Graphs

Chunk 1: Graph Traversal & BFS/DFS Patterns

Pattern: DFS / BFS on Grid (2D Array)

Used for connected components, flood fill, shortest path in unweighted grids.

How to Recognize:

- Problem involves a 2D grid (`matrix`, `grid`, `board`) with cells that can be visited or blocked.
- You're asked to:
  - Count islands
  - Find shortest path
  - Flood-fill a region
  - Traverse all reachable cells from a starting point
- Movement is typically up/down/left/right (sometimes diagonals)

Step-by-Step Thinking Process (The Recipe):

1. **Check bounds**: Ensure row/col indices are within grid dimensions.
2. **Check visited state**: Use a `visited` set or mark grid in-place (e.g., change value).
3. **Base case**: If cell is invalid, out of bounds, or already visited → return.
4. **Process current cell**: Do work (e.g., increment count, update distance).
5. **Recursive/Queue-based traversal**: Call DFS/BFS on neighbors.
6. **Return result**: Based on problem (count, path length, etc.).

Common Pitfalls & Edge Cases:

- Forgetting to mark visited nodes → infinite loop.
- Not handling edge cases like empty grid, single cell, or no valid path.
- Using recursion for large grids → stack overflow (use iterative BFS instead).
- Misinterpreting movement rules (e.g., only 4-directional vs 8-directional).

Problem 1: Flood Fill

Summary:

Given a 2D image represented as a matrix of integers, start at a pixel `(sr, sc)` and replace all adjacent pixels with the same color as the starting pixel with a new color. Adjacency is defined as 4-directional.

Pattern(s):

- **DFS / BFS on Grid**
- **Connected Components**

Solution with Inline Comments:

```
def floodFill(image, sr, sc, newColor):
    # Get dimensions of the image
    rows, cols = len(image), len(image[0])

    # Original color at the starting pixel
    original_color = image[sr][sc]

    # If the new color is same as old, no need to do anything
    if original_color == newColor:
        return image

    # Use DFS to fill the region
    def dfs(r, c):
        # Base case: check bounds and if pixel has correct original color
        if (r < 0 or r >= rows or
            c < 0 or c >= cols or
            image[r][c] != original_color):
            return

        # Change the color of current pixel
        image[r][c] = newColor

        # Recursively apply DFS to 4 neighbors
        dfs(r + 1, c)  # down
        dfs(r - 1, c)  # up
```

```
        dfs(r, c + 1)  # right
        dfs(r, c - 1)  # left

    # Start DFS from the initial pixel
    dfs(sr, sc)

    return image
```

Official Example Walkthrough:

Input:
`image = [[1,1,1],[1,1,0],[1,0,1]]`, `sr = 1, sc = 1, newColor = 2`

Step-by-step: 1. Start at `(1,1)` — original color = `1`, new color = `2`. 2. Mark `(1,1)` → `2`. 3. Visit neighbors: - `(2,1)` → color `0` `1` → skip - `(0,1)` → color `1` → mark as `2` - From `(0,1)`: visit `(0,0)` → `1` → mark as `2` - From `(0,0)`: neighbors `(0,1)` already done; `(1,0)` → `1` → mark as `2` - From `(1,0)`: neighbors `(2,0)` → `1` → mark as `2` - From `(2,0)`: neighbor `(2,1)` → `0` → skip - `(1,2)` → `0` → skip - `(1,0)` → already processed via chain

Final Image:

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

Output matches expected.

Complexity:

- **Time:** O(m×n) — each cell visited once
- **Space:** O(m×n) — worst-case recursion depth (stack) if entire grid is filled

---

Problem 2: 01 Matrix

Summary:

Given a binary matrix, find the distance from each cell to the nearest `0`. Distance is Manhattan distance.

Pattern(s):

- **Multi-source BFS**
- **Shortest Path in Unweighted Grid**

Solution with Inline Comments:

```python
from collections import deque

def updateMatrix(mat):
    rows, cols = len(mat), len(mat[0])

    # Initialize result matrix with infinity (or large number)
    dist = [[float('inf')] * cols for _ in range(rows)]

    # Queue for BFS: store (row, col, distance)
    queue = deque()

    # Add all '0' cells as sources with distance 0
    for i in range(rows):
        for j in range(cols):
            if mat[i][j] == 0:
                dist[i][j] = 0
                queue.append((i, j))

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

    # Multi-source BFS
    while queue:
        r, c = queue.popleft()
        current_dist = dist[r][c]

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

            # Check bounds and if we can update distance
            if (0 <= nr < rows and 0 <= nc < cols and
                dist[nr][nc] > current_dist + 1):
```

```
                # Update distance and add to queue
                dist[nr][nc] = current_dist + 1
                queue.append((nr, nc))

    return dist
```

Official Example Walkthrough:

Input:
`mat = [[0,0,0],[0,1,0],[1,1,1]]`

Step-by-step: 1. All `0`s go into queue with distance `0`. 2. Process `(0,0)`, `(0,1)`, `(0,2)`, `(1,0)`, `(1,2)` → update their neighbors: - From `(0,0)`: `(1,0)` already in queue → skip - From `(0,1)`: `(1,1)` → set to `1`, add to queue - From `(1,0)`: `(2,0)` → set to `1`, add to queue - From `(1,2)`: `(2,2)` → set to `1`, add to queue 3. Now process `(1,1)` → update `(2,1)` → set to 2 4. Process `(2,0)` → `(2,1)` already updated 5. Process `(2,2)` → `(2,1)` already updated 6. Final distances:

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

Matches expected output.

Complexity:

- **Time:** O(m×n) — each cell processed at most once
- **Space:** O(m×n) — for `dist` array and queue (max size ~m×n)

Problem 3: Number of Islands

Summary:

Given a 2D grid of `'1'`s (land) and `'0'`s (water), count the number of distinct islands (connected groups of `'1'`s).

Pattern(s):

- **DFS/BFS on Grid**
- **Connected Components**

Solution with Inline Comments:

```python
def numIslands(grid):
    if not grid or not grid[0]:
        return 0

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

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

    def dfs(r, c):
        # Base case: out of bounds or water or already visited
        if (r < 0 or r >= rows or
            c < 0 or c >= cols or
            grid[r][c] == '0' or
            visited[r][c]):
            return

        # Mark as visited
        visited[r][c] = True

        # Explore all 4 neighbors
        for dr, dc in directions:
            dfs(r + dr, c + dc)

    # Iterate through every cell
    for i in range(rows):
        for j in range(cols):
            # If land not visited, start DFS → new island
            if grid[i][j] == '1' and not visited[i][j]:
                dfs(i, j)
                islands += 1
```

```python
    return islands
```

Official Example Walkthrough:

Input:

```
[
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
```

Step-by-step: - Start at `(0,0)` → land → DFS covers entire top-left island → count = 1 - Move to `(0,4)` → water → skip - Continue until `(1,1)` → already visited - At `(1,3)` → land → but not visited? Wait: `(1,3)` is part of first island? - Actually, `(1,3)` connects to `(0,3)` → which connects to `(0,0)` → so it's one island! - Only one connected component exists.

Wait! But input shows two separate islands?

Let me recheck:

```
Row 0: 1 1 1 1 0
Row 1: 1 1 0 1 0
Row 2: 1 1 0 0 0
Row 3: 0 0 0 0 0
```

So: - Top-left block: all ones connected → one island - `(1,3)` is isolated from others? No — `(0,3)` → `(1,3)` → yes, connected!

But `(1,3)` is connected to `(0,3)` → which is part of main island → so still one island.

Wait — this is actually **one island**!

But expected output is `1`.

Yes — only **one** connected group of `'1'`.

If we had:

```
["1","1","0"],
["0","0","1"]
```

→ Two islands.

So this example gives `1`.

Complexity:

- **Time:** O(m×n) — each cell visited once
- **Space:** O(m×n) — `visited` array + recursion stack (worst-case depth m×n)

Problem 4: Rotting Oranges

Summary:

In a grid with `0` (empty), `1` (fresh orange), `2` (rotten orange), every minute rotten oranges infect adjacent fresh oranges. Return minutes until all oranges rot, or `-1` if impossible.

Pattern(s):

- **Multi-source BFS**
- **Level-order traversal (time progression)**

Solution with Inline Comments:

```python
from collections import deque

def orangesRotting(grid):
    rows, cols = len(grid), len(grid[0])

    # Count fresh oranges and collect all rotten ones
    fresh_count = 0
    queue = deque()

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
                fresh_count += 1
            elif grid[i][j] == 2:
```

```python
                queue.append((i, j))

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

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

    # Multi-source BFS
    while queue and fresh_count > 0:
        # Process all rotten oranges at current time level
        level_size = len(queue)

        for _ in range(level_size):
            r, c = queue.popleft()

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

                # Check bounds and if fresh orange
                if (0 <= nr < rows and 0 <= nc < cols and
                    grid[nr][nc] == 1):

                    # Rot the orange
                    grid[nr][nc] = 2
                    fresh_count -= 1
                    queue.append((nr, nc))

        minutes += 1  # One minute passed

    # If any fresh orange remains, return -1
    return minutes if fresh_count == 0 else -1
```

Official Example Walkthrough:

Input:

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

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

Step-by-step: - Rotten oranges: `(0,0)` - Fresh: 5 - Minute 0: process `(0,0)` → infect `(0,1)` and `(1,0)` - Minute 1: process `(0,1)`, `(1,0)` → infect `(0,2)`, `(1,1)` - Minute 2: process `(0,2)`, `(1,1)` → infect `(2,1)` - Minute 3: process `(2,1)` → no new infections - All fresh oranges now rotten → return `3`

Output: `3`

Complexity:

- **Time:** O(m×n) — each cell processed once
- **Space:** O(m×n) — queue holds up to all rotten oranges

Problem 5: Word Search

Summary:

Given a 2D board and a word, determine if the word exists in the grid by moving up/down/left/right. Each letter can be used only once per path.

Pattern(s):

- **DFS with Backtracking**
- **Grid Traversal**

Solution with Inline Comments:

```python
def exist(board, word):
    if not board or not board[0]:
        return False

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

    # Directions: up, down, left, right
```

```python
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    def dfs(r, c, index):
        # Base case: if we've matched all characters
        if index == len(word):
            return True

        # Check bounds and character match
        if (r < 0 or r >= rows or
            c < 0 or c >= cols or
            board[r][c] != word[index]):
            return False

        # Temporarily mark current cell as visited (by changing char)
        temp = board[r][c]
        board[r][c] = '#'  # mark as visited

        # Try all 4 directions
        for dr, dc in directions:
            if dfs(r + dr, c + dc, index + 1):
                board[r][c] = temp  # backtrack
                return True

        # Backtrack: restore original character
        board[r][c] = temp
        return False

    # Try starting from every cell
    for i in range(rows):
        for j in range(cols):
            if dfs(i, j, 0):
                return True

    return False
```

Official Example Walkthrough:

Input:
`board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, `word = "ABCCED"`

Step-by-step: - Start at `(0,0)` → 'A' → good - Go to `(0,1)` → 'B' → good - Go to `(0,2)` →

'C' → good - Go to `(0,3)` → 'E' → good - Go to `(1,3)` → 'S' → bad - Backtrack → try `(1,2)` → 'C' → good - Then `(2,2)` → 'E' → good - Then `(2,3)` → 'E' → good → word found!

Returns `True`

Complexity:

- **Time:** O(m×n×4^L) where L = length of word (each step has up to 4 choices)
- **Space:** O(L) — recursion depth (path length)

Problem 6: Pacific Atlantic Water Flow

Summary:

Given a height map, return all cells from which water can flow to both Pacific (top/left edges) and Atlantic (bottom/right edges). Water flows to adjacent cells with equal or lower height.

Pattern(s):

- **BFS/DFS from boundaries**
- **Reverse flow simulation**

Solution with Inline Comments:

```python
from collections import deque

def pacificAtlantic(heights):
    if not heights or not heights[0]:
        return []

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

    # Sets to track cells reachable from each ocean
    pacific_reachable = set()
    atlantic_reachable = set()

    # Directions: up, down, left, right
```

```python
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    def bfs(queue, reachable_set):
        while queue:
            r, c = queue.popleft()
            reachable_set.add((r, c))

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

                # Check bounds
                if (0 <= nr < rows and 0 <= nc < cols and
                    (nr, nc) not in reachable_set and
                    heights[nr][nc] >= heights[r][c]):  # reverse flow

                    queue.append((nr, nc))

    # Initialize queues: start from top row (pacific) and leftmost column
    pacific_queue = deque()
    atlantic_queue = deque()

    for c in range(cols):
        pacific_queue.append((0, c))  # top row
        atlantic_queue.append((rows - 1, c))  # bottom row

    for r in range(rows):
        pacific_queue.append((r, 0))  # leftmost col
        atlantic_queue.append((r, cols - 1))  # rightmost col

    # Run BFS from both oceans
    bfs(pacific_queue, pacific_reachable)
    bfs(atlantic_queue, atlantic_reachable)

    # Return intersection of both sets
    return list(pacific_reachable & atlantic_reachable)
```

Official Example Walkthrough:

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]]

Step-by-step: - Pacific starts from top and left edges → propagate inward where height previous - Atlantic starts from bottom and right edges → same - Cells reachable from both → returned

Example output: `[[0,4],[1,3],[1,4],[2,3],[2,4],[3,0],[4,0],[4,1],[4,2],[4,3],[4,4]]`

Correct.

Complexity:

- **Time:** O(m×n) — each cell visited at most twice
- **Space:** O(m×n) — sets and queues

Problem 7: Shortest Path to Get Food

Summary:

In a grid with `'X'` (walls), `'.'` (empty), `'*'` (food), find minimum steps from robot (start) to food. Can move in 4 directions.

Pattern(s):

- **BFS (Shortest Path in Unweighted Grid)**

Solution with Inline Comments:

```python
from collections import deque

def getFood(grid):
    rows, cols = len(grid), len(grid[0])

    # Find robot position
    start_r, start_c = None, None
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == '*':
                start_r, start_c = i, j
```

```python
                break
        if start_r is not None:
            break

    # BFS setup
    queue = deque([(start_r, start_c, 0)])  # (row, col, steps)
    visited = {(start_r, start_c)}

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

    while queue:
        r, c, steps = queue.popleft()

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

            # Check bounds and wall
            if (0 <= nr < rows and 0 <= nc < cols and
                grid[nr][nc] != 'X' and
                (nr, nc) not in visited):

                if grid[nr][nc] == '#':  # food found!
                    return steps + 1

                visited.add((nr, nc))
                queue.append((nr, nc, steps + 1))

    return -1  # no path to food
```

Official Example Walkthrough:

Input:

```
grid = [
  ["X","X","X","X","X","X"],
  ["X","*","O","O","O","X"],
  ["X","O","O","#","O","X"],
  ["X","X","X","X","X","X"]
]
```

- Robot at `(1,1)`, food at `(2,3)`
- BFS:
  - Step 0: `(1,1)`
  - Step 1: `(1,2)`, `(2,1)`
  - Step 2: `(1,3)`, `(2,2)`
  - Step 3: `(2,3)` → food → return `3`

Output: `3`

Complexity:

- **Time:** O(m×n)
- **Space:** O(m×n)

Chunk 2: Topological Sort, Union-Find, Shortest Path, Backtracking + DFS

We'll now cover the following problems:

- **Topological Sort**
  - Course Schedule → Detect cycle (DFS/Kahn's)
  - Course Schedule II → Return ordering (DFS/Kahn's)
  - Alien Dictionary → Build graph from dictionary order + topo sort
- **Union-Find**
  - Accounts Merge → DSU with emails
  - Graph Valid Tree → DSU cycle detection
  - Number of Connected Components in Undirected Graph → DSU
- **Shortest Path**
  - Word Ladder → BFS
  - Minimum Knight Moves → BFS on infinite grid (symmetry trick)
  - Bus Routes → BFS with routes as graph
  - Cheapest Flights Within K Stops → BFS/Dijkstra with (node, stops) state
- **Backtracking + DFS**
  - Word Search → DFS with visited (already covered in Chunk 1 — skip)
  - Word Search II → DFS + Trie for pruning

  We'll reprocess **Word Search II** here since it's a more advanced variant.

Pattern: Topological Sort

Used when tasks have dependencies; output is an ordering where each task comes after its prerequisites.

How to Recognize:

- Problem involves dependencies: "A must happen before B"
- You're asked to find an order of execution or detect if impossible
- Graph is **Directed Acyclic Graph (DAG)** — no cycles allowed

Step-by-Step Thinking Process (The Recipe):

1. **Build adjacency list and indegree array** from edges.
2. **Find all nodes with indegree 0** → they can start first.
3. **Use queue (Kahn's Algorithm)**: repeatedly remove indegree 0 nodes.
4. For each removed node, decrease indegree of its neighbors.
5. If all nodes are processed → valid topological order exists.
6. Else → cycle → return empty list.

Common Pitfalls & Edge Cases:

- Not checking for cycles → return invalid order.
- Using DFS without proper backtracking → missing nodes.
- Forgetting that multiple valid orders exist → any valid one is acceptable.
- Handling self-loops or duplicate edges.

Problem 1: Course Schedule

Summary:

Given `n` courses labeled 0 to n−1 and a list of prerequisite pairs, determine if it's possible to finish all courses.

Pattern(s):

- **Topological Sort (Cycle Detection)**

Solution with Inline Comments:

```python
from collections import deque

def canFinish(numCourses, prerequisites):
    # Build adjacency list and indegree array
    graph = [[] for _ in range(numCourses)]
    indegree = [0] * numCourses

    for course, prereq in prerequisites:
        graph[prereq].append(course)  # prereq -> course
        indegree[course] += 1

    # Queue for nodes with indegree 0
    queue = deque()
    for i in range(numCourses):
        if indegree[i] == 0:
            queue.append(i)

    # Count processed nodes
    processed = 0

    while queue:
        node = queue.popleft()
        processed += 1

        # Reduce indegree of neighbors
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    # If all courses processed → no cycle
    return processed == numCourses
```

Official Example Walkthrough:

Input:
`numCourses = 2`, `prerequisites = [[1,0]]`

- Course 1 depends on course 0.

- Graph: `0 → 1`
- Indegree: `[0,1]`
- Start: `queue = [0]`
- Process 0 → reduce indegree of 1 → becomes 0 → add to queue
- Process 1 → done
- `processed = 2` → equals `numCourses` → return `True`

Output: `true`

Complexity:

- **Time:** O(V + E) — visit each node and edge once
- **Space:** O(V + E) — graph + indegree + queue

Problem 2: Course Schedule II

Summary:

Return **any valid order** of courses to complete all courses, or return an empty array if impossible.

Pattern(s):

- **Topological Sort (Ordering)**

Solution with Inline Comments:

```python
from collections import deque

def findOrder(numCourses, prerequisites):
    # Build graph and indegree
    graph = [[] for _ in range(numCourses)]
    indegree = [0] * numCourses

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

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

    result = []

    while queue:
        node = queue.popleft()
        result.append(node)

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

    # Check if all courses are included
    return result if len(result) == numCourses else []
```

Official Example Walkthrough:

Input:
`numCourses = 4`, `prerequisites = [[1,0],[2,0],[3,1],[3,2]]`

- Dependencies: 0→1, 0→2, 1→3, 2→3
- Graph: `0 → 1 → 3`, `0 → 2 → 3`
- Indegree: `[0,1,1,2]`
- Start: `queue = [0]`
- Process 0 → update 1 and 2 → both indegree=0 → add to queue
- Process 1 → update 3 → indegree[3]=1
- Process 2 → update 3 → indegree[3]=0 → add to queue
- Process 3 → done
- Result: `[0,1,2,3]`

Output: `[0,1,2,3]` (or any valid order like `[0,2,1,3]`)

Complexity:

- **Time:** O(V + E)
- **Space:** O(V + E)

Problem 3: Alien Dictionary

Summary:

Given a list of words sorted lexicographically in an alien language, reconstruct the order of characters.

Pattern(s):

- **Topological Sort**
- **Graph Construction from String Comparison**

Solution with Inline Comments:

```python
from collections import defaultdict, deque

def alienOrder(words):
    # Step 1: Initialize graph and indegree
    graph = defaultdict(set)
    indegree = {char: 0 for word in words for char in word}

    # Step 2: Compare adjacent words to build edges
    for i in range(len(words) - 1):
        w1, w2 = words[i], words[i + 1]
        min_len = min(len(w1), len(w2))

        found_diff = False
        for j in range(min_len):
            c1, c2 = w1[j], w2[j]
            if c1 != c2:
                # Add edge: c1 -> c2
                if c2 not in graph[c1]:
                    graph[c1].add(c2)
                    indegree[c2] += 1
                found_diff = True
                break
```

```python
        # If shorter word is prefix of longer → invalid (e.g., "abc" before "ab")
        if not found_diff and len(w1) > len(w2):
            return ""

    # Step 3: Kahn's algorithm
    queue = deque([c for c in indegree if indegree[c] == 0])
    result = []

    while queue:
        node = queue.popleft()
        result.append(node)

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

    # Check if all characters are used
    return ''.join(result) if len(result) == len(indegree) else ""
```

Official Example Walkthrough:

Input: `words = ["wrt","wrf","er","ett","rft"]`

- Compare `"wrt"` and `"wrf"` → 't' vs 'f' → add `t → f`
- Compare `"wrf"` and `"er"` → 'w' vs 'e' → add `w → e`
- Compare `"er"` and `"ett"` → 'e' vs 'e', then 'r' vs 't' → add `r → t`
- Compare `"ett"` and `"rft"` → 'e' vs 'r' → add `e → r`

Edges: `t→f`, `w→e`, `r→t`, `e→r`

Graph: - `w → e` - `e → r` - `r → t` - `t → f`

Indegree: `w:0`, `e:1`, `r:1`, `t:1`, `f:1`

Start: `queue = [w]` - Process `w` → `e` now indegree 0 - Process `e` → `r` → 0 - Process `r` → `t` → 0 - Process `t` → `f` → 0 - Process `f`

Result: `"wertf"`

Output: `"wertf"` (or valid permutation)

Complexity:

- **Time:** O(C) where C = total characters across all words
- **Space:** O(1) — at most 26 letters, so constant

Pattern: Union-Find (Disjoint Set Union)

Used to manage disjoint sets, detect cycles, merge groups, count components.

How to Recognize:

- Problems about connected components, merging groups, detecting cycles in undirected graphs.
- You're told to "merge" or "union" things, or check if two items are connected.

Step-by-Step Thinking Process (The Recipe):

1. **Initialize parent and rank arrays** (each node points to itself).
2. **Find root with path compression** (`find(x)`).
3. **Union by rank** (`union(x,y)`): attach smaller tree under larger.
4. Use `union` to connect components.
5. Use `find` to check connectivity or detect cycles.

Common Pitfalls & Edge Cases:

- Not using path compression/rank → slow performance.
- Incorrect union logic (e.g., always attaching x to y).
- Not handling self-loops or duplicate edges.
- Returning wrong result (e.g., returning `True` when union fails due to cycle).

Problem 4: Accounts Merge

Summary:

Given accounts (name + emails), merge accounts with common emails into one.

Pattern(s):

- **Union-Find (DSU)**
- **Graph of Emails**

Solution with Inline Comments:

```python
class UnionFind:
    def __init__(self):
        self.parent = {}

    def find(self, x):
        if x not in self.parent:
            self.parent[x] = x
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px != py:
            self.parent[px] = py

def accountsMerge(accounts):
    uf = UnionFind()
    email_to_name = {}

    # Step 1: Union emails within same account
    for account in accounts:
        name = account[0]
        emails = account[1:]

        # Link all emails in this account
        for email in emails:
            email_to_name[email] = name
            uf.union(emails[0], email)  # link all to first email

    # Step 2: Group emails by their root parent
    groups = defaultdict(list)
    for email in email_to_name:
        root = uf.find(email)
```

```
        groups[root].append(email)

    # Step 3: Sort and format output
    result = []
    for group in groups.values():
        result.append([email_to_name[group[0]]] + sorted(group))

    return result
```

Official Example Walkthrough:

Input:

```
accounts = [["John","johnsmith@mail.com","john_newyork@mail.com"],
            ["John","johnsmith@mail.com","john00@mail.com"],
            ["Mary","mary@mail.com"]]
```

- John's emails: `johnsmith@mail.com`, `john_newyork@mail.com`, `john00@mail.com`
- First account: union `johnsmith  john_newyork`
- Second account: union `johnsmith  john00` → so all three linked
- Mary: standalone

Groups: - Root of johnsmith → all three emails grouped - Mary → her own

Output:

```
[
  ["John", "john00@mail.com", "john_newyork@mail.com", "johnsmith@mail.com"],
  ["Mary", "mary@mail.com"]
]
```

Correct.

Complexity:

- **Time:** O(A × M × (M)) where A = accounts, M = avg emails per account, = inverse Ackermann
- **Space:** O(E) — number of emails

Problem 5: Graph Valid Tree

Summary:

Given `n` nodes and `edges`, determine if the graph forms a valid tree (connected, acyclic, n-1 edges).

Pattern(s):

- **Union-Find (Cycle Detection)**
- **Connectivity Check**

Solution with Inline Comments:

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False  # cycle detected
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        return True

def validTree(n, edges):
    # Must have exactly n-1 edges
    if len(edges) != n - 1:
        return False
```

```python
    uf = UnionFind(n)

    # Try to union all edges
    for u, v in edges:
        if not uf.union(u, v):
            return False  # cycle found

    return True
```

Official Example Walkthrough:

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

- Edges = 4 → n-1 = 4 → ok
- Union: 0-1, 0-2, 0-3, 1-4 → no cycles
- All nodes connected? Yes
- Return `True`

Output: `true`

Complexity:

- **Time:** O(E × (N))
- **Space:** O(N)

Problem 6: Number of Connected Components in Undirected Graph

Summary:

Count how many connected components exist in an undirected graph.

Pattern(s):

- **Union-Find**
- **Component Counting**

Solution with Inline Comments:

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.components = n  # count of distinct components

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        self.components -= 1

def countComponents(n, edges):
    uf = UnionFind(n)

    for u, v in edges:
        uf.union(u, v)

    return uf.components
```

Official Example Walkthrough:

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

- Initially 5 components
- Union 0-1 → 4
- Union 1-2 → 3
- Union 3-4 → 2
- Final count: 2

Output: 2

Complexity:

- **Time:** O(E × (N))
- **Space:** O(N)

Pattern: Shortest Path in Weighted Graphs

BFS for unweighted, Dijkstra for weighted, with constraints (e.g., max stops).

How to Recognize:

- Find shortest path in graph with weights or steps.
- Constraints: max stops, limited moves, etc.
- Often use **priority queue (Dijkstra)** or **BFS with state tracking**.

Problem 7: Word Ladder

Summary:

Transform `beginWord` to `endWord` by changing one letter at a time, with each intermediate word in `wordList`.

Pattern(s):

- **BFS (Unweighted Shortest Path)**
- **Graph via Word Transformation**

Solution with Inline Comments:

```python
from collections import deque

def ladderLength(beginWord, endWord, wordList):
    wordSet = set(wordList)

    # If endWord not in wordList, impossible
    if endWord not in wordSet:
        return 0

    # BFS queue: (word, steps)
    queue = deque([(beginWord, 1)])
    visited = {beginWord}

    # Letters a-z
    alphabet = 'abcdefghijklmnopqrstuvwxyz'

    while queue:
        word, steps = queue.popleft()

        # Try changing each character
        for i in range(len(word)):
            for c in alphabet:
                new_word = word[:i] + c + word[i+1:]

                if new_word == endWord:
                    return steps + 1

                if new_word in wordSet and new_word not in visited:
                    visited.add(new_word)
                    queue.append((new_word, steps + 1))

    return 0
```

Official Example Walkthrough:

```
**Input:** `beginWord = "hit"`,
            `endWord = "cog"`,
            `wordList = ["hot","dot","dog","lot","log","cog"]`
```

- hit → hot → dot → dog → cog → 5 steps
- But output counts number of words in sequence → 5

Output: `5`

Complexity:

- **Time:** O(L × M × 26) where L = word length, M = number of words
- **Space:** O(M)

Problem 8: Minimum Knight Moves

Summary:

Find minimum moves for knight to go from `(0,0)` to `(x,y)` on infinite chessboard.

Pattern(s):

- **BFS on Infinite Grid**
- **Symmetry Optimization**

Solution with Inline Comments:

```python
from collections import deque

def minKnightMoves(x, y):
    # Use symmetry: only consider positive quadrant
    x, y = abs(x), abs(y)

    # Directions for knight
    directions = [(2,1),(2,-1),(-2,1),(-2,-1),
                  (1,2),(1,-2),(-1,2),(-1,-2)]

    # BFS
    queue = deque([(0, 0, 0)])  # (row, col, steps)
    visited = {(0, 0)}

    while queue:
        r, c, steps = queue.popleft()
```

```python
        if r == x and c == y:
            return steps

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

            # Use symmetry: mirror negative coordinates
            nr, nc = abs(nr), abs(nc)

            if (nr, nc) not in visited and nr <= x + 2 and nc <= y + 2:
                visited.add((nr, nc))
                queue.append((nr, nc, steps + 1))

    return -1
```

Official Example Walkthrough:

Input: `x = 2, y = 1`

- Knight at (0,0) → (2,1) in 1 move? No → try (1,2), (2,1) → yes!
- From (0,0) → (2,1) → valid move → 1 step

Output: `1`

Complexity:

- **Time:** O(max(|x|,|y|)^2) — bounded by symmetry
- **Space:** O(max(|x|,|y|)^2)

Problem 9: Bus Routes

Summary:

Given bus routes, find minimum number of buses to take from source to target stop.

Pattern(s):

- **BFS with Route-to-Stop Mapping**
- **Graph: Stop Bus**

Solution with Inline Comments:

```python
from collections import deque, defaultdict

def numBusesToDestination(routes, source, target):
    if source == target:
        return 0

    # Map each stop to list of buses (routes)
    stop_to_buses = defaultdict(list)
    for bus_id, route in enumerate(routes):
        for stop in route:
            stop_to_buses[stop].append(bus_id)

    # BFS: (bus_id, stops_taken)
    queue = deque()
    visited_buses = set()

    # Start from all buses that serve source
    for bus_id in stop_to_buses[source]:
        queue.append((bus_id, 1))
        visited_buses.add(bus_id)

    while queue:
        bus_id, count = queue.popleft()

        # Visit all stops on this bus
        for stop in routes[bus_id]:
            if stop == target:
                return count

            # Take other buses from this stop
            for next_bus_id in stop_to_buses[stop]:
                if next_bus_id not in visited_buses:
                    visited_buses.add(next_bus_id)
                    queue.append((next_bus_id, count + 1))
```

```python
    return -1
```

Official Example Walkthrough:

Input: `routes = [[1,2,7],[3,6,7]], source = 1, target = 6`

- Bus 0 serves stops 1,2,7
- Bus 1 serves stops 3,6,7
- Source = 1 → take bus 0 → reach stop 7
- Stop 7 → take bus 1 → reach stop 6 → target!

So: 2 buses

Output: 2

Complexity:

- **Time:** O(B×S) where B = buses, S = stops
- **Space:** O(B×S)

Problem 10: Cheapest Flights Within K Stops

Summary:

Find cheapest price from `src` to `dst` with at most `k` stops.

Pattern(s):

- **BFS / Dijkstra with (node, stops, cost) state**
- **State-space search**

Solution with Inline Comments:

```python
from collections import deque
import heapq

def findCheapestPrice(n, flights, src, dst, k):
    # Build adjacency list
    graph = [[] for _ in range(n)]
    for u, v, w in flights:
        graph[u].append((v, w))

    # Min-heap: (cost, stops, node)
    heap = [(0, 0, src)]
    min_cost = [float('inf')] * n

    while heap:
        cost, stops, node = heapq.heappop(heap)

        if node == dst:
            return cost

        if stops > k:
            continue

        if cost > min_cost[node]:
            continue

        for neighbor, flight_cost in graph[node]:
            new_cost = cost + flight_cost
            new_stops = stops + 1

            if new_cost < min_cost[neighbor]:
                min_cost[neighbor] = new_cost
                heapq.heappush(heap, (new_cost, new_stops, neighbor))

    return -1
```

Official Example Walkthrough:

Input: `n = 3`, `flights = [[0,1,100],[1,2,100],[0,2,500]]`, `src = 0`, `dst = 2`, `k = 1`

- $0 \to 1 \to 2$: cost = 200, stops = 1 → valid
- $0 \to 2$: cost = 500, stops = 0 → but only 1 stop allowed → valid
- Min = 200

Output: 200

Complexity:

- **Time:** O(E × log V × (K+1)) — each edge may be pushed multiple times
- **Space:** O(V + E)

Problem 11: Word Search II

Summary:

Given a board and list of words, return all words found in the board.

Pattern(s):

- **Trie + DFS with Backtracking**
- **Pruning via Prefix Matching**

Solution with Inline Comments:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.word = None  # store full word at leaf

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for c in word:
            if c not in node.children:
                node.children[c] = TrieNode()
            node = node.children[c]
        node.word = word
```

```python
def findWords(board, words):
    if not board or not board[0]:
        return []

    rows, cols = len(board), len(board[0])
    trie = Trie()
    for word in words:
        trie.insert(word)

    result = set()
    directions = [(0,1),(0,-1),(1,0),(-1,0)]

    def dfs(r, c, node):
        char = board[r][c]
        current_node = node.children.get(char)

        if not current_node:
            return

        # Found a word
        if current_node.word:
            result.add(current_node.word)

        # Mark visited
        board[r][c] = '#'

        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if (0 <= nr < rows and 0 <= nc < cols and
                board[nr][nc] != '#'):
                dfs(nr, nc, current_node)

        # Backtrack
        board[r][c] = char

    # Try starting from every cell
    for i in range(rows):
        for j in range(cols):
            dfs(i, j, trie.root)

    return list(result)
```

Official Example Walkthrough:

Input: `board = [["o","a","a","n"], ["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]]`, `words = ["oath","pea","eat","rain"]`

- Words "oath", "eat" found → return `["oath","eat"]`

Output: `["oath","eat"]`

Complexity:

- **Time:** O(M × N × 4^L) where L = max word length
- **Space:** O(W × L) — trie size

Chunk 3: Dynamic Programming on Graphs

We'll now cover the following two advanced problems that combine **Graph Traversal** with **Dynamic Programming**:

- Longest Increasing Path in a Matrix

- Minimum Height Trees

These are excellent examples of **DP on graphs**, where we use memoization or iterative pruning (like topological sort) to solve path optimization problems.

Pattern: Dynamic Programming on Graphs

> *Used when you need to compute optimal paths, longest/shortest sequences, or values that depend on neighbors — often with constraints like monotonicity.*

How to Recognize:

- Problem involves a grid or graph where each node has a value.
- You're asked to find the **longest increasing path**, **minimum height tree**, or similar.
- The solution depends on values of neighboring nodes (e.g., `current > neighbor`).
- Recursion with memoization is natural → avoid recomputation.

Step-by-Step Thinking Process (The Recipe):

1. **Define state**: What does `dp[i][j]` represent? (e.g., longest path starting at `(i,j)`)
2. **Base case**: If no valid next step → return 1.
3. **Transition**: For each neighbor, if condition holds (e.g., `matrix[i][j] < neighbor`), take max of `1 + dp[neighbor]`.
4. **Memoize**: Cache results to avoid recalculating.
5. **Iterate over all starting points** and return maximum.

Common Pitfalls & Edge Cases:

- Forgetting to **memoize** → exponential time.
- Misunderstanding directionality (e.g., increasing vs decreasing).
- Not handling edge cases like single cell, or all equal values.
- Using recursion without stack limit awareness (use iterative DP or increase recursion limit if needed).

Problem 1: Longest Increasing Path in a Matrix

Summary:

Given an `m x n` matrix of integers, find the length of the longest increasing path where you can only move up/down/left/right, and each step must go to a strictly larger number.

Pattern(s):

- **DFS + Memoization**
- **DP on Grid**

Solution with Inline Comments:

```python
def longestIncreasingPath(matrix):
    if not matrix or not matrix[0]:
        return 0

    rows, cols = len(matrix), len(matrix[0])
```

```python
# Memoization cache: dp[r][c] = longest path starting at (r,c)
memo = {}

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

def dfs(r, c):
    # If already computed, return cached result
    if (r, c) in memo:
        return memo[(r, c)]

    # Base case: start with path length 1 (just current cell)
    max_length = 1

    # Try all 4 neighbors
    for dr, dc in directions:
        nr, nc = r + dr, c + dc

        # Check bounds and increasing condition
        if (0 <= nr < rows and 0 <= nc < cols and
            matrix[nr][nc] > matrix[r][c]):

            # Recursively compute path from neighbor
            length = 1 + dfs(nr, nc)
            max_length = max(max_length, length)

    # Cache result
    memo[(r, c)] = max_length
    return max_length

# Try starting from every cell
result = 0
for i in range(rows):
    for j in range(cols):
        result = max(result, dfs(i, j))

return result
```

Official Example Walkthrough:

Input:

```
matrix = [
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
```

Step-by-step: - Start at `(0,0)` = 9 → can't go anywhere → path = 1 - Start at `(0,1)` = 9 → same - Start at `(0,2)` = 4 → go to `(1,2)` = 8 → then to `(1,1)` = 6? No → 8 > 6 → invalid - But wait: `4 → 8` → `8 → ?` → nothing bigger → path = 2 - Start at `(1,0)` = 6 → go to `(1,1)` = 6 → not greater → skip - Start at `(1,1)` = 6 → no outgoing - Start at `(1,2)` = 8 → go to `(0,2)` = 4 → smaller → invalid - Start at `(2,0)` = 2 → go to `(1,0)` = 6 → good → then to `(0,0)` = 9 → good → then stop - Path: `2 → 6 → 9` → length = 3 - Also: `2 → 6 → 8` → length = 3 - So max = 3

Output: `3`

Complexity:

- **Time:** O(m×n) — each cell computed once due to memoization
- **Space:** O(m×n) — memoization table + recursion stack (depth m×n)

Problem 2: Minimum Height Trees

Summary:

Given an undirected tree (graph with n nodes and n−1 edges), find all **root nodes** such that the tree's height is minimized.

> A "height" of a tree is the longest path from root to any leaf.

Pattern(s):

- **BFS (Topological Sort-like)** — trimming leaves iteratively
- **Graph DP / Greedy Pruning**
- **Tree Center Finding**

Solution with Inline Comments:

```
from collections import deque, defaultdict

def findMinHeightTrees(n, edges):
    # Handle base case
    if n == 1:
        return [0]

    # Build adjacency list and degree array
    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

    # Queue for leaf nodes (degree = 1)
    queue = deque()
    for i in range(n):
        if degree[i] == 1:
            queue.append(i)

    remaining_nodes = n
    # Trim leaves layer by layer until 1 or 2 nodes remain
    while remaining_nodes > 2:
        # Remove current level of leaves
        leaves_count = len(queue)
        remaining_nodes -= leaves_count

        for _ in range(leaves_count):
            leaf = queue.popleft()

            # Remove leaf from its neighbor
            for neighbor in graph[leaf]:
                degree[neighbor] -= 1
                graph[neighbor].remove(leaf)

                # If neighbor becomes leaf, add it
                if degree[neighbor] == 1:
```

```
                queue.append(neighbor)

    # Remaining nodes are centers (min height roots)
    return list(queue)
```

Official Example Walkthrough:

Input: `n = 6`, `edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]`

Graph:

```
0   1   2
 \ | /
   3
  / \
 4   5
```

- Leaves: 0,1,2,5 → degree 1
- First round: remove 0,1,2,5 → update neighbors
  - Node 3 loses 3 connections → degree becomes 1
  - Node 4 loses one connection (5) → still degree 1
- Now leaves: 3,4 → both have degree 1
- Remaining nodes: 2 → stop
- Return `[3,4]`

But wait: `remaining_nodes > 2` → continue?

Let's simulate: - Initial: degree = [1,1,1,4,1,1] - Queue: [0,1,2,5] - Remove them → reduce degree of 3 and 4 → degree[3]=1, degree[4]=0? Wait:

Wait: edge `5-4` → so removing 5 → reduces degree of 4 from 2 to 1
Edge `3-0`, `3-1`, `3-2` → remove 0,1,2 → degree[3] goes from 4 → 1

So after first round: - degree: [0,0,0,1,1,0] - Queue: [3,4]

Now `remaining_nodes = 2` → break loop

Return `[3,4]`

Output: `[3,4]`

This makes sense — center nodes minimize height.

Complexity:

- **Time:** O(n) — each edge processed once
- **Space:** O(n) — graph, degree, queue

Problem 3: Clone Graph

Summary:

Given a reference to a node in a connected undirected graph, return a deep copy (clone) of the graph. Each node has a `val` and a list of its neighbors.

> This is a classic **graph traversal + hashing** problem using **DFS or BFS** with a **visited map**.

Pattern(s):

- **Graph Traversal (DFS/BFS)**
- **Hash Map for Cloning (Node → Clone Mapping)**
- **Deep Copy**

Why It Fits Your List:

- It's a **core graph problem** like others you've done.
- Requires **handling adjacency lists** and **avoiding cycles** via visited tracking.
- Perfect fit for **DFS + memoization** or **BFS + queue**.

Step-by-Step Thinking Process (The Recipe):

1. Use a **hash map** (`old_to_new`) to store mapping from original node to its clone.
2. Start from the given node.
3. If already cloned → return the clone.
4. Otherwise:
   - Create a new node with same `val`.
   - Add to map.
   - Recursively clone all neighbors.
5. Return the cloned node.

   Key insight: **Don't re-create nodes you've already cloned** → avoid infinite loops.

Solution Template (DFS + Memoization)

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

def cloneGraph(node):
    # Hash map: original node -> cloned node
    old_to_new = {}

    def dfs(n):
        # Base case: if already cloned, return it
        if n in old_to_new:
            return old_to_new[n]

        # Create new node
        clone = Node(n.val)
        old_to_new[n] = clone  # mark as cloned

        # Clone all neighbors
        for neighbor in n.neighbors:
            clone.neighbors.append(dfs(neighbor))
```

```
        return clone

    return dfs(node) if node else None
```

Time: O(V + E) — each node and edge processed once
Space: O(V) — hash map + recursion stack

Official Example Walkthrough:

Input:
`node = 1`, neighbors: `[2,4]`
`node 2`: neighbors: `[1,3]`
`node 3`: neighbors: `[2,4]`
`node 4`: neighbors: `[1,3]`

This forms a cycle: `1-2-3-4-1`

Steps: 1. Start at `node 1` 2. Create clone of `1` → `new_1` 3. Clone `2` → `new_2`, then clone `3` → `new_3`, then clone `4` → `new_4` 4. All connections are preserved in new graph 5. Return `new_1`

Output: Deep copy of the graph

LeetCode Interview Cheat Sheet: 10 Core Patterns + Templates

Designed for rapid review before interviews
Covers **Graphs, DP, BFS/DFS, Topo Sort, Union-Find**, and more
Each pattern includes: - How to recognize - Step-by-step thinking recipe - Common pitfalls - Code template (Python) - Example walkthrough (from real problems)

---

1. DFS / BFS on Grid

Recognize:

- 2D grid (`matrix`, `board`)
- Flood fill, island counting, shortest path in unweighted grid

Thinking Process:

1. Check bounds & visited state
2. Base case: invalid or visited → return
3. Mark current cell as visited
4. Recursively visit neighbors
5. Return result (count, path, etc.)

Pitfalls:

- Stack overflow (use iterative BFS for large grids)
- Forgetting to mark visited → infinite loop
- Wrong movement directions (4 vs 8)

Template (DFS):

```
def dfs(grid, r, c, visited):
    if (r < 0 or r >= len(grid) or
        c < 0 or c >= len(grid[0]) or
        grid[r][c] == '0' or (r,c) in visited):
        return

    visited.add((r,c))
    # Do work here
    for dr, dc in [(0,1),(0,-1),(1,0),(-1,0)]:
        dfs(grid, r+dr, c+dc, visited)
```

Example: Number of Islands

→ Count connected `'1'` groups using DFS.

---

2. Multi-source BFS

Recognize:

- Shortest distance to nearest target (e.g., `0`s)

- Multiple starting points (e.g., all rotten oranges, all sources)
- Unweighted graph/grid

Thinking Process:

1. Add all sources to queue with distance 0
2. Process level by level (BFS)
3. Update neighbor distances only if better
4. Stop when queue empty

Pitfalls:

- Not initializing distances properly
- Revisiting nodes without checking improvement

Template:

```
queue = deque()
dist = [[float('inf')] * cols for _ in range(rows)]
for each source:
    queue.append((r, c))
    dist[r][c] = 0

while queue:
    r, c = queue.popleft()
    for dr, dc in directions:
        nr, nc = r+dr, c+dc
        if valid and dist[nr][nc] > dist[r][c] + 1:
            dist[nr][nc] = dist[r][c] + 1
            queue.append((nr, nc))
```

Example: 01 Matrix

→ Find distance to nearest `0`.

---

3. Topological Sort (Kahn's Algorithm)

Recognize:

- Tasks with dependencies ("A must come before B")
- Detect cycle or find valid order
- Graph is directed

Thinking Process:

1. Build graph + indegree array
2. Queue all nodes with indegree 0
3. Remove node → reduce indegree of neighbors
4. If all nodes processed → no cycle → valid order

Pitfalls:

- Not handling self-loops
- Returning invalid order when cycle exists
- Forgetting to check `len(result) == n`

Template:

```
graph = [[] for _ in range(n)]
indegree = [0] * n
for u, v in edges:
    graph[u].append(v)
    indegree[v] += 1

queue = deque([i for i in range(n) if indegree[i] == 0])
result = []

while queue:
    u = queue.popleft()
    result.append(u)
    for v in graph[u]:
        indegree[v] -= 1
        if indegree[v] == 0:
            queue.append(v)
```

```python
return result if len(result) == n else []
```

Examples:

- Course Schedule
- Alien Dictionary

---

4. Union-Find (DSU)

Recognize:

- Connected components
- Cycle detection in undirected graphs
- Merge groups (e.g., accounts, islands)

Thinking Process:

1. Initialize parent and rank arrays
2. Use `find(x)` with path compression
3. Use `union(x,y)` by rank
4. Use `find` to check connectivity

Pitfalls:

- Not using path compression/rank → slow
- Incorrect union logic (e.g., always attach x to y)

Template:

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
```

```python
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        return True
```

Examples:

- Accounts Merge
- Graph Valid Tree

---

5. Shortest Path (BFS/Dijkstra)

Recognize:

- Minimum steps/distance in weighted/unweighted graph
- Constraints: max stops, limited moves

Thinking Process:

- **Unweighted**: BFS (queue)
- **Weighted**: Dijkstra (min-heap)
- Track state: `(node, steps, cost)`
- Prune if worse than best known

Pitfalls:

- Using BFS for weighted graphs → wrong answer
- Not tracking state (e.g., stops, cost)

Template (Dijkstra):

```
heapq.heappush(heap, (cost, node))
while heap:
    cost, node = heapq.heappop(heap)
    if cost > min_cost[node]: continue
    for neighbor, w in graph[node]:
        new_cost = cost + w
        if new_cost < min_cost[neighbor]:
            min_cost[neighbor] = new_cost
            heapq.heappush(heap, (new_cost, neighbor))
```

Examples:

- Word Ladder
- Cheapest Flights Within K Stops

---

6. Backtracking + DFS

Recognize:

- Search for combinations/subsets/permutations
- "Try all paths" — e.g., word search, Sudoku
- One cell used once per path

Thinking Process:

1. Try placing item at current position
2. Recurse
3. Backtrack: restore state
4. Avoid revisiting same path

Pitfalls:

- Not marking/unmarking visited → incorrect results
- Exponential time → optimize with pruning

Template:

```
def backtrack(path, choices):
    if goal_reached:
        result.append(path[:])
        return

    for choice in choices:
        path.append(choice)
        backtrack(path, remaining_choices)
        path.pop()  # backtracking
```

Example: Word Search II

→ Use Trie + DFS + backtracking

---

7. Dynamic Programming on Graphs

Recognize:

- Longest increasing path
- Optimal path based on values
- Recursive dependency on neighbors

Thinking Process:

1. Define `dp[r][c]` = longest path starting at `(r,c)`
2. Base: 1 (just current cell)
3. Recursion: `max(1 + dp[neighbor])` if condition holds
4. Memoize to avoid recomputation

Pitfalls:

- No memoization → exponential time
- Wrong direction (increasing vs decreasing)

Template:

```
memo = {}
def dfs(r, c):
    if (r,c) in memo: return memo[(r,c)]
    max_len = 1
    for dr,dc in directions:
        nr, nc = r+dr, c+dc
        if valid and matrix[nr][nc] > matrix[r][c]:
            max_len = max(max_len, 1 + dfs(nr, nc))
    memo[(r,c)] = max_len
    return max_len
```

Example: Longest Increasing Path in Matrix

→ DFS + memoization

---

8. Tree Center / Minimum Height Trees

Recognize:

- Find root(s) that minimize tree height
- Only works on trees (acyclic undirected graphs)

Thinking Process:

1. Trim leaves layer by layer
2. Until 1–2 nodes remain
3. These are the centers

Pitfalls:

- Not handling base case (n=1)
- Misunderstanding "height" definition

Template:

```
queue = deque([i for i in range(n) if degree[i] == 1])
while len(nodes) > 2:
    size = len(queue)
    for _ in range(size):
        leaf = queue.popleft()
        for neighbor in graph[leaf]:
            degree[neighbor] -= 1
            if degree[neighbor] == 1:
                queue.append(neighbor)
return list(queue)
```

Example: Minimum Height Trees

→ Iterative leaf trimming

---

9. Trie + DFS (Word Search II)

Recognize:

- Multiple words to search in grid
- Prefix-based pruning needed

Thinking Process:

1. Build Trie from all words
2. DFS from each cell
3. Prune if prefix not in Trie
4. Collect full words when found

Pitfalls:

- Not storing full word at leaf
- Not removing duplicates (use set)

Template:

```
class TrieNode:
    def __init__(self): self.children = {}; self.word = None

def insert(root, word):
    node = root
    for c in word:
        if c not in node.children:
            node.children[c] = TrieNode()
        node = node.children[c]
    node.word = word

def dfs(board, i, j, node, result):
    char = board[i][j]
    if char not in node.children: return
    node = node.children[char]
    if node.word:
        result.add(node.word)
    board[i][j] = '#'
    for di, dj in directions:
        ni, nj = i+di, j+dj
        if 0<=ni<m and 0<=nj<n and board[ni][nj]!='#':
            dfs(board, ni, nj, node, result)
    board[i][j] = char
```

---

10. Symmetry Optimization (e.g., Knight Moves)

Recognize:

- Infinite grid
- Symmetric solution space
- Can reduce search to one quadrant

Thinking Process:

1. Use absolute coordinates
2. Mirror negative indices

3. Limit search space (e.g., `x+2`, `y+2`)
4. BFS with symmetry

Template:

```
x, y = abs(x), abs(y)
queue = deque([(0,0,0)])
visited = {(0,0)}
while queue:
    r, c, steps = queue.popleft()
    if r == x and c == y: return steps
    for dr,dc in knight_moves:
        nr, nc = r+dr, c+dc
        nr, nc = abs(nr), abs(nc)
        if (nr,nc) not in visited and nr <= x+2 and nc <= y+2:
            visited.add((nr,nc))
            queue.append((nr,nc,steps+1))
```

Example: Minimum Knight Moves

→ Use symmetry to reduce state space

Final Tips for Interviews

Pattern	When to Use
DFS/BFS Grid	Island count, flood fill, shortest path
Multi-source BFS	Distance to nearest target
Topo Sort	Task scheduling, dependency resolution
Union-Find	Connected components, cycle detection
Dijkstra/BFS	Shortest path with constraints
Backtracking	Word search, permutation generation
DP on Graphs	Longest increasing path, optimal sequences
Tree Center	Minimize tree height
Trie + DFS	Multiple word search
Symmetry	Infinite grid problems