Задача A. Компоненты связности

Заметим, что один запуск поиска в глубину из любой вершины какой-либо компоненты связности посетит все вершины этой компоненты и никакие больше. Давайте в качестве `used` сохранять не просто `true или false`, а номер компоненты связности или 0, если вершина еще не посещена. Будем запускать `dfs` из всех вершин графа, которые мы не посетили в предыдущих запусках поиска в глубину. При каждом запуске `dfs` будем запоминать вершины, которые мы посетили. Таким образом после всех запусков мы получим номера вершин, которые есть в каждой из компонент связности.

Асимптотика решения равна асимптотике поиска в глубину — $O(n+m)$.

```
import sys
sys.setrecursionlimit(1000000000)

def dfs(v, c):
    used[v] = c
    comps.append(v+1)
    for u in g[v]:
        if used[u] == 0:
            dfs(u, c)

n, m = map(int, input().split())
g = [[] for i in range(n)]
for i in range(m):
    fr, to = map(int, input().split())
    g[fr-1].append(to-1)
    g[to-1].append(fr-1)

used = [0 for _ in range(n)]

c = 1
res = []
for i in range(n):
    if used[i] == 0:
        comps = []
        dfs(i, c)
        res.append(sorted(comps))
        c += 1

print(len(res))
for i in res:
    print(len(i))
    print(*i)
```

Задача B. Есть ли цикл?

Рассмотрим работу `dfs` при обходе ориентированного графа с циклом. Если в графе есть цикл, то при обходе графа найдется вершина, которая есть где-то выше в стеке рекурсии, и при этом в нее есть ребро из какой-то другой вершины в этом же обходе графа. Чтобы найти такие вершины, будем красить вершины в три цвета: 0 — вершина еще не посещена, 1 — `dfs` зашел в это вершину, но еще не обработал все ее ребра, 2 — вершина полностью обработана. Если при обходе графа мы найдем ребро в вершину, которая покрашена в цвет 1, то мы из этой вершины вышли и пришли к ней же, а значит в графе есть цикл. Такое решение работает за $O(n+m)$.

```
def has_cycle(graph):
    def dfs(node, color):
        color[node] = 1
        for neighbor in graph[node]:
```

```
            if color[neighbor] == 0:
                if dfs(neighbor, color):
                    return True
            elif color[neighbor] == 1:
                return True
        color[node] = 2
        return False

    n = len(graph)
    color = bytearray(n)
    for i in range(n):
        if color[i] == 0:
            if dfs(i, color):
                return 1
    return 0

n, m = map(int, input().split())
graph = [set() for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    graph[u - 1].add(v - 1)

print(has_cycle(graph))
```

Задача C. Один голодный конь

Представим клетки шахматной доски в виде вершин неориентированного графа. Проведем ребра между двумя вершинам a и b, если из a можно попасть в b за один ход шахматного коня. Заметим, что построение такого графа происходит за $O(n^2)$.

Дальше можно запустить поиск в ширину из стартовой вершины и найти минимальные количество ходов коня до каждой из клеток доски. Таким образом мы найдем наименьшее число ходов коня, чтобы добраться до (x_2, y_2). Чтобы вывести путь, необходимо запоминать позиции, из которых мы попали в каждую их клеток. Время работы такого решения — $O(n^2)$.

```
from collections import deque

n = int(input())
start = tuple(map(int, input().split()))
finish = tuple(map(int, input().split()))

def is_in(cords):
    return 0 < cords[0] <= n and 0 < cords[1] <= n

p = dict()
p[start] = (-1, -1)
shifts = ((1, 2), (-1, 2), (1, -2), (-1, -2),
        (2, 1), (2, -1), (-2, 1), (-2, -1))

q = deque()
q.append(start)
while q:
    cords = q.popleft()
    for shift_x, shift_y in shifts:
        ncords = cords[0] + shift_x, cords[1] + shift_y
        if is_in(ncords) and ncords not in p:
```

```
            p[ncords] = cords
            q.append(ncords)
            if ncords == finish:
                q = False
                break
way = [f'{finish[0]} {finish[1]}']
cords = finish
while p[cords][0] != -1:
    way.append(f'{p[cords][0]} {p[cords][1]}')
    cords = p[cords]
print(len(way) - 1)
print('\n'.join(reversed(way)))
```

Задача D. Теория чисел

Будем рассматривать в этой задаче не числа, а остатки от деления этих чисел на K. Тогда ответ на задачу — минимальное количество цифр в десятичной записи числа с остатком 0.

Пусть существует число x с остатком от деления i и суммой цифр c. Тогда

1. Можно умножить x на 10, и получить число с остатком $10i \mod K$ и суммой цифр c.
2. Можно прибавить к x единицу, и получить число с остатком $(i+1) \mod K$ и суммой цифр *не больше*, чем $c+1$.

Представим все возможные остатки от деления на K в виде ориентированного графа с K вершинами, проведем в нем ребра по правилам выше с весами 0 и 1 (1, если операция увеличивает сумму цифр, и 0 иначе).

Будем считать расстоянием в полученном графе то, на сколько увеличится сумма цифр в числе. В качестве начальной точки будем использовать число с остатком 1. Минимальная сумма цифр для такого числа — 1. Тогда минимальной суммой цифр для чисел с остатком 0 будет минимальное расстояние от чисел с остатком 1 до чисел с остатком 0, увеличенное на 1. Для нахождения это значения, можно воспользоваться поиском в ширину.

В этом решении мы используем поиск в ширину в графе с весами ребер 0 и 1, однако не сложно показать, что каждая вершина попадет в очередь не больше двух раз, и асимптотика такого решения — $O(n+m)$.

```
import math
from collections import deque


def minSum(k):
    for i in range(1, k + 1):
        g[i % k].append([(i + 1) % k, 1])
    for i in range(1, k + 1):
        g[i % k].append([(10 * i) % k, 0])
    q = deque()
    dist = [math.inf for _ in range(k + 2)]
    q.append([0, 1])
    dist[1] = 0
    while q:
        u = q.popleft()[1]
        for v, w in g[u]:
            if dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
                q.append([dist[v], v])
    return 1 + dist[0]
```

```
g = [[] for _ in range(10 ** 5 + 1)]
k = int(input())
print(minSum(k))
```

Задача E. Кратчайший путь

Поскольку граф неориентированный, то можно найти такой порядок обработки вершин, что минимальное расстояние от s до произвольной вершины v можно будет находить как минимум по всем входящим ребрам суммы расстояния до вершины, из которой исходит ребро к v, и веса ребра. То есть необходимо, чтобы до обработки вершины v, были посчитаны минимальные расстояния до всех вершин до v.

Такой порядок вершин называется топологической сортировкой вершин. Топологическую сортировку можно построить при помощи поиска в глубину. Будем запоминать порядок выхода из вершин в `dfs`. В полученном порядке, если есть ребро из a в b, то b стоит раньше a. Для дальнейшего пересчета необходим порядок, обратный полученному.

Временная сложность такого решения — $O(n + m)$.

```
from collections import deque
import sys
sys.setrecursionlimit(10**8)

def main():
    n, m, s, t = map(int, input().split())
    g = [[] for i in range(n + 1)]
    for i in range(m):
        b, e, w = map(int, input().split())
        g[b].append((e, w))

    used = [False] * (n + 1)
    out_order = []
    def dfs(v):
        used[v] = True
        for u, _ in g[v]:
            if not used[u]:
                dfs(u)
        out_order.append(v)
    dfs(s)

    dist = [None] * (n + 1)
    dist[s] = 0
    for v in out_order[::-1]:
        for (e, w) in g[v]:
            if dist[e] is None or dist[v] + w < dist[e]:
                dist[e] = dist[v] + w
    if dist[t] is None:
        print("Unreachable")
    else:
        print(dist[t])

if __name__ == '__main__':
    main()
```