Exercícios-Problemas (EP)

Semana 3:

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

EP1: UVa 00315 - Network

Uma empresa está estabelecendo uma rede de cabos telefônicos, eles conectam diversos lugares que são numerados de 1 a N. De qualquer lugar é possível se encontrar outro lugar, através destas linhas. Porém eles perceberam que em caso de falta de energia em alguns pontos, além do ponto sem energia, outros pontos também ficam inalcançáveis, estes pontos foram chamados de pontos críticos.

Analisando o enunciado, percebe-se que estes pontos críticos se tratam de pontos de articulação.

EP1: UVa 00315 - Network

O arquivo de entrada, consiste em:

1. Um numero N de lugares (0 para)
2. Até N linhas depois. Indicando os adjacentes de cada lugar (0 para)

A saída será:

- Um inteiro que representa a quantidade de pontos críticos em cada grafo

Entrada:

```
Quantidade de pontos: 5
Pontos: 5 1 2 3 4
Pontos: 0
Quantidade de pontos: 6
Pontos: 2 1 3
Pontos: 5 4 6 2
Pontos: 0
Quantidade de pontos: 0
```

Saida:

```
1
2
```

EP1: UVa 00315 - Network

Codigo:

```python
def pontos_de_articulacao(self):
    pontos_articulacao = []

    for vertice in self.adj:
        atual = self.componentes_conexas()
        #Cria adjacencia temporaria, removendo um vertice
        temp_adj = {v: [viz for viz in self.adj[v] if viz != vertice] for v in self.adj if v != vertice}
        temp_grafo = Grafo()
        temp_grafo.adj = temp_adj

       #Verifica a quantidade de componentes sem o vertice
        num_componentes_sem_vertice = temp_grafo.componentes_conexas()

        #Se aumentar a quantidade se trata de um ponto de articulacao
        if num_componentes_sem_vertice > atual:
            pontos_articulacao.append(vertice)

    return pontos_articulacao
```

EP1: UVa 00315 - Network

Codigo:

```
def main2():
  grafos = []

  while True:
    grafo = Grafo()
    n_pontos = int (input("Quantidade de pontos: "))
    if n_pontos == 0:
      break

    for i in range(n_pontos):
      grafo.add_vertice(i+1)

    for i in range(n_pontos):
      adj = []
      entrada = list(map(int, input("Pontos: ").split()))
      if entrada[0] == 0:
        break
      vertice = entrada[0]
      adj.extend(entrada[1:])

      for ponto in adj:
        grafo.add_aresta(vertice, ponto)

    grafos.append(grafo)

  for grafo in grafos:
    pontos_articulacao = grafo.pontos_de_articulacao()
    print(len(pontos_articulacao))
```

EP1: UVa 00315 - Network

Codigo:

```python
class Grafo:
    def __init__(self):
        self.adj = {}  # Dicionário para armazenar listas de adjacência

    def add_vertice(self, v):
        if v not in self.adj:
            self.adj[v] = []  # Adiciona um novo vértice com uma lista vazia de vizinhos

    def add_aresta(self, v1, v2):
        if v1 in self.adj and v2 in self.adj:
            self.adj[v1].append(v2)  # Adiciona v2 à lista de vizinhos de v1
            self.adj[v2].append(v1)  # Adiciona v1 à lista de vizinhos de v2

    def dfs(self, v, visitados):
        visitados.add(v)
        for vizinho in self.adj[v]:
            if vizinho not in visitados:
                self.dfs(vizinho, visitados)

    def componentes_conexas(self):
        visitados = set()
        num_componentes = 0
        for vertice in self.adj: #Visita os vértices pela lista de adj
            if vertice not in visitados: #Se um vértice não foi visitado, aumenta a quantidade de componentes
                num_componentes += 1
                self.dfs(vertice, visitados) #Percorre a partir deste novo vértice
        return num_componentes
```

EP3: UVa 12363 - Hedge Mazes

The Queen of Nlogonia is a fan of mazes, and therefore the queendom's architects built several mazes around the Queen's palace. Every maze built for the Queen is made of rooms connected by corridors. Each corridor connects a different pair of distinct rooms and can be transversed in both directions. The Queen loves to stroll through a maze's rooms and corridors in the late afternoon. Her servants choose a different challenge for every day, that consists of finding a simple path from a start room to an end room in a maze. A simple path is a sequence of distinct rooms such that each pair of consecutive rooms in the sequence is connected by a corridor. In this case the first room of the sequence must be the start room, and the last room of the sequence must be the end room. The Queen thinks that a challenge is good when, among the routes from the start room to the end room, exactly one of them is a simple path. Can you help the Queen's servants to choose a challenge that pleases the Queen? For doing so, write a program that given the description of a maze and a list of queries defining the start and end rooms, determines for each query whether that choice of rooms is a good challenge or not.

Input

Each test case is described using several lines. The first line contains three integers R, C and Q representing respectively the number of rooms in a maze ($2 \leq R \leq 104$), the number of corridors ($1 \leq C \leq 105$), and the number of queries ($1 \leq Q \leq 1000$). Rooms are identified by different integers from 1 to R. Each of the next C lines describes a corridor using two distinct integers A and B, indicating that there is a corridor connecting rooms A and B ($1 \leq A < B \leq R$). After that, each of the next Q lines describes a query using two distinct integers S and T indicating respectively the start and end rooms of a challenge ($1 \leq S < T \leq R$). You may assume that within each test case there is at most one corridor connecting each pair of rooms, and no two queries are the same. The last test case is followed by a line containing three zeros.

Output

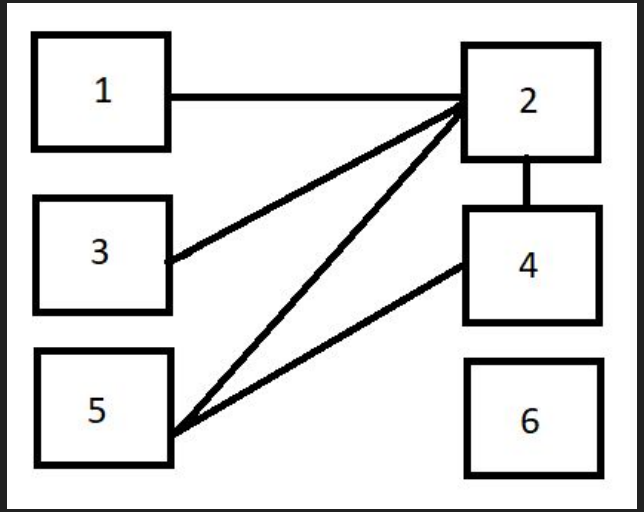
For each test case output Q + 1 lines. In the i-th line write the answer to the i-th query. If the rooms make a good challenge, then write the character 'Y' (uppercase). Otherwise write the character 'N' (uppercase). Print a line containing a single character '-' (hyphen) after each test case.

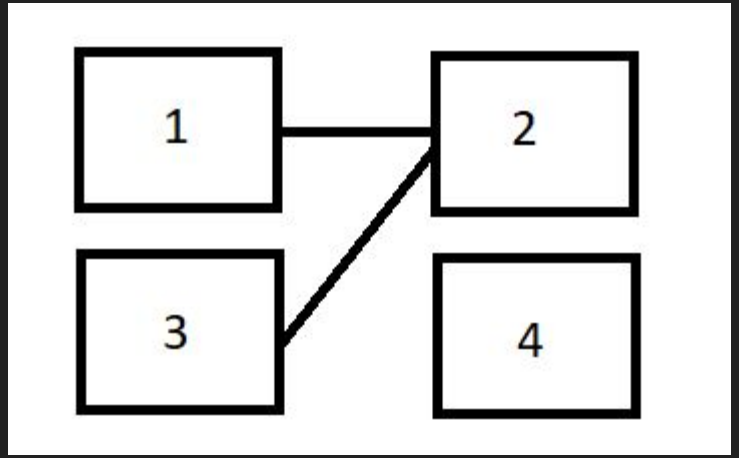
EP3

Sample Input

```
6 5 3
1 2
2 3
2 4
2 5
4 5
1 3
1 5
2 6

4 2 3
1 2
2 3
1 4
1 3
1 2
```





Sample Output

```
Y
N
N
-
N
Y
Y
-
```

- Leitura das estruturas
- Quartos (nós/vértices) e queries (pesquisa dos caminhos)
- Trata finalização do programa

```python
def main():
    while True:
        R, C, Q = map(int, input().split()) # Leitura da quantidade de quartos, corredores e pesquisas
        if R == 0 and C == 0 and Q == 0: # Finaliza o programa
            break

        graph = set_graph(R, C) # Inicializa o grafo com os quartos (nós) e lê os corredores (arestas)

        queries = []
        for _ in range(Q):
            S, T = map(int, input().split()) # Leitura das pesquisas de caminhos
            queries.append((S, T))
```

- Leitura dos corredores (arestas)
- Preenche o grafo

```python
def set_graph(R, C): # Cria o grafo com as informações da quantidade de quartos e dos corredores
    graph = [[] for _ in range(R + 1)]
    for _ in range(C):
        A, B = map(int, input().split()) # Leitura dos corredores
        graph[A].append(B)
        graph[B].append(A)
    return graph
```

- Utiliza a dfs para contar quantos caminhos existem entre 2 quartos (vértices do grafo)
- Incrementa recursivamente a variável total_path que conta os caminhos possíveis entre 2 vértices
- Backtracking permite explorar todas as soluções possíveis, testando recursivamente diferentes caminhos

```python
def check_simple_path(graph, queries):
    def dfs(start, end, visited):
        if start == end: # Encontra um caminho e retorna 1 para o count
            return 1
        visited[start] = True
        total_paths = 0  # Inicializa a variável para contar os caminhos
        for neighbor in graph[start]:
            if not visited[neighbor]:
                total_paths += dfs(neighbor, end, visited) # Chamada recursiva da função incrementando o número de caminhos
        visited[start] = False  # Backtrack, permite ser revisitado no futuro enquanto estiver explorando novos caminhos
        return total_paths
```

- Chamada da função para checar se a query é caminho simples
- Se o número de caminhos para aquela query for 1 significa que é um caminho simples e printa ‘Y’
- Se for maior que 1, existe mais de 1 caminho possível, não é caminho simples, printa ‘N’
- Se for 0, não existe caminho possível, printa ‘N’

```
results = check_simple_path(graph, queries)
for result in results:
    print(result)
print('-')
```

```
results = []
for S, T in queries:
    visited = [False] * len(graph) # Inicializa array de visitados
    paths = dfs(S, T, visited)
    if paths == 1: # Simple path
        results.append('Y')
    else: # Ou não tem caminho possível ou tem mais de um caminho
        results.append('N')
return results
```

Ep 10 - Tourist Guide

O problema se trata de encontrar para Bruno quais cidades terão câmeras da polícia. Para saber onde essas câmeras estão o problema descreve da seguinte maneira: Se existem 2 cidades A e B, e para se chegar de A para B, ou de B para A, for necessário passar por C, C terá uma câmera.

Com base neste enunciado, o grupo chegou a conclusão que basta modelar o grafo da seguinte forma:

- Vertices: Cidades
- Arestas: Rotas

Ep 10 - Tourist Guide

Os dados de entrada são organizados da seguinte forma:

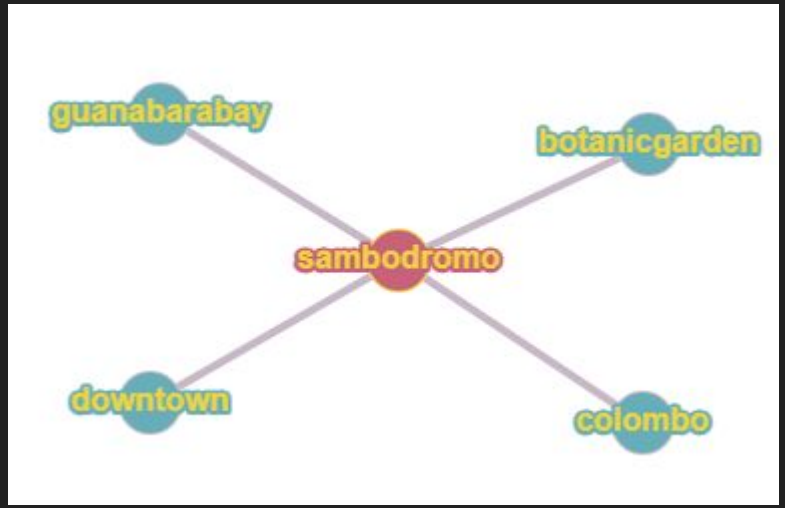
1. N, que será a quantidade de cidades (0 para sair)
2. N cidades
3. M, que será a quantidade de rotas
4. E as M rotas

```
5
guanabarabay
downtown
botanicgarden
colombo
sambodromo
4
guanabarabay sambodromo
downtown sambodromo
sambodromo botanicgarden
colombo sambodromo
0
```

Ep 10 - Tourist Guide

Modelando o grafo deste exemplo:

- Para se ir de downton para colombo é OBRIGATÓRIO passar pelo sambodromo, logo lá terá uma camera



Ep 10 - Tourist Guide

Codigo:

```python
class Grafo:
    def __init__(self):
        self.adj = {}  # Dicionário para armazenar listas de adjacência

    def add_vertice(self, v):
        if v not in self.adj:
            self.adj[v] = []  # Adiciona um novo vértice com uma lista vazia de vizinhos

    def add_aresta(self, v1, v2):
        if v1 in self.adj and v2 in self.adj:
            self.adj[v1].append(v2)  # Adiciona v2 à lista de vizinhos de v1
            self.adj[v2].append(v1)  # Adiciona v1 à lista de vizinhos de v2

    def dfs(self, v, visitados):
        visitados.add(v)
        for vizinho in self.adj[v]:
            if vizinho not in visitados:
                self.dfs(vizinho, visitados)

    def componentes_conexas(self):
        visitados = set()
        num_componentes = 0
        for vertice in self.adj: #Visita os vértices pela lista de adj
            if vertice not in visitados: #Se um vértice não foi visitado, aumenta a quantidade de componentes
                num_componentes += 1
                self.dfs(vertice, visitados) #Percorre a partir deste novo vértice
        return num_componentes
```

Ep 10 - Tourist Guide

Codigo:

```python
def pontos_de_articulacao(self):
    pontos_articulacao = []

    for vertice in self.adj:
        atual = self.componentes_conexas()
        #Cria adjacencia temporaria, removendo um vertice
        temp_adj = {v: [viz for viz in self.adj[v] if viz != vertice] for v in self.adj if v != vertice}
        temp_grafo = Grafo()
        temp_grafo.adj = temp_adj

       #Verifica a quantidade de componentes sem o vertice
        num_componentes_sem_vertice = temp_grafo.componentes_conexas()

        #Se aumentar a quantidade se trata de um ponto de articulacao
        if num_componentes_sem_vertice > atual:
            pontos_articulacao.append(vertice)

    return pontos_articulacao
```

Ep 10 - Tourist Guide

Codigo:

```python
def main():
    grafos = []

    while True:
        n_cidades = int(input("Quantidade de cidades (0 para sair): "))

        if n_cidades == 0:
            break

        grafo = Grafo()

        for i in range(n_cidades):
            cidade = input(f"Cidade {i+1}: ")
            grafo.add_vertice(cidade)

        n_rotas = int(input("Quantidade de rotas: "))
        for _ in range(n_rotas):
            cidade1, cidade2 = input("Digite os nomes das cidades ligadas por uma rota: ").split()
            grafo.add_aresta(cidade1, cidade2)

        grafos.append(grafo)

    for idx, grafo in enumerate(grafos):
        pontos_articulacao = grafo.pontos_de_articulacao()
        print(f"City Map #{idx + 1}:")

        print(f"{len(pontos_articulacao)} camera(s) found")

        for cidade in pontos_articulacao:
            print(f"{cidade }")
```

Execução:

```
Quantidade de cidades (0 para sair): 5
Cidade 1: guanabaray
Cidade 2: downtown
Cidade 3: sambodromo
Cidade 4: botanicgarden
Cidade 5: colombo
Quantidade de rotas: 4
Digite os nomes das cidades ligadas por uma rota: guanabaray sambodromo
Digite os nomes das cidades ligadas por uma rota: downtown sambodromo
Digite os nomes das cidades ligadas por uma rota: sambodromo botanicgarden
Digite os nomes das cidades ligadas por uma rota: sambodromo colombo
Quantidade de cidades (0 para sair): 0
City Map #1:
1 camera(s) found
sambodromo
```