그래프와 DFS, BFS

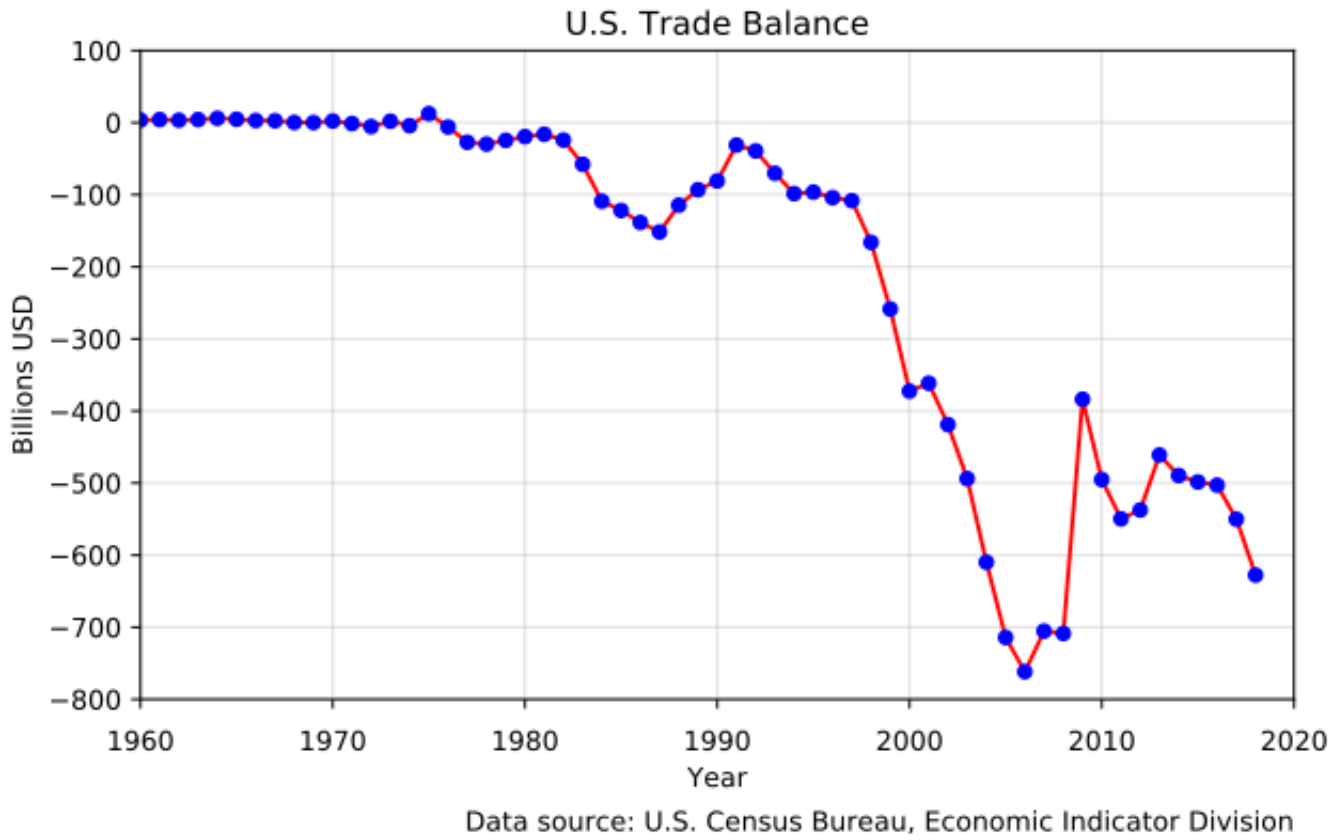
경북대학교 전현승
dogdriip@gmail.com

Table of contents

- 0x00

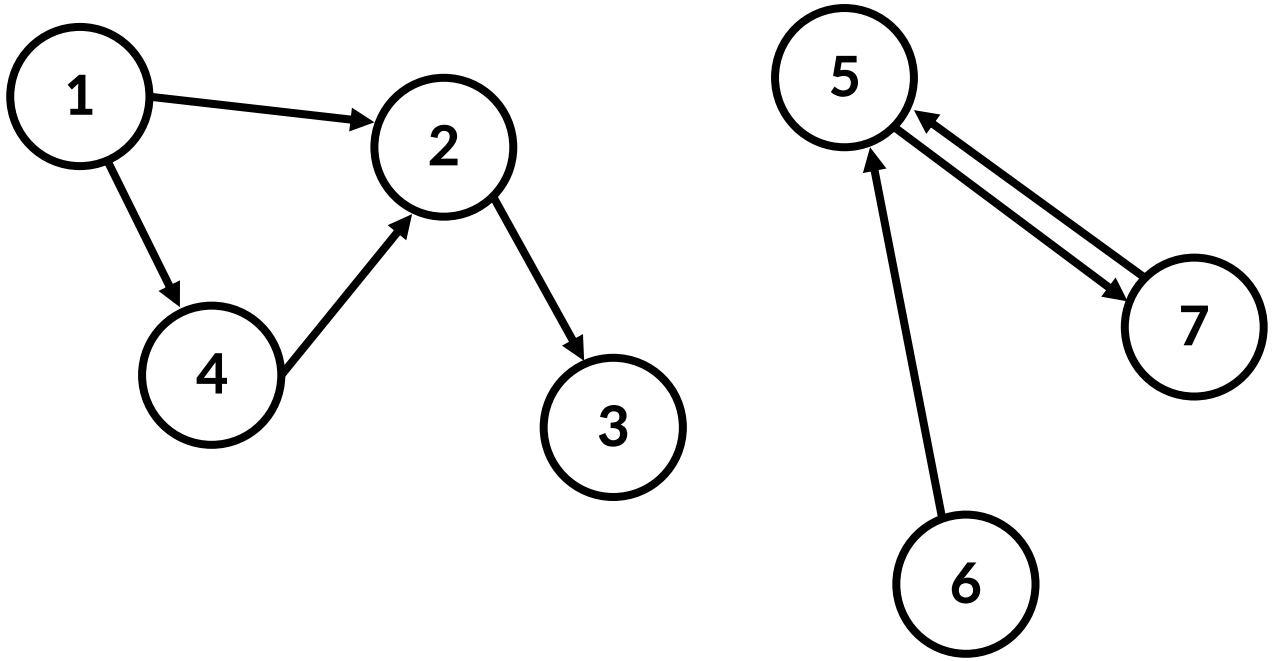
그래프가 뭔데?
인접 행렬, 인접 리스트?

그래프가 뭔데?



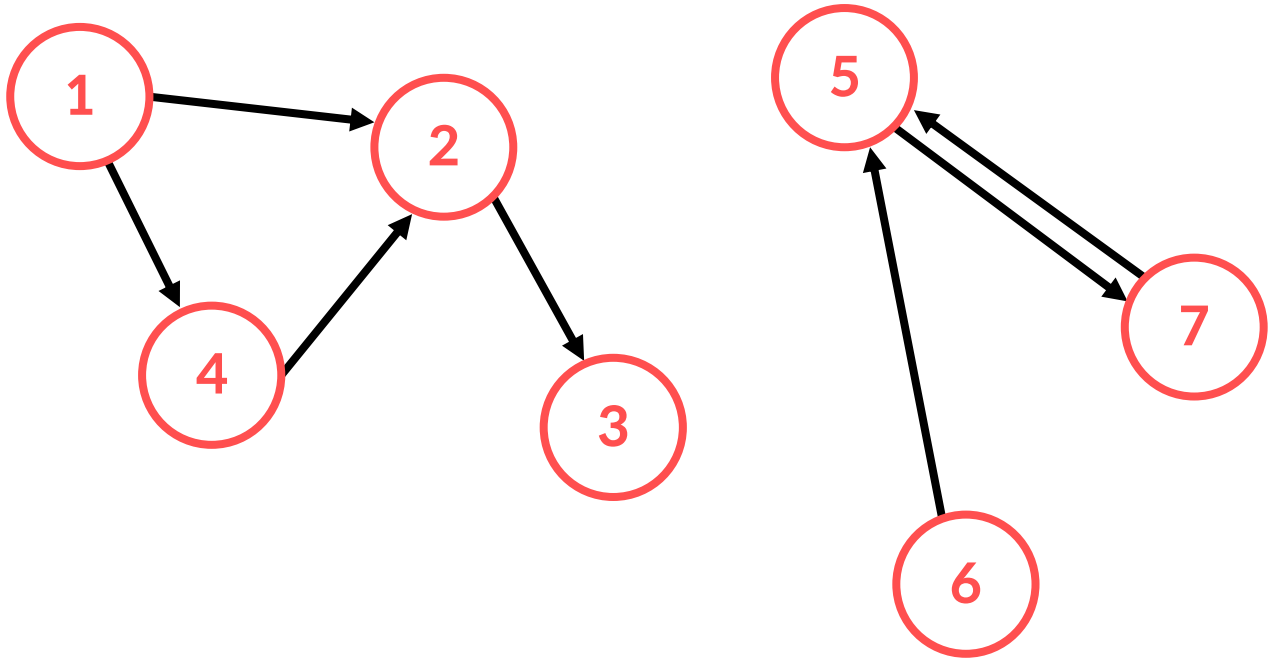
- 이런 게 아니다

그래프가 뭔데?



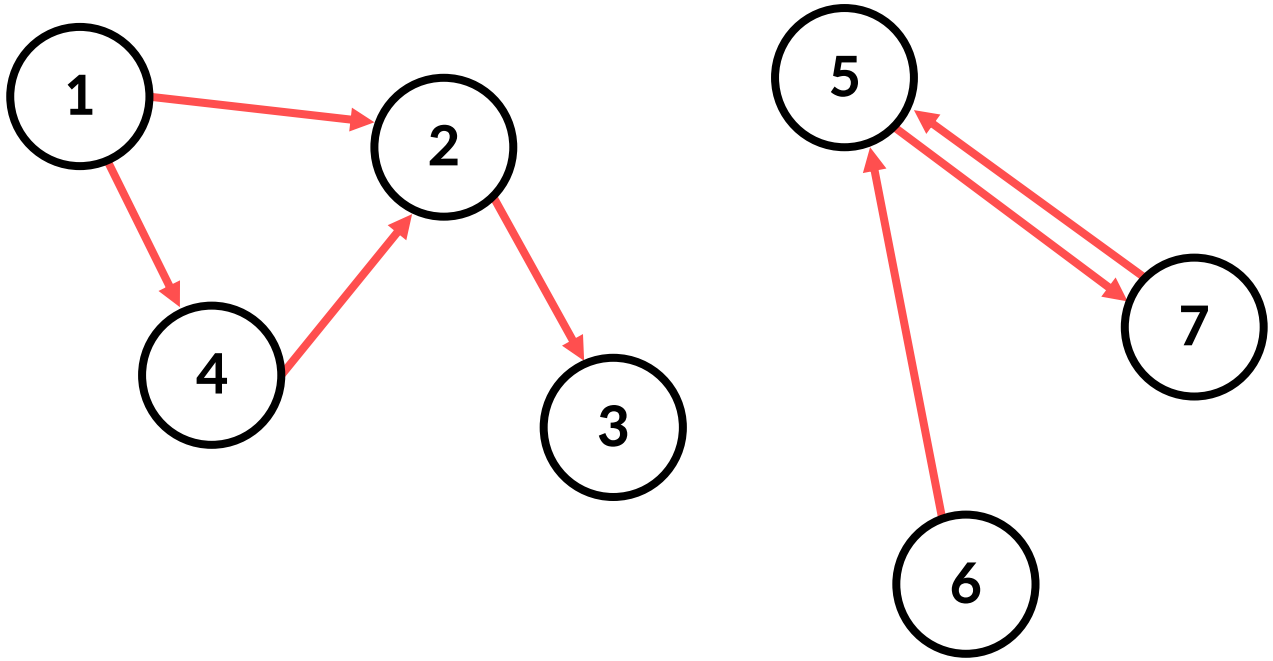
- 이런 식으로 생긴 것들

그래프가 뭔데?



- 정점 (Vertex, Node)

그래프가 뭔데?



- 간선 (Edge) : 정점과 정점을 연결

그래프의 정의

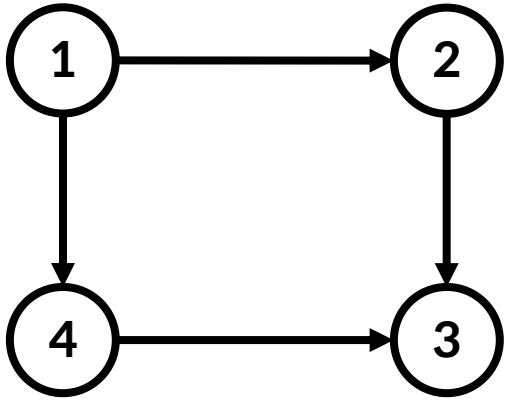
- **그래프** : **정점**(Vertex, Node)들과 **간선**(Edge)들로 이루어진 자료구조
- $G(V, E)$ 등의 표기법으로 나타냄

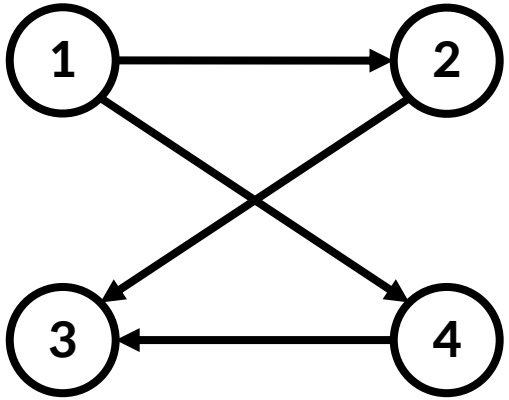
- V = 정점들의 집합 ($|V|$ = 정점 개수)
- E = 간선들의 집합 ($|E|$ = 간선 개수)

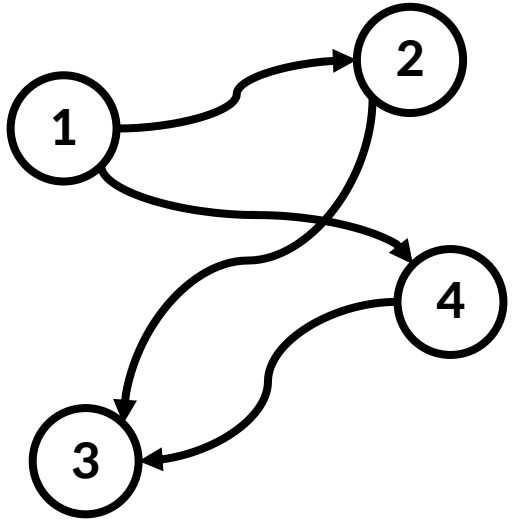
그래프의 정의

- 그래프의 정의에서 알 수 있듯이, **그래프는 정점, 간선의 정보만 저장한다.**
- 그래프 모양, 정점 위치 등은 그래프의 정의와 무관하다.

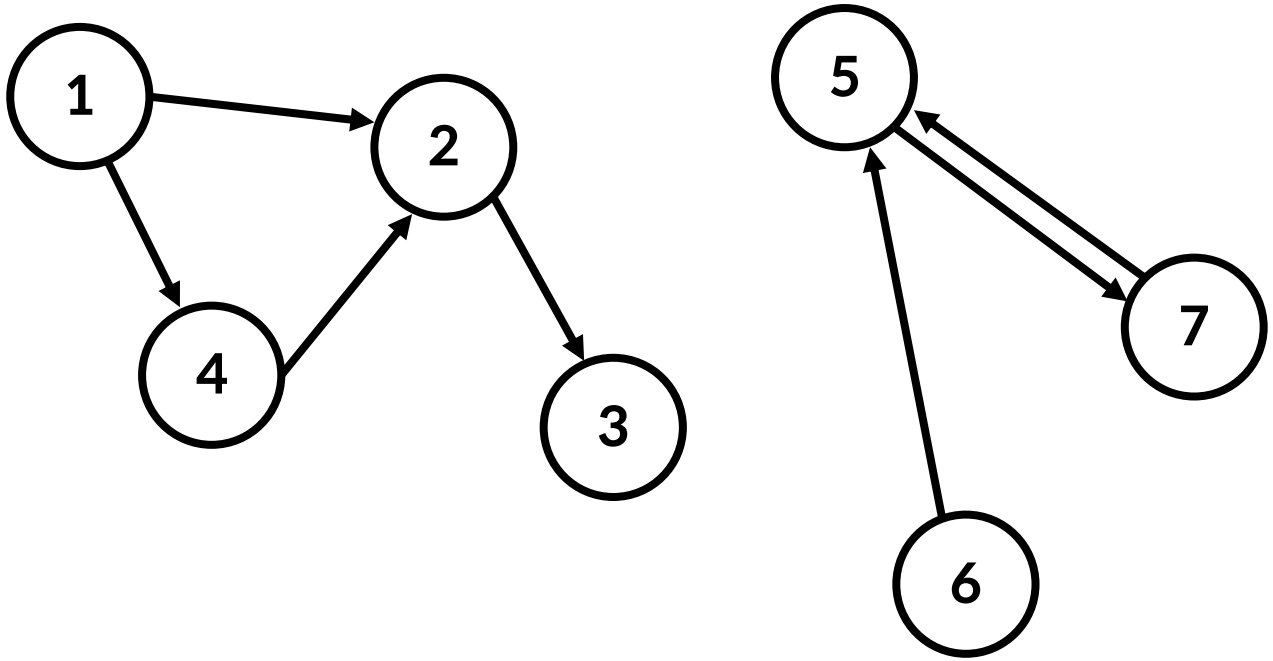
- 아래 세 그래프는 모두 같은 그래프이다.





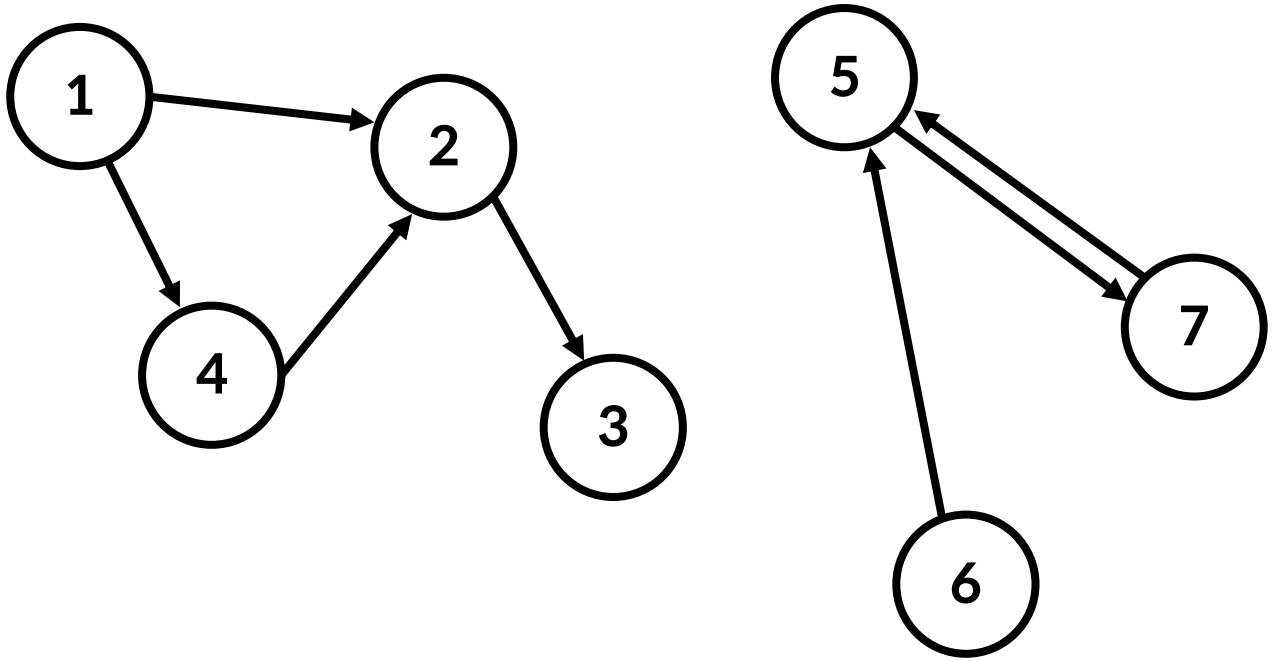


그래프의 종류



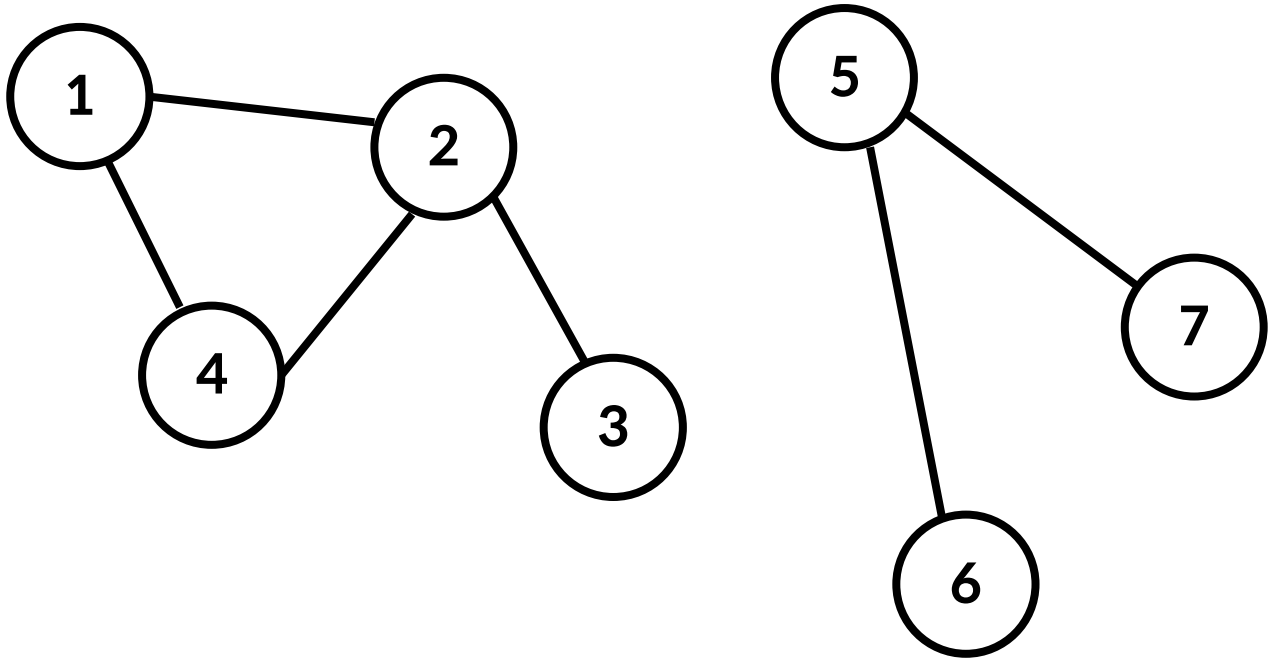
- 그래프도 특성에 따라 이름을 붙이거나, 종류를 나눠볼 수 있다.
- 몇 가지만 살펴보자

그래프의 종류 – 간선의 방향 존재 여부



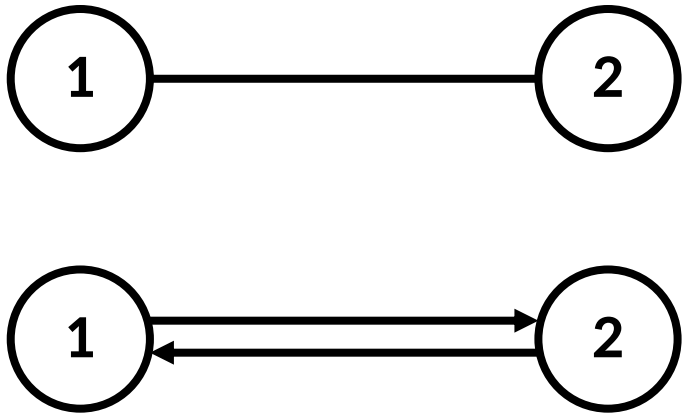
- 간선은 방향이 있을 수도, 없을 수도 있다
- 간선에 방향이 있는 그래프 – **Directed graph (유향 그래프)**

그래프의 종류 – 간선의 방향 존재 여부



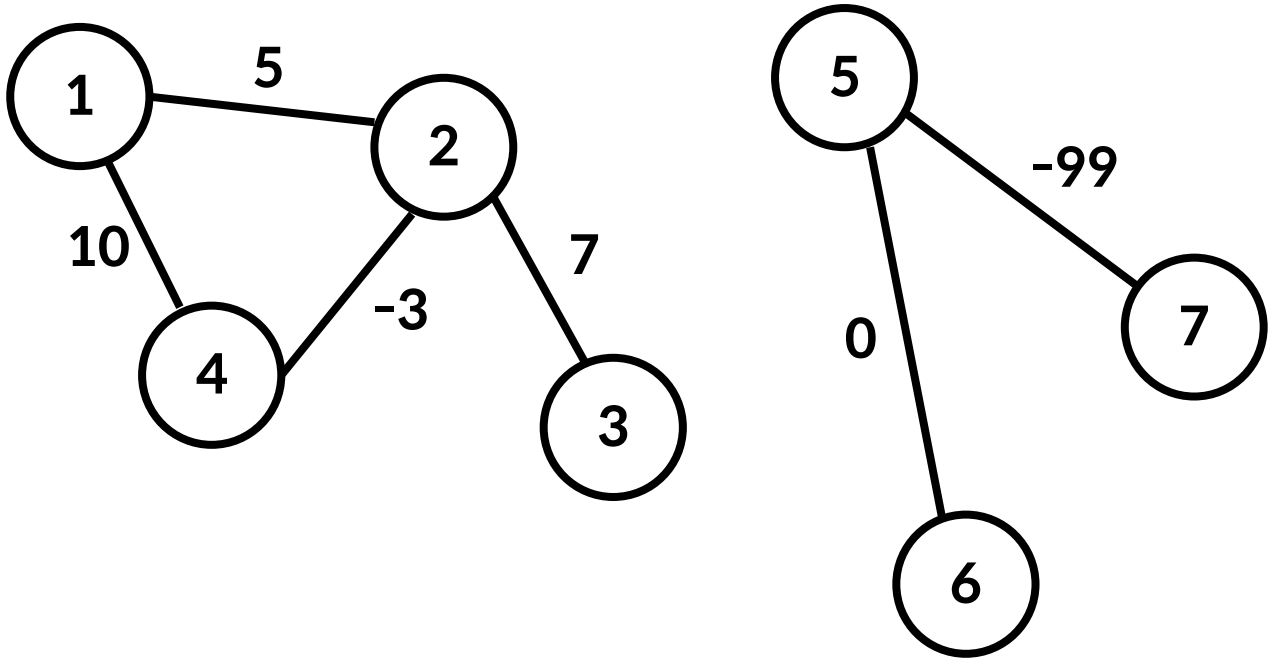
- 간선에 방향이 없는 그래프 – Undirected graph (무향/무방향/양방향 그래프)

그래프의 종류 – 간선의 방향 존재 여부



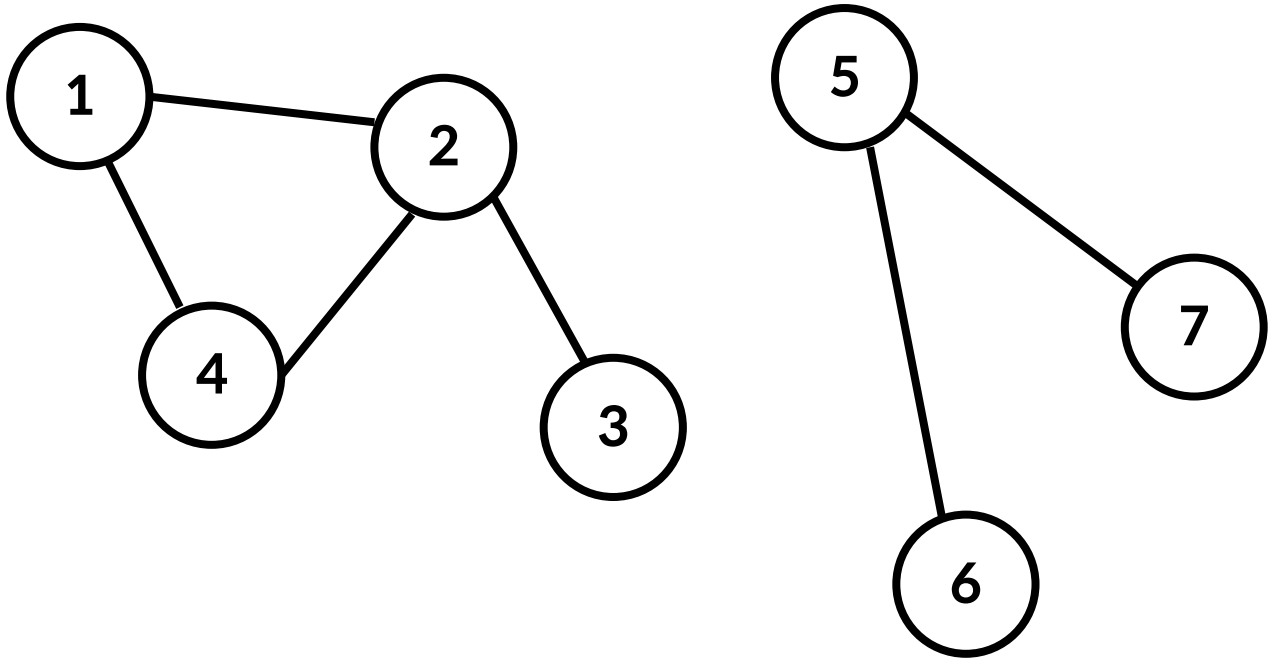
- 무향 간선 하나는 유향 간선 두 개로 모델링할 수 있다

그래프의 종류 – 간선의 가중치 존재 여부



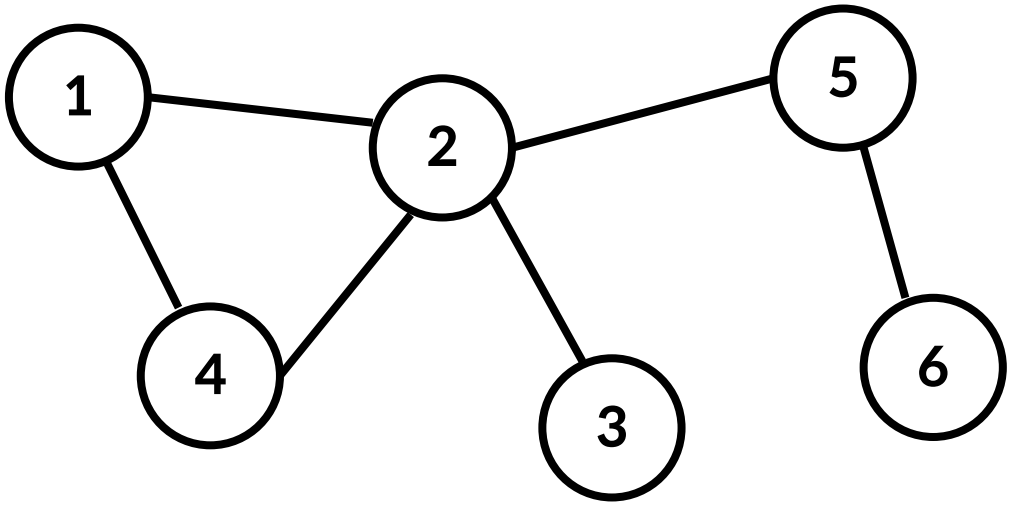
- 간선에 가중치가 있을 수도, 없을 수도 있다
- 간선에 가중치가 있는 그래프 – Weighted graph

그래프의 종류 – 간선의 가중치 존재 여부



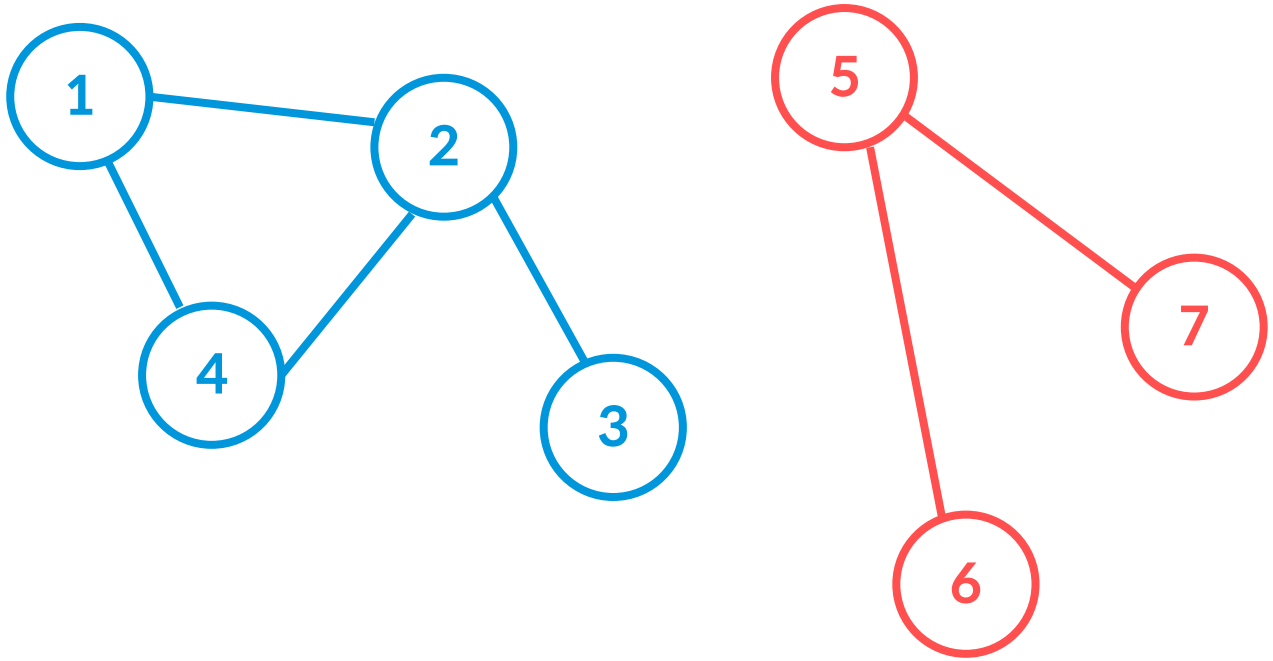
- 간선에 가중치가 없는 그래프 – Unweighted graph

그래프의 종류 – 모든 정점이 연결되어 있는가



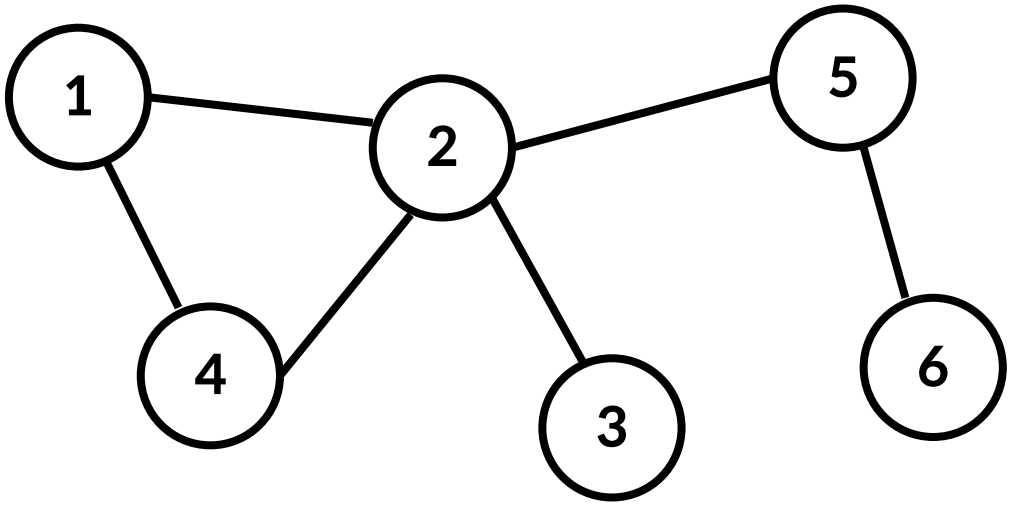
- 그래프의 정의에 따르면, 그래프의 정점들이 모두 연결되어 있을 필요도 없다.
- 모든 두 정점 사이에 경로가 존재하는 그래프 – **Connected graph (연결 그래프)**

+ 그래프의 연결 요소



- 연결 요소 (Connected component)
- 위의 그래프는 연결 요소가 2개인 그래프이다.

+ 그래프의 연결 요소



- 따라서, Connected graph의 경우, **연결 요소의 개수가 하나인 그래프**라고 할 수 있다.

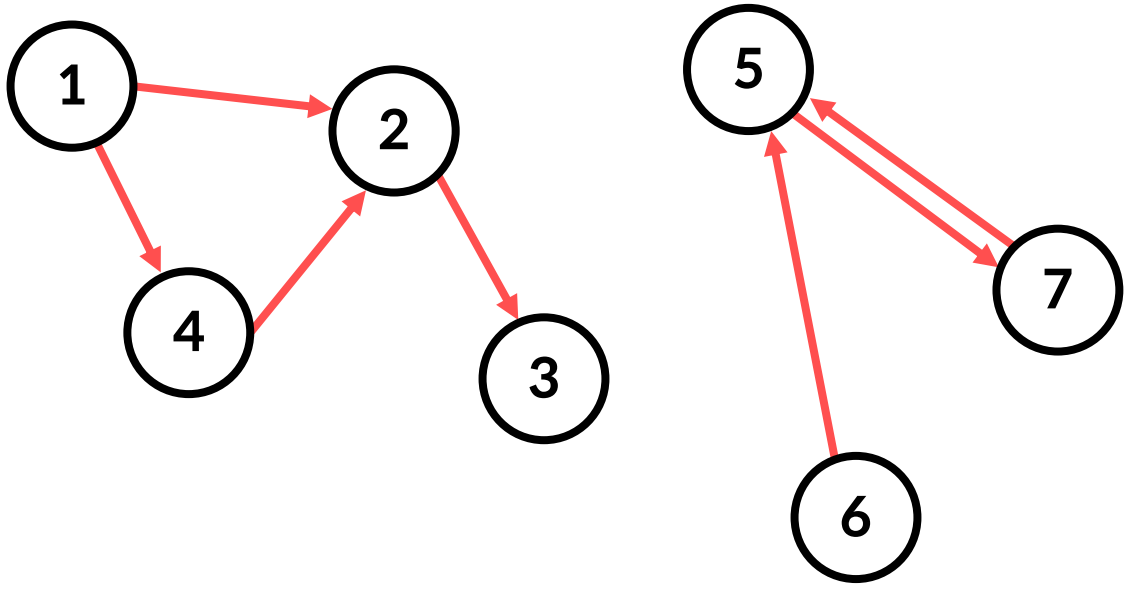
그래프 표현 방법

- 이러한 그래프 구조들을 컴퓨터에서는 어떻게 표현할까?
- 코드로 어떻게 옮길까? 어떻게 저장할까?

- 크게 두 가지 방법이 있다
- **인접 행렬**과 **인접 리스트**

그래프 표현 방법 – 인접 행렬

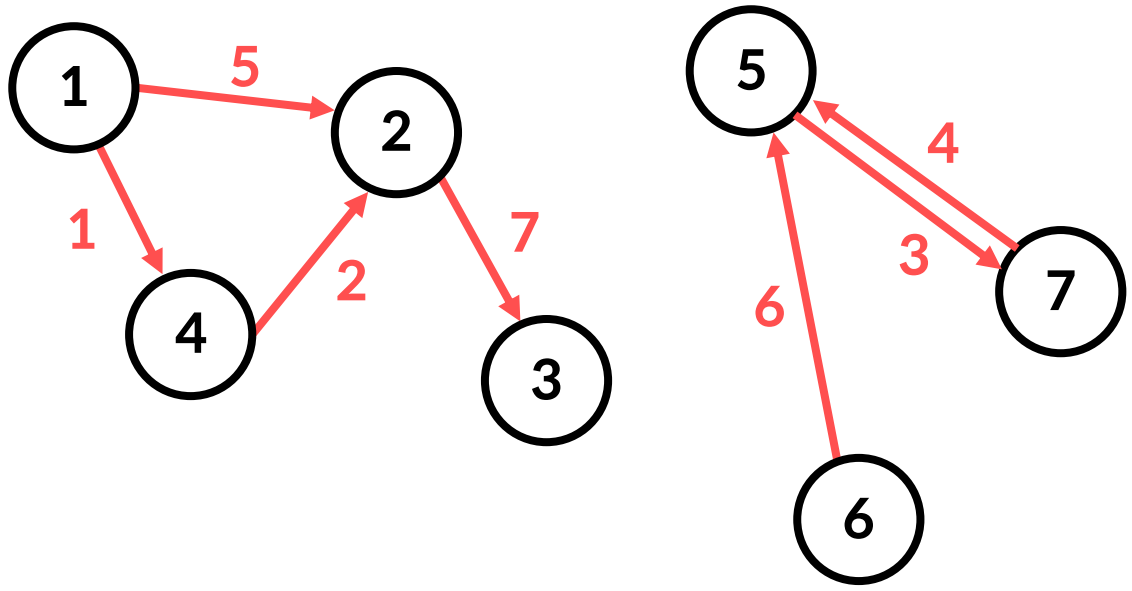
- **인접 행렬 (Adjacency matrix)** : 2차원 행렬
- 코드로는 보통 2차원 배열로 나타낸다 – `bool adj[SIZE][SIZE];`
- `adj[a][b]` : 정점 a에서 정점 b로 가는 간선이 존재하는가?



	1	2	3	4	5	6	7
1	0	1	0	1	0	0	0
2	0	0	1	0	0	0	0
3	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	1	0	0
7	0	0	0	0	1	0	0

그래프 표현 방법 – 인접 행렬

- 다양하게 변형이 가능하다
- `adj[a][b]` : 정점 a에서 정점 b로 가는 간선의 가중치 (0 : 간선이 없음)
  - 가중치가 양의 정수라고 가정했을 때



	1	2	3	4	5	6	7
1	0	5	0	1	0	0	0
2	0	0	7	0	0	0	0
3	0	0	0	0	0	0	0
4	0	2	0	0	0	0	0
5	0	0	0	0	0	0	3
6	0	0	0	0	6	0	0
7	0	0	0	0	4	0	0

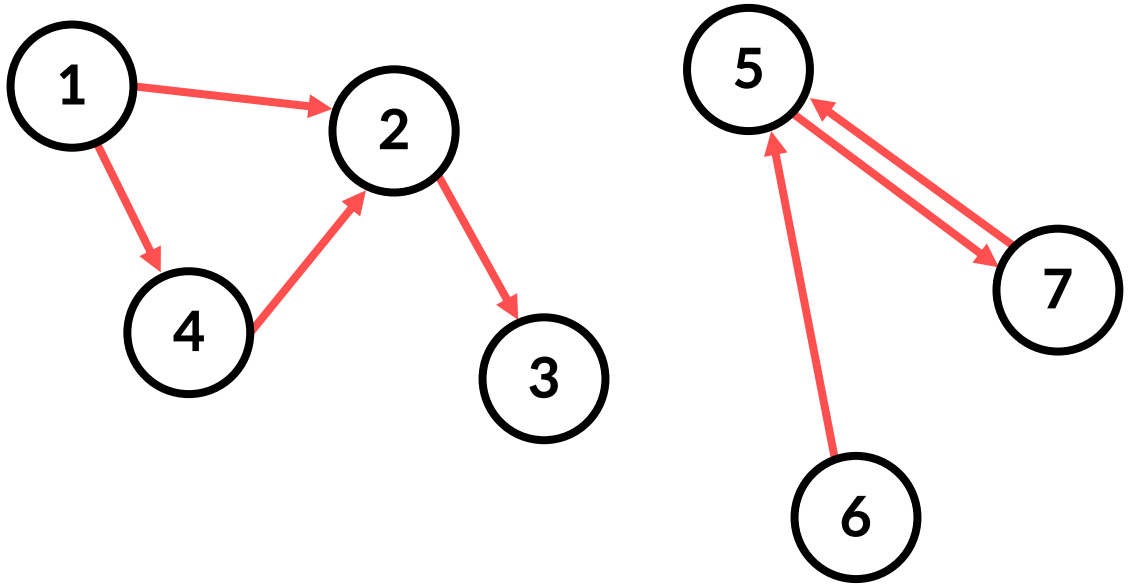
그래프 표현 방법 – 인접 행렬

- 모든 정점 쌍에 대해 간선의 존재 여부를 저장하므로, 행렬의 크기는 $|V| \times |V|$
- 따라서 공간 복잡도 : $O(|V|^2)$

- 특정 간선이 존재하는가를 상수 시간에 판단 가능 (`adj[a][b]`만 확인하면 되니까)
- 그러나 공간 복잡도가 후에 설명할 인접 리스트보다 큰 편이다
  - 정점이 엄청 많고, 간선이 하나만 있는 그래프라면? → 인접 행렬은 비효율적
- 특정 노드와 인접한 노드들을 확인하려면?
  - → `adj[a][1]`부터 `adj[a][|V|]`까지 행렬의 한 행을 모두 봐야 하므로 시간 복잡도 : $O(|V|)$

그래프 표현 방법 – 인접 리스트

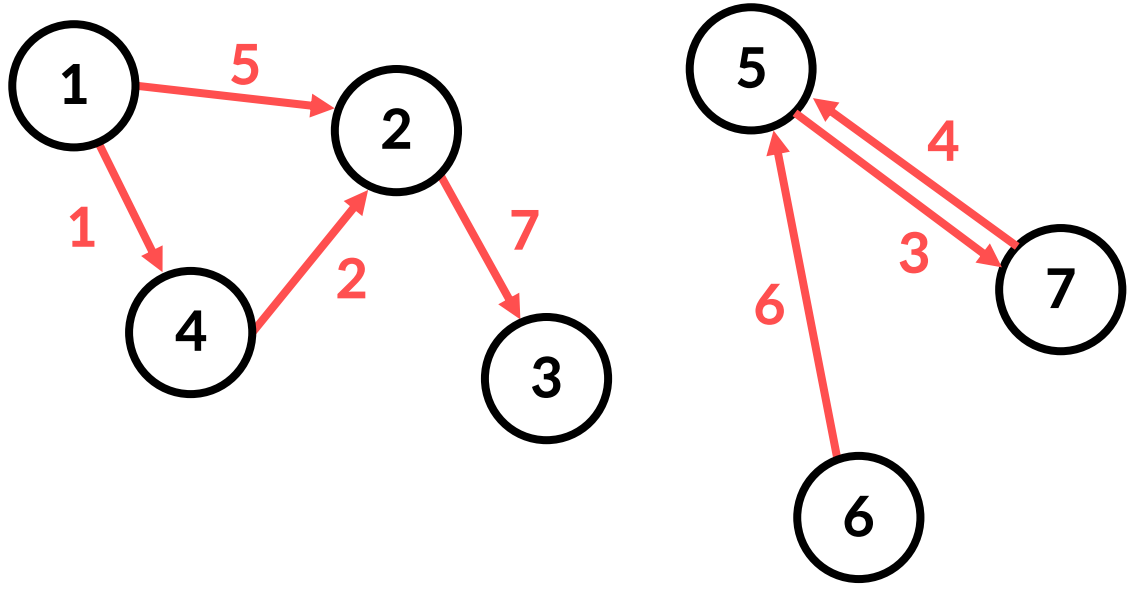
- **인접 리스트 (Adjacency list)** : 그래프 연결 관계를 나타내는 연결 리스트들의 배열
- C++ STL의 vector를 사용하면 편리 – `vector<int> adj[SIZE];`
- `adj[a]` : 정점 a와 연결되어 있는 정점들의 목록 (vector)




```
adj[1] = {2, 4}
adj[2] = {3}
adj[3] = {}
adj[4] = {2}
adj[5] = {7}
adj[6] = {5}
adj[7] = {5}
```

그래프 표현 방법 – 인접 리스트

- 이 친구도 변형이 가능하다
- 여러 정보를 담도록 정의할 수 있다 – `vector<pair<int, int> > adj[SIZE];`
- `adj[a]` : `pair<int, int>`를 담는 `vector`. 정점 a와 연결된 정점과 그 간선의 가중치




```
adj[1] = {(2, 5), (4, 1)}
adj[2] = {(3, 7)}
adj[3] = {}
adj[4] = {(2, 2)}
adj[5] = {(7, 3)}
adj[6] = {(5, 6)}
adj[7] = {(5, 4)}
```

그래프 표현 방법 – 인접 리스트

- 인접 행렬과 달리, 연결된 정점들끼리의 정보만 저장하므로
  공간 복잡도는 대략 $O(|V| + |E|)$

- 인접 행렬처럼 특정 간선이 존재하는지 바로 확인은 불가
  - 정점 a에 인접한 정점들의 리스트 `adj[a]`를 돌면서 확인해봐야 한다
- 그래도 공간 복잡도가 크게 줄었다

그래프 표현 방법 – Which to choose?

- 뒤에 설명할 DFS, BFS의 경우:
  - 인접 행렬을 이용해 저장했을 경우, DFS, BFS의 시간 복잡도가 대략 $O(|V|^2)$
  - 인접 리스트를 이용해 저장했을 경우, DFS, BFS의 시간 복잡도가 대략 $O(|V| + |E|)$
- 여러모로 인접 리스트가 시간 복잡도, 공간 복잡도 면에서 효율적이긴 하다

- 하지만, 인접 행렬과 인접 리스트 두 방법 모두 일장일단이 있다
- 문제에서 어떠한 형태로 입력이 주어지는지, 어떤 알고리즘을 써야 하는지에 따라 특정 방법이 더 편할 때가 있다
- 그러니 두 방법 모두 숙지하면 좋다

- 0x01

그래프의 탐색?
DFS, BFS?

그래프의 탐색?

- **그래프 탐색 (그래프 순회, Graph traversal)** : 그래프의 모든 정점들을 방문하는 것

- 그래프 탐색에는 대표적으로 두 방법이 있다
- 각 정점들을 어느 순서로, 어느 규칙에 따라 방문할 것인가의 차이

- **깊이 우선 탐색** (Depth First Search, **DFS**)
- **너비 우선 탐색** (Breadth First Search, **BFS**)

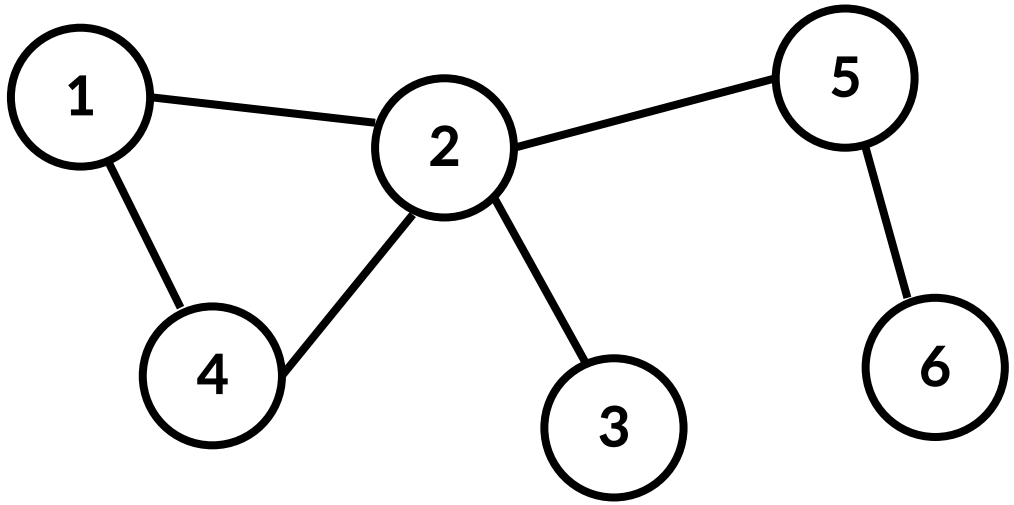
그래프의 탐색 – DFS

- **깊이 우선 탐색** (Depth First Search, **DFS**)

- **깊이 우선** : 무조건 보이는 쪽으로 계속 탐색해간다
- == 현재 정점에서 **갈 수 있는 만큼 최대한 많이 가고, 갈 곳이 없으면 되돌아온다**

- 스택을 이용하여 구현
- 보통 실제 구현에서는 편의를 위해 재귀함수를 사용
  - Call stack!

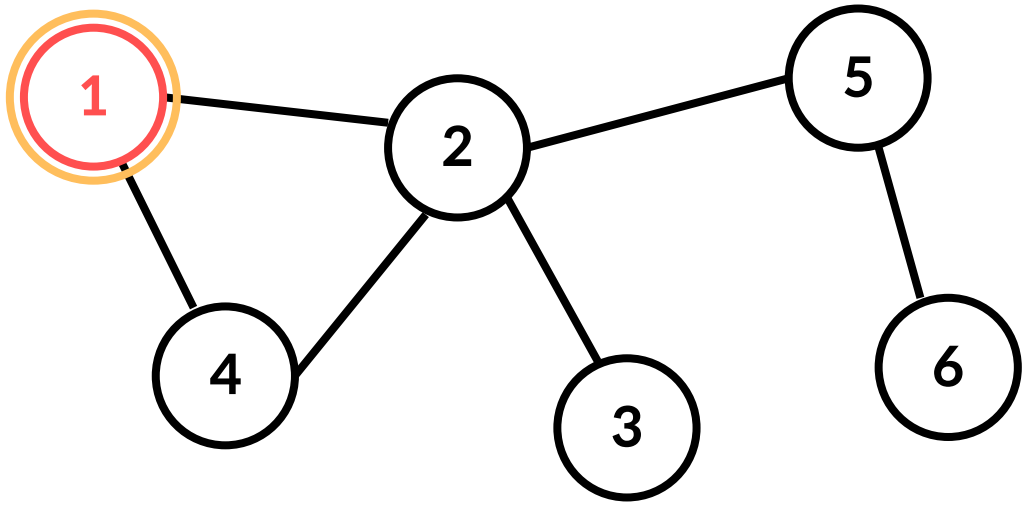
그래프의 탐색 – DFS

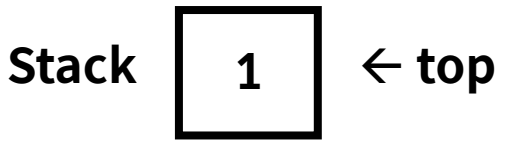


Stack ← top

- 초기 상태 – 스택은 비어 있고, 정점들은 아직 모두 미방문 상태
- 1번 정점부터 DFS를 시작한다고 하자

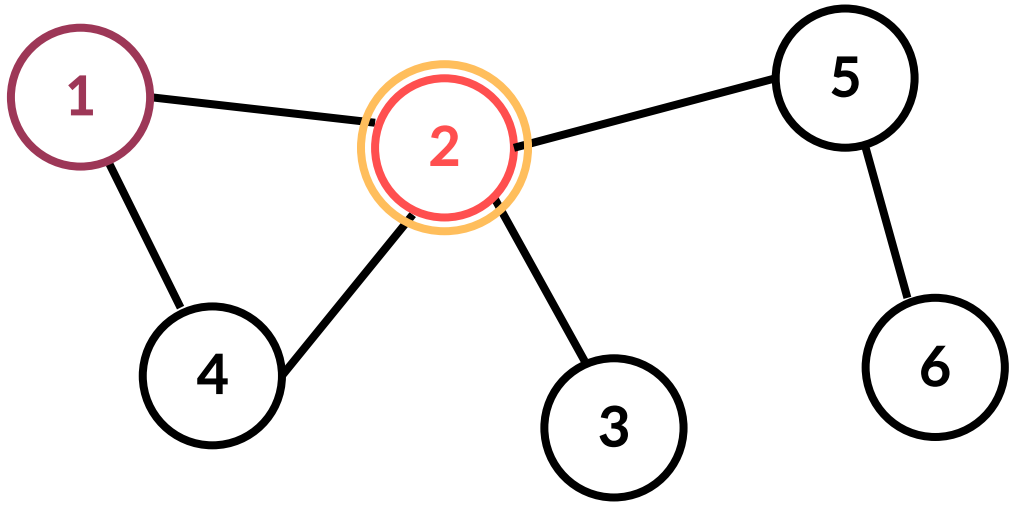
그래프의 탐색 – DFS

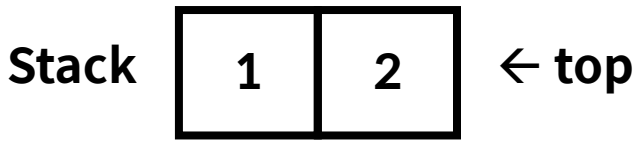




- 시작점인 1번 정점은 (당연히) 아직 미방문 상태이므로 1번 정점 방문, 스택에 추가

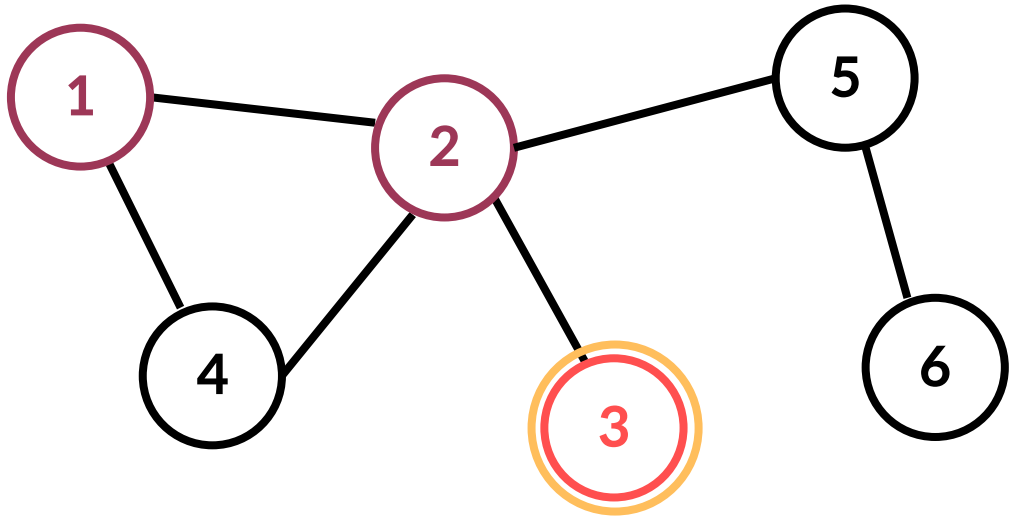
그래프의 탐색 – DFS

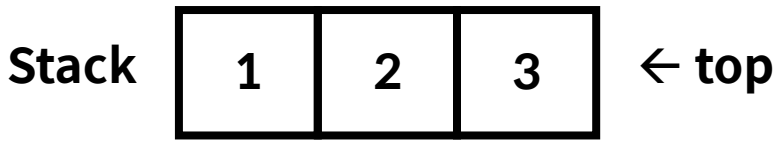




- 1번 정점과 인접한 정점들 중 2번 정점으로 먼저 이동
- 2번 정점도 미방문 상태이므로 2번 정점 방문, 스택에 추가

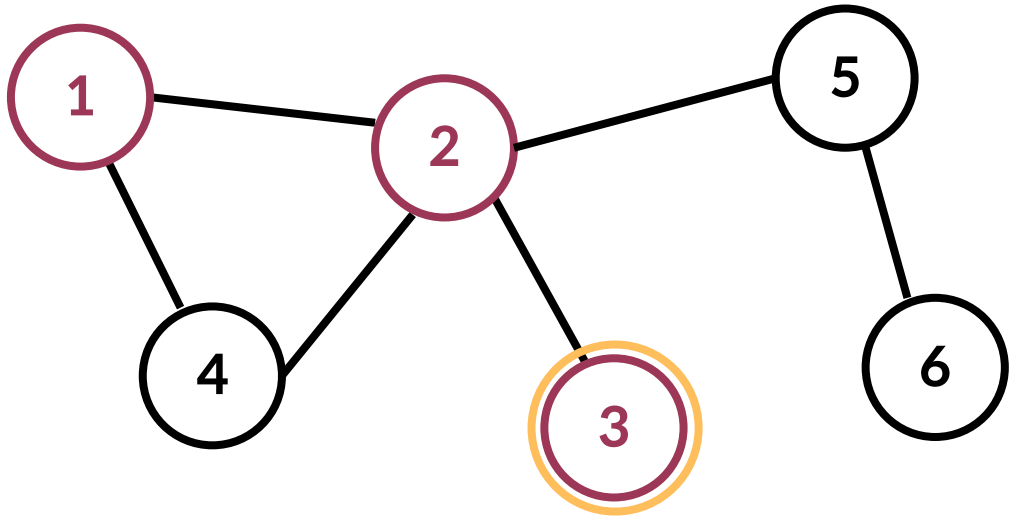
그래프의 탐색 – DFS

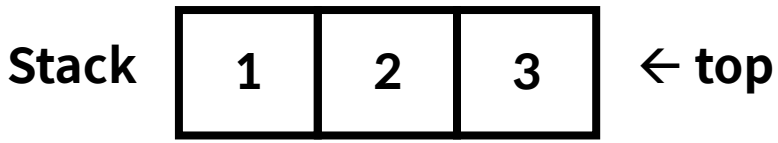




- 2번 정점과 인접한 정점 중 3번 정점으로 이동
- 3번 정점도 미방문 상태 → 3번 방문, 스택에 추가

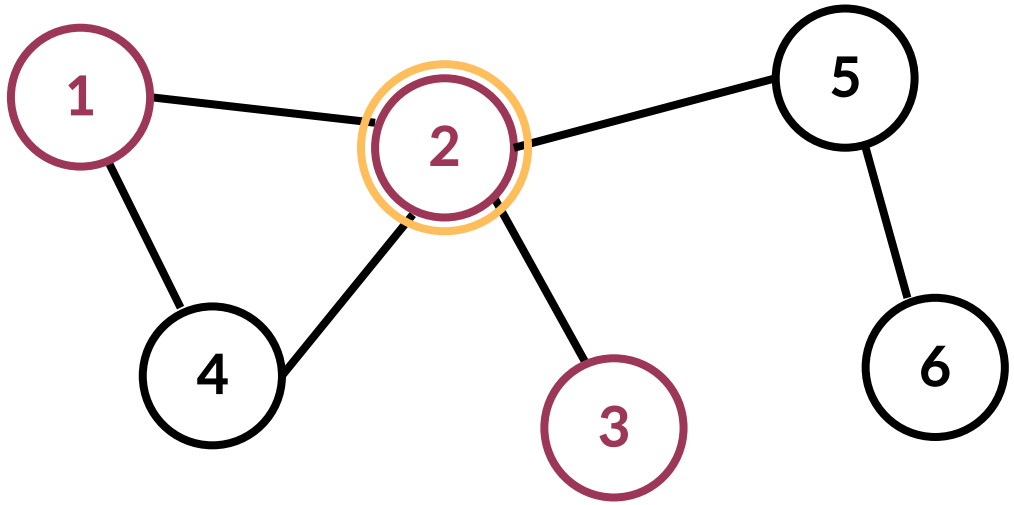
그래프의 탐색 – DFS

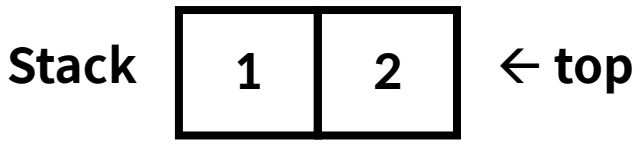




- 3번 정점에서 더 이상 방문할 수 있는 정점이 없다

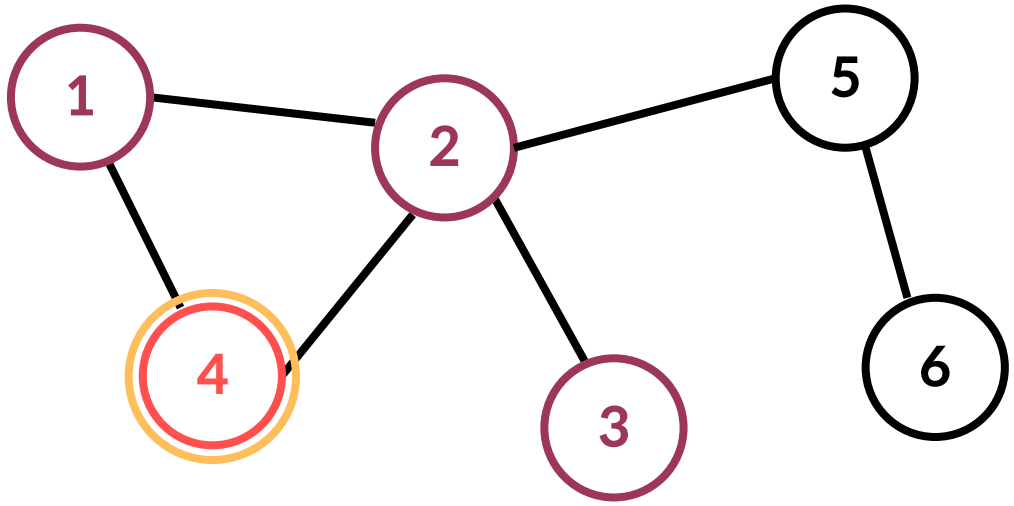
그래프의 탐색 – DFS

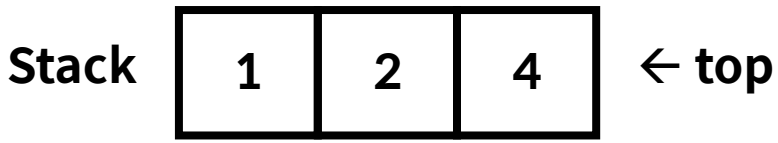




- 스택에서 3번 정점을 제거하고, 다시 스택의 top인 2번 정점으로 돌아가자

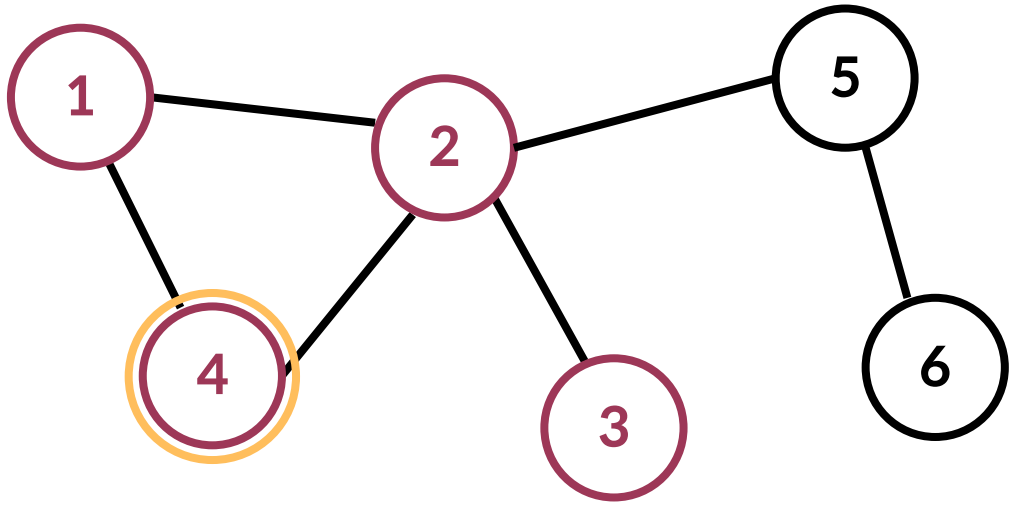
그래프의 탐색 – DFS

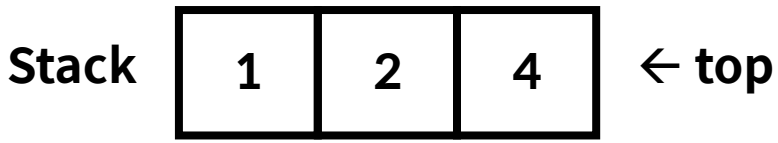




- 2번 정점과 인접한 4번 정점 방문, 스택에 추가

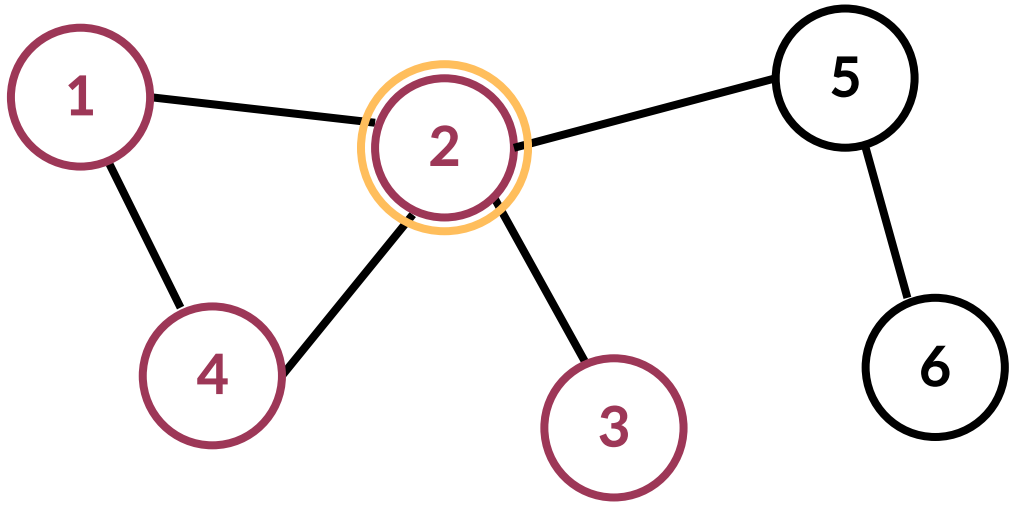
그래프의 탐색 – DFS

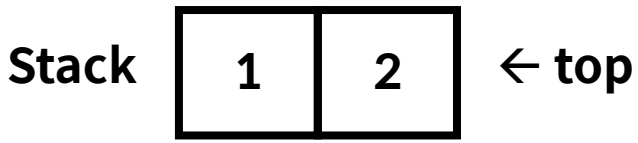




- 4번 정점에서도 더 이상 방문할 수 있는 인접한 정점이 없다

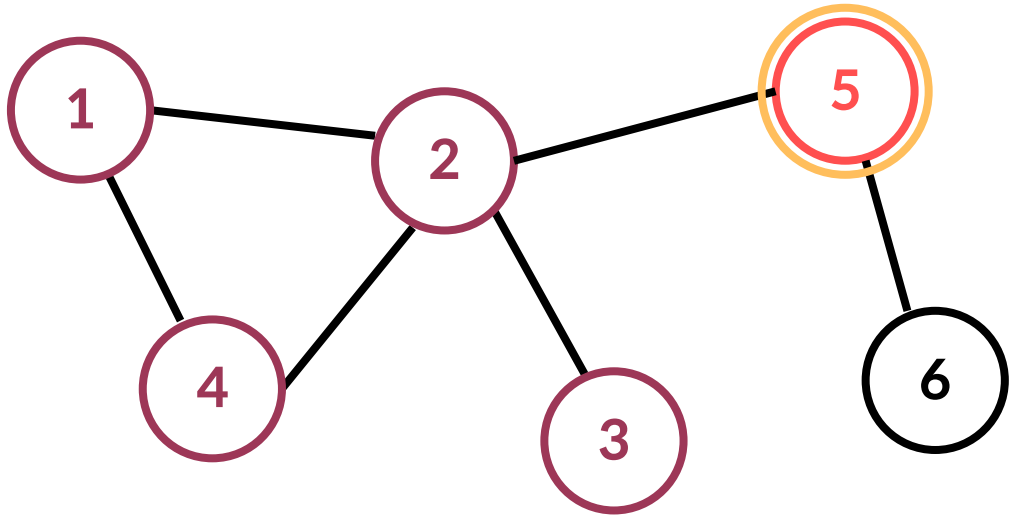
그래프의 탐색 – DFS

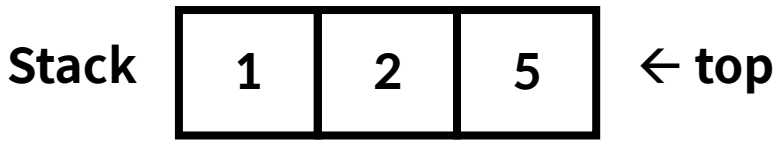




- 스택에서 4번 정점을 제거, 다시 2번 정점으로 돌아왔다

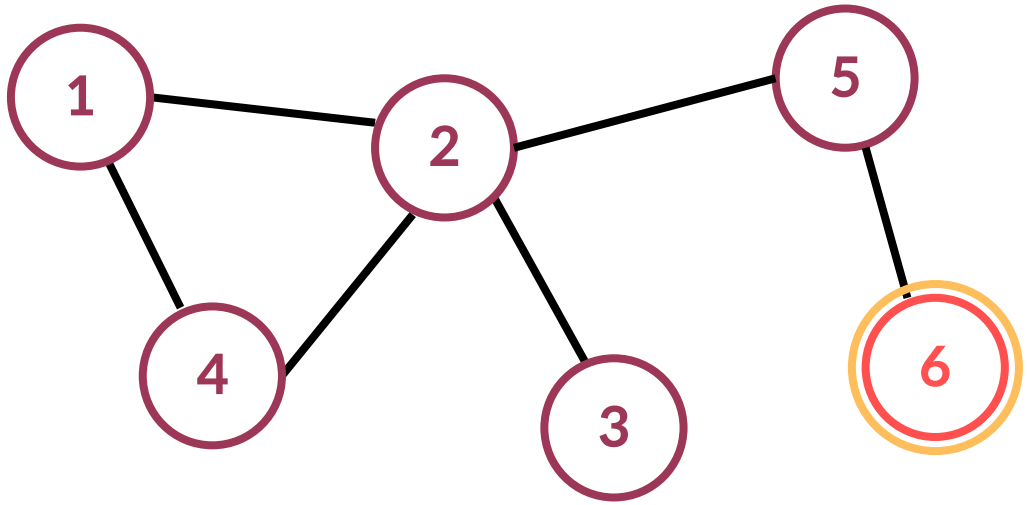
그래프의 탐색 – DFS

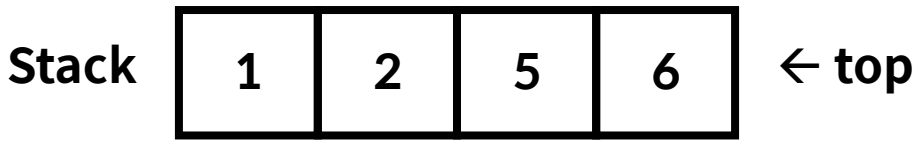




- 2번 정점과 인접한 정점들 중 아직 방문하지 않은 5번 정점으로 이동
- 5번 방문, 스택에 추가

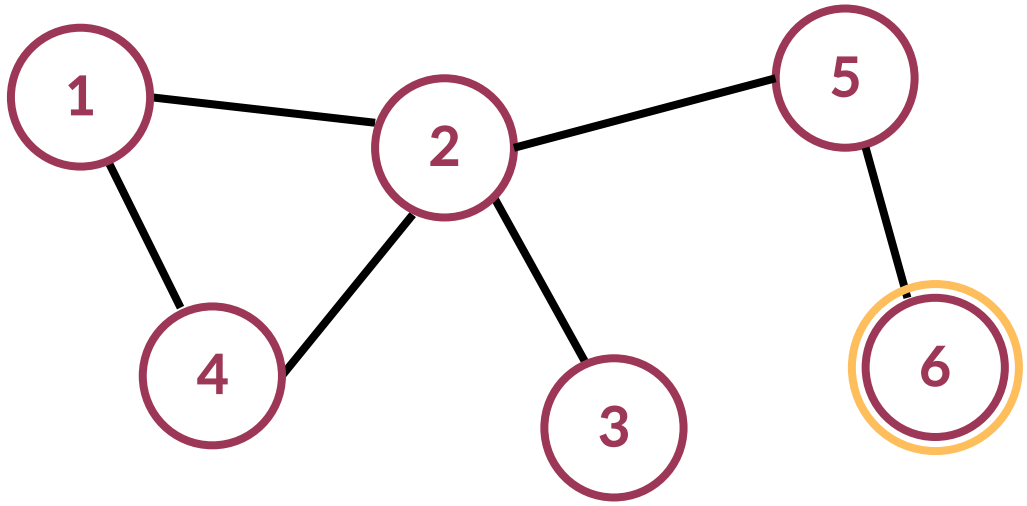
그래프의 탐색 – DFS

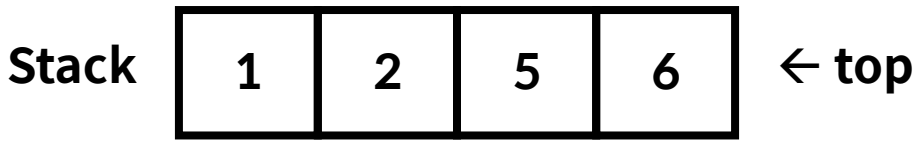




- 5번에서 방문할 수 있는 정점인 6번 정점으로 이동
- 6번 방문, 스택에 추가

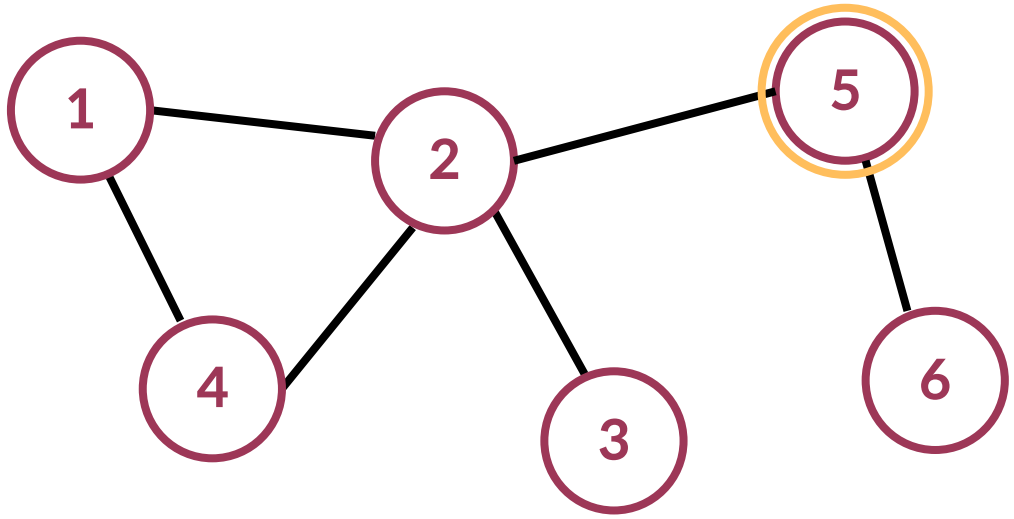
그래프의 탐색 – DFS

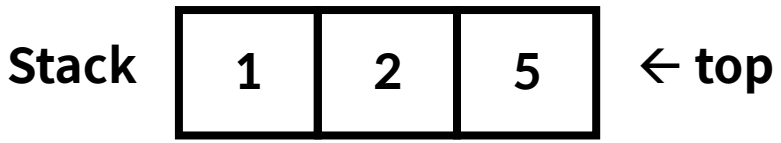




- 6번 정점에서 더 이상 방문할 수 있는 정점이 없다

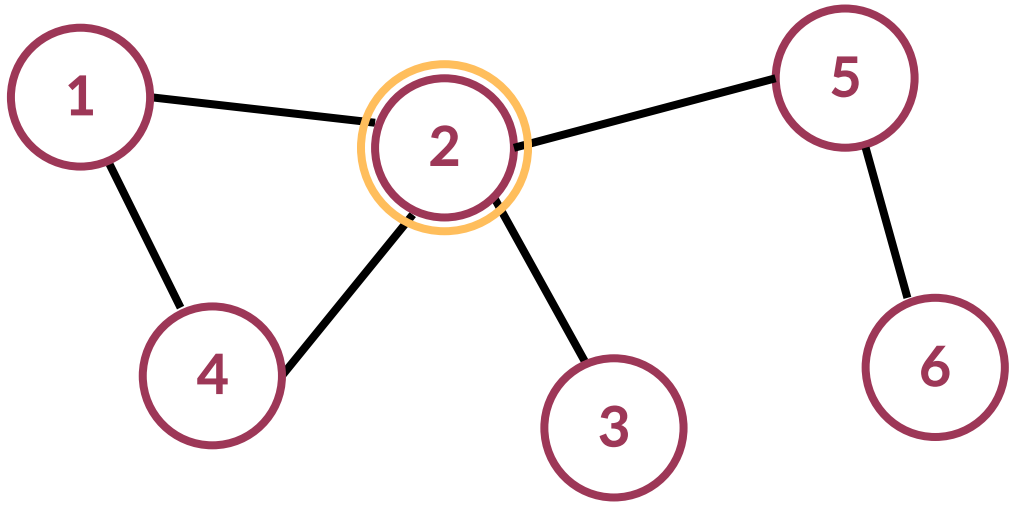
그래프의 탐색 – DFS

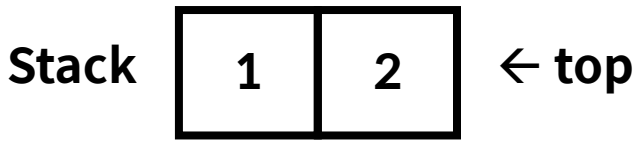




- 스택에서 6번 정점을 제거하고, top인 5번 정점으로 이동

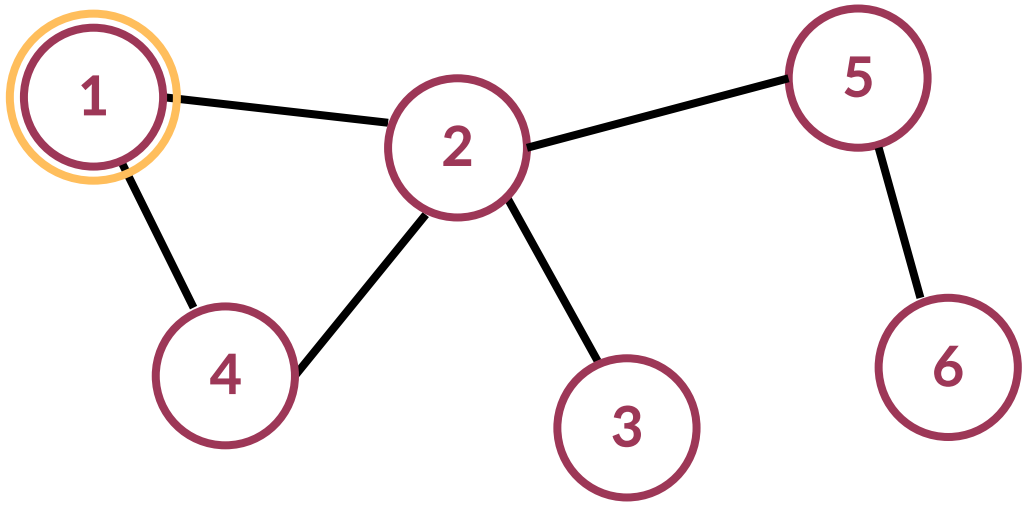
그래프의 탐색 – DFS

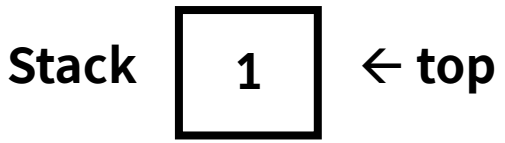




- 5번에서도 방문할 수 있는 정점이 없으니 스택에서 제거, top인 2번 정점으로 이동

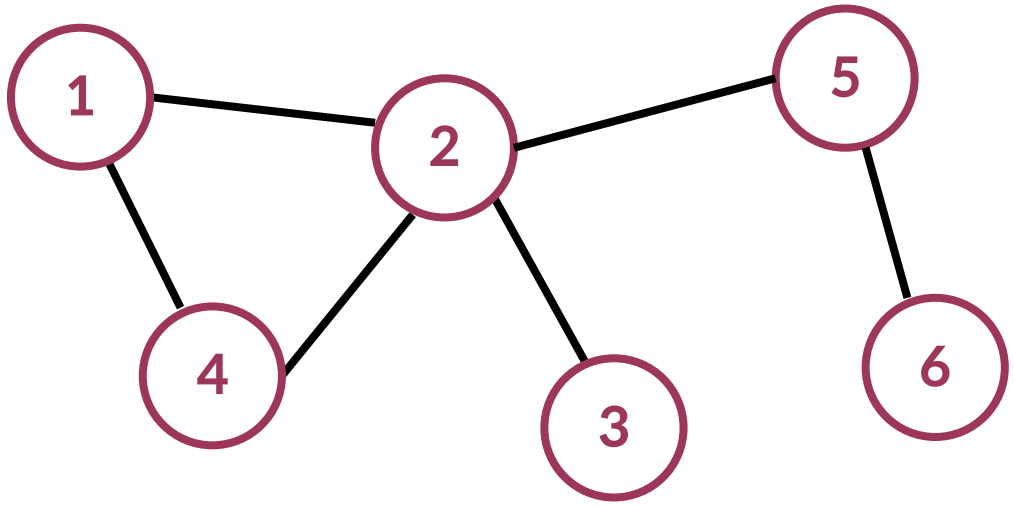
그래프의 탐색 – DFS





- 2번에서도 마찬가지로 인접한 정점들을 모두 방문했기 때문에, 스택에서 제거
- 1번 정점으로 이동

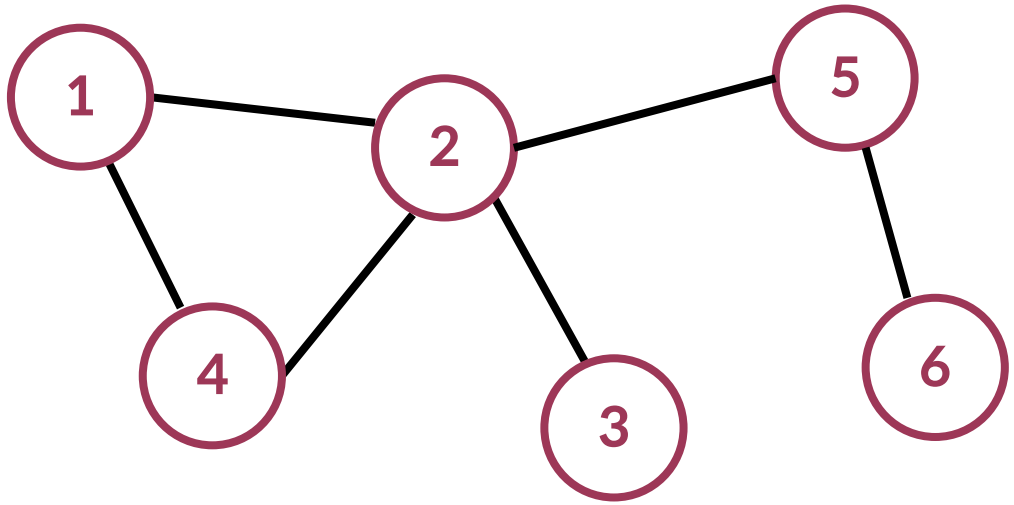
그래프의 탐색 – DFS



Stack ← top

- 1번 정점의 인접한 정점들도 모두 방문 상태이기 때문에 더 이상 진행할 정점이 없다
- 1번 정점도 스택에서 제거

그래프의 탐색 – DFS



Stack ← top

- 모든 정점들이 방문 상태가 되었고, 스택이 비었다
- DFS 끝!

그래프의 탐색 – DFS

- DFS 구현 코드 (재귀함수 사용)

```
void dfs(int x) {  // 정점 x에서 dfs를 수행하는 함수
    visited[x] = true;  // 방문했으니 x를 방문처리한다
    cout << x << ' ';

    // adj[x]는 x와 인접한 (x에서 갈 수 있는) 정점들의 목록
    // 여기서 nx로 다음 정점들을 하나씩 뽑아오자
    for (int nx : adj[x]) {
        if (!visited[nx]) {  // 아직 nx가 미방문 정점이라면
            dfs(nx);  // nx에서 시작하는 dfs를 또 돌린다
        }
    }
}
```

그래프의 탐색 – DFS

- 일단 x를 방문처리한다

```
void dfs(int x) {  // 정점 x에서 dfs를 수행하는 함수
    visited[x] = true;  // 방문했으니 x를 방문처리한다
    cout << x << ' ';

    // adj[x]는 x와 인접한 (x에서 갈 수 있는) 정점들의 목록
    // 여기서 nx로 다음 정점들을 하나씩 뽑아오자
    for (int nx : adj[x]) {
        if (!visited[nx]) {  // 아직 nx가 미방문 정점이라면
            dfs(nx);  // nx에서 시작하는 dfs를 또 돌린다
        }
    }
}
```

그래프의 탐색 – DFS

- 정점 x와 인접한 정점들을 모두 살펴보면서, 아직 방문하지 않은 정점이 있다면 바로 이동

```
void dfs(int x) {  // 정점 x에서 dfs를 수행하는 함수
    visited[x] = true;  // 방문했으니 x를 방문처리한다
    cout << x << ' ';

    // adj[x]는 x와 인접한 (x에서 갈 수 있는) 정점들의 목록
    // 여기서 nx로 다음 정점들을 하나씩 뽑아오자
    for (int nx : adj[x]) {
        if (!visited[nx]) {  // 아직 nx가 미방문 정점이라면
            dfs(nx);  // nx에서 시작하는 dfs를 또 돌린다
        }
    }
}
```

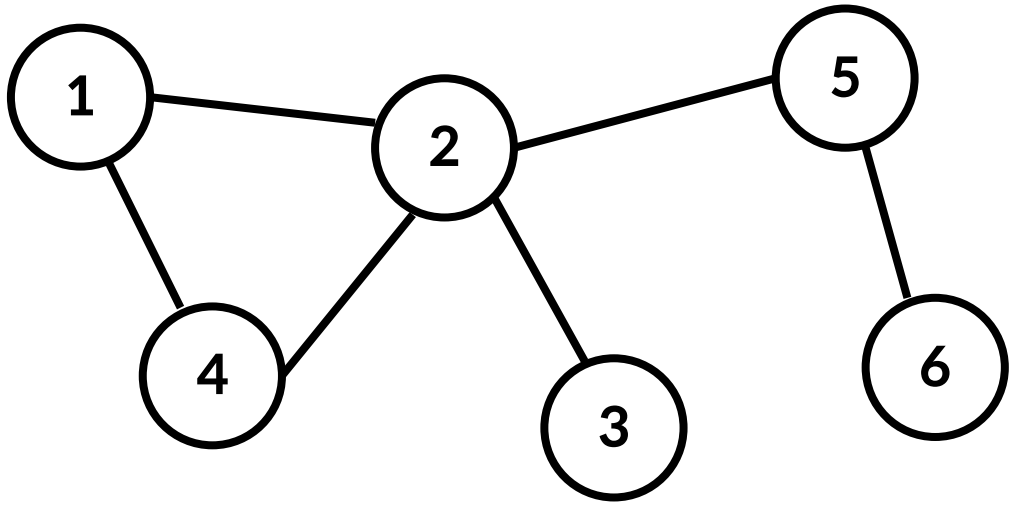
그래프의 탐색 – BFS

- **너비 우선 탐색** (Breadth First Search, **BFS**)

- **너비 우선** : 현재 정점에서 갈 수 있는 곳은 일단 다 돌고, 그 다음 정점으로 이동
- == 현재 정점과 인접한 정점들을 일단 모두 탐색,
  그 다음 정점으로 이동해서 또 인접한 정점들을 모두 탐색, …

- 큐를 이용하여 구현

그래프의 탐색 – BFS

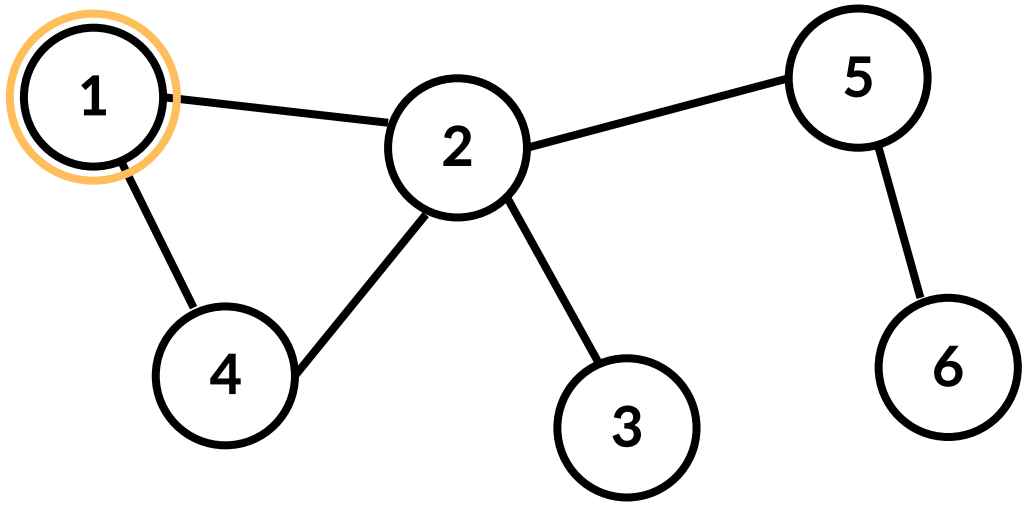


Queue

front → ← rear

- 초기 상태 – 큐는 비어 있고, 정점들은 아직 모두 미방문 상태
- 마찬가지로 1번 정점부터 BFS를 시작한다고 하자

그래프의 탐색 – BFS

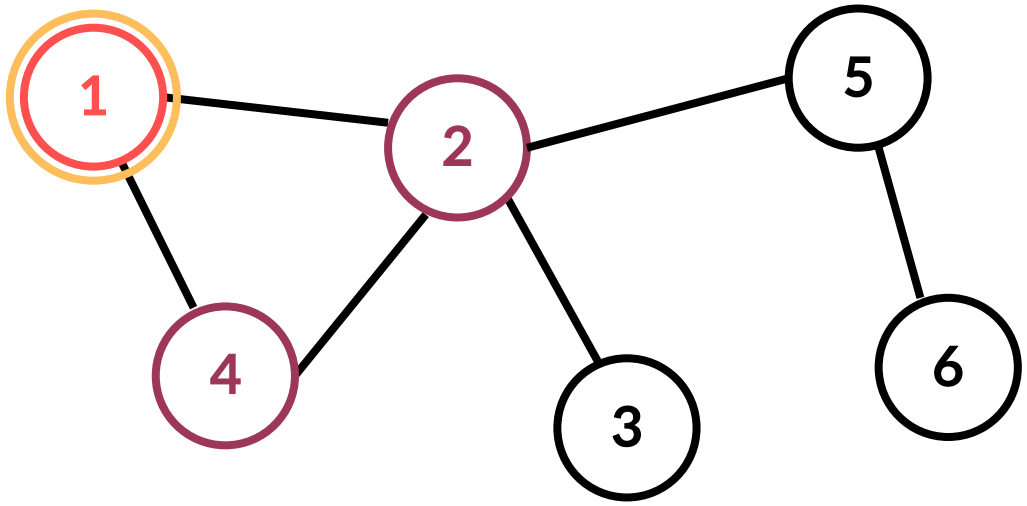


Queue

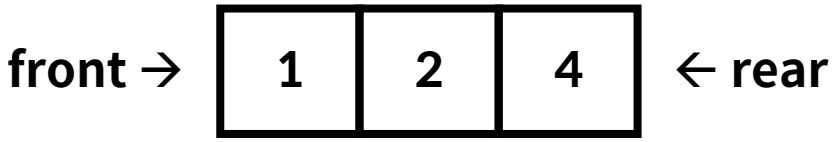
front → 1 **← rear**

- 우선 1번 정점을 큐에 넣고 시작한다

그래프의 탐색 – BFS

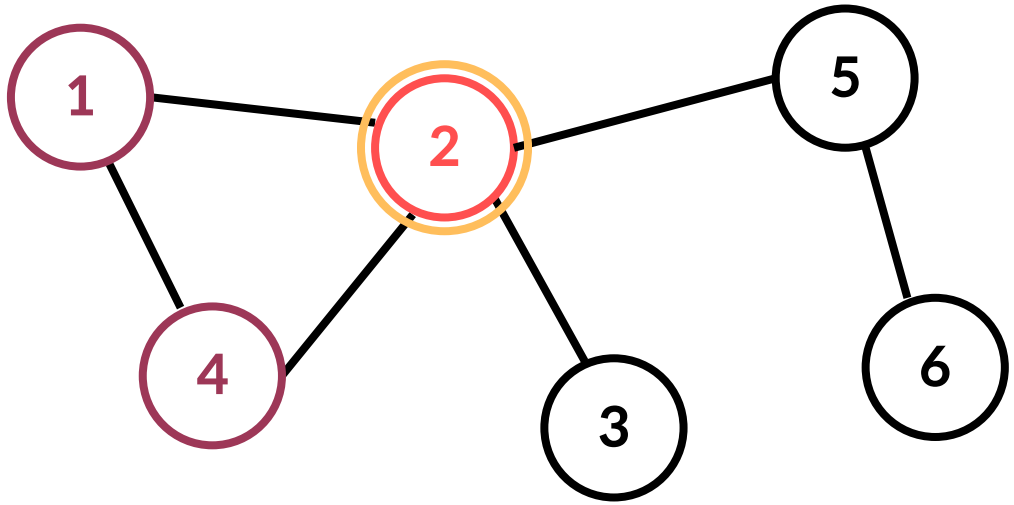


Queue



- front인 1번 정점과 인접한 정점들 중 방문할 수 있는 정점들을 모두 큐에 넣는다
- 2번, 4번 정점이 방문처리되고, 큐에 들어간다

그래프의 탐색 – BFS

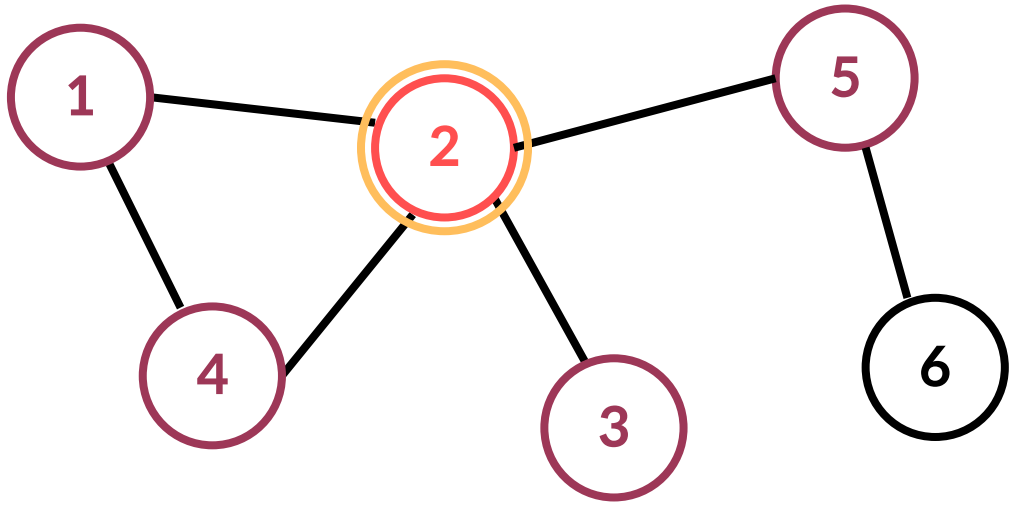


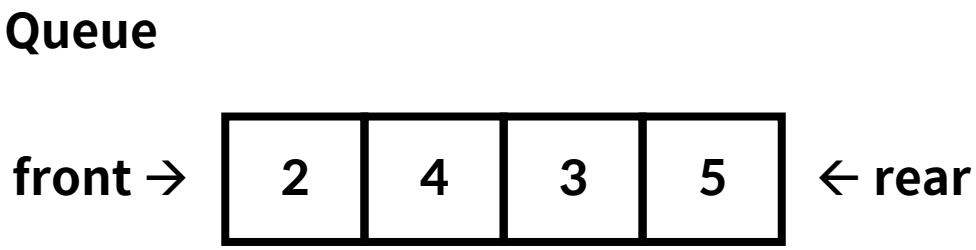
Queue

front → | 2 | 4 | **← rear**

- 1번 정점에서 할 수 있는 게 모두 끝났으면, 큐에서 1번 정점 (front)를 제거
- 다음으로 볼 정점은 front인 2번 정점

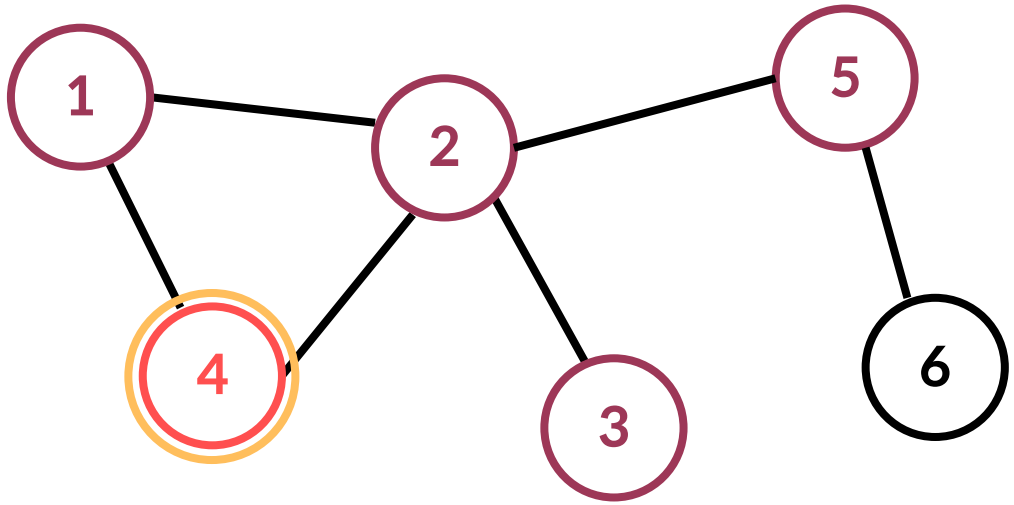
그래프의 탐색 – BFS

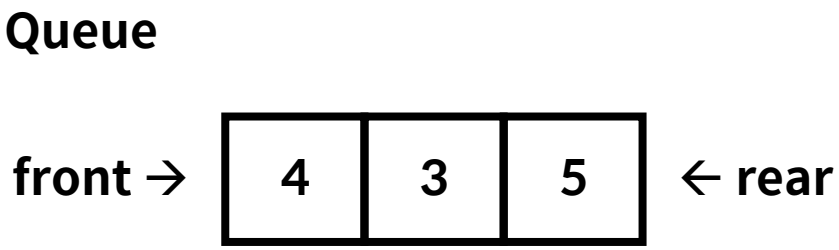




- 2번 정점과 인접한 정점들 중 아직 방문하지 않은 정점들을 모두 큐에 넣는다
- 4번은 이미 방문한 정점이고, 3번과 5번 정점이 큐에 들어간다

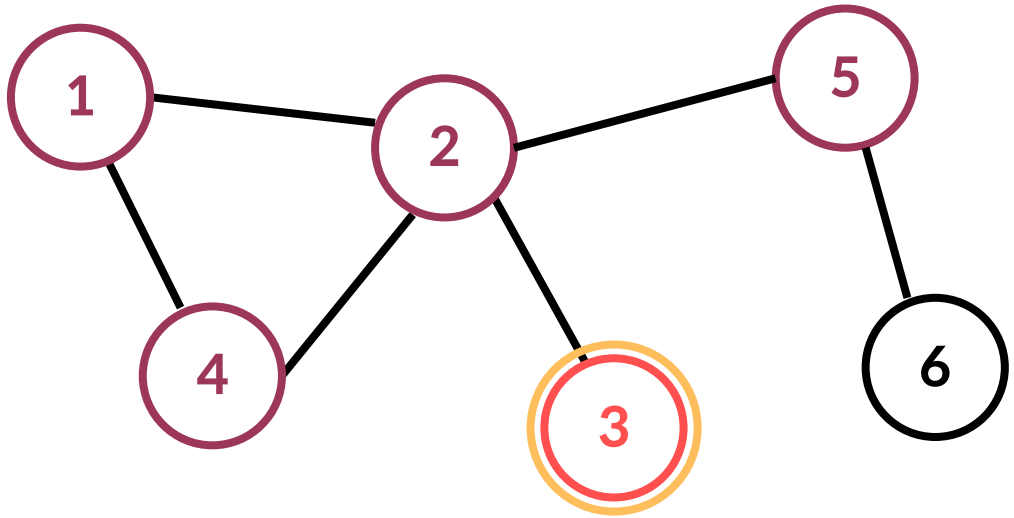
그래프의 탐색 – BFS





- 2번 정점도 끝났으니 큐의 front에서 2번 정점 제거
- 다음으로 볼 정점은 4번 정점

그래프의 탐색 – BFS

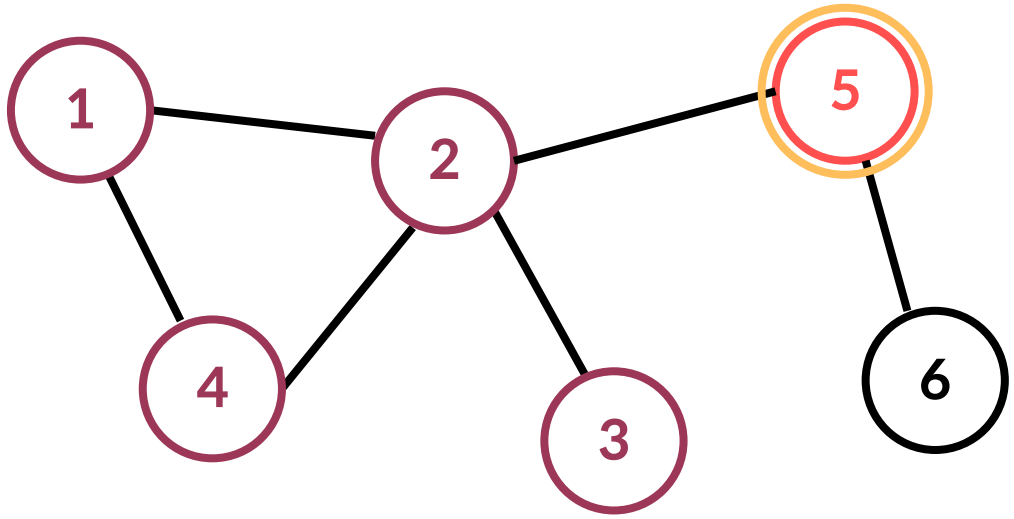


Queue

front → | 3 | 5 | **← rear**

- 4번 정점의 인접한 정점들은 아까 모두 방문했기 때문에, 4번에서 할 수 있는 게 없다
- 큐에서 바로 빼고, 다음으로 볼 정점은 3번 정점

그래프의 탐색 – BFS

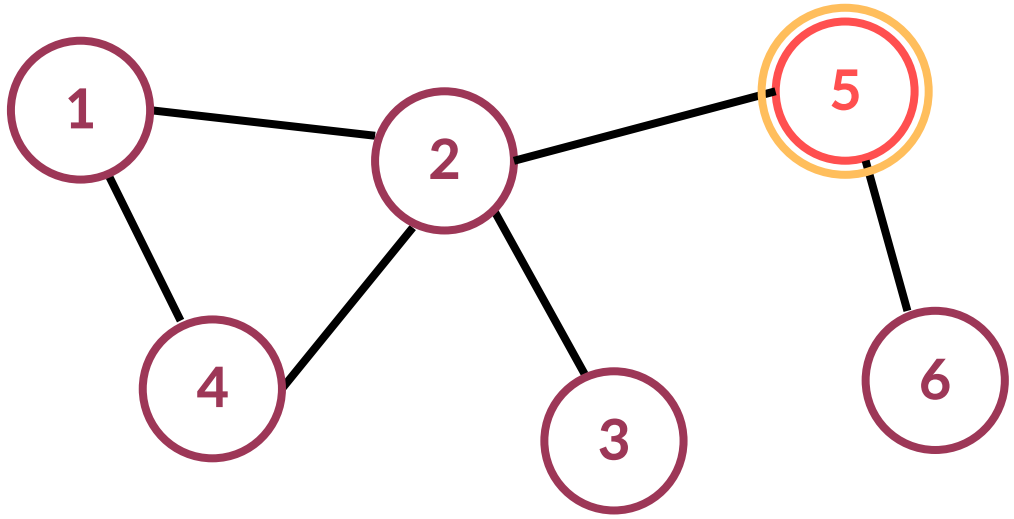


Queue

front → 5 **← rear**

- 3번 정점에서도 할 수 있는 게 없다
- 큐에서 빼고, 다음으로 볼 정점은 5번 정점

그래프의 탐색 – BFS

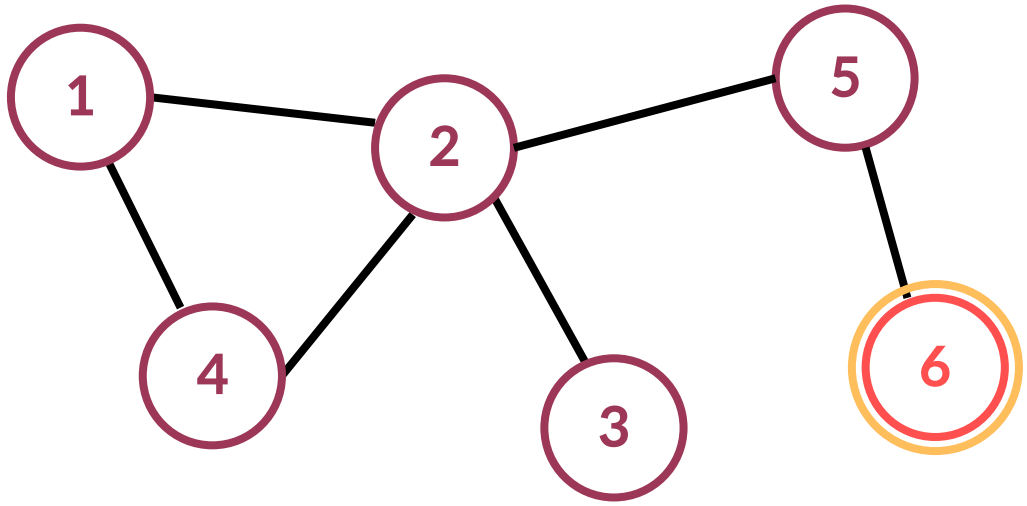


Queue

front → | 5 | 6 | **← rear**

- 5번 정점의 인접한 정점들 중, 6번 정점을 아직 방문하지 않았다
- 6번 정점을 방문처리하고 큐에 넣자

그래프의 탐색 – BFS

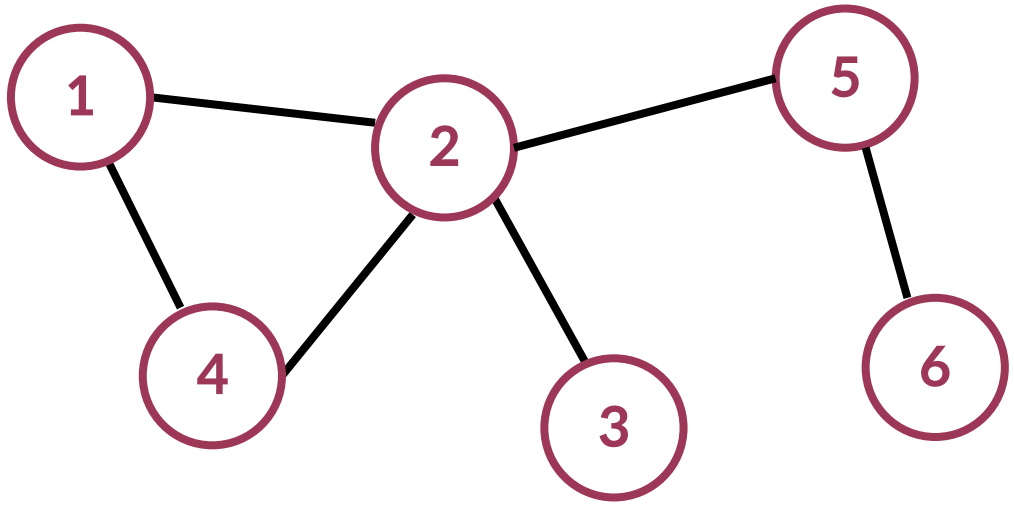


Queue

front → 6 **← rear**

- 5번 정점에서 할 수 있는 게 없으니 큐에서 제거
- 6번 정점의 주변을 보자

그래프의 탐색 – BFS

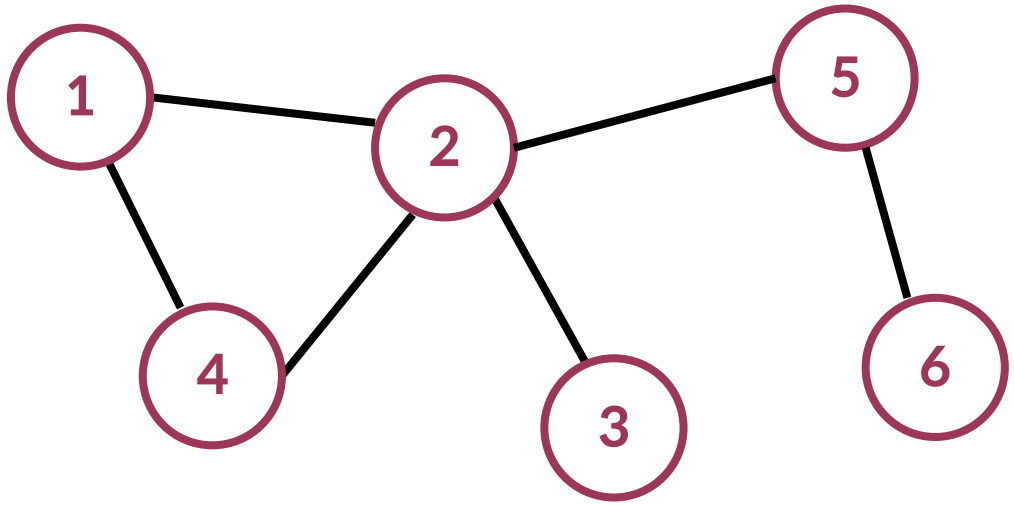


Queue

front → ← rear

- 6번 정점에서도 할 수 있는 게 없다
- 6번 정점도 큐에서 빼면, 큐가 비어서 더 이상 진행할 다음 정점이 없다

그래프의 탐색 – BFS



Queue

front → ← rear

- 모든 정점이 방문 상태가 되었고, 큐가 비었다
- BFS 끝!

그래프의 탐색 – BFS

- BFS 구현 코드

```
queue<int> q;
// 처음에 시작 정점은 큐에 먼저 넣고, 방문처리하고 시작한다
q.push(1); visited[1] = true;
while (!q.empty()) {  // 큐가 빌 때까지 (bfs가 끝날 때까지) 반복
    int x = q.front(); q.pop();  // 큐의 front가 현재 봐야 할 정점
    cout << x << '\n';

    // dfs와 마찬가지로 adj[x]에서 하나씩 뽑아온다
    for (int nx : adj[x]) {
        if (!visited[nx]) {  // 아직 미방문 정점이라면
            q.push(nx); visited[nx] = true;  // 큐에 넣고, 방문처리까지
        }
    }
}
```

그래프의 탐색 – BFS

- 시작 정점을 먼저 넣고 시작

```
queue<int> q;
// 처음에 시작 정점은 큐에 먼저 넣고, 방문처리하고 시작한다
q.push(1); visited[1] = true;
while (!q.empty()) {  // 큐가 빌 때까지 (bfs가 끝날 때까지) 반복
    int x = q.front(); q.pop();  // 큐의 front가 현재 봐야 할 정점
    cout << x << '\n';

    // dfs와 마찬가지로 adj[x]에서 하나씩 뽑아온다
    for (int nx : adj[x]) {
        if (!visited[nx]) {  // 아직 미방문 정점이라면
            q.push(nx); visited[nx] = true;  // 큐에 넣고, 방문처리까지
        }
    }
}
```

그래프의 탐색 – BFS

- 큐가 빌 때까지 front의 인접한 정점들을 보며 큐에 계속 넣어준다

```
queue<int> q;
// 처음에 시작 정점은 큐에 먼저 넣고, 방문처리하고 시작한다
q.push(1); visited[1] = true;
while (!q.empty()) {  // 큐가 빌 때까지 (bfs가 끝날 때까지) 반복
    int x = q.front(); q.pop();  // 큐의 front가 현재 봐야 할 정점
    cout << x << '\n';

    // dfs와 마찬가지로 adj[x]에서 하나씩 뽑아온다
    for (int nx : adj[x]) {
        if (!visited[nx]) {  // 아직 미방문 정점이라면
            q.push(nx); visited[nx] = true;  // 큐에 넣고, 방문처리까지
        }
    }
}
```

그래프의 탐색 – BFS

- BFS 구현 코드

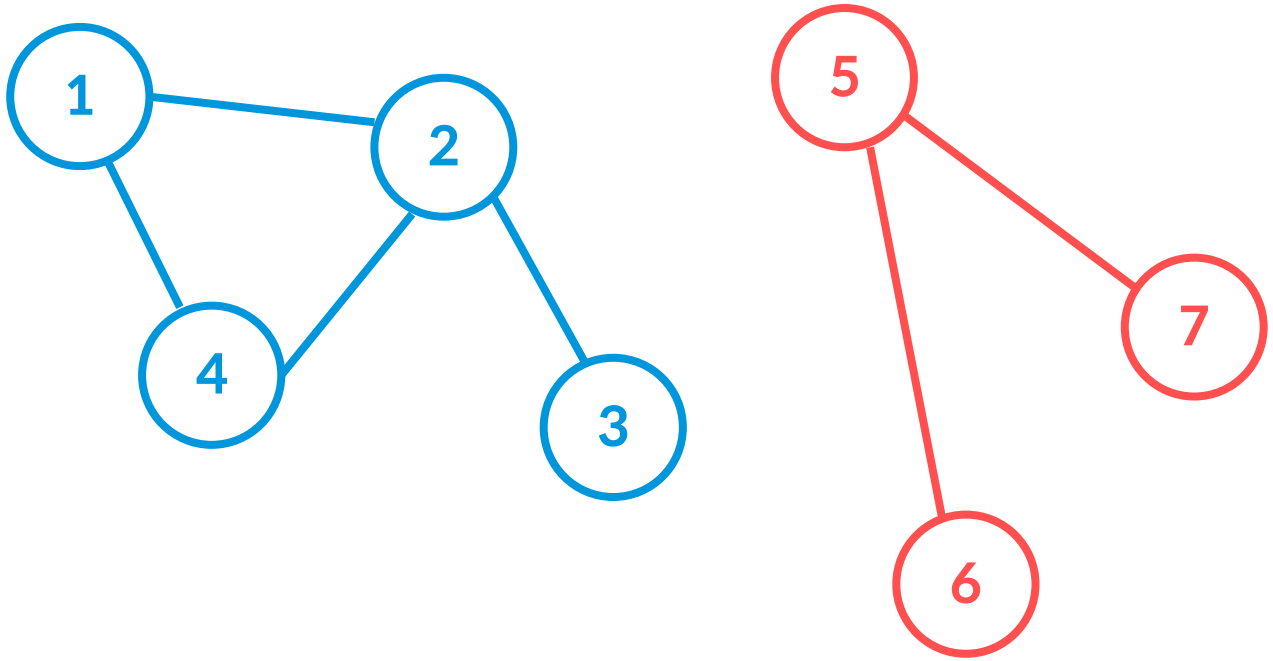
```
queue<int> q;
// 처음에 시작 정점은 큐에 먼저 넣고, 방문처리하고 시작한다
q.push(1); visited[1] = true;
while (!q.empty()) {  // 큐가 빌 때까지 (bfs가 끝날 때까지) 반복
    int x = q.front(); q.pop();  // 큐의 front가 현재 봐야 할 정점
    cout << x << '\n';

    // dfs와 마찬가지로 adj[x]에서 하나씩 뽑아온다
    for (int nx : adj[x]) {
        if (!visited[nx]) {  // 아직 미방문 정점이라면
            q.push(nx); visited[nx] = true;  // 큐에 넣고, 방문처리까지
        }
    }
}
```

- 0x02

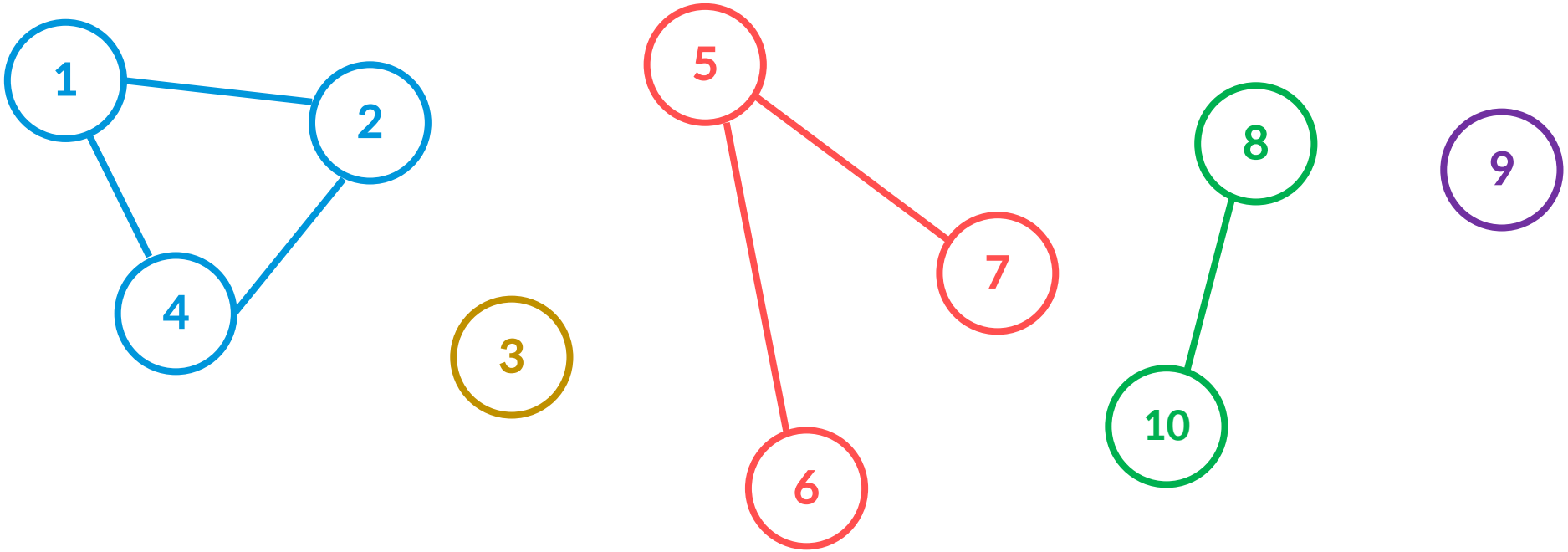
탐색 응용 – 연결 요소의 개수

Remind : 연결 요소



- 2개

Remind : 연결 요소



- 5개

연결 요소의 개수

#11724 연결 요소의 개수

- 방향 없는 그래프가 주어졌을 때,
  연결 요소 (Connected Component)의 개수를 구하는 프로그램을 작성하시오.

- 어떻게 하면 될까?

연결 요소의 개수

#11724 연결 요소의 개수

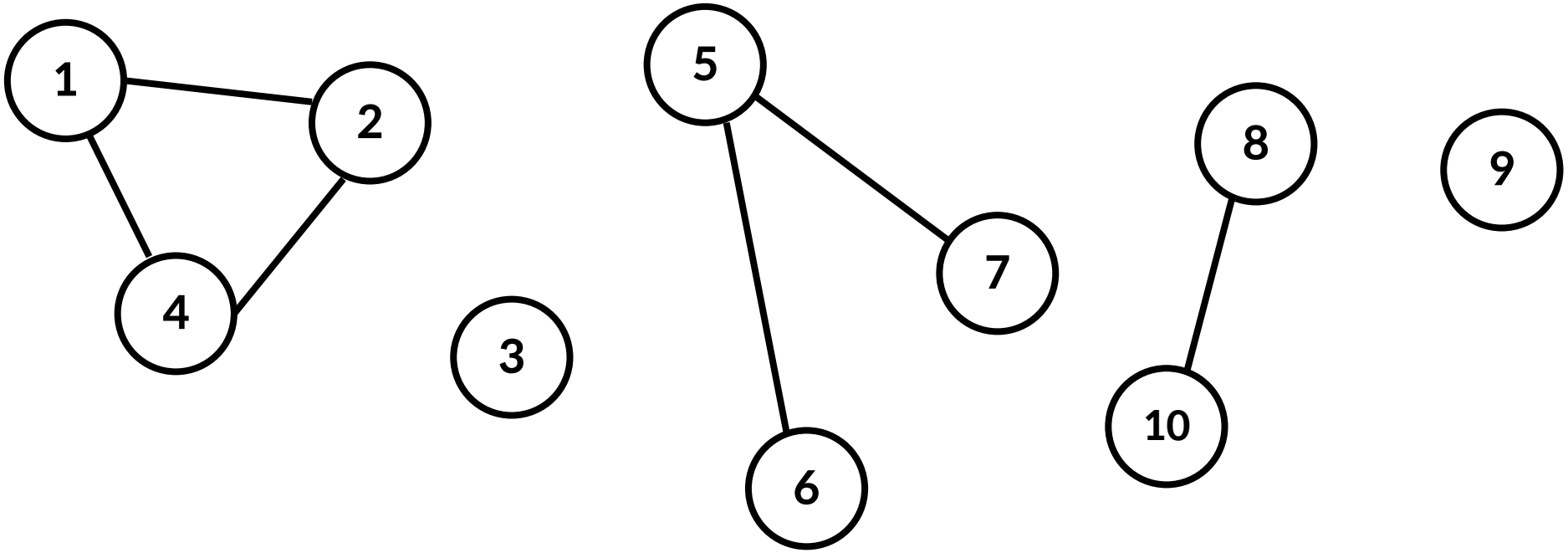
- 단순히 DFS 하나만으로 풀 수 있다!
- 아직 방문하지 않은 정점에서 DFS를 한 번씩 돌려보면 된다
- **어느 한 점에서 DFS를 돌리면,**
  같은 연결 요소 내의 모든 점들이 방문처리될 것이기 때문이다

- **main에서 DFS 탐색을 시작하는 횟수**가 연결 요소의 개수가 된다

- DFS만 되는 것은 아니고, BFS도 가능하다
  - 과정을 알고 나면 당연함

연결 요소의 개수

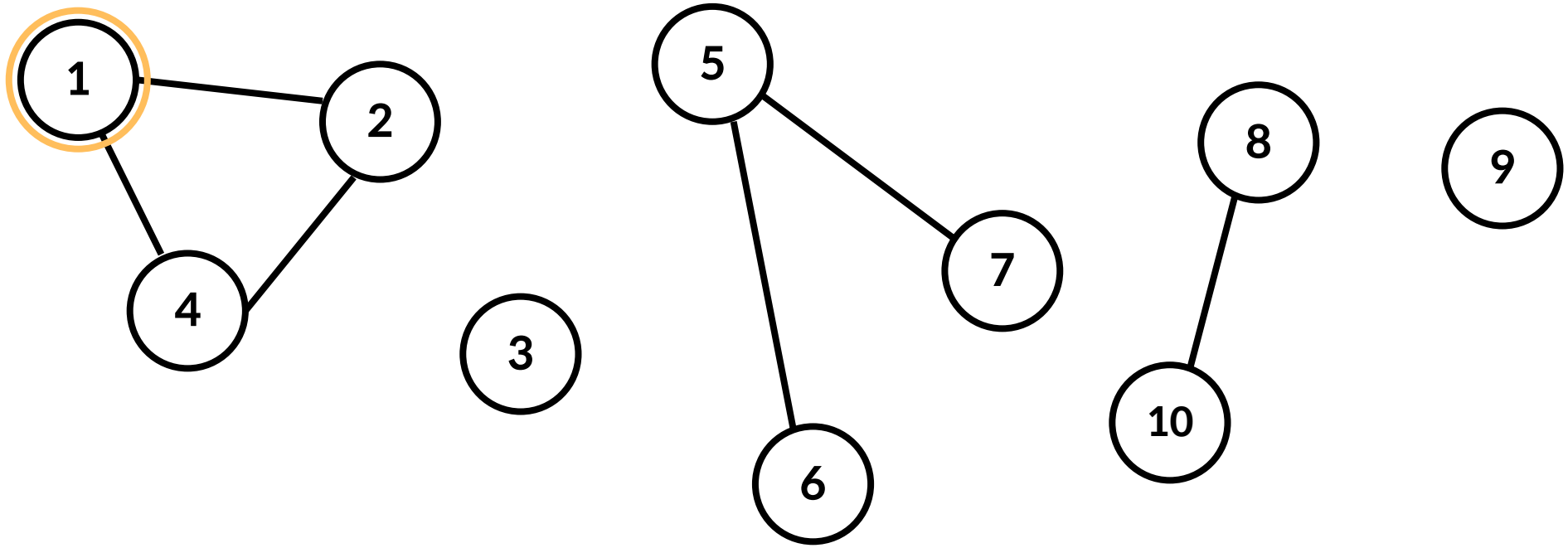
#11724 연결 요소의 개수



- 이런 그래프가 있을 때, 연결 요소의 개수를 구해보자
- DFS 탐색을 시작하는 횟수에 주목

연결 요소의 개수

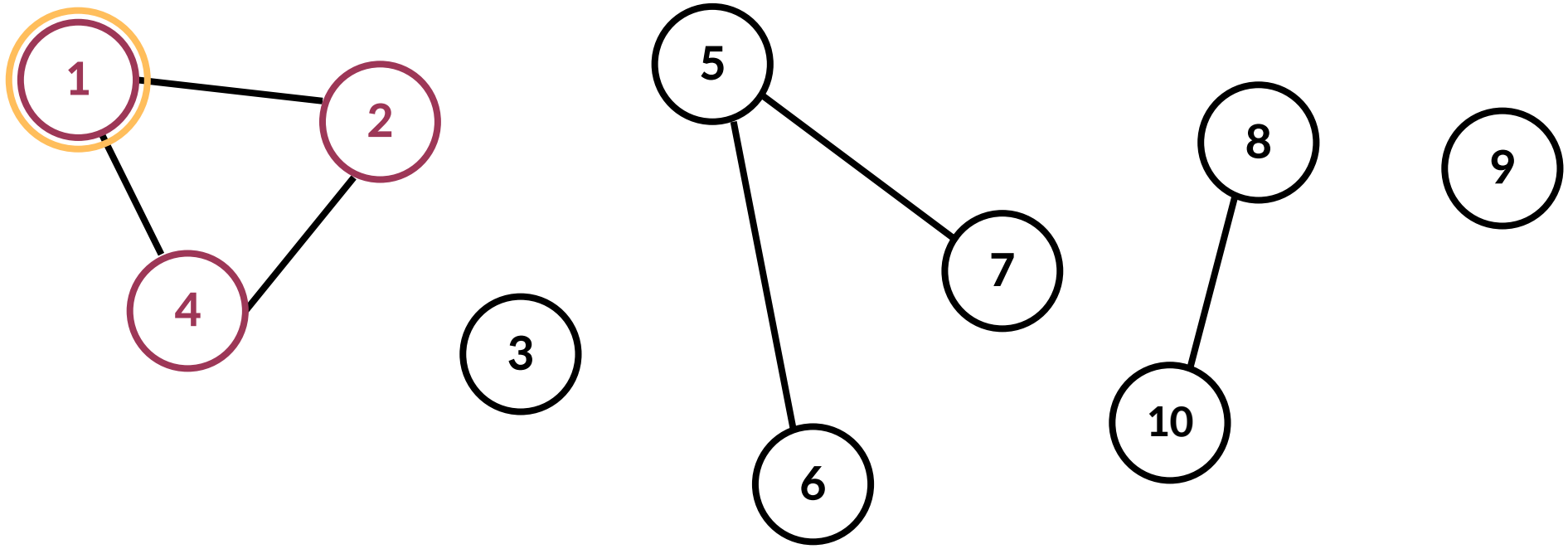
#11724 연결 요소의 개수



- 1번 정점이 아직 미방문 상태이므로, 1번 정점에서 DFS를 시작한다 **(DFS 1회째)**

연결 요소의 개수

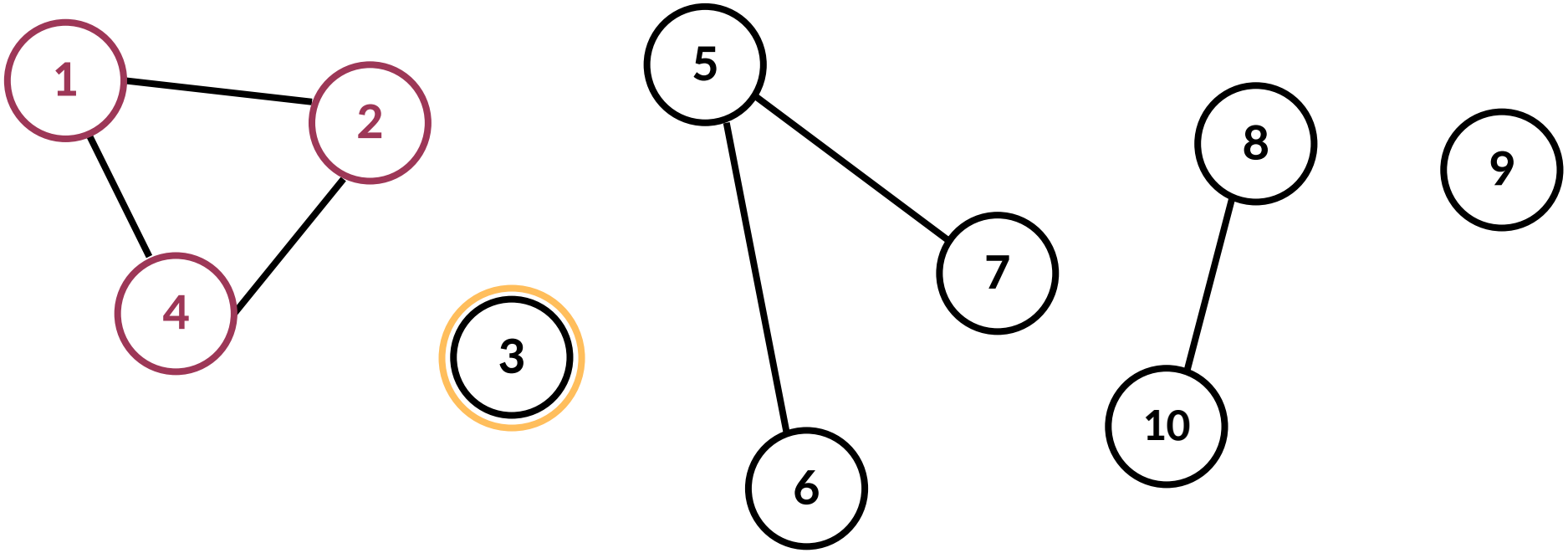
#11724 연결 요소의 개수



- 1번 정점이 속한 연결 요소의 정점들이 모두 방문 처리된다

연결 요소의 개수

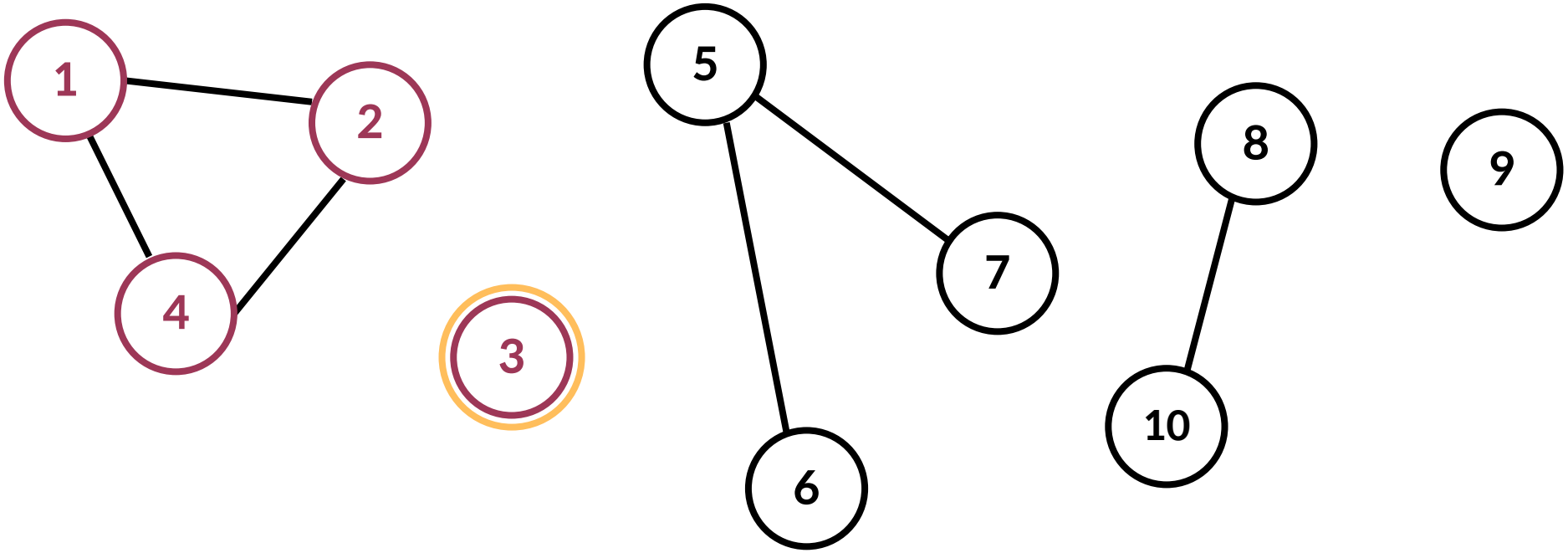
#11724 연결 요소의 개수



- 2번 정점은 이미 방문 상태이므로 스킵
- 3번 정점에서 DFS를 시작한다 (DFS 2번째)

연결 요소의 개수

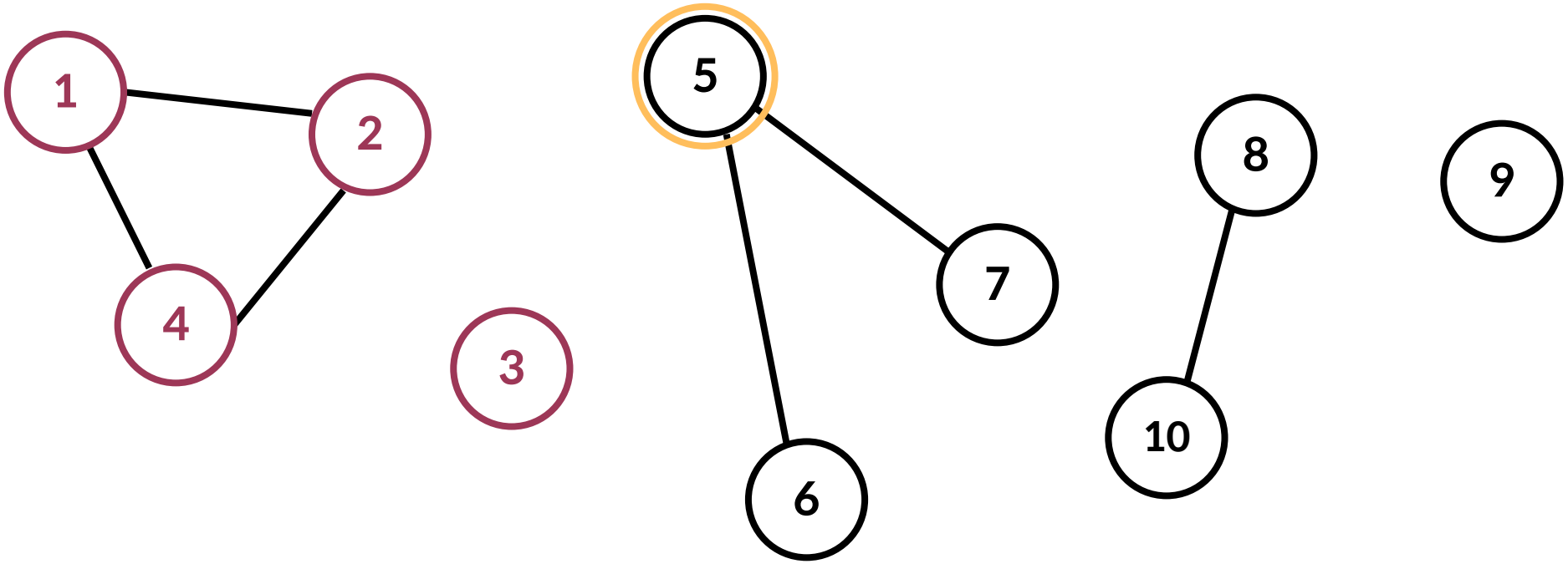
#11724 연결 요소의 개수



- 3번 정점은 아무 정점과도 연결되어 있지 않으므로 3번 정점만 방문처리되고, DFS 끝

연결 요소의 개수

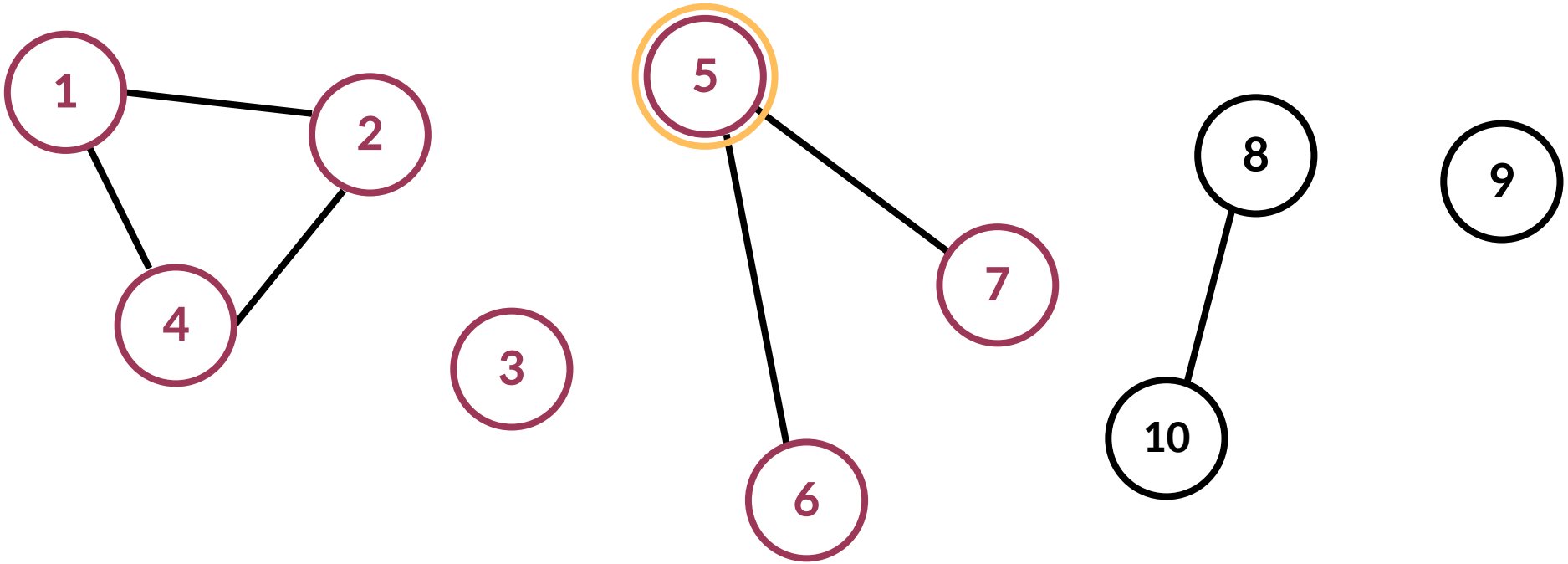
#11724 연결 요소의 개수



- 4번 정점도 방문처리 되었으므로 스킵
- 5번 정점은 미방문 상태이므로 5번 정점에서 DFS를 시작한다 (DFS 3번째)

연결 요소의 개수

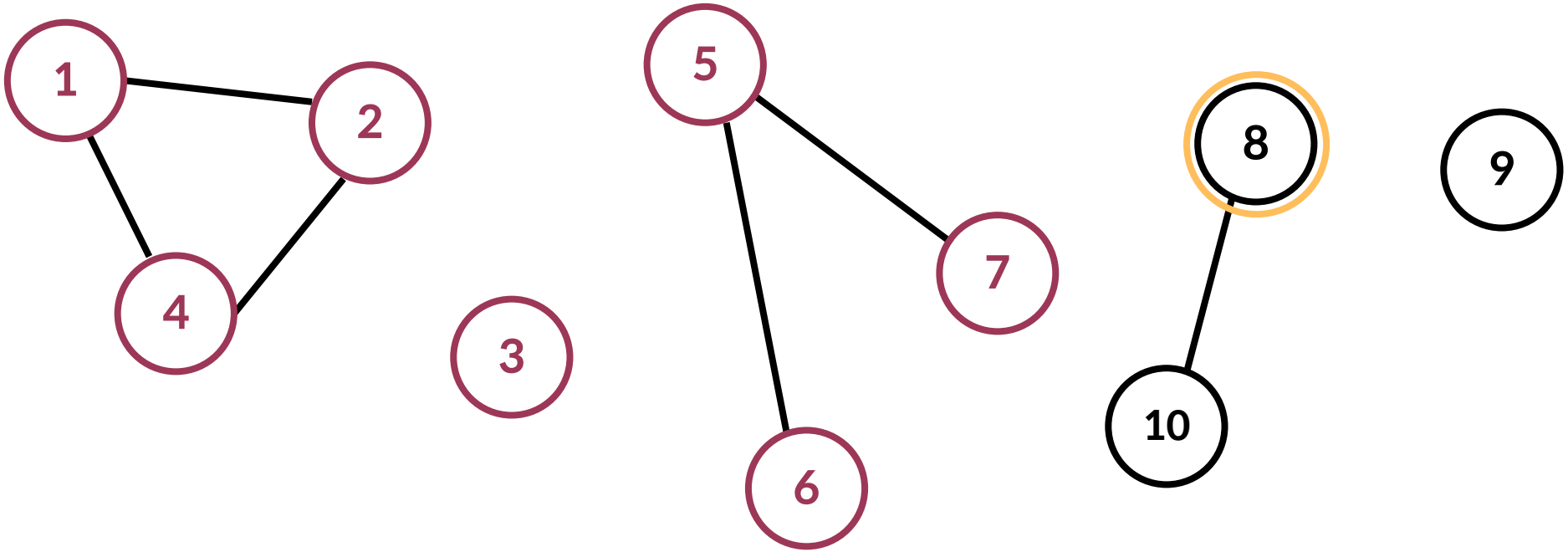
#11724 연결 요소의 개수



- 5, 6, 7번 정점이 모두 방문처리되고, DFS 끝

연결 요소의 개수

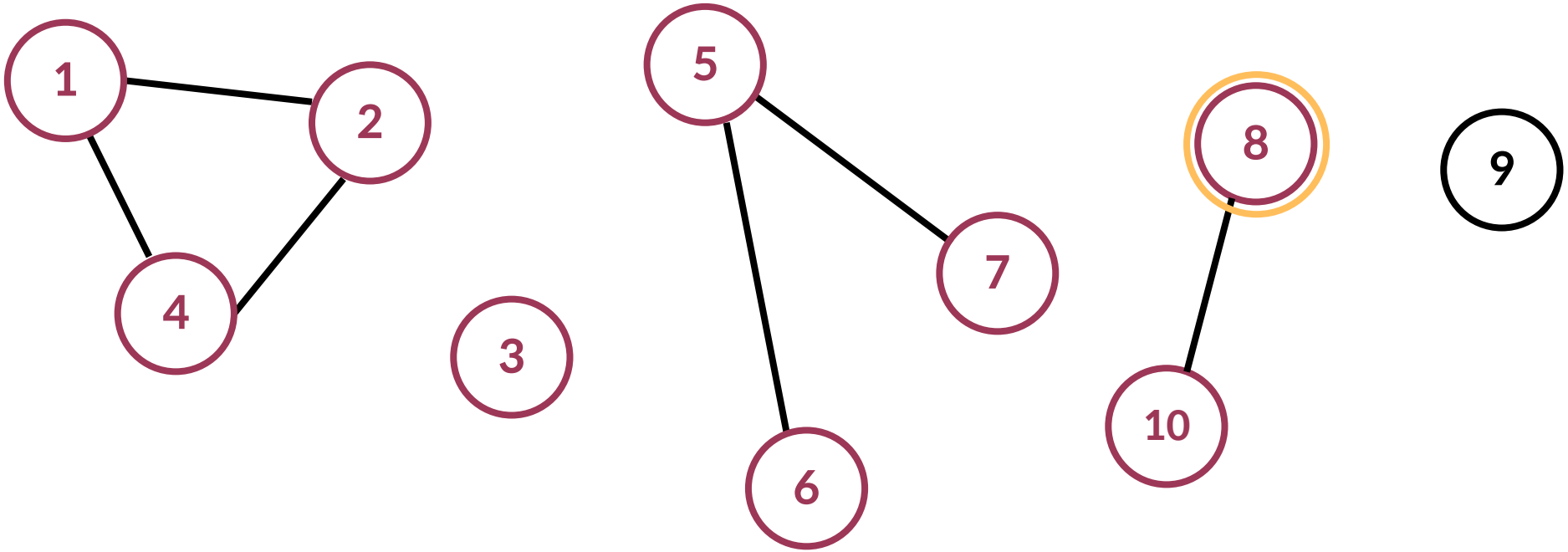
#11724 연결 요소의 개수



- 6, 7번 정점은 방문처리 되었으므로, 둘 다 스킵
- 8번 정점에서 또 DFS 호출 (DFS 4번째)

연결 요소의 개수

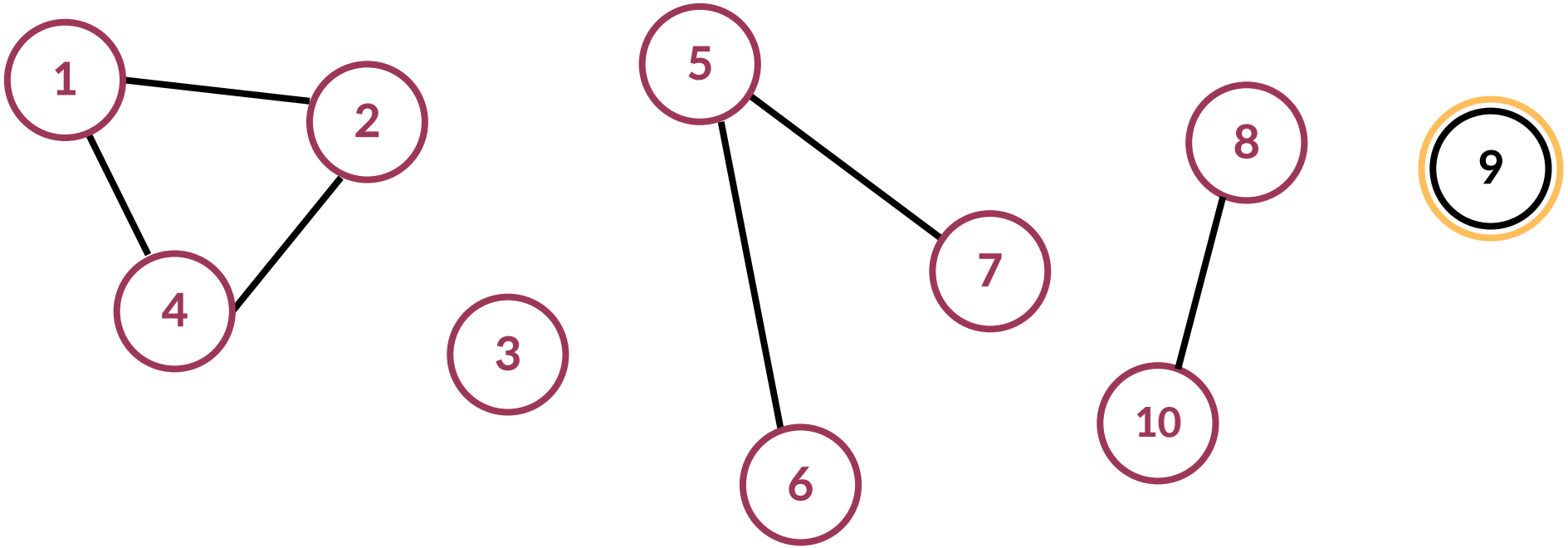
#11724 연결 요소의 개수



- 8, 10번이 방문처리되고 끝난다

연결 요소의 개수

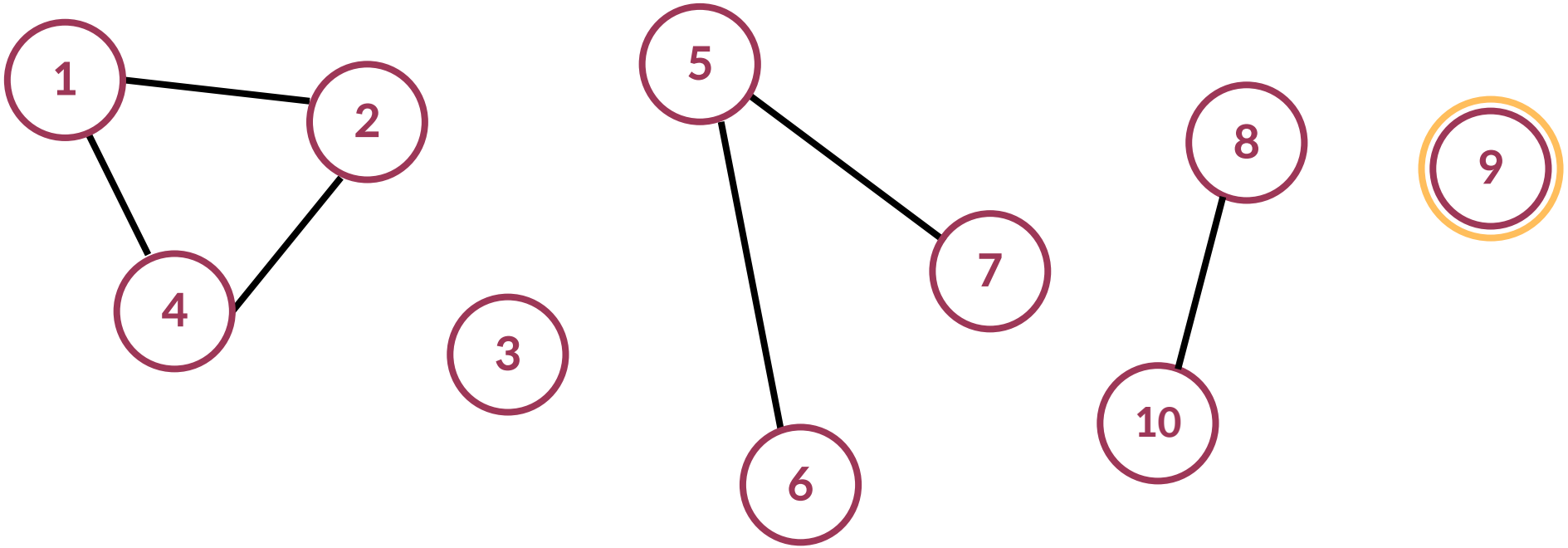
#11724 연결 요소의 개수



- 9번이 아직 미방문 상태이므로 DFS 수행 (DFS 5번째)

연결 요소의 개수

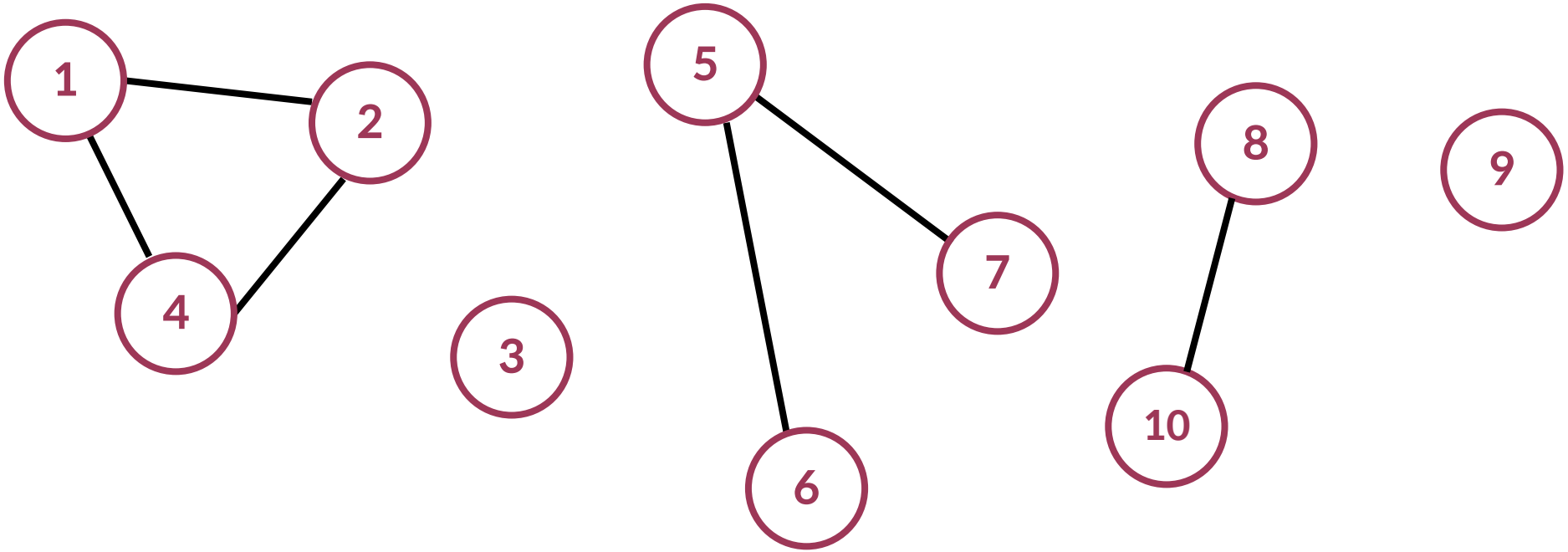
#11724 연결 요소의 개수



- 9번 정점만 방문처리되고 끝
- 10번 정점은 이미 방문 상태이므로 DFS를 수행하지 않음

연결 요소의 개수

#11724 연결 요소의 개수



- 모든 정점에서 DFS를 한 번씩 수행했고, main에서 총 5번 호출됨
- 결국 DFS가 (재귀가 아닌 곳에서) 호출되는 횟수가 연결 요소의 개수

연결 요소의 개수

#11724 연결 요소의 개수

- 간선을 입력받아 인접 리스트를 구성

```
int n, m; cin >> n >> m;  // 정점 개수, 간선 개수
for (int i = 0; i < m; i++) {
    int u, v; cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

연결 요소의 개수

#11724 연결 요소의 개수

- 1번 정점부터 n번 정점까지 돌면서 dfs 수행

```
// 정점 1~n을 모두 체크하면서,
// 아직 방문하지 않았다면 dfs를 수행, 연결 요소 개수 1 증가
int comp_cnt = 0;
for (int i = 1; i <= n; i++) {
    if (!visited[i]) {
        comp_cnt++;
        dfs(i);
    }
}

cout << comp_cnt << '\n';
```

격자 그래프로 하나만 더 보자

#1012 유기농 배추

- 0은 배추가 심어져 있지 않은 땅이고, 1은 배추가 심어져 있는 땅을 나타낸다.
- 한 배추의 상하좌우 네 방향에 다른 배추가 위치한 경우에 서로 인접해 있다고 간주한다.
- 문제를 읽어보자

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

격자 그래프로 하나만 더 보자

#1012 유기농 배추

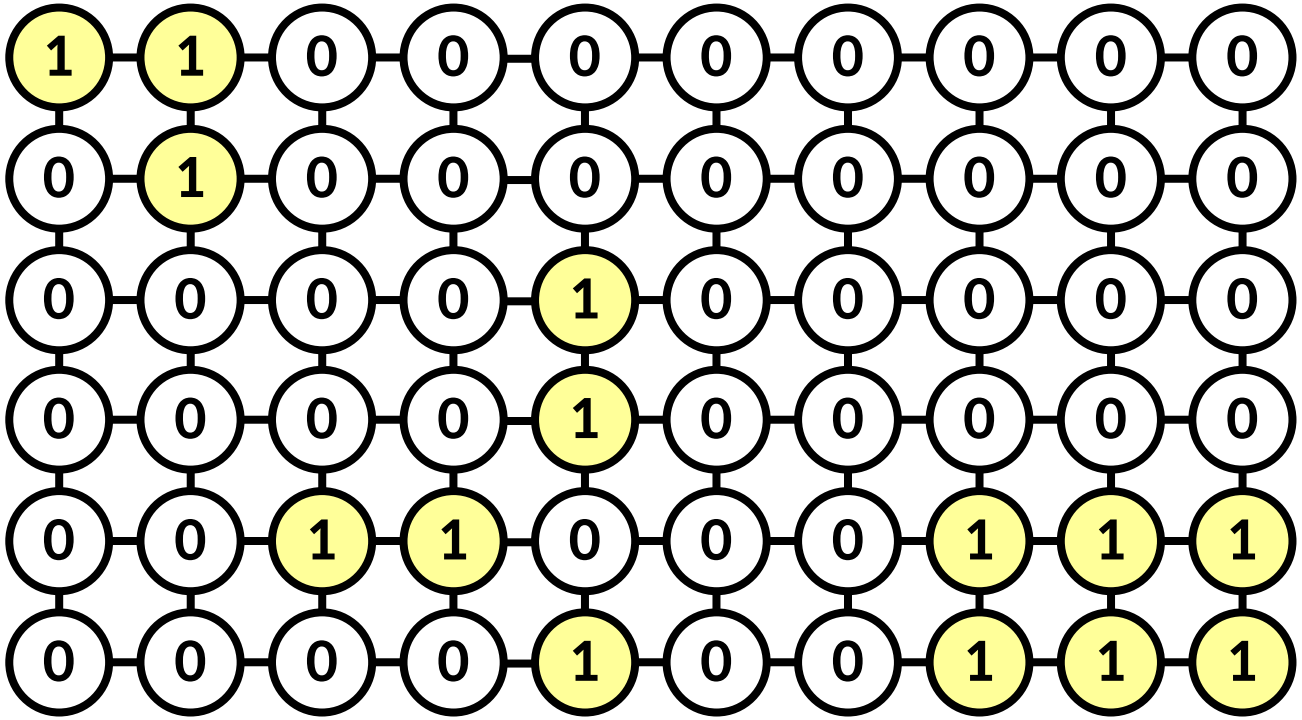
- 문제를 읽어보면, 결국 상하좌우 인접한 1의 덩어리 개수를 세라는 소리이다
- 어떻게 풀 수 있을까?

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

격자 그래프

#1012 유기농 배추

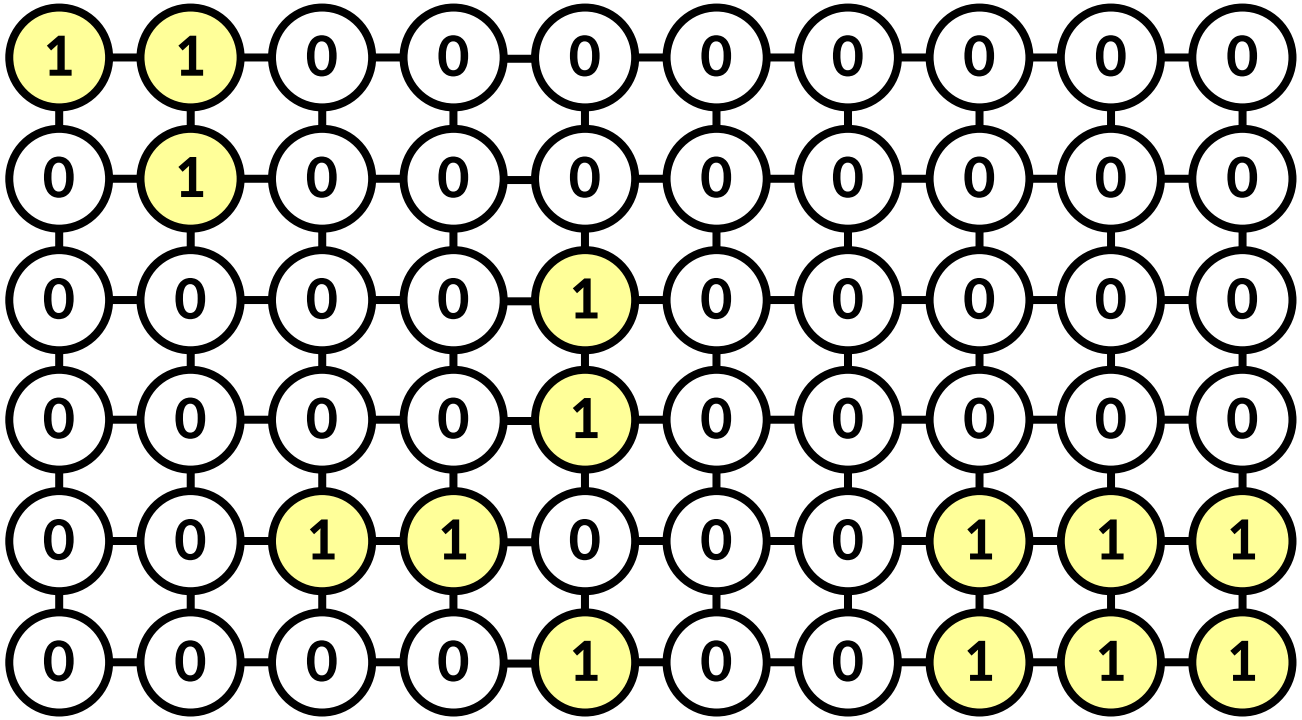
- 각 칸 하나를 하나의 정점으로 보면, 상하좌우 정점이 모두 연결된 그래프로 볼 수 있다.
  - 이렇게 생긴 그래프를 **격자 그래프** (Lattice Graph)라 한다
- 이렇게 그래프를 구성하고, 연결 요소의 개수를 세면 된다.



격자 그래프

#1012 유기농 배추

- 이런 격자 그래프처럼 간선 관계가 너무 뻔하게 정해져 있는 경우,
  굳이 간선을 모두 연결할 필요 없이 **이동할 수 있는 방향을 정의**해줘도 된다



격자 그래프의 그래프 구성

#1012 유기농 배추

- 대충 이런 식이다

```
int dy[4] = {-1, 0, 1, 0};
int dx[4] = {0, 1, 0, -1};

void dfs(int y, int x) {
    visited[y][x] = true;
    for (int i = 0; i < 4; i++) {
        ny = y + dy[i];
        nx = x + dx[i];
        if (ma[ny][nx] == 1 && !visited[ny][nx] &&
            (ny >= 0 && ny < n) && (nx >= 0 && nx < m)) {
            dfs(ny, nx);
        }
    }
}
```

격자 그래프의 그래프 구성

#1012 유기농 배추

- 방문 체크 배열과 범위 체크 부분에 주의

```
int dy[4] = {-1, 0, 1, 0};
int dx[4] = {0, 1, 0, -1};

void dfs(int y, int x) {
    visited[y][x] = true;
    for (int i = 0; i < 4; i++) {
        ny = y + dy[i];
        nx = x + dx[i];
        if (ma[ny][nx] == 1 && !visited[ny][nx] &&
            (ny >= 0 && ny < n) && (nx >= 0 && nx < m)) {
            dfs(ny, nx);
        }
    }
}
```

Practice

#1260 DFS와 BFS

#10451 순열 사이클

#2606 바이러스

#2667 단지번호붙이기

#4963 섬의 개수

#2583 영역 구하기

#2468 안전 영역

#1389 케빈 베이컨의 6단계 법칙

#7576 토마토

#2178 미로 탐색

#2206 벽 부수고 이동하기

Sources

- https://commons.wikimedia.org/wiki/File:US_Trade_Balance_from_1960.svg