Exercícios-Problemas (EP)

Semana 2:

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

EP1

A rare book collector recently discovered a book written in an unfamiliar language that used the same characters as the English language. The book contained a short index, but the ordering of the items in the index was different from what one would expect if the characters were ordered the same way as in the English alphabet. The collector tried to use the index to determine the ordering of characters (i.e., the collating sequence) of the strange alphabet, then gave up with frustration at the tedium of the task.

You are to write a program to complete the collector's work. In particular, your program will take a set of strings that has been sorted according to a particular collating sequence and determine what that sequence is.

Input

The input consists of an ordered list of strings of uppercase letters, one string per line. Each string contains at most 20 characters. The end of the list is signalled by a line with the single character '#'. Not all letters are necessarily used, but the list will imply a complete ordering among those letters that are used.

Output

Your output should be a single line containing uppercase letters in the order that specifies the collating sequence used to produce the input data file.

EP1

Sample Input

XWY

ZX

ZXY

ZXW

YWWX

#

Sample Output

XZYW

- Leitura das palavras
- Criação do grafo usando estrutura de dicionário

```
strings = []

while True:
    string = input()
    if string == "#":
        break
    strings.append(string)

# Cria um dicionário para representar o grafo de ordem de sequência
grafo_sequencia = {char: set() for string in strings for char in string}
```

```
{'X': set(), 'W': set(), 'Y': set(), 'Z': set()}
```

- Preenche o grafo
- Compara a string de cima com a de baixo
- Acha o primeiro caractere diferente entre as duas e adiciona no grafo
- Se caractere da string1 for diferente do caracter da string2, logo ele precede na ordem

```python
# Preenche o grafo de ordem de sequência com base nas strings de entrada
for i in range(len(strings) - 1):
    string1, string2 = strings[i], strings[i + 1]
    min_length = min(len(string1), len(string2))

    for j in range(min_length):
        if string1[j] != string2[j]:
            grafo_sequencia[string1[j]].add(string2[j])
            break
```

```
{'X': {'Z'}, 'W': set(), 'Y': {'W'}, 'Z': {'Y'}}
```

- Preenche o grafo
- Verifica as relações transitivas entre todos os vértices
- Adiciona os vértices v que podem ser alcançados a partir de u, levando em conta as relações transitivas

```
# Algoritmo de Warshall para completar o grafo de ordem de sequência
for k in grafo_sequencia:
    for u in grafo_sequencia:
        for v in grafo_sequencia:
            if v != k and v != u:
                if u in grafo_sequencia[k] and v in grafo_sequencia[u]:
                    grafo_sequencia[k].add(v)
```

```
{'X': {'Z', 'Y'}, 'W': set(), 'Y': {'W'}, 'Z': {'W', 'Y'}}
```

- Conta as arestas de entrada
- Percorre o dict contando os vértices que aparecem como “filhos”
- Ordena baseado na contagem e imprime a sequência

```python
# Conta as arestas de entrada em cada nó do grafo
areastas_entrada = {node: 0 for node in grafo_sequencia}

for node, neighbors in grafo_sequencia.items():
    for neighbor in neighbors:
        areastas_entrada[neighbor] += 1

# Ordena as chaves do dicionário com base nas contagens de arestas  de entrada
sequencia_ordenacao = ''.join(sorted(areastas_entrada, key=areastas_entrada.get))

# Imprime a sequência de ordem
print(sequencia_ordenacao)
```

```
{'X': 0, 'W': 2, 'Y': 2, 'Z': 1}
XZWY
```

EP2

The bad thing about being an international Man of Mystery (MoM) is there's usually someone who wants to kill you. Sometimes you have to stay on the run just to stay alive. You have to think ahead. You have to make sure you don't end up trapped somewhere with no escape.

Of course, not all MoMs are blessed with a great deal of intelligence. You are going to write a program to help them out. You are going to make sure our MoM knows what cities are safe to visit and which are not. It's not enough to just be able to run (or fly) for one or two days, we have to guarantee that the MoM can keep running for as long as might be necessary. Given a list of regular, daily flights between pairs of cities, you are going to make sure our MoM never gets stuck in a city from which there is no escape. We'll say there is an escape from some location if there is an infinitely long sequence of cities the MoM could fly to making one flight each day.

Input

Input starts with a number, 1<= n <= 5000, giving the number of daily flights there are between pairs of cities. Each of the next n lines contain a pair of city names separated by a space. Each city name is a string of up to 30 characters using only characters a–z, A–Z, and underscore. A line containing the name o followed by d indicates that there is a one-way flight from city o to city d every day. There are no flights that originate from and are destined for the same city.

The description of daily flights is followed by a list of up to 1000 city names that have previously been named, one per line. The list ends at end of file.

Output

Your job is to examine the list of city names at the end and determine whether or not there is an escape from each one. For each, output the name of the city, followed by the word "safe" if there is an escape and "trapped" if there is no escape.

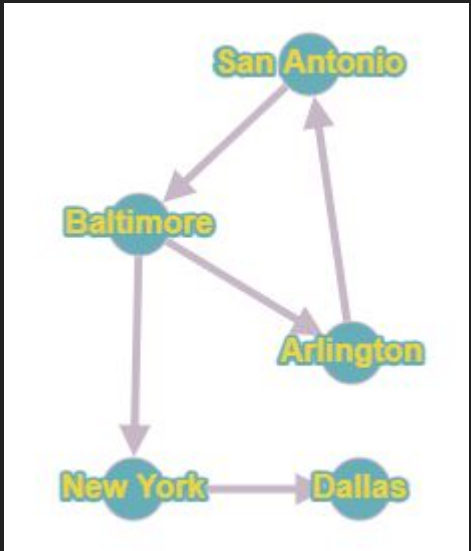
Sample Input

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

Sample Output

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

Exemplo de grafo:
Vertices: cidades
Arestas: voos



Para que homem nunca fique preso em algum lugar é necessário que ele esteja em uma cidade que está contida em um ciclo no grafo

Representação do grafo

```python
class Grafo:
    def __init__(self):
        self.adj = {}

    def add_vertice(self, v):
        if v not in self.adj:
            self.adj[v] = []

    def add_aresta(self, v1, v2):
        if v1 in self.adj and v2 in self.adj:
            self.adj[v1].append(v2)

    def __str__(self):
        result = ""
        for vertice, vizinhos in self.adj.items():
            result += f"{vertice}: {' '.join(vizinhos)}\n"
        return result
```

```
Arlington: San_Antonio
San_Antonio: Baltimore
Baltimore: New_York Arlington
New_York: Dallas
Dallas:
```

Código para verificar se um vértice faz parte de um ciclo

```python
def tem_ciclo(self, v):
    visitados = set()       # Nós visitados
    pilha_recursao = set()  # Nós da pilha de recursão

    def dfs(v):
        visitados.add(v)            # Marca o nó como visitado
        pilha_recursao.add(v)       # Adiciona o nó à pilha de recursão

        for vizinho in self.adj.get(v, []):  # Itera pelos vizinhos do nó atual
            if vizinho not in visitados:     # Se o vizinho ainda não foi visitado
                if dfs(vizinho):             # Chama recursivamente a DFS no vizinho, se for encontrado um ciclo, retorna True
                    return True
            elif vizinho in pilha_recursao:   # Se o vizinho já está na pilha de recursão indica a detecção de um ciclo
                return True

        pilha_recursao.remove(v)   # Remove o nó da pilha de recursão ao retroceder
        return False               # Retorna False, indicando que nenhum ciclo foi encontrado

    return dfs(v)  # Inicia a busca em profundidade a partir do nó 'v' e retorna o resultado
```

Código para receber os inputs das cidades, criar o grafo e retornar as mensagens após verificação

```python
def main():
    num = int(input()) # Número de voos
    g = Grafo()

    for _ in range(num):
        cidade1, cidade2 = input().split() # Par de cidades
        # Adicionar as cidades como vértices e cria aresta entre elas
        g.add_vertice(cidade1)
        g.add_vertice(cidade2)
        g.add_aresta(cidade1, cidade2)

    cidades_verificar = []
    while True:
        cidade = input() # Cidades para verificação
        if not cidade:
            break
        cidades_verificar.append(cidade)

    for cidade in cidades_verificar:
        if g.tem_ciclo(cidade):
            print(f"{cidade} safe")
        else:
            print(f"{cidade} trapped")
```

EP3

Pick up sticks is a fascinating game. A collection of coloured sticks are dumped in a tangled heap on the table. Players take turns trying to pick up a single stick at a time without moving any of the other sticks. It is very difficult to pick up a stick if there is another stick lying on top of it. The players therefore try to pick up the sticks in an order such that they never have to pick up a stick from underneath another stick.

Input

The input consists of several test cases. The first line of each test case contains two integers n and m each at least one and no greater than one million. The integer n is the number of sticks, and m is the number of lines that follow. The sticks are numbered from 1 to n. Each of the following lines contains a pair of integers a, b, indicating that there is a point where stick a lies on top of stick b. The last line of input is ‘0 0’. These zeros are not values of n and m, and should not be processed as such.

Output

For each test case, output n lines of integers, listing the sticks in the order in which they could be picked up without ever picking up a stick with another stick on top of it. If there are multiple such correct orders, any one will do. If there is no such correct order, output a single line containing the word ‘IMPOSSIBLE’.

Sample Input

3 2

1 2

2 3

0 0

Sample Output

1

2

3

- Lê n e m
- Lê as linhas
- Cria o grafo adicionando uma aresta direcionada de a para b, indicando que o a está tocando por cima de b

```python
n, m = map(int, input().split())  # Lê o número de palitos e linhas

if n == 0 and m == 0:
    break  # Encerra o loop se n e m forem ambos 0

grafo = {i: [] for i in range(1, n+1)}  # Inicializa um grafo vazio

for _ in range(m):
    a, b = map(int, input().split())  # Lê um par de inteiros
    grafo[a].append(b)  # Adiciona uma aresta direcionada de a para b
```

```
{1: [2], 2: [3], 3: []}
```

```python
def ordenacao_topologica(grafo):
    grau_entrada = [0] * (len(grafo) + 1)
    ordem = []  # Inicializa a lista de ordem

    # Calcula os graus de entrada de todos os nós
    for u in grafo:
        for v in grafo[u]:
            grau_entrada[v] += 1
```

- Ordenação topológica
- Cria uma lista de zeros que irá armazenar os graus de entrada de cada nó do grafo
- Percorre cada nó u do grafo e conta quantas arestas estão apontando para o nó v

```
[0, 0, 1, 1]
```

```python
# Inicializa uma fila com nós que possuem grau de entrada 0
fila = [u for u in grafo if grau_entrada[u] == 0]

while fila:
    u = fila.pop(0)
    ordem.append(u)

    for v in grafo[u]:
        grau_entrada[v] -= 1
        if grau_entrada[v] == 0:
            fila.append(v)

return ordem
```

- Inicia uma fila com os nós que têm grau de entrada = 0
- Caso tenha grau 0 sai da fila e entra na lista de ordem
- Percorre os outros nós decrementando em 1 o grau de entrada e verificando se é 0, caso for 0 insere na fila
- Repete isso até esvaziar a fila
- Retorna a ordem

```
[1, 2, 3]
```

- Compara o tamanho da lista retornada da ordem topológica com o número de palitos
- Caso for igual, então é possível retirar em ordem, sem ter que pegar um palito que esteja embaixo de outro
- Caso não for igual, retorna a string “IMPOSSIBLE”

```
if len(ordem) == n:
    # Se for possível ordenar todos os palitos sem levantar outros palitos em cima deles, imprime a ordem
    for palito in ordem:
        print(palito)
else:
    print("IMPOSSIBLE")
```

```
1
2
3
```