2022 AlgoLive 4th study

Graph I

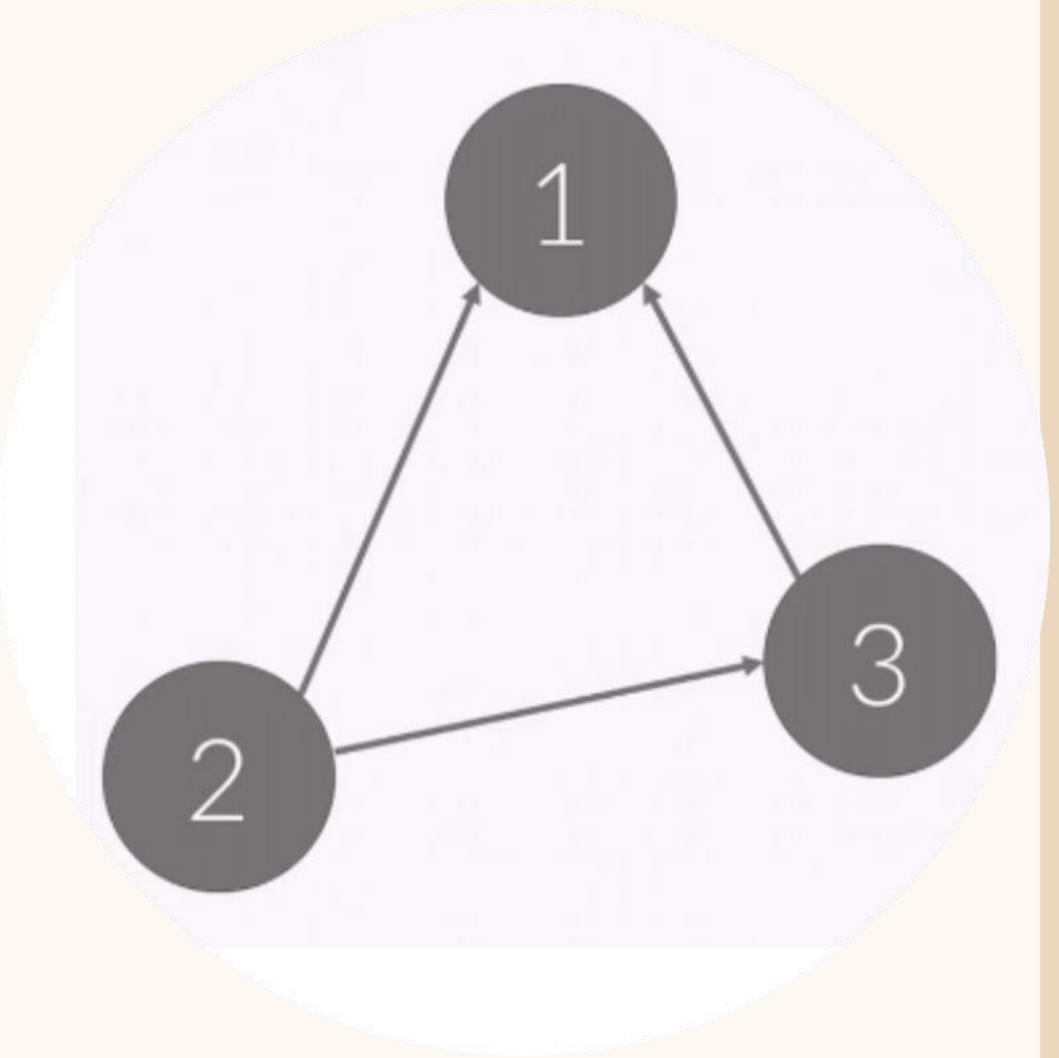
그래프 I : 기초 ~ DFS/BFS

목 차

CHAPTER. 1

그래프의 개념

정점(Node/Vertex)와 간선(Edge)으로 이루어진 자료구조



2022 AlgoLive 4th study

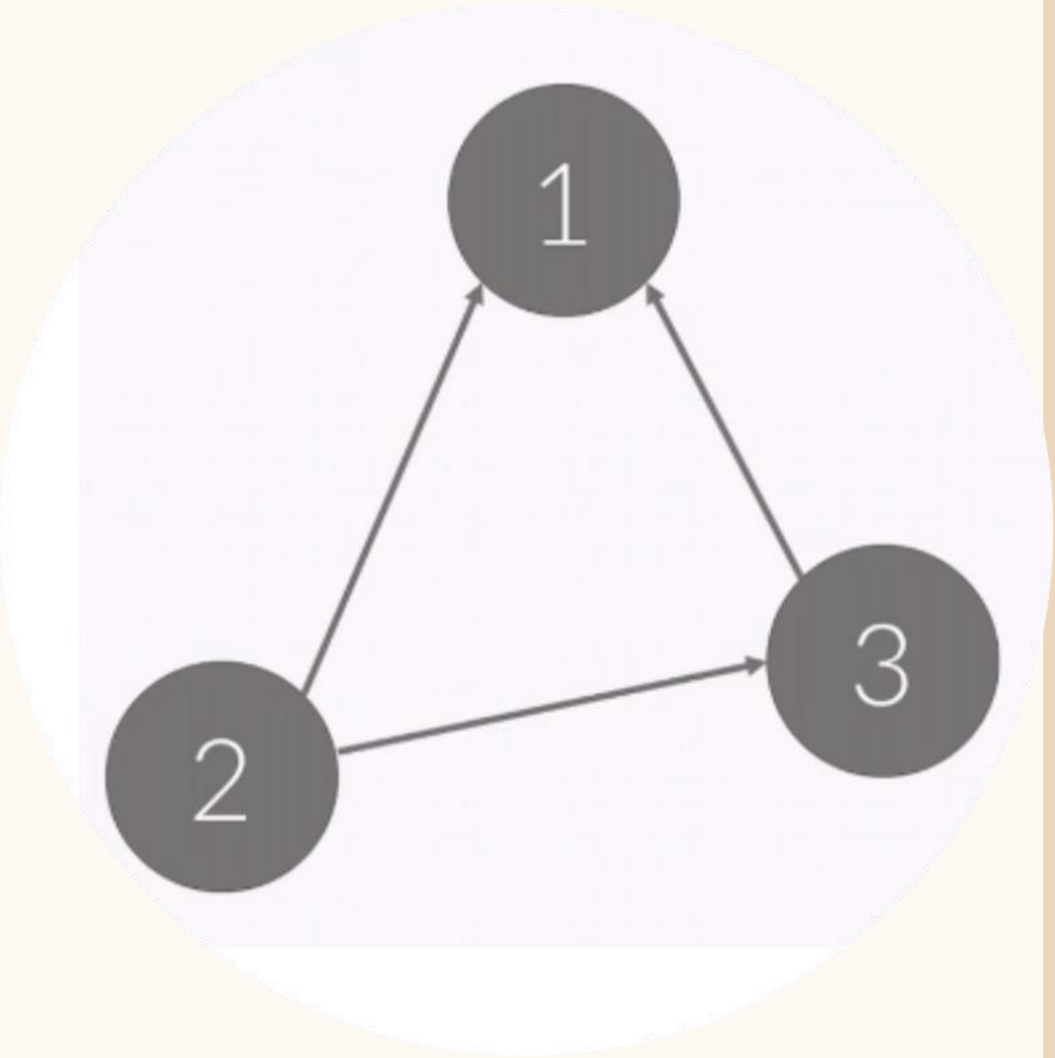
정점: 노드(Node)라고도 하며 데이터가 저장(1,2,3)

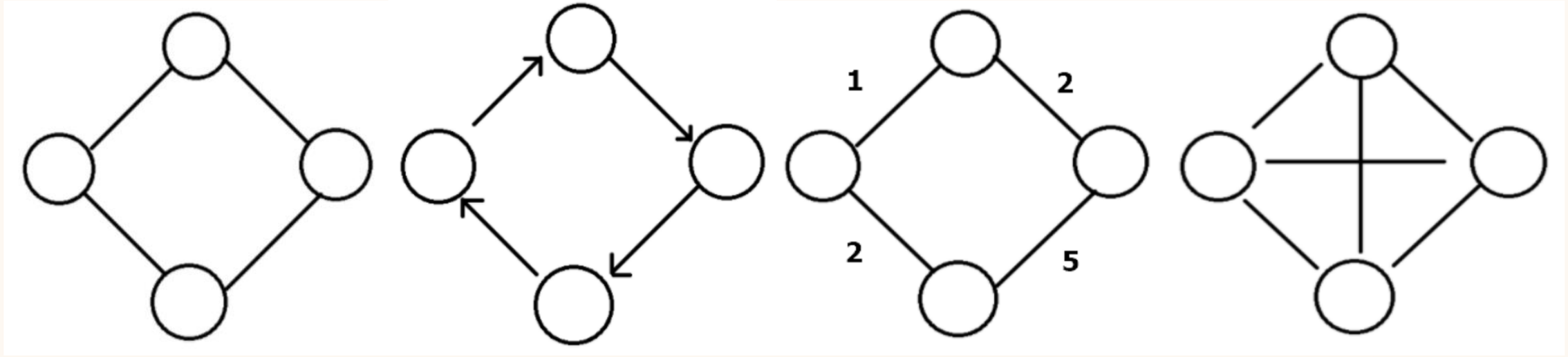
간선(edge): 링크(arcs)라고도 하며 노드간의 관계를 나타냄

차수(degree): 무방향 그래프에서 하나의 정점에 인접한 정점의 수

진출차수(out-deree): 방향그래프에서 한 노드에서 외부로 향하는 간선의 수

진입차수(in-degree): 방향 그래프에서 외부 노드에서 다가오는 간선의 수





무방향

두 정점을 연결하는
간선에
방향이 없음

방향

두 정점을 연결하는 간선에
방향이 존재하고
그 방향으로만 이동 가능

가중치

두 정점을 이동할 때
비용이 듦

완전

모든 정점이 간선으
로
연결되어 있음

2022 AlgoLive 4th study

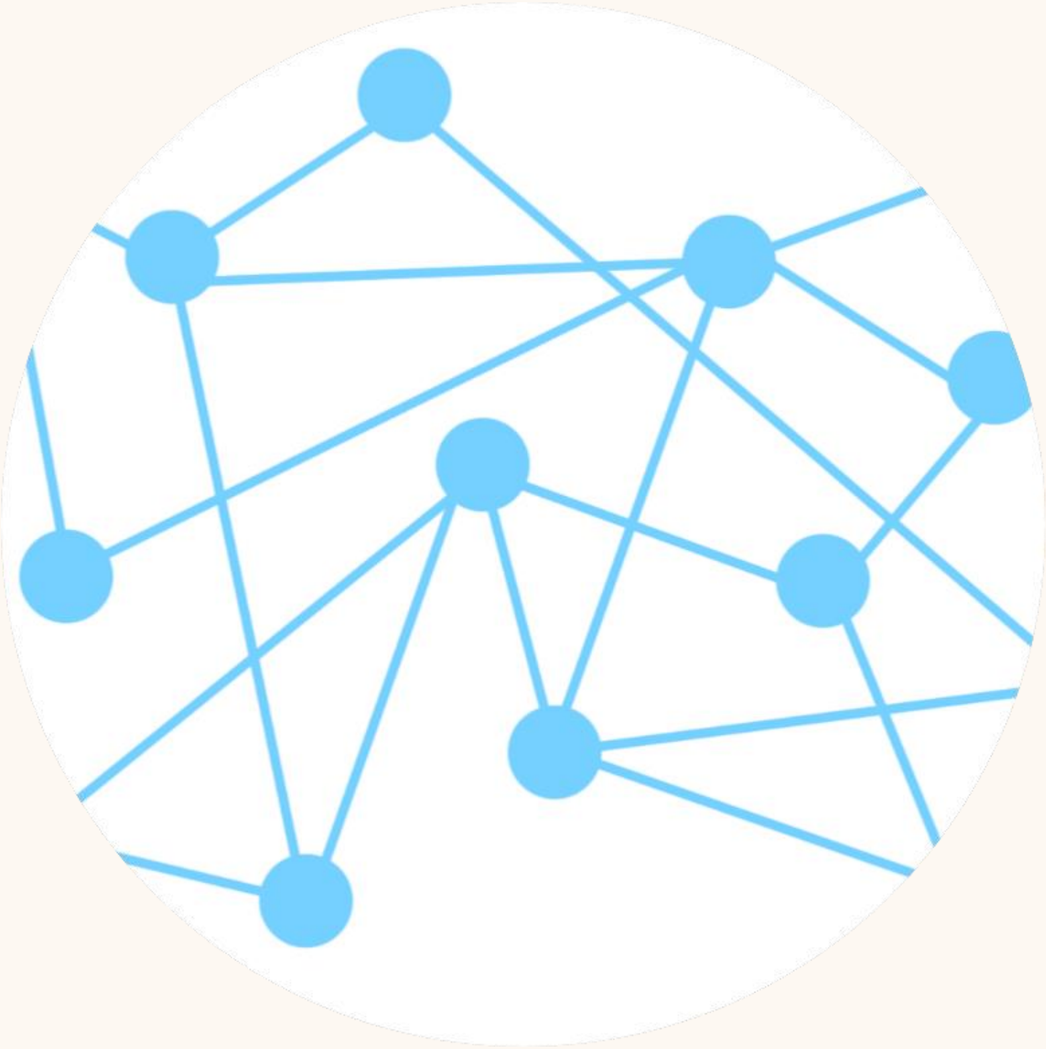
	그래프	트리
정의	노드(node)와 그 노드를 연결하는 간선(edge)을 하나로 모아 놓은 자료 구조	그래프의 한 종류 DAG(Directed Acyclic Graph, 방향성이 있는 비순환 그래프)의 한 종류
방향성	방향 그래프(Directed), 무방향 그래프(Undirected) 모두 존재	방향 그래프(Directed Graph)
사이클	사이클(Cycle) 가능, 자체 간선(self-loop)도 가능, 순환 그래프(Cyclic), 비순환 그래프(Acyclic) 모두 존재	사이클(Cycle) 불가능, 자체 간선(self-loop)도 불가능, 비순환 그래프(Acyclic Graph)
루트 노드	루트 노드의 개념이 없음	한 개의 루트 노드만이 존재, 모든 자식 노드는 한 개의 부모 노드 만을 가짐
부모-자식	부모-자식의 개념이 없음	부모-자식 관계 top-bottom 또는 bottom-top으로 이루어짐
모델	네트워크 모델	계층 모델
순회	DFS, BFS	DFS, BFS안의 Pre-, In-, Post-order
간선의 수	그래프에 따라 간선의 수가 다름, 간선이 없을 수도 있음	노드가 N인 트리는 항상 N-1의 간선을 가짐
경로	-	임의의 두 노드 간의 경로는 유일
예시 및 종류	지도, 지하철 노선도의 최단 경로, 전기 회로의 소자들, 도로(교차점과 일방 통행길), 선수 과목	이진 트리, 이진 탐색 트리, 균형 트리(AVL 트리, red-black 트리), 이진 힙(최대힙, 최소힙) 등

CHAPTER. 2

그래프의 표현

C++, Python, Java에서

그래프는 어떻게 표현되는가?



2022 AlgoLive 4th study

그래프의 두 가지 표현

인접 행렬

행렬은 정점, 행렬 값은 가중치 값

(가중치 없는 그래프의 경우 1)

행과 열은 정점을 나타내고 [i][j]값은 정점 i 로부터 j가 연결되어있다면 1, 그렇지 않다면 0으로 표시

가중치가 있다면 [i][j]에 가중치를 저장

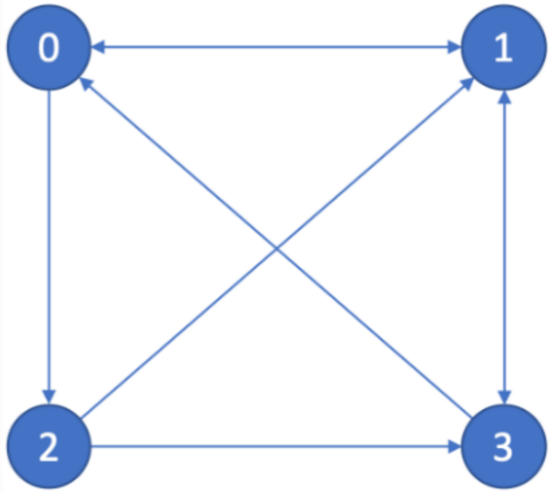
인접 리스트

Vector를 LinkedList처럼 사용

정점의 수만큼 리스트를 생성하고

해당 정점 리스트에다가 연결된 정점들을 추가

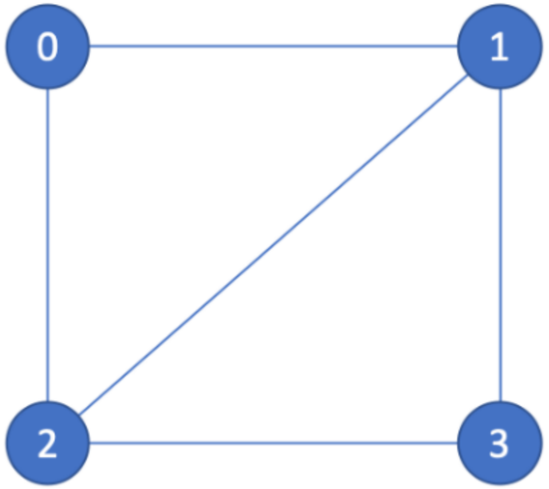
그래프의 표현: 인접 행렬



<to>

<from>	0	1	2	3
0	0	1	1	0
1	1	0	0	1
2	0	1	0	1
3	1	1	0	0

Directed Graph

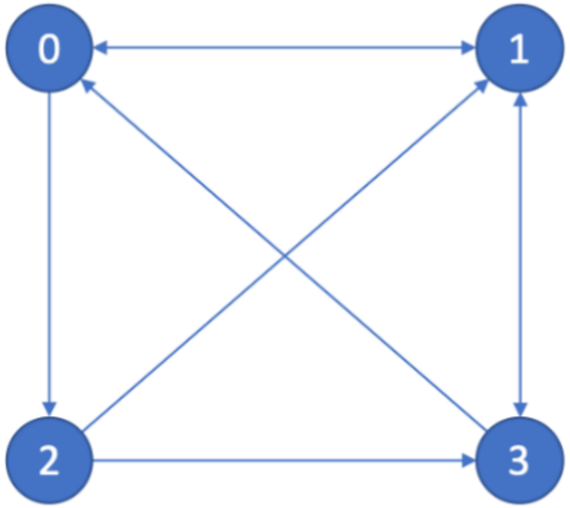


<to>

<from>	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0

Undirected Graph

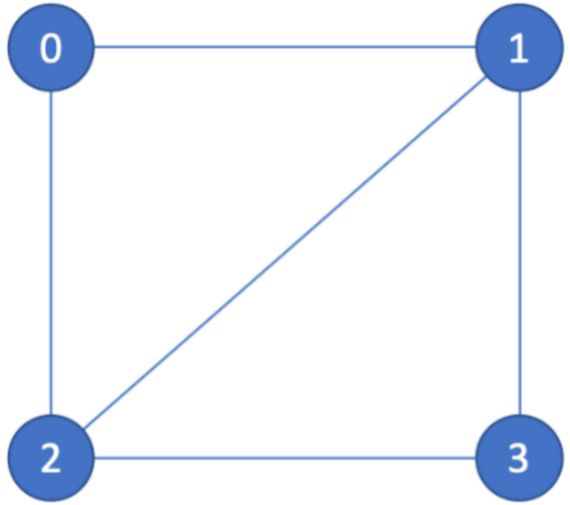
그래프의 표현: 인접 리스트



<vertex>

0	-> 1 -> 2
1	-> 0 -> 3
2	-> 1 -> 3
3	-> 0 -> 1

Directed Graph



<vertex>

0	-> 1 -> 2
1	-> 0 -> 2 -> 3
2	-> 0 -> 1 -> 3
3	-> 1 -> 2

Undirected Graph

인접행렬 vs 인접 리스트

인접 행렬과 인접 리스트의 장단점

그렇다면 두 방식은 각각 어떤 장단점을 가지고 있을까?

	인접 행렬	인접 리스트
시간 복잡도	O(N^2) 정점 N * N만큼 필요	O(N) N : 간선의 개수
두 정점의 연결 여부	graph[x][y] 의 값으로 한번에 확인	graph<x> 의 원소에서 y가 나올때까지 탐색
인접 노드 파악 여부	N * N만큼 반복문을 돌아 확인한다.	각 리스트에 담겨있는 원소를 확인한다.

먼저 행렬의 경우, 두 정점이 연결되있는지를 확인하는 방법이 쉽다. **graph[x][y]의 값을 바로 확인해서** 유무를 판단할 수 있기 때문이다. 단, 정점이 N개인 경우, 행렬을 만들기 위해선 **N * N만큼의 공간**이 필요하게 된다.
무방향 그래프의 경우는 절반의 공간이 낭비되는 셈이다.

이와 반대로 리스트의 경우, 실제 연결된 노드들만 리스트 원소에 담겨있으므로 **공간 복잡도가 N(간선)이다.**
다만, 두 정점 x, y가 연결되있는지 알고 싶다면 노드x 리스트로 들어가 원소 y가 있는지 처음부터 쭉 탐색해야 하므로 행렬보다 더 많은 시간이 소요된다.

둘 다 각각 장단점을 가지고 있으므로 상황에 따라 맞게 골라쓰면 된다.
간선이 많은 그래프의 경우, 인접 행렬을 통해 빠르게 **연결 여부**를 확인할 수 있다.
반면 **간선이 적은 그래프**의 경우는 인접 리스트를 통해 **인접 노드**를 빠르게 확인할 수 있다.

그래프의 구현: C++

```cpp
#include <iostream>
#include <stdio.h>
#include <vector>

using namespace std;

int main(){
    int n,m;
    cin >> n >> m; // 정점과 간선의 개수를 입력받음

    vector<int> graph[n+1];

    // Visual Studio의 경우
    /* 변수를 통해서 배열을 동적으로 생성할 때
    vector<int> * graph = new vector<int>[n+1];
    */

    for(int i=0; i<m; i++){
        int u,v;
        scanf("%d %d", &u, &v);
        graph[u].push_back(v);
        graph[v].push_back(u);
        // 단방향의 경우 graph[u].push_back(v);만 작성
        // 가중치가 있는 경우 vector<pair<int,int>> graph[n+1];로 만들거나 구조체를 만들
        // graph[u].push_back(make_pair(v,w)); u->v 가중치: w
    }
}
```

```cpp
#include <iostream>
#include <stdio.h>

using namespace std;

int main(){
    int n,m;
    cin >> n >> m; // 정점과 간선의 개수를 입력받음

    int graph[n+1][n+1];

    // Visual Studio의 경우
    /* 변수를 통해서 배열을 동적으로 생성할 때
    int ** graph = new int * [n];
    for (int i = 0; i <= n; ++i)
        graph[i] = new int[m];
    */

    // 정적으로 선언했기 때문에 값들을 전부 -1로 초기화
    for(int i=0; i<=n; i++){
        for(int j=0; j<=n; j++){
            graph[i][j] = -1;
        }
    }

    for(int i=0; i<m; i++){
        int u,v;
        scanf("%d %d", &u, &v);
        graph[u][v] = graph[v][u] = 1;
        // 단방향의 경우 graph[u][v] = 1;
        // 가중치가 있는 경우 단방향의 경우 graph[u][v] = 가중치값;
    }
}
```

그래프의 구현: Python

```python
n,m=map(int, input().split())
g=[[0]*(n+1) for _ in range(n+1)]

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

for i in range(1, n+1) :
    for j in range(1, n+1) :
        print(g[i][j], end=' ')
    print()
```

```python
# 그래프 객체를 나타내는 클래스
class Graph:
    # 생성자
    def __init__(self, edges, n):
        # 인접 목록에 대한 메모리 할당
        self.adjList = [[] for _ in range(n)]

        #는 방향 그래프에 간선을 추가합니다.
        for (src, dest) in edges:
            #는 인접 목록의 노드를 src에서 dest로 할당합니다.
            self.adjList[src].append(dest)


# 그래프의 인접 목록 표현을 인쇄하는 기능
def printGraph(graph):
    for src in range(len(graph.adjList)):
        # 현재 정점과 모든 인접 정점을 인쇄합니다.
        for dest in graph.adjList[src]:
            print(f'({src} -> {dest}) ', end='')
        print()


if __name__ == '__main__':

    # 입력: 유향 그래프의 간선
    edges = [(0, 1), (1, 2), (2, 0), (2, 1), (3, 2), (4, 5), (5, 4)]

    # 꼭짓점 수(0에서 5까지 레이블 지정)
    n = 6

    #는 주어진 간선 목록에서 그래프를 구성합니다.
    graph = Graph(edges, n)

    # 그래프의 인접 목록 인쇄
    printGraph(graph)
```

그래프의 구현: Java

```java
public class Main {

    public static void print(int[][] graph) {
        for (int i = 1; i < graph.length; i++) {
            for (int j = 1; j < graph.length; j++)
                System.out.print(graph[i][j]+ " ");
            System.out.println();
        }
    }

    public static void putEdge(int[][] graph, int x, int y) {
        graph[x][y] = 1;
        graph[y][x] = 1;
    }

    public static void main(String[] args) {
        int n = 5; //그래프 정점의 개수
        int[][] graph = new int[n+1][n+1]; //index를 1부터 맞추기 위해 n+1

        putEdge(graph, 1, 2);
        putEdge(graph, 1, 3);
        putEdge(graph, 1, 4);
        putEdge(graph, 2, 3);
        putEdge(graph, 2, 5);
        putEdge(graph, 3, 4);
        putEdge(graph, 4, 5);

        print(graph);

    }
}
```

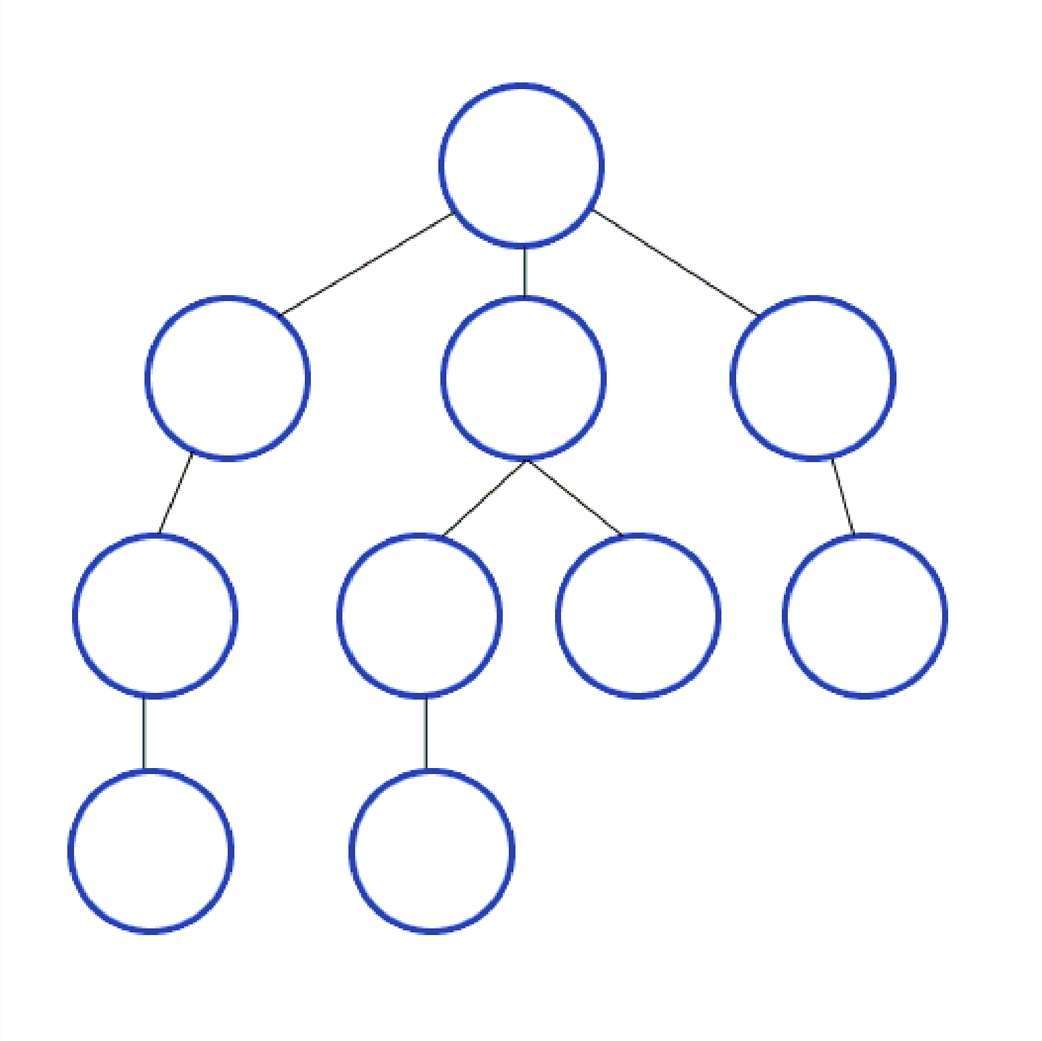
```java
public class Main {

    public static void print(ArrayList<ArrayList<Integer>> graph) {
        for (int i = 1; i < graph.size(); i++) {
            ArrayList<Integer> node = graph.get(i);
            System.out.print("node"+"["+i+"] : ");
            for (int j = 0; j < node.size(); j++)
                System.out.print(node.get(j)+ "->");
            System.out.println();
        }
    }

    public static void putEdge(ArrayList<ArrayList<Integer>> graph, int x, int y) {
        graph.get(x).add(y);
        graph.get(y).add(x);
    }

    public static void main(String[] args) {
        int n = 5; //그래프 정점의 개수
        ArrayList<ArrayList<Integer>> graph = new ArrayList<>();

        for (int i = 0; i <= n; i++)
            graph.add(new ArrayList<>()); //각 노드 별 리스트를 만들어준다.
        putEdge(graph, 1, 2);
        putEdge(graph, 1, 3);
        putEdge(graph, 1, 4);
        putEdge(graph, 2, 3);
        putEdge(graph, 2, 5);
        putEdge(graph, 3, 4);
        putEdge(graph, 4, 5);

        print(graph);

    }
}
```

CHAPTER. 3~4

그래프의 탐색
DFS / BFS

2022 AlgoLive 4th study

DFS <깊이 우선 탐색>



* 최대한 깊이 내려간 뒤, 더 이상 갈 곳이 없으면 옆으로 이동

루트 노드에서 시작하여, 다음 brranch로 넘어가기 전에 해방 분기를 완벽하게 탐색함.

예를 들어, 미로찾기를 할 때 최대한 한 방향으로 갈 수 있을 때까지 쭉 가다가 더 이상 갈 수 없게 되면 다시 가장 가까운 갈림길로 돌아와서 그 갈림길부터 다시 다른 방향으로 탐색을 진행하는 것이 깊이 우선 탐색

1. 모든 노드를 방문하고자 하는 경우에 이 방법을 선택함
2. 깊이 우선 탐색(DFS)이 너비 우선 탐색(BFS)보다 좀 더 간단함
3. 검색 속도 자체는 너비 우선 탐색(BFS)에 비해서 느림

주로 스택 또는 재귀함수로 구현

DFS 의 구현

```java
package study.blog.codingnojam;

public class Study_DFS_Recursion {

    // 방문처리에 사용 할 배열선언
    static boolean[] vistied = new boolean[9];

    // 그림예시 그래프의 연결상태를 2차원 배열로 표현
    // 인덱스가 각각의 노드번호가 될 수 있게 0번인덱스는 아무것도 없는 상태라고 생각하시면됩니다.
    static int[][] graph = {{}, {2,3,8}, {1,6,8}, {1,5}, {5,7}, {3,4,7}, {2}, {4,5}, {1,2}};

    public static void main(String[] args) {
        dfs(1);
    }

    static void dfs(int nodeIndex) {
        // 방문 처리
        vistied[nodeIndex] = true;

        // 방문 노드 출력
        System.out.print(nodeIndex + " -> ");

        // 방문한 노드에 인접한 노드 찾기
        for (int node : graph[nodeIndex]) {
            // 인접한 노드가 방문한 적이 없다면 DFS 수행
            if(!vistied[node]) {
                dfs(node);
            }
        }
    }
}
```

```java
    // 방문처리에 사용 할 배열선언
    static boolean[] vistied = new boolean[9];

    // 그림예시 그래프의 연결상태를 2차원 배열로 표현
    // 인덱스가 각각의 노드번호가 될 수 있게 0번인덱스는 아무것도 없는 상태라고 생각하시면됩니다.
    static int[][] graph = {{}, {2,3,8}, {1,6,8}, {1,5}, {5,7}, {3,4,7}, {2}, {4,5}, {1,2}};

    // DFS 사용 할 스택
    static Stack<Integer> stack = new Stack<>();

    public static void main(String[] args) {

        // 시작 노드를 스택에 넣어줍니다.
        stack.push(1);
        // 시작 노드 방문처리
        vistied[1] = true;

        // 스택이 비어있지 않으면 계속 반복
        while(!stack.isEmpty()) {

            // 스택에서 하나를 꺼냅니다.
            int nodeIndex = stack.pop();

            // 방문 노드 출력
            System.out.print(nodeIndex + " -> ");

            // 꺼낸 노드와 인접한 노드 찾기
            for (int LinkedNode : graph[nodeIndex]) {
                // 인접한 노드를 방문하지 않았을 경우에 스택에 넣고 방문처리
                if(!vistied[LinkedNode]) {
                    stack.push(LinkedNode);
                    vistied[LinkedNode] = true;
                }
            }
        }
    }
}
```

2022 AlgoLive 4th study

DFS 의 구현

```python
def dfs(graph, start_node):

    ## 기본은 항상 두개의 리스트를 별도로 관리해주는 것
    need_visited, visited = list(), list()

    ## 시작 노드를 시정하기
    need_visited.append(start_node)

    ## 만약 아직도 방문이 필요한 노드가 있다면,
    while need_visited:

        ## 그 중에서 가장 마지막 데이터를 추출 (스택 구조의 활용)
        node = need_visited.pop()

        ## 만약 그 노드가 방문한 목록에 없다면
        if node not in visited:

            ## 방문한 목록에 추가하기
            visited.append(node)

            ## 그 노드에 연결된 노드를
            need_visited.extend(graph[node])

    return visited
```

```python
def dfs2(graph, start_node):
    ## deque 패키지 불러오기
    from collections import deque
    visited = []
    need_visited = deque()

    ##시작 노드 설정해주기
    need_visited.append(start_node)

    ## 방문이 필요한 리스트가 아직 존재한다면
    while need_visited:
        ## 시작 노드를 지정하고
        node = need_visited.pop()

        ##만약 방문한 리스트에 없다면
        if node not in visited:

            ## 방문 리스트에 노드를 추가
            visited.append(node)
            ## 인접 노드들을 방문 예정 리스트에 추가
            need_visited.extend(graph[node])

    return visited

```

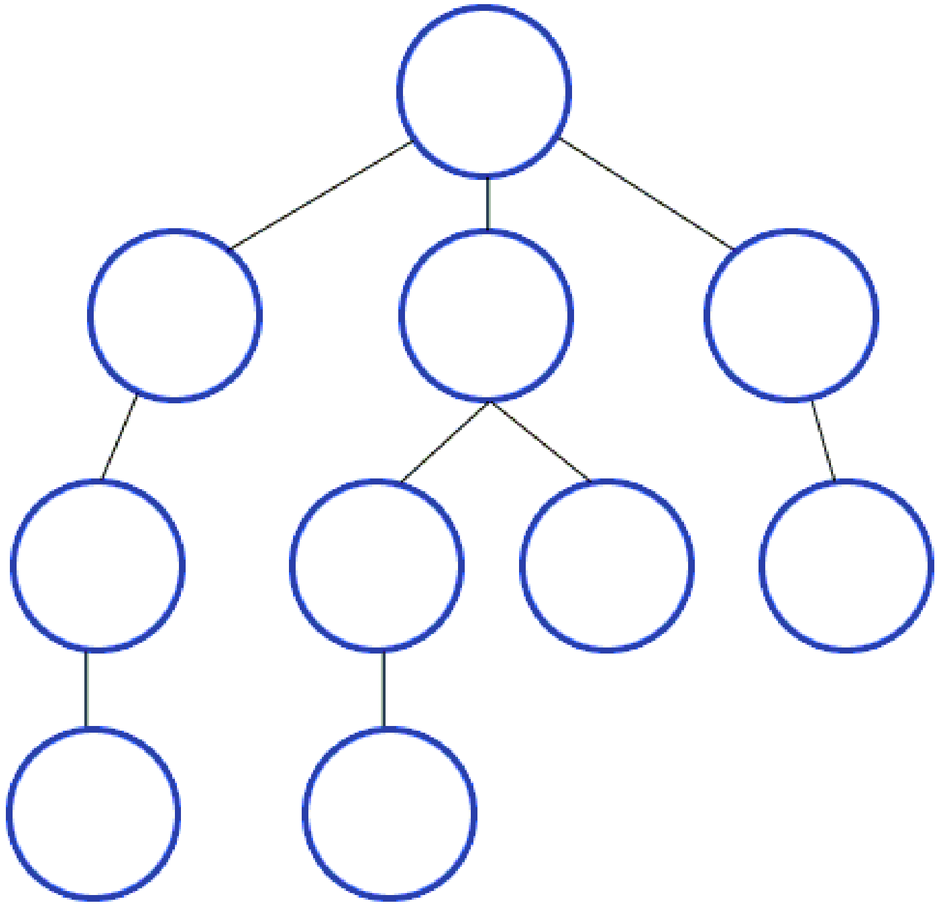
```python
def dfs_recursive(graph, start, visited = []):
## 데이터를 추가하는 명령어 / 재귀가 이루어짐
    visited.append(start)

    for node in graph[start]:
        if node not in visited:
            dfs_recursive(graph, node, visited)
    return visited
```

2022 AlgoLive 4th study

DFS 의 구현

```cpp
#include <iostream>
#include <vector>
using namespace std;

// index 0은 사용하지 않음으로 배열을 하나 더 추가
bool visited[9];
vector<int> graph[9];

void dfs(int x)
{
        visited[x] = true;
        cout << x << " ";
        for (int i = 0; i < graph[x].size(); i++) // 인접한 노드 사이즈만큼 탐색
        {
                int y = graph[x][i];
                if (!visited[y]) // 방문하지 않았으면 즉 visited가 False일 때 not을 해주면 True가 되므로 아래 dfs 실
행
            dfs(y); // 재귀적으로 방문
        }
}

int main(void)
{
    /* 위 그래프와 동일하게 정의 */
    graph[1].push_back(2);
    graph[1].push_back(3);
    graph[1].push_back(8);

    graph[2].push_back(1);
    graph[2].push_back(7);

    graph[3].push_back(1);
    graph[3].push_back(4);
    graph[3].push_back(5);

    graph[4].push_back(3);
    graph[4].push_back(5);

    graph[5].push_back(3);
    graph[5].push_back(4);

    graph[6].push_back(7);

    graph[7].push_back(2);
    graph[7].push_back(6);
    graph[7].push_back(8);

    graph[8].push_back(1);
    graph[8].push_back(7);

    dfs(1);
}
```

BFS <너비 우선 탐색>



* 최대한 넓게 이동한 뒤, 더 이상 갈 수 없을 때 아래로 내려감

루트 노드(혹은 다른 임의의 노드)에서 시작해서 인접한 노드를 먼저 탐색하는 방법으로, 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법

주로 두 노드 사이의 최단 경로를 찾고 싶을 때 이 방법을 선택

ex) 지구 상에 존재하는 모든 친구 관계를 그래프로 표현한 후 Sam과 Eddie사이에 존재하는 경로를 찾는 경우

* 깊이 우선 탐색의 경우 - 모든 친구 관계를 다 살펴봐야 할지도 모름
* 너비 우선 탐색의 경우 - Sam과 가까운 관계부터 탐색

주로 큐를 이용해서 구현

2022 AlgoLive 4th study

BFS의 구현

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

bool visited[9];
vector<int> graph[9];

// BFS 함수 정의
void bfs(int start) {
    queue<int> q;
    q.push(start); // 첫 노드를 queue에 삽입
    visited[start] = true; // 첫 노드를 방문 처리

    // 큐가 빌 때까지 반복
    while (!q.empty()) {
        // 큐에서 하나의 원소를 뽑아 출력
        int x = q.front();
        q.pop();
        cout << x << ' ';
        // 해당 원소와 연결된, 아직 방문하지 않은 원소들을 큐에 삽입
        for (int i = 0; i < graph[x].size(); i++) {
            int y = graph[x][i];
            if (!visited[y]) {
                q.push(y);
                visited[y] = true;
            }
        }
    }
}

int main(void) {
    // 노드 1에 연결된 노드 정보 저장
    graph[1].push_back(2);
    graph[1].push_back(3);
    graph[1].push_back(8);
```

```java
public class StudyBFS {

    public static void main(String[] args) {

        // 그래프를 2차원 배열로 표현해줍니다.
        // 배열의 인덱스를 노드와 매칭시켜서 사용하기 위해 인덱스 0은 아무것도 저장하지 않습니다.
        // 1번인덱스는 1번노드를 뜻하고 노드의 배열의 값은 연결된 노드들입니다.
        int[][] graph = {{}, {2,3,8}, {1,6,8}, {1,5}, {5,7}, {3,4,7}, {2}, {4,5}, {1,2}};

        // 방문처리를 위한 boolean배열 선언
        boolean[] visited = new boolean[9];

        System.out.println(bfs(1, graph, visited));
        //출력 내용 : 1 -> 2 -> 3 -> 8 -> 6 -> 5 -> 4 -> 7 ->
    }

    static String bfs(int start, int[][] graph, boolean[] visited) {
        // 탐색 순서를 출력하기 위한 용도
        StringBuilder sb = new StringBuilder();
        // BFS에 사용할 큐를 생성해줍니다.
        Queue<Integer> q = new LinkedList<Integer>();

        // 큐에 BFS를 시작 할 노드 번호를 넣어줍니다.
        q.offer(start);

        // 시작노드 방문처리
        visited[start] = true;

        // 큐가 빌 때까지 반복
        while(!q.isEmpty()) {
            int nodeIndex = q.poll();
            sb.append(nodeIndex + " -> ");
            //큐에서 꺼낸 노드와 연결된 노드를 체크
            for(int i=0; i<graph[nodeIndex].length; i++) {
                int temp = graph[nodeIndex][i];
                // 방문하지 않았으면 방문처리 후 큐에 넣기
                if(!visited[temp]) {
                    visited[temp] = true;
                    q.offer(temp);
                }
            }
        }
        // 탐색순서 리턴
        return sb.toString() ;
    }
}
```

study

BFS의 구현

```python
from collections import deque

# BFS 함수 정의
def bfs(graph, start, visited):
    # 큐(Queue) 구현을 위해 deque 라이브러리 사용
    queue = deque([start])
    # 현재 노드를 방문 처리
    visited[start] = True
    # 큐가 빌 때까지 반복
    while queue:
        # 큐에서 하나의 원소를 뽑아 출력
        v = queue.popleft()
        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)
```

문제풀이

2606번: 바이러스 (실버 3)
https://www.acmicpc.net/problem/2606

1260번: DFS와 BFS (실버 2)
https://www.acmicpc.net/problem/1260

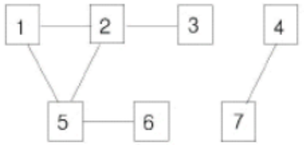
2606번: 바이러스 (실버 3)

문제풀이

문제

신종 바이러스인 웜 바이러스는 네트워크를 통해 전파된다. 한 컴퓨터가 웜 바이러스에 걸리면 그 컴퓨터와 네트워크 상에서 연결되어 있는 모든 컴퓨터는 웜 바이러스에 걸리게 된다.

예를 들어 7대의 컴퓨터가 <그림 1>과 같이 네트워크 상에서 연결되어 있다고 하자. 1번 컴퓨터가 웜 바이러스에 걸리면 웜 바이러스는 2번과 5번 컴퓨터를 거쳐 3번과 6번 컴퓨터까지 전파되어 2, 3, 5, 6 네 대의 컴퓨터는 웜 바이러스에 걸리게 된다. 하지만 4번과 7번 컴퓨터는 1번 컴퓨터와 네트워크상에서 연결되어 있지 않기 때문에 영향을 받지 않는다.



< 그림 1 >

어느 날 1번 컴퓨터가 웜 바이러스에 걸렸다. 컴퓨터의 수와 네트워크 상에서 서로 연결되어 있는 정보가 주어질 때, 1번 컴퓨터를 통해 웜 바이러스에 걸리게 되는 컴퓨터의 수를 출력하는 프로그램을 작성하시오.

입력

첫째 줄에는 컴퓨터의 수가 주어진다. 컴퓨터의 수는 100 이하이고 각 컴퓨터에는 1번 부터 차례대로 번호가 매겨진다. 둘째 줄에는 네트워크 상에서 직접 연결되어 있는 컴퓨터 쌍의 수가 주어진다. 이어서 그 수만큼 한 줄에 한 쌍씩 네트워크 상에서 직접 연결되어 있는 컴퓨터의 번호 쌍이 주어진다.

출력

1번 컴퓨터가 웜 바이러스에 걸렸을 때, 1번 컴퓨터를 통해 웜 바이러스에 걸리게 되는 컴퓨터의 수를 첫째 줄에 출력한다.

예제 입력 1 복사

예제 출력 1 복사

2606번: 바이러스 (실버 3)

문제풀이

```cpp
// Authored by : seastar105
// Co-authored by : -
// Link : http://boj.kr/f4a2942a69264111a18e39d0c384209e
#include <bits/stdc++.h>
using namespace std;
vector<vector<int>> G(105);
bool vis[105];
int N, M;

void dfs(int cur) {
    vis[cur] = true;
    for (const int &nxt : G[cur]) {
        if (!vis[nxt]) dfs(nxt);
    }
}

int main() {
    cin.tie(nullptr);
    ios::sync_with_stdio(false);
    cin >> N >> M;
    for (int i = 0; i < M; ++i) {
        int u, v;
        cin >> u >> v;
        G[u].push_back(v);
        G[v].push_back(u);
    }
    dfs(1);
    int ans = 0;
    for (int i = 1; i <= N; ++i) ans += vis[i];
    cout << ans - 1 << '\n'; // except vertex 1
    return 0;
}
```

```python
n = int(input())
m = int(input())
graph = [[]*n for _ in range(n+1)]
for _ in range(m):
    a,b = map(int,input().split())
    graph[a].append(b)
    graph[b].append(a)

cnt = 0
visited = [0]*(n+1)
def dfs(start):
    global cnt
    visited[start] = 1
    for i in graph[start]:
        if visited[i]==0:
            dfs(i)
            cnt +=1

dfs(1)
print(cnt)
```

```java
static boolean[] check;
static int[][] arr;
static int count = 0;

static int node, line;

static Queue<Integer> q = new LinkedList<>();

public static void main(String[] args) throws IOException {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));


        node = Integer.parseInt(br.readLine());
        line = Integer.parseInt(br.readLine());

        arr = new int[node+1][node+1];
        check = new boolean[node+1];

        for(int i = 0 ; i < line ; i ++) {
                StringTokenizer str = new StringTokenizer(br.readLine());

                int a = Integer.parseInt(str.nextToken());
                int b = Integer.parseInt(str.nextToken());

                arr[a][b] = arr[b][a] =  1;
        }

                dfs(1);

                System.out.println(count-1);

        }
public static void dfs(int start) {

        check[start] = true;
        count++;

        for(int i = 0 ; i <= node ; i++) {
                if(arr[start][i] == 1 && !check[i])
                        dfs(i);
        }

}
```

udy

1260번: DFS와 BFS (실버 2)

문제풀이

문제

그래프를 DFS로 탐색한 결과와 BFS로 탐색한 결과를 출력하는 프로그램을 작성하시오. 단, 방문할 수 있는 정점이 여러 개인 경우에는 정점 번호가 작은 것을 먼저 방문하고, 더 이상 방문할 수 있는 점이 없는 경우 종료한다. 정점 번호는 1번부터 N번까지이다.

입력

첫째 줄에 정점의 개수 N(1 ≤ N ≤ 1,000), 간선의 개수 M(1 ≤ M ≤ 10,000), 탐색을 시작할 정점의 번호 V가 주어진다. 다음 M개의 줄에는 간선이 연결하는 두 정점의 번호가 주어진다. 어떤 두 정점 사이에 여러 개의 간선이 있을 수 있다. 입력으로 주어지는 간선은 양방향이다.

출력

첫째 줄에 DFS를 수행한 결과를, 그 다음 줄에는 BFS를 수행한 결과를 출력한다. V부터 방문된 점을 순서대로 출력하면 된다.

예제 입력 1 복사

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

예제 출력 1 복사

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

예제 입력 2 복사

예제 출력 2 복사

1260번: DFS와 BFS (실버 2)

```python
from collections import deque
import sys
read = sys.stdin.readline

def bfs(v):
  q = deque()
  q.append(v)
  visit_list[v] = 1
  while q:
    v = q.popleft()
    print(v, end = " ")
    for i in range(1, n + 1):
      if visit_list[i] == 0 and graph[v][i] == 1:
        q.append(i)
        visit_list[i] = 1

def dfs(v):
  visit_list2[v] = 1
  print(v, end = " ")
  for i in range(1, n + 1):
    if visit_list2[i] == 0 and graph[v][i] == 1:
      dfs(i)

n, m, v = map(int, read().split())

graph = [[0] * (n + 1) for _ in range(n + 1)]
visit_list = [0] * (n + 1)
visit_list2 = [0] * (n + 1)

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

dfs(v)
print()
bfs(v)
```

문제풀이

```java
        public static void DFS(int node) {
                Dvisit[node] = true;
                System.out.print(node + " ");

                for(int i = 1; i <= N; i++) {
                        if(!Dvisit[i] && Dgraph[node][i] == 1) {
                                DFS(i);
                        }
                }
        }

        public static void BFS(int node) {
                boolean[] Bvisit = new boolean[10001];
                Queue<Integer> que = new LinkedList<Integer>();
                Bvisit[node] = true;
                que.offer(node);

                while(!que.isEmpty()) {
                        int P = que.poll();
                        System.out.print(P + " ");

                        for(int i = 1; i <= N; i++) {
                                if(!Bvisit[i] && Bgraph[P][i] == 1) {
                                        Bvisit[i] = true;
                                        que.offer(i);
                                }
                        }
                }
        }

}
```

```cpp
//백준1260 DFS와BFS

#include <iostream>
#include <queue>
using namespace std;
#define MAX 1001

int N, M, V; //정점개수, 간선개수, 시작정점
int map[MAX][MAX]; //인접 행렬 그래프
bool visited[MAX]; //정점 방문 여부
queue<int> q;

void reset() {
    for (int i = 1; i <= N; i++) {
        visited[i] = 0;
    }
}

void DFS(int v) {
    visited[v] = true;
    cout << v << " ";

    for (int i = 1; i <= N; i++) {
        if (map[v][i] == 1 && visited[i] == 0) { //현재 정점과 연결되어있고 방문되지 않았으면
            DFS(i);
        }
    }
}

void BFS(int v) {
    q.push(v);
    visited[v] = true;
    cout << v << " ";

    while (!q.empty()) {
        v = q.front();
        q.pop();

        for (int w = 1; w <= N; w++) {
            if (map[v][w] == 1 && visited[w] == 0) { //현재 정점과 연결되어있고 방문되지 않았으면
                q.push(w);
                visited[w] = true;
                cout << w << " ";
            }
        }
    }
}

int main() {
    cin >> N >> M >> V;

    for (int i = 0; i < M; i++) {
        int a, b;
        cin >> a >> b;
        map[a][b] = 1;
        map[b][a] = 1;
    }

    reset();
    DFS(V);

    cout << '\n';

    reset();
    BFS(V);

    return 0;
}
```

감사합니다

2022 AlgoLive 4th study