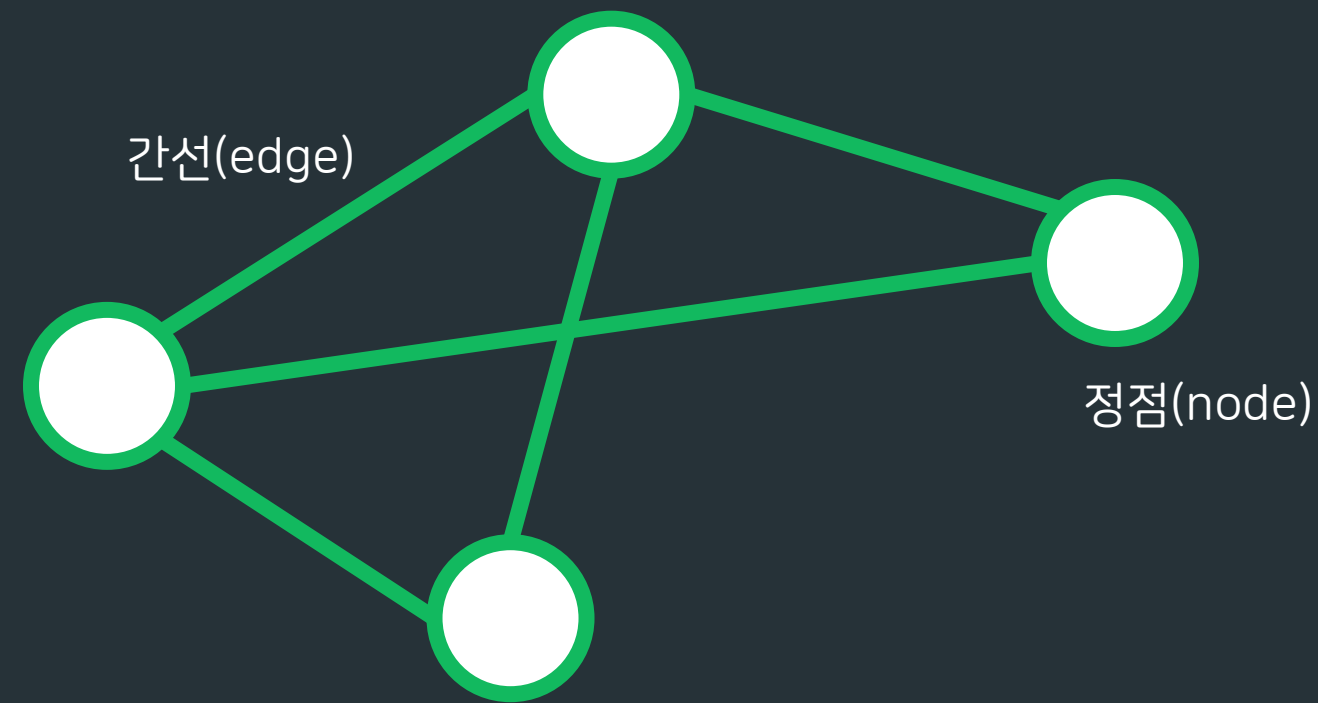


# 알튜비튜

## DFS & BFS

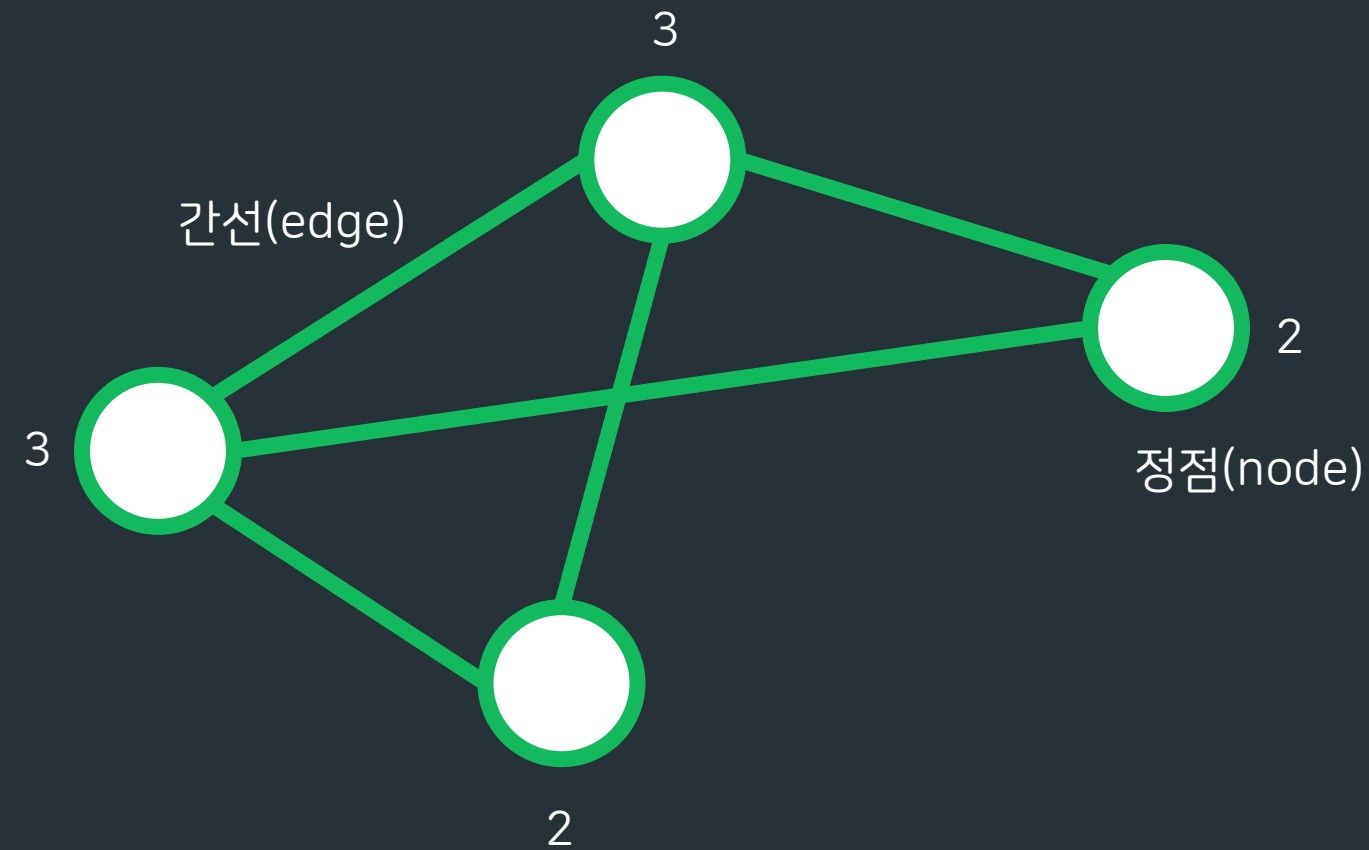
오늘은 그래프 탐색 알고리즘인 깊이 우선 탐색(DFS)과 너비 우선 탐색(BFS)을 배웁니다.  
앞으로 배울 그래프 알고리즘의 시작이자 코딩테스트에 높은 확률로 한 문제 이상 나오는 알고리즘이죠.

# 이거 먼저 봅시다!



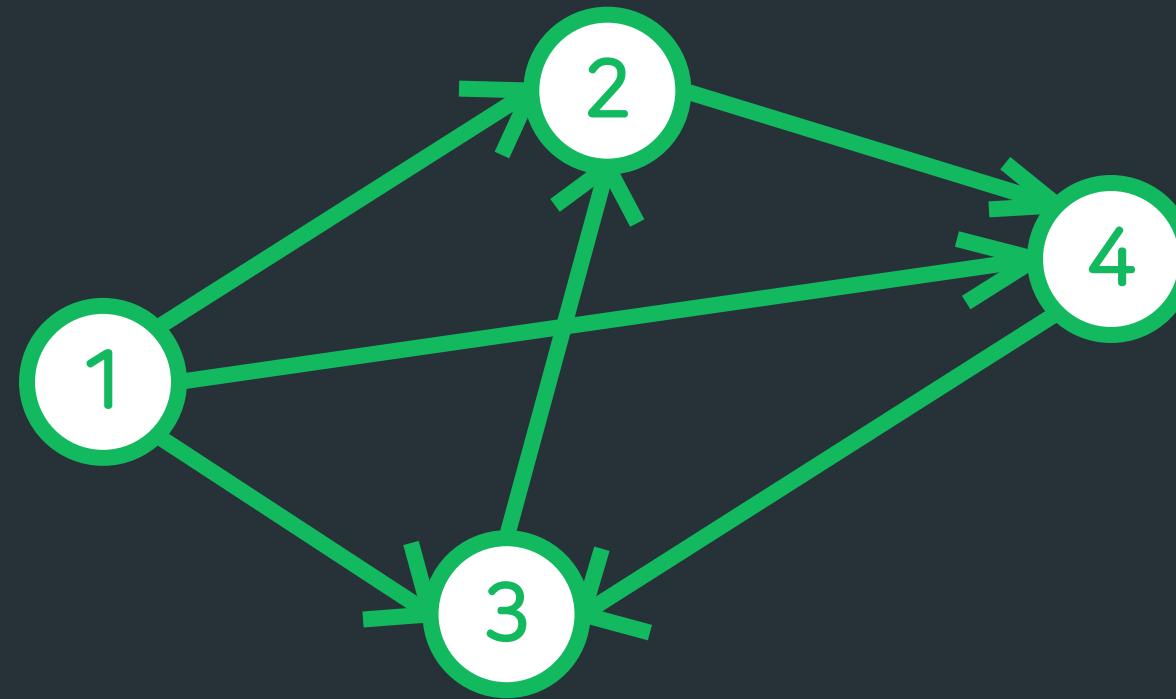
## 그래프

- 정점(node)과 그 정점을 연결하는 간선(edge)으로 이루어진 자료구조
- 간선의 방향은 단방향(방향 그래프)이나 양방향(무방향 그래프)으로 나뉨
- 간선에 가중치가 있을 수 있음



## 그래프

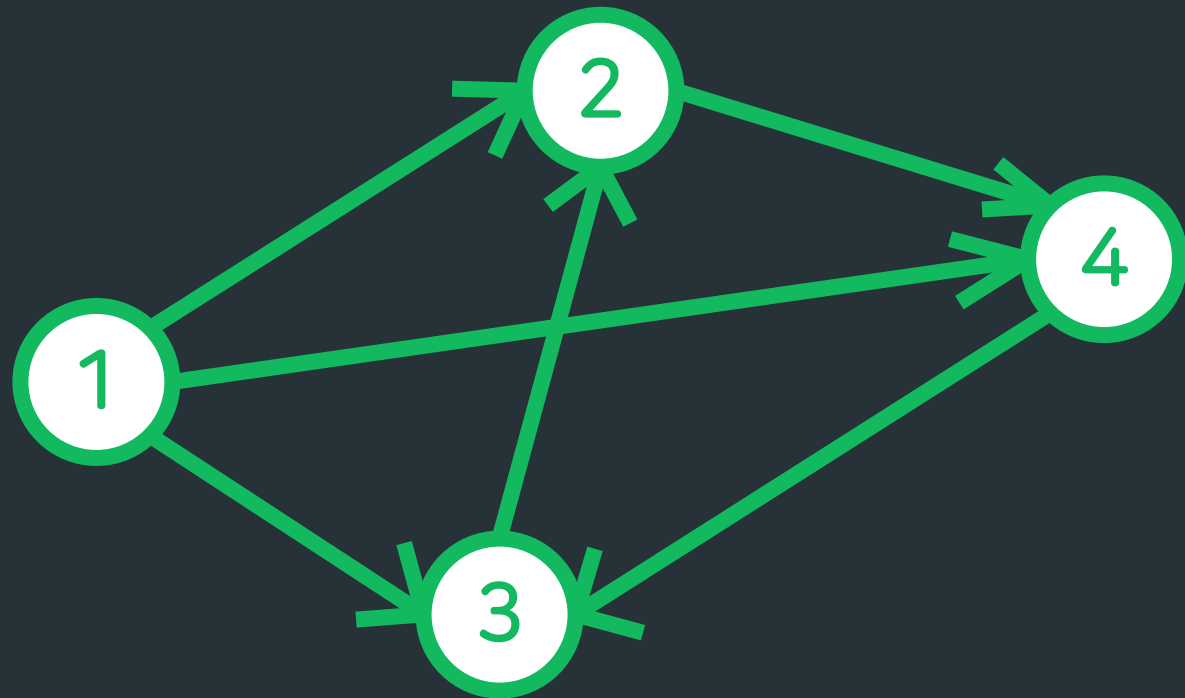
- degree(차수): 무방향 그래프에서 정점에 연결된 간선의 수
  - 무방향 그래프의 총 차수(degree)의 합 = 간선의 수  $\times 2$
- indegree: 방향 그래프에서 해당 정점으로 들어오는 간선의 수
- outdegree: 방향 그래프에서 해당 정점에서 나가는 간선의 수
  - 방향 그래프의 총 차수(indegree, outdegree)의 합 = 간선의 수



- 정점 1 → 4 경로
  - 1 → 4
  - 1 → 2 → 4
  - 1 → 3 → 2 → 4

## 경로(path)

- 경로는 여러 개일 수 있다
- 사이클(cycle): 한 정점에서 다시 동일한 정점으로 돌아오는 경로 (ex. 3 → 2 → 4)

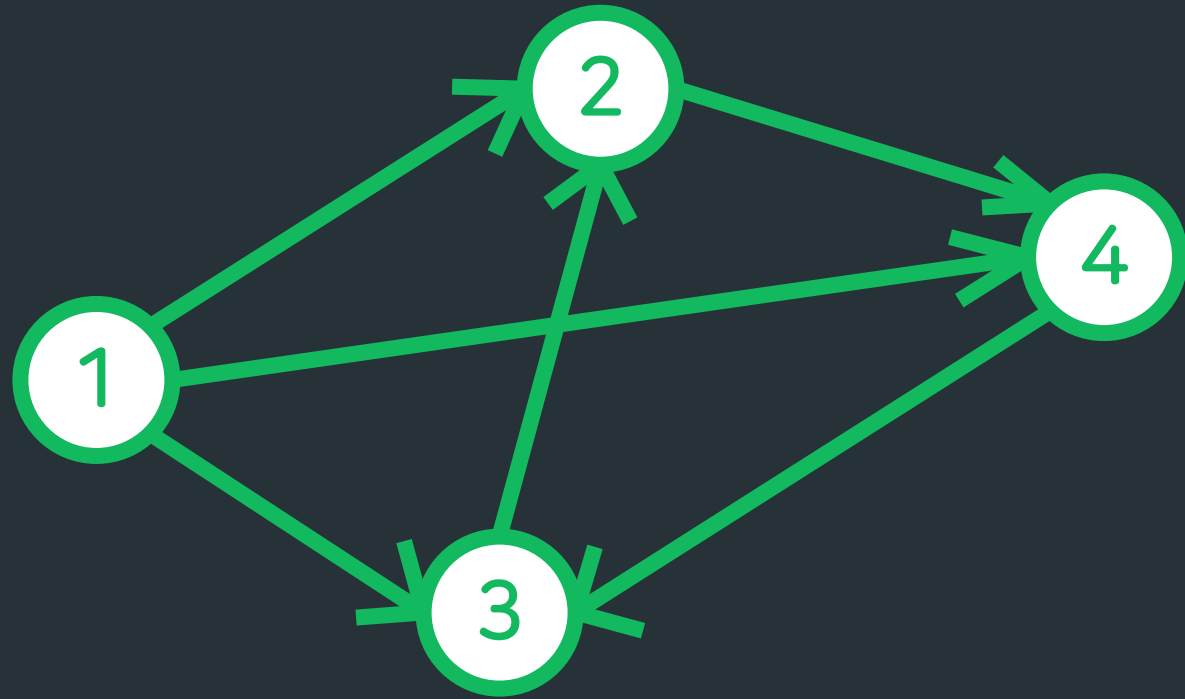


	1	2	3	4
1	0	1	1	1
2	0	0	0	1
3	0	1	0	0
4	0	0	1	0

\* 만약 가중치가 주어졌다면 가중치 값 저장

## 인접 행렬 (Adjacency Matrix)

- 두 정점 간의 연결 관계를  $N * N$  크기의 행렬(2차원 배열)로 나타냄
  - 두 정점  $i - j$  간의 연결 관계 확인:  $O(1)$
  - 특정 노드와 연결되어 있는 모든 노드 확인:  $O(N)$
  - 공간 복잡도:  $O(N^2)$
- 정점이 많아지면 사용 불가 (메모리 초과)



1	2	3	4
2	4		
3	2		
4	3		

## 인접 리스트 (Adjacency List)

- 각 정점에 연결된 정점들을 리스트(1차원 배열)에 담아 보관
- 두 정점  $i-j$  간의 연결 관계 확인:  $O(\min(\text{degree}(i), \text{degree}(j)))$
- 특정 노드( $i$ )와 연결되어 있는 모든 노드 확인:  $O(\text{degree}(i))$
- 공간 복잡도:  $O(N+E)$

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

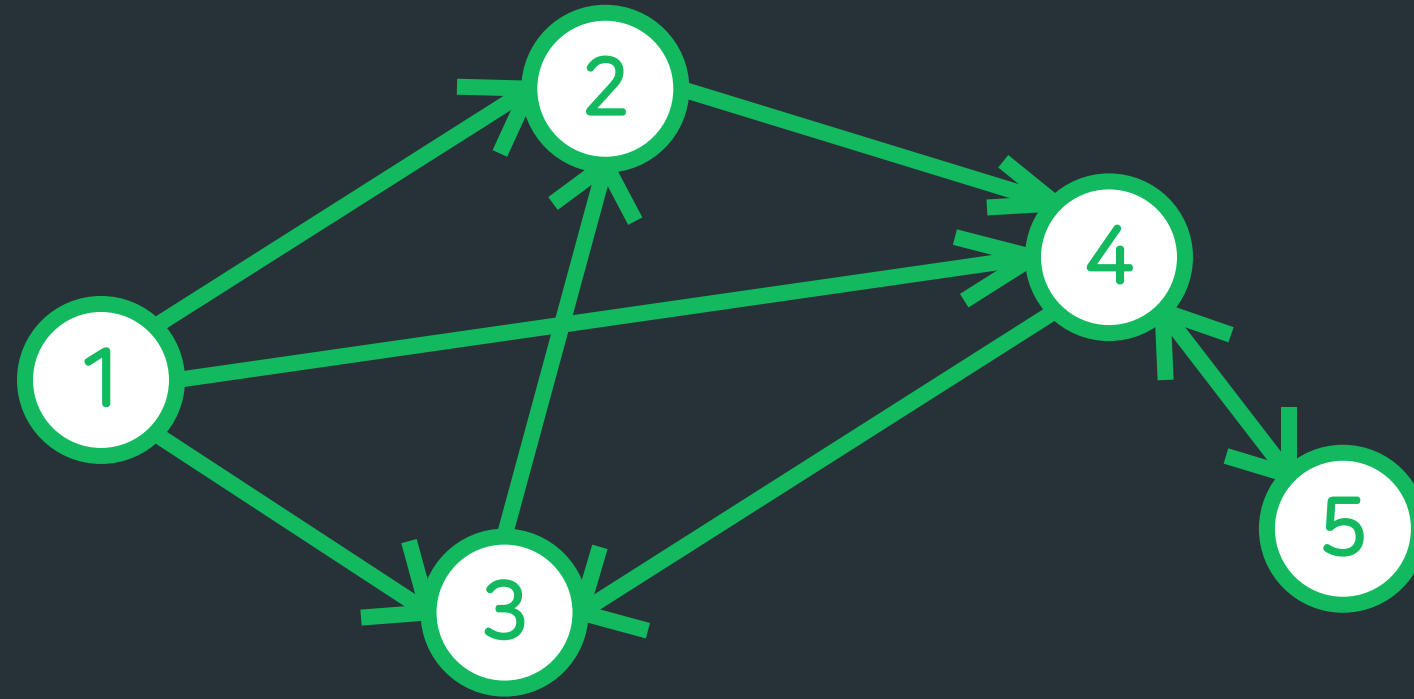
## 인접 행렬

- 정점 사이의 연결 관계(간선)가 많은 경우
- 특정 정점과 정점의 연결 관계를 많이 확인해야 하는 경우

1	2	3	4
2	4		
3	2		
4	3		

## 인접 리스트

- 정점 사이의 연결관계(간선)가 적은 경우
- 연결된 정점들을 탐색해야 하는 경우



## 그래프 탐색

- 그래프의 모든 노드를 탐색하기 위해 간선을 따라 순회하는 것
- 탐색 방법에 따라 BFS(너비 우선 탐색), DFS(깊이 우선 탐색)로 나뉨
- 탐색 시, 방문 체크 꼭 필요
- 그래프는 가장 폭 넓고 깊은 주제 중 하나인데, 탐색이 그 중 기본이라 할 수 있음
- 시간 복잡도: 인접 행렬로 구현 시  $O(N^2)$ , 인접 리스트로 구현 시  $O(N+E)$

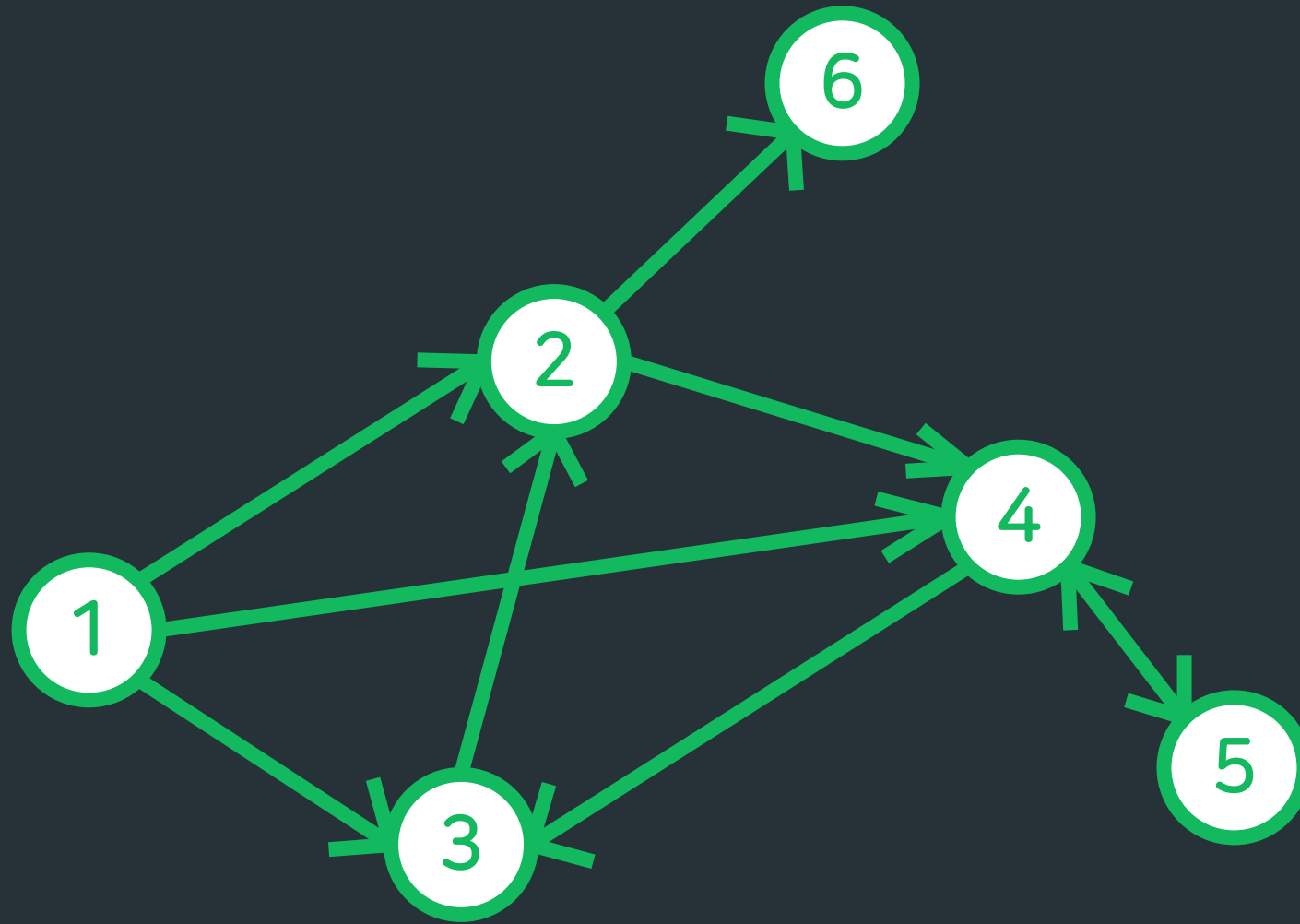


## DFS (깊이 우선 탐색)

- 최대한 깊게 탐색 후 빠져 나옴
- 한 정점을 깊게 탐색해서 빠져 나왔다면, 나머지 정점 계속 동일하게 탐색
- 스택(stack), 재귀함수로 구현

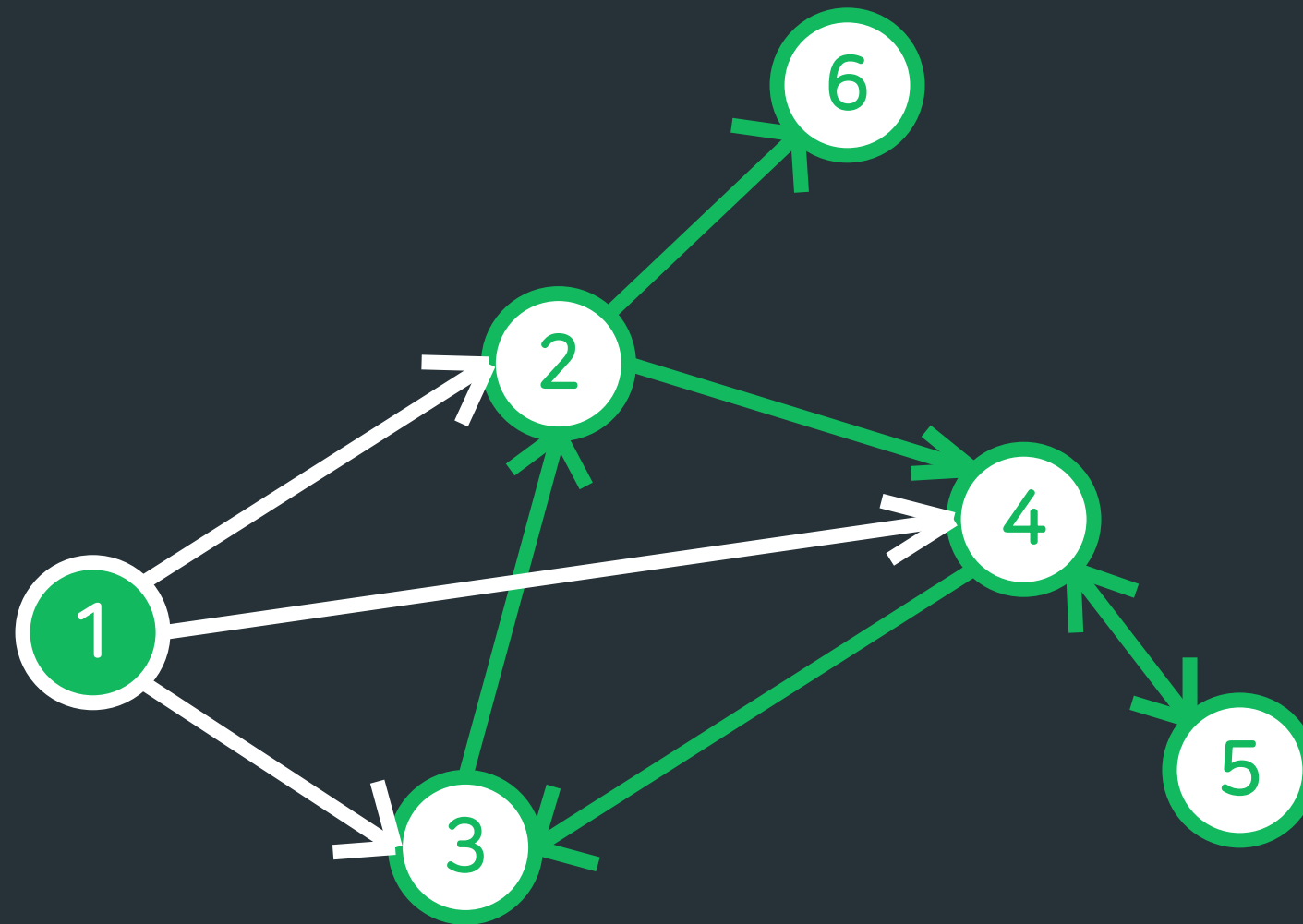
## BFS (너비 우선 탐색)

- 자신의 자식들부터 순차적으로 탐색
- 순차 탐색 이후, 다른 정점의 자식들 탐색
- 큐(queue)로 구현



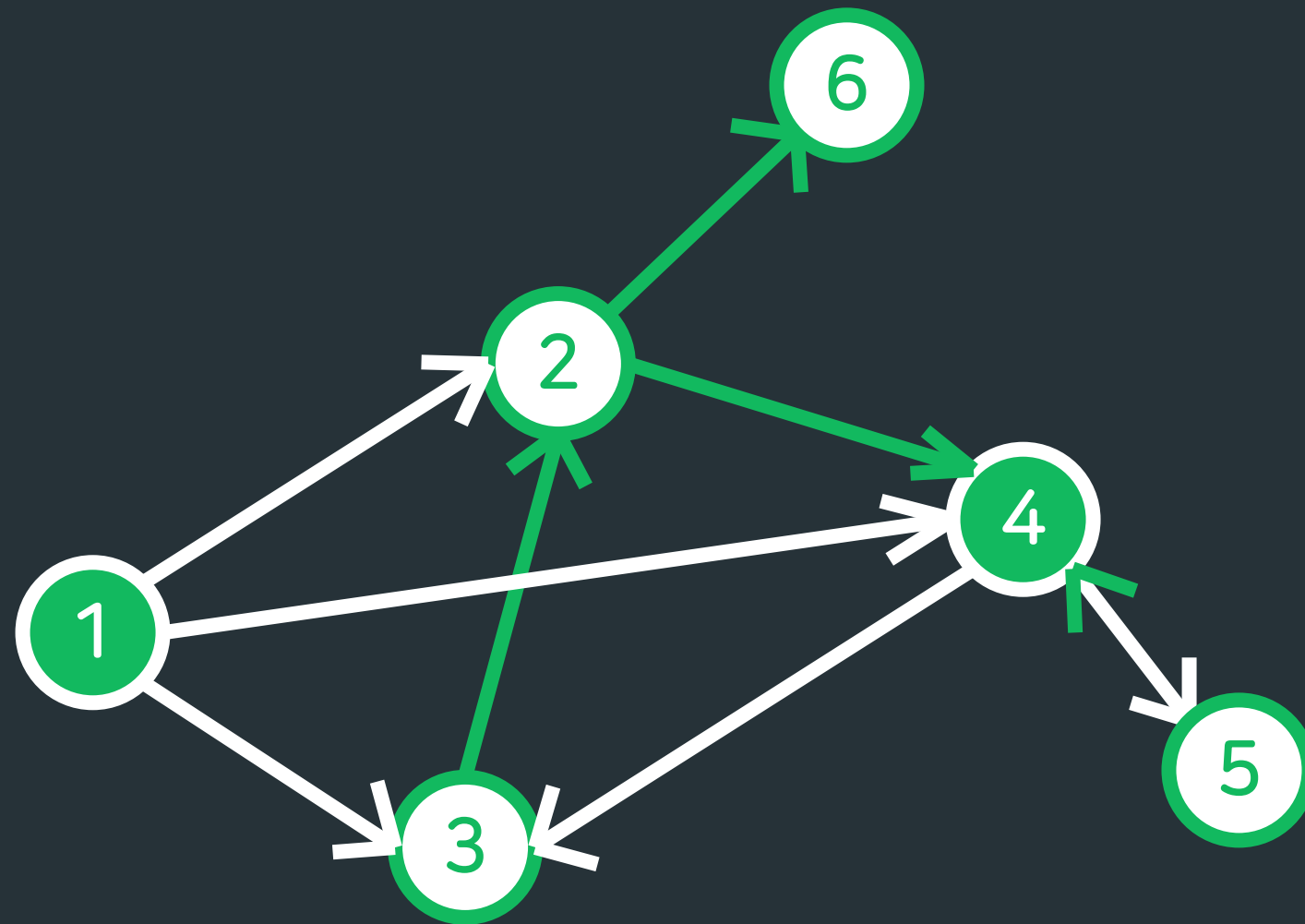
\* 탐색은 어디에서든 시작 가능. 시작 노드는 주로 문제에서 주어줌





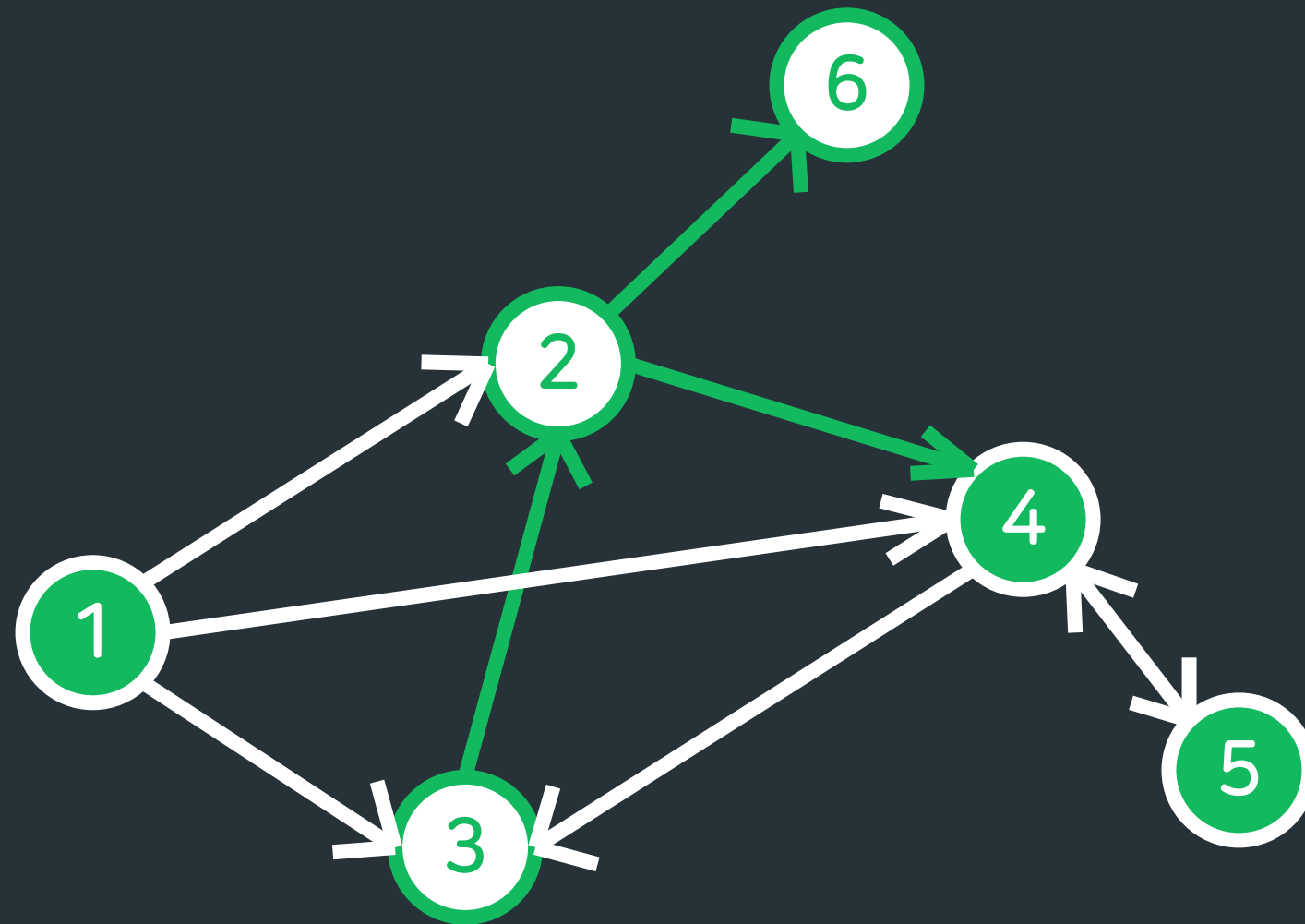
● 탐색 순서: 1





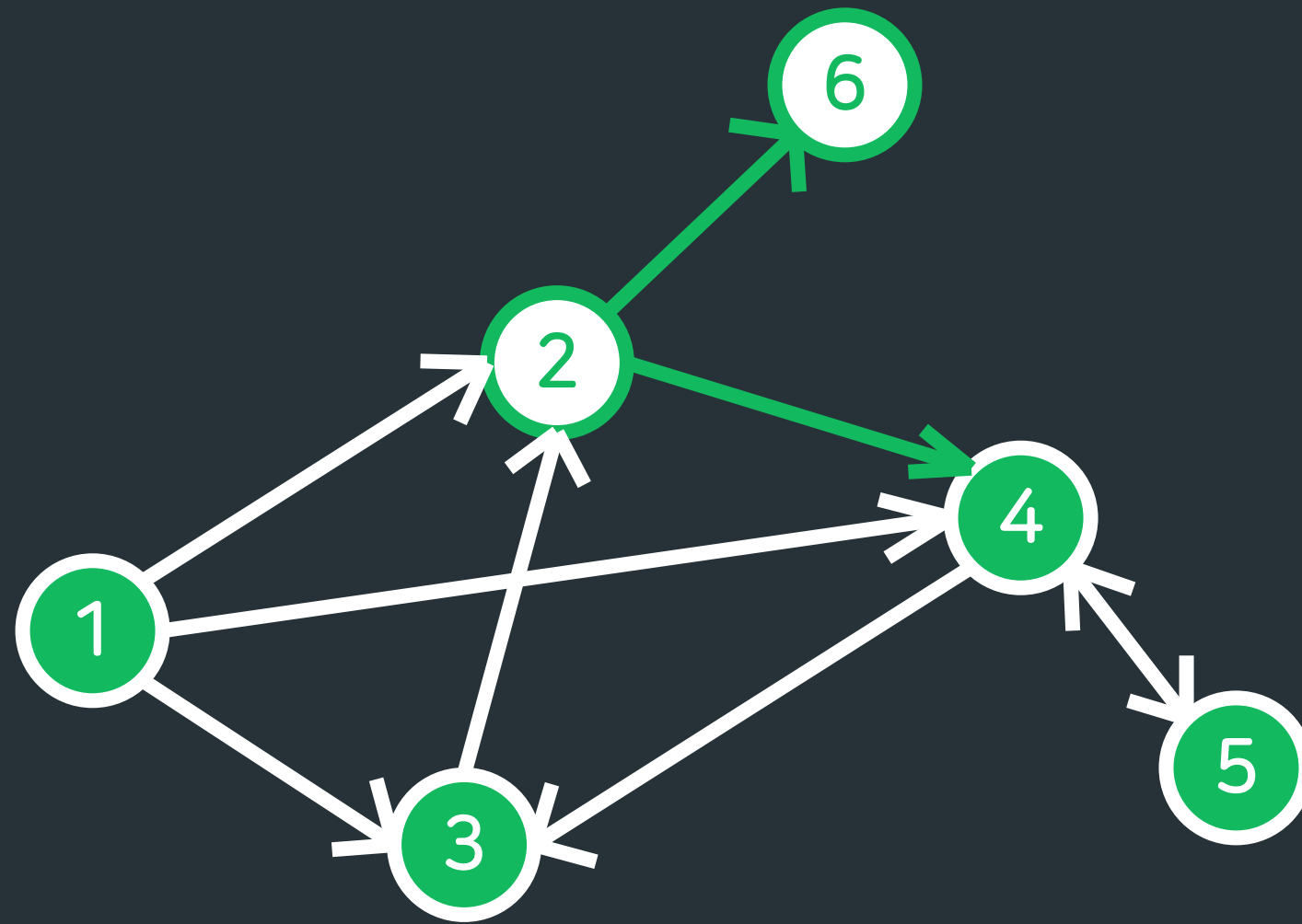
- 탐색 순서: 1 → 4





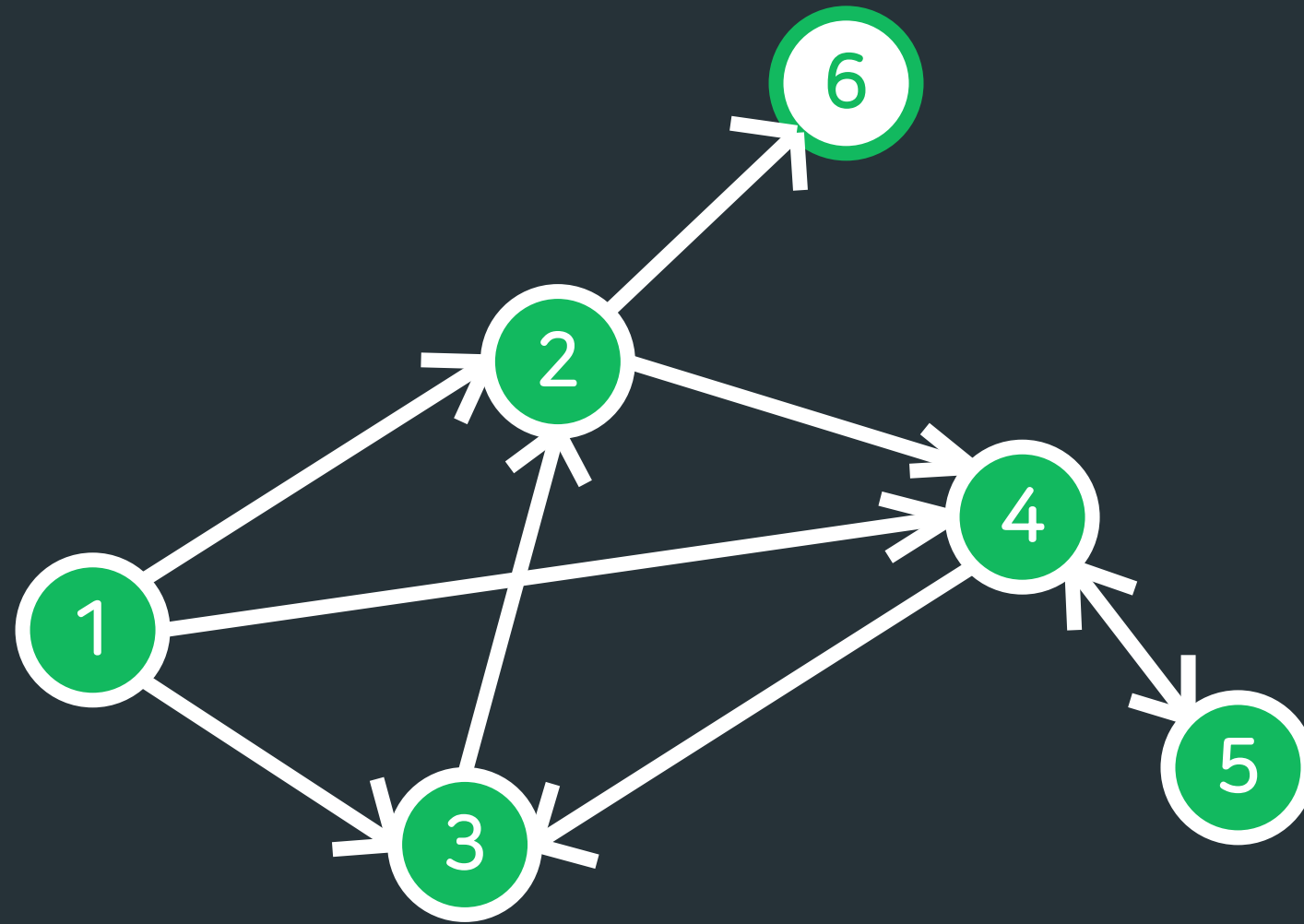
- 탐색 순서: 1 → 4 → 5





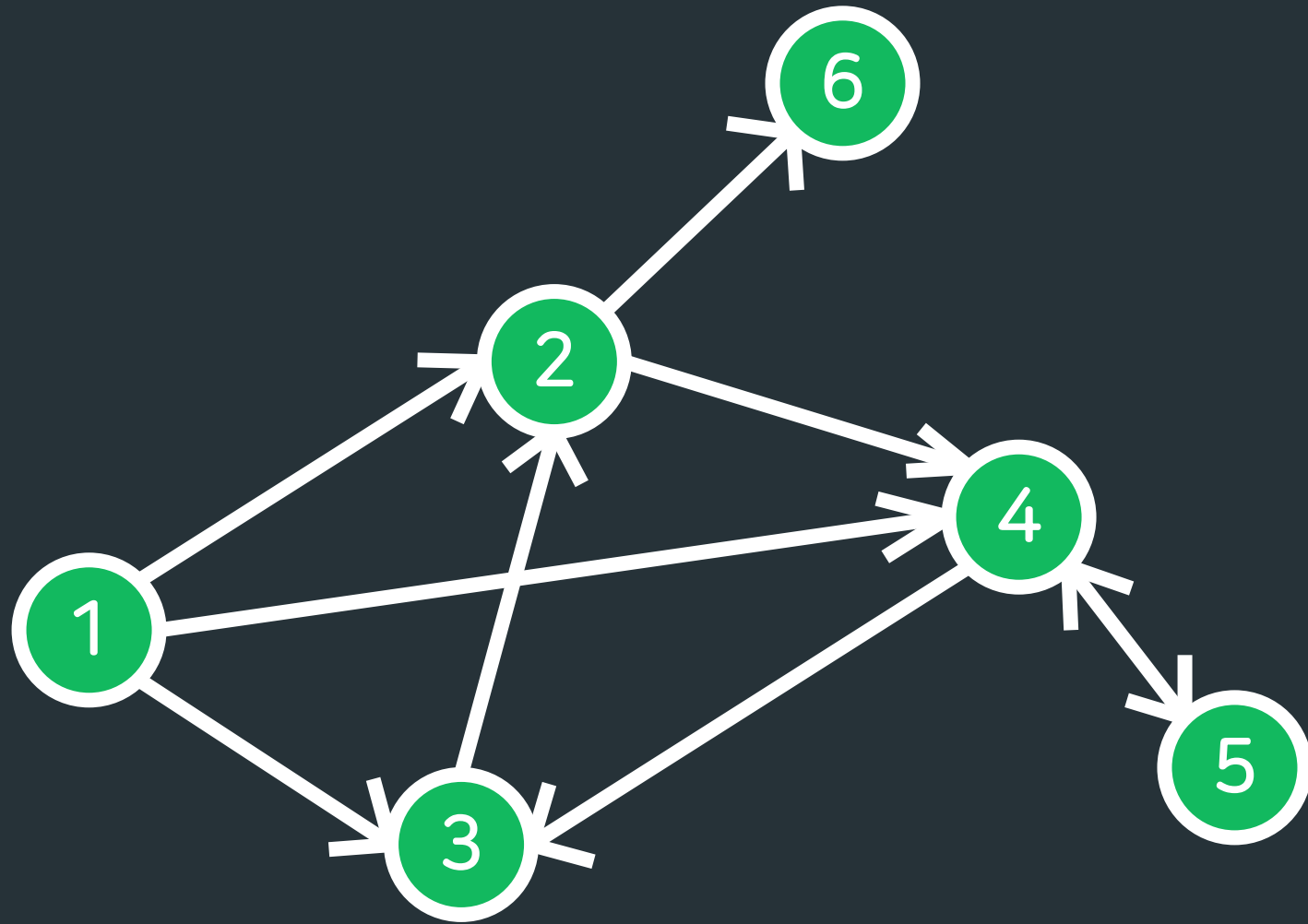
- 탐색 순서: 1 → 4 → 5 → 3





- 탐색 순서: 1 → 4 → 5 → 3 → 2

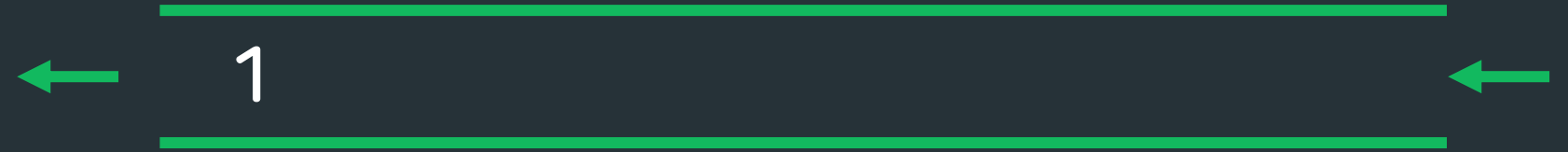
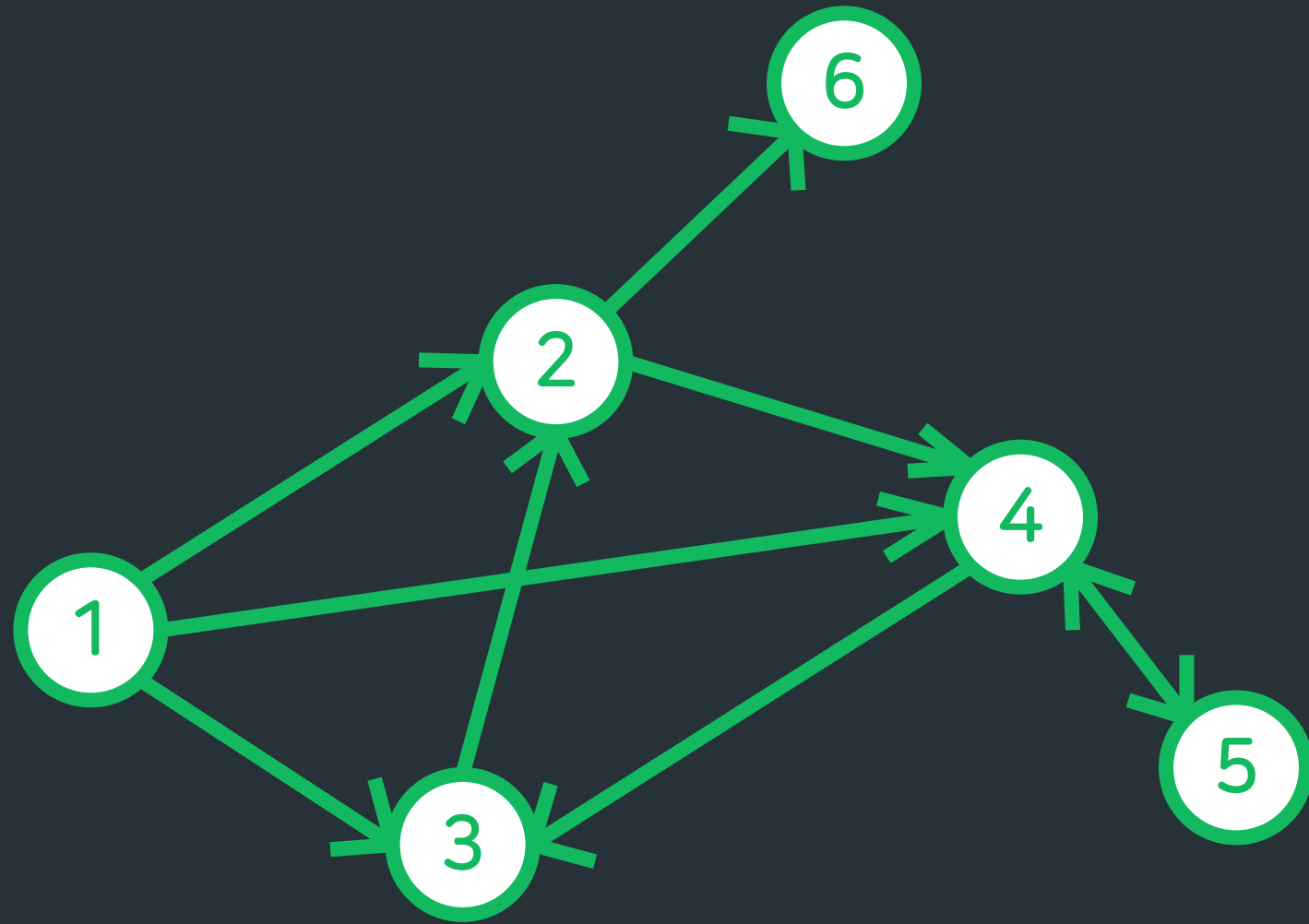




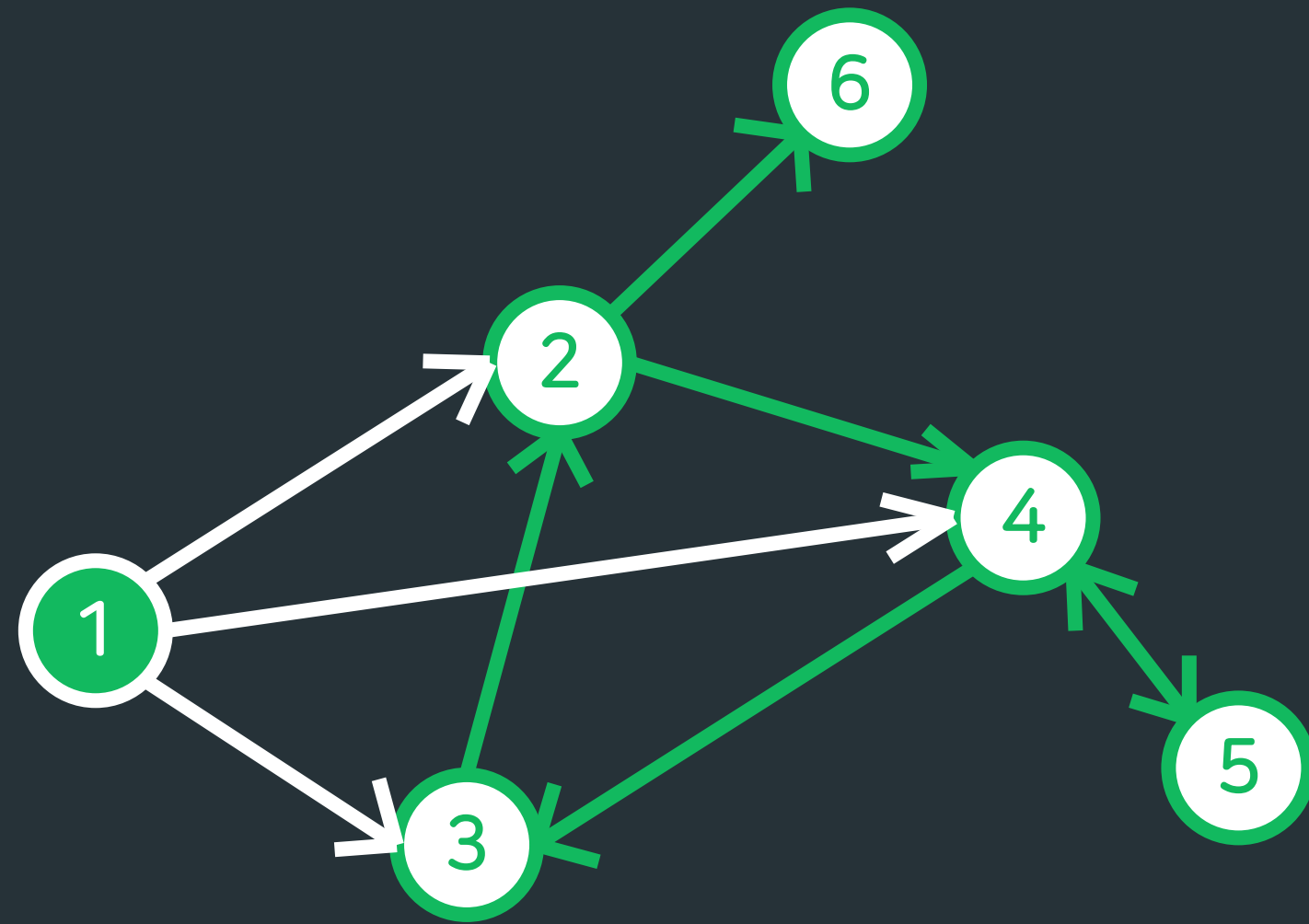
- 탐색 순서: 1 → 4 → 5 → 3 → 2 → 6



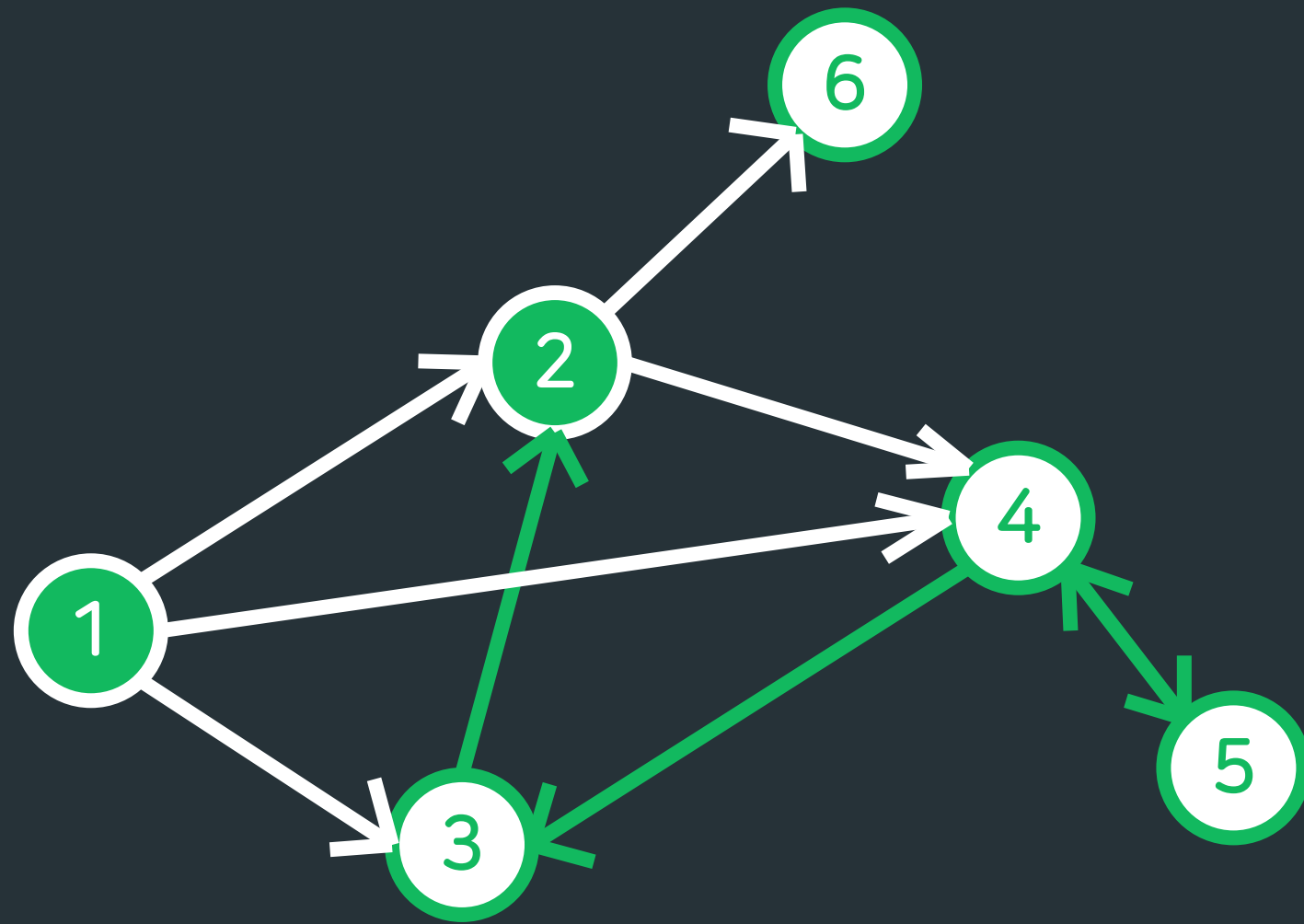




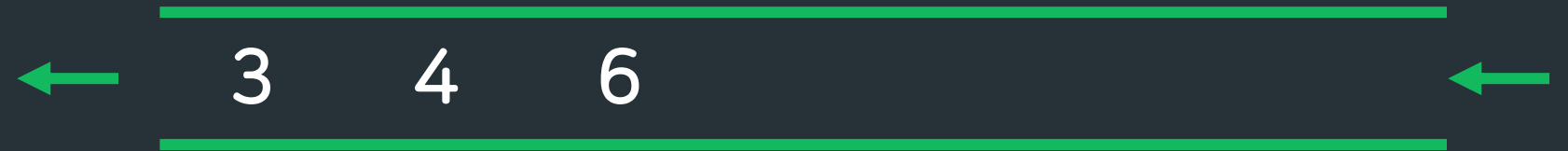
\* 탐색은 어디에서든 시작 가능. 시작 노드는 주로 문제에서 주어줌

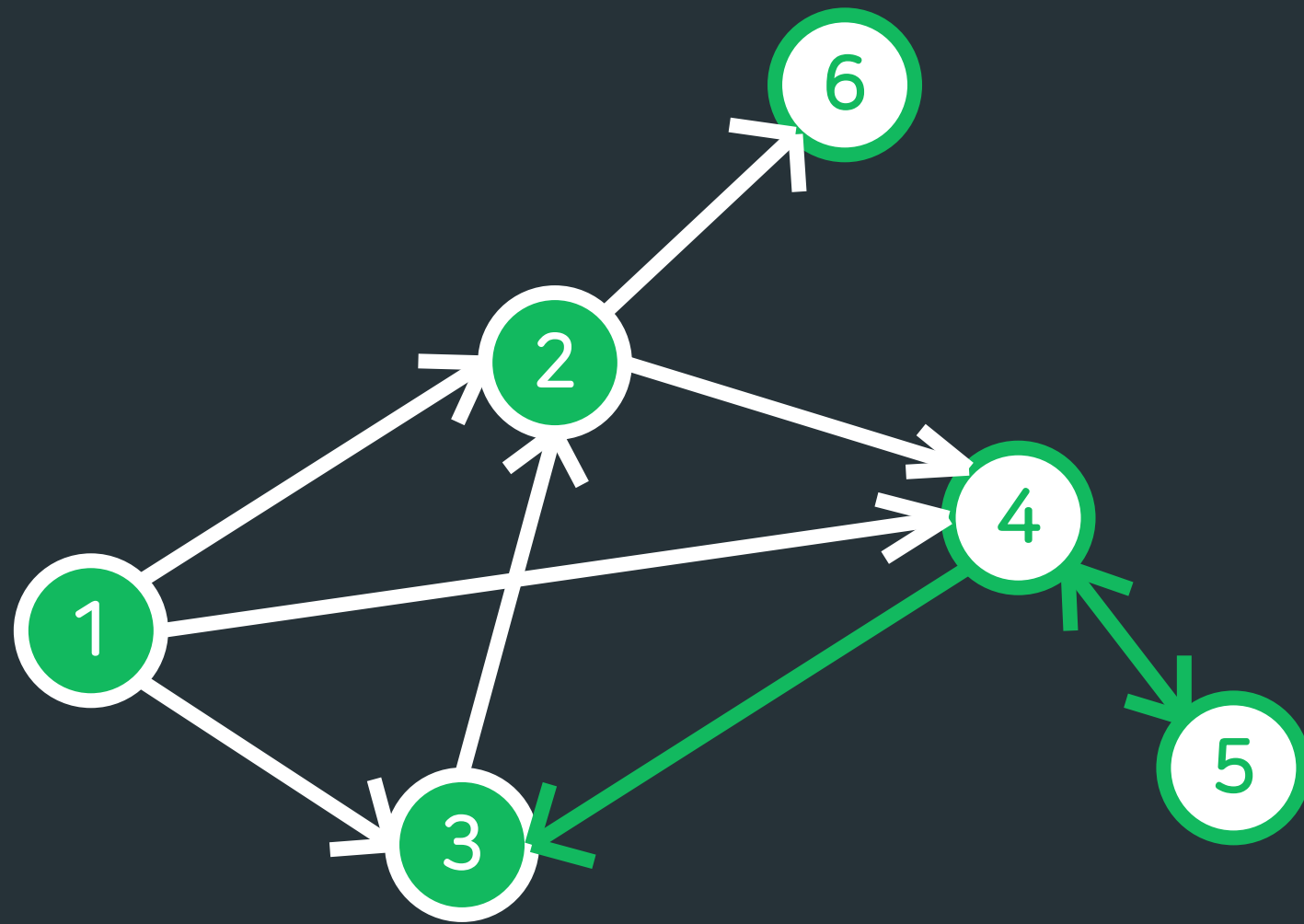


- 탐색 순서: 1

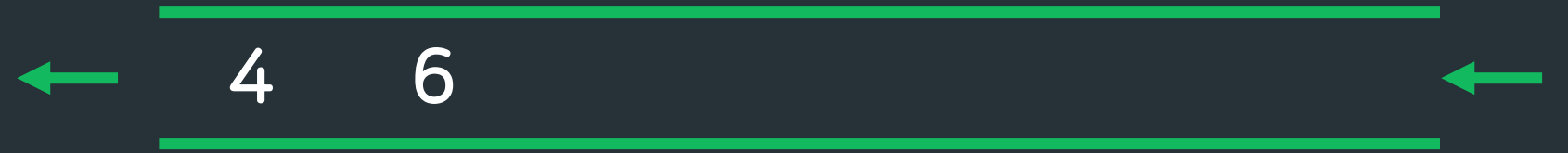


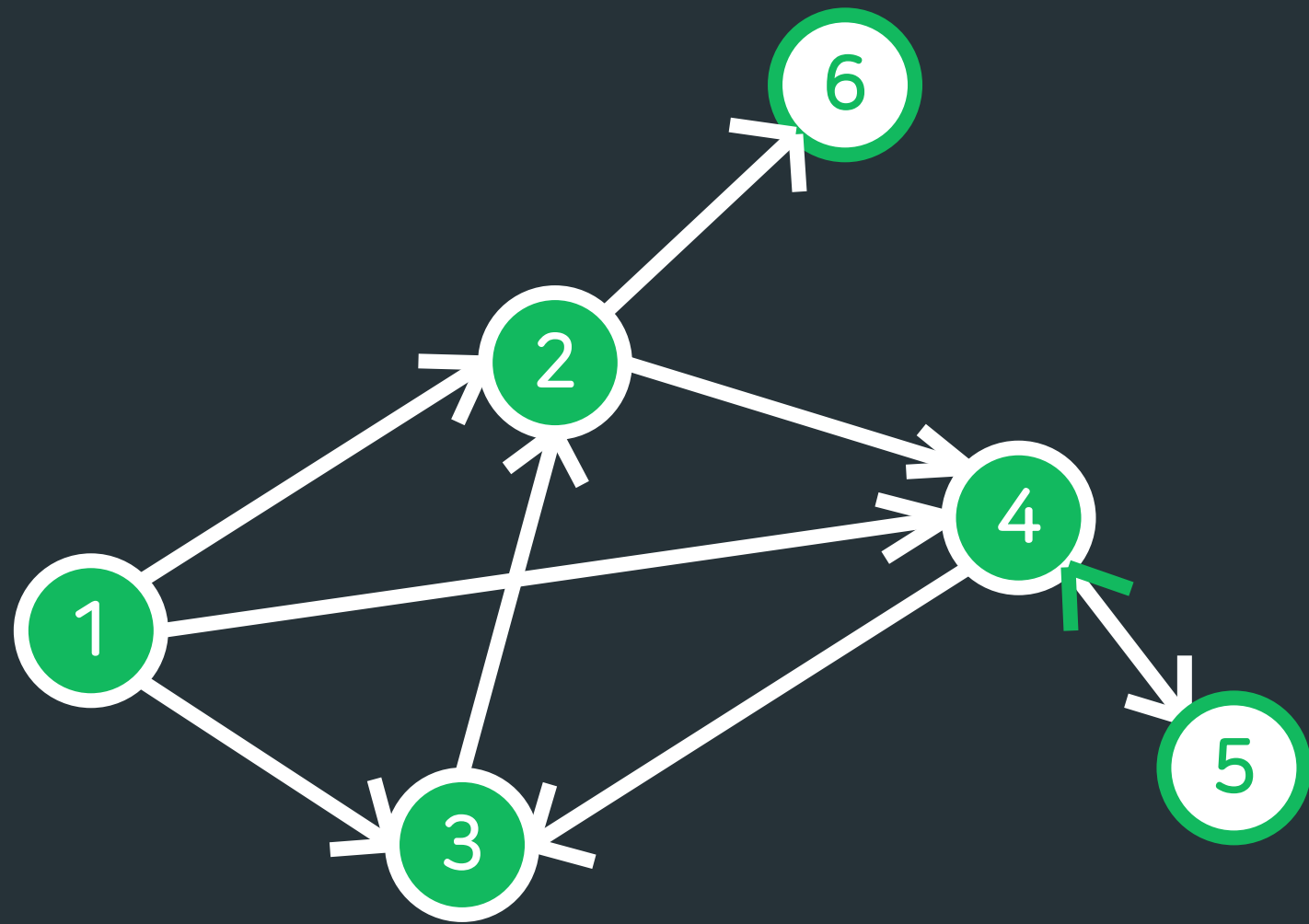
- 탐색 순서: 1 → 2



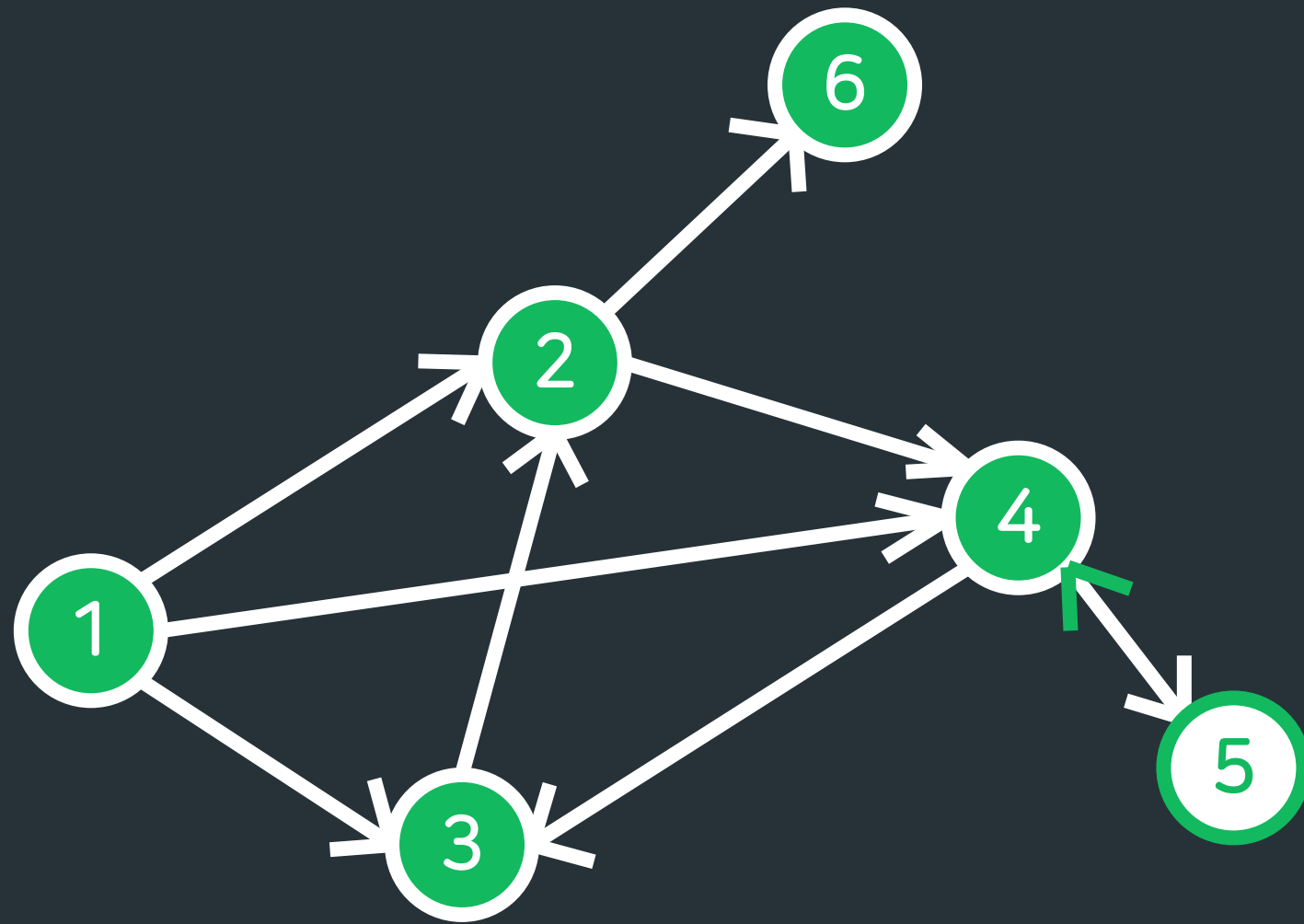


- 탐색 순서: 1 → 2 → 3

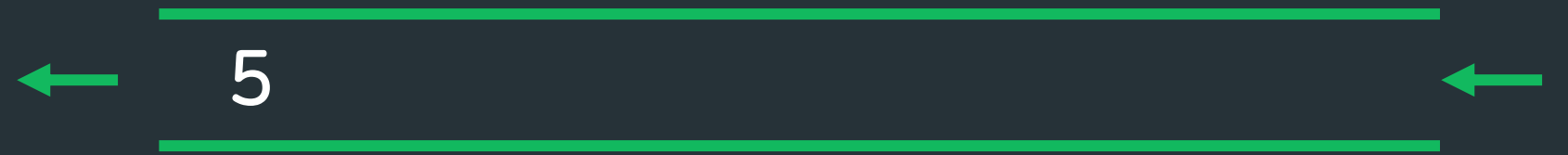


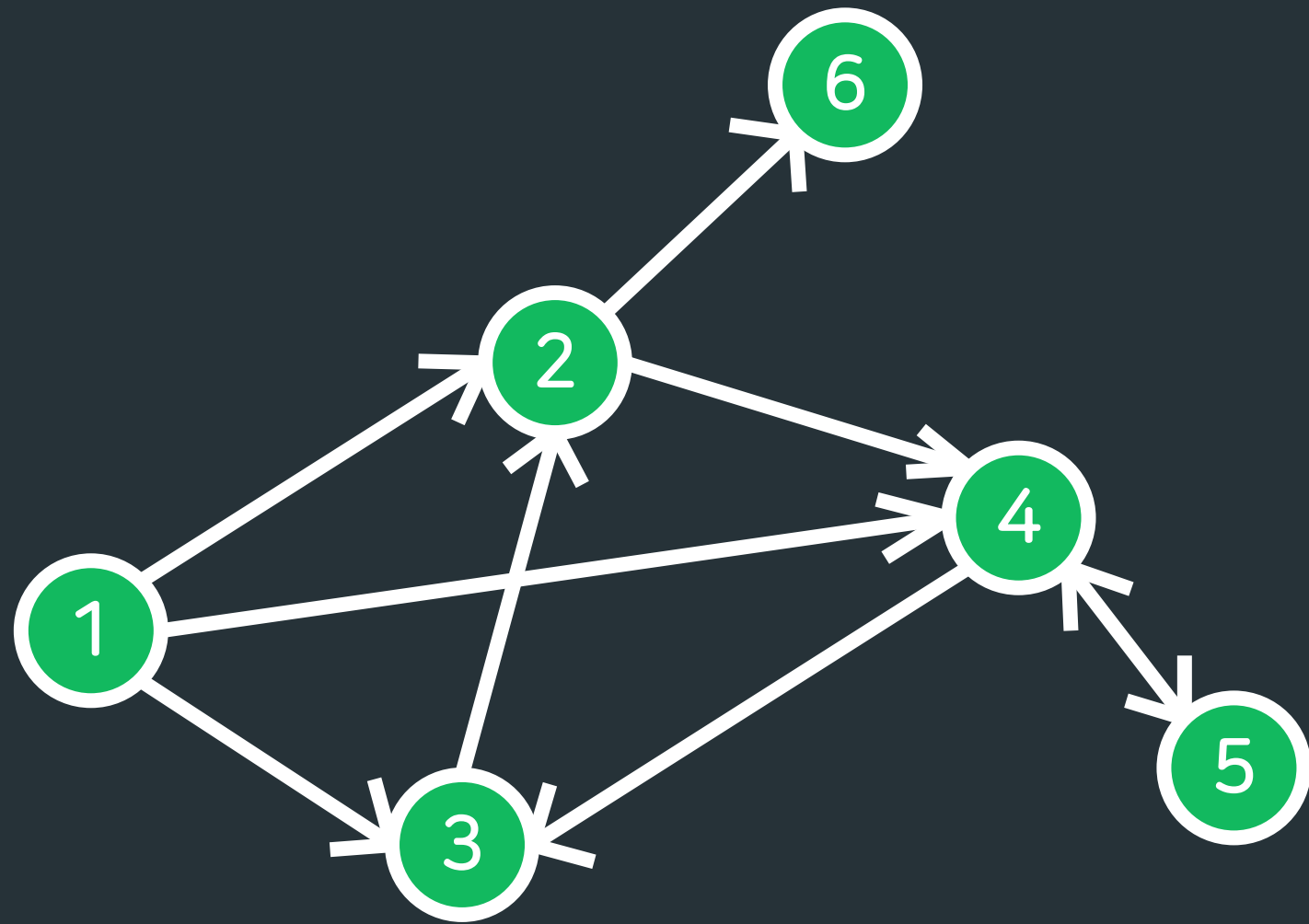


- 탐색 순서: 1 → 2 → 3 → 4

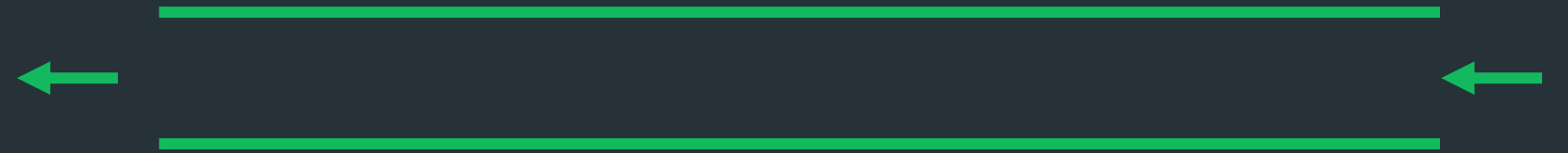


- 탐색 순서: 1 → 2 → 3 → 4 → 6





- 탐색 순서: 1 → 2 → 3 → 4 → 6 → 5



## 특징

- 모든 노드를 방문하고자 하는 경우엔 DFS, BFS 상관없음
- 가중치가 주어지거나 특정 경로를 찾아야 할 때 DFS
- 두 노드 사이의 최단거리를 찾을 때는 BFS (ex. 미로찾기)  
→ BFS는 현재 노드에서 가장 가까운 곳(자식 노드)부터 탐색하기 때문
- 그래프가 단방향으로 주어지는지 양방향으로 주어지는지 잘 살펴보자!



## /<> 1260번 : DFS와 BFS – Silver 2

### 문제

- 그래프를 DFS로 탐색한 결과와 BFS로 탐색한 결과를 출력하는 문제
- 단, 방문할 수 있는 정점이 여러 개인 경우, 정점 번호가 작은 것부터 방문

### 제한 사항

- 정점 개수 N의 범위  $1 \leq N \leq 1,000$
- 간선 개수 M의 범위  $1 \leq M \leq 10,000$

## 예제 입력

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

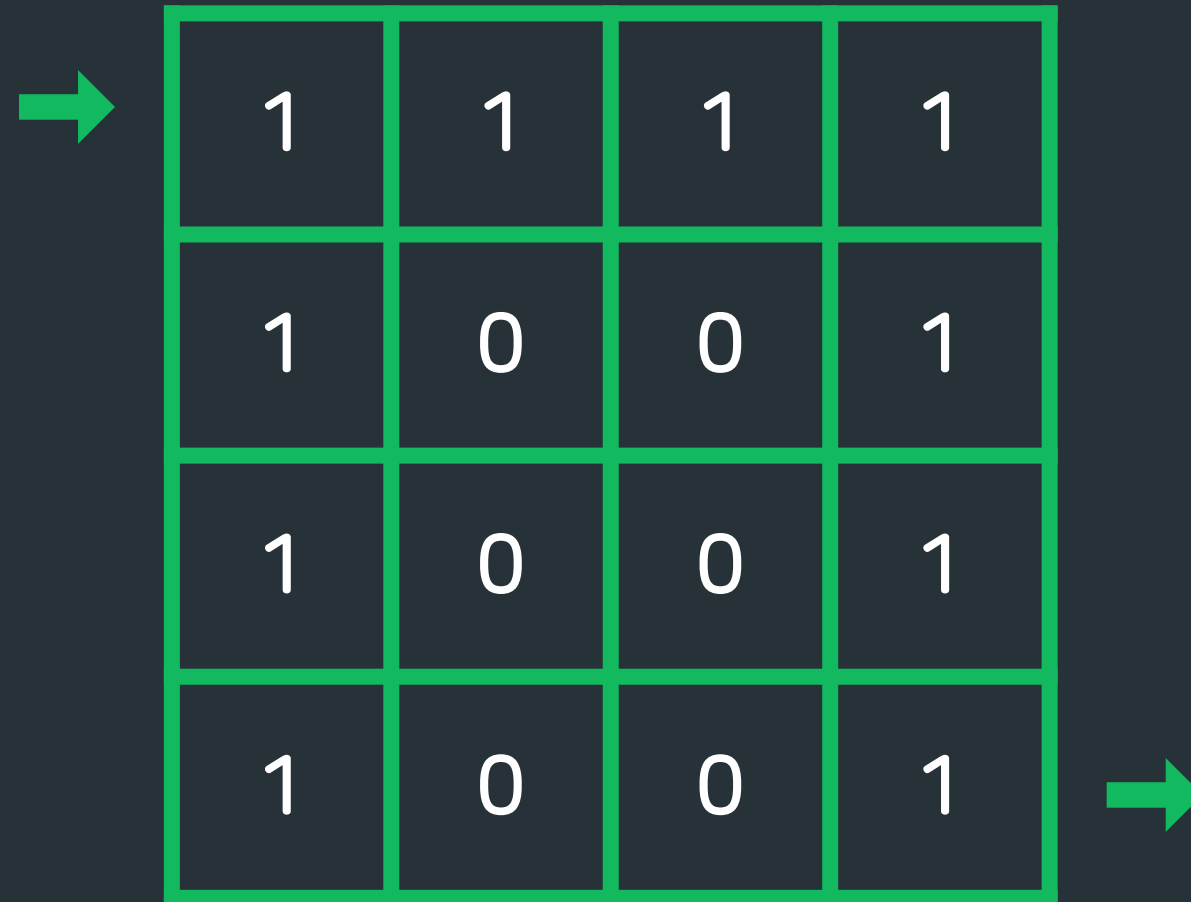
## 예제 출력

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

← DFS  
← BFS

## 다음과 같은 문제가 있어요

- $N \times M$ 으로 표현되는 **미로**가 있을 때,  
(1,1)에서 시작해서 (N,M)까지 이동하는 **최소 칸 수**를 구하여라
- 1: 지나갈 수 있는 길, 0: 막힌 길



# BFS로 풀어보자



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

# BFS로 풀어보자



1	2	1	1
2	0	0	1
1	0	0	1
1	0	0	1

# BFS로 풀어보자



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

# BFS로 풀어보자

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

→ 최단거리 7

- 방문 순서는  
상→하→좌→우



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

- **막다른 길**로 탐색할 가능성 있음



- 방문 순서는  
상→하→좌→우



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

- 막다른 길로 탐색할 가능성 있음  
→ 다시 돌아와서 재탐색 가능하다  
→ 그렇담 DFS도 가능한 거 아닐까..?

# 다른 미로를 살펴봅시다

## ● DFS

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

## ● BFS

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

# 다른 미로를 살펴봅시다

## ● DFS

1	1	1	1
2	1	0	1
0	0	0	1
0	0	0	1

## ● BFS

1	2	1	1
2	1	0	1
0	0	0	1
0	0	0	1

# 다른 미로를 살펴봅시다

## ● DFS

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

## ● BFS

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

# 다른 미로를 살펴봅시다

## ● DFS

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

## ● BFS

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

# 다른 미로를 살펴봅시다

## ● DFS

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

## ● BFS

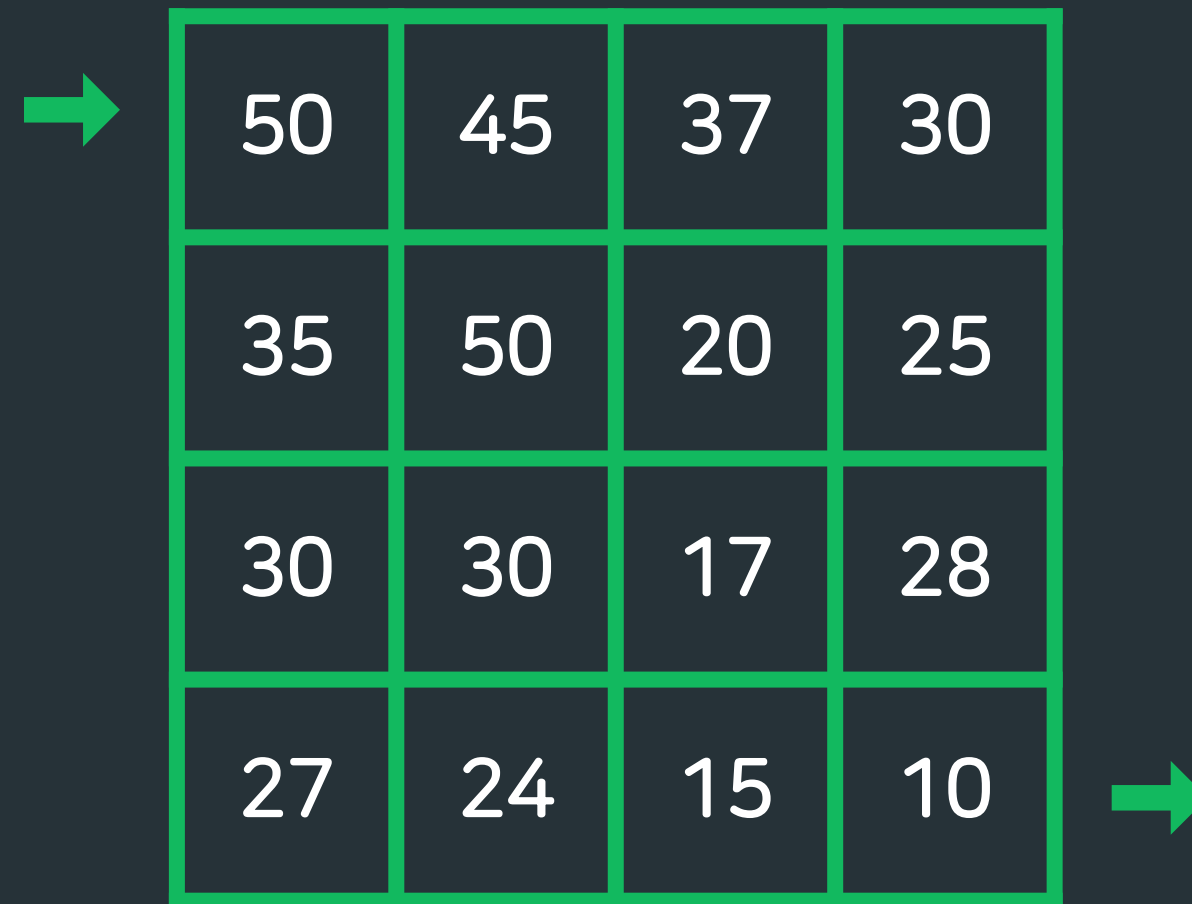
1	2	3	4
2	3	0	5
0	0	0	1
0	0	0	1

● DFS는 깊게 탐색하기 때문에,  
최단 거리를 구하지 못한다!

→ 최단 거리 문제에 적합한 건 BFS

## 이번엔 이 문제를 볼까요

- $N \times M$ 으로 표현되는 지도가 있고 각 칸마다 지점의 **높이**가 쓰여있다.
- (1,1)에서 시작해서 (N,M)까지 항상 **높이가 더 낮은 지점으로만 이동**할 때, 가능한 **이동 경로의 개수**를 구하여라

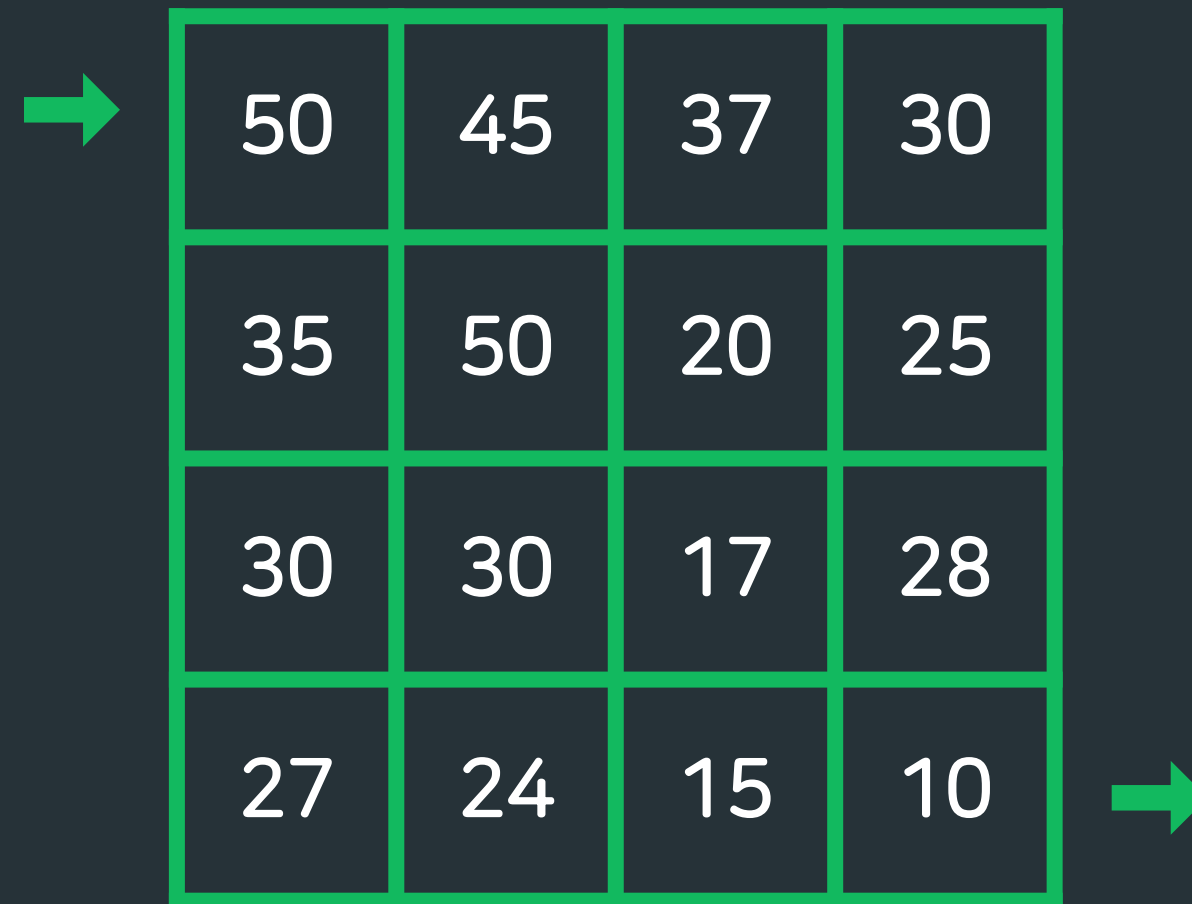


50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

→ BFS로 가능한 이동 경로를 계속 구해 나가면, 완전탐색에 가까우므로 **시간초과**

## 이번엔 이 문제를 볼까요

- $N \times M$ 으로 표현되는 지도가 있고 각 칸마다 지점의 **높이**가 쓰여있다.
- (1,1)에서 시작해서 (N,M)까지 항상 **높이가 더 낮은 지점으로만 이동**할 때, 가능한 **이동 경로의 개수**를 구하여라



50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

- BFS로 가능한 이동 경로를 계속 구해 나가면, 완전탐색에 가까우므로 **시간초과**
- 사실 DFS로 풀어도 **시간초과**



50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

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

→ 내리막길을 찾아가기 (DFS)

50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

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

→ 내리막길을 찾아가기 (DFS)

→ 오른쪽 끝까지 왔다면 다시 돌아간다. 이때, 경우의 수를 저장하며 돌아가기 (DP)

50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

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

→ 내리막길을 찾아가기 (DFS)

→ 오른쪽 끝까지 왔다면 다시 돌아간다. 이때, 경우의 수를 저장하며 돌아가기 (DP)

50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

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

→ 내리막길을 찾아가기 (DFS)

→ 오른쪽 끝까지 왔다면 다시 돌아간다. 이때, 경우의 수를 저장하며 돌아가기 (DP)

50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

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

→ 내리막길을 찾아가기 (DFS)

→ 오른쪽 끝까지 왔다면 다시 돌아간다. 이때, 경우의 수를 저장하며 돌아가기 (DP)

→ 이미 DP 값이 존재한다면 더 이상 탐색하지 않고 바로 리턴

50	45	37	30
35	50	20	25
30	30	17	28
27	24	15	10

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

→ 내리막길을 찾아가기 (DFS)

→ 오른쪽 끝까지 왔다면 다시 돌아간다. 이때, 경우의 수를 저장하며 돌아가기 (DP)

→ 이미 DP 값이 존재한다면 더 이상 탐색하지 않고 바로 리턴

## /<> 7576번 : 토마토 - Silver 1

### 문제

- $N * M$  격자 모양 상자에 토마토를 한 칸씩 넣어서 보관
- 보관된 토마토는 익거나(1) 익지 않은 토마토(0)로 나뉨
- 하루가 지나면 익은 토마토들의 인접한 곳에 있는 토마토가 영향을 받아서 익음
- 며칠이 지나야 모든 토마토가 익는지 구하는 문제. 최소 일수를 구해라
- 단, 상자의 일부 칸에는 토마토가 없을 수 있음(-1)

### 제한 사항

- 입력 범위는  $2 \leq N, M < 1,000$



## 예제 입력

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

\* !주의! **M(열)**이 먼저 주어짐



## 예제 출력

```
6
```





## Hint

1. 인접한 자식 노드를 어떻게 탐색해야 할까요? 일단 하루가 지나고 익는 토마토는 인접한 모든 자식 노드일거예요
2. 모두 익는데까지 걸리는 최소 일수를 구해야 하네요!

# 직접 탐색해 봅시다







	-1	0	0	0	0
0	-1	0	0	0	0
0	0	0	0	-1	0
0	0	0	0	-1	

# 직접 탐색해 봅시다

	-1	0	0	0	0
	-1	0	0	0	0
0	0	0	0	-1	
0	0	0	0	-1	






● DAY + 1

# 직접 탐색해 봅시다

	-1	0	0	0	0
	-1	0	0	0	
	0	0	0	-1	
0	0	0	0	-1	

● DAY + 2

# 직접 탐색해 봅시다

	-1	0	0	0	
	-1	0	0		
		0	0	-1	
	0	0	0	-1	













● DAY + 3

# 직접 탐색해 봅시다

	-1	0	0		
	-1	0			
			0	-1	
		0	0	-1	

● DAY + 4





















# 직접 탐색해 봅시다

	-1	0			
	-1				
				-1	
			0	-1	

● DAY + 5



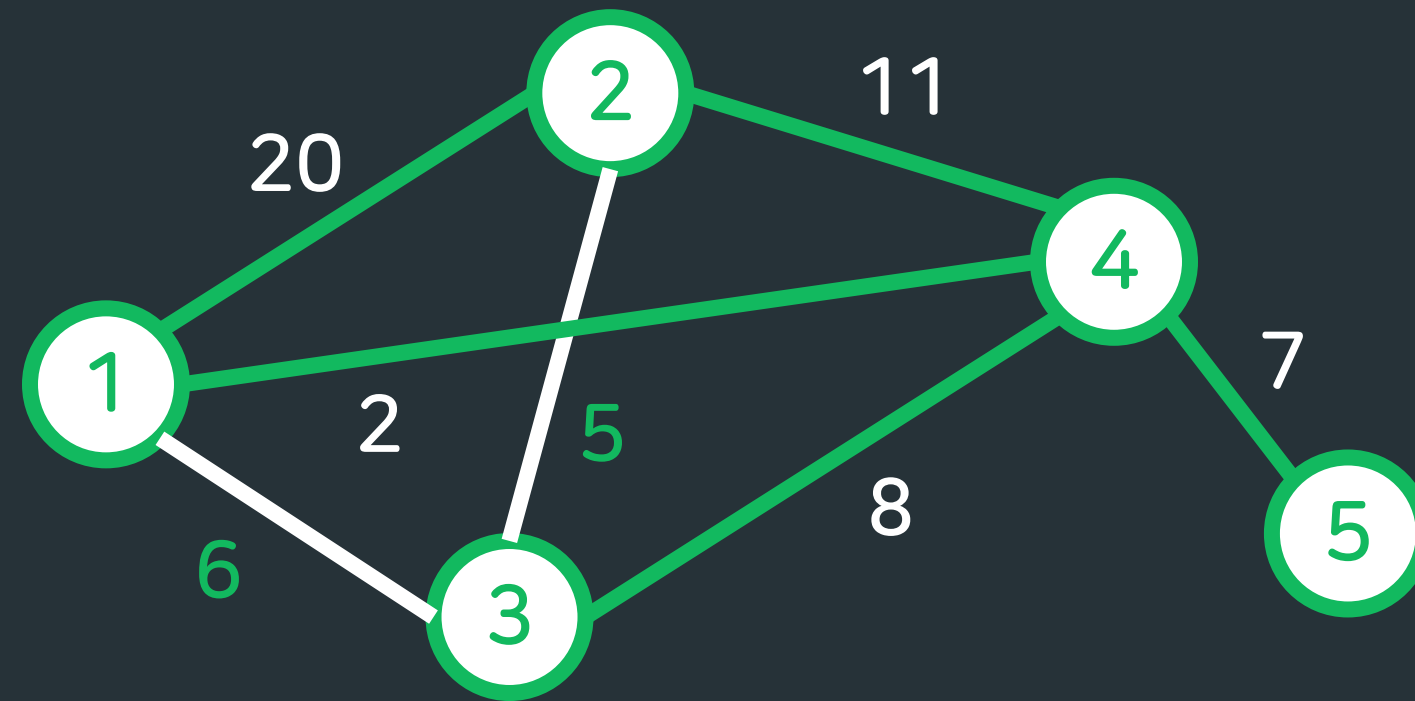
# 직접 탐색해 봅시다

	-1				
	-1				
				-1	
				-1	

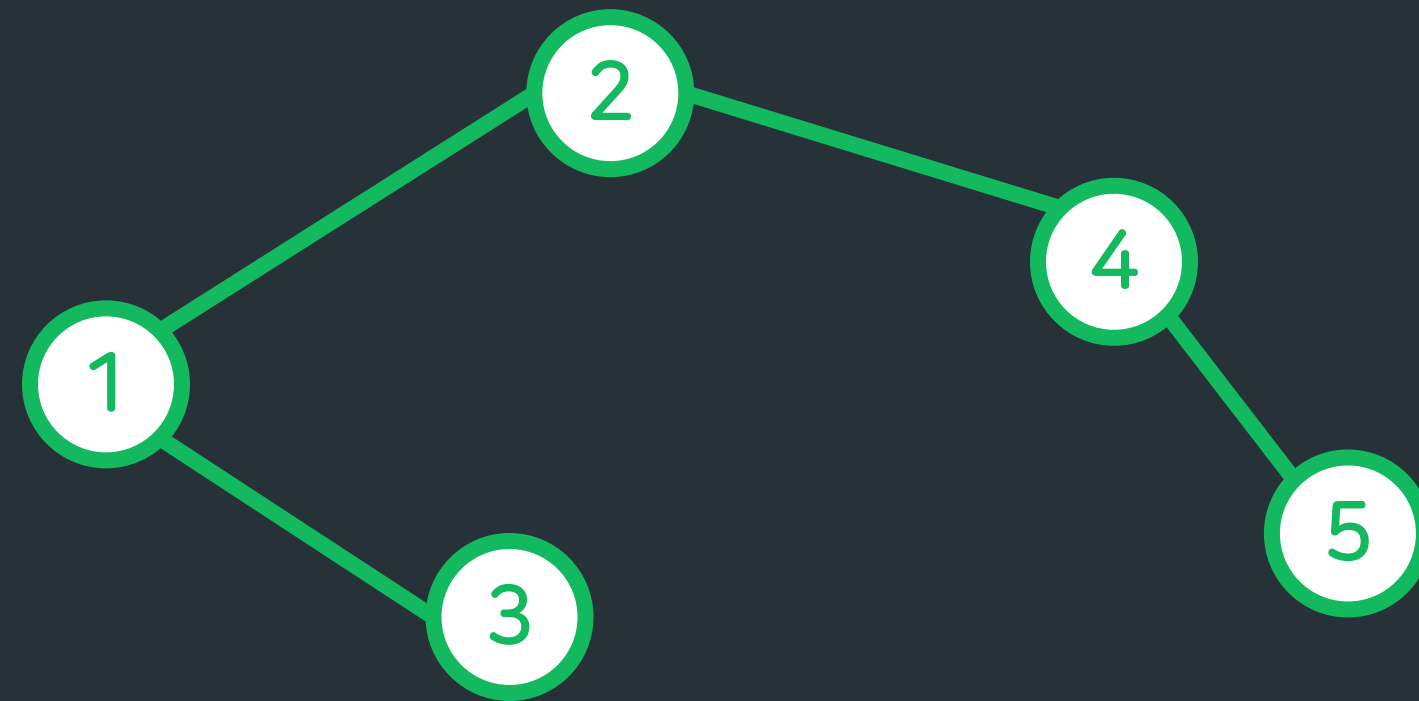
● DAY + 6

## 그래프 알고리즘

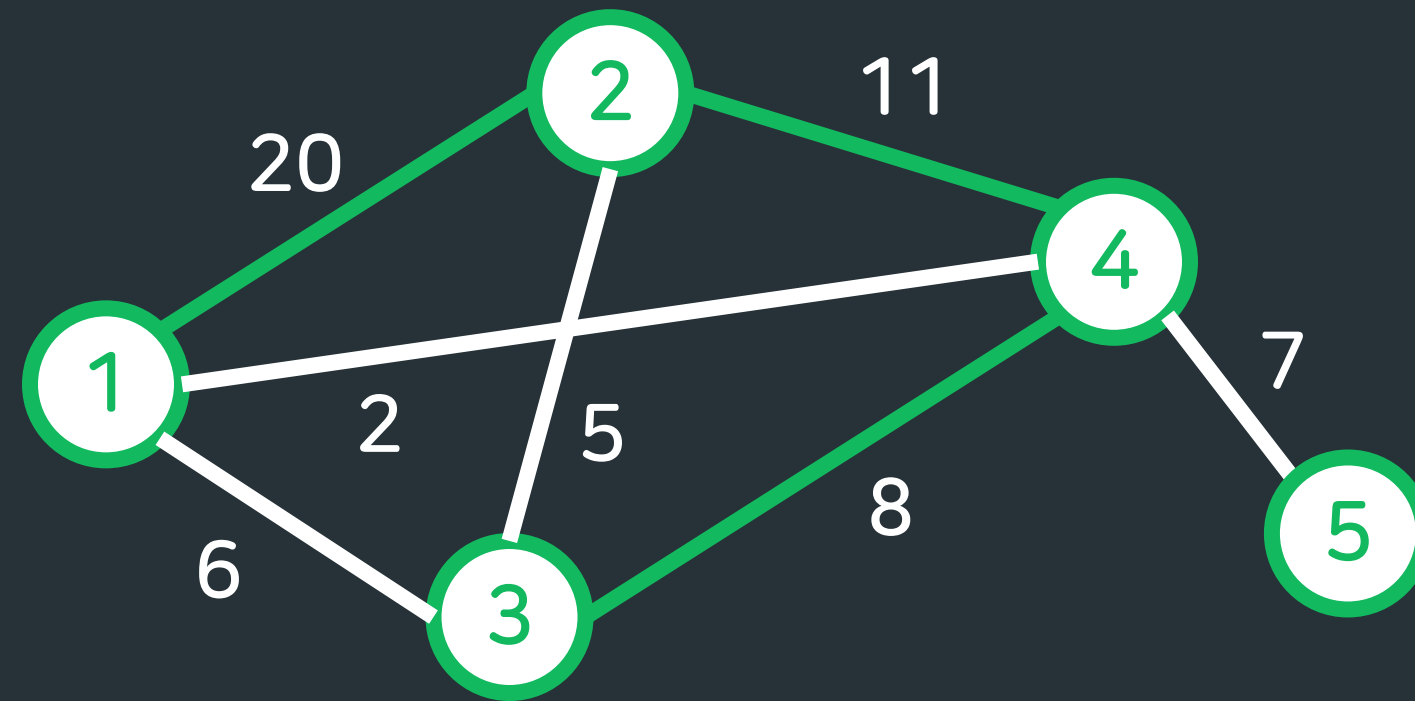
- 정점 사이의 최단 거리를 구해야 할 때 → 최단 경로 알고리즘
- 사이클이 없는 그래프를 다룰 때 → 트리
- 가중치의 총 합이 가장 작은 트리를 만들어야 할 때 → 최소 비용 신장 트리
- 방향 그래프에서 선후관계를 기반으로 정점을 나열해야 할 때 → 위상 정렬
- 그래프에서 정점 사이의 관계를 서로소로 정의해 집합으로 나누어야 할 때 → 유니온 파인드



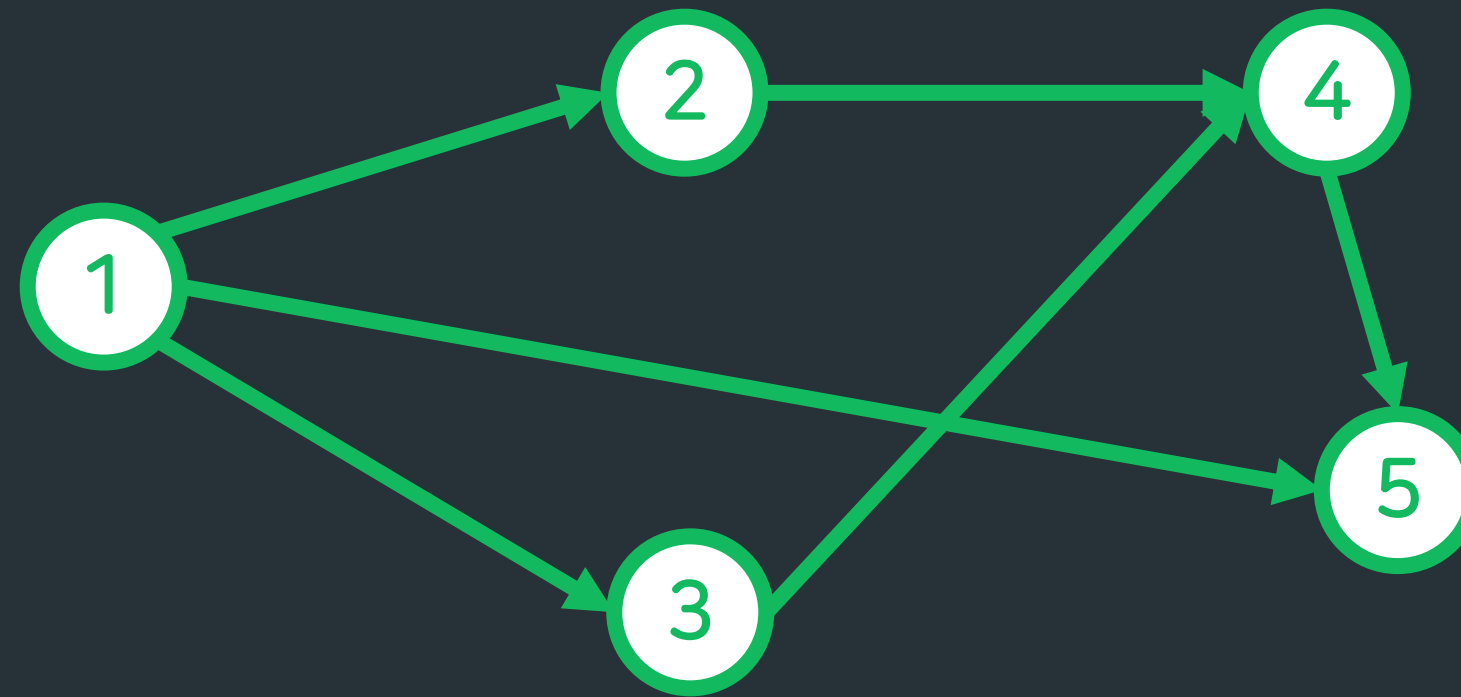
정점 사이의 최단 거리를 구해야 할 때 → 최단 경로 알고리즘



사이클이 없는 그래프를 다룰 때 → 트리

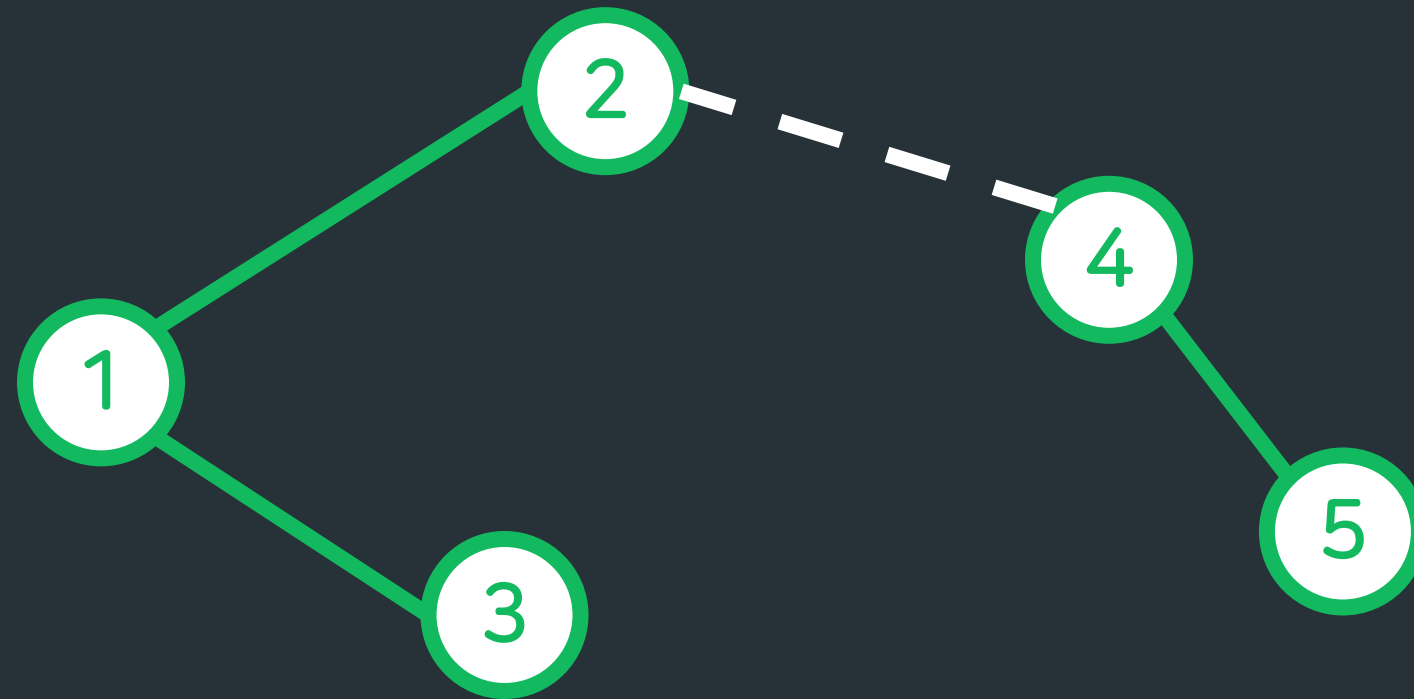


가중치의 총 합이 가장 작은 트리를 만들어야 할 때 → 최소 비용 신장 트리



1 -> 2 -> 3 -> 4 -> 5

방향 그래프에서 **선후 관계**를 기반으로 **정점을 나열**해야 할 때 → 위상 정렬



그래프에서 정점 사이의 관계를 서로소로 정의해 집합으로 나누어야 할 때 → 유니온 파인드

## 정리

- 그래프는 인접 행렬과 인접 리스트로 구현 가능
- 깊이 우선 탐색이라 불리는 DFS와 너비 우선 탐색이라 불리는 BFS가 대표적인 탐색 알고리즘
- 해당 노드를 끝까지 파고들어야 한다면 DFS, 해당 노드 주변을 먼저 탐색한다면 BFS
- DFS는 stack, 재귀 호출을 통해 구현, BFS는 queue를 통해 구현
- DFS와 BFS 모두 방문 체크 꼭 필요

## 이것도 알아보세요!

- 방문 체크를 하는 방법이나 위치는 문제에 따라 달라질 수 있어요. 코드가 어떻게 돌아가는지 직접 디버깅을 많이 해봅시다
- DFS는 재귀 호출로 구현한다는 점에서 백트래킹과 비슷한 점이 많아요. 백트래킹과 DFS의 차이를 알아봅시다



## 3문제 이상 선택

/<> 1697번 : 숨바꼭질 - Silver 1

/<> 4963번 : 섬의 개수 - Silver 2

/<> 19538번 : 루머 - Gold 4

/<> 1520번 : 내리막 길 - Gold 4

 2021 카카오 채용연계형 인턴십 : 거리두기 확인하기 - Level 2

 깊이/너비우선탐색(DFS/BFS): 단어 변환 - Level 3