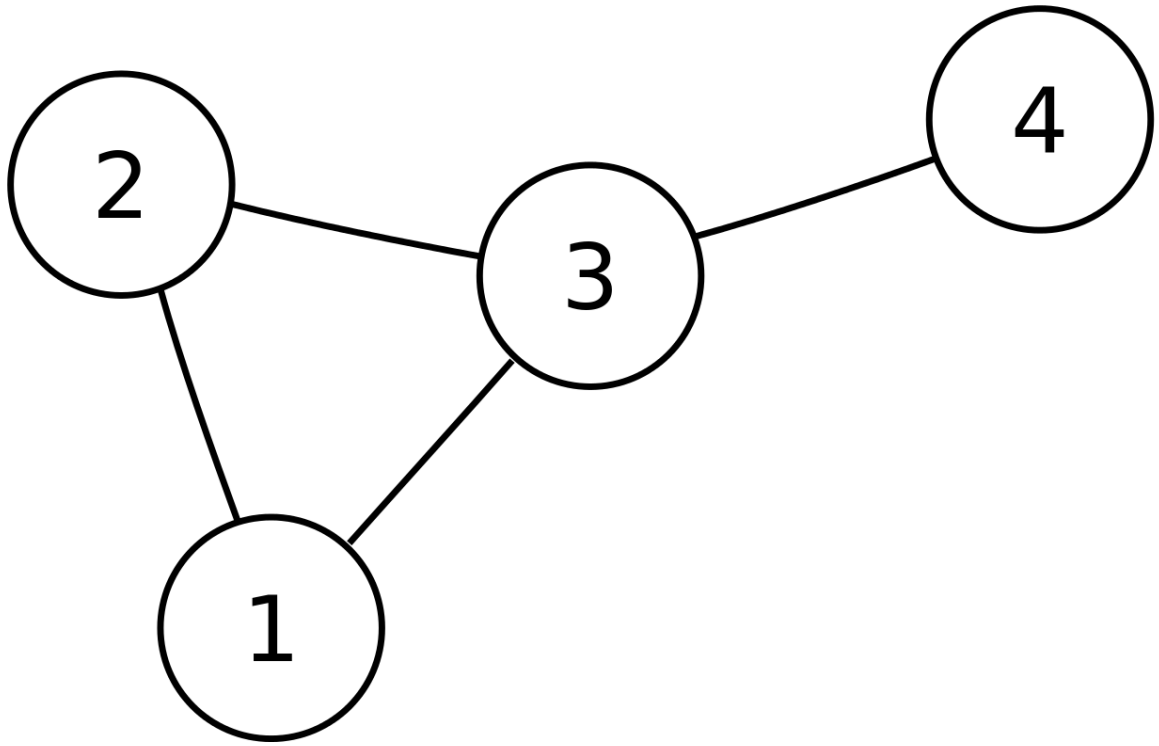
그래프, DFS

그래프, DFS

그래프

그래프

정점(Node)와 그 정점들을 잇는 간선(Edge)로 구성된 자료구조



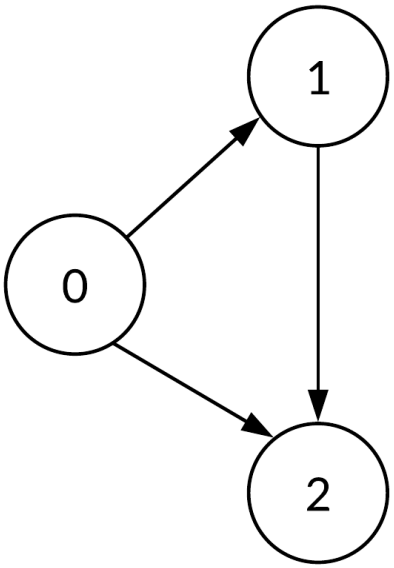
그래프 용어 정리

- **정점(Node, Verticle)** : 데이터가 담기는 지점
  - **인접 정점(Adjacent Node)**: 간선에 의해 연결된 정점
- **간선(Edge)** : 노드와의 관계를 표현
- **차수(Degree)**: 하나의 정점에 인접한 정점의 수
- **가중치(Weight)**: 간선이 가지는 고유값 (거리, 비용, 시간 등등)

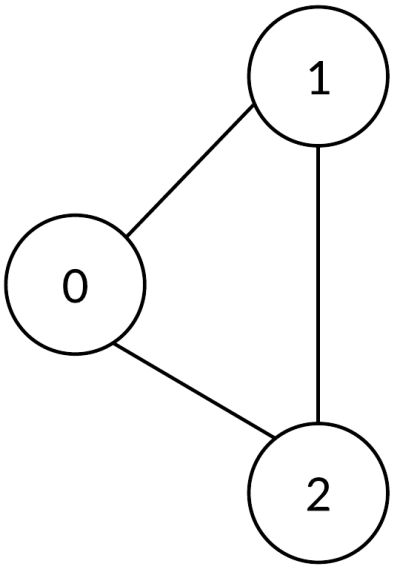
그래프의 종류

- 방향 그래프
  - **단방향** 그래프
  - **양방향(무방향)** 그래프

Directed Graph

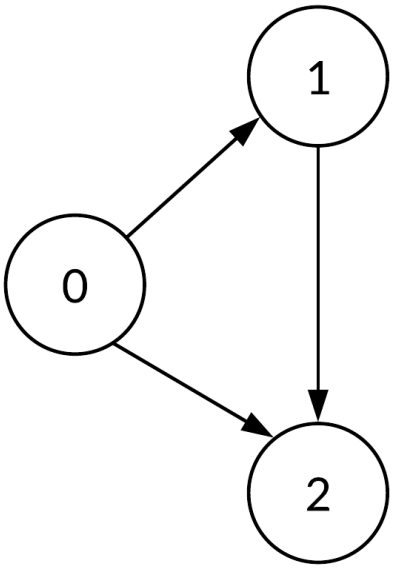


Undirected Graph

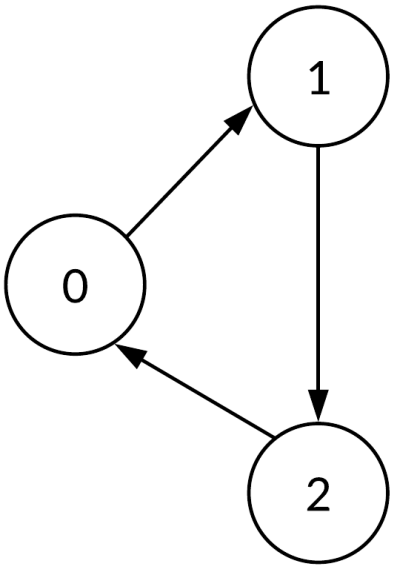


- 사이클 그래프
  - **순환(Cycle)** 그래프
  - **비순환(Acycle)** 그래프

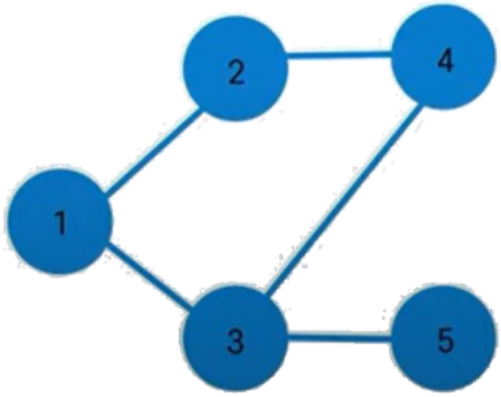
Acyclic Graph



Cyclic Graph



그래프의 구현



인접행렬을 이용한 구현

- 정점의 개수 N개에 대해 N*N 배열을 이용하여 표현
  - N[i][j]: i번 노드와 j번 노드를 잇는 간선의 여부

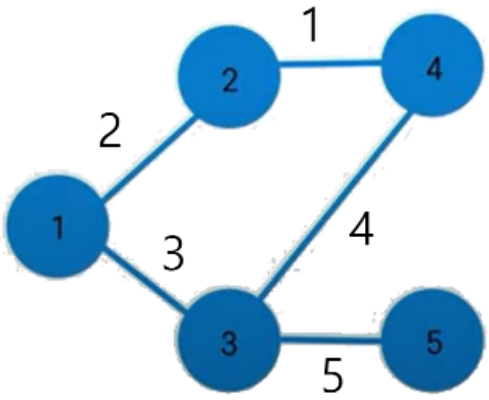
```
adj_matrix = [[0, 1, 1, 0, 0],
[1, 0, 0, 1, 0],
[1, 0, 0, 1, 1],
[0, 1, 1, 0, 0],
[0, 0, 1, 0, 0]]
```

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	1
4	0	1	1	0	0
5	0	0	1	0	0

- 가중치가 있는 경우

가중치가 있는 경우
? : 절대 나올 수 없는 수

	1	2	3	4	5
1	?	2	3	?	?
2	2	?	?	1	?
3	1	?	?	1	1
4	?	1	1	?	?
5	?	?	1	?	?



인접리스트를 이용한 구현

- V[i] = [i와 연결되어 있는 모든 정점]

```
adj_list = [[],
[2, 3],
[1, 4],
[1, 4, 5],
[2, 3],
[3]]
```

인접행렬 vs 인접리스트

- 일반적으로는 인접행렬이 좋다
- 그러나!
  - 특정 정점과 정점의 연결관계를 자주 확인해야 하는 경우: 인접 행렬
  - 연결된 정점들을 탐색해야 하는 경우: 인접 리스트

DFS

DFS (Depth First Search)

- 시작 **정점의** 한 방향으로 **갈 수 있는 경로가 있는 곳까지 깊이 탐색**해 가다가 더 이상 갈 곳이 없게 되면, **가장 마지막에 만났던 갈림길 간선이 있는 정점**으로 되돌아와서 **다른 방향의 정점**으로 탐색을 계속 반복하여 결국 모든 정점을 순회

- 가장 마지막에 만났던 갈림길의 정점으로 되돌아가서 다시 깊이 우선 탐색을 반복해야 하므로 후입선출 구조의 **스택 사용**

DFS 예시(그래프)

- 재귀로 구현

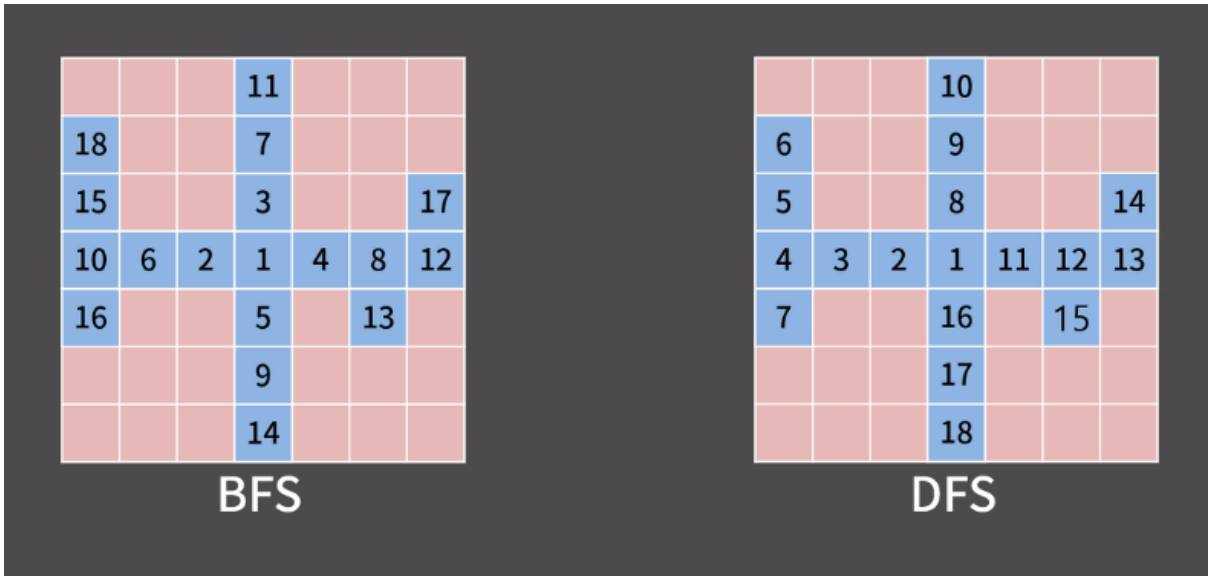
```
visited = [False] * 9
adj_list = [[],
[2, 3, 8],
[1, 7],
[1, 4, 5],
[3, 5],
[3, 4],
[7],
[2, 6, 8],
[1, 7]]

def dfs(adj_list, s):
    global visited
    # 현재노드 방문
    visited[s] = True
    print(s, end=' ')
    # 현재 노드의 인접 노드 재귀적으로 방문
    for i in adj_list[s]:
        if not visited[i]:
            dfs(adj_list, i)

dfs(adj_list, 1) # 1 2 7 6 8 3 4 5
```

- 반복문으로 구현

```
visited = [False] * 9
stack = []
adj_list = [[],
[2, 3, 8],
[1, 7],
[1, 4, 5],
[3, 5],
[3, 4],
[7],
[2, 6, 8],
[1, 7]]

def dfs(adj_list, s):
    global visited
    global stack

    # 출발점 방문
    visited[s] = True
    print(s, end=' ')

    while True:
        # 인접노드 탐색
        for w in adj_list[s]:
            # 미방문 인접노드 있을 시 해당 노드로 이동
            if visited[w] == 0:
                stack.append(s)
                s = w
                visited[w] = 1
                print(s, end=' ')
                break
        # 미방문 인접노드 없을 시
        else:
            # 미방문 인접노드가 있는 노드까지 거슬러 올라가서 다시 탐색한다.
            if stack:
```

```
                s = stack.pop()
            # 경로에 가능한 모든 노드를 탐색했을 경우 탐색을 중단한다.
            else:
                break
    return

dfs(adj_list, 1) # 1 2 7 6 8 3 4 5
```

BFS vs DFS



- 대부분의 경우(일반적인 기업 코테)는 어느 것을 선택해도 무방한 경우가 많음!

1. DFS & BFS

   단순히 모든 정점을 방문하는 것이 중요한 문제인 경우

2. DFS
   - 경로의 특징을 저장해둬야할 경우
   - 모든 경우를 하나하나 다 탐색해야할 경우 (조합, 순열 등등)
   - 백트래킹
3. BFS
   - 거리(깊이)를 계산해야 하는 경우
   - 미로 찾기, 최단 거리 등등

예제: 단지 번호 붙이기 (BOJ 2667)

https://www.acmicpc.net/problem/2667

- 2차원 배열에서의 DFS?
- 출발점이나 도착점이 주어지지 않았을 때?

문제

<그림 1>과 같이 정사각형 모양의 지도가 있다. 1은 집이 있는 곳을, 0은 집이 없는 곳을 나타낸다. 철수는 이 지도를 가지고 연결된 집의 모임인 단지를 정의하고, 단지에 번호를 붙이려 한다. 여기서 연결되었다는 것은 어떤 집이 좌우, 혹은 아래위로 다른 집이 있는 경우를 말한다. 대각선상에 집이 있는 경우는 연결된 것이 아니다. <그림 2>는 <그림 1>을 단지별로 번호를 붙인 것이다. 지도를 입력하여 단지수를 출력하고, 각 단지에 속하는 집의 수를 오름차순으로 정렬하여 출력하는 프로그램을 작성하시오.

0	1	1	0	1	0	0
0	1	1	0	1	0	1
1	1	1	0	1	0	1
0	0	0	0	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

<그림 1>

0	1	1	0	2	0	0
0	1	1	0	2	0	2
1	1	1	0	2	0	2
0	0	0	0	2	2	2
0	3	0	0	0	0	0
0	3	3	3	3	3	0
0	3	3	3	0	0	0

<그림 2>

입력

첫 번째 줄에는 지도의 크기 N(정사각형이므로 가로와 세로의 크기는 같으며 5≤N≤25)이 입력되고, 그 다음 N줄에는 각각 N개의 자료(0혹은 1)가 입력된다.

입력

첫 번째 줄에는 지도의 크기 N(정사각형이므로 가로와 세로의 크기는 같으며 5≤N≤25)이 입력되고, 그 다음 N줄에는 각각 N개의 자료(0혹은 1)가 입력된다.

출력

첫 번째 줄에는 총 단지수를 출력하시오. 그리고 각 단지내 집의 수를 오름차순으로 정렬하여 한 줄에 하나씩 출력하시오.

예제 입력 1 복사

```
7
0110100
0110101
1110101
0000111
0100000
0111110
0111000
```

예제 출력 1 복사

```
3
7
8
9
```

▼ 정답코드

```
# 220824 2667 단지번호붙이기

import sys

input = sys.stdin.readline

# N: 지도의 크기, map: 지도, visited: 방문여부
N = int(input())
map = [list(map(int, input().rstrip())) for _ in range(N)]
visited = [[False for _ in range(N)] for _ in range(N)]


# 델타 탐색 방향 설정
dx = [-1, 1, 0, 0]
dy = [0, 0, 1, -1]
```

```
# DFS 함수 정의
def dfs(x, y):
    # 글로벌 변수 선언
    global cnt

    # 방문표시
    visited[x][y] = True

    # 단지에 집이 있으면 세준다.
    if map[x][y]:
        cnt += 1

    # 델타 탐색 시행
    for i in range(4):
        nx = x + dx[i]
        ny = y + dy[i]
        # 탐색 가능하고, 방문하지 않았고 연결된 집이 있는 경우(같은 단지)
        if 0 <= nx < N and 0 <= ny < N:
            if not visited[nx][ny] and map[nx][ny] == 1:
                dfs(nx, ny)

# cnt: 단지에 포함된 집의 개수, house: 각 단지 내 집의 개수
cnt = 0
house = []

# 전체를 돌면서 집이 있고(1), 방문하지 않은 곳으로부터 탐색을 시작한다.
for i in range(N):
    for j in range(N):
        if map[i][j] == 1 and not visited[i][j]:
            dfs(i, j)
            house.append(cnt)
            cnt = 0

# 오름차순 정렬
house.sort()
# 총 단지 수와 단지 내 집의 개수 출력
print(len(house))
for i in house:
    print(i)
```

Reference

https://github.com/dev-dain/Dukgorithm/blob/master/week5/그래프 이론과 구현%2C DFS과 BFS.pdf

https://github.com/VSFe/Algorithm_Study/blob/main/Concept/Prev/vol.2/06_Graph_Traversal/Ch.06_그래프_탐색.pdf

https://blog.encrypted.gg/942?category=773649

https://foameraserblue.tistory.com/188?category=481823