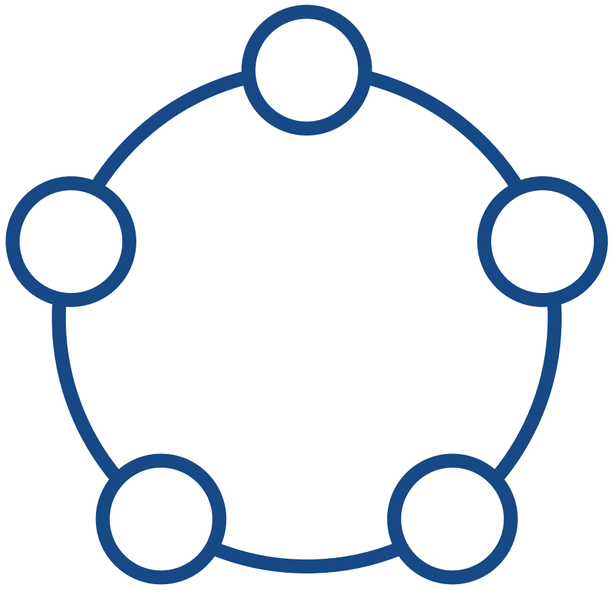


Semana 1

Teoria de Grafos

Ana Luiza Custódio, Gabriel Antônio, Guilherme Cabral

Problemas

01 Dinheiro Importa

02 Bicolor

03 Mapa do Mestre

04 Depósitos de petróleo

01 Dinheiro Importa

Dinheiro Importa

Nossa triste história começa com um grupo restrito de amigos. Juntos, eles fizeram uma viagem ao pitoresco país da Molvânia. Durante sua estada, ocorreram vários eventos que são horríveis demais para serem mencionados. O resultado líquido foi que a última noite da viagem terminou com uma importante troca de “nunca mais quero ver você!”. Um cálculo rápido mostra que pode ter sido dito quase 50 milhões de vezes! De volta à Escandinávia, nosso grupo de ex-amigos percebe que não dividiu igualmente os custos da viagem. Algumas pessoas podem perder vários milhares de coroas. Pagar as dívidas acaba sendo um pouco mais problemático do que deveria, pois muitos do grupo não querem mais se falar e muito menos dar dinheiro uns aos outros. Naturalmente, você quer ajudar, então pede a cada pessoa que lhe diga quanto dinheiro ela deve ou deve e de quem ela ainda é amiga. Dada essa informação, você tem certeza de que pode descobrir se é possível todos ficarem quites, e com dinheiro sendo dado apenas entre pessoas que ainda são amigas.

Dinheiro Importa

Entrada:
A primeira linha contém dois inteiros, N ($2 \leq N \leq 10000$) e M ($0 \leq M \leq 50000$), o número de amigos e o número de amizades restantes. Então N linhas seguem, cada uma contendo um inteiro O ($-10000 \leq O \leq 10000$) indicando quanto cada pessoa deve (ou é devido se $O < 0$). A soma desses valores é zero. Depois disso, vêm M linhas com as amizades restantes, cada linha contendo dois inteiros X, Y ($0 \leq X < Y \leq N - 1$) indicando que as pessoas X e Y ainda são amigas.

Saída:
Sua saída deve consistir em uma única linha dizendo “POSSÍVEL” ou “IMPOSSÍVEL”.

Dinheiro Importa

Estrutura de dados: Lista de adjacência

Vértices: Os vértices são pessoas, além disso cada vértice tem um peso (que seria o valor que ele deve receber ou pagar)

Arestas: Relacionamento de amizade entre as pessoas

```cpp
// Link do problema: https://www.beecrowd.com.br/judge/pt/problems/view/3215
// Status - ACCEPTED
// Comeplexidade - O(N + M)

#include <bits/stdc++.h>
using namespace std;

#define NMAX 10010
// No pior caso, terá esse número de vértices então a ideia é criar
// a estrutura que já suporta esse número máximo de vértices

vector< int > grafo[NMAX];
// Lista de adjacência

int valor[NMAX];
// Valor de cada pessoa

bool marc[NMAX];
// Se já passou por essa pessoa na DFS
```

```cpp
int DFS(int u) // Busca em profundidade - O(N + M)
{

        marc[u] = true;

        int x = valor[u];

        for(auto v : grafo[u])
        {

                if(marc[v] == true) continue;

                x += DFS(v);

        }

        return x;

}
// A ideia é que todos os vértices que o vértice inicial alcançar
// será somada à resposta
```

```cpp
int main()
{

        int n, m, u, v, i;

        string resp = "POSSIBLE"; // Suponha que a resposta seja POSSIBLE

        cin >> n >> m;

        for(i = 0;i < n;i++) cin >> valor[i];

        for(i = 0;i < m;i++)
        {

                cin >> u >> v;

                grafo[u].push_back(v); // Adiciona aresta
                grafo[v].push_back(u); // Adiciona aresta

        }
```

```
64            for(i = 0;i < n;i++) // Passa por todos os vértices
65            {
66
67                    if(marc[i] == true) continue;
68                    // Se o vértice já foi vizitado, sua componente já foi
69                    // verificada, assim não a nescessidade de rodar uma busca
70                    // nesse vértice
71
72                    if(DFS(i) != 0) resp = "IMPOSSIBLE";
73                    // Caso alguma componente tenha a soma diferente de 0, a
74                    // resposta é IMPOSSIBLE
75
76            }
77
78            // Apesar de aparentar que o código no pior dos casos irá rodar
79            // várias DFS, porém cada vértice passará pela função somente uma
80            // vez, assim todo vértice (e por consequência toda aresta) será
81            // analizado somente uma vez, ficando o código assim com
82            // complexidade O(N + M) amortizado
83
84            cout << resp << endl;
85
86        return 0;
87
88    }
```

Exemplos rodados

```
cabral@cabral-notebook:~/Área de Trabalho$ g++ codigo.cpp
cabral@cabral-notebook:~/Área de Trabalho$ ./a.out
5 3
100
-75
-25
-42
42
0 1
1 2
3 4
POSSIBLE
cabral@cabral-notebook:~/Área de Trabalho$ ./a.out
4 2
15
20
-10
-25
0 2
1 3
IMPOSSIBLE
```

02 Bicolor

Bicolor

Em 1976, o “Teorema do Mapa das Quatro Cores” foi provado com a ajuda de um computador. Este teorema afirma que todo mapa pode ser colorido com apenas quatro cores, de forma que nenhuma região seja colorida usando a mesma cor de uma região vizinha. Aqui você é solicitado a resolver um problema semelhante mais simples. Você tem que decidir se um dado arbitrário grafo conectado pode ser bicolor. Ou seja, se for possível atribuir cores (de uma paleta de duas) aos nós de tal forma que não haja dois nós adjacentes com a mesma cor. Para simplificar o problema, você pode assumir:

- nenhum nó terá uma aresta para si mesmo.
- o gráfico é não direcionado. Ou seja, se diz-se que um nó a está conectado a um nó b, então você deve suponha que b está conectado a a.
- o gráfico será fortemente conectado. Ou seja, haverá pelo menos um caminho de qualquer nó para qualquer outro nó.

Bicolor

Entrada:
A entrada consiste em vários casos de teste. Cada caso de teste começa com uma linha contendo o número n ($1 < n < 200$) de nós diferentes. A segunda linha contém o número de arestas l. Depois disso, l linhas serão seguem, cada um contendo dois números que especificam uma borda entre os dois nós que eles representam. Um nó no grafo será rotulado usando um número a ($0 \leq a < n$). Uma entrada com $n = 0$ marcará o fim da entrada e não deve ser processada.

Saída:
Você deve decidir se o gráfico de entrada pode ser bicolor ou não e imprimi-lo conforme mostrado abaixo.

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.
```

Bicolor

Estrutura de dados: Lista de adjacência

Vértices: São as localidades representadas em um mapa

Arestas: Se duas localidades fazem fronteira uma com a outra, elas possuem uma aresta interligando-as

```java
import java.util.Scanner;
//Gabriel Antonio

public class Main {
    /*Motivacao: Um mapa deve ser colorido utilizando apenas duas cores,
     o problema pede que se mostre se é possivel realizar tal tarefa
     levando em conta que cada localidade deve ter cor diferente dos seus adjacentes*/

    /* Aqui, cada localidade do mapa é representada por um vértice do grafo, e, as fronteiras que
     as ligam são representadas por arestas*/

    /* Cada vértice do grafo guarda como informação seu indice e sua cor, para facilitar o trabalho
     definimos a cor como um inteiro, são 4 possiveis cores, 0, que significa que o vértice ainda não foi colorido(inicial),
     1 que representa a primeira cor, 2 que representa a segunda e 3 que representa um erro na coloração, assim que
     um vértice recebe a cor 3, o programa identifica e já responde à sua execução com a mensagem (NOT BICOLORABLE.)*/

    /*A função principal que possibilita a coloração é a função "decideCor", que recebe como parametro o indice do vértice a colorir
    e avalia as cores de todos os adjacentes, retornando um inteiro que é a cor que será atribuida ao vértice,
    tanto na função baseada na DFS quando na BSF a função é utilizada de maneira semelhante*/

    /*O grafo é representado por listas de adjacencia*/
```

```java
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Grafo mapa;
        int aux, a, b;

        System.out.println("Digite numero de vertices(max 200):");
        aux = sc.nextInt();
        mapa = new Grafo(aux);

        System.out.println("Digite numero de arestas:");
        aux = sc.nextInt();

        //Recebe cada dupla de Vertices e adiciona nas respectivas listas de adjacencia
        for(int i = 0; i < aux; i++){
            System.out.println("Digite dois numeros referentes aos vertices que compoem arestas:");
            a = sc.nextInt();
            b = sc.nextInt();
            mapa.novaAresta(a,b);
        }
        System.out.println("1 - Colorir por DFS");
        System.out.println("2 - Colorir por BFS");
        aux = sc.nextInt();

        if(aux == 1) mapa.coloreDFS(0);
        else mapa.coloreBFS(0);

        if(mapa.getRes() == 1){
            System.out.println("NOT BICOLORABLE.");
        }else System.out.println("BICOLORABLE.");
    }
}
```

```java
import java.util.ArrayList;

public class Vertice {
    int indice;
    int cor;

    public Vertice(int i){
        indice = i;
        cor = 0;
    }
}
```

```java
import java.util.ArrayList;
public class Lista {
    ArrayList<Vertice> lista;

    public Lista(){
        lista = new ArrayList<Vertice>();
    }

}
```

```java
import java.util.ArrayList;
import java.util.Scanner;

public class Grafo {
    int n_vert; //numero de vertices
    Lista[] adj; //vetor de listas de adjacencia
    private int res; //resposta final à coloração

    public Grafo(int v){
        n_vert = v;
        adj = new Lista[v];
        res = 0;
        for(int i = 0; i < v; i++){
            adj[i] = new Lista();
            adj[i].lista.add(new Vertice(i));
        }
    }
    public void novaAresta(int a, int b){
        adj[a].lista.add(adj[b].lista.get(0));
        adj[b].lista.add(adj[a].lista.get(0));
    }

```

```java
public void mostraG(){
    System.out.println("Grafo: ");
    for(int i = 0; i < adj.length; i++){
        for(int j = 0; j < adj[i].lista.size(); j++){
            System.out.println(adj[i].lista.get(j).indice + " de cor " + adj[i].lista.get(j).cor);
        }
        System.out.println();
    }
}
```

```java
public int decideCor(int a){
    int c1 = 0, c2 = 0, nl = 0;
    for(int i = 1; i < adj[a].lista.size(); i++){
        if(adj[a].lista.get(i).cor == 1){
            c1++;
        }else if(adj[a].lista.get(i).cor == 2){
            c2++;
        }else nl++;
    }
    if(c1 != 0 && c2 != 0){
        return 3;
    }
    if(c2 == 0){
        return 2;
    }else return 1;
}
```

```java
/*Função: ColoreDFS
* Recebe como parametro o indice do vértice inicial, atribui a ele uma cor,
* avalia a validez da cor, então entra em um laço for que recursivamente chama a função
* para todos os vértices adjacentes ao inicial, promovendo uma busca em profundidade colorindo cada um
* que ainda não tenha sido colorido*/
public void coloreDFS(int a){
    adj[a].lista.get(0).cor = decideCor(a);
    if(adj[a].lista.get(0).cor == 3){
        setRes(1);
        return;
    }
    for(int i = 1; i < adj[a].lista.size(); i++) {
        if (adj[a].lista.get(i).cor == 0) {
            coloreDFS(adj[a].lista.get(i).indice);
        }
    }
}
```

```java
/*Função: ColoreBFS
* Recebe o primeiro vértice como parametro, cria uma fila e coloca este vértice nela,
* então, num laço while, que se repetirá enquanto a fila não está vazia, percorre todos os
* vértices adjacentes ao véertice inicial da fila, adicionando-os a ela se eles ainda não foram
* coloridos e se eles ainda não estão presentes na fila.
* Por fim, colore o primeiro vértice da fila com sua respectiva cor, testa se sua cor é valida,
* então remove-o do inicio da fila, fazendo com que o primeiro de seus vértices adjacentes ocupe
* esta posição, com o qual o proximo ciclo de while vai trabalhar, assim, colorindo o vetor de acordo
* com a estratégia BFS de busca*/
public void coloreBFS(int a){
    ArrayList<Integer> fila = new ArrayList<>();
    fila.add(a);

    while(!fila.isEmpty()){
        for(int i = 1; i < adj[fila.get(0)].lista.size(); i++) {
            if(!(fila.contains(adj[fila.get(0)].lista.get(i).indice)) && adj[fila.get(0)].lista.get(i).cor == 0){
                fila.add(adj[fila.get(0)].lista.get(i).indice);
                System.out.println("Adicionando " + adj[fila.get(0)].lista.get(i).indice + " a fila");
            }
        }
        adj[fila.get(0)].lista.get(0).cor = decideCor(fila.get(0));
        if(adj[fila.get(0)].lista.get(0).cor == 3){
            setRes(1);
            return;
        }
        fila.remove(0);
    }
}
```

```
95          public int getRes() {
96              return res;
97          }
98
99          public void setRes(int res) {
100             this.res = res;
101         }
102
103     }
```

Exemplos rodados

```
"C:\Program Files\Eclipse Adoptium\jdk-17.0.8.7-hotspot\bin\java.exe"
Digite numero de vertices(max 200):
4
Digite numero de arestas:
4
Digite dois numeros referentes aos vertices que compoem arestas:
0 1
Digite dois numeros referentes aos vertices que compoem arestas:
0 3
Digite dois numeros referentes aos vertices que compoem arestas:
1 2
Digite dois numeros referentes aos vertices que compoem arestas:
2 3
1 - Colorir por DFS
2 - Colorir por BFS
1
BICOLORABLE.

Process finished with exit code 0
```

03 Mapa do Mestre

Mapa do Mestre

Sam encontrou um monte de mapas do velho Mestre Aemon, que à primeira vista, deveriam apontar, cada um, a localização de um baú cheio de obsidiana. Porém, olhando melhor, alguns mapas apresentavam erros óbvios, enquanto outros, apenas enviando uma equipe de exploradores para saber. O que se sabe é que alguns mapas apontam para um local absurdo fora do mapa e alguns acabam em círculos, acabando por ser completamente inúteis. Como os mapas são muitos, os irmãos da Patrulha da Noite são poucos e o inverno está chegando, seu trabalho é escrever um programa para verificar se um mapa leva ou não a um baú com obsidiana.

Os mapas têm estes recursos:

- O ponto de partida está sempre no canto superior esquerdo.
- Os mapas são retangulares e cada ponto do mapa possui um destes símbolos:
  - Um espaço de terreno atravessável.
  - Uma seta, representando uma possível mudança de direção.
  - Um baú.

Mapa do Mestre

Entrada:

A primeira linha contém um inteiro positivo x < 100 com a largura do mapa.
A segunda linha contém um inteiro positivo y < 100 com a altura do mapa.
As linhas a seguir contêm vários caracteres dentro das dimensões do mapa.
Os caracteres válidos são:

Uma seta para a direita: >
Uma seta para a esquerda: <
Uma seta apontando para baixo: v
Uma seta apontando para cima: ^
Um espaço de terreno percorrível: .
Um baú: *

Saída:

A saída deve consistir em uma única linha contendo um único caractere! ou *.
! significa que o mapa é inválido. * significa que o mapa é válido.

Mapa do Mestre

Estrutura de dados: Matriz de Adjacência

Vértices: Cada caractere de direção (>,<,^,v) e o baú (*) presente na matriz

Arestas: Distância (.) entre os vértices

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {

        int width, height; //dimensões do mapa (grafo)

        //Recebendo as dimensões do mapa pelo usuário
        Scanner scanner = new Scanner(System.in);
        width = scanner.nextInt();
        height = scanner.nextInt();

        Grafo mapa = new Grafo(width, height); //Criar mapa
        mapa.preencherMapa(); //Preencher o para com as informações dadas pelo usuário
        mapa.validarMapa(); //Verificar se o mapa é válido ou não
        scanner.close();
    }
}
```

```java
import java.util.ArrayList;
import java.util.Scanner;

public class Grafo {
    int width, height; //dimensões do mapa (grafo)
    char[][] matriz; //matriz de adjacência

    //Criação do mapa
    public Grafo(int width, int height){
        this.width = width;
        this.height = height;
        matriz = new char[height][width];
    }
```

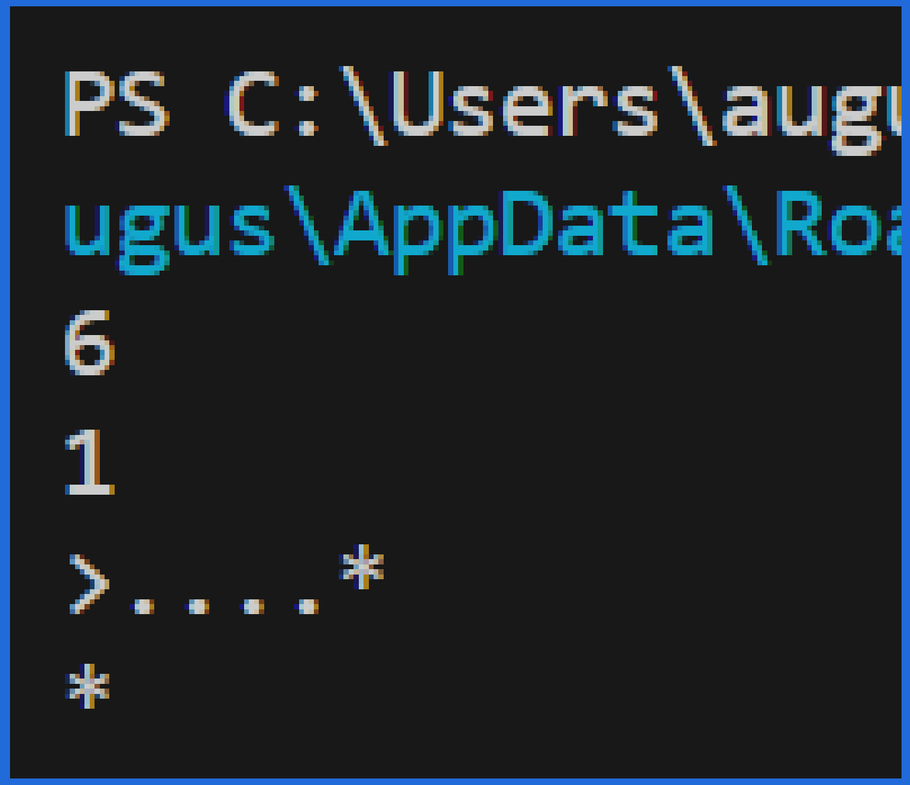
```java
//Preenchendo o mapa com as informações dada pelo usuário
public void preencherMapa(){
    Scanner scanner = new Scanner(System.in);
    for (int i = 0; i < height; i++) {
        matriz[i] = scanner.next().toCharArray(); //Preenche cada linha da matriz com a linha digitada
    }
}
```

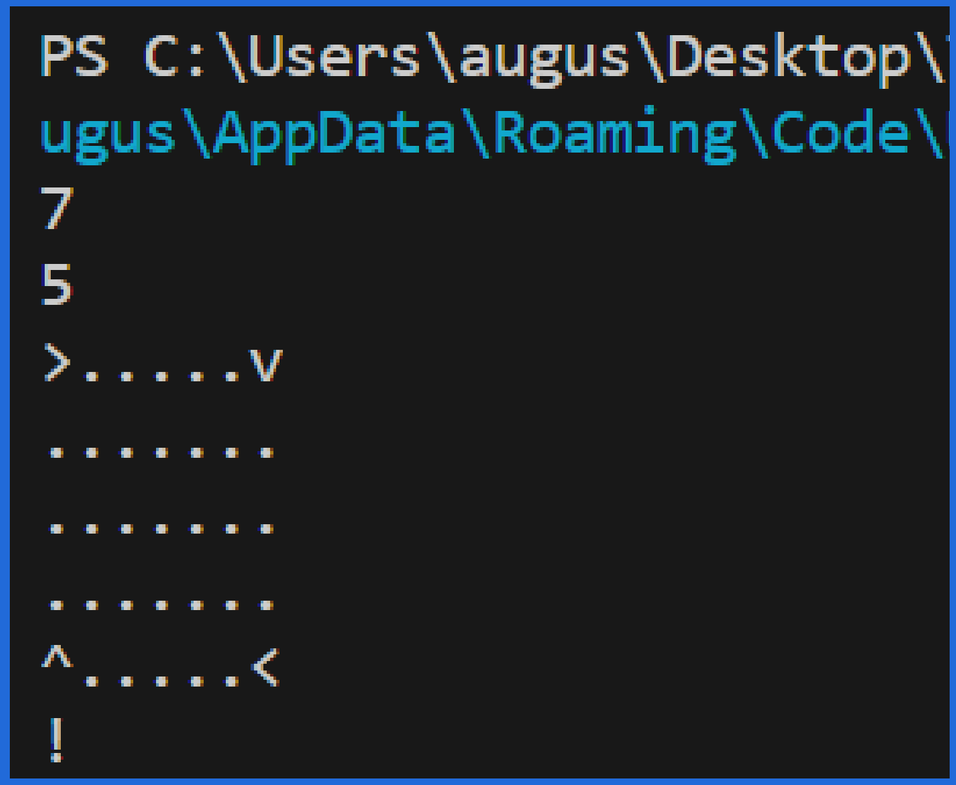
```java
//Validação do mapa fornecido
public void validarMapa(){
    boolean[][] visitado = new boolean[height][width];

    //Movimentação dentro do mapa
    int x = 0, y = 0, novo_X = 0, novo_Y = 0; //posições atuais e novas posições
    //O laço continua enquanto a posição atual (x, y)
    //é válida (dentro das dimensões do mapa)
    while (validarPosicao(x, y, height, width)) {
        // Se a posição atual já foi visitada (vis[x][y] == true),
        // isso significa que o labirinto não tem solução e imprime "!"
        if (visitado[x][y] == true) {
            System.out.println("!");
        }
        // Se a posição atual contém o baú ("*")
        // Então o mapa é validado e imprime "*"
        if (matriz[x][y] == '*') {
            System.out.println("*");
        }
```

```
            //Movimentação para direita(>), esquerda(<),
            //para baixo(v) ou para cima(^)
            if (matriz[x][y] == '>') {
                novo_X = 0;
                novo_Y = 1;
            } else if (matriz[x][y] == '<') {
                novo_X = 0;
                novo_Y = -1;
            } else if (matriz[x][y] == 'v') {
                novo_X = 1;
                novo_Y = 0;
            } else if (matriz[x][y] == '^') {
                novo_X = -1;
                novo_Y = 0;
            }

            // A posição atual é marcada como visitada (vistado[x][y] = true).
            visitado[x][y] = true;
            //Atualiza a posição de x e y
            x += novo_X;
            y += novo_Y;
        }
    }
```

```java
    //Verifica se a posição é válida dentro dos limites do mapa
    public static boolean validarPosicao(int x, int y, int height, int width) {
        return x > -1 && y > -1 && x < height && y < width;
    }
}
```

Exemplos rodados





04 Depósitos de Petróleo

Depósitos de Petróleo

A empresa de pesquisa geológica GeoSurvComp é responsável pela detecção de depósitos de petróleo subterrâneos. O GeoSurvComp trabalha com uma grande região retangular de terra por vez e cria uma grade que divide a terra em vários lotes quadrados. Em seguida, analisa cada parcela separadamente, usando equipamentos de detecção para determinar se a parcela contém óleo ou não. Um lote contendo óleo é chamado de bolso. Se dois bolsos são adjacentes, eles fazem parte do mesmo depósito de petróleo. Os depósitos de petróleo podem ser bastante grandes e conter vários bolsões. Seu trabalho é determinar quantos depósitos de petróleo diferentes estão contidos em uma grade.

Depósitos de Petróleo

Entrada:
O arquivo de entrada contém uma ou mais grades. Cada grade começa com uma linha contendo m e n, o número de linhas e colunas na grade, separadas por um único espaço. Se m = 0 sinaliza o fim da entrada; caso contrário, 1 ≤ m ≤ 100 e 1 ≤ n ≤ 100. Em seguida, há m linhas de n caracteres cada (sem contar os caracteres de fim de linha). Cada caractere corresponde a um enredo e é '*', representando o ausência de óleo, ou '@', representando uma bolsa de óleo.

Saída:
Para cada grade, forneça o número de depósitos de petróleo distintos. Dois bolsos diferentes fazem parte do mesmo depósito de óleo se forem adjacentes horizontalmente, verticalmente ou diagonalmente. Um depósito de petróleo não conterámais de 100 bolsos.

Sample Input

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

Sample Output

```
0
1
2
2
```

```cpp
// Link do problema: https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=513
// Status - ACCEPTED
// Comeplexidade - O(NM)

#include <bits/stdc++.h>
using namespace std;

#define NMAX 110
// No pior caso, o grid será desse tamanho, então a ideia é criar
// a estrutura que já suporta esse grid máximo

bool mat[NMAX][NMAX];
// True : Posição é '@' e não foi vizitada
// False: Posição é '*' ou já foi vizitada

int dlin[] = {-1, -1, -1,  0, 0,  1, 1, 1};
int dcol[] = {-1,  0,  1, -1, 1, -1, 0, 1};
// Esses dois vetores as combinações de todas as direções possíveis,
// isso é, a alteração de linha e coluna em cada direção.
```

```
void DFS(int lin, int col)
{

        int nlin, ncol, i;

        mat[lin][col] = false;
        // Nesse ponto sabemos que ali pertencia o caractere '@', e
        // marcando false, não o vizitaremos novamente

        for(i = 0;i < 8;i++) // Andar pelas 8 direções
        {

                nlin = lin + dlin[i]; // Cálculo da nova linha
                ncol = col + dcol[i]; // Cálculo da nova coluna

                if(mat[nlin][ncol] == false) continue;
                // se a posição é '*' ou já foi vizitada, não faz sentido ir

                DFS(nlin, ncol);

        }

}
```

```cpp
int main()
{

        int n, m, resp, i, j;

        string s;

        while(cin >> n >> m)
        {

                if(n == 0 && m == 0) break;

                for(i = 0;i <= n + 1;i++)
                {

                        for(j = 0;j <= m + 1;j++)
                        {

                                mat[i][j] = false;
                                // Suponha que não podemos andar em nenhuma posição,
                                // como se todas fossem '*', assim serão setadas
                                // como false

                        }

                }
```

```
// Ir do 0 até o n + 1 pode gerar um estranhamento, porém
// aqui foi utilizada uma técnica para deixar a codificação
// mais simples, pois para evitar a DFS sair da matriz, foi
// adicionado uma espécie de 'padding' na matriz original,
// afim de deixar ela com bordas preenchidas por '*', vamos
// analizar um caso teste para melhor explicação:
//
//            *******
// *@*@*      **@*@**
// **@** ==> ***@***
// *@*@*      **@*@**
//            *******
//
```

```
for(i = 1;i <= n;i++)
{

        cin >> s;

        for(j = 1;j <= m;j++)
        {

                if(s[j - 1] == '@') mat[i][j] = true;
                // Todas as posições que podemos andar '@' serão
                // setadas como true

        }

}

resp = 0;
```

```
for(i = 1;i <= n;i++)
{

        for(j = 1;j <= m;j++)
        {

                if(mat[i][j] == false) continue;
                // Se estamos em um '*' ou já passarmos por essa
                // posição já contabilizamos sua componente, assim
                // podemos ignorar-lá

                resp++; // Achamos uma nova componente

                DFS(i, j);

        }

}
```

```
            // Apesar de aparentar que o código no pior dos casos irá
            // rodar várias DFS, porém cada vértice passará pela função
            // somente uma vez, assim todo vértice (e por consequência
            // toda aresta) será analizado somente uma vez, ficando o
            // código assim com complexidade O(NM) amortizado

            cout << resp << endl;

        }

    return 0;

}
```

Exemplos rodados

```
cabral@cabral-notebook:~/Área de Trabalho$ g++ codigo.cpp
cabral@cabral-notebook:~/Área de Trabalho$ ./a.out
1 1
*
0
3 5
*@*@*
**@**
*@*@*
1
1 8
@@****@*
2
5 5
****@
*@@*@
*@**@
@@@*@
@@**@
2
0 0
```