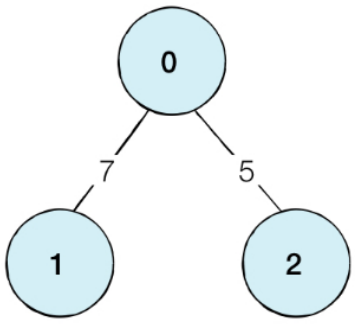
Week5 - DFS/BFS

#개념

그래프

- Node(정점)와 Edge(간선)로 표현
  - 그래프 탐색: 하나의 노드를 시작으로 다수의 노드를 방문
  - 인접: 두 노드가 Edge로 연결되어 있을 경우
    - 인접 행렬: 2차원 배열로 그래프의 연결 관계를 표현하는 방식

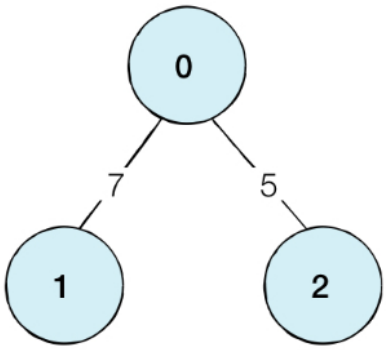


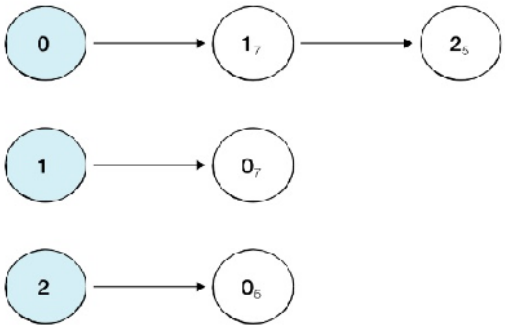
	0	1	2
0	0	7	5
1	7	0	무한
2	5	무한	0

→ 연결되지 않은 노드끼리는 "무한"으로 표현

→ 모든 관계를 저장 ∴ 메모리 낭비

    - 인접 리스트: 리스트로 그래프의 연결 관계를 표현하는 방식





→ C++, Java: 연결 리스트 자료 구조 사용

→ Python: 2차원 리스트 이용

→ 모든 노드에 연결된 노드의 정보를 차례대로 연결하여 저장

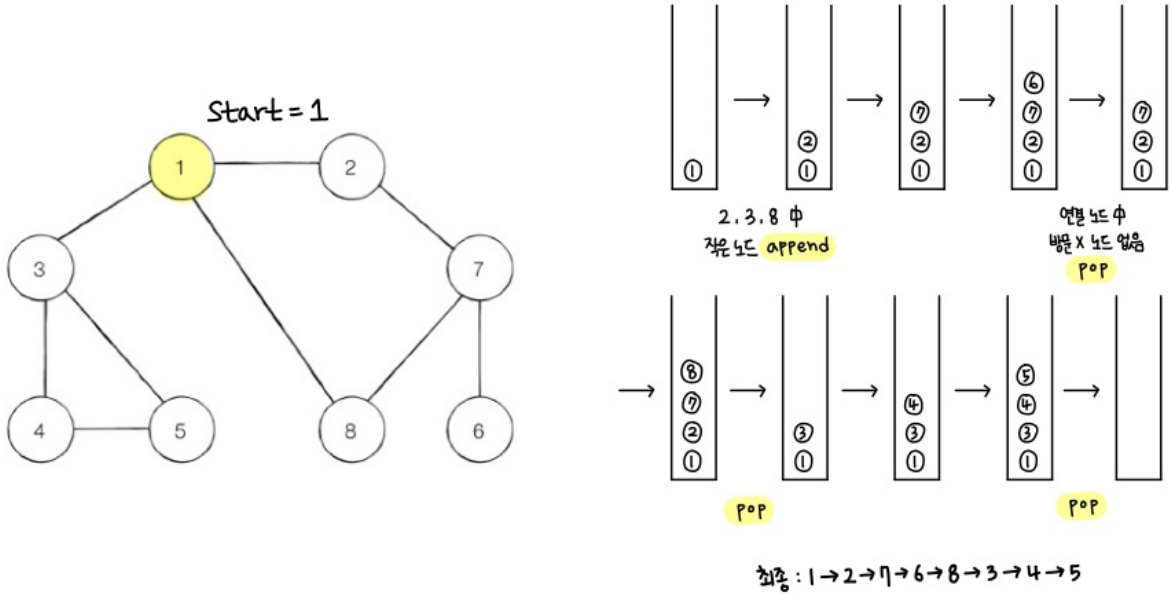
→ 필요 관계만 저장 = 메모리 효율적

DFS(Depth-First Search): 깊이 우선 탐색

- 그래프에서 깊은 부분을 우선적으로 탐색하는 알고리즘

  → 한 노드의 연결 노드를 끝까지 순회

- 스택 자료구조 이용



- 동작 과정

  ① 탐색 시작 노드를 스택에 삽입하고 방문 처리

  ② 스택의 최상단 노드에 방문하지 않은 인접 노드가 있을 경우, 그 인접 노드를 스택에 넣고 방문 처리

  ② 스택의 최상단 노드에 방문하지 않은 인접 노드가 없을 경우, 스택에서 최상단 노드 인출

  ③ 2번 과정을 더 이상 수행할 수 없을 때까지 반복

  → 방문 처리: 스택에 한 번 삽입되어 처리된 노드가 다시 삽입되지 않게 체크

- 시간복잡도: O(N)

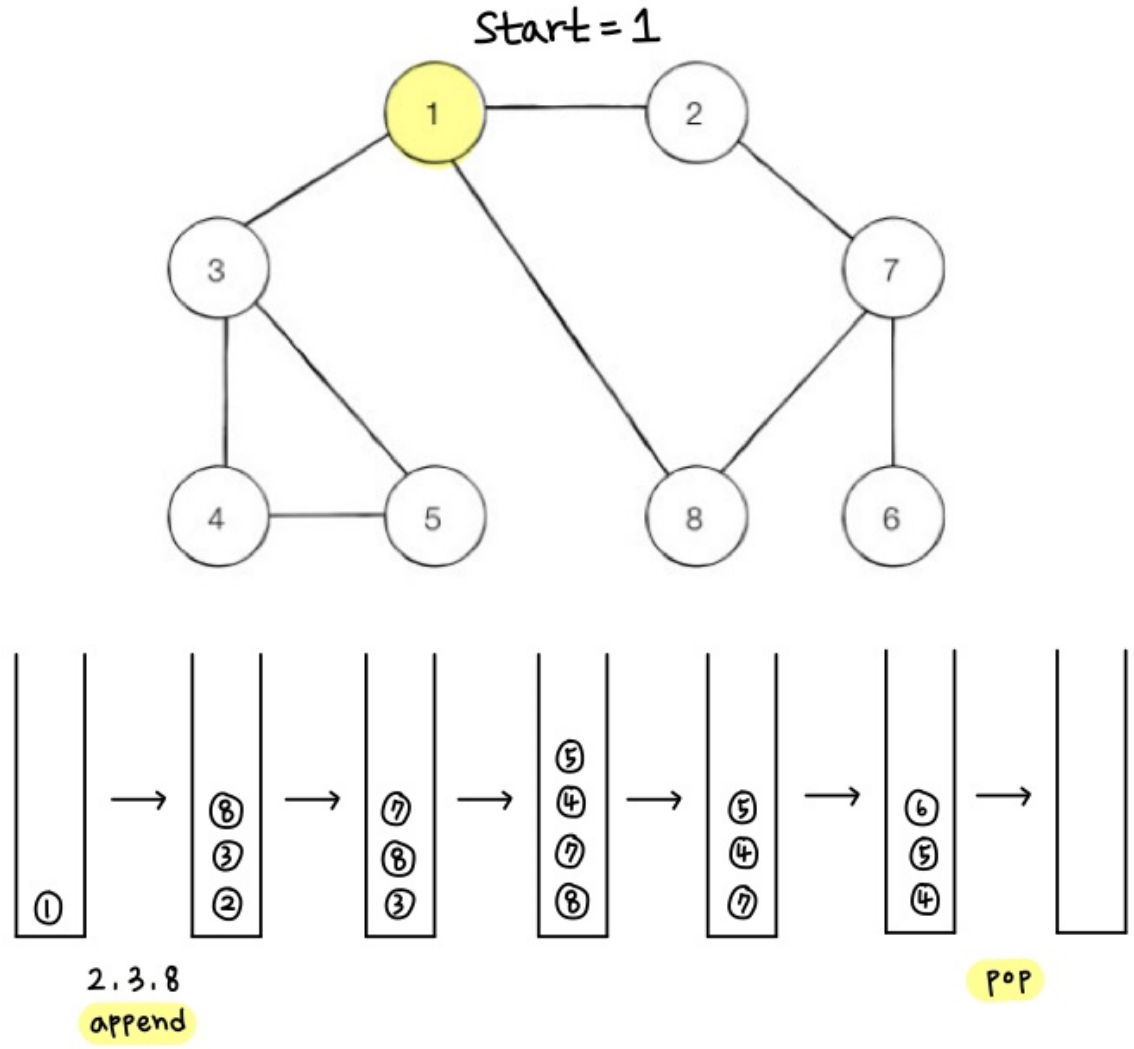
▼ Ex.

```
# DFS 함수 정의
def dfs(graph, v, visited):
    # 현재 노드를 방문 처리
    visited[v] = True
    print(v, end=' ')

    # 현재 노드와 연결된 다른 노드를 재귀적으로 방문
    for i in graph[v]:
        if not visited[i]: # 인접 노드가 방문한 적 없는 상태일 경우
            dfs(graph, i, visited)

# 각 노드가 연결된 정보를 리스트 자료형으로 표현(2차원 리스트)
graph = [
  [], # node 0은 없음 -> index 0 = 사용X
  [2, 3, 8],
  [1, 7],
  [1, 4, 5],
  [3, 5],
  [3, 4],
  [7],
  [2, 6, 8],
  [1, 7]
]

# 각 노드가 방문된 정보를 리스트 자료형으로 표현(1차원 리스트)
visited = [False] * 9 # node 0은 없음 -> index 0 = 사용X

# 정의된 DFS 함수 호출
dfs(graph, 1, visited)
```

BFS(Breadth First Search): 너비 우선 탐색

- 그래프에서 가까운 노드부터 탐색하는 알고리즘

  → 같은 깊이의 노드를 먼저 순회
- 선입선출 방식의 큐 자료구조 이용
- 특정 조건에서 최단 경로 계산 문제에 출제(Edge 크기 동일 시)



- 동작 과정

  ① 탐색 시작 노드를 큐에 삽입하고 방문 처리

  ② 큐에서 노드를 꺼내 해당 노드의 인접 노드 중 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리

  ③ 2번 과정을 더 이상 수행할 수 없을 때까지 반복

- 시간복잡도: O(N)

▼ Ex.

```
from collections import deque

# BFS 함수 정의
def bfs(graph, start, visited):
    # 큐(Queue) 구현을 위해 deque 라이브러리 사용
    queue = deque([start]) # 시작 노드

    # 현재 노드를 방문 처리
    visited[start] = True

    # 큐가 빌 때까지 반복
    while queue:
        # 큐에서 하나의 원소를 뽑아 출력
        v = queue.popleft() # 선입선출
        print(v, end=' ')

        # 해당 원소와 연결된, 아직 방문하지 않은 원소들을 큐에 삽입
        for i in graph[v]:
            if not visited[i]:
                queue.append(i)
                visited[i] = True

# 각 노드가 연결된 정보를 리스트 자료형으로 표현(2차원 리스트)
graph = [
  [],
  [2, 3, 8],
  [1, 7],
  [1, 4, 5],
  [3, 5],
  [3, 4],
  [7],
  [2, 6, 8],
  [1, 7]
]

# 각 노드가 방문된 정보를 리스트 자료형으로 표현(1차원 리스트)
visited = [False] * 9

# 정의된 BFS 함수 호출
bfs(graph, 1, visited)
```

DFS/BFS 비교

- DFS, BFS의 수행 시간은 O(N)으로 동일하나, BFS가 실제 시간 효율성↑

#문제

네트워크

네트워크란 컴퓨터 상호 간에 정보를 교환할 수 있도록 연결된 형태를 의미합니다.

예를 들어, 컴퓨터 A와 컴퓨터 B가 직접적으로 연결되어있고, 컴퓨터 B와 컴퓨터 C가 직접적으로 연결되어 있을 때 컴퓨터 A와 컴퓨터 C도 간접적으로 연결되어 정보를 교환할 수 있습니다. 따라서 컴퓨터 A, B, C는 모두 같은 네트워크 상에 있다고 할 수 있습니다.

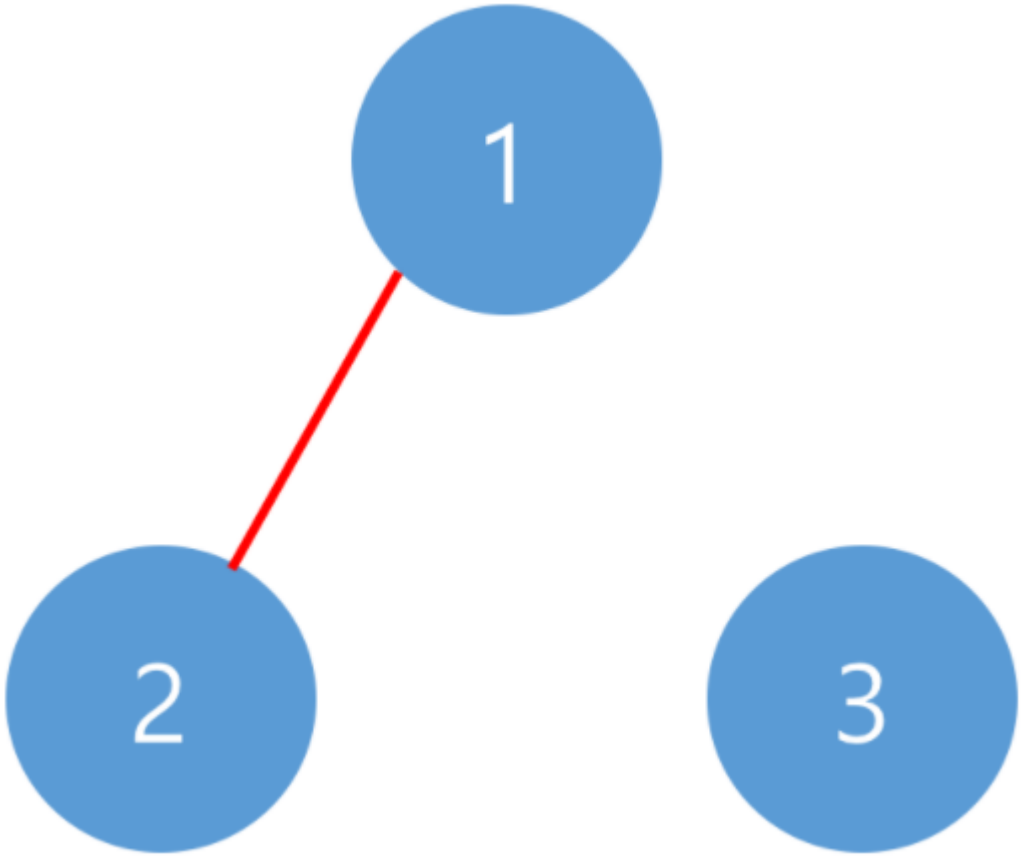
컴퓨터의 개수 n, 연결에 대한 정보가 담긴 2차원 배열 computers가 매개변수로 주어질 때, 네트워크의 개수를 return 하도록 solution 함수를 작성하시오.

제한 사항

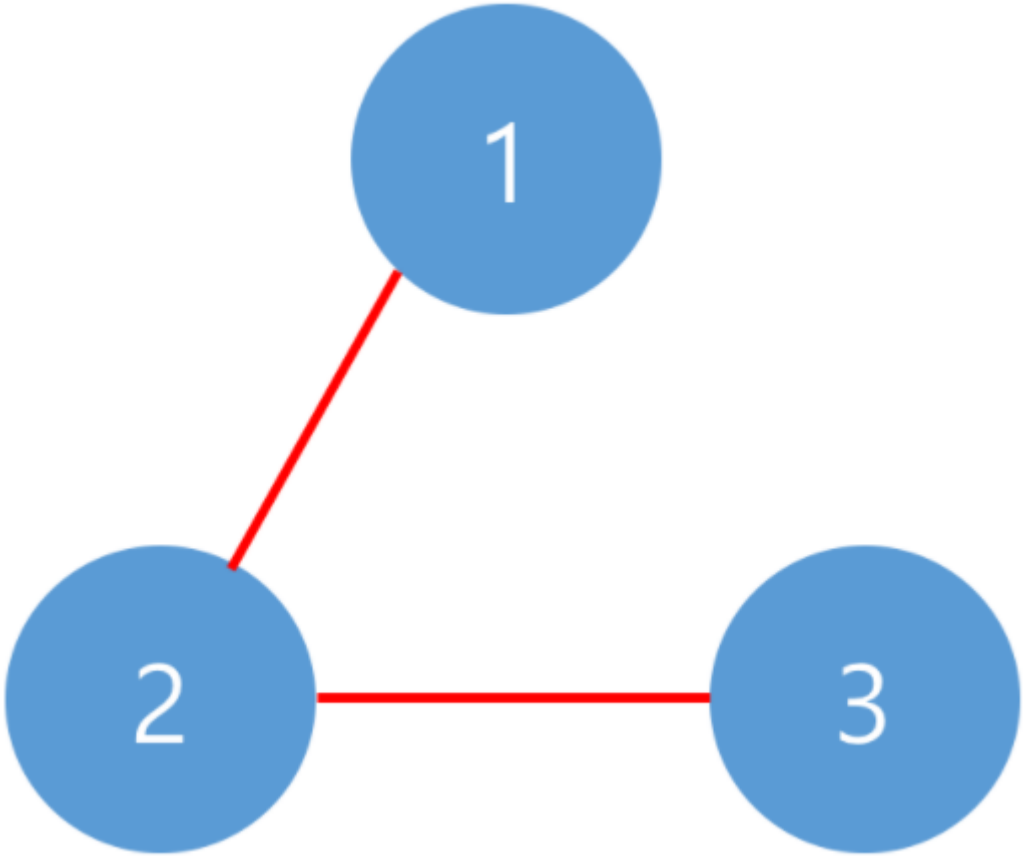
- 컴퓨터의 개수 n은 1 이상 200 이하인 자연수입니다.
- 각 컴퓨터는 0부터 `n-1`인 정수로 표현합니다.
- i번 컴퓨터와 j번 컴퓨터가 연결되어 있으면 computers[i][j]를 1로 표현합니다.
- computer[i][i]는 항상 1입니다.

입출력

입출력 예 #1



입출력 예 #2



풀이

```
def dfs(graph, v, visited):
    visited[v] = True

    for i in graph[v]:
        if not visited[i]:
            dfs(graph, i, visited)

def solution(n, computers):
    # 인접 행렬 -> 인접 리스트
    nodes = []
    for i in range(n):
        computer = computers[i]
        nodes.append([])
        for j in range(n):
            if i == j or computer[j] == 0:
                continue
            nodes[i].append(j)

    # 방문 기록
    visited = [False] * n
```

```
    cnt = 1
    start = 0
    while True:
        dfs(nodes, start, visited) # 연결된 노드 check
        if sum(visited) == n:
            break

        start = visited.index(False)
        cnt += 1

    return cnt
```

접근 방법

- 인접 리스트 이용
- 재귀 방식

단어 변환

두 개의 단어 begin, target과 단어의 집합 words가 있습니다. 아래와 같은 규칙을 이용하여 begin에서 target으로 변환하는 가장 짧은 변환 과정을 찾으려고 합니다.

```
1. 한 번에 한 개의 알파벳만 바꿀 수 있습니다.
2. words에 있는 단어로만 변환할 수 있습니다.
```

두 개의 단어 begin, target과 단어의 집합 words가 매개변수로 주어질 때, 최소 몇 단계의 과정을 거쳐 begin을 target으로 변환할 수 있는지 return 하도록 solution 함수를 작성해주세요.

제한 사항

- 각 단어는 알파벳 소문자로만 이루어져 있습니다.
- 각 단어의 길이는 3 이상 10 이하이며 모든 단어의 길이는 같습니다.
- words에는 3개 이상 50개 이하의 단어가 있으며 중복되는 단어는 없습니다.
- begin과 target은 같지 않습니다.

- 변환할 수 없는 경우에는 0를 return 합니다.

입출력

입출력 예 #1

"hit" -> "hot" -> "dot" -> "dog" -> "cog"와 같이 4단계를 거쳐 변환할 수 있습니다.

입출력 예 #2

target인 "cog"는 words 안에 없기 때문에 변환할 수 없습니다.

풀이

```
def dfs(begin, target, words, visited):
    stack = [begin]
    cnt = 0
    while stack:
        visit = stack.pop()
        if visit == target:
            return cnt
        for v in range(len(words)):
            word = words[v]
            diff = 0
            for i in range(len(word)):
                alp = word[i]
                if alp != visit[i]:
                    diff += 1
            if diff == 1 and visited[v] == False:
                visited[v] = True
                stack.append(word)
        cnt += 1
    return cnt

def solution(begin, target, words):
    answer = 0

    if target not in words:
        return 0

    visited = [False] * len(words)
    answer = dfs(begin, target, words, visited)
    return answer
```

접근 방법

- 스택을 이용하여 기준 단어와 check할 단어의 알파벳 차이를 세고 1개만 다르면서 check하지 않았던 단어인 경우, 방문 처리 및 기준 단어로 설정

여행 경로

주어진 항공권을 모두 이용하여 여행경로를 짜려고 합니다. **항상 "ICN" 공항에서 출발**합니다.

항공권 정보가 담긴 2차원 배열 tickets가 매개변수로 주어질 때, **방문하는 공항 경로**를 배열에 담아 return 하도록 solution 함수를 작성해주세요.

제한 사항

- 모든 공항은 알파벳 대문자 3글자로 이루어집니다.
- 주어진 공항 수는 3개 이상 10,000개 이하입니다.
- tickets의 각 행 **[a, b]는 a 공항에서 b 공항으로 가는 항공권**이 있다는 의미입니다.
- **주어진 항공권은 모두 사용**해야 합니다.
- 만일 가능한 경로가 2개 이상일 경우 알파벳 순서가 앞서는 경로를 return 합니다.
- 모든 도시를 방문할 수 없는 경우는 주어지지 않습니다.

입출력

입출력 예 #1

["ICN", "JFK", "HND", "IAD"] 순으로 방문할 수 있습니다.

입출력 예 #2

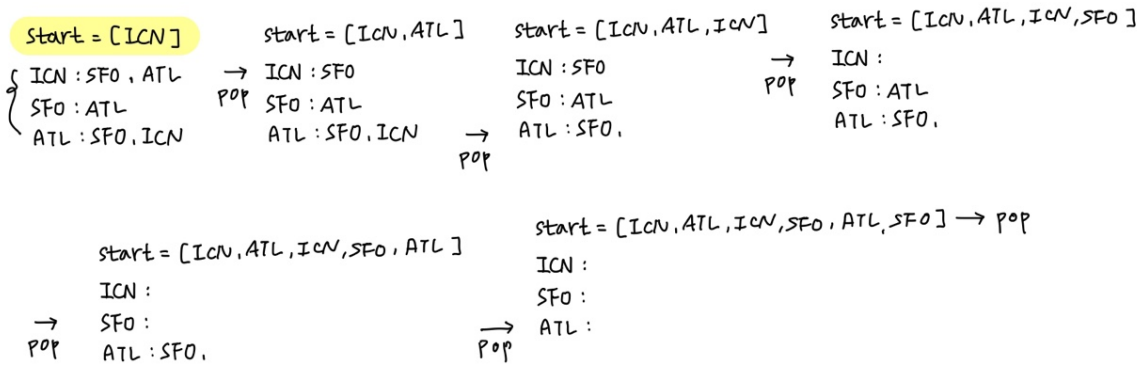
["ICN", "SFO", "ATL", "ICN", "ATL", "SFO"] 순으로 방문할 수도 있지만 ["ICN", "ATL", "ICN", "SFO", "ATL", "SFO"] 가 알파벳 순으로 앞섭니다.

풀이

```
from collections import defaultdict

def solution(tickets):
    answer = []

    nodes = defaultdict(list)
    n = len(tickets)

    # ex. {"ICN" : ["SFO", "ATL"]}
    for i in range(n):
        node = tickets[i][0]
        linked_node = tickets[i][1]
        nodes[node].append(linked_node)
        nodes[node].sort(reverse=True)

    start = ['ICN'] # 항상 "ICN" 공항에서 출발
    while len(answer) < (n+1):
        target = start[-1]
        print(target)

        # 해당 공항에서 출발하는 항공권이 없다면 마지막 경로라는 의미
        if nodes[target] == [] or target not in nodes:
            answer.append(start.pop())
        else:
            start.append(nodes[target].pop())

    return answer[::-1]
```

접근 방법

- 스택 이용



토마토

철수의 토마토 농장에서는 토마토를 보관하는 큰 창고를 가지고 있다.

창고에 보관되는 토마토들 중에는 잘 익은 것도 있지만, 아직 익지 않은 토마토들도 있을 수 있다. 보관 후 하루가 지나면, 익은 토마토들의 인접한 곳에 있는 익지 않은 토마토들은 익은 토마토의 영향을 받아 익게 된다. 하나의 토마토의 인접한 곳은 왼쪽, 오른쪽, 앞, 뒤 네 방향에 있는 토마토를 의미한다. 대각선 방향에 있는 토마토들에게는 영향을 주지 못하며, 토마토가 혼자 저절로 익는 경우는 없다고 가정한다. 철수는 창고에 보관된 토마토들이 며칠이 지나면 다 익게 되는지, 그 최소 일수를 알고 싶어 한다.

토마토를 창고에 보관하는 격자모양의 상자들의 크기와 익은 토마토들과 익지 않은 토마토들의 정보가 주어졌을 때, **며칠이 지나면 토마토들이 모두 익는지, 그 최소 일수를 구하는 프로그램**을 작성하라. 단, 상자의 일부 칸에는 토마토가 들어있지 않을 수도 있다.

입력

첫 줄에는 상자의 크기를 나타내는 두 정수 M,N이 주어진다. M은 상자의 가로 칸의 수, N은 상자의 세로 칸의 수를 나타낸다. 단, 2 ≤ M,N ≤ 1,000 이다. 둘째 줄부터는 하나의 상자에 저장된 토마토들의 정보가 주어진다. 즉, 둘째 줄부터 N개의 줄에는 상자에 담긴 토마토의 정보가 주어진다. 하나의 줄에는 상자 가로줄에 들어있는 토마토의 상태가 M개의 정수로 주어진다. 정수 1은 익은 토마토, 정수 0은 익지 않은 토마토, 정수 -1은 토마토가 들어있지 않은 칸을 나타낸다.

토마토가 하나 이상 있는 경우만 입력으로 주어진다.

출력

여러분은 토마토가 모두 익을 때까지의 최소 날짜를 출력해야 한다. 만약, 저장될 때부터 모든 토마토가 익어있는 상태이면 0을 출력해야 하고, 토마토가 모두 익지는 못하는 상황이면 -1을 출력해야 한다.

예제 입출력

예제 입력 1 복사

```
6 4
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 1
```

예제 출력 1 복사

```
8
```

예제 입력 2 복사

```
6 4
0 -1 0 0 0 0
-1 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 1
```

예제 출력 2 복사

```
-1
```

예제 입력 3 복사

```
6 4
1 -1 0 0 0 0
0 -1 0 0 0 0
0 0 0 0 -1 0
0 0 0 0 -1 1
```

예제 출력 3 복사

```
6
```

예제 입력 4 복사

```
5 5
-1 1 0 0 0
0 -1 -1 -1 0
0 -1 -1 -1 0
0 -1 -1 -1 0
0 0 0 0 0
```

예제 출력 4 복사

```
14
```

풀이

```
from collections import deque
```

```
def bfs(queue):
    dx = [1, -1, 0, 0]
    dy = [0, 0, 1, -1]

    while queue:
        x, y = queue.popleft()
        for i in range(4):
            nx = x + dx[i]
            ny = y + dy[i]

            if nx < 0 or nx >= n or ny < 0 or ny >= m:
                continue

            if tomatos[nx][ny] == 0:
                tomatos[nx][ny] = tomatos[x][y] + 1
                queue.append((nx, ny))

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

tomatos = [list(map(int, input().split())) for _ in range(n)]
ripe = []
for row in range(n):
    for col in range(m):
        if tomatos[row][col] == 1:
            ripe.append((row, col))
queue = deque(ripe)
bfs(queue)

_max = max(max(row) for row in tomatos)
for row in range(n):
    for col in range(m):
        if tomatos[row][col] == 0:
            print(-1)
            break
    else:
        continue
    break
else:
    print(_max-1)
```

접근 방법

- BFS

케빈 베이컨의 6단계 법칙

케빈 베이컨 게임은 임의의 두 사람이 최소 몇 단계 만에 이어질 수 있는지 계산하는 게임이다.

케빈 베이컨의 수가 가장 작은 사람을 찾으려고 한다. 케빈 베이컨 수는 모든 사람과 케빈 베이컨 게임을 했을 때, 나오는 단계의 합이다.

BOJ 유저의 수와 친구 관계가 입력으로 주어졌을 때, 케빈 베이컨의 수가 가장 작은 사람을 구하는 프로그램을 작성하시오.

입력

첫째 줄에 유저의 수 **N (2 ≤ N ≤ 100)**과 친구 관계의 수 M (1 ≤ M ≤ 5,000)이 주어진다. 둘째 줄부터 M개의 줄에는 친구 관계가 주어진다.

친구 관계는 중복되어 들어올 수도 있으며, 친구가 한 명도 없는 사람은 없다. 또, 모든 사람은 친구 관계로 연결되어져 있다. 사람의 번호는 1부터 N까지이며, 두 사람이 같은 번호를 갖는 경우는 없다.

출력

첫째 줄에 BOJ의 유저 중에서 케빈 베이컨의 수가 가장 작은 사람을 출력한다. 그런 사람이 여러 명일 경우에는 번호가 가장 작은 사람을 출력한다.

예제 입력

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

예제 출력

```
3
```

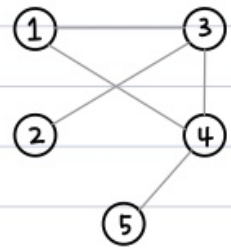
코드

```
n, m = map(int, input().split())
graph = [[int(1e9)] * (n+1) for _ in range(n+1)]

# 그래프 초기화
for a in range(1, n+1):
    for b in range(1, n+1):
        if a == b:
            graph[a][b] = 0

for _ in range(m):
    a, b = map(int, input().split())
    graph[a][b] = 1
    graph[b][a] = 1


# 플로이드 와샬 알고리즘
for k in range(1, n+1):
    for a in range(1, n+1):
        for b in range(1, n+1):
            graph[a][b] = min(graph[a][b], graph[a][k]+graph[k][b])

# 최소 노드 찾기
dist_cnt = []
for start in range(1, n+1):
    dist = 0
    for end in range(1, n+1):
        if start == end:
            continue
        dist += graph[start][end]
    dist_cnt.append((start, dist))
dist_cnt.sort(key = lambda x : x[1])
print(dist_cnt[0][0])
```

접근 방식

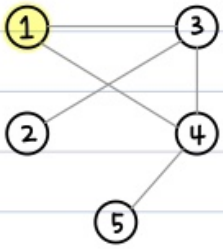
- 플로이드 워샬 알고리즘 → 시간복잡도: O(N^3) ∴ n ≤ 500 때, 효율적
- **각 단계마다 특정 노드 k를 거쳐가는 경우** 확인
- a → b 최단거리 vs a→k→b 거리
- 점화식: $D_{ab} = min(D_{ab}, D_{ak} + D_{kb})$
    - 기존 a-b 거리 = 기존 최단 거리 vs k를 거친 거리

초기화)



S \ E	1	2	3	4	5
1	0	Inf	1	1	Inf
2	Inf	0	1	Inf	Inf
3	1	1	0	1	Inf
4	1	Inf	1	0	1
5	Inf	Inf	Inf	1	0

K=1)



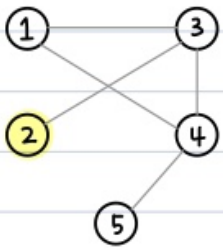
$D_{23} = Min(D_{23}, D_{21} + D_{13})$ (1, Inf + 1)

$D_{24} = Min(D_{24}, D_{21} + D_{14})$ (Inf, Inf + 1)

⋮

S \ E	1	2	3	4	5
1	0	Inf	1	1	Inf
2	Inf	0	1	Inf	Inf
3	1	1	0	1	Inf
4	1	Inf	1	0	1
5	Inf	Inf	Inf	1	0

K=2)



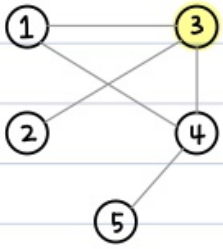
$D_{13} = Min(D_{13}, D_{12} + D_{23})$

$D_{14} = Min(D_{14}, D_{12} + D_{24})$

⋮

S \ E	1	2	3	4	5
1	0	Inf	1	1	Inf
2	Inf	0	1	Inf	Inf
3	1	1	0	1	Inf
4	1	Inf	1	0	1
5	Inf	Inf	Inf	1	0

K=3)



$D_{12} = Min(D_{12}, D_{13} + D_{32}) = (\infty, 2)$

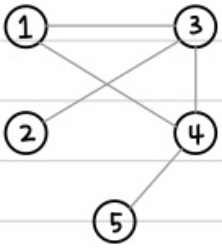
$D_{21} = Min(D_{21}, D_{23} + D_{31}) = (\infty, 2)$

$D_{24} = Min(D_{24}, D_{23} + D_{34}) = (\infty, 2)$

$D_{42} = Min(D_{42}, D_{43} + D_{32}) = (\infty, 2)$

S \ E	1	2	3	4	5
1	0	**2**	1	1	Inf
2	**2**	0	1	**2**	Inf
3	1	1	0	1	Inf
4	1	**2**	1	0	1
5	Inf	Inf	Inf	1	0

⋮



S \ E	1	2	3	4	5
1	0	2	1	1	**2**
2	2	0	1	2	**3**
3	1	1	0	1	**2**
4	1	2	1	0	1
5	**2**	**3**	**2**	1	0