126. Word Ladder II

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

class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> Lis
t[List[str]]:
        wordSet = set(wordList)
        if endWord not in wordSet:
            return []

        # Build graph using BFS for shortest paths
        graph = defaultdict(list)
        found = False
        queue = deque([beginWord])
        visited = set([beginWord])

        while queue and not found:
            local_visited = set()
            for _ in range(len(queue)):
                word = queue.popleft()
                for i in range(len(word)):
                    for c in 'abcdefghijklmnopqrstuvwxyz':
                        if word[i] == c:
                            continue
                        newWord = word[:i] + c + word[i+1:]
                        if newWord in wordSet:
                            if newWord not in visited:
                                if newWord == endWord:
                                    found = True
                                if newWord not in local_visited:
                                    queue.append(newWord)
                                    local_visited.add(newWord)
                                graph[word].append(newWord)
            visited.update(local_visited)

        # Backtrack all paths using DFS
        res = []
        path = [beginWord]

        def dfs(current):
            if current == endWord:
                res.append(path[:])
                return
            for neighbor in graph[current]:
                path.append(neighbor)
                dfs(neighbor)
                path.pop()
        dfs(beginWord)
        return res
```

127. Word Ladder

```
from collections import deque
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> in
t:
        que = deque([[beginWord,1]])
        hashmap = set(wordList)
        if endWord not in hashmap:
            return 0
        while que :
            cur_word,step = que.popleft()
            if cur_word == endWord:return step
            for i in range(len(cur_word)):
                for char in range(97,123):
                    new_word = cur_word[:i] + chr(char)+cur_word[i+1:]
                    if new_word in hashmap:
                        hashmap.remove(new_word)
                        que.append([new_word,step+1])
        return 0
```

130. Surrounded Regions

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """

        if not board:
            return

        m, n = len(board), len(board[0])
        dirs = [(-1,0), (1,0), (0,-1), (0,1)]

        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or board[i][j] != 'O':
                return
            board[i][j] = 'E'  # Mark as escaped
            for dx, dy in dirs:
                dfs(i + dx, j + dy)

        # Step 1: Mark border-connected 'O's
        for i in range(m):
            if board[i][0] == 'O':
                dfs(i, 0)
            if board[i][n-1] == 'O':
                dfs(i, n-1)

        for j in range(n):
            if board[0][j] == 'O':
                dfs(0, j)
            if board[m-1][j] == 'O':
                dfs(m-1, j)

        # Step 2: Flip and revert
        for i in range(m):
            for j in range(n):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                elif board[i][j] == 'E':
                    board[i][j] = 'O'
```

207. Course Schedule

```
from collections import defaultdict
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        adj = defaultdict(list)
        for u,v in prerequisites:
            adj[v].append(u)
        #print(adj)
        path_vis = [0]*numCourses
        visited = set()
        def dfs(node):
            path_vis[node] = 1
            visited.add(node)
            for neigh in adj[node]:
                if neigh not in visited:
                    if dfs(neigh):
                        return True
                elif path_vis[neigh] == 1:
                    return True
            path_vis[node] = 0
            return False

        for i in range(numCourses):
            if dfs(i):
                return not True

        return True
```

210. Course Schedule II

```
from collections import defaultdict,deque
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[in
t]:
        vis_set = set()
        adj = defaultdict(list)
        vis = 0
        que = deque([])
        indegree = [0]*numCourses
        for x,y in prerequisites:
            adj[y].append(x)
            indegree[x] +=1

        for course in range(numCourses):
            if indegree[course] == 0:
                que.append(course)
        ans = []
        while que:
            node = que.popleft()
            vis_set.add(node)
            vis+=1
            ans.append(node)
            for neigh in adj[node]:
                indegree[neigh]-=1
                if indegree[neigh] == 0:
                    if neigh not in vis_set:
                        que.append(neigh)
        return [] if vis != numCourses else ans
```

542. 01 Matrix

```
from collections import deque
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m = len(mat)
        n = len(mat[0])
        que = deque()
        dirs = [(1,0),(0,1),(-1,0),(0,-1)]
        for i in range(m):
            for j in range(n):
                if mat[i][j] == 0:
                    que.append([i,j])
                else:
                    mat[i][j] = -1
        while que:
            x,y = que.popleft()
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0<=nx<m and 0<=ny<n and mat[nx][ny] == -1 :
                    mat[nx][ny]= mat[x][y]+1
                    que.append((nx,ny))
        return mat
```

547. Number of Provinces

```
class Solution:
    def findCircleNum(self, nums: List[List[int]]) -> int:
        n=len(nums)
        adj_list = [[] for _ in range(n)]
        visited = [False] * n

        for i in range(n):
            for j in range(len(nums[0])):
                if i!=j and nums[i][j]==1:
                    adj_list[i].append(j)

        def dfs(node: int):
            visited[node]=True
            for neighbours in adj_list[node]:
                if not visited[neighbours]:
                    dfs(neighbours)
        ans=0
        for i in range(n):
            if not visited[i]:
                dfs(i)
                ans+=1
        return  ans
```

721. Accounts Merge

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

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

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

    def union(self, x, y):
        self.parent[self.find(x)] = self.find(y)

class Solution:
    def accountsMerge(self, accounts: List[List[str]]) -> List[List[str]]:
        uf = UnionFind()
        email_to_name = {}

        # Step 1: Union emails within the same account
        for account in accounts:
            name = account[0]
            first_email = account[1]
            for email in account[1:]:
                uf.union(first_email, email)
                email_to_name[email] = name

        # 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: Build the result
        result = []
        for root, emails in groups.items():
            name = email_to_name[root]
            result.append([name] + sorted(emails))

        return result
```

733. Flood Fill

```
from collections import deque
class Solution:
    def floodFill(self, image: List[List[int]], sr: int, sc: int, color: int) -> Li
st[List[int]]:
        que = deque([[sr,sc]])
        dirs = [(-1,0),(0,-1),(1,0),(0,1)]
        m = len(image)
        n = len(image[0])
        col = image[sr][sc]
        if image[sr][sc] == color:
            return image
        while que:
            x,y = que.popleft()
            image[x][y] = color
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0 <=nx <m and 0 <=ny < n and col == image[nx][ny]:
                    que.append([nx,ny])
        return image
```

743. Network Delay Time

```
from collections import defaultdict
import heapq
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        adj = defaultdict(list)
        dist = [float("inf")]*(n+1)
        dist[k] = 0
        for x,y,z in times:
            adj[x].append([y,z])

        heap = [[0,k]]
        while heap:
            d,node = heapq.heappop(heap)
            for neigh,weight in adj[node]:
                if  weight+dist[node] < dist[neigh]:
                    dist[neigh] = weight+dist[node]
                    heapq.heappush(heap,[dist[neigh],neigh])
        maxx = -float("inf")
        for i in range(1,n+1):
            if dist[i] == float("inf"):
                return -1
            maxx = max(maxx,dist[i])
        return maxx
```

778. Swim in Rising Water

```
class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        time = 0
        dirs = [(0,1),(1,0),(-1,0),(0,-1)]
        zero = grid[0][0]
        heap = [(zero,0,0)]
        time = 0
        n = len(grid)
        while heap:
            weight,x,y = heapq.heappop(heap)
            grid[x][y] = "#"
            if time < weight:
                time = weight
            if x == n-1 and y == n-1:
                return time
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0<=nx<n and 0<=ny<n and grid[nx][ny] != "#":
                    weight = grid[nx][ny]
                    heapq.heappush(heap,(weight,nx,ny))
        return -1
```

785. Is Graph Bipartite?

```
from collections import deque
class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        n = len(graph)
        color = [-1]*n
        for start in range(n):
            if color[start] == -1:
                que = deque([start])
                color[start] = 0
                while que:
                    node = que.popleft()
                    for neigh in graph[node]:
                        if color[neigh] == -1:
                            color[neigh] = 1-color[node]
                            que.append(neigh)
                        elif color[neigh] == color[node]:
                            return False
        return True
```

787. Cheapest Flights Within K Stops

```
import heapq
from collections import defaultdict
from typing import List

class Solution:
    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: in
t, k: int) -> int:
        adj = defaultdict(list)
        for u, v, w in flights:
            adj[u].append((v, w))

        # Min-heap: (total cost so far, current city, stops used so far)
        heap = [(0, src, 0)]
        # best[node] = (min_cost, min_stops)
        best = dict()

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

            if node == dst:
                return cost

            if stops > k:
                continue

            # Avoid exploring worse paths
            if (node in best and best[node] <= stops):
                continue

            best[node] = stops

            for nei, price in adj[node]:
                heapq.heappush(heap, (cost + price, nei, stops + 1))

        return -1
```

802. Find Eventual Safe States

```
from collections import defaultdict,deque
class Solution:
    def eventualSafeNodes(self, graph: List[List[int]]) -> List[int]:
        adj = defaultdict(list)
        indegree = [0]*len(graph)
        for j in range(len(graph)):
            for i in range(len(graph[j])):
                adj[graph[j][i]].append(j)
                indegree[j] +=1
        que = deque()
        for i in range(len(indegree)):
            if indegree[i] == 0:
                que.append(i)

        ans = []
        while que:
            node = que.popleft()
            ans.append(node)
            for neigh in adj[node]:
                indegree[neigh]-=1
                if indegree[neigh] == 0:
                    que.append(neigh)
        return sorted(ans)
```

827. Making A Large Island

```
class DSU:
    def __init__(self, size):
        self.parent = list(range(size))
        self.size = [1] * size

    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):
        x_root = self.find(x)
        y_root = self.find(y)
        if x_root == y_root:
            return
        if self.size[x_root] < self.size[y_root]:
            x_root, y_root = y_root, x_root
        self.parent[y_root] = x_root
        self.size[x_root] += self.size[y_root]

class Solution:
    def largestIsland(self, grid: List[List[int]]) -> int:
        n = len(grid)
        dsu = DSU(n * n)

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

        # Step 1: Union all connected 1s and record their sizes
        for i in range(n):
            for j in range(n):
                if grid[i][j] == 1:
                    for di, dj in directions:
                        ni, nj = i + di, j + dj
                        if 0 <= ni < n and 0 <= nj < n and grid[ni][nj] == 1:
                            dsu.union(i * n + j, ni * n + nj)

        # Step 2: Find the maximum island size after converting each 0 to 1
        max_size = 0
        has_zero = False

        for i in range(n):
            for j in range(n):
                if grid[i][j] == 0:
                    has_zero = True
                    neighbors = set()
                    for di, dj in directions:
```

```
                    ni, nj = i + di, j + dj
                    if 0 <= ni < n and 0 <= nj < n and grid[ni][nj] == 1:
                        root = dsu.find(ni * n + nj)
                        neighbors.add(root)
                current_size = 1  # the current 0 turned to 1
                for root in neighbors:
                    current_size += dsu.size[root]
                max_size = max(max_size, current_size)

        # If there are no zeros, the entire grid is the largest island
        if not has_zero:
            return n * n

        return max_size
```

947. Most Stones Removed with Same Row or Column

```
from typing import List

class UnionFind:
    def __init__(self, size):
        self.par = list(range(size))

    def find(self, n):
        while n != self.par[n]:
            self.par[n] = self.par[self.par[n]]
            n = self.par[n]
        return n

    def union(self, a, b):
        root_a = self.find(a)
        root_b = self.find(b)
        if root_a == root_b:
            return False
        self.par[root_a] = root_b
        return True

class Solution:
    def removeStones(self, stones: List[List[int]]) -> int:
        max_coord = 10001  # problem constraints: 0 <= x, y <= 10000
        uf = UnionFind(2 * max_coord)
        seen = set()

        for x, y in stones:
            uf.union(x, y + max_coord)  # offset y to avoid collision with x
            seen.add(x)
            seen.add(y + max_coord)

        # count number of unique roots
        roots = set()
        for node in seen:
            roots.add(uf.find(node))

        return len(stones) - len(roots)
```

994. Rotting Oranges

```
from collections import deque
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        que = deque()
        m = len(grid)
        n = len(grid[0])
        fresh = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 2:
                    que.append([i,j])
                elif grid[i][j] == 1:
                    fresh+=1
        if fresh == 0:
            return 0
        time = 0
        dirs = [(-1,0),(0,-1),(1,0),(0,1)]
        while que:
            for _ in range(len(que)):
                x,y = que.popleft()
                for dx,dy in dirs:
                    nx,ny = x+dx,y+dy
                    if  0<=nx<m and 0<=ny<n and grid[nx][ny] == 1:
                        grid[nx][ny] = 2
                        fresh-=1
                        que.append([nx,ny])
            if que:
                time +=1
        return time if fresh == 0 else -1
```

1020. Number of Enclaves

```
class Solution:
    def numEnclaves(self, grid: List[List[int]]) -> int:
        m = len(grid)
        n = len(grid[0])
        dirs = [(0,1),(1,0),(-1,0),(0,-1)]
        def dfs(i,j):
            grid[i][j] = "E"
            for nx,ny in dirs:
                nx,ny = i+nx,j+ny
                if 0<=nx<m and 0<=ny<n and grid[nx][ny] == 1:
                    dfs(nx,ny)



        for i in range(m):
            if grid[i][0] == 1:
                dfs(i,0)
            if grid[i][n-1] == 1:
                dfs(i,n-1)

        for i in range(n):
            if grid[0][i] == 1:
                dfs(0,i)
            if grid[m-1][i] == 1:
                dfs(m-1,i)

        c  = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    c+=1
        return c
```

1091. Shortest Path in Binary Matrix

```
import heapq
from collections import deque
class Solution:
    def shortestPathBinaryMatrix(self, grid: List[List[int]]) -> int:
        pq = deque([[0,0]])
        n = len(grid)
        if grid[0][0] == 1 or grid[n-1][n-1] == 1:return -1
        dist = [[float("inf") for _ in range(n)] for _ in range(n)]
        dist[0][0] = 1
        dirs = [(0,1),(1,0),(-1,0),(0,-1),(1,1),(-1,-1),(1,-1),(-1,1)]
        while pq:
            x,y = pq.popleft()
            d = dist[x][y]
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0<=nx < n and 0<=ny<n and grid[nx][ny] == 0 and d+1 <dist[nx][n
y]:
                    dist[nx][ny] = d+1
                    pq.append((nx,ny))
        return dist[n-1][n-1] if dist[n-1][n-1] != float("inf") else -1
```

1192. Critical Connections in a Network

```
from typing import List

class Solution:
    def criticalConnections(self, n: int, connections: List[List[int]]) -> List[Lis
t[int]]:
        graph = [[] for _ in range(n)]
        for u, v in connections:
            graph[u].append(v)
            graph[v].append(u)

        disc = [-1] * n      # Discovery times
        low = [-1] * n       # Lowest discovery times
        time = [0]           # Global timer
        result = []

        def dfs(u, parent):
            disc[u] = low[u] = time[0]
            time[0] += 1

            for v in graph[u]:
                if v == parent:
                    continue
                if disc[v] == -1:
                    dfs(v, u)
                    low[u] = min(low[u], low[v])
                    if low[v] > disc[u]:
                        result.append([u, v])
                else:
                    low[u] = min(low[u], disc[v])

        dfs(0, -1)
        return result
```

1319. Number of Operations to Make Network Connected

```
class UnionFind:
    def __init__(self,V):
        self.par = list(range(V))
        self.rank = [0]*V
    def get_parent(self):
        return self.par
    def find(self,n):
        while n != self.par[n]:
            self.par[n] = self.par[self.par[n]]
            n = self.par[n]
        return n

    def union(self,a,b):
        root_a = self.find(a)
        root_b = self.find(b)
        if root_a == root_b:
            return False
        self.par[root_a] = root_b
        return True
class Solution:
    def makeConnected(self, n: int, connections: List[List[int]]) -> int:
        if len(connections) < n-1:
            return -1
        cables = len(connections)
        uf = UnionFind(n)
        same_par = 0
        for u,v in connections:
            if not uf.union(u,v):
                same_par+=1
        par = uf.get_parent()
        c = 0
        for i in range(n):
            if par[i] == i:
                c+=1
        c-=1
        if same_par >= c:
            return c

        return same_par if same_par else -1
```

1334. Find the City With the Smallest Number of Neighbors at a Threshold Distance

```
class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], distanceThreshold: int) -
> int:
        INF = float('inf')
        # Step 1: Initialize distance matrix
        dist = [[INF] * n for _ in range(n)]
        for i in range(n):
            dist[i][i] = 0

        # Step 2: Set direct edge distances
        for u, v, w in edges:
            dist[u][v] = w
            dist[v][u] = w

        # Step 3: Floyd-Warshall to compute all-pairs shortest path
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    if dist[i][k] + dist[k][j] < dist[i][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]

        # Step 4: Find city with minimum reachable cities
        minCount = n + 1
        resultCity = -1

        for i in range(n):
            count = 0
            for j in range(n):
                if i != j and dist[i][j] <= distanceThreshold:
                    count += 1
            # Break ties by choosing the larger index
            if count <= minCount:
                minCount = count
                resultCity = i

        return resultCity
```

1631. Path With Minimum Effort

```
from collections import deque
import heapq
class Solution:
    def minimumEffortPath(self, heights: List[List[int]]) -> int:
        row = len(heights)
        col = len(heights[0])
        pq = [(0,0,0)]
        dirs = [(0,1),(1,0),(0,-1),(-1,0)]
        max_diff = 0
        vis = [ [False for _ in range(col)] for _ in range(row) ]
        while pq:
            e,x,y = heapq.heappop(pq)
            max_diff = max(e,max_diff)
            if row-1 == x and y == col-1 :
                return max_diff
            if vis[x][y] == True:
                continue
            vis[x][y] = True
            for nx,ny in dirs:
                nx,ny = x+nx,y+ny
                if 0<=nx<row and 0<=ny<col:
                    effort = abs(heights[nx][ny] - heights[x][y])
                    heapq.heappush(pq,(effort,nx,ny))
        return max_diff
```

1976. Number of Ways to Arrive at Destination

```
from heapq import heappush,heappop
from collections import deque
class Solution:
    def countPaths(self, n: int, roads: List[List[int]]) -> int:
        MOD = 10**9 + 7
        heap = [[0,0]]
        adj = {i:[] for i in range(n)}
        dist = [float("inf") ]*n
        dist[0] = 0
        for x,y,z in roads:
            adj[x].append([y,z])
            adj[y].append([x,z])
        ways = [0]*n
        ways[0] = 1
        while heap:
            d,node = heappop(heap)
            if d > dist[node]:
                continue
            for neigh,weight in adj[node]:
                if weight+dist[node] < dist[neigh]:
                    dist[neigh]  = weight+dist[node]
                    heappush(heap,[dist[neigh],neigh])
                    ways[neigh] = ways[node]
                elif weight+dist[node] == dist[neigh]:
                    ways[neigh] = (ways[neigh]+ways[node])%MOD
        return ways[n-1]
```