Notebook

August 6, 2024

[ ]:
```
from typing import List
from collections import defaultdict
```

Simple Computation

[ ]:
```
# https://leetcode.com/problems/maximum-star-sum-of-a-graph/description/
# find maximum star sum of a graph with atmost k edges neighbors to it
# pick onlyy positive values from the graph
# find all possible stars and then find the maximum sum of the star

import collections



class Solution:
    # just find the center of the graph which has (n-1) edges
    def findCenter(self, e: List[List[int]]) -> int:
        return e[0][0] if e[0][0] == e[1][0] or e[0][0] == e[1][1] else e[0][1]

    def maxStarSum(self, vals: List[int], edges: List[List[int]], k: int) ->
 ↪int:
        g = collections.defaultdict(set)
        for u, v in edges:
            if vals[v] > 0 : g[u].add(v)
            if vals[u] > 0 : g[v].add(u)

        stars = []
        for idx , val in enumerate(vals):
            vv = [vals[j] for j in g[idx]]
            vv.sort(reverse=True)
            stars.append(val + sum(vv[:k]))

        return max(stars)
```

[ ]:
```
# https://leetcode.com/problems/maximal-network-rank/description/
# count the indegree of each pair of nodes and then find the maximum rank

class Solution:
```

```python
    def maximalNetworkRank(self, n: int, roads: List[List[int]]) -> int:
        g = defaultdict(set)
        for u, v in roads:
            g[u].add(v)
            g[v].add(u)

        ans = 0
        for i in range(n):
            for j in range(i+1, n):
                ans = max(ans, (len(g[i]) + len(g[j]) - (i in g[j])))
        return ans
```

DFS

[ ]:
```python
# https://leetcode.com/problems/reconstruct-itinerary/description/
# start from JFK and reconstruct the itinerary

class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        # Construct the graph
        self.flights = defaultdict(list)
        self.itinerary = []
        for ticket in tickets:
            self.flights[ticket[0]].append(ticket[1])

        # Ensure lexicographical order
        for key in self.flights:
            self.flights[key].sort(reverse=True)

        self.visit("JFK")
        return self.itinerary[::-1]

    def visit(self, airport):
        while self.flights[airport]:
            next_airport = self.flights[airport].pop()
            self.visit(next_airport)
        self.itinerary.append(airport)
```

[ ]:
```python
#### https://leetcode.com/problems/word-search/description/

def exist(self, board, word):
    if not board:
        return False
    for i in range(len(board)):
        for j in range(len(board[0])):
            if self.dfs(board, i, j, word):
                return True
    return False
```

```
# check whether can find word, start at (i,j) position
def dfs(self, board, i, j, word):
    if len(word) == 0: # all the characters are checked
        return True
    if i<0 or i>=len(board) or j<0 or j>=len(board[0]) or word[0]!=board[i][j]:
        return False
    tmp = board[i][j]  # first character is found, check the remaining part
    board[i][j] = "#"  # avoid visit agian
    # check whether can find "word" along one direction
    res = self.dfs(board, i+1, j, word[1:]) or self.dfs(board, i-1, j, word[1:
↪]) \
    or self.dfs(board, i, j+1, word[1:]) or self.dfs(board, i, j-1, word[1:])
    board[i][j] = tmp
    return res


#### https://leetcode.com/problems/word-search-ii/description/

class TrieNode:
    def __init__(self):
        self.children = defaultdict(TrieNode)
        self.word = None

    def addWord(self, word):
        cur = self
        for c in word:
            cur = cur.children[c]
        cur.word = word

class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        m, n = len(board), len(board[0])
        DIR = [0, 1, 0, -1, 0]
        trieNode = TrieNode()
        ans = []
        for word in words:
            trieNode.addWord(word)

        def dfs(r, c, cur):
            if r < 0 or r == m or c < 0 or c == n or board[r][c] not in cur.
↪children: return
            orgChar = board[r][c]
            cur = cur.children[orgChar]
            board[r][c] = '#'  # Mark as visited
            if cur.word != None:
                ans.append(cur.word)
```

```
            cur.word = None  # Avoid duplication!
        for i in range(4): dfs(r + DIR[i], c + DIR[i + 1], cur)
        board[r][c] = orgChar  # Restore to org state

    for r in range(m):
        for c in range(n):
            dfs(r, c, trieNode)
    return ans
```

[ ]:
```
# https://leetcode.com/problems/
↪reorder-routes-to-make-all-paths-lead-to-the-city-zero/description/
# reorder the routes to make all paths lead to city 0
# we make graph with edges in two directions from u to v and v to u and mark␣
↪the opposite direction as negative since we dont need to change it

# Time complexity: O(n) where n is the number of cities / nodes
# Space complexity: O(n)
class Solution:
    def minReorder(self, n: int, connections: List[List[int]]) -> int:
        def dfs(graph, visited, node):
            change = 0
            visited[node] = True
            for next_node in graph[node]:
                if not visited[abs(next_node)]:
                    change += dfs(graph, visited, abs(next_node)) + (1 if␣
↪next_node > 0 else 0)
            return change
        graph = [[] for _ in range(n)]
        for c in connections:
            graph[c[0]].append(c[1])
            graph[c[1]].append(-c[0])
        visited = [False] * n
        return dfs(graph, visited, 0)
```

0.0.1 BFS

[ ]:
```
# https://leetcode.com/problems/the-time-when-the-network-becomes-idle/
↪description/

class Solution:
    def networkBecomesIdle(self, edges: List[List[int]], patience: List[int])␣
↪-> int:
        g = defaultdict(list)
        for u, v in edges:
            g[u].append(v)
            g[v].append(u)
```

```
        dist = {}
        q = deque()
        q.append((0, 0))
        seen = set()
        while q:
            u, d = q.popleft()
            if u in seen: continue
            seen.add(u)
            dist[u] = d

            for v in g[u]:
                q.append((v, d+1))
        ans = 0

        for index in range(1, len(patience)):
            resendInterval = patience[index]
            shutOffTime = dist[index] * 2
            lastSecond = shutOffTime - 1
            print(resendInterval, shutOffTime)
            lastResentTime = (lastSecond // resendInterval) * resendInterval
            lastPacketTime = lastResentTime + shutOffTime
            ans = max(ans, lastPacketTime)
        return ans+1
```

[ ]:
```
# https://github.com/doocs/leetcode/blob/main/solution/0300-0399/0317.
↪Shortest%20Distance%20from%20All%20Buildings/README_EN.md
### maintain cnt and dist for each cell and then find the minimum distance cell
"""
You are given an m x n grid grid of values 0, 1, or 2, where:

each 0 marks an empty land that you can pass by freely,
each 1 marks a building that you cannot pass through, and
each 2 marks an obstacle that you cannot pass through.
You want to build a house on an empty land that reaches all buildings in the␣
↪shortest total travel distance. You can only move up, down, left, and right.

Return the shortest travel distance for such a house. If it is not possible to␣
↪build such a house according to the above rules, return -1.

The total travel distance is the sum of the distances between the houses of the␣
↪friends and the meeting point.
```

```
The distance is calculated using Manhattan Distance, where distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|
"""

from collections import deque

class Solution:
    def shortestDistance(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        q = deque()
        total = 0
        cnt = [[0] * n for _ in range(m)]
        dist = [[0] * n for _ in range(m)]
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    total += 1
                    q.append((i, j))
                    visited = [[False] * n for _ in range(m)]
                    visited[i][j] = True
                    level = 0
                    while q:
                        level += 1
                        for _ in range(len(q)):
                            x, y = q.popleft()
                            for a, b in [[0, 1], [1, 0], [0, -1], [-1, 0]]:
                                nx, ny = x + a, y + b
                                if 0 <= nx < m and 0 <= ny < n and not visited[nx][ny] and grid[nx][ny] == 0:
                                    visited[nx][ny] = True
                                    cnt[nx][ny] += 1
                                    dist[nx][ny] += level
                                    q.append((nx, ny))

        res = float('inf')
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 0 and cnt[i][j] == total:
                    res = min(res, dist[i][j])

        return res if res != float('inf') else -1

grid = [[1,0,2,0,1],[0,0,0,0,0],[0,0,1,0,0]]
s = Solution()
print(s.shortestDistance(grid)) # 7
```

```
[ ]: # https://leetcode.com/problems/01-matrix/description/

class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        DIR = [0, 1, 0, -1, 0]

        q = deque([])
        for r in range(m):
            for c in range(n):
                if mat[r][c] == 0:
                    q.append((r, c))
                else:
                    mat[r][c] = -1  # Marked as not processed yet!

        while q:
            r, c = q.popleft()
            for i in range(4):
                nr, nc = r + DIR[i], c + DIR[i + 1]
                if nr < 0 or nr == m or nc < 0 or nc == n or mat[nr][nc] != -1:
 ↪continue
                mat[nr][nc] = mat[r][c] + 1
                q.append((nr, nc))
        return mat
```

Kahn's Algorithm (Topological Sort)

```
[ ]:
```

```
[ ]: # https://leetcode.com/problems/build-a-matrix-with-conditions/description/
# fill the matrix with the given row and column conditions
# topological sort - find row and column order and then fill the matrix
"""
Input: k = 3, rowConditions = [[1,2],[3,2]], colConditions = [[2,1],[3,2]]
Output: [[3,0,0],[0,0,1],[0,2,0]]
"""

class Solution:
    def buildMatrix(self, k: int, rowConditions: List[List[int]], colConditions:
 ↪ List[List[int]]) -> List[List[int]]:
        def topo_sort(A):
            deg, graph, order = defaultdict(int), defaultdict(list), []
            for x, y in A:
                graph[x].append(y)
                deg[y] += 1
            q = deque([i for i in range(1, k+1) if deg[i] == 0])

            while q:
```

```
                x = q.popleft()
                order.append(x)
                for y in graph[x]:
                    deg[y] -= 1
                    if deg[y] == 0: q.append(y)

            return order

        left_right_order = topo_sort(colConditions)
        above_below_order = topo_sort(rowConditions)
        if len(left_right_order) < k  or len(above_below_order) < k : return []

        val_to_row = {x: i for i,x in enumerate(above_below_order)}
        val_to_col = {x: i for i,x in enumerate(left_right_order)}

        ans = [[0] * k for _ in range(k)]
        for num in range(1, k+1):
            r, c = val_to_row[num], val_to_col[num]
            ans[r][c] = num

        return ans
```

Dijkstra Algorithm (Priority Queue)

[ ]:
```
# https://leetcode.com/problems/minimum-cost-of-a-path-with-special-roads/
 ↪description/
# https://leetcode.com/problems/minimum-path-sum/description/
# https://leetcode.com/problems/
 ↪number-of-restricted-paths-from-first-to-last-node/description/
# https://leetcode.com/problems/find-root-of-n-ary-tree/description/
# https://leetcode.com/problems/maximal-network-rank/description/
# https://leetcode.com/problems/minimum-number-of-vertices-to-reach-all-nodes/
 ↪description/
# https://leetcode.com/discuss/interview-question/5366542/
 ↪all-leetcode-articles-on-coding-patterns-summarized-in-one-page
# https://leetcode.com/discuss/interview-question/5377854/
 ↪flipkart-sde2-machine-coding-round
```

[ ]:
```

```

[ ]:
```
# https://leetcode.com/problems/
 ↪find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/
 ↪description
# given undirected graph with n cities and distance threshold, find the city␣
 ↪with smallest number of neighbors at a threshold distance
```

```
# use dijkstra to find distance from each city to all other cities and then␣
↪find the number of cities within the threshold distance
# Time complexity: O(n^2 log n)

class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], distanceThreshold:␣
↪int) -> int:
        graph = defaultdict(list)
        for u, v, w in edges:
            graph[u].append((v, w))
            graph[v].append((u, w))

        rechable = [0] * n
        def dijkstra(node):
            dist = [float('inf')] * n
            dist[node] = 0
            minHeap = []
            heappush(minHeap, (0, node))
            while minHeap:
                curr_dist, curr_node = heappop(minHeap)
                for next_node, w in graph[curr_node]:
                    if w + curr_dist < dist[next_node]:
                        dist[next_node] = w + curr_dist
                        heappush(minHeap, (dist[next_node], next_node))
            return dist

        ans_node, ans_reachable = -1, n
        for i in range(n):
            dist = dijkstra(i)
            dlen = len([x for idx, x in enumerate(dist) if x <=␣
↪distanceThreshold and idx != i])
            if dlen <= ans_reachable:
                ans_reachable = dlen
                ans_node = i

        return ans_node
```

[ ]:
```
# https://leetcode.com/problems/
↪minimum-cost-to-make-at-least-one-valid-path-in-a-grid/description/

# Can solve using simple queue or priority queue
# when the grid value matches the direction of the path, then the cost is 0␣
↪else 1
# Time complexity is O(m*n) and space complexity is O(m*n)
```

```python
class Solution:
    def minCost(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        vis =[[False]*n for _ in range(m)]
        q = deque()
        q.append((0, 0, 0))
        dirs = [[0, 1], [0, -1], [1, 0], [-1, 0]]
        while q:
            x, y, cost = q.popleft()
            if vis[x][y]: continue
            vis[x][y] = True
            if x == m-1 and y == n-1: return cost
            for i in range(len(dirs)):
                dx, dy = dirs[i][0], dirs[i][1]
                nx, ny = x+dx, y+dy
                if nx < 0 or nx >= m or ny < 0 or ny >= n: continue
                if i+1 == grid[x][y]:
                    q.appendleft((nx, ny, cost))
                else:
                    q.append((nx, ny, cost+1))

        return -1
```

[ ]:
```python
# https://leetcode.com/problems/path-with-minimum-effort/description/
# start from 0,0 and reach m-1,n-1 with minimum effort
# A route's effort is the maximum absolute difference in heights between two␣
↪consecutive cells of the route.

"""
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2 /  Explanation: The route of [1,3,5,3,5] has a maximum absolute␣
↪difference of 2 in consecutive cells.
"""

# standard dijkstra algorithm to solve this problem
# Time complexity -  O(ElogV) = O(M*N log M*N)
# Space complexity - O(M*N)
class Solution:
    def minimumEffortPath(self, heights: List[List[int]]) -> int:
        m, n = len(heights), len(heights[0])
        dist = [[inf] * n for _ in range(m)]
        dist[0][0] = 0
        minHeap = [(0, 0, 0)] # (dist, row, col)
        dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]]

        while minHeap:
```

```
            d, x, y = heappop(minHeap)
            if d > dist[x][y]: continue  # this is an outdated version -> skip␣
 ↪it
            if x == m-1 and y == n-1: return d
            for dx, dy in dirs:
                nx, ny = x + dx, y + dy
                if 0 <= nx < m and 0 <= ny < n:
                    newDist = max(d, abs(heights[nx][ny]-heights[x][y]))
                    if dist[nx][ny] > newDist:
                        dist[nx][ny] = newDist
                        heappush(minHeap, (dist[nx][ny], nx, ny))

# Binary search + DFS solution
```

[ ]:
```
# https://leetcode.com/problems/swim-in-rising-water/description/

# we have to react end of the grid(m-1, n-1) with minimum time can swim to␣
 ↪equal or less than the value of the grid neighbors
# at time t, we can swim to the cell with value <= t
# can be solved used (binary search and dfs) / Priority queue

# Priority queue solution
# ans is max(grid[0][0], grid[m-1][n-1]) and then do standard priority queue␣
 ↪dijkstra algorithm

class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        pq = []
        vis = [[False]*n for _ in range(m)]
        vis[0][0] = True
        ans = max(grid[0][0], grid[m-1][n-1])
        heappush(pq, (ans, 0, 0))
        dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]]
        while pq:
            cost, x, y = heappop(pq)
            ans = max(ans, cost)
            if x == m-1 and y == n-1: return ans
            for dx, dy in dirs:
                nx, ny = dx + x, dy + y
                if nx == m-1 and ny == n-1: return ans
                if nx < 0 or nx >= m or ny < 0 or ny >= n or vis[nx][ny]:
                    continue
                heappush(pq, (grid[nx][ny], nx, ny))
                vis[nx][ny] = True

        return -1
```

```
# Binary search and (dfs or bfs) solution

class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        n = len(grid)
        l, h = grid[0][0], n*n-1
        dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]]

        def bfs(m):
            q = [(0, 0)]
            seen = set((0, 0))
            for x, y in q:
                if grid[x][y] <= m:
                    if x == y == n-1: return True
                    for dx, dy in dirs:
                        nx, ny = x+dx, y+dy
                        if 0 <= nx < n and 0 <= ny < n and (nx, ny) not in seen:
                            seen.add((nx, ny))
                            q.append((nx, ny))
            return False

        def dfs(vis, x, y, m):
            vis[x][y] = True
            for dx, dy in dirs:
                nx, ny = dx+x, dy+y
                if 0 <= nx < n and 0 <= ny < n and not vis[nx][ny] and␣
 ↪grid[nx][ny] <=m :
                    if nx == n-1 and ny == n-1: return True
                    if dfs(vis, nx, ny, m): return True
            return False

        def isValid(m):
            vis = [[False]*n for _ in range(n)]
            # return dfs(vis, 0, 0, m)
            return bfs(m)

        while l < h:
            m = l + (h - l) // 2
            if isValid(m):
                h = m
            else:
                l = m+1
        return l
```

[ ]:
```
# https://leetcode.com/problems/find-the-safest-path-in-a-grid/description/
```

```python
# given a grid with two types of cells, 0 and 1 --> 0 is empty cell and 1 is␣
↪obstacle (thief)
# first check if start (0,0) and end (m-1, n-1) is reachable
# do bfs from all thief cells and calculate the distance to all empty cells
# use dijkstra algorithm to find the minimum distance from start to end cell

class Solution:
    def maximumSafenessFactor(self, grid: List[List[int]]) -> int:
        n = len(grid)
        if grid[0][0] or grid[n-1][n-1]: return 0
        score = [[inf] * n for _ in range(n)]
        dirs = [[-1, 0], [0, -1], [0, 1], [1, 0]]
        def bfs():
            q = deque()
            for i in range(n):
                for j in range(n):
                    if grid[i][j]:
                        score[i][j] = 0
                        q.append((i, j))
            while q:
                x, y = q.popleft()
                s = score[x][y]
                for dx, dy in dirs:
                    nx, ny = x + dx, y + dy
                    if 0 <= nx < n and 0 <= ny < n and score[nx][ny] > s + 1:
                        score[nx][ny] = s + 1
                        q.append((nx, ny))

        bfs()
        vis = [[False] * n for _ in range(n)]
        pq = [(-score[0][0], 0, 0)]
        while pq:
            safe, x, y = heappop(pq)
            safe = -safe
            if x == n-1 and y == n-1: return safe
            vis[x][y] = True
            for dx, dy in dirs:
                nx, ny = dx + x, dy + y
                if 0 <= nx < n and 0 <= ny < n and not vis[nx][ny]:
                    s = min(score[nx][ny], safe)
                    heappush(pq, (-s, nx, ny))
                    vis[nx][ny] = True
        return -1
```

[ ]:
```python
# https://github.com/doocs/leetcode/blob/main/solution/1100-1199/1102.
↪Path%20With%20Maximum%20Minimum%20Value/README_EN.md
```

```python
import heapq

def maximumMinimumPath(grid):
    # Define the directions for moving in 4 cardinal directions
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    m, n = len(grid), len(grid[0])

    # Max-heap (we invert the values to use heapq as a max-heap)
    max_heap = [(-grid[0][0], 0, 0)]

    # To track the maximum minimum value path to each cell
    max_min_values = [[-float('inf')] * n for _ in range(m)]
    max_min_values[0][0] = grid[0][0]

    while max_heap:
        current_min_val, x, y = heapq.heappop(max_heap)
        current_min_val = -current_min_val

        # If we reach the bottom-right corner, return the value
        if x == m - 1 and y == n - 1:
            return current_min_val

        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            if 0 <= nx < m and 0 <= ny < n:
                new_min_val = min(current_min_val, grid[nx][ny])

                if new_min_val > max_min_values[nx][ny]:
                    max_min_values[nx][ny] = new_min_val
                    heapq.heappush(max_heap, (-new_min_val, nx, ny))

    # If there is no valid path (although per problem statement, there should
    ↪always be one)
    return -1

# Example usage:
grid = [[5, 4, 5], [1, 2, 6], [7, 4, 6]]
print(maximumMinimumPath(grid))  # Output: 4
grid = [[3,4,6,3,4],[0,2,1,1,7],[8,8,3,2,7],[3,2,4,9,8],[4,1,2,0,0],[4,6,5,4,3]]
print(maximumMinimumPath(grid)) # Output: 3
```

[ ]:
```python
# https://leetcode.com/problems/network-delay-time/description/
# will send a signal from node k to all other nodes in the network
# dijkstra algorithm can be used to solve this problem
```

```python
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        graph = defaultdict(list)
        dist = {}
        minHeap = [(0, k)]
        for u, v, w in times:
            graph[u].append((v, w))

        while minHeap:
            time, node = heappop(minHeap)
            if node not in dist:
                dist[node] = time
                for v, w in graph[node]:
                    heapq.heappush(minHeap, (time+w, v))

        return max(dist.values()) if len(dist) == n else -1
```

[ ]:
```python
# https://leetcode.com/problems/
 ↪minimum-weighted-subgraph-with-the-required-paths/description/

# we need to find the shortest path from src1 and src2 to dst
# To solve with we have intermediate node (IM) we try to go from src1 to IM and␣
 ↪then from IM to dst
# standard dijkstra algorithm with priority queue but need three times call␣
 ↪dijkstra(graph, src1), dijkstra(graph, src2), dijkstra(graph, dst)

from heapq import heappop, heappush


class Solution:
    def minimumWeight(self, n: int, edges: List[List[int]], src1: int, src2:␣
 ↪int, dest: int) -> int:
        def dijkstra(g, src):
            distances = defaultdict(lambda : inf)
            heap = [(0, src)]

            while heap:
                dist, node = heappop(heap)
                if node in distances: continue
                distances[node] = dist

                for neigh in g[node].keys():
                    if neigh in distances: continue
                    newdist = distances[node] + g[node][neigh]
                    heappush(heap, (newdist, neigh))
```

```
        return distances

    graph = defaultdict(dict)
    rev_graph = defaultdict(dict)
    for u, v, w in edges:
        graph[u][v] = w if v not in graph[u] else min(w, graph[u][v])
        rev_graph[v][u] = w if u not in rev_graph[v] else min(w,␣
↪rev_graph[v][u])

    src1_distances = dijkstra(graph, src1)
    src2_distances = dijkstra(graph, src2)
    dest_distances = dijkstra(rev_graph, dest)

    res = inf
    for node in range(n):
        local_res = src1_distances[node] + src2_distances[node] +␣
↪dest_distances[node]
        res = min(local_res, res)

    return res if res != inf else -1
```

Union Find

[ ]:
```
# https://github.com/doocs/leetcode/blob/main/solution/1100-1199/1102.
↪Path%20With%20Maximum%20Minimum%20Value/README_EN.md

# we need to find the maximum minimum value path from (0,0) to (m-1, n-1)
# we can solve it using union find / dijkstra algorithm
# we can sort the values and then start from the largest value and then find␣
↪the path to reach the destination
# we do union find until parent of 0 and parent of m*n-1 are same

from typing import List

class UnionFind:
    __slots__ = ("p", "size")

    def __init__(self, n):
        self.p = list(range(n))
        self.size = [1] * n

    def find(self, x: int) -> int:
        if self.p[x] != x:
            self.p[x] = self.find(self.p[x])
        return self.p[x]

    def union(self, a: int, b: int) -> bool:
```

```
        pa, pb = self.find(a), self.find(b)
        if pa == pb:
            return False
        if self.size[pa] > self.size[pb]:
            self.p[pb] = pa
            self.size[pa] += self.size[pb]
        else:
            self.p[pa] = pb
            self.size[pb] += self.size[pa]
        return True


class Solution:
    def maximumMinimumPath(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        uf = UnionFind(m * n)
        q = [(v, i, j) for i, row in enumerate(grid) for j, v in enumerate(row)]
        print(q)
        q.sort()
        ans = 0
        vis = set()
        dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]]
        while uf.find(0) != uf.find(m * n - 1):
            v, i, j = q.pop() # pop the largest value, pop gets the last element
            print(v, i, j)
            ans = v
            vis.add((i, j))
            for a, b in dirs:
                x, y = i + a, j + b
                if (x, y) in vis:
                    uf.union(x * n + y, i * n + j)
        return ans

grid = [[3,4,6,3,4],[0,2,1,1,7],[8,8,3,2,7],[3,2,4,9,8],[4,1,2,0,0],[4,6,5,4,3]]
s = Solution()
print(f'ans is {s.maximumMinimumPath(grid)}')
```

Minimum Spanning Tree

- Prim Algorithm - Priority Queue
- Kruskal Algorithm - Union Find

[ ]:
```
# https://leetcode.com/problems/min-cost-to-connect-all-points/description/
# we need to find the minimum cost to connect all points - minimum spanning tree
# given points and manhattan distance between them is the cost (abs(x1-x2) +
↪abs(y1-y2))
```

```python
# Prim's algorithm - standard minimum spanning tree algorithm (Priority queue)
# Time complexity - O(n^2 log n) , due to priority queue operations
# Space complexity - O(n)
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        def manhattanDistance(p1, p2):
            return abs(p1[0]-p2[0]) + abs(p1[1]-p2[1])

        n = len(points)
        visited = [False] * n
        dist = {}
        minHeap = [(0, 0)]
        mst_weight = 0
        while minHeap:
            w, u = heappop(minHeap)
            if visited[u] or dist.get(u, inf) < w:
                continue
            visited[u] = True
            mst_weight += w
            for v in range(n):
                if not visited[v]:
                    new_dist = manhattanDistance(points[u], points[v])
                    if new_dist < dist.get(v, inf):
                        dist[v] = inf
                        heappush(minHeap, (new_dist, v))
        return mst_weight


# Kruskal's algorithm - standard minimum spanning tree algorithm (Union find)
# Time complexity - O(n^2 log n) , due to priority queue operations / edges sort
# Space complexity - O(n)

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

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

    def union(self, u, v):
        pu, pv = self.find(u), self.find(v)
        if pu == pv: return False
        if self.rank[pu] > self.rank[pv]:
            pu, pv = pv, pu
```

```
        self.parent[pu] = pv
        if self.rank[pu] == self.rank[pv]:
            self.rank[pv] += 1
        return True

class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        def manhattanDistance(p1, p2):
            return abs(p1[0]-p2[0]) + abs(p1[1]-p2[1])
        n = len(points)
        uf = UnionFind(n)
        edges = []
        for i in range(n):
            for j in range(i+1, n):
                distance = manhattanDistance(points[i], points[j])
                heappush(edges, (distance, i, j))
                #edges.append((distance, i, j))

        #edges.sort()
        mst_weight, mst_edges = 0, 0
        while edges:
            # w, u, v = edges.pop(0)
            w, u, v = heappop(edges)
            if uf.union(u, v):
                mst_weight += w
                mst_edges += 1
                if mst_edges == n-1: break

        return mst_weight
```

[ ]:

```
# https://leetcode.com/problems/second-minimum-time-to-reach-destination/
 ↪description/
# since we need to find the second minimum time to reach the destination, we␣
 ↪need to maintain two distances
# since weight of the edge is time and it is same for all edges we can use BFS␣
 ↪instead of dijkstra algorithm
# since red signal happens when time % change == 0, we can update the new cost␣
 ↪based on the red signal

class Solution:
    def secondMinimum(self, n: int, edges: List[List[int]], time: int, change:␣
 ↪int) -> int:
        dist, dist2 = [float('inf')]*(n+1), [float('inf')]*(n+1)
        graph = collections.defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)
```

```
        dist[1] = 0
        Q = deque()
        Q.append((0, 1))
        while Q:
            cost, node = Q.popleft()
            for nei in graph[node]:
                new_cost = cost + time
                # update red signal
                if (cost // change) % 2 == 1:
                    new_cost += change - (cost % change)
                # update two distance
                if dist[nei] > new_cost:
                    dist2[nei], dist[nei] = dist[nei], new_cost
                    Q.append((new_cost, nei))
                elif new_cost > dist[nei] and new_cost < dist2[nei]:
                    dist2[nei] = new_cost
                    Q.append((new_cost, nei))
        return dist2[n]
```

[ ]:
```
# https://leetcode.com/problems/path-with-maximum-probability/solutions/731767/
↪java-python-3-2-codes-bellman-ford-and-dijkstra-s-algorithm-w-brief-explanation-and-analysi
↪
```

Flyod Warshall Algorithm

[ ]:
```
# https://leetcode.com/problems/
↪find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/
# given undirected graph with n cities and distance threshold, find the city␣
↪with smallest number of neighbors at a threshold distance

# flyod warshall algorithm to find the shortest distance between all pairs of␣
↪cities
# Time complexity - O(n^3)

class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], distanceThreshold:␣
↪int) -> int:
        dist = [[math.inf] * n for _ in range(n)]
        for i in range(n):
            dist[i][i] = 0

        for u, v, w in edges:
            dist[u][v] = dist[v][u] = w

        # Floy-Warshall's algorithm
        for k in range(n):
            for i in range(n):
```

```
                for j in range(n):
                    if dist[i][j] > dist[i][k] + dist[k][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]

        cnt = [0] * n
        for i in range(n):
            for j in range(n):
                if i == j: continue
                if dist[i][j] <= distanceThreshold:
                    cnt[j] += 1

        ans = 0
        for i in range(n):
            if cnt[i] <= cnt[ans]:
                ans = i
        return ans
```

[ ]:
```
# https://leetcode.com/problems/minimum-cost-to-convert-string-i/description/
# use floyd warshall algorithm to find the shortest path between all pairs of
 ↪nodes
# dijkstra algorithm can also be used to solve this problem but leads to TLE

from typing import List
from collections import defaultdict
import sys

class Solution:
    def minimumCost(self, source: str, target: str, original: List[str],
 ↪changed: List[str], cost: List[int]) -> int:
        # Create a set of all characters in original and changed lists
        all_chars = set(original + changed)

        # Initialize the distance dictionary with infinity
        dist = defaultdict(lambda: defaultdict(lambda: float('inf')))

        # Distance from a node to itself is zero
        for char in all_chars:
            dist[char][char] = 0

        # Fill initial distances based on the input lists
        for orig, chng, c in zip(original, changed, cost):
            dist[orig][chng] = min(dist[orig][chng], c)

        # Floyd-Warshall algorithm to compute shortest paths between all pairs
 ↪of nodes
        for k in all_chars:
            for i in all_chars:
```

```python
                for j in all_chars:
                    if dist[i][k] < float('inf') and dist[k][j] < float('inf'):
                        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

        total_cost = 0

        # Compute the total cost by looking up the precomputed shortest paths
        for s, t in zip(source, target):
            if s == t:
                continue
            if dist[s][t] == float('inf'):
                return -1
            total_cost += dist[s][t]

        return total_cost


# https://leetcode.com/problems/minimum-cost-to-convert-string-ii/description/
```