Графы, DFS

1 A. От списка ребер к матрице смежности, ориентированный вариант

Разбор

Считаем список рёбер и для каждого ребра (u, v) выполним присвоение $m_{u,v} := 1$
Асимптотика решения: $O(M)$

Решение

```
n, m = map(int, input().split())
g = [tuple(map(int, input().split())) for _ in range(m)]

res = [[0] * n for _ in range(n)]
for u, v in g:
    res[u - 1][v - 1] = 1
for i in range(n):
    print(*res[i])
```

2 B. Степени вершин по списку ребер

Разбор

Степень вершины определяется как число рёбер, входящих и выходящих из неё. Нам изначально дан список рёбер. Так как ребро задаётся в виде пары вершин (u, v), для каждой из вершин увеличим счётчик степени на единицу.

Асимптотика решения: $O(M)$

Решение

```
n, m = map(int, input().split())
g = [tuple(map(int, input().split())) for _ in range(m)]
p = [0] * n

for u, v in g:
    p[u - 1] += 1
    p[v - 1] += 1
print(*p)
```

3 C. Обход графа

Разбор

Так как граф неориентированный, то просто запустим поиск в глубину из заданной вершины. Поиск в глубину гарантирует, что мы достигнем всех вершин, находящихся в одной компоненте связности с заданной вершиной. Поэтому в конце просто посчитаем, сколько значений в списке *used* равно единице.

Асимптотика решения: $O(N + M)$

Решение

```
n, u = map(int, input().split())
g = [list(map(int, input().split())) for _ in range(n)]

def dfs(v, g, used):
    used[v] = 1
    for i in range(n):
        if g[v][i] and not used[i]:
            dfs(i, g, used)

used = [0] * n
dfs(u - 1, g, used)
print(sum(used))
```

4 D. Дерево?

Разбор

Вспомним, что дерево – это связный граф без циклов, петель и кратных рёбер. Из этого следует замечательное свойство дерева – число рёбер m в дереве должно быть на единицу меньше числа вершин n: $m = n - 1$. Будем считать это необходимым условием существования дерева. Если для заданных n и m это условие не выполняется, то сразу выведем ответ "NO". Иначе запустим поиск в глубину по графу и проверим, что все вершины графа g достижимы из некоторой начальной вершины v. Если это так, то выведем ответ "YES".

Асимптотика решения: $O(N + M)$

Решение

```
def dfs(n, g, used):
    used[n] = 1
    for u, v in g:
        if u == n and not used[v]:
            dfs(v, g, used)
        if v == n and not used[u]:
            dfs(u, g, used)

def int_vertex(v):
    return int(v) - 1

def solve():
    n, m = map(int, input().split())
    if m != n - 1:
        return 'NO'
    g = [tuple(map(int_vertex, input().split())) for _ in range(m)]
    used = [0] * n
    dfs(0, g, used)
    return 'YES' if sum(used) == n else 'NO'

print(solve())
```

5 E. Компоненты связности

Разбор

Чтобы выделить все компоненты связности, достаточно просто в массиве *used* запоминать номер компоненты связности в процессе поиска в глубину. Обойдём все вершины нашего графа и будем запускать поиск в глубину каждый раз, когда встретим вершину, для которой не задан номер компоненты связности, после чего будем увеличивать счётчик числа компонент связности на 1.

Асимптотика решения: $O(N + M)$

Решение

```
from sys import setrecursionlimit, stdin
from collections import defaultdict

setrecursionlimit(100069)
input = stdin.readline

def dfs(v, g, used, c):
    used[v] = c
    for n in g[v]:
        if not used[n]:
            dfs(n, g, used, c)

n, m = map(int, input().split())
g = defaultdict(list)
for _ in range(m):
    u, v = map(int, input().split())
    g[u - 1].append(v - 1)
    g[v - 1].append(u - 1)
used, num_c = [0] * n, 1
for i in range(n):
    if not used[i]:
        dfs(i, g, used, num_c)
        num_c += 1
components = [[] for _ in range(num_c - 1)]
for i in range(n):
    components[used[i] - 1].append(i + 1)
print(num_c - 1)
for cmp in components:
    print(len(cmp))
    print(*cmp)
```

6 F. Удаление клеток

Разбор

Проведём аналогию тетрадного листа с картой в какой-нибудь игре. Из каждой клетки можно делать переход только вверх, вниз, влево и вправо. Помимо обычных клеток на карте есть препятствия, обозначенные точками '.' – их мы не будем добавлять в список смежности. После того как мы сформировали граф, просто выделим число компонент связности.

Асимптотика решения: $O(N \times M)$

Решение

```
from sys import setrecursionlimit, stdin
from collections import defaultdict

setrecursionlimit(100069)
input = stdin.readline

n, m = map(int, input().split())
lines = [input() for _ in range(n)]
num_v = n * m

def dfs(n, g, used):
    used[n] = 1
    for v in g[n]:
        if not used[v]:
            dfs(v, g, used)

used = [0] * num_v
g = defaultdict(list)
for i in range(n):
    for j in range(m):
```

```
            if lines[i][j] == '.':
                used[m * i + j] = 1
            if j + 1 < m and lines[i][j + 1] == '#':
                g[m * i + j].append(m * i + (j + 1))
                g[m * i + (j + 1)].append(m * i + j)
            if i + 1 < n and lines[i + 1][j] == '#':
                g[m * i + j].append(m * (i + 1) + j)
                g[m * (i + 1) + j].append(m * i + j)

cnt = 0
for i in range(num_v):
    if not used[i]:
        cnt += 1
        dfs(i, g, used)
print(cnt)
```

7 G. Получи дерево

Разбор

Чтобы получить из произвольного графа дерево, мы можем просто выкинуть рёбра, ведущие в уже посещённые вершины. Т.к. изначальный граф является связным, не будем заморачиваться с компонентами связности. Запустим поиск в глубину из произвольной вершины v и будем выводить в процессе обхода каждое ребро, взятое нами.

Асимптотика решения: $O(N + M)$

Решение

```
from sys import setrecursionlimit, stdin
from collections import defaultdict

setrecursionlimit(100069)
input = stdin.readline

def dfs(u, g, used):
    used[u] = 1
    for v in g[u]:
        if not used[v]:
            print(u + 1, v + 1)
            dfs(v, g, used)

n, m = map(int, input().split())
g = defaultdict(list)
for _ in range(m):
    u, v = map(int, input().split())
    g[u - 1].append(v - 1)
    g[v - 1].append(u - 1)
used = [0] * n
dfs(0, g, used)
```

8 H. Долой списывание!

Разбор

В данной задаче нам нужно проверить граф связей лкшат друг с другом на двудольность, об этом почти прямым текстом говорится в условии. Единственное, что требовалось учесть – что граф связей лкшат являлся неориентированным и мог не являться односвязным. Проверку на двудольность будем вести в лоб: запустим рекурсивный обход графа, который будет пытаться красить вершины в 2 цвета. Если же обнаружим конфликт – две соседние вершины одного цвета – значит граф не является двудольным и разделить лкшат на две группы нельзя.

Асимптотика решения: $O(N + M)$

Решение

```
from sys import setrecursionlimit, stdin
from collections import defaultdict

setrecursionlimit(100069)
input = stdin.readline

def dfs(u, g, colors, c):
    ok = 1
    colors[u] = c
    for v in g[u]:
        if colors[u] == colors[v]:
            return 0
        if colors[v] < 0:
            ok &= dfs(v, g, colors, c ^ 1)
    return ok

n, m = map(int, input().split())
g = defaultdict(list)
for _ in range(m):
    u, v = map(int, input().split())
    g[u - 1].append(v - 1)
    g[v - 1].append(u - 1)
colors = [-1] * n
ok = 1
for i in range(n):
    if colors[i] < 0:
        ok &= dfs(i, g, colors, 0)
print('YES' if ok else 'NO')
```

9 I. Есть ли цикл?

Разбор

Проверим наличие цикла в графе, используя время входа и выхода из вершины графа. Пусть мы сейчас стоим в вершине a и смотрим в вершину b. Поймём, что мы находимся в цикле, если время входа в вершину b **меньше** времени входа в вершину a (т.е. в вершину b мы уже входили ранее) и при этом время выхода из вершины b **не задано** (т.е. мы ещё не вышли из этой вершины – не посетили всех её потомков).

Так как граф ориентированный, запустим обход в глубину из каждой ранее непосещённой вершины.

Асимптотика решения: $O(N + M)$

Решение

```
from sys import setrecursionlimit, stdin
from collections import defaultdict

setrecursionlimit(100069)
input = stdin.readline

def dfs(u, g, t_in, t_out, t):
    t_in[u] = t
    for v in g[u]:
        if t_in[v] >= 0 and t_out[v] < 0:
            return None
        if t_in[v] < 0:
            t = dfs(v, g, t_in, t_out, t + 1)
            if not t:
                return None
    t_out[u] = t + 1
    return t_out[u]

n, m = map(int, input().split())
```

```
g = defaultdict(list)
for _ in range(m):
    u, v = map(int, input().split())
    g[u - 1].append(v - 1)
t_in, t_out = [-1] * n, [-1] * n
t = 0
for i in range(n):
    if t_in[i] < 0:
        t = dfs(i, g, t_in, t_out, t)
    if not t:
        break
print(1 if not t else 0)
```

10 J. Свинки-копилки

Разбор

Во-первых, решим вопрос с тем, как ориентировать рёбра в этой задаче. Будем направлять их от копилки u с ключом к копилке v, которую можно этим ключом открыть. Это будет означать, что если мы получили доступ (каким-либо образом) к содержимому копилки u, то и до содержимого копилки v мы также сможем добраться.

Далее обратим внимание на то, что в графе копилок обязательно будет хотя бы цикл, т.к. число вершин совпадает с числом рёбер. Заметим, что для того, чтобы открыть все копилки из цикла, нам достаточно разбить только одну копилку.

Будем обходить все ещё не разбитые копилки по порядку. Пусть мы начали обход с копилки x. В процессе обхода возможны два случая:

1. Мы собираемся сделать переход в ещё не вскрытую копилку. Тогда запомним для неё стартовую копилку (какую копилку мы изначально разбили, чтобы вскрыть текущую)

2. Мы собираемся сделать переход в уже вскрытую копилку и мы её вскрыли при помощи той же копилки, что и текущую. Тогда прекратим обход (ведь мы попали в цикл) и увеличим счётчик разбитых копилок на единицу

3. Мы собираемся сделать переход в уже вскрытую копилку и она была вскрыта при помощи другой копилки w. Тогда прекратим обход и **не** увеличим ответ, т.к. мы уже посчитали ранее разбитую копилку (будем считать, что мы разбили теперь изначально не w, а x)

Асимптотика решения: $O(N + M)$

Решение

```
from sys import setrecursionlimit, stdin
from collections import defaultdict

setrecursionlimit(100069)
input = stdin.readline

n = int(input())
key_in = [int(input()) - 1 for _ in range(n)]

def dfs(n, g, used, start):
    used[n] = start
    ans = -1
    for v in g[n]:
        if used[v] < 0:
            ans = dfs(v, g, used, start)
            if ans >= 0:
                return ans
        else:
            return int(used[v] == start)
    return ans
```

```
g = defaultdict(list)
for i in range(n):
    g[i].append(key_in[i])

cnt = 0
used = [-1] * n
for k in range(n):
    if used[k] < 0:
        cnt += dfs(k, g, used, k)
print(cnt)
```

11 K. Топологическая сортировка

Разбор

В этой задаче изначально хочется выполнить топологическую сортировку и сравнить со входной строкой. Но важно учитывать, что топологическая сортировка допускает разный порядок обхода. Поэтому вспомним, о чём вообще нам говорит топсорт. А говорит он нам о том, что мы идём от наиболее независимых вершин графа к более зависимым. Рассмотрим две вершины u и v. Если во входной строке u стоит левее v, это значит, что из u не существует пути в v, но верно обратное. Поэтому пройдёмся по всем рёбрам и найдём несоответствия индексов в заданной топологической сортировке. Если таких несоответствий нет – заданная топологическая сортировка является корректной.

Асимптотика решения: $O(N + M)$

Решение

```
from sys import setrecursionlimit, stdin
from collections import defaultdict

setrecursionlimit(100069)
input = stdin.readline

def solve():
    n, m = map(int, input().split())
    g = defaultdict(list)
    for _ in range(m):
        i, j = map(int, input().split())
        g[i].append(j)
    top_sort = list(map(int, input().split()))
    index = { top_sort[i]: i for i in range(n) }
    for i in range(n):
        index[top_sort[i]] = i
    for v in reversed(top_sort):
        for n in g[v]:
            if index[n] < index[v]:
                return 'NO'
    return 'YES'

print(solve())
```

12 L. Topsort

Разбор

Решим эту задачу аналогично задаче "I. Есть ли цикл?", только при выходе из вершины будем добавлять её в список вершин. Если цикл есть, то топологической сортировки не существует, иначе выведем список добавленных вершин в обратном порядке.

Асимптотика решения: $O(N + M)$

Решение

```
from sys import setrecursionlimit
from collections import defaultdict

setrecursionlimit(100069)
f_in = open('topsort.in', 'r')
f_out = open('topsort.out', 'w')
input = f_in.readline

def dfs(u, g, t_in, t_out, t, res: list):
    t_in[u] = t
    for v in g[u]:
        if t_in[v] >= 0 and t_out[v] < 0:
            return None
        if t_in[v] < 0:
            t = dfs(v, g, t_in, t_out, t + 1, res)
            if not t:
                return None
    res.append(u + 1)
    t_out[u] = t + 1
    return t_out[u]


n, m = map(int, input().split())
g = defaultdict(list)
for _ in range(m):
    u, v = map(int, input().split())
    g[u - 1].append(v - 1)
t_in, t_out, nodes = [-1] * n, [-1] * n, []
t = 0
for i in range(n):
    if t_in[i] < 0:
        t = dfs(i, g, t_in, t_out, t, nodes)
    if not t:
        print(-1, file=f_out)
        break
if t:
    print(*reversed(nodes), file=f_out)
```

13 M. Предок

Разбор

Т.к. нам задано дерево, просто обойдём его и посчитаем для каждой вершины время входа и выхода из вершины. Тогда на каждый запрос мы можем ответить за константу. Чтобы проверить, является ли вершина u предком вершины v, достаточно просто проверить, что $t_{in}(u) < t_{in}(v)$ и $t_{out}(u) > t_{out}(v)$

Асимптотика решения: $O(N + M + Q)$

Решение

```
from sys import setrecursionlimit, stdin
from collections import defaultdict

setrecursionlimit(100069)
input = stdin.readline

def dfs(u, g, t_in, t_out, t):
    t_in[u] = t
    for v in g[u]:
        t = dfs(v, g, t_in, t_out, t + 1)
    t_out[u] = t + 1
    return t_out[u]

n = int(input())
```

```
p = list(map(int, input().split()))
g = defaultdict(list)
for i in range(n):
    u, v = p[i] - 1, i
    g[u].append(v)
root = g[-1][0]

t_in, t_out, t = [-1] * n, [-1] * n, 0
dfs(root, g, t_in, t_out, t)

q = int(input())
for _ in range(q):
    u, v = map(int, input().split())
    ans = int(t_in[u - 1] < t_in[v - 1] and t_out[u - 1] > t_out[v - 1])
    print(ans)
```