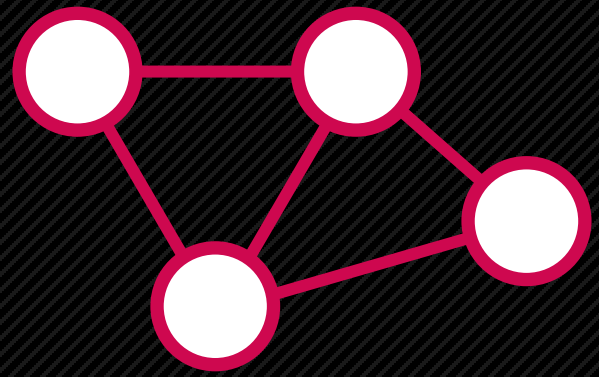
알고리즘 특강
그래프 탐색

코테 단골손님 중 하나인 그래프입니다.
DFS/BFS는 정말 능숙하게 사용할 줄 알아야 합니다!

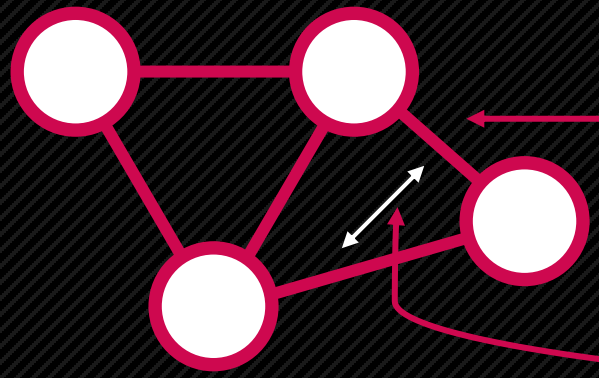
그래프



Graph

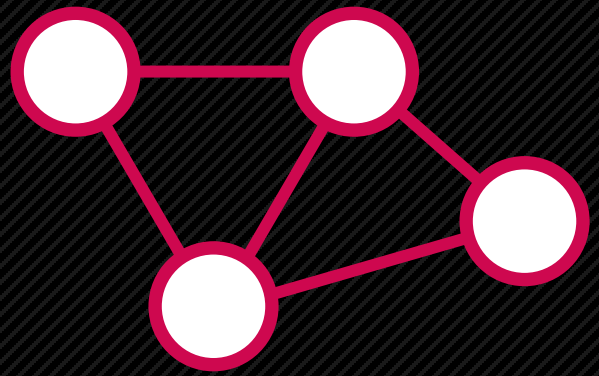
- 노드와 그 노드를 잇는 간선을 하나로 모아 놓은 자료구조.

용어 설명

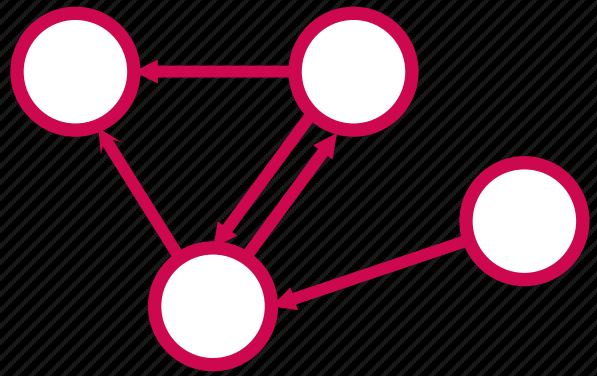


- **간선 (Edge)**
  노드와의 관계를 표현.
- **정점 (Vertex, Node)**
  데이터가 담기는 지점.
- **차수 (Degree)**
  하나의 정점에 인접한 정점의 수

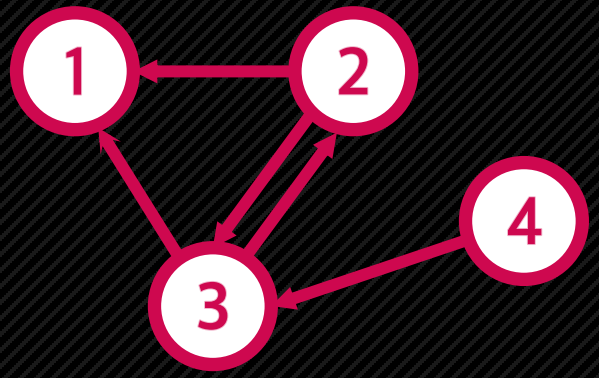
무방향/방향



- **무방향 그래프**
  간선의 방향이 없음.



- **방향 그래프**
  간선에 특정한 방향이 있음.

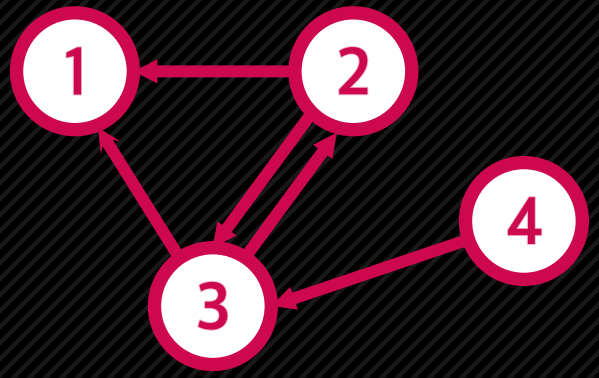


	1	2	3	4
1	0	0	0	0
2	1	0	1	0
3	1	1	0	0
4	0	0	1	0

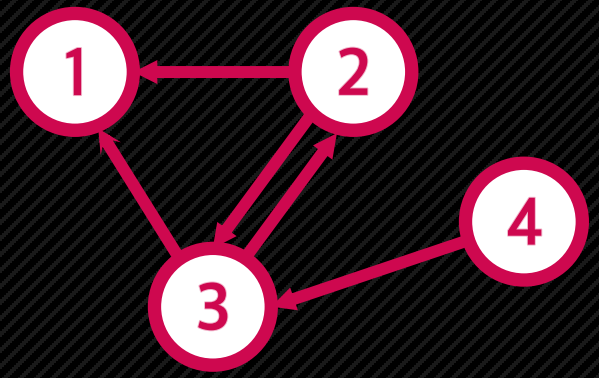
Adjust Array

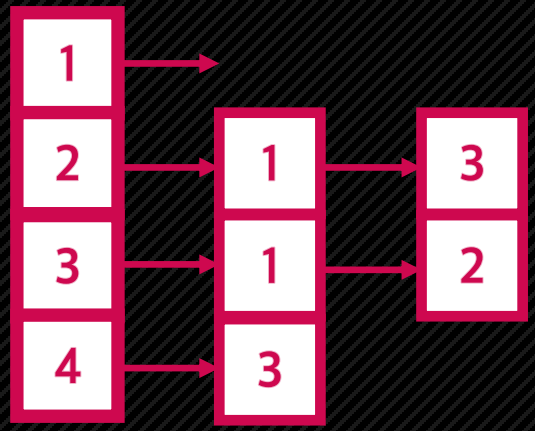
- N * N 크기의 **2차원 배열**을 생성해서 연결 상태를 나타내는 방법.
- 특정 노드 N_i와 N_j가 **연결되어 있는지** 확인: O(1)
- 특정 노드와 연결되어 있는 **모든 노드를 확인**: O(N)
- 공간 복잡도: O(N * N)

구현




```
graph = [[0 for i in range(5)] for j in range(5)]
graph[2][1] = 1
graph[2][3] = 1
graph[3][1] = 1
graph[3][2] = 1
graph[4][3] = 1
```

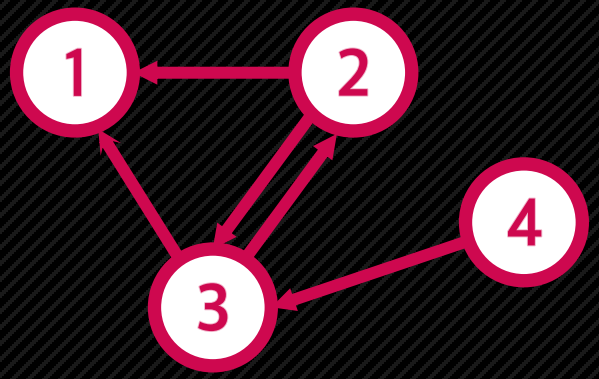




Adjust List

- N개의 **리스트**를 생성해서 연결 상태를 나타내는 방법.
- 특정 노드 N_i와 N_j가 **연결되어 있는지** 확인: $O(\min(\text{Degree}(N_i), \text{Degree}(N_j))$
- 특정 노드와 연결되어 있는 **모든 노드를 확인**: $O(\text{Degree}(N_i))$
- 공간 복잡도: $O(M)$

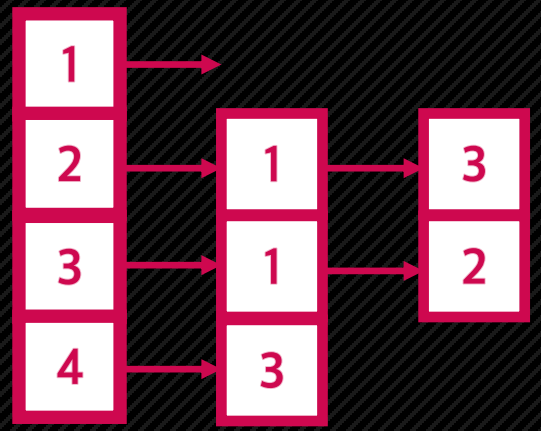
구현




```
graph = [[] for i in range(5)]
graph[2].append(1)
graph[2].append(3)
graph[3].append(1)
graph[3].append(2)
graph[4].append(3)
```

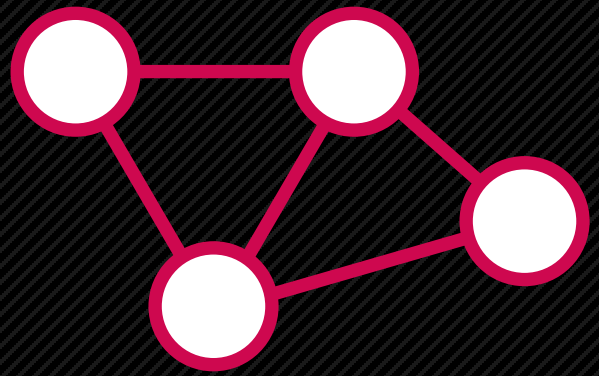
뭘 쓰면 좋을까?

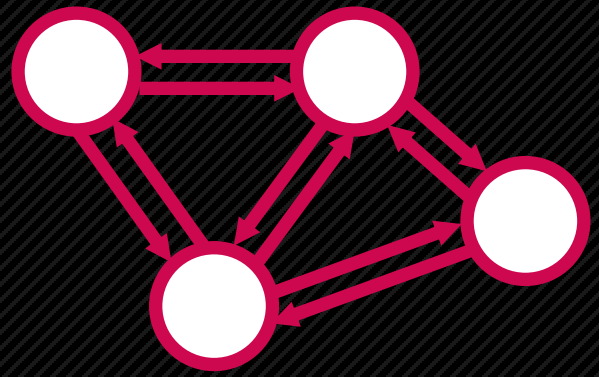
	1	2	3	4
1	0	0	0	0
2	1	0	1	0
3	1	1	0	0
4	0	0	1	0



- 특정 정점과 정점의 연결 관계를 많이 확인해야 하는 경우: 인접 행렬
- 연결된 정점들을 탐색해야 하는 경우: 인접 리스트

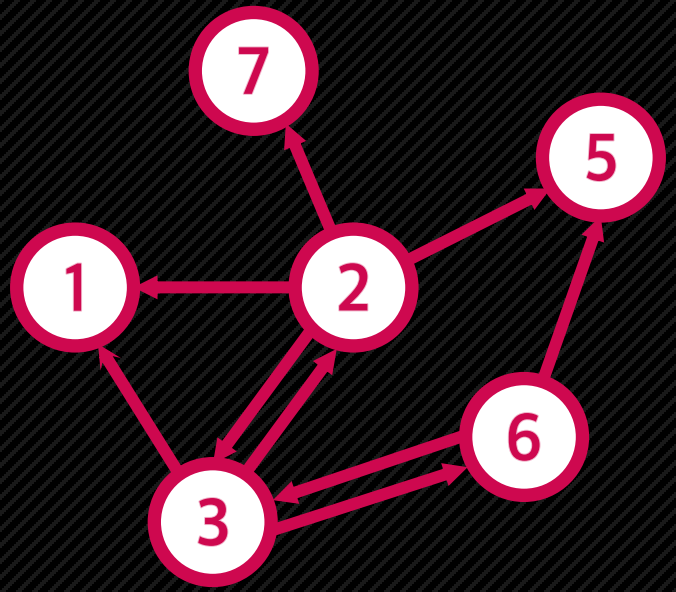
구현에서의 무방향과 방향





- 결국, 무방향 그래프는 모든 간선이 왕복간선 → 양방향을 다 만들어주면 된다!

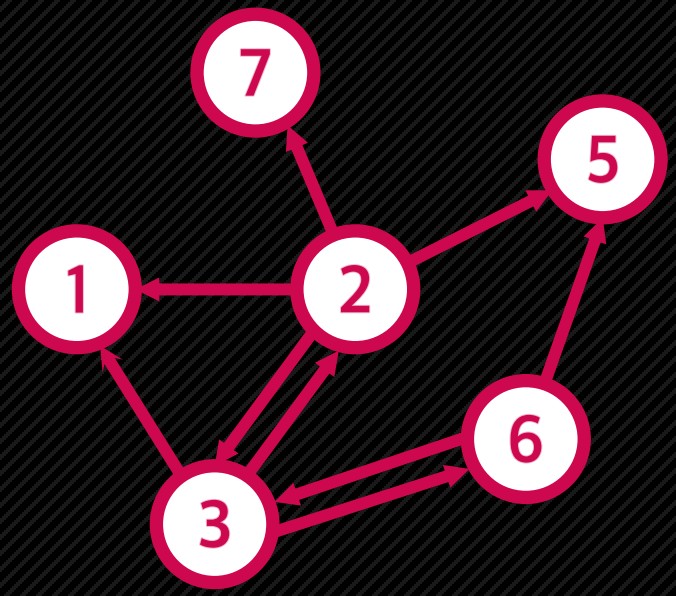
그래프 탐색



Graph Traversal

- 그래프의 모든 노드를 탐색하기 위해 간선을 따라 순회하는 것.
- 탐색 방법에 따라 BFS (Breadth First Search), DFS (Depth First Search)로 나뉜다.
- 그래프의 다양한 응용문제는 탐색을 기반으로 이뤄짐!

그래프 탐색



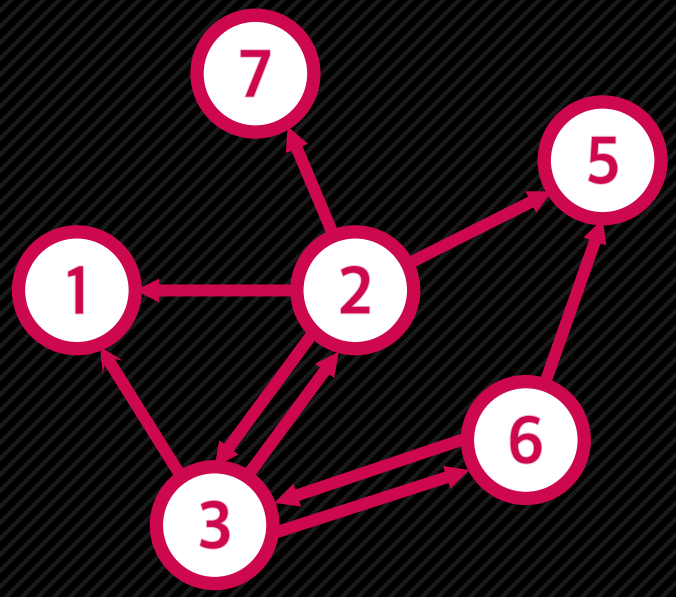
Breadth First Search

- 너비 우선 탐색이라고도 하며, 자신의 자식들부터 순차적으로 탐색.
- 순차 탐색 이후 다른 자식 노드의 자식을 확인하기 위해 큐를 사용.

Depth First Search

- 깊이 우선 탐색이라고도 하며, 최대한 깊게 탐색 후 빠져 나옴.
- 백트래킹의 일종이며, 재귀를 활용함.

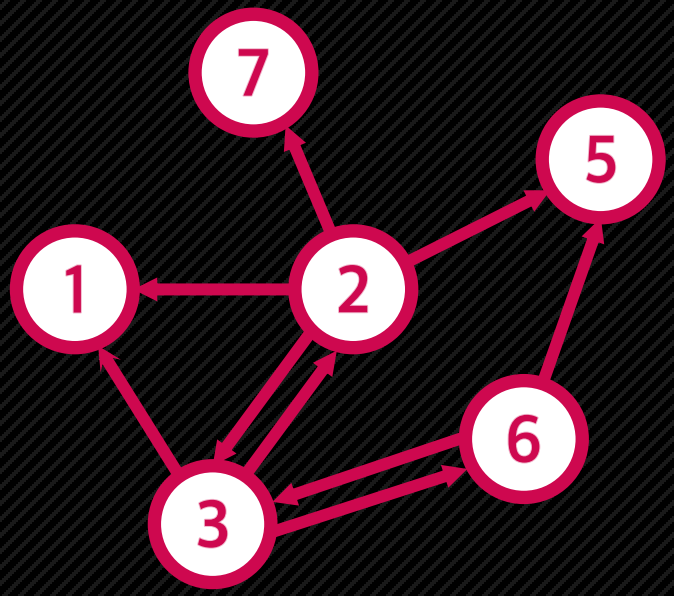
그래프 탐색




```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}

function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

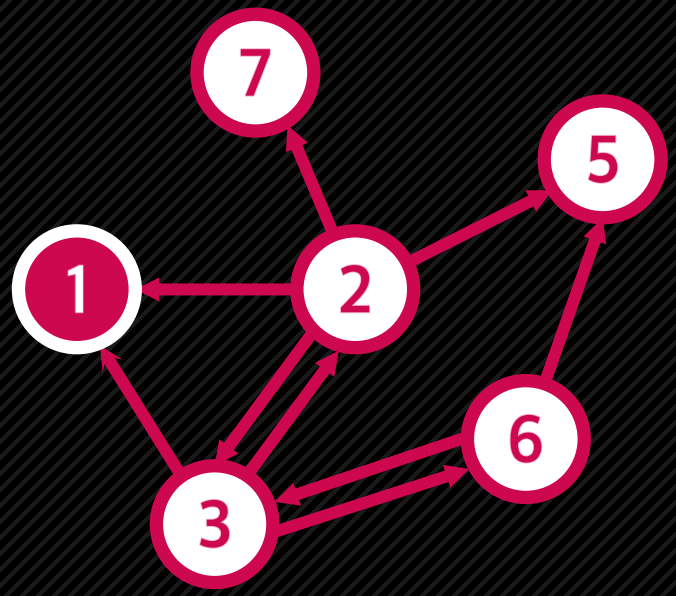
BFS




```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

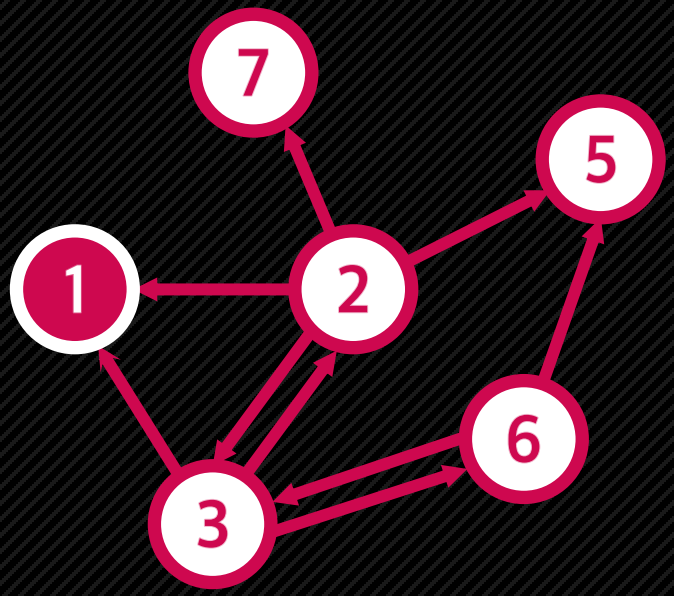
유의: 탐색을 꼭 1번부터 할 필요는 없습니다. 다만 일반적인 문제에선 특정 노드부터 방문할 것을 암시적으로 드러냅니다!

BFS



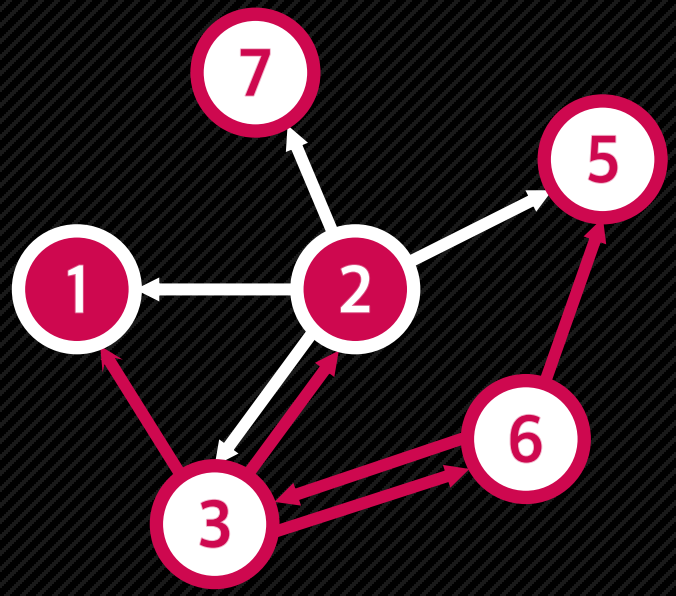

```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

BFS



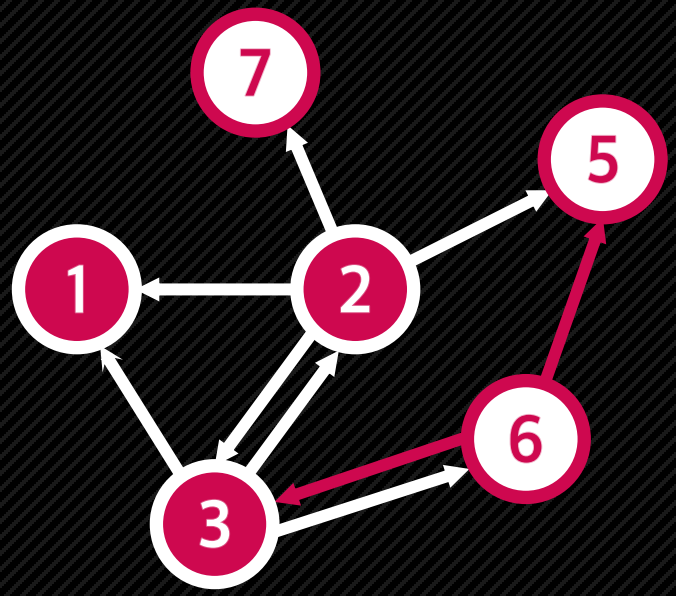

```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

BFS



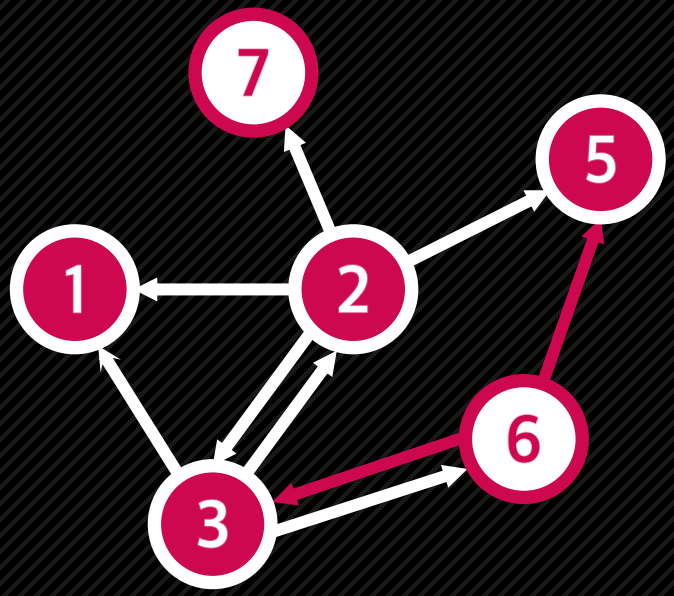

```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

BFS



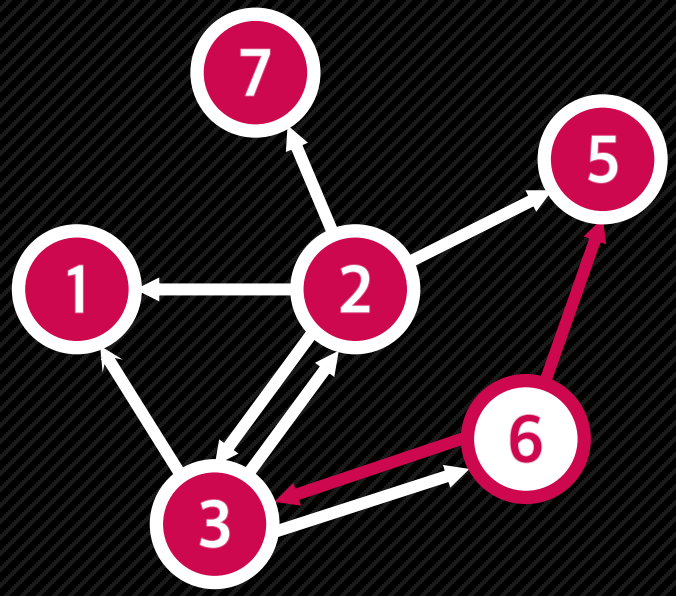

```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

BFS



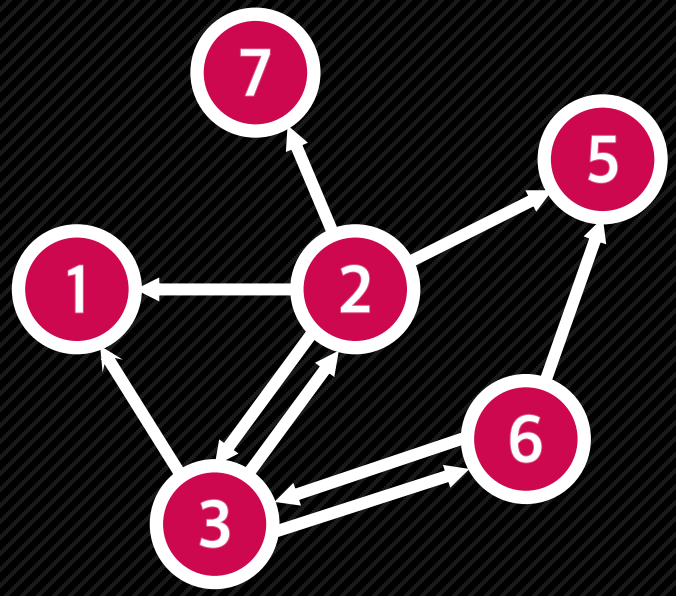

```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

BFS



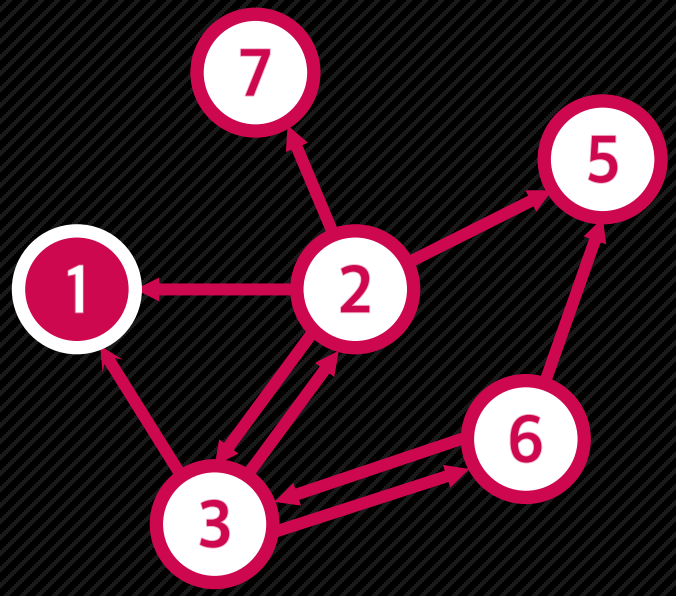

```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

BFS




```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

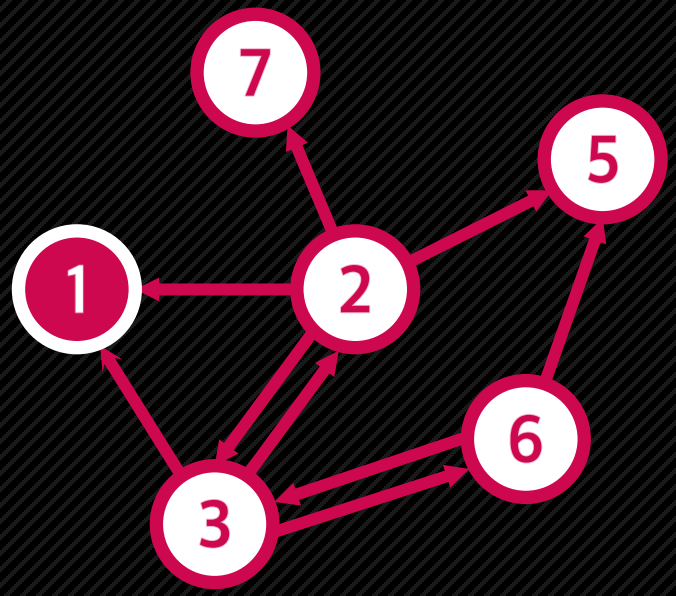
DFS




```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

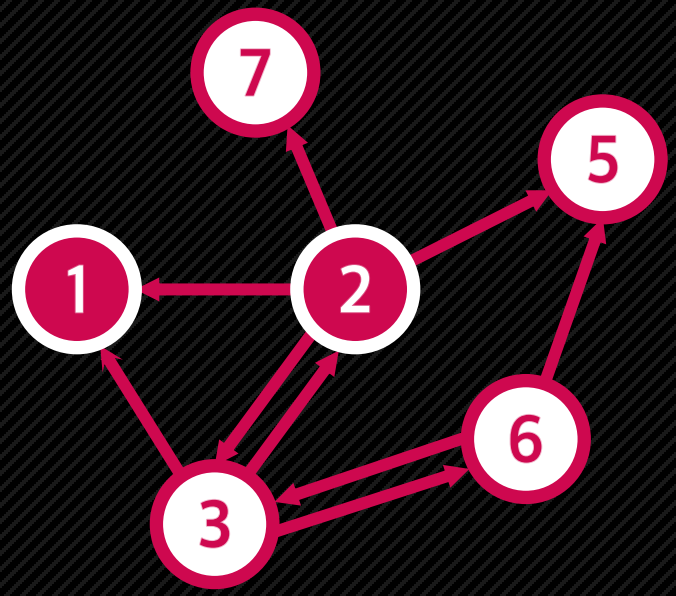
유의: 탐색을 꼭 1번부터 할 필요는 없습니다. 다만 일반적인 문제에선 특정 노드부터 방문할 것을 암시적으로 드러냅니다!

DFS

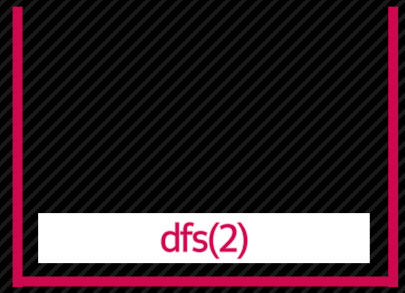



```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

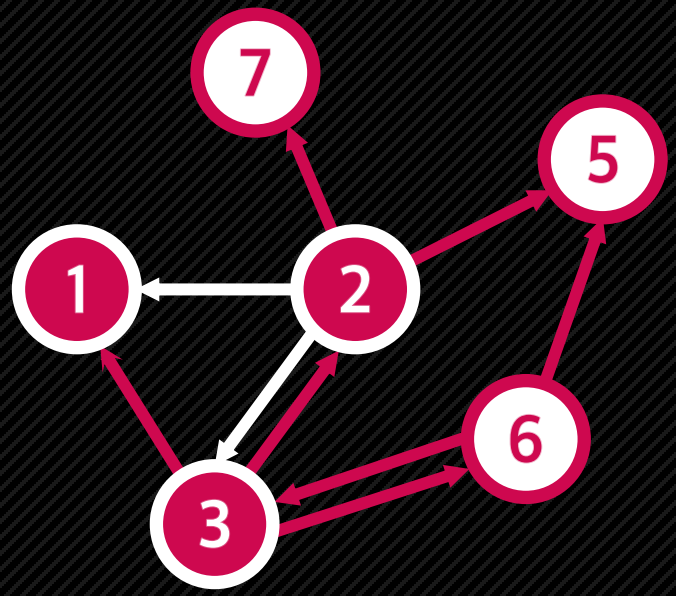
DFS




```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

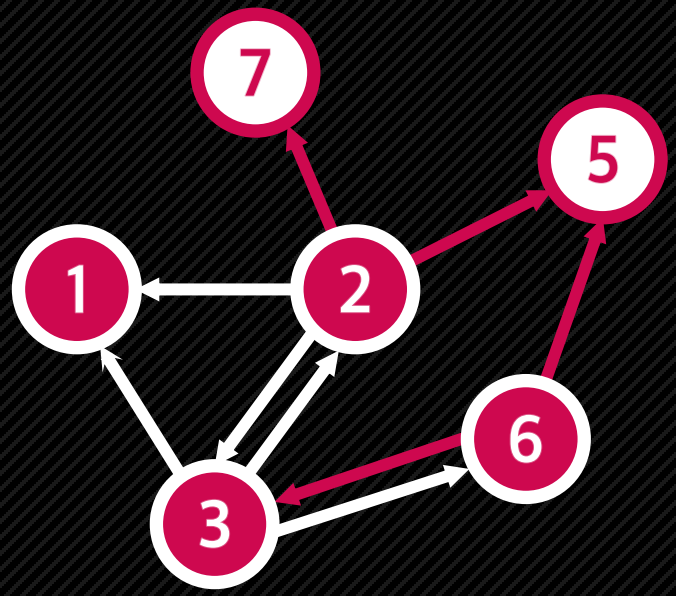


DFS



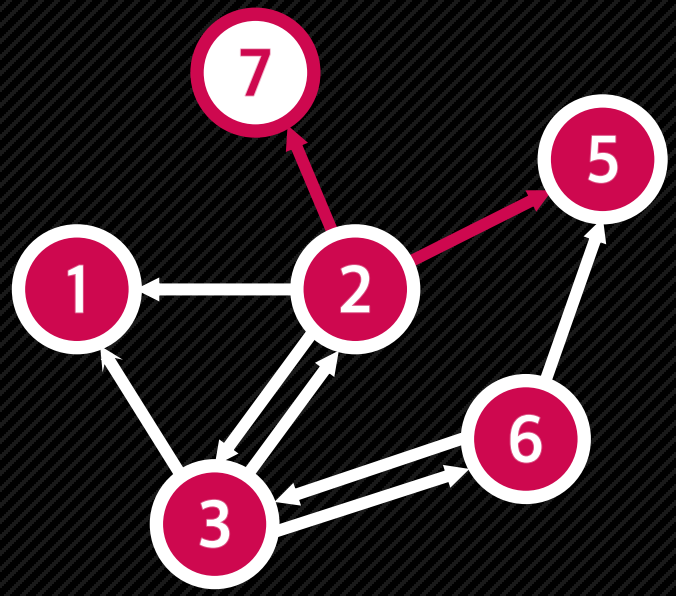

```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

DFS



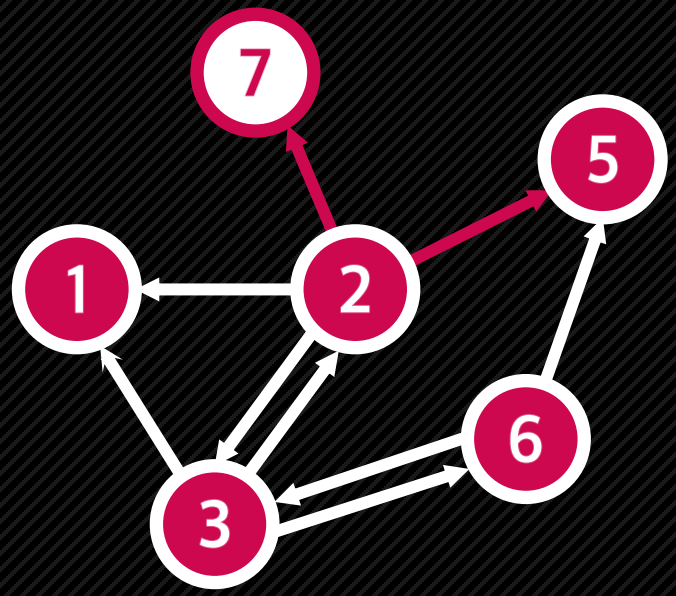

```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

DFS



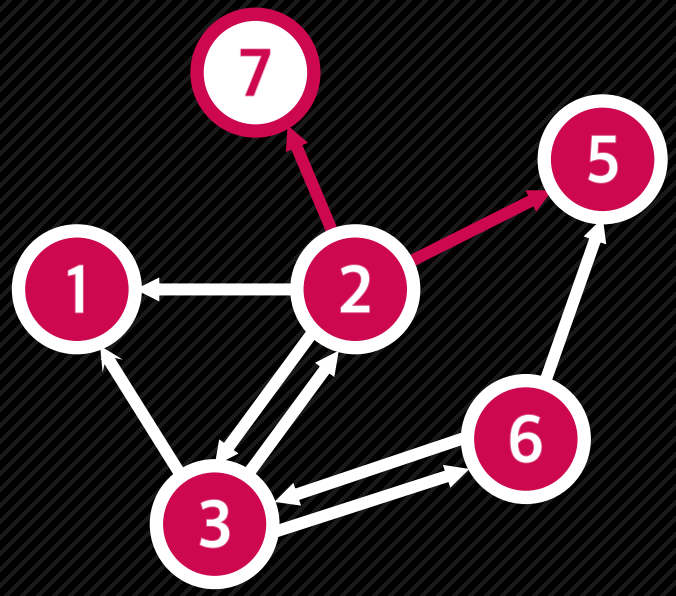

```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

DFS

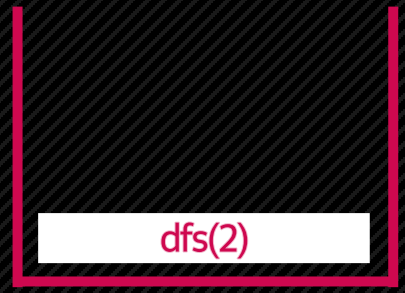



```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

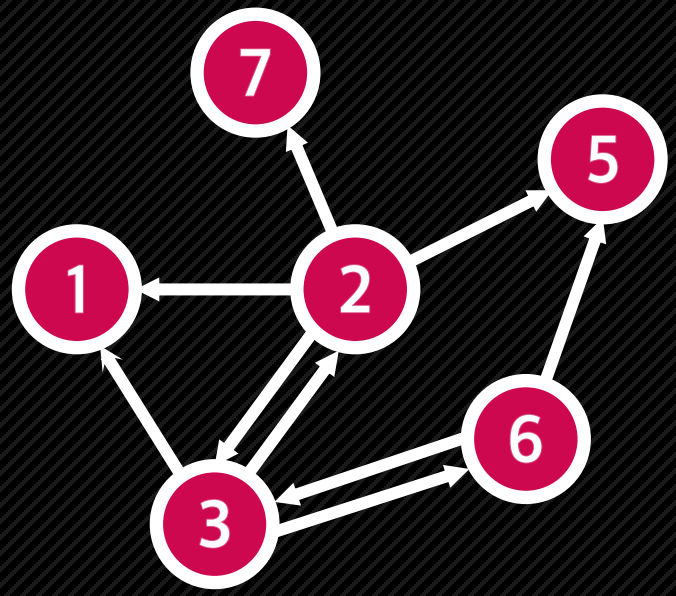
DFS




```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

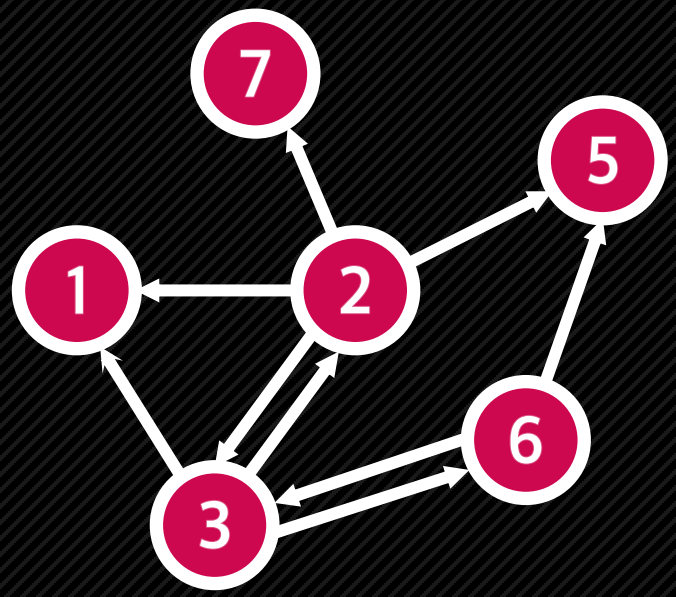


DFS




```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

DFS




```
function dfs(pos) {
    visit pos
    for children of pos
        if each child has not been visited
            dfs(pos)
}
```

연습해보자!

Silver 2 - BFS와 DFS (#1260)

요약

- 그래프를 BFS와 DFS로 탐색한 결과를 출력하는 프로그램을 작성하시오.
- 단, 방문할 수 있는 정점이 여러 개 인 경우 번호가 작은 정점을 먼저 방문한다.

제약조건

- 정점의 수의 범위는 1 <= N <= 1,000 이다.
- 간선의 수의 범위는 1 <= M <= 10,000 이다.

유형을 하나씩 알아보자!

Silver 3 - 바이러스 (#2606)

요약

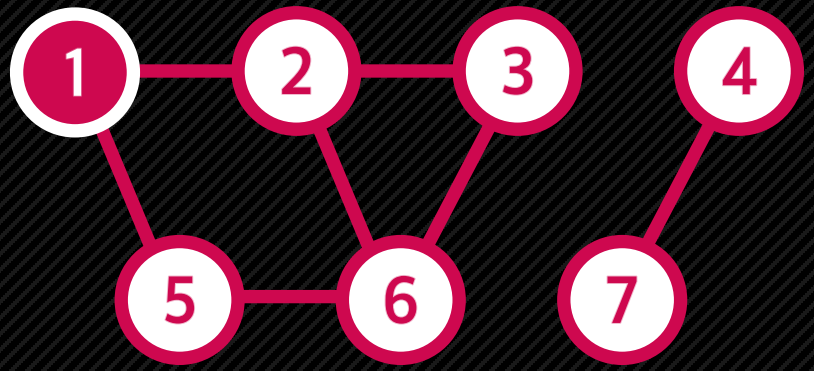
- 한 컴퓨터가 웜 바이러스에 걸리면 네트워크에 연결된 모든 컴퓨터가 바이러스에 걸린다.
- 컴퓨터 수와 네트워크 상에 서로 연결된 정보가 주어질 때, 1번을 통해 감염된 컴퓨터의 수를 구하는 프로그램을 작성하시오.

제약조건

- 컴퓨터의 수는 100 이하이다.

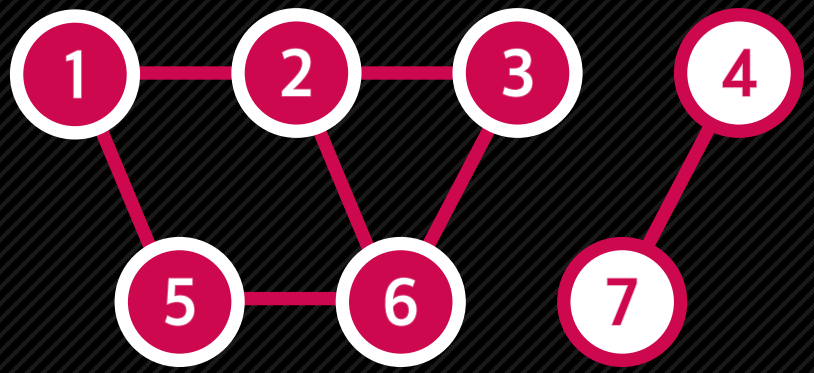
유형을 하나씩 알아보자!

Silver 3 - 바이러스 (#2606)



유형을 하나씩 알아보자!

Silver 3 - 바이러스 (#2606)



Silver 2 - 연결 요소의 개수 (#11724)

요약

- 방향 없는 그래프가 주어졌을 때, 연결 요소의 개수를 구하는 프로그램을 작성하시오.

Silver 1 - 단지번호 붙이기 (#2667)

요약

- 집이 있는 곳을 1, 집이 없는 곳을 0으로 하는 그래프가 주어진다.
- 상하좌우에 다른 집이 있는 경우, 그 집은 붙어있다고 한다.
- 이때, 붙어있는 단지의 수와 각 단지 마다 속한 집의 수를 출력하는 프로그램을 작성하시오.

제약조건

- 그래프의 가로/세로 크기는 25 이하이다.

그래프인데 배열?

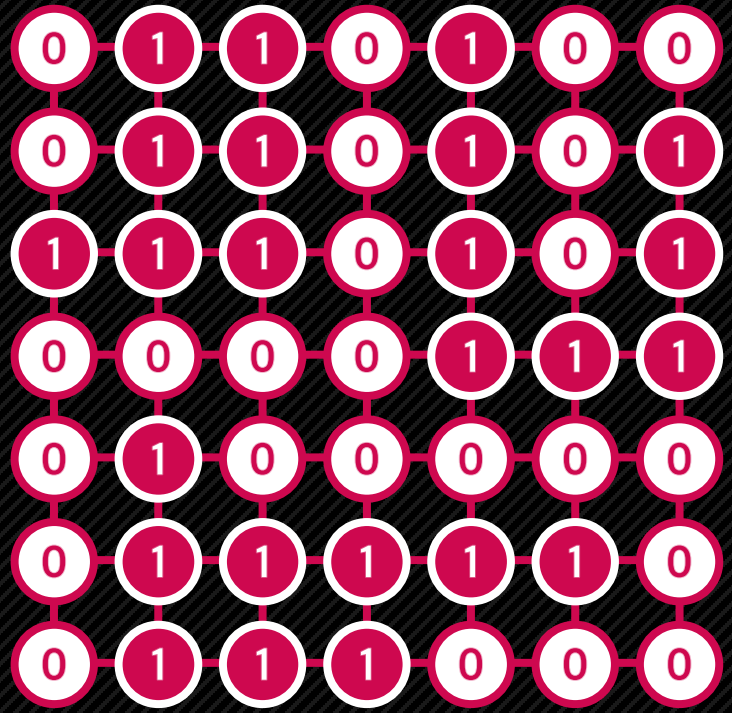
Silver 1 - 단지번호 붙이기 (#2667)

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

그래프인데 배열?

Silver 1 - 단지번호 붙이기 (#2667)

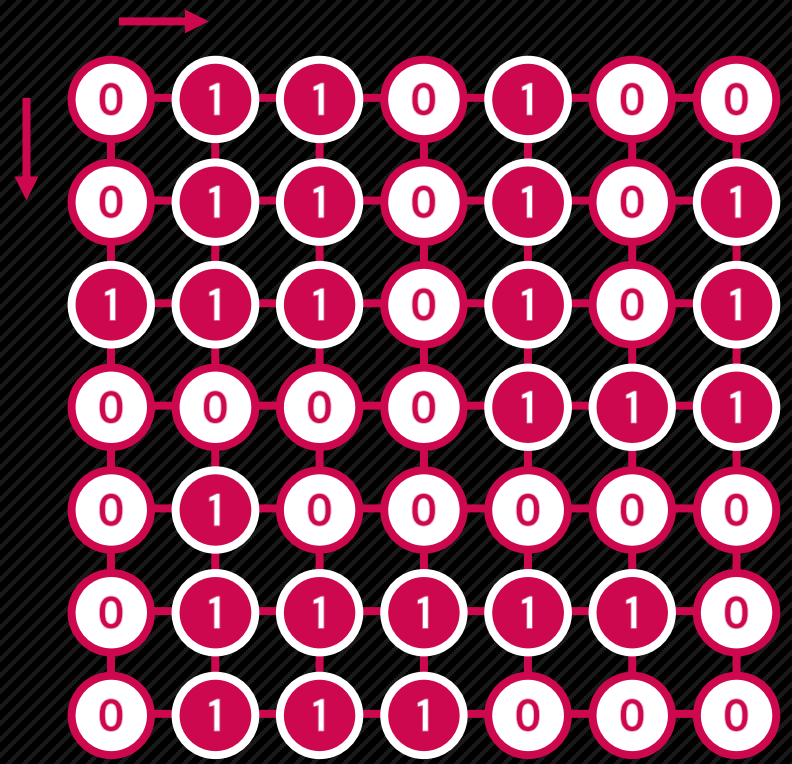
```
0110100
0110101
1110101
0000111
0100000
0111110
0111000
```



그래프인데 배열?

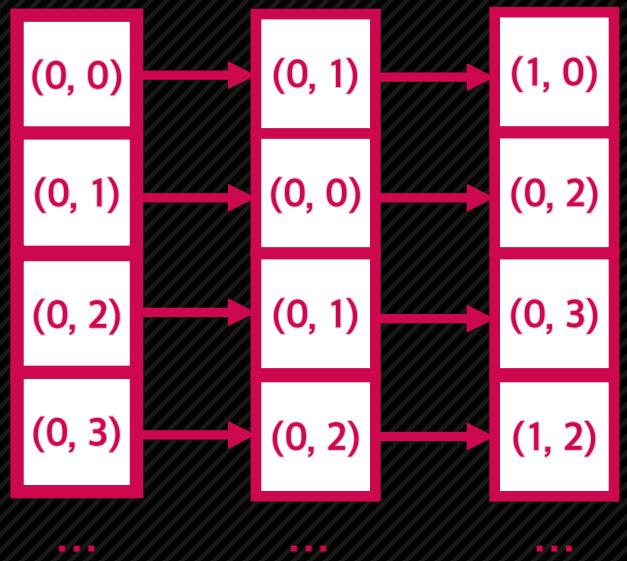
Silver 1 - 단지번호 붙이기 (#2667)

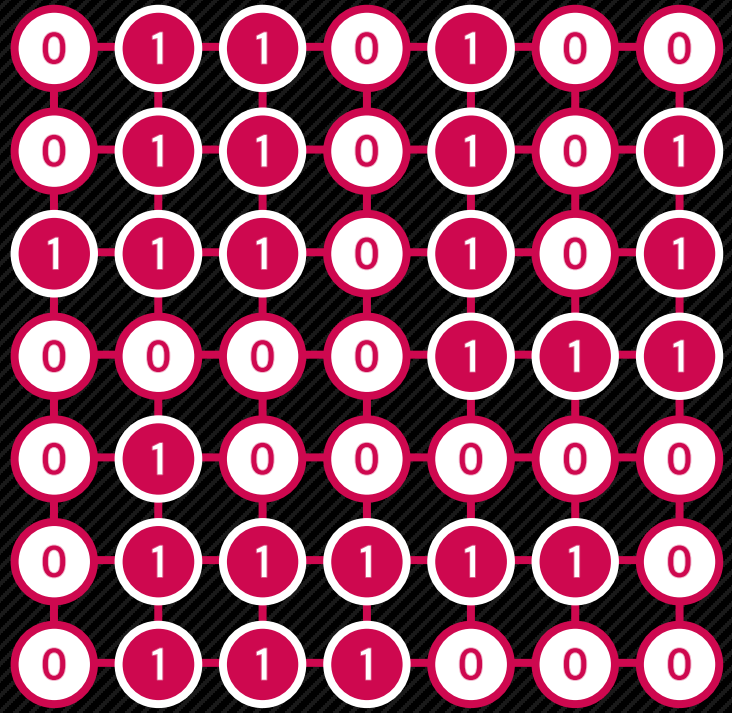
```
0110100
0110101
1110101
0000111
0100000
0111110
0111000
```



그래프인데 배열?

Silver 1 - 단지번호 붙이기 (#2667)



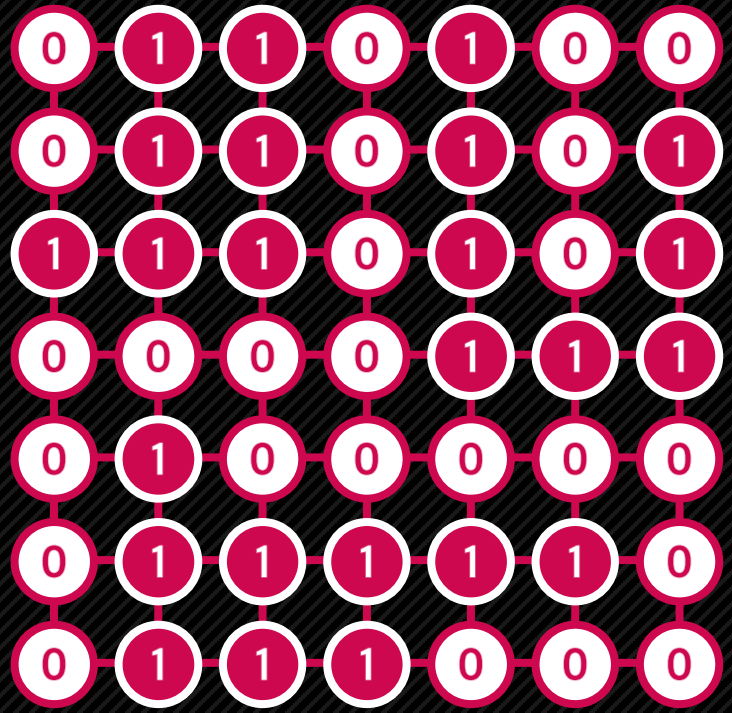


그래프인데 배열?

Silver 1 - 단지번호 붙이기 (#2667)

```
dx = [1, -1, 0, 0]
dy = [0, 0, 1, -1]

def dfs(x, y):
    visited[x][y] = 1
    for i in range(4):
        X = x + dx[i], Y = y + dy[i]
        if X < 0 or X == N or Y < 0 or Y == M:
            continue
        if not(visited[x][y]) and graph[x][y]:
            dfs(X, Y)
```



다시 문제로 돌아가서…

Silver 1 - 단지번호 붙이기 (#2667)

요약

- 집이 있는 곳을 1, 집이 없는 곳을 0으로 하는 그래프가 주어진다.
- 상하좌우에 다른 집이 있는 경우, 그 집은 붙어있다고 한다.
- 이때, 붙어있는 단지의 수와 각 단지 마다 속한 집의 수를 출력하는 프로그램을 작성하시오.

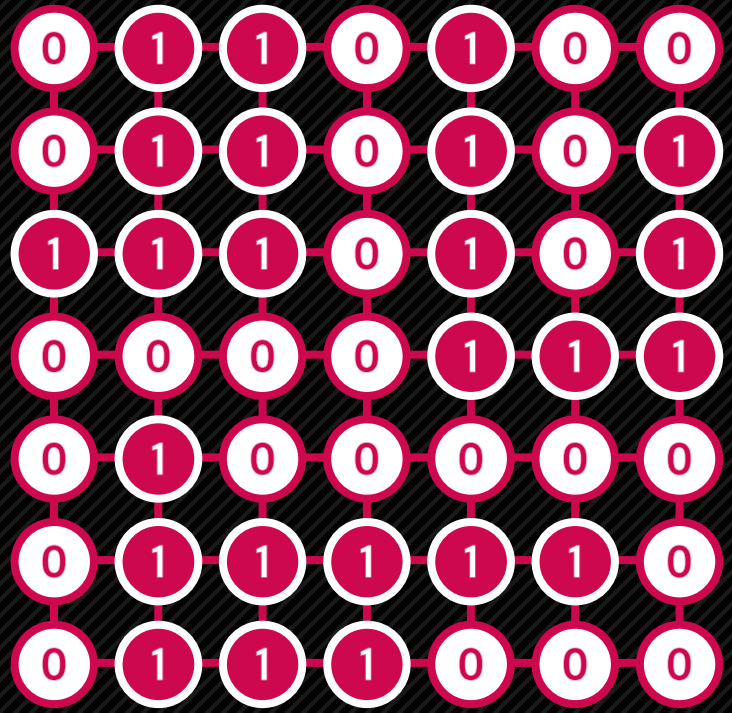
제약조건

- 그래프의 가로/세로 크기는 25 이하이다.

다시 문제로 돌아가서…

Silver 1 - 단지번호 붙이기 (#2667)

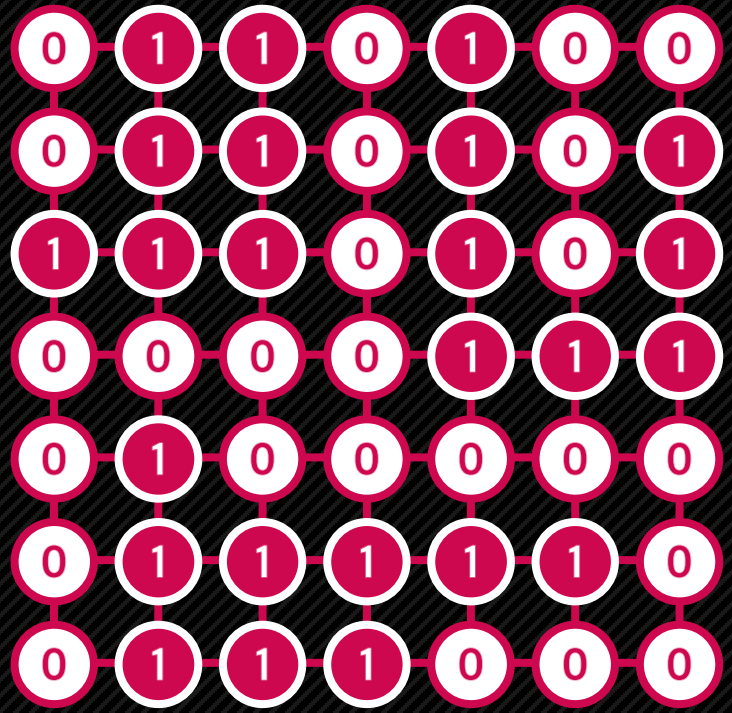
그래프의 진입점?



다시 문제로 돌아가서…

Silver 1 - 단지번호 붙이기 (#2667)

```
for i in range(N):
    for j in range(M):
        if not(visited[i][j]):
            dfs(i, j)
```



BFS/DFS
무엇을 사용해야 할까?

Silver 1 - 토마토 (#7576)

요약

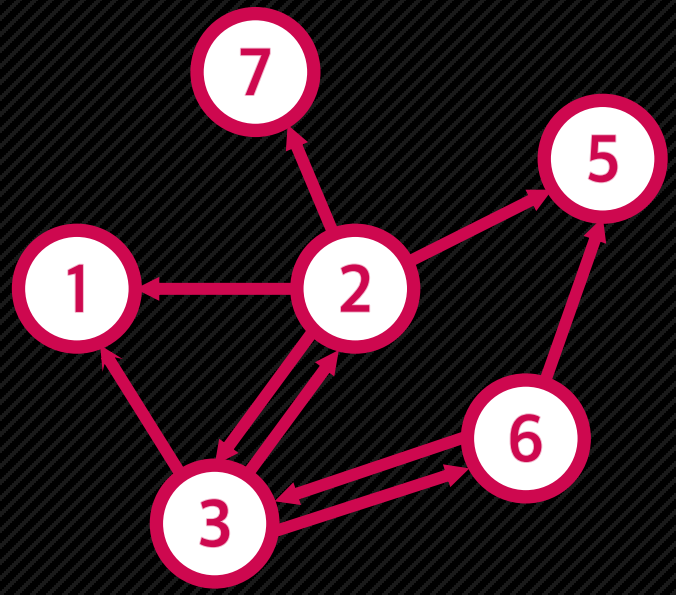
- 토마토를 N × M 크기의 상자에 넣는다.
- 만약 익지 않은 토마토 주변 (상, 하, 좌, 우) 에 익은 토마토가 있다면, 다음 날 그 토마토는 익는다.
- 상자 안에 있는 토마토가 며칠안에 모두 익는지 구하는 프로그램을 작성하시오.

제약조건

- M과 N의 범위는 2 <= N, M <= 1,000 이다.

BFS/DFS
무엇을 사용해야 할까?

Silver 1 - 토마토 (#7576)



Breadth First Search

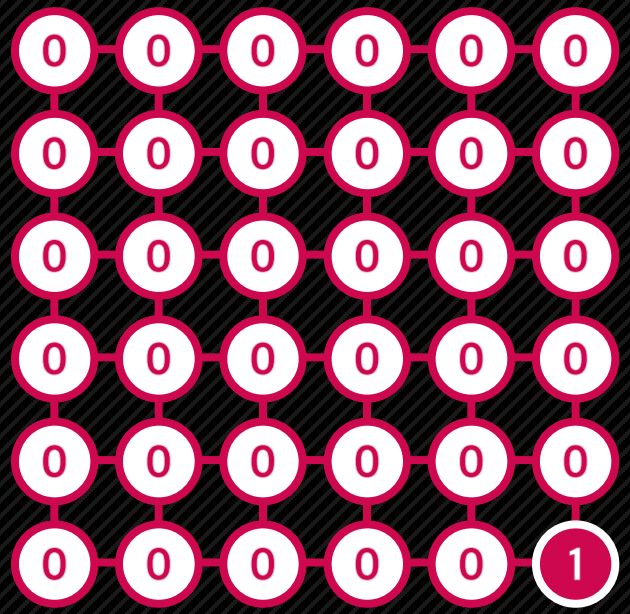
- 너비 우선 탐색이라고도 하며, 자신의 자식들부터 순차적으로 탐색.
- 순차 탐색 이후 다른 자식 노드의 자식을 확인하기 위해 큐를 사용.

Depth First Search

- 깊이 우선 탐색이라고도 하며, 최대한 깊게 탐색 후 빠져 나옴.
- 백트래킹의 일종이며, 재귀를 활용함.

BFS/DFS
무엇을 사용해야 할까?

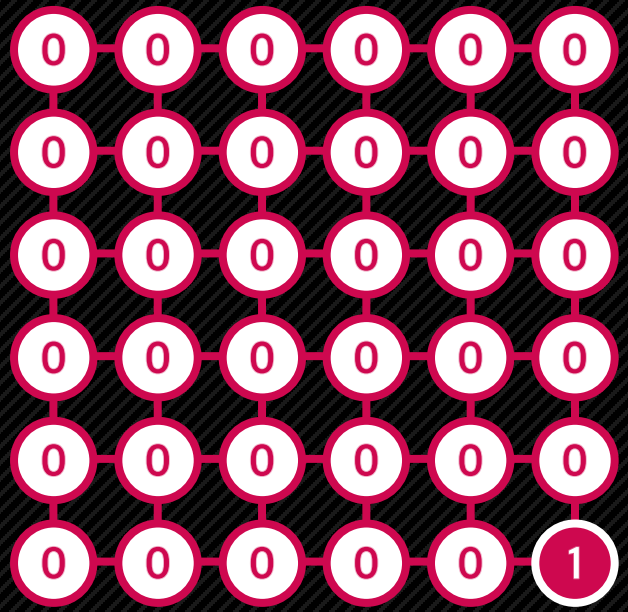
Silver 1 - 토마토 (#7576)



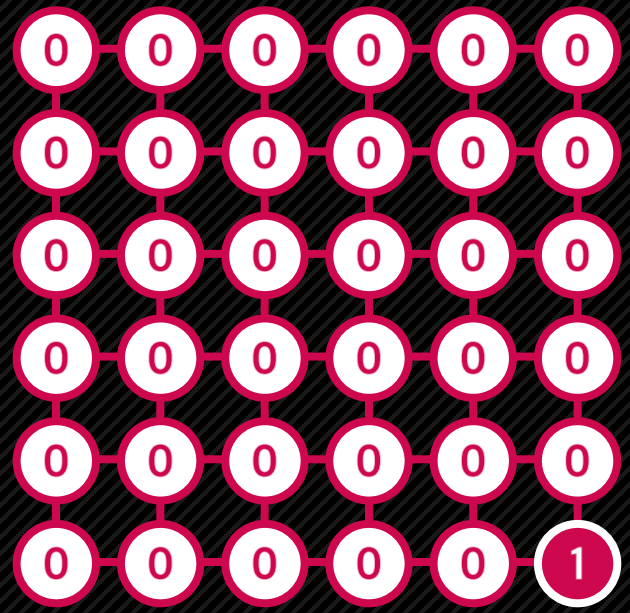
BFS/DFS
무엇을 사용해야 할까?

Silver 1 - 토마토 (#7576)

Breadth First Search



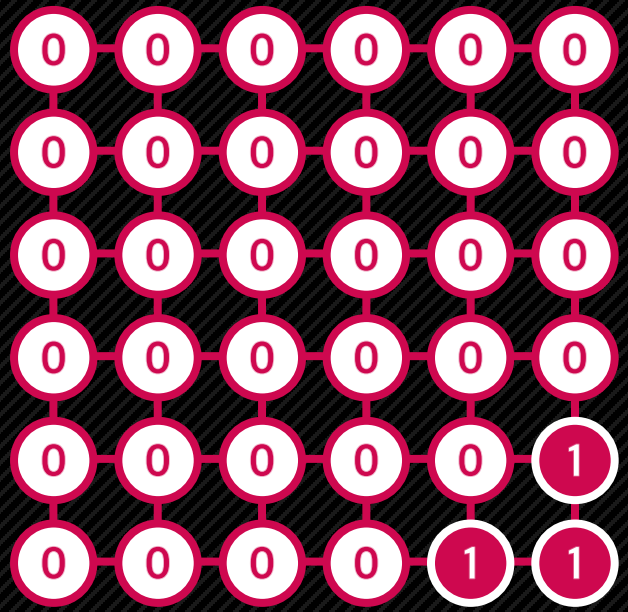
Depth First Search



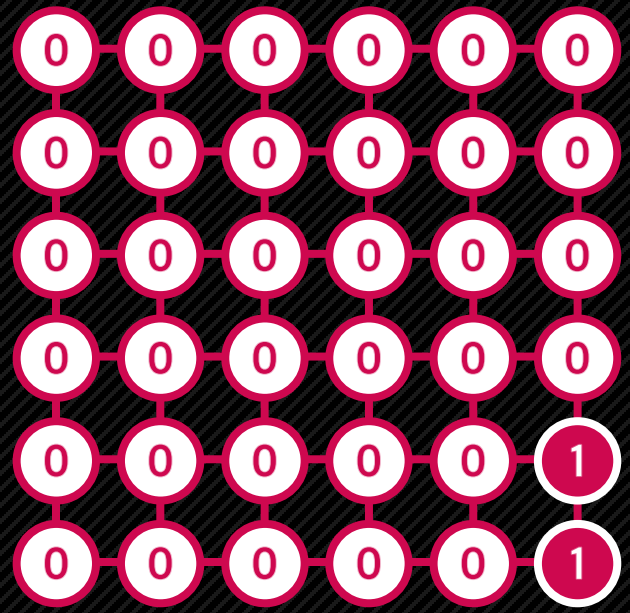
BFS/DFS
무엇을 사용해야 할까?

Silver 1 - 토마토 (#7576)

Breadth First Search



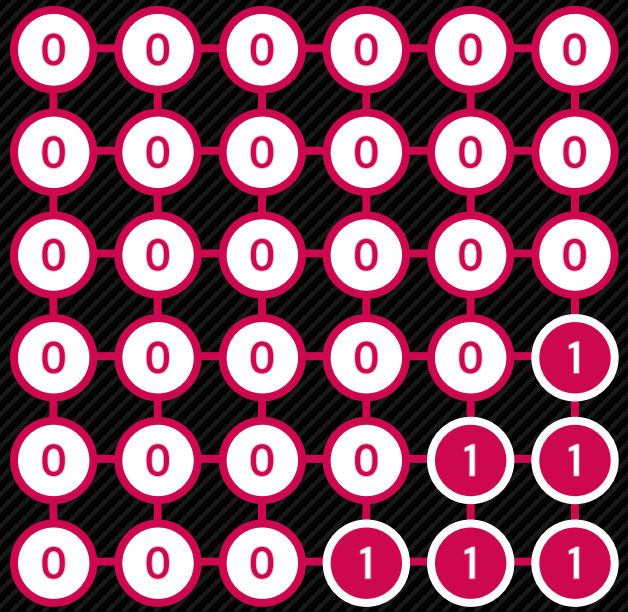
Depth First Search



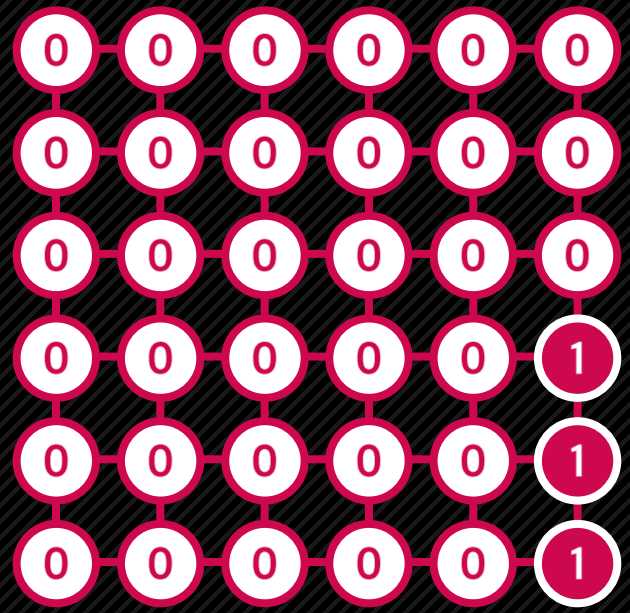
BFS/DFS
무엇을 사용해야 할까?

Silver 1 - 토마토 (#7576)

Breadth First Search



Depth First Search



별로 바뀐건 없어요.

Silver 1 - 토마토 (#7569)

요약

- 토마토를 N × M × H 크기의 상자에 넣는다.
- 만약 익지 않은 토마토 주변에 익은 토마토가 있다면, 다음 날 그 토마토는 익는다.
- 상자 안에 있는 토마토가 며칠안에 모두 익는지 구하는 프로그램을 작성하시오.

제약조건

- M과 N의 범위는 2 <= N, M, H <= 100 이다.

최단거리!

Silver 1 - 숨바꼭질 (#1697)

요약

- 수빈이와 동생은 숨바꼭질을 하고 있는데, 현재 수빈이는 점 N에 있고, 동생은 M에 있다.
- 수빈이는 동생이 있는 곳 까지 가기 위해 걷거나 (X - 1 또는 X + 1로 이동) 순간이동 (2 × X로 이동)을 할 수 있다.
- 수빈이가 동생을 찾을 수 있는 가장 빠른 시간을 구하는 프로그램을 작성하시오.

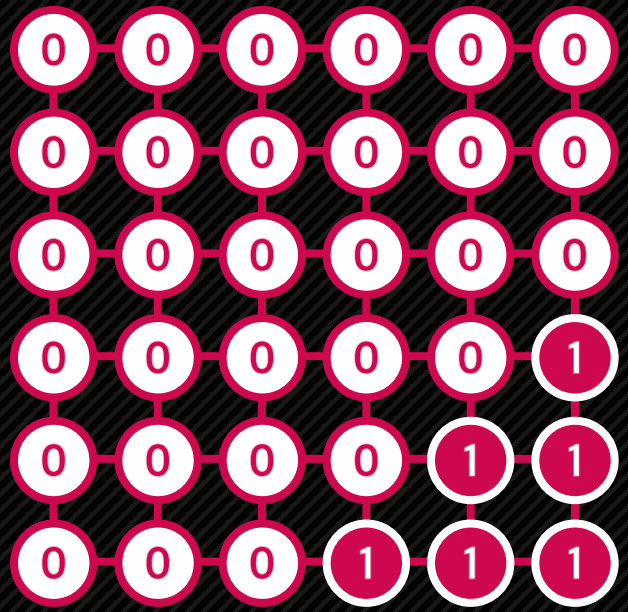
제약조건

- N과 M의 범위는 0 <= N, M <= 100,000 이다.

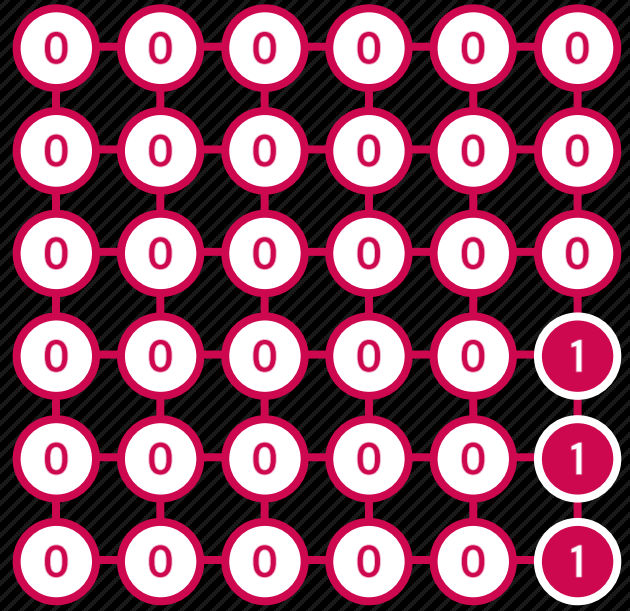
토마토를 떠올려보면…

Silver 1 - 숨바꼭질 (#1697)

Breadth First Search

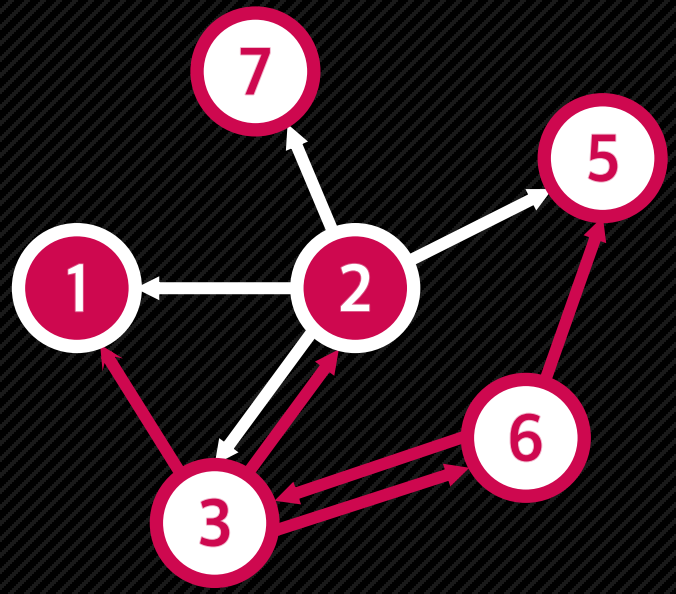


Depth First Search



BFS의 또다른 발견

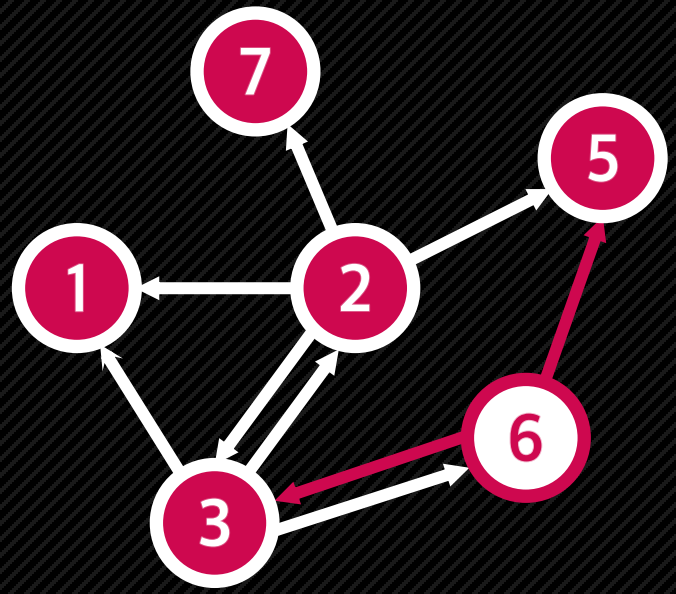
Silver 1 - 숨바꼭질 (#1697)




```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

BFS의 또다른 발견

Silver 1 - 숨바꼭질 (#1697)




```
function bfs(pos) {
    set Q = Queue
    pos -> Q
    while Q is not empty
        set node = popped element of Q
        for children of pos
            if each child has not been visited
                visit child
                child -> Q
}
```

BFS의 또다른 발견

Silver 1 - 숨바꼭질 (#1697)

- N번 탐색 후 큐에 들어있는 내용을 모두 꺼내서 한 번 씩 탐색하면,
  큐에는 N + 1 번째로 방문한 값들이 들어있음이 보장됨!

최단거리!

Silver 1 - 숨바꼭질 (#1697)

요약

- 수빈이와 동생은 숨바꼭질을 하고 있는데, 현재 수빈이는 점 N에 있고, 동생은 M에 있다.
- 수빈이는 동생이 있는 곳 까지 가기 위해 걷거나 (X - 1 또는 X + 1로 이동) 순간이동 (2 × X로 이동)을 할 수 있다.
- 수빈이가 동생을 찾을 수 있는 가장 빠른 시간을 구하는 프로그램을 작성하시오.

제약조건

- N과 M의 범위는 0 <= N, M <= 100,000 이다.

최단거리!

Silver 1 - 숨바꼭질 (#1697)

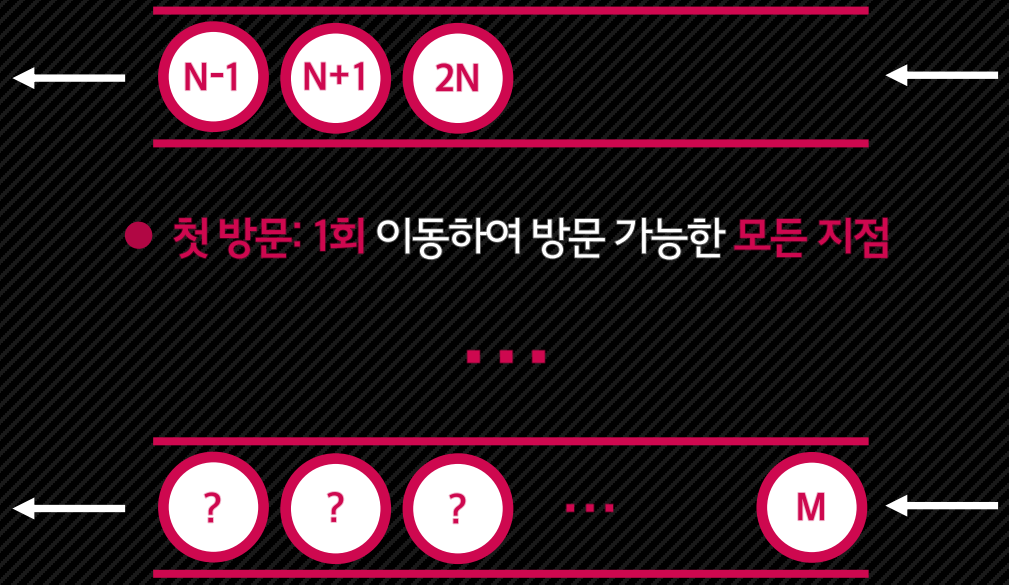
요약

- 수빈이가 동생을 찾을 수 있는 **가장 빠른 시간을 구하는 프로그램**을 작성하시오.

- **N번 탐색 후 큐에 들어있는 내용**을 모두 꺼내서 한 번 씩 탐색하면,
  큐에는 **N + 1 번째로 방문한 값**들이 들어있음이 보장됨!

최단거리!

Silver 1 - 숨바꼭질 (#1697)

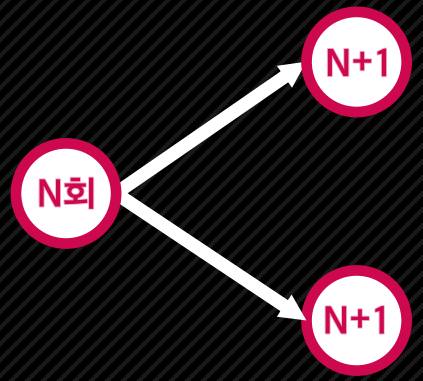


- 첫 방문: 1회 이동하여 방문 가능한 모든 지점

- K번째 방문 중 M이 큐에 들어오게 되면 최단시간!

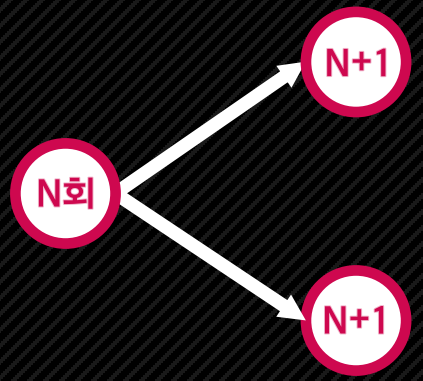
살짝만 틀어볼게요.

Silver 1 - 숨바꼭질 (#1697)



살짝만 틀어볼게요.

Silver 1 - 숨바꼭질 (#1697)




```
if visited[next] == -1:
    visited[next] = visited[prev] + 1
```

어?

Gold 5 - 숨바꼭질 3 (#13549)

요약

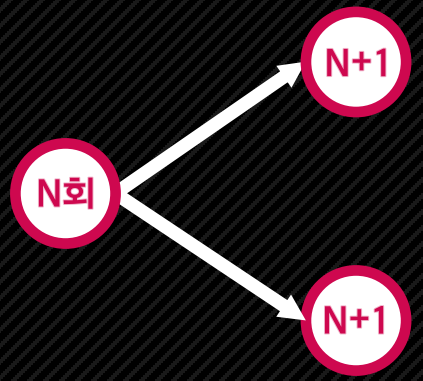
- 수빈이와 동생은 숨바꼭질을 하고 있는데, 현재 수빈이는 점 N에 있고, 동생은 M에 있다.
- 수빈이는 동생이 있는 곳 까지 가기 위해 걷거나 (X - 1 또는 X + 1로 이동) 순간이동 (2 × X로 이동, 0초)을 할 수 있다.
- 수빈이가 동생을 찾을 수 있는 가장 빠른 시간을 구하는 프로그램을 작성하시오.

제약조건

- N과 M의 범위는 0 <= N, M <= 100,000 이다.

이렇게 하면 안 될 것 같은데…

Gold 5 - 숨바꼭질 3 (#13549)




```
if visited[next] == -1:
    visited[next] = visited[prev] + 1
```

요런식으로 해야해요.

Gold 5 - 숨바꼭질 3 (#13549)

```
if visited[prev] < visited[prev * 2] :
    visited[prev * 2] = visited[prev]
```

하나의 플래그를 고려해야 할 때?

Gold 4 - 벽 부수고 이동하기 (#2206)

요약

- N × M 크기의 맵이 있는데, (1, 1)에서 (N, M)까지 이동하려고 한다.
- 만약 이동하는 중 벽을 부수고 이동하는 것이 더 빠르다면, 벽을 한 개 까지 부수고 이동해도 된다.
- 맵이 주어졌을 때, 최단 거리를 구하는 프로그램을 작성하시오.

제약조건

- N과 M의 범위는 0 <= N, M <= 1,000 이다.

하나의 플래그를 고려해야 할 때?

Gold 4 - 벽 부수고 이동하기 (#2206)

- 일반적인 DFS, BFS를 한다면…

??: 너 벽 몇 개 부쉈어?
??: 그러게요?

하나의 플래그를 고려해야 할 때?

Gold 4 - 벽 부수고 이동하기 (#2206)

- 플래그를 넣어주자! (벽을 현재 부순 상태인지)

```
def bfs():
    _q = deque()
    _q.append((0, 0, 0)) # 세번째 인자: flag
```

하나의 플래그를 고려해야 할 때?

Gold 4 - 벽 부수고 이동하기 (#2206)

- 플래그만 넣으면 다야?

하나의 플래그를 고려해야 할 때?

Gold 4 - 벽 부수고 이동하기 (#2206)

- 플래그만 넣으면 다야?

- 벽을 부순 경우, 해당 칸에 5회 만에, 아닌 경우 7회 만에 도달함.

하나의 플래그를 고려해야 할 때?

Gold 4 - 벽 부수고 이동하기 (#2206)

- 플래그만 넣으면 다야?

- 벽을 부순 경우, 해당 칸에 5회 만에, 아닌 경우 7회 만에 도달함.
- 벽을 부수지 않고 도달하면 이미 visit 상태라 방문하지 않고 넘어감.

하나의 플래그를 고려해야 할 때?

Gold 4 - 벽 부수고 이동하기 (#2206)

```
visited[1001][1001][2];

def bfs():
    _q = deque()
    _q.append((0, 0, 0)) # 세번째 인자: flag
```

- 벽을 부순 상태와 부수지 않은 상태의 visit 값 분리!

Gold 5 - 연구소 (#14502)

요약

- N × M 크기의 연구소에 바이러스가 퍼졌다. 퍼지지 않은 다른 곳을 보호하기 위해 **벽을 설치해야 한다.**
- 벽은 **정확히 3개**를 세울 수 있다.
- 이때, 얻을 수 있는 **안전 구역의 크기의 최댓값**을 구하는 프로그램을 작성하시오.

제약조건

- N과 M의 범위는 **3 <= N, M <= 8** 이다.

완전탐색!

Gold 5 - 연구소 (#14502)

요약

- N × M 크기의 연구소에 바이러스가 퍼졌다. 퍼지지 않은 다른 곳을 보호하기 위해 벽을 설치해야 한다.
- 벽은 정확히 3개를 세울 수 있다.
- 이때, 얻을 수 있는 안전 구역의 크기의 최댓값을 구하는 프로그램을 작성하시오.

제약조건

- N과 M의 범위는 3 <= N, M <= 8 이다.

접근

- 벽 3개를 고르는 경우의 수는?
  - → 64C3 = 41,664
  - → 그러면 벽 3개 고르고 탐색해도 시간 남겠네?

“To be continue,,,”