

import sys

sys.path.insert(0, 'networkx.zip')

import networkx as nx

# Tworzymy graf zgodnie z rysunkiem

G = nx.Graph()

G.add\_edges\_from([

(0, 7),

(1, 2), (1, 5), (1, 4),

(2, 3),

(3, 4), (3, 5), (3, 6),

(4, 6),

(5, 6),

(8, 9), (8, 10), (9, 10), (10, 11)

])

# Dodajemy wszystkie wierzchołki od 0 do 13 (nawet odizolowane)

G.add\_nodes\_from(range(14))

# Wczytanie podzbioru U jako lista liczb całkowitych

U = list(map(int, input().split()))

# Podgraf indukowany

H = G.subgraph(U)

# Wypisanie posortowanej listy wierzchołków

vertices = sorted(H.nodes())

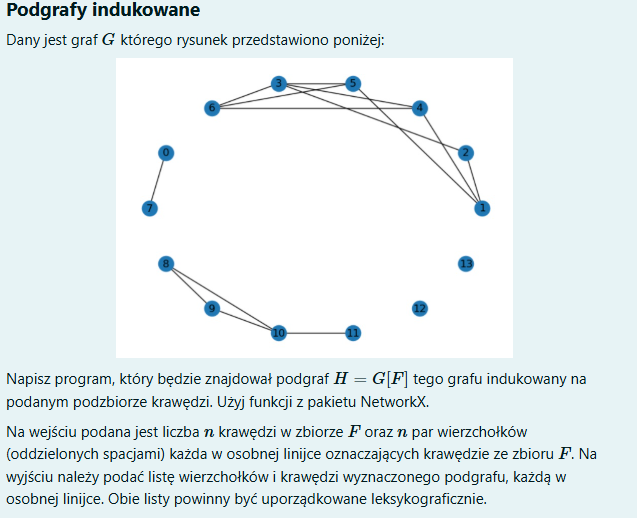
# Wypisanie posortowanej listy krawędzi w postaci (mniejszy, większy)

edges = sorted((min(u, v), max(u, v)) for u, v in H.edges())

# Wynik

print(vertices)

print(edges)



import sys

sys.path.insert(0, 'networkx.zip')

import networkx as nx

# Tworzymy graf zgodny z rysunkiem

G = nx.Graph()

G.add\_edges\_from([

(0, 7),

(1, 2), (1, 3), (1, 4), (1, 5), (1, 6),

(2, 3),

(3, 4), (3, 5),

(4, 6),

(5, 6),

(8, 9), (9, 10), (10, 11), (11, 8)

])

G.add\_nodes\_from(range(14)) # dodajemy brakujące wierzchołki

# Wczytanie liczby krawędzi

n = int(input())

# Wczytanie krawędzi jako lista krotek

F = []

for \_ in range(n):

u, v = map(int, input().split())

F.append((u, v))

# Budujemy nowy graf zawierający tylko te krawędzie

H = nx.Graph()

H.add\_edges\_from(F)

# Posortowane wierzchołki i krawędzie

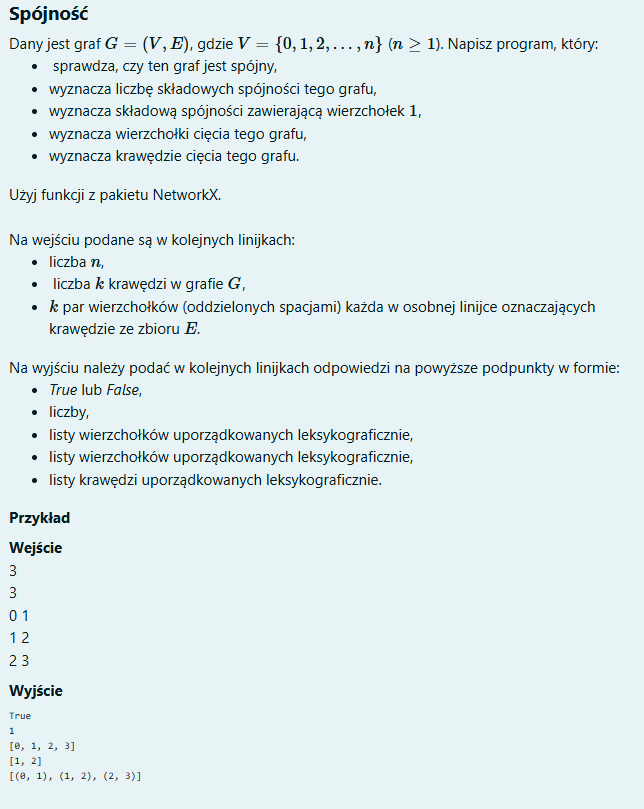
vertices = sorted(H.nodes())

edges = sorted((min(u, v), max(u, v)) for u, v in H.edges())

# Wynik

print(vertices)

print(edges)



import sys

sys.path.insert(0, 'networkx.zip')

import networkx as nx

# Wczytanie danych

n = int(input()) # liczba wierzchołków

m = int(input()) # liczba krawędzi

edges = [tuple(map(int, input().split())) for \_ in range(m)]

# Tworzenie grafu

G = nx.Graph()

G.add\_nodes\_from(range(n))

G.add\_edges\_from(edges)

# 1. Czy graf jest spójny?

is\_connected = nx.is\_connected(G)

# 2. Liczba spójnych składowych

num\_components = nx.number\_connected\_components(G)

# 3. Składowa zawierająca wierzchołek 1

component\_with\_1 = sorted(c for c in nx.node\_connected\_component(G, 1))

# 4. Wierzchołki artykulacji (cut vertices)

cut\_vertices = sorted(nx.articulation\_points(G))

# 5. Krawędzie artykulacji (mosty)

cut\_edges = sorted((min(u, v), max(u, v)) for u, v in nx.bridges(G))

# Wynik

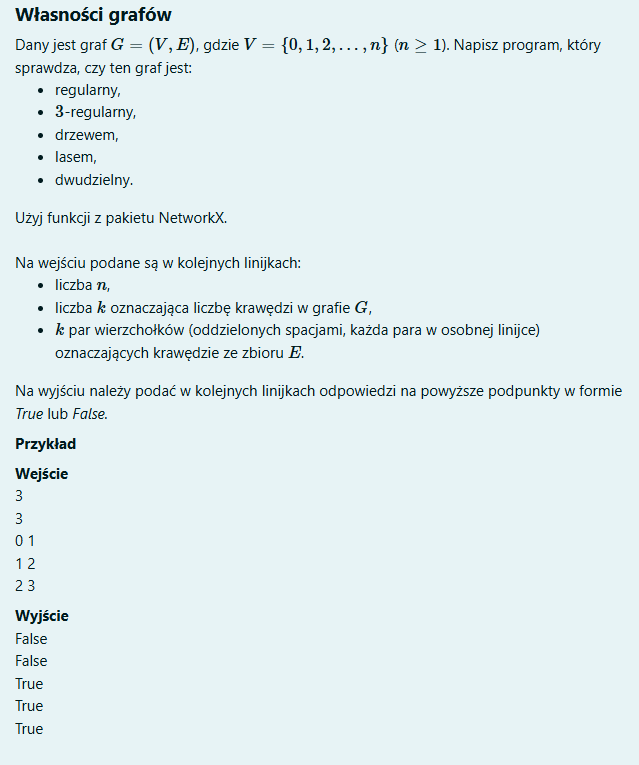
print(is\_connected)

print(num\_components)

print(component\_with\_1)

print(cut\_vertices)

print(cut\_edges)



import sys

sys.path.insert(0, 'networkx.zip')

import sys

import networkx as nx

def main():

# Wczytywanie danych wejściowych

lines = [line.strip() for line in sys.stdin if line.strip()]

n = int(lines[0]) # liczba wierzchołków

k = int(lines[1]) # liczba krawędzi

edges = [tuple(map(int, line.split())) for line in lines[2:]]

G = nx.Graph()

G.add\_nodes\_from(range(n))

G.add\_edges\_from(edges)

degrees = [deg for \_, deg in G.degree()]

# Czy graf jest regularny

is\_regular = all(deg == degrees[0] for deg in degrees)

# Czy graf jest 3-regularny

is\_3\_regular = all(deg == 3 for deg in degrees)

# Czy graf jest drzewem (spójny i bez cykli)

is\_tree = nx.is\_tree(G)

# Czy graf jest lasem (może być niespójny, ale bez cykli)

is\_forest = nx.is\_forest(G)

# Czy graf jest dwudzielny

is\_bipartite = nx.is\_bipartite(G)

print(is\_regular)

print(is\_3\_regular)

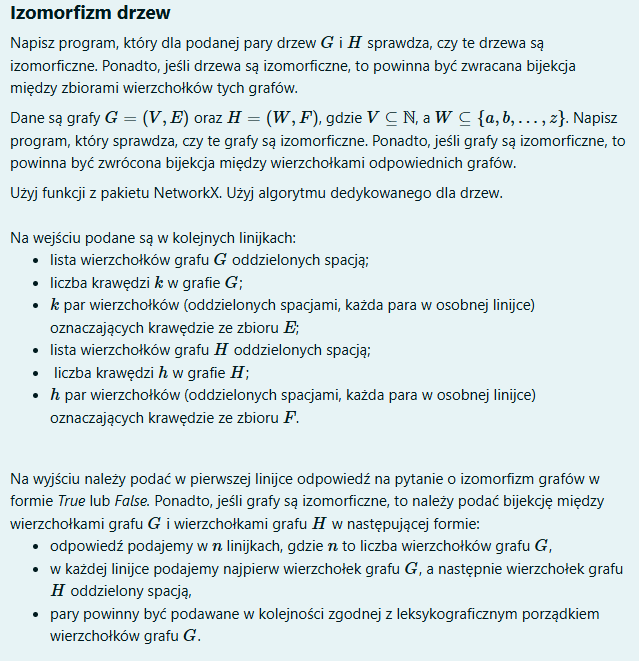
print(is\_tree)

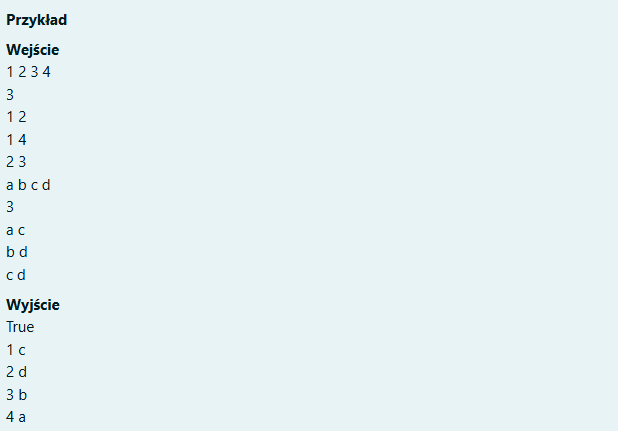
print(is\_forest)

print(is\_bipartite)

if \_\_name\_\_ == "\_\_main\_\_":

main()





import sys

sys.path.insert(0, 'networkx.zip')

import sys

import networkx as nx

from networkx.algorithms.isomorphism import GraphMatcher

def read\_graph(lines, offset):

nodes = lines[offset].split()

edge\_count = int(lines[offset + 1])

edges = [tuple(lines[offset + 2 + i].split()) for i in range(edge\_count)]

G = nx.Graph()

G.add\_nodes\_from(nodes)

G.add\_edges\_from(edges)

return G, nodes

def main():

lines = [line.strip() for line in sys.stdin if line.strip()]

# Wczytaj graf G

G, nodes\_G = read\_graph(lines, 0)

# Przesunięcie o linie G: 1 na wierzchołki, 1 na liczbę krawędzi, k linii krawędzi

k = int(lines[1])

H\_offset = 2 + k

H, nodes\_H = read\_graph(lines, H\_offset)

# Sprawdzenie izomorfizmu

GM = GraphMatcher(G, H)

if GM.is\_isomorphic():

print("True")

mapping = GM.mapping # dict: node\_G -> node\_H

result = sorted((g, mapping[g]) for g in sorted(nodes\_G))

for pair in result:

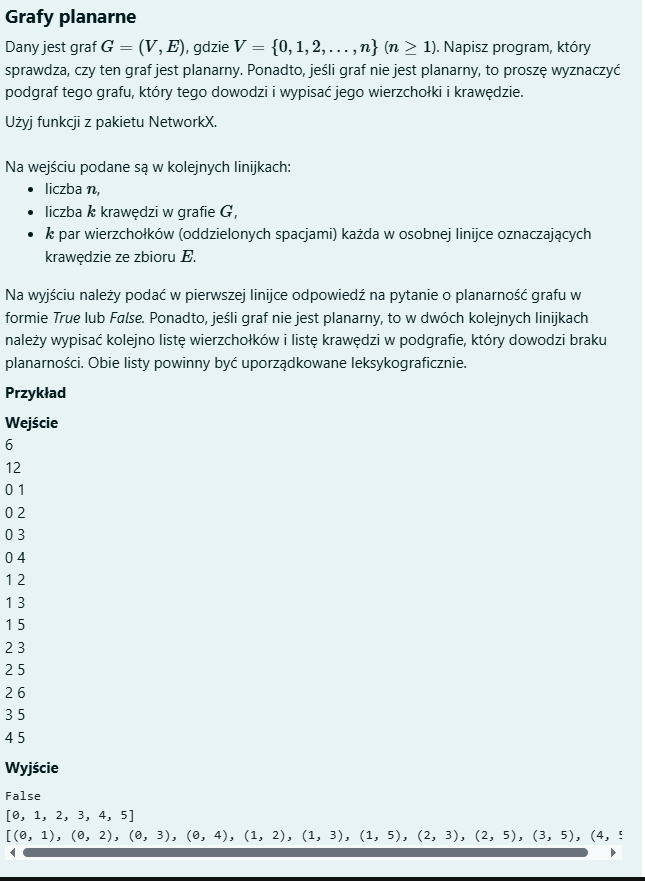
print(f"{pair[0]} {pair[1]}")

else:

print("False")

if \_\_name\_\_ == "\_\_main\_\_":

main()



import sys

sys.path.insert(0, 'networkx.zip')

import networkx as nx

def main():

import sys

input\_lines = [line.strip() for line in sys.stdin if line.strip()]

n = int(input\_lines[0])

k = int(input\_lines[1])

edges = [tuple(map(int, line.split())) for line in input\_lines[2:2 + k]]

G = nx.Graph()

G.add\_nodes\_from(range(n))

G.add\_edges\_from(edges)

is\_planar, counter\_example = nx.check\_planarity(G, counterexample=True)

print(is\_planar)

if not is\_planar:

subgraph = counter\_example

# Wierzchołki i krawędzie posortowane leksykograficznie

nodes = sorted(subgraph.nodes())

edges = sorted((min(u, v), max(u, v)) for u, v in subgraph.edges())

print(nodes)

print(edges)

if \_\_name\_\_ == "\_\_main\_\_":

main()