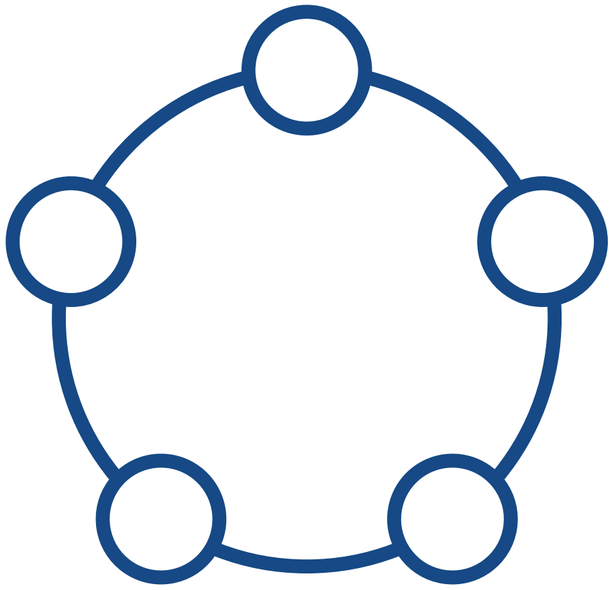


Semana 2

Teoria de Grafos

Ana Luiza Custódio, Gabriel Antônio, Guilherme Cabral

Problemas

01 Rare Order

02 Running MoM

03 Pick up sticks

01 Rare Order

Rare Order

Um colecionador de livros raros descobriu recentemente um livro escrito em um idioma desconhecido que usava o mesmo caracteres como o idioma Inglês. O livro continha um índice curto, mas a ordem dos itens no índice era diferente do que se esperaria se os caracteres fossem ordenados da mesma forma que no alfabeto inglês.
O colecionador tentou usar o índice para determinar a ordem dos caracteres (ou seja, a sequência de agrupamento) do alfabeto estranho, então desistiu frustrado com o tédio de a tarefa. Você deve escrever um programa para completar o trabalho do coletor. Em particular, seu programa levará um conjunto de strings que foi classificado de acordo com uma sequência de agrupamento específica e determina o que essa sequência é.

Rare Order

Entrada: consiste em uma lista ordenada de strings de letras maiúsculas, uma string por linha. Cada string contém no máximo 20 caracteres. O fim da lista é sinalizado por uma linha com o único caractere ‘#’. Nem todas as letras são necessariamente usadas, mas a lista implicará uma ordenação completa entre as letras que são usados.

Saída: deve ser uma única linha contendo letras maiúsculas na ordem que especifica o agrupamento sequência usada para produzir o arquivo de dados de entrada.

Sample Input

```
XWY
ZX
ZXY
ZXW
YWWX
#
```

Sample Output

```
XZYW
```

Rare Order

Estrutura de dados: Lista de adjacências

Vértices: Cada letra da string

Arestas: Se letra i vem antes de j na nova ordem alfabética, então temos uma aresta de i para j.

Rare Order (versão 1)

Para uma string x vir antes de outra string y no livro basta que a primeira posição que diferir entre x e y a letra de x vi antes da que de y na ordem à definir (no caso onde x = "ZX" e y = "ZXY", x sempre virá antes, como se o caracter '' viesse primeiro na ordem).

Ao invés de verificar todas as string dois a dois (o que nos resultaria em O(n^2)) podemos apenas verificar a string i com a i+1.

Prova: Vamos analizar três string x, y, z tais que x < y e y < z. Temos que a primeira posição em que x e y difere (p1), o caracter de x é menor, pois assumimos x < y. Analizando a primeira posição em que y e z diferem (p2) temos que podem ocorrer três casos:

Caso 1: p1 < p2: Temos que x[p1] < y[p1] = z[p1] ==> x[p1] < z[p1]

AAA

ABA

ABB

Caso 2: p1 = p2: Temos que x[p1] < y[p1] < z[p1] ==> x[p1] < z[p1]

AAA

ABA

ACA

Caso 3: p1 > p2: Temos que x[p2] = y[p2] < z[p2] ==> x[p2] < z[p2]

AAA

AAB

ABA

Logo se x < y e y < z temos garantido que x < z. ■

Com a observação provada acima temos que se garatirmos que a string 1 seja menor que 2, que a 2 seja menor que a 3, ... e que a n-1 seja menor que a n, então i será menor que j, para todo i < j.

Agora para garatir que a string i seja menor que a i+1, vamos denotar que a posição que elas diferem seja p, logo basta criar um grafo direcionado e liga os vértices i[p] e i+1[p] por uma aresta (isso é claro se i[p] != ''), e aplicando o algoritmo de ordem topológica garatimos que i[p] venha primeiro que i+1[p] na ordem final das letras.

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

#define NMAX 310
#define fi first
#define se second
#define pb push_back
#define mp make_pair

vector< int > grafo[NMAX]; // Grafo por lista de adjacência
bool has[NMAX];                        // Letras que apareceram
bool marc[NMAX];              // Se já passou pelo i-esimo vértice

string resp;
```

```cpp
void DFS(int u)
{

        marc[u] = true; // Marca que passamos pelo vértice

        for(auto v : grafo[u]) // Percorre os vizinhos de u
        {

                if(marc[v] == true) continue; // Ignora se já passou em v

                DFS(v);

        }

        // Parte do algoritmo de ordenação topológica
        resp.pb((char)(u));

        return;

}
```

```cpp
int main()
{

        int i;

        string s, last;

        while(cin >> s)
        {

                if(s == "#") break;

                for(auto letra : s) has[letra] = true;

                for(i = 0;i < last.size();i++) // Encontra a primeira letra
                                               // que diferem
                {

                        if(s[i] != last[i]) break;

                }

                if(i != last.size()) // Se i[p] != ''
                        grafo[last[i]].pb(s[i]);

                last = s;
```

```
    resp = "";

    for(i = 'A';i <= 'Z';i++)
    {

            // Pula a letra se ela não existe nas palavras ou já
            // passamos nela
            if(has[i] == false || marc[i] == true) continue;

            DFS(i);

    }

    // Parte do algoritmo de ordenação topológica
    reverse(resp.begin(), resp.end());

    cout << resp << endl;

return 0;
```

Rare Order (versão 2)

```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Stack;

public class Grafo {
    private int V;
    private LinkedList<Character> adj[];

    //Constructor
    public Grafo(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }
    public void adicionarAresta(char v,char w, int p) {
        adj[p].add(w); //p é o index do vértice v
    }
```

```
void dfs(int v, boolean marcado[], Stack<Character> stack, char [] c) {
    marcado[v] = true; //marca que o elemento foi visitado

    System.out.print("+ " + v + "\n");
    Iterator<Character> i = adj[v].listIterator(); // i será usado para percorrer os vizinhos do vértice v
    while (i.hasNext()) { //enquanto exister vizinhos de i, a busca continua
        int n = adj[v].indexOf(i.next()); //n é a posição do vizinho

        char viz = adj[v].get(n); // pega a letra do vizinho na posição n
        for(int p = 0; p < c.length; p++)
        {
```

```
            if(c[p] == viz)
            {

                    n = p; // n é a posição do vizinho em c

                    break;

            }

        }

        if (!marcado[n]) //caso o vizinho não tenha sido visitado, ocorrerá o dfs nesse vértice
            dfs(n, marcado, stack, c);
    }
    System.out.print("- " + v + "\n");
    stack.push(c[v]); //após analisar o vértice, ele é empilhado
}
```

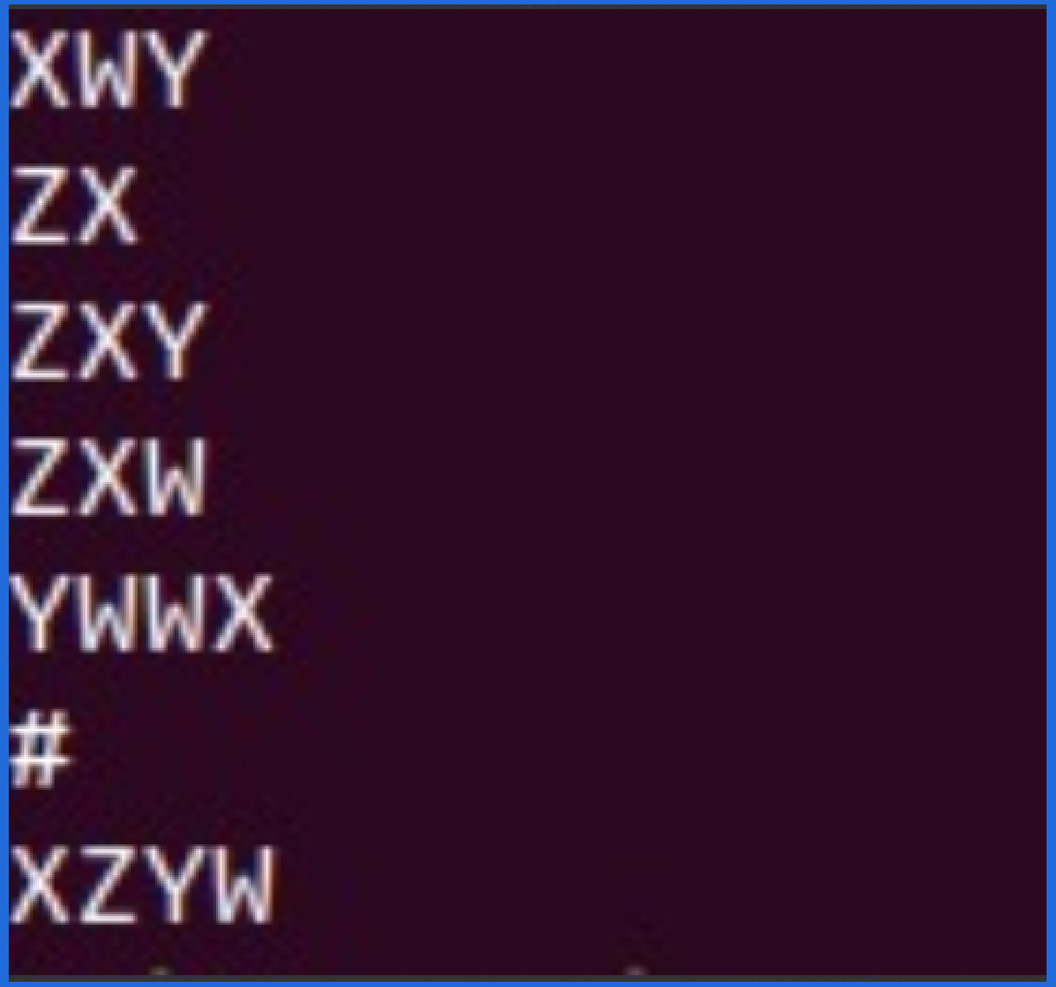
```java
void ordenacaoTopologica(char [] c) {
    Stack<Character> stack = new Stack<>();
    boolean marcado[] = new boolean[V];

    //aplica a dfs em cada vértice
    for (int i = 0; i < V; i++)
        if (marcado[i] == false)
            dfs(i, marcado, stack, c);

    //Imprime o resultado da ordenação topológica
    while (stack.empty()==false)
        System.out.print(stack.pop() + " ");

    System.out.print("\n");
}
```

Exemplos rodados



02 Running MoM

Running MoM

O ruim de ser um Homem do Mistério (MoM) internacional é que geralmente há alguém que quer te matar. Às vezes você tem que ficar fugindo apenas para se manter vivo. Você tem que pensar à frente. Você tem que ter certeza de não acabar preso em algum lugar sem saída.

Claro, nem todos os MoMs são abençoados com muita inteligência. Você vai escrever um programa para ajudá-los. Você vai garantir que nosso MoM saiba quais cidades são seguras para visitar e quais não são. Não basta apenas poder correr (ou voar) por um ou dois dias, temos que garantir que o MoM continue rodando o tempo que for necessário. Dada uma lista de voos diários regulares entre pares de cidades, você garantirá que nosso MoM nunca fique preso em uma cidade da qual não há como escapar. Diremos que há uma fuga de algum local se houver uma sequência infinitamente longa de cidades para as quais o MoM poderia voar, fazendo um voo por dia.

Running MoM

Entrada: começa com um número, 1 <= n <= 5000, dando o número de voos diários entre pares de cidades. Cada uma das próximas n linhas contém um par de nomes de cidades separados por um espaço. Cada nome de cidade é uma string de até 30 caracteres usando apenas os caracteres a–z, A–Z e sublinhado. Uma linha contendo o nome o seguido de d indica que há um voo só de ida da cidade o para a cidade d todos os dias. Não há voos com origem e destino na mesma cidade. A descrição dos voos diários é seguida por uma lista de até 1000 nomes de cidades previamente nomeadas, uma por linha. A lista termina no final do arquivo.

Saída: Seu trabalho é examinar a lista de nomes de cidades no final e determinar se há ou não uma fuga de cada uma. Para cada um, imprima o nome da cidade, seguido da palavra “safe” se houver fuga e “preso” se não houver fuga.

Sample Input 1

```
5
Arlington San_Antonio
San_Antonio Baltimore
Baltimore New_York
New_York Dallas
Baltimore Arlington
San_Antonio
Baltimore
New_York
```

Sample Output 1

```
San_Antonio safe
Baltimore safe
New_York trapped
```

Running MoM

Estrutura de dados: Lista de adjacências

Vértices: Cidades

Arestas: Se há um voo da cidade i para a j, então temos uma aresta de i para j

Running MoM (versão 1)

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

#define NMAX 100010
#define fi first
#define se second
#define pb push_back
#define mp make_pair
#define WHITE 0 // Não vizitado
#define GRAY  1 // Vizitando (ou chega em um ciclo)
#define BLACK 2 // Vizitado

vector< int > grafo[NMAX]; // Grafo por lista de adjacência
int color[NMAX];                    // Cor do i-esimo vértice

int newId = 1;                    //
map< string, int > id;  // Mapeando as cidades
```

```cpp
bool DFS(int u)
{

        // Estamos vizitando u
        color[u] = GRAY;

        for(auto v : grafo[u])
        {

                // Ignora se v já acabou
                if(color[v] == BLACK) continue;

                // Se v está sendo vizitado (ou chega em um ciclo), u chega
                // em um ciclo, assim retornamos true (e deixamos u GRAY)
                if(color[v] == GRAY) return true;

                if(DFS(v) == true) return true;

        }

        // Acabamos de visitar u, logo ele não pertence a um ciclo
        color[u] = BLACK;

        return false;
```

```cpp
int main()
{

        int n, m, u, v, i;

        string s;

        cin >> n;

        while(n--)
        {

                cin >> s;
                if(id[s] == 0) id[s] = newId++;
                u = id[s];

                cin >> s;
                if(id[s] == 0) id[s] = newId++;
                v = id[s];

                grafo[u].pb(v);
```

```
n = newId - 1;

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

        if(color[i] == WHITE) DFS(i);

}

// Apartir desse momento:
// color[u] = GRAY  ==> u chega num ciclo (safe)
// color[u] = BLACK ==> u não chega num ciclo (trapped)
```

```
    while(cin >> s)
    {

            u = id[s];

            if(color[u] == GRAY)    cout << s << " safe" << endl;
            else                                    cout << s << " trapped" << endl;

    }

return 0;
```

Running MoM (versão 2)

```java
import java.util.ArrayList;

public class Vertice {
    String city;
    int indice;
    int sfty; //-1 - não visitado // 0 - visitando// 1 ou 2 - visitado (1 - segura, 2 - não segura)

    public Vertice(String c,int ind){
        city = c;
        indice = ind;
        sfty = -1;
    }
}
```

```java
public class Grafo {
    ArrayList <ArrayList<Vertice>> adj; //vetor dinamico de listas de adjacencia

    public Grafo(){
        adj = new ArrayList<ArrayList<Vertice>>();
    }
    public void novaAresta(String a, String b){
        int ao, af;
        ao = buscaV(a);
        af = buscaV(b);

        if(ao == -1){
            addVr(a);
            ao = adj.size()-1;
        }
        if(af == -1){
            addVr(b);
            af = adj.size()-1;
        }
```

```
        adj.get(ao).add(adj.get(af).get(0));

    }
    public void addVr(String a){
        adj.add(new ArrayList<Vertice>());
        adj.get(adj.size()-1).add(new Vertice(a, adj.size()-1));
    }
    public int buscaV(String a){
        for(int i = 0; i < adj.size(); i++){
            if(a.equals(adj.get(i).get(0).city)){
                return i;
            }
        }
        return -1;
    }
```

```
public void mostraGrafo(){
    for(int i = 0; i< adj.size(); i++){
        System.out.println((i+1) + ": ");
        System.out.print("Raiz: ");
        for(int j = 0; j < adj.get(i).size(); j++){
            System.out.print(adj.get(i).get(j).city + " - ");
        }
        System.out.println();
    }
}
```

```java
public void mostraV(String a){
    int n = buscaV(a);
    if(adj.get(n).get(0).sfty == 1){
        System.out.println(adj.get(n).get(0).city + " safe");
    }else{
        System.out.println(adj.get(n).get(0).city + " trapped");
    }
}
```

```java
public void classificaGrafo(Vertice inic){
    adj.get(inic.indice).get(0).sfty = 0;
    int saida = 0;

    for(int i = 1; i < adj.get(inic.indice).size(); i++){
        if(adj.get(inic.indice).get(i).sfty == -1){
            classificaGrafo(adj.get(inic.indice).get(i));
        }
        if(adj.get(inic.indice).get(i).sfty == 0){
            adj.get(inic.indice).get(0).sfty = 1;
            saida++;
        }else if(adj.get(inic.indice).get(i).sfty == 1){
            adj.get(inic.indice).get(0).sfty = 1;
            saida++;
        }
    }
    if(saida == 0){
        adj.get(inic.indice).get(0).sfty = 2;
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Grafo g = new Grafo();
        String city1, city2;
        String teste = " ";
        int n;
        n = sc.nextInt();

        for(int i = 0; i < n; i++){
            city1 = sc.next();
            city2 = sc.next();
            g.novaAresta(city1, city2);
        }
        g.classificaGrafo(g.adj.get(0).get(0));

        while(!teste.equals(null)){
            teste = sc.next();
            g.mostraV(teste);
        }
    }
```

Exemplos rodados

```
5
Arlington San_Antonio
San_Antonio Baltimore
Baltimore New_York
New_York Dallas
Baltimore Arlington
San_Antonio
San_Antonio safe
Baltimore
Baltimore safe
New_York
New_York trapped
Arlington
Arlington safe
Dallas
Dallas trapped
```

03 Pick up sticks

Pick up sticks

Pega palitos é um jogo fascinante. Uma coleção de palitos coloridos são despejados em uma pilha emaranhada na mesa. Os jogadores se revezam tentando para pegar um único palito de cada vez sem mover nenhuma dos outros palitos. É muito difícil pegar um palito se houver outro palito deitado no topo. Os jogadores, portanto, tentam pegar os palitos em uma ordem que eles nunca tenham que pegar o palito de baixo.

Pick up sticks

Entrada: A entrada consiste em vários testes casos. A primeira linha de cada teste contém dois inteiros n e m cada um com pelo menos um e não mais que um milhão. O inteiro n é o número de palitos, e m é o número de linhas que se seguem. As varetas são numeradas de 1 a n. Cada uma das linhas a seguir contém um par de inteiros a, b, indicando que existe um ponto onde o palito a fica em cima do palito b. A última linha de entrada é '0 0'. Esses zeros não são valores de n e m, e devem não ser processado como tal.

Saída: Para cada caso de teste, imprima n linhas de números inteiros, listando os palitos na ordem em que podem ser pegou sem nunca pegar um palito com outro palito em cima. Se houver vários desses ordens corretas, qualquer um serve. Se não houver essa ordem correta, imprima uma única linha contendo opalavra ‘IMPOSSÍVEL’.

Sample Input

```
3  2
1  2
2  3
0  0
```

Sample Output

```
1
2
3
```

Pick up sticks

Estrutura de dados: Lista de adjacências

Vértices: Varetas

Arestas: Se a vareta i está acima da j, então temos uma aresta de i para j

Pick up sticks (versão 1)

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

#define NMAX 100010
#define fi first
#define se second
#define pb push_back
#define mp make_pair
#define WHITE 0 // Não visitado
#define GRAY  1 // Vizitando
#define BLACK 2 // Vizitado

vector< int > grafo[NMAX]; // Grafo por lista de adjacência
int color[NMAX];                    // Cor do i-esimo vértice

vector< int > resp;
```

```cpp
bool DFS(int u)
{

        // Estamos vizitando u
        color[u] = GRAY;

        for(auto v : grafo[u])
        {

                // Ignora se v já acabou
                if(color[v] == BLACK) continue;

                // Se v está sendo vizitado, encontramos um ciclo, assim
                // retornamos false
                if(color[v] == GRAY) return false;

                if(DFS(v) == false) return false;
```

```
        }

        // Acabamos de visitar u
        color[u] = BLACK;

        // Parte do algoritmo de ordenação topológica
        resp.pb(u);

        return true;

}
```

```cpp
int main()
{

        int n, m, u, v, i;

        while(cin >> n >> m)
        {

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

                resp.clear();
                for(i = 1;i <= n;i++) grafo[i].clear();
                for(i = 1;i <= n;i++) color[i] = WHITE;
                // Inicialmente todos os vértices estão não vizitados
```

```
while(m--)
{

        cin >> u >> v;

        grafo[u].pb(v);

}

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

        // Se já passamos por i, ignoramos ele
        if(color[i] != WHITE) continue;
```

```
        // Se já passamos por i, ignoramos ele
        if(color[i] != WHITE) continue;

        if(DFS(i) == false)
        {

                // Achamos um ciclo :(

                cout << "IMPOSSIBLE" << endl;

                resp.clear();

                break;

        }

}

// Parte do algoritmo de ordenação topológica
reverse(resp.begin(), resp.end());

for(auto cur : resp) cout << cur << endl;
```

Pick up sticks (versão 2)

```java
import java.util.ArrayList;

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

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

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

}
```

```java
public class Grafo {
    int n_vert; //numero de vertices
    Integer[] estado; // -1 nao visitado, 0 visitando, 1 visitado
    Stack<Integer> p;
    Lista[] adj; //vetor de listas de adjacencia

    public Grafo(int v){
        n_vert = v;
        adj = new Lista[v];
        estado = new Integer[v];
        p = new Stack<Integer>();

        for(int i = 0; i < v; i++){
            estado[i] = -1;
            adj[i] = new Lista();
            adj[i].lista.add(new Vertice(i));
        }
    }
```

```java
public boolean ordena(){
    boolean res = true;

    for(int i = 0; i < n_vert; i++){
        if(estado[i] == -1){
            res = dfs(i);
        }
        if(!res){
            break;
        }
    }
    return res;
}
```

```
public boolean dfs(int a){
    estado[a] = 0;
    boolean r = true;
    for(int i = 1; i < adj[a].lista.size(); i++){
        if(estado[adj[a].lista.get(i).indice] == -1){
            r = dfs(adj[a].lista.get(i).indice);
        }else if(estado[adj[a].lista.get(i).indice] == 0){
            r = false;
        }
    }
    estado[a] = 1;
    p.push(a);
    return r;
}
```

Exemplos rodados