1)` to queue, `visited_stops = {1,2,7}`

Queue now: `[(2,1), (7,1)]`

Step 2: Dequeue `(2, 1)`
- Buses at stop `2`: `[0]` → already visited → skip
- No new stops added.

Queue now: `[(7,1)]`

Step 3: Dequeue `(7, 1)`
- Buses at stop `7`: `[0,1]`
- Bus `0`: already visited → skip
- Bus `1`: not visited → add to `visited_buses = {0,1}`
- Explore stops on bus `1`: `[3,6,7]`
- `3`: not visited → add `(3,2)`, `visited_stops = {1,2,7,3}`
- `6`: **this is target!** → return `1 + 1 = 2`

Final Output: `2`

Key Insight:
We don't BFS over individual stops one-by-one. Instead, **when we reach any stop on a bus route, we "take" that entire bus**, and all its stops become reachable in one additional step. This avoids $O(N^2)$ edge explosion.

Complexity Analysis

- **Time Complexity**: `O(S)`, where `S = sum(len(route) for route in routes)`

  We visit each stop at most once, and for each stop, we iterate over all buses that serve it. But each bus is processed only once (via `visited_buses`), and processing a bus touches all its stops. Total work is proportional to total number of stop-bus memberships, which is `S 10`.

- **Space Complexity**: `O(S)`

  `stop_to_buses` stores `S` entries. `visited_stops` and `queue` store at most `O(unique stops) S`. `visited_buses` stores at most `O(number of buses) 500`. Dominated by `O(S)`.

17. Cheapest Flights Within K Stops

Pattern: **Graph — Shortest Path with Limited Steps (BFS / Modified Dijkstra / DP)**

Problem Statement

There are `n` cities connected by some number of flights. You are given an array `flights` where `flights[i] = [from_i, to_i, price_i]` indicates that there is a flight from city `from_i` to city `to_i` with cost `price_i`.

You are also given three integers `src`, `dst`, and `k`, and you need to find the cheapest price from `src` to `dst` with **at most `k` stops**. If there is no such route, return `-1`.

Sample Input & Output

```
Input: n = 4, flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]],
src = 0, dst = 3, k = 1
Output: 700
Explanation: The path 0 → 1 → 3 uses 1 stop (  k) and costs 100 + 600 = 700.
```

```
Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]],
src = 0, dst = 2, k = 0
Output: 500
Explanation: Direct flight 0 → 2 is the only option with 0 stops.
```

```
Input: n = 3, flights = [[0,1,100]], src = 0, dst = 2, k = 1
Output: -1
Explanation: No route from 0 to 2 within 1 stop.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import deque

class Solution:
    def findCheapestPrice(
        self, n: int, flights: List[List[int]], src: int, dst: int, k: int
    ) -> int:
        # STEP 1: Build adjacency list for graph
        #   - Each entry: {city: [(neighbor, price), ...]}
        graph = [[] for _ in range(n)]
        for u, v, w in flights:
            graph[u].append((v, w))

        # STEP 2: BFS with stops tracking
        #   - Queue stores (cost_so_far, city, stops_used)
        #   - We allow up to k stops → max k+1 edges
        queue = deque()
        queue.append((0, src, 0))  # (cost, city, stops)

        # Track min cost to reach each city (pruning)
        min_cost = [float('inf')] * n
        min_cost[src] = 0

        # STEP 3: Process queue level by level
        while queue:
            cost, city, stops = queue.popleft()
```

```
            # Skip if we've exceeded stop limit
            if stops > k:
                continue

            # Explore neighbors
            for neighbor, price in graph[city]:
                new_cost = cost + price

                # Only proceed if we found a cheaper way
                # to reach 'neighbor'
                if new_cost < min_cost[neighbor]:
                    min_cost[neighbor] = new_cost
                    queue.append((new_cost, neighbor, stops + 1))

        # STEP 4: Return result or -1 if unreachable
        return min_cost[dst] if min_cost[dst] != float('inf') else -1

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    n1 = 4
    flights1 = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]]
    src1, dst1, k1 = 0, 3, 1
    print(f"Test 1: {sol.findCheapestPrice(n1, flights1, src1, dst1, k1)}")
    # Expected: 700

    #  Test 2: Edge case - direct flight only
    n2 = 3
    flights2 = [[0,1,100],[1,2,100],[0,2,500]]
    src2, dst2, k2 = 0, 2, 0
    print(f"Test 2: {sol.findCheapestPrice(n2, flights2, src2, dst2, k2)}")
    # Expected: 500

    #  Test 3: Tricky/negative - unreachable
    n3 = 3
    flights3 = [[0,1,100]]
    src3, dst3, k3 = 0, 2, 1
    print(f"Test 3: {sol.findCheapestPrice(n3, flights3, src3, dst3, k3)}")
    # Expected: -1
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**:
`n = 4`, `flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]]`,
`src = 0`, `dst = 3`, `k = 1`

Goal: Find cheapest price from city 0 to city 3 with **1 stop**.

Step 0: Build Graph

We create an adjacency list:
- `graph[0] = [(1, 100)]`
- `graph[1] = [(2, 100), (3, 600)]`
- `graph[2] = [(0, 100), (3, 200)]`
- `graph[3] = []`

`min_cost = [0, ∞, ∞, ∞]` (only src=0 has cost 0)

Queue starts as: `deque([(0, 0, 0)])` → (cost=0, city=0, stops=0)

Step 1: Pop (0, 0, 0)

- `stops = 0   k=1` → OK
- From city 0, go to city 1 with price 100
  - `new_cost = 0 + 100 = 100`
  - `100 < ∞` → update `min_cost[1] = 100`
  - Push `(100, 1, 1)` into queue

Queue: `[(100, 1, 1)]`
`min_cost = [0, 100, ∞, ∞]`

Step 2: Pop (100, 1, 1)

- `stops = 1   k=1` → OK
- From city 1:
  - To city 2: `new_cost = 100 + 100 = 200` → update `min_cost[2] = 200`, push `(200, 2, 2)`
  - To city 3: `new_cost = 100 + 600 = 700` → update `min_cost[3] = 700`, push `(700, 3, 2)`

Queue: `[(200, 2, 2), (700, 3, 2)]`
`min_cost = [0, 100, 200, 700]`

Step 3: Pop (200, 2, 2)

- `stops = 2 > k=1` → **skip** (too many stops)

Step 4: Pop (700, 3, 2)

- `stops = 2 > k=1` → **skip**

Queue empty → done.

Final `min_cost[3] = 700` → return **700**

Matches expected output!

Complexity Analysis

- **Time Complexity**: `O((k + 1) * E)`
  > In worst case, we explore each edge up to `k+1` times (once per allowed stop level).
  > `E = len(flights)` → each edge may be relaxed once per stop layer.
- **Space Complexity**: `O(n + E)`
  > `O(E)` for adjacency list, `O(n)` for `min_cost` array and queue (queue holds at most `O(n)` per level, but total bounded by graph size).

18. Pacific Atlantic Water Flow

Pattern: Graph Traversal (Multi-source BFS/DFS)

Problem Statement

There is an `m x n` rectangular island that borders both the **Pacific Ocean** (top and left edges) and **Atlantic Ocean** (bottom and right edges).

The island is partitioned into a grid of square cells. You are given an `m x n` integer matrix `heights` where `heights[r][c]` represents the **height above sea level** of the cell at coordinate `(r, c)`.

The island receives a lot of rain, and the rainwater can flow to neighboring cells directly north, south, east, or west **if the neighboring cell's height is less than or equal to** the current cell's height.

Water can flow from any cell adjacent to an ocean into the ocean.

Return a **2D list** of grid coordinates `result` where `result[i] = [r , c ]` denotes that **rainwater can flow from cell `(r , c )` to both the Pacific and Atlantic oceans**.

Sample Input & Output

```
Input: heights = [[1,2,2,3,5],
                  [3,2,3,4,4],
                  [2,4,5,3,1],
                  [6,7,1,4,5],
                  [5,1,1,2,4]]
Output: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]
Explanation: These cells can reach both oceans via non-increasing paths.
```

```
Input: heights = [[1]]
Output: [[0,0]]
Explanation: Single cell touches both oceans
(top-left = Pacific, bottom-right = Atlantic).
```

```
Input: heights = [[1,2,3],
                  [8,9,4],
                  [7,6,5]]
Output: [[0,2],[1,1],[1,2],[2,0],[2,1],[2,2]]
Explanation: Spiral matrix - high center allows flow outward in
multiple directions.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        # STEP 1: Initialize structures
        #   - Use two sets to track cells reachable from each ocean.
        #   - Start DFS from ocean edges (multi-source).
        if not heights or not heights[0]:
            return []

        m, n = len(heights), len(heights[0])
        pacific_reachable = set()
        atlantic_reachable = set()

        # STEP 2: Main loop / recursion
        # - Perform DFS from all Pacific-border cells (top row, left col).
        # - Perform DFS from all Atlantic-border cells (bottom row, right col).
        # - DFS moves to neighbors with >= height (reverse flow logic).
        def dfs(r: int, c: int, reachable: set):
            reachable.add((r, c))
            for dr, dc in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                nr, nc = r + dr, c + dc
                if (0 <= nr < m and 0 <= nc < n and
                    (nr, nc) not in reachable and
                    heights[nr][nc] >= heights[r][c]):
                    dfs(nr, nc, reachable)

        # Pacific: top row and left column
        for i in range(m):
```

```python
            dfs(i, 0, pacific_reachable)
        for j in range(n):
            dfs(0, j, pacific_reachable)

        # Atlantic: bottom row and right column
        for i in range(m):
            dfs(i, n - 1, atlantic_reachable)
        for j in range(n):
            dfs(m - 1, j, atlantic_reachable)

        # STEP 3: Update state / bookkeeping
        #   - Intersection of both sets = cells reaching both oceans.
        #   - Convert to list of lists for output.
        both = pacific_reachable & atlantic_reachable

        # STEP 4: Return result
        #   - Order doesn't matter per problem statement.
        return [[r, c] for r, c in both]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    heights1 = [
        [1,2,2,3,5],
        [3,2,3,4,4],
        [2,4,5,3,1],
        [6,7,1,4,5],
        [5,1,1,2,4]
    ]
    expected1 = [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]
    result1 = sol.pacificAtlantic(heights1)
    assert sorted(result1) == sorted(expected1), f"Test 1 failed: {result1}"

    #  Test 2: Edge case - single cell
    heights2 = [[1]]
    expected2 = [[0,0]]
    result2 = sol.pacificAtlantic(heights2)
    assert result2 == expected2, f"Test 2 failed: {result2}"

    #  Test 3: Tricky/negative - spiral with high center
```

```
    heights3 = [
        [1,2,3],
        [8,9,4],
        [7,6,5]
    ]
    expected3 = [[0,2],[1,1],[1,2],[2,0],[2,1],[2,2]]
    result3 = sol.pacificAtlantic(heights3)
    assert sorted(result3) == sorted(expected3), f"Test 3 failed: {result3}"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1** (`heights1`) step by step:

1. **Initialization**:
    - `m = 5`, `n = 5`
    - `pacific_reachable = set()`, `atlantic_reachable = set()`
2. **Pacific DFS starts**:
    - From **left column**: `(0,0)`, `(1,0)`, `(2,0)`, `(3,0)`, `(4,0)`
    - From **top row**: `(0,1)`, `(0,2)`, `(0,3)`, `(0,4)`
    - DFS from `(0,0)` (height=1): can go to `(1,0)` (3 1) → then `(2,0)` (2 3? No! Wait—reverse logic: we allow **higher or equal** neighbors because we're flowing *from ocean inward*. So from `(1,0)=3`, we can go to `(2,0)=2`? No—2 < 3 → not allowed. But from `(3,0)=6`, we can go to `(2,0)=2`? No. However, from `(3,0)=6`, we can go to `(3,1)=7` (7 6) → yes! So `(3,1)` gets added to Pacific set.
3. **Atlantic DFS starts**:
    - From **right column**: `(0,4)`, `(1,4)`, `(2,4)`, `(3,4)`, `(4,4)`
    - From **bottom row**: `(4,0)`, `(4,1)`, `(4,2)`, `(4,3)`

- DFS from `(4,4)=4` → can go to `(4,3)=2`? No (2 < 4). But `(3,4)=5  4` → yes. Then from `(3,4)=5`, go to `(3,3)=4` (4 5 → allowed in reverse), etc.

4. **After both DFS runs**:

   - `pacific_reachable` includes `(3,0)`, `(3,1)`, `(4,0)`, `(0,4)`, etc.

   - `atlantic_reachable` includes `(0,4)`, `(1,4)`, `(2,2)`, `(3,0)`, etc.

5. **Intersection**:

   - Common cells: `(0,4)`, `(1,3)`, `(1,4)`, `(2,2)`, `(3,0)`, `(3,1)`, `(4,0)`

   - Converted to list of lists → matches expected output.

Key Insight: Instead of checking every cell (expensive), we **reverse the flow**—start from oceans and climb *up* (to higher/equal ground). This ensures we only traverse reachable regions.

Complexity Analysis

- **Time Complexity**: `O(m × n)`

  Each cell is visited at most once by Pacific DFS and once by Atlantic DFS. Total work is linear in number of cells.

- **Space Complexity**: `O(m × n)`

  In worst case (flat terrain), both `pacific_reachable` and `atlantic_reachable` store all `m×n` cells. Recursion stack depth also up to `O(m×n)` in worst case (though typically much less).

19. Longest Increasing Path in a Matrix

Pattern: DFS with Memoization (Graph Traversal + Dynamic Programming)

Problem Statement

Given an `m x n` integers `matrix`, return *the length of the longest increasing path in the matrix.*
From each cell, you can move in **four directions**: left, right, up, or down.
You **may not** move diagonally or move outside the boundary (i.e., wrap-around is not allowed).
A path is *increasing* if each subsequent value is **strictly greater** than the previous.

Sample Input & Output

```
Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]
Output: 4
Explanation: The longest increasing path is [1 → 2 → 6 → 9].
```

```
Input: matrix = [[3,4,5],[3,2,6],[2,2,1]]
Output: 4
Explanation: One path is [3 → 4 → 5 → 6].
```

```
Input: matrix = [[1]]
Output: 1
Explanation: Single cell → path length = 1.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

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])
```

```
        memo = [[0] * cols for _ in range(rows)]
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        def dfs(r: int, c: int) -> int:
            # STEP 1: Return cached result if already computed
            if memo[r][c] != 0:
                return memo[r][c]

            max_path = 1  # At least the cell itself

            # STEP 2: Explore all 4 directions
            for dr, dc in directions:
                nr, nc = r + dr, c + dc

                # STEP 3: Only move to strictly greater neighbor
                if (0 <= nr < rows and 0 <= nc < cols and
                    matrix[nr][nc] > matrix[r][c]):

                    # Recursively get path length from neighbor
                    path_from_neighbor = dfs(nr, nc)
                    max_path = max(max_path, 1 + path_from_neighbor)

            # STEP 4: Cache and return result
            memo[r][c] = max_path
            return max_path

        result = 0
        for r in range(rows):
            for c in range(cols):
                result = max(result, dfs(r, c))

        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.longestIncreasingPath([[9,9,4],[6,6,8],[2,1,1]]) == 4

    #  Test 2: Edge case - single cell
    assert sol.longestIncreasingPath([[1]]) == 1
```

```python
#  Test 3: Tricky/negative - all equal values
assert sol.longestIncreasingPath([[7,7,7],[7,7,7],[7,7,7]]) == 1
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `matrix = [[9,9,4],[6,6,8],[2,1,1]]`.

1. **Initialization**:

   - `rows = 3`, `cols = 3`
   - `memo` is a 3×3 grid of zeros.
   - Directions: right, down, left, up.

2. **Outer loop starts at (0,0)** → value = 9.

   - All neighbors (9,6) 9 → no valid moves.
   - `dfs(0,0)` returns 1 → `memo[0][0] = 1`.

3. **Continue to (0,1)** → also 9 → same → `memo[0][1] = 1`.

4. **At (0,2)** → value = 4.

   - Down to (1,2) = 8 (>4) → call `dfs(1,2)`.
     - From (1,2)=8: down to (2,2)=1 (no), up=4 (no), left=(1,1)=6 (<8 → no), right=invalid.
     - So `dfs(1,2)` returns 1 → `memo[1][2] = 1`.
   - So `dfs(0,2) = 1 + 1 = 2` → `memo[0][2] = 2`.

5. **At (1,0)** → value = 6.

   - Down to (2,0)=2 (<6 → no).
   - Up to (0,0)=9 (>6) → call `dfs(0,0)` → returns 1 (cached).
   - So path: 6 → 9 → length = 2.
   - But also check left/right: (1,1)=6 (not >), so max = 2.

6. **At (2,0)** → value = 2.

   - Up to (1,0)=6 (>2) → call `dfs(1,0)` → returns 2 (from above).
   - So path: 2 → 6 → 9 → length = 3.

- Also right to (2,1)=1 (<2 → no).
- So `memo[2][0] = 3`.

7. **At (2,1)** → value = 1.

   - Up to (1,1)=6 (>1) → call `dfs(1,1)`.
     - From (1,1)=6: up=(0,1)=9 → `dfs(0,1)=1` → path = 2.
     - Left=(1,0)=6 (not >), right=(1,2)=8 (>6) → `dfs(1,2)=1` → path = 2.
     - So `dfs(1,1) = 2`.
   - So from (2,1): 1 → 6 → 9 → length = 3.
   - Also left to (2,0)=2 (>1) → `dfs(2,0)=3` → path = 1 + 3 = **4**!
   - So `memo[2][1] = 4`.

8. Final `result = max(..., 4) = 4`.

Final Output: `4`

Key Insight: Without memoization, we'd recompute paths like `6→9` many times. Memoization turns exponential time into polynomial.

Complexity Analysis

- **Time Complexity**: `O(m * n)`

  Each cell is visited **once** due to memoization. For each cell, we check 4 neighbors → constant work per cell. Total = `m * n * O(1) = O(mn)`.

- **Space Complexity**: `O(m * n)`

  The `memo` table uses `O(mn)` space.
  Recursion depth is at most `O(mn)` in worst-case (e.g., strictly increasing spiral), but in practice limited by path length. Still, worst-case stack space is `O(mn)`.
  Thus, total space = `O(mn)`.

20. Word Search II

Pattern: Trie + Backtracking (DFS)

Problem Statement

Given an `m x n` `board` of characters and a list of strings `words`, return **all words on the board**.

Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may **not be used more than once in a word**.

Sample Input & Output

```
Input: board = [["o","a","a","n"],
                ["e","t","a","e"],
                ["i","h","k","r"],
                ["i","f","l","v"]],
       words = ["oath","pea","eat","rain"]
Output: ["eat","oath"]
Explanation: "oath" and "eat" can be formed from the board;
             "pea" and "rain" cannot.
```

```
Input: board = [["a","b"],["c","d"]], words = ["abcb"]
Output: []
Explanation: "abcb" requires reusing 'b', which is not allowed.
```

```
Input: board = [["a"]], words = ["a","aa"]
Output: ["a"]
Explanation: Only single-letter word "a" is possible.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class TrieNode:
    def __init__(self):
```

```python
        self.children = {}
        self.word = None  # Stores full word at end node

class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        # STEP 1: Build Trie from words
        #   - Why Trie? Avoid recomputing shared prefixes;
        #     enables early pruning during DFS.
        root = TrieNode()
        for word in words:
            node = root
            for char in word:
                if char not in node.children:
                    node.children[char] = TrieNode()
                node = node.children[char]
            node.word = word  # Mark end of word

        result = []
        rows, cols = len(board), len(board[0])

        # STEP 2: DFS from every cell
        #   - Invariant: current path forms a prefix in Trie
        #   - Signal: reached a node where node.word is set
        def dfs(r, c, node):
            char = board[r][c]
            if char not in node.children:
                return

            next_node = node.children[char]
            if next_node.word:
                result.append(next_node.word)
                next_node.word = None  # Avoid duplicates

            # STEP 3: Mark visited & recurse
            #   - Why mark? Prevent reuse in same path
            #   - What breaks? Infinite loops or invalid paths
            board[r][c] = "#"  # Temporary mark
            for dr, dc in [(0,1), (1,0), (0,-1), (-1,0)]:
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols:
                    dfs(nr, nc, next_node)
            board[r][c] = char  # Backtrack
```

```python
            # Optional optimization: prune leaf nodes
            if not next_node.children:
                del node.children[char]

        # STEP 4: Launch DFS from every cell
        #   - Handle edge: empty board or words
        for r in range(rows):
            for c in range(cols):
                dfs(r, c, root)

        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    board1 = [["o","a","a","n"],
              ["e","t","a","e"],
              ["i","h","k","r"],
              ["i","f","l","v"]]
    words1 = ["oath","pea","eat","rain"]
    print("Test 1:", sorted(sol.findWords(board1, words1)) == ["eat","oath"])

    #  Test 2: Edge case - no matches
    board2 = [["a","b"],["c","d"]]
    words2 = ["abcb"]
    print("Test 2:", sol.findWords(board2, words2) == [])

    #  Test 3: Tricky - single cell, duplicate prevention
    board3 = [["a"]]
    words3 = ["a", "aa"]
    print("Test 3:", sorted(sol.findWords(board3, words3)) == ["a"])
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1** with `board1` and `words1 = ["oath","pea","eat","rain"]`.

1. **Build Trie**:
   - Insert "oath": root → 'o' → 'a' → 't' → 'h' (word="oath")
   - Insert "pea": root → 'p' → 'e' → 'a' (word="pea")
   - Insert "eat": root → 'e' → 'a' → 't' (word="eat")
   - Insert "rain": root → 'r' → 'a' → 'i' → 'n' (word="rain")
2. **Start DFS at (0,0) = 'o'**:
   - 'o' in root.children → go to node 'o'
   - From 'o', check neighbors: (0,1)='a' → valid
   - Path: "oa" → continue to 't' at (1,1), then 'h' at (1,2)
   - At 'h', node.word = "oath" → add to result, set word=None
3. **Later, start DFS at (1,0) = 'e'**:
   - 'e' in root.children → go to 'e'
   - (1,1)='t' → but need 'a' next! Wait—(0,0) is 'o', (1,0)='e'
   - Actually: (1,0)='e' → (0,0)='o' (not 'a'), (1,1)='t', (2,0)='i'
   - But (0,1)='a' is neighbor of (1,0)? No—(1,0) neighbors: up=(0,0)='o', right=(1,1)='t', down=(2,0)='i'
   - So how do we get "eat"?

   Correction: "eat" starts at (1,2)='a'? No.

   Actually:

   - (1,0) = 'e'
   - (0,0) = 'o' → skip
   - (1,1) = 't' → not 'a'
   - (2,0) = 'i' → not 'a'
   - Wait—look again:
     Row 0: o a a n
     Row 1: e t a e ← (1,2) = 'a'
     So "eat" = (1,0)='e' → (1,1)='t'? No, that's "et"

   Correct path for "eat":

- Start at (1,2) = ‘a’? No.
- Actually: (1,0)=‘e’ → (0,0)=‘o’ (no), (1,1)=‘t’ (no), (2,0)=‘i’ (no)
- But (0,1)=‘a’ is **not adjacent to (1,0)** — it’s diagonal! Not allowed.

Ah! Real path:

- Start at (1,2) = ‘a’ → no, need ‘e’ first.
- Look at (1,3) = ‘e’ → up to (0,3)=‘n’, left to (1,2)=‘a’, down to (2,3)=‘r’
- From (1,3)=‘e’ → left to (1,2)=‘a’ → up to (0,2)=‘a’ → no ‘t’

Correct path:

- (1,0) = ‘e’
- (0,0) = ‘o’ → skip
- (1,1) = ‘t’ → skip
- (2,0) = ‘i’ → skip
- **Wait!** (1,0)=‘e’ → **right is (1,1)=‘t’**, but we need ‘a’ next for “ea”

Mistake in board reading:
Board row 1: `["e","t","a","e"]`
So:

- (1,0) = ‘e’
- (1,1) = ‘t’
- (1,2) = ‘a’
- (1,3) = ‘e’

So “eat” = (1,0)=‘e’ → **(0,0) is ‘o’**, but **(1,0) up is (0,0), right is (1,1)=‘t’** — no ‘a’ adjacent?

Actually:

- (0,1) = ‘a’ is **above (1,1)**, not (1,0).

- But (1,0) has **no 'a' neighbor**.

Then how is "eat" found?

Answer: Start at (1,3) = 'e'

- (1,3) = 'e'
- Left → (1,2) = 'a'
- Up → (0,2) = 'a' → not 't'
- Down → (2,3) = 'r'
- Left again? No.

Correct path:

- Start at (1,0) = 'e'
- **Up**: (0,0) = 'o'
- **Right**: (1,1) = 't'
- **Down**: (2,0) = 'i'
- **Left**: invalid

Still no 'a'.

Re-express board with coordinates:

```
(0,0)=o  (0,1)=a  (0,2)=a  (0,3)=n
(1,0)=e  (1,1)=t  (1,2)=a  (1,3)=e
(2,0)=i  (2,1)=h  (2,2)=k  (2,3)=r
(3,0)=i  (3,1)=f  (3,2)=l  (3,3)=v
```

"eat" path:

- Start at (1,3) = 'e'
- Move **left** to (1,2) = 'a'
- Move **up** to (0,2) = 'a' → not 't'
- Move **down** to (2,2) = 'k' → no

Alternative:

- Start at (1,0) = ‘e’
- Is there an ‘a’ adjacent? **No**.

Wait! Look at (0,1) = ‘a’ — who is its neighbor?

- Down = (1,1) = ‘t’
- Left = (0,0) = ‘o’
- Right = (0,2) = ‘a’
- Up = invalid

So to get “eat”, we need: ‘e’ → ‘a’ → ‘t’
Where is an ‘e’ next to an ‘a’ that is next to a ‘t’?

Found it:

- (1,3) = ‘e’
- (1,2) = ‘a’ (left of ‘e’)
- (1,1) = ‘t’ (left of ‘a’)
  → Path: (1,3) → (1,2) → (1,1) = “e”-“a”-“t” = “eat” (but backwards!)

But word is “eat”, so must start with ‘e’.
So: start at (1,3)=‘e’ → (1,2)=‘a’ → (1,1)=‘t’ → forms “eat”

DFS will find this when starting at (1,3).

4. **DFS from (1,3)**:
   - char = ‘e’ → in root.children (from “eat” and “pea”? No, “pea” starts with ‘p’)
   - ‘e’ is in root (from “eat”) → proceed
   - next_node = node after ‘e’
   - Check neighbors of (1,3): (0,3)=‘n’, (1,2)=‘a’, (2,3)=‘r’
   - (1,2)=‘a’ is in next_node.children → recurse
   - Now at (1,2), node after “ea”
   - Neighbors: (1,1)=‘t’ → in children → recurse
   - At (1,1), node after “eat” → word=“eat” → add to result
5. **Backtracking**:
   - After exploring (1,1), mark restored to ‘t’

- Then (1,2) restored to ‘a’, then (1,3) to ‘e’

6. **Result**: [“oath”, “eat”] (order may vary)

Complexity Analysis

- **Time Complexity**: `O(M × N × 4^L)`
  > Where M×N is board size, L is max word length.
  > In worst case, DFS explores 4 directions up to L depth per cell.
  > **But Trie pruning reduces this significantly** — paths not in Trie are cut early.
- **Space Complexity**: `O(K × L)`
  > K = number of words, L = average word length.
  > Space used by Trie.
  > DFS recursion stack: `O(L)` — not dominant.

21. Minimum Height Trees

Pattern: Graphs — Topological Sorting (Leaf Pruning)

Problem Statement

> A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.
>
> Given a tree of `n` nodes labeled from `0` to `n - 1`, and an array `edges` where `edges[i] = [ai, bi]` indicates that there is an undirected edge between nodes `ai` and `bi` in the tree.
>
> You are asked to find **all the minimum height trees (MHTs)** and return a list of their root labels. You can return the answer in **any order**.
>
> The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Sample Input & Output

```
Input: n = 4, edges = [[1,0],[1,2],[1,3]]
Output: [1]
Explanation: Rooting the tree at node 1 gives height 1,
             while any leaf as root gives height 2.
```

```
Input: n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]
Output: [3,4]
Explanation: Both node 3 and node 4 yield trees of height 2,
             which is minimal.
```

```
Input: n = 1, edges = []
Output: [0]
Explanation: Only one node exists - it's the root and leaf.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
from collections import defaultdict, deque

class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
        # STEP 1: Handle trivial case
        #   - Single node has no edges; it's the only MHT root.
        if n == 1:
            return [0]

        # Build adjacency list and degree count
        graph = defaultdict(set)
        degree = [0] * n
        for u, v in edges:
            graph[u].add(v)
            graph[v].add(u)
            degree[u] += 1
            degree[v] += 1
```

```python
        # STEP 2: Initialize queue with all leaves (degree == 1)
        #   - Leaves cannot be optimal roots (they maximize height).
        #   - We iteratively prune leaves layer by layer.
        leaves = deque()
        for i in range(n):
            if degree[i] == 1:
                leaves.append(i)

        # STEP 3: Topological pruning (BFS from leaves inward)
        #   - Remove current leaves, reduce neighbor degrees.
        #   - New leaves emerge; repeat until  2 nodes remain.
        #   - Why  2? In a tree, the "center" is either 1 or 2 nodes.
        remaining_nodes = n
        while remaining_nodes > 2:
            leaves_count = len(leaves)
            remaining_nodes -= leaves_count

            for _ in range(leaves_count):
                leaf = leaves.popleft()
                for neighbor in graph[leaf]:
                    degree[neighbor] -= 1
                    graph[neighbor].discard(leaf)
                    if degree[neighbor] == 1:
                        leaves.append(neighbor)

        # STEP 4: Remaining nodes in queue are MHT roots
        return list(leaves)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.findMinHeightTrees(4, [[1,0],[1,2],[1,3]]) == [1]

    #  Test 2: Edge case - single node
    assert sol.findMinHeightTrees(1, []) == [0]

    #  Test 3: Tricky case - two centers
    result = sol.findMinHeightTrees(6, [[3,0],[3,1],[3,2],[3,4],[5,4]])
    assert sorted(result) == [3, 4]  # order doesn't matter
```

```
    print("  All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 3**: `n = 6`, `edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]`.

Initial Graph (adjacency list):

```
0: {3}
1: {3}
2: {3}
3: {0,1,2,4}
4: {3,5}
5: {4}
```

Degrees: `[1, 1, 1, 4, 2, 1]` → Leaves: nodes `0,1,2,5` (degree = 1).

Round 1 (prune leaves):
- Remove `0,1,2,5` → `remaining_nodes = 6 - 4 = 2`
- Update neighbors:
- Remove `0,1,2` from node `3` → degree[3] = 4 - 3 = **1**
- Remove `5` from node `4` → degree[4] = 2 - 1 = **1**
- Now both `3` and `4` have degree 1 → added to new leaves queue.

But wait! `remaining_nodes = 2`, so we **stop** before next pruning.

Final leaves queue: `[3, 4]` → returned as answer.

This matches expected output `[3,4]`.

Complexity Analysis

- **Time Complexity**: `O(n)`

    Each edge is processed exactly twice (once per endpoint).
    Each node is enqueued and dequeued at most once.
    Total operations scale linearly with `n`.

- **Space Complexity**: `O(n)`

    Adjacency list stores `2*(n-1)` edges → `O(n)`.
    Degree array and queue also use `O(n)` space.