Exercícios-Problemas (EP)

Semana 1:

https://www.facom.ufu.br/~albertini/grafos/

EP1

Our sad tale begins with a tight clique of friends. Together they went on a trip to the picturesque country of Molvania. During their stay, various events which are too horrible to mention occurred. The net result was that the last evening of the trip ended with a momentous exchange of “I never want to see you again!”s. A quick calculation tells you it may have been said almost 50 million times! Back home in Scandinavia, our group of ex-friends realize that they haven’t split the costs incurred during the trip evenly. Some people may be out several thousand crowns. Settling the debts turns out to be a bit more problematic than it ought to be, as many in the group no longer wish to speak to one another, and even less to give each other money. Naturally, you want to help out, so you ask each person to tell you how much money she owes or is owed, and whom she is still friends with. Given this information, you’re sure you can figure out if it’s possible for everyone to get even, and with money only being given between persons who are still friends.

Input

The first line contains two integers, N ($2 \leq N \leq 10000$), and M ($0 \leq M \leq 50000$), the number of friends and the number of remaining friendships. Then N lines follow, each containing an integer O ($-10000 \leq O \leq 10000$) indicating how much each person owes (or is owed if $O < 0$). The sum of these values is zero. After this comes M lines giving the remaining friendships, each line containing two integers X, Y ($0 \leq X < Y \leq N - 1$) indicating that persons X and Y are still friends.

Output

Your output should consist of a single line saying “POSSIBLE” or “IMPOSSIBLE”.

```python
class Grafo:
    def __init__(self, num_nos):
        self.grafo = {}
        for i in range(num_nos):
            self.grafo[i] = []

    def insere_aresta(self, u, v):
        self.grafo[u].append(v)
        self.grafo[v].append(u)

def dfs(grafo, no, visitado, divida):
    visitado[no] = True
    divida_total = divida[no]

    for vizinho in grafo[no]:
        if not visitado[vizinho]:
            divida_total += dfs(grafo, vizinho, visitado, divida)

    return divida_total


def main():
    try:
        # Lê os valores de N e M na mesma linha
        N, M = map(int, input().split())

        # Verifica se os valores estão dentro das restrições
        if 2 <= N <= 10000 and 0 <= M <= 50000:
            # Lê os valores das dívidas em N linhas
            dividas = []
            for _ in range(N):
                num = int(input())
                if -10000 <= num <= 10000:
                    dividas.append(num)
                else:
                    print("IMPOSSIBLE")
                    return

            grafo = Grafo(N)
            # Lê os pares de valores X e Y em M linhas
            for _ in range(M):
                X, Y = map(int, input().split())
                if 0 <= X < Y <= N - 1:
                    grafo.insere_aresta(X,Y)
                else:
                    print("IMPOSSIBLE")
                    return

            visitado = [False] * N

            for no in range(N):
                if not visitado[no]:
                    divida_total = dfs(grafo.grafo, no, visitado, dividas)
                    if divida_total != 0:
                        print("IMPOSSIBLE")
                        return
            print("POSSIBLE")
            return
        else:
            print("IMPOSSIBLE")
            return
    except ValueError:
        print("IMPOSSIBLE")
        return

if __name__ == "__main__":
    main()
```

EP2

In 1976 the “Four Color Map Theorem” was proven with the assistance of a computer. This theorem states that every map can be colored using only four colors, in such a way that no region is colored using the same color as a neighbor region. Here you are asked to solve a simpler similar problem. You have to decide whether a given arbitrary connected graph can be bicolored. That is, if one can assign colors (from a palette of two) to the nodes in such a way that no two adjacent nodes have the same color. To simplify the problem you can assume:

• no node will have an edge to itself.

• the graph is nondirected. That is, if a node a is said to be connected to a node b, then you must assume that b is connected to a.

• the graph will be strongly connected. That is, there will be at least one path from any node to any other node.

Input

The input consists of several test cases. Each test case starts with a line containing the number n ($1 < n < 200$) of different nodes. The second line contains the number of edges l. After this, l lines will follow, each containing two numbers that specify an edge between the two nodes that they represent.

A node in the graph will be labeled using a number a ($0 \leq a < n$).

An input with n = 0 will mark the end of the input and is not to be processed.

Output

You have to decide whether the input graph can be bicolored or not, and print it as shown below.

EP2

Sample Input

```
3
3
0 1
1 2
2 0
3
2
0 1
1 2
9
8
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0
```

Sample Output

NOT BICOLORABLE.

BICOLORABLE.

BICOLORABLE.

```python
def bicolorable_bfs(grafo, no_inicial, cor):
    visitar = [no_inicial]
    cor[no_inicial] = 0  # Atribui a primeira cor ao nó inicial

    while visitar:
        no_atual = visitar.pop(0)

        for vizinho in grafo.get(no_atual, []):
            if cor[vizinho] == -1:
                # Atribui a cor oposta ao vizinho
                cor[vizinho] = 1 - cor[no_atual]
                visitar.append(vizinho)
            elif cor[vizinho] == cor[no_atual]:
                return False  # O grafo não pode ser bicolorido se nós a
                                        adjacentes têm a mesma cor
    return True

def bicolorable_dfs(grafo, no_atual, cor_atual, cor):
    cor[no_atual] = cor_atual

    for vizinho in grafo.get(no_atual, []):
        if cor[vizinho] == -1:
            if not bicolorable_dfs(grafo, vizinho, 1 - cor_atual, cor):
                return False
        elif cor[vizinho] == cor_atual:
            return False

    return True
```

```python
def main():
    while True:
        n = int(input())
        if n < 1 or n > 200:
            print("NOT BICOLORABLE.")
            break

        l = int(input())
        arestas = []
        for _ in range(l):
            a, b = map(int, input().split())
            arestas.append((a, b))

        grafo = {}
        #Construção grafo não-direcionado
        for a, b in arestas:
            if a not in grafo:
                grafo[a] = []
            if b not in grafo:
                grafo[b] = []
            grafo[a].append(b)
            grafo[b].append(a)

        cores = [-1] * n

        if bicolorable_bfs(grafo, 0, cores) and bicolorable_dfs(grafo, 0, 0, cores):
            print("BICOLORABLE.")
        else:
            print("NOT BICOLORABLE.")

if __name__ == "__main__":
    main()
```

EP3

Sam found a big bunch of maps from old Maester Aemon, which at a first look, should point, each one, a location of a chest full of obsidian. However, after taking a better look, some maps had obvious errors, while others, only sending a team of explorers to know.

What is known is that some maps point to an absurd location outside of the map and some end up in circles, ending up to be completely useless.

Since the maps are many, the brothers of the Nights Watch are few and winter is comming, your work is to write a program to check if a map leads or not to a chest with obsidian.

Maps have these features:

The starting point is always at the top left corner.

The maps are rectangular and each point of the map has one of these symbols:

A traversable terrain space.

An arrow, representing a possible change of direction.

A chest.

Since the places these maps describe are very dangerous, it is vital that the path described in the map is strictly followed.

EP3

Input

The first line contains a positive integer x < 100 with the width of the map.

The second line contains a positive integer y < 100 with the height of the map.

The following lines contain various characters within the map's dimensions.

The valid characters are:

An arrow to the right: >
An arrow to the left: <
An arrow pointing down: v
An arrow pointing up: ^
A space of traversable terrain: .
A chest: *

Output

The output must consist of a single line containing a single character ! or *.

! means that the map is invalid. * means that the map is valid.

EP4

The GeoSurvComp geologic survey company is responsible for detecting underground oil deposits.

GeoSurvComp works with one large rectangular region of land at a time, and creates a grid that divides the land into numerous square plots. It then analyzes each plot separately, using sensing equipment to determine whether or not the plot contains oil.

A plot containing oil is called a pocket. If two pockets are adjacent, then they are part of the same oil deposit. Oil deposits can be quite large and may contain numerous pockets. Your job is to determine how many different oil deposits are contained in a grid.

Input

The input file contains one or more grids. Each grid begins with a line containing m and n, the number of rows and columns in the grid, separated by a single space. If m = 0 it signals the end of the input; otherwise $1 \leq m \leq 100$ and $1 \leq n \leq 100$. Following this are m lines of n characters each (not counting the end-of-line characters). Each character corresponds to one plot, and is either '*', representing the absence of oil, or '@', representing an oil pocket.

Output

For each grid, output the number of distinct oil deposits. Two different pockets are part of the same oil deposit if they are adjacent horizontally, vertically, or diagonally. An oil deposit will not contain more than 100 pockets.

EP4

Sample Input

```
1 1
*
3 5
*@*@*
**@**
*@*@*
1 8
@@****@*
5 5
****@
*@@*@
*@**@
@@@*@
@@**@
0 0
```

Sample Output

```
0
1
2
2
```

```python
def find_adjacent_pockets(grid, row, col):
    # Função para encontrar pockets adjacentes a partir de uma posição (row, col)
    # Retorna uma lista de posições adjacentes

    adjacent_positions = []

    # Possíveis movimentos (horizontal, vertical e diagonal)
    moves = [(-1, -1), (-1, 0), (-1, 1),
             (0, -1),           (0, 1),
             (1, -1), (1, 0), (1, 1)]

    for dr, dc in moves:
        newRow, newCol = row + dr, col + dc
        # Verificar se existe a linha e a coluna adjacente no grid e se é pocket de petróleo
        if 0 <= newRow < len(grid) and 0 <= newCol < len(grid[0]) and grid[newRow][newCol] == '@':
            adjacent_positions.append((newRow, newCol))

    return adjacent_positions

def explore_deposit(grid, row, col):
    # Função para explorar um depósito a partir de uma posição (row, col)
    # Marca o depósito como visitado e explora os pockets adjacentes

    grid[row][col] = '*'  # Marca com '*' os pockets de petróleo adjacentes visitados
    adjacent_pockets = find_adjacent_pockets(grid, row, col)

    for adj_row, adj_col in adjacent_pockets:
        if grid[adj_row][adj_col] == '@':
            # Recursão para visitar todos os pockets adjacentes possíveis
            explore_deposit(grid, adj_row, adj_col)

```

```python
def count_oil_deposits(grid):
    # Função para contar o número de depósitos de petróleo em um grid
    num_deposits = 0
    for row in range(len(grid)):
        for col in range(len(grid[0])):
            if grid[row][col] == '@':
                num_deposits += 1
                explore_deposit(grid, row, col)

    return num_deposits

def main():
    while(True):
        # Lê os valores de m e n na mesma linha
        m, n = map(int, input().split())
        if m == 0 or n == 0:
            print("Fim da execução")
            break
        # Verifica se os valores estão dentro das restrições
        if 1 <= m <= 100 and 1 <= n <= 100:
            grid = []
            for _ in range(m):
                linha = input()
                if len(linha) != n:
                    print("Largura da linha é: ", n)
                    break
                if any(caracter not in ['@', '*'] for caracter in linha):
                    print("Caracteres permitidos da linha são @ e *: ")
                    break
                grid.append(list(linha))

                num_deposits = count_oil_deposits(grid)
                print(num_deposits)
            else:
                print("Erro: 1 ≤ m ≤ 100 and 1 ≤ n ≤ 100")
                break

if __name__ == "__main__":
    main()
```