Quest

우아한테크코스 알고리즘 스터디
Week 4. 그래프 이론과 깊이 우선 탐색(DFS)

by 요술토끼(curious.wzrabbit@gmail.com)

이번 주차의 목표

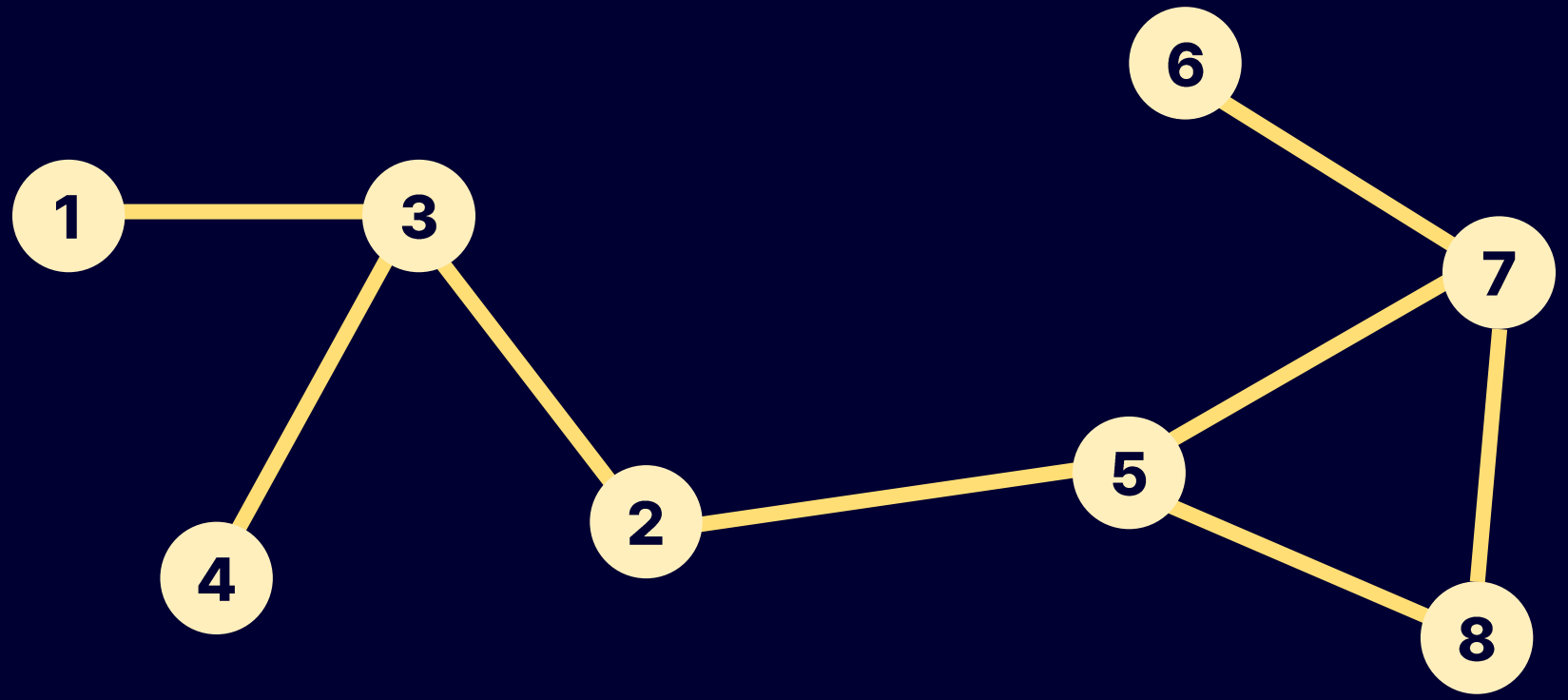
- 이번 주차부터 드디어 그래프의 기본적인 개념, 그리고 그래프를 탐색하는 방법을 학습하게 됩니다! 지금까지 저랑 배워오셨던 스택, 큐, 재귀가 널리 쓰일 것입니다.
- 본 주차에서는 알고리즘에서 다루는 그래프가 무엇인지, 그리고 그래프 탐색이 무엇인지를 다룹니다.
- 또한, 그래프를 **깊이 우선 탐색(DFS)**를 이용해 탐색을 하는 방법을 다루고, 이를 이용해 기초적인 그래프 문제를 풀어 보고자 합니다.
- 여러분에게 정해 드리는 목표는 **A, B번 문제의 해결**입니다. 깊이 우선 탐색을 배우는 기간을 감안하여 연습은 **2주 동안** 진행됩니다.

그래프란

- 그래프는 **정점**과 **간선**으로 이루어진 **자료구조**입니다.
- 정점은 특정한 “지점” 으로 흔히 비유됩니다. node나 vertex라고도 많이 불립니다.
- 간선은 “길” 로 흔히 비유됩니다. edge라고 많이 불립니다.
- 정점은 주로 **점**으로 나타내고, 간선은 주로 **선** 또는 **화살표**로 나타냅니다. 아래처럼요!

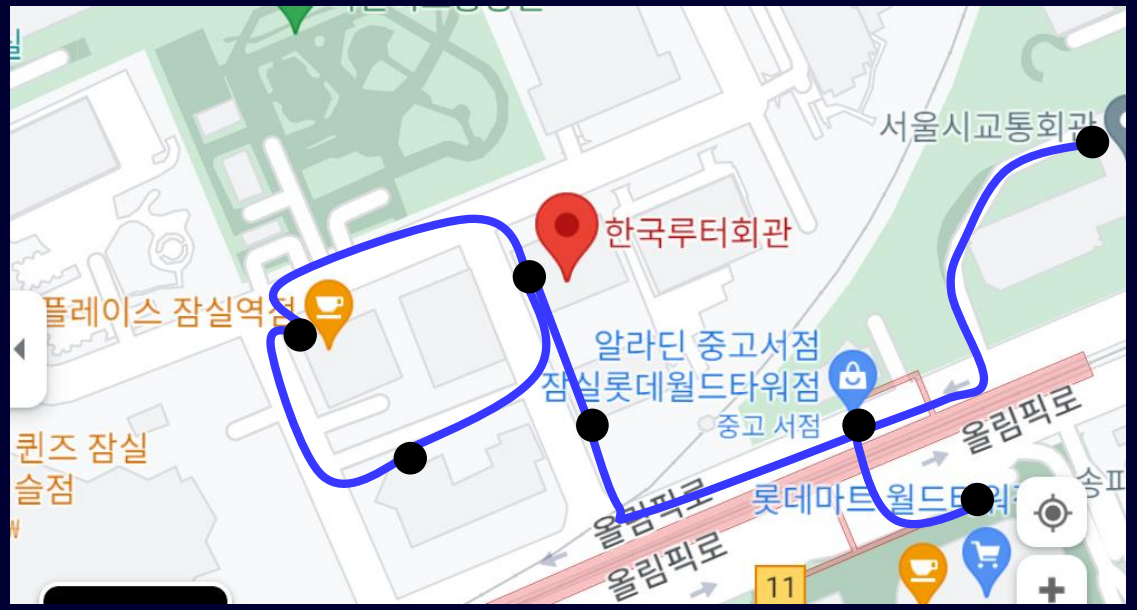
그래프란

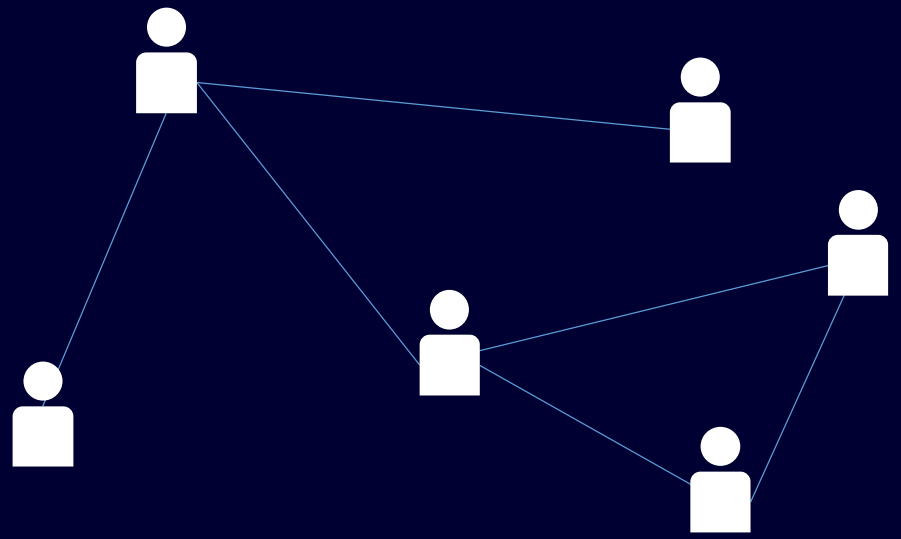
- 정점과 정점은 간선으로 연결될 수 있으며, 이 경우 두 정점은 서로 **인접하다**고 이야기합니다.
- 아래는 그래프의 예시입니다. 정점에 적혀 있는 수는 정점의 번호이며, 그냥 특정 정점을 가리키기 위함입니다. 대부분의 알고리즘 문제에서는 이런 식으로 정점에 번호를 붙입니다.



그래프란

- 그래프는 **현실의 복잡한 상호작용을 비롯한, 연결 관계들을 표현하는 데** 적합한 자료구조입니다. 그래프를 이용해 현실의 여러 상황을 모델링할 수 있습니다.
  - 도시와 도로 – 도시가 정점, 도로는 간선
  - SNS 친구 관계 – 사용자가 정점, 친구 관계는 간선. 서로 연결되어 있을 경우 친구 관계





그래프란

- 여기까지가 기본적인 그래프의 정의입니다.
- 다음 페이지에서부터는 여러 그래프의 유형 중 지금 당장 접해보면 좋을 두 유형만 다루고 넘어가도록 하겠습니다. 나머지 유형은 지금 당장 알아보기보다는, 필요할 때 알아보셔도 충분합니다.
- 문제에 단순 그래프라는 용어가 등장 → 어, 단순 그래프가 뭐지? → 검색
  이 정도면 충분하다는 뜻입니다.

무방향 그래프와 방향 그래프

- 그래프에서 간선을 이용하여 정점에서 정점으로 이동할 때에는, 양방향 모두로 자유로이 이동할 수 있는 경우도 있지만, **한 방향으로만 이동할 수 있는 경우도 있습니다.**
- 예를 들어 말하면 두 정점 A와 B가 있고 이를 잇는 간선이 있을 때, A → B, B → A 의 방법으로 이동하는 것이 모두 가능한 간선도 있는 반면 A → B 의 방법으로는 이동할 수 있지만 B → A 로의 이동은 불가능한 간선도 있다는 의미입니다.
- **양쪽으로 이동이 가능한 간선의 경우 선으로** 나타내며, **한쪽으로만 이동이 가능한 간선의 경우 화살표로** 나타냅니다. 전자의 경우를 **양방향 간선**, 후자의 경우를 **단방향 간선**이라고 부릅니다.

1번에서 2번 정점으로 이동 가능

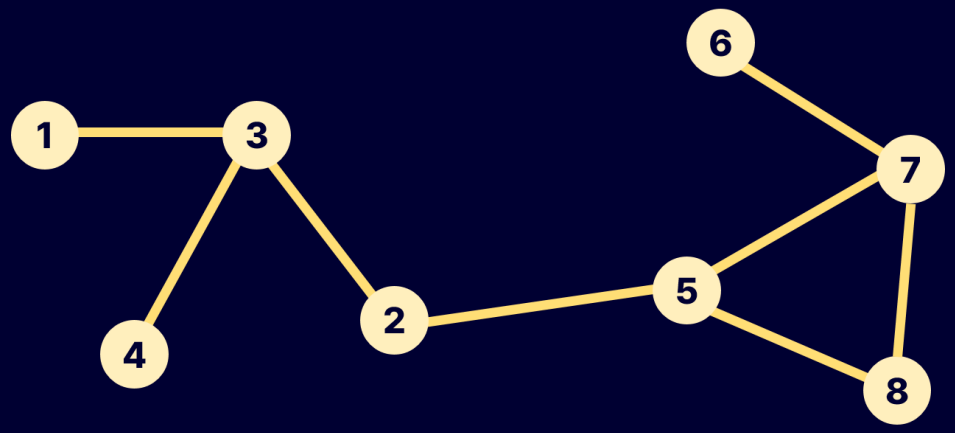
2번에서 1번 정점으로 이동 가능

1번에서 2번 정점으로 이동 가능

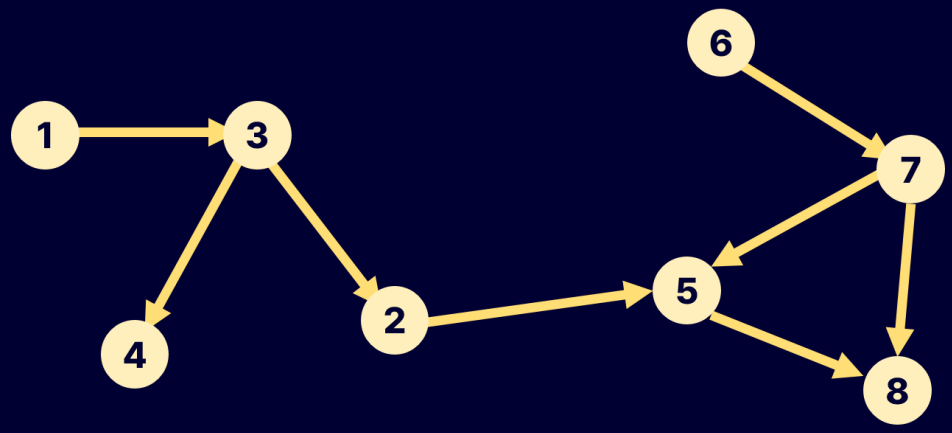
2번에서 1번 정점으로 이동 불가

무방향 그래프와 방향 그래프

- 양방향 간선으로 이루어진 그래프를 **무방향 그래프(undirected graph)**라고 하며, 단방향 간선으로 이루어진 그래프를 **방향 그래프(directed graph)**라고 합니다.



[무방향 그래프]



[방향 그래프]

무방향 그래프와 방향 그래프

용어가 정말 자주 나옵니다. 그러니 꼭 알아두셔야 합니다!

output: standard output

ZS the Coder has drawn an undirected graph of n vertices numbered from 0 to n - 1 and m edges between them. Each edge of the graph is weighted, each weight is a **positive integer**.

The next day, ZS the Coder realized that some of the weights were erased! So he wants to

문제

방향그래프가 주어지면 주어진 시작점에서 다른 모든 정점으로의 최단 경로를 구하는 프로그램을 작성하시오. 단, 도

무방향 그래프 G가 주어진다. 이 그래프에는 self edge는 존재하지 않지만 존재할 수 있다. 이때, 그래프 G의 edge들 중 몇 개를 골라서 색칠하려고 한

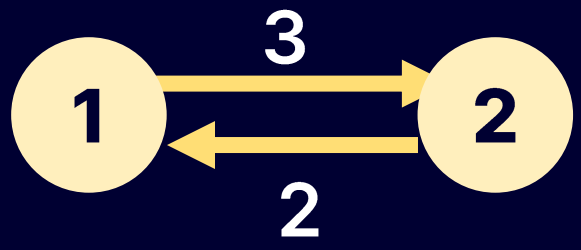
Problem Statement

There is a directed graph with N vertices and M edges.

, and the i-th directed edge goes from vertex a_i to vertex

가중치 그래프

- 이전에 그래프 모델링의 예시를 설명할 때 도시와 도로를 예로 들어 설명한 적이 있습니다! 도시와 도로가 연결되어 있는 것을 표현할 수 있는 것까지는 알겠는데, **각 도시 간의 거리**, 아니면 **통행료** 등을 표현할 수 있으려면 어떻게 해야 할까요?
- 간선에 가중치(비용)을 나타낸다면 가능합니다! 아래의 예시를 확인해 보세요. 간선에 수가 표기되어 있다면 이는 가중치(비용)을 의미합니다. **해당 간선을 이용하여 이동할 때 드는 비용을 의미합니다.**

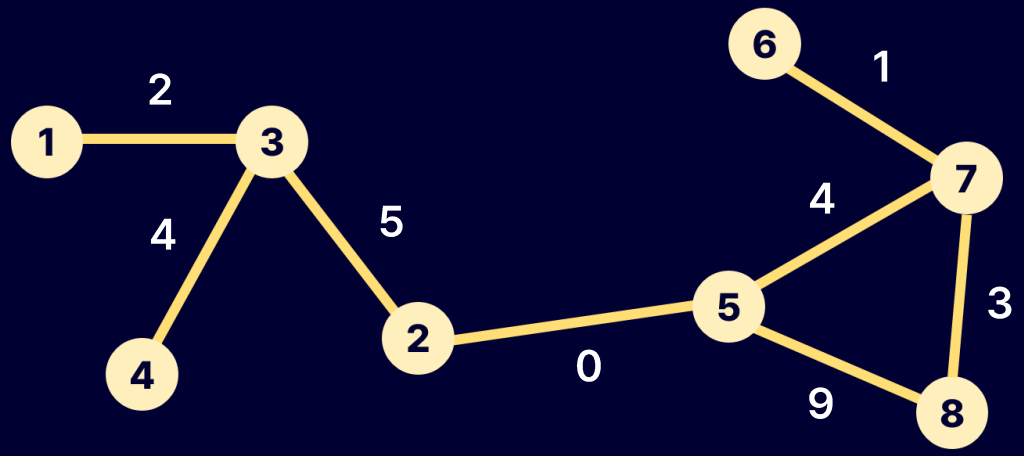


1번에서 2번 정점으로 이동 가능, 가중치는 5
2번에서 1번 정점으로 이동 가능, 가중치는 5

1번에서 2번 정점으로 이동 가능, 가중치는 3
2번에서 1번 정점으로 이동 가능, 가중치는 2

가중치 그래프

- 간선에 가중치가 부여되어 있는 그래프를 **가중치 그래프**라고 부릅니다.
- 가중치 그래프를 활용하면 정점 간의 연결 정보 뿐만 아니라, 정점 간의 연결 비용도 알 수 있습니다. 가장 흔히들 비유를 댈 때 정점은 도시, 간선은 도로, 그리고 가중치는 거리로들 이야기하죠.
- 이번 스터디에서는 학습하지는 않지만 다익스트라 알고리즘 등을 이용하여 도시 간의 최단 거리를 구할 때에는 이 가중치 그래프가 정말 단골로 많이 쓰입니다.



[가중치 그래프]

가중치 그래프

- 이로써 여러분께 그래프의 개념, 그리고 기본적인 문제풀이에서 필요한 방향/무방향 그래프, 그리고 가중치 그래프를 설명했습니다.
- 다음 페이지부터는 **그래프 탐색**이 무엇인지, 그리고 이번 주제에서 다루게 될 **DFS 알고리즘**을 이용해 그래프를 탐색하는 방법을 본격적으로 다루도록 하겠습니다.

그래프 탐색이란

- 그래프 탐색이란, 보통 **특정 정점에서 출발하여 정점과 연결된 정점들을 간선을 통해 탐색해 나가는 과정**을 이야기합니다.
- 탐색 과정에서 찾고자 했던 정점을 찾거나, 연결되어 있는 정점의 개수를 구하는 등 원하는 정보를 찾고는 합니다.
- 그래프 탐색에 대해 너무 추상적으로 느끼시는 분들이 많이 계십니다. 저 또한 그랬습니다. 저는 그럴 때 미로 찾기를 설명하곤 합니다. 미로의 각 지점을 정점으로, 그리고 미로의 지점끼리 연결되어 있는 길을 간선으로 생각한다면, 그래프 탐색을 하는 과정은 미로의 여러 부분을 헤집으면서 출구를 찾는 것과도 생각할 수 있습니다.

깊이 우선 탐색 (DFS)

- 그래프를 탐색하기 위해서는 그래프 탐색을 위한 알고리즘을 사용하셔야 합니다. 보통 **깊이 우선 탐색(DFS)**와 **너비 우선 탐색(BFS)**가 여기서 많이 언급됩니다만, 이번 주차에서는 비교적 직관적인 **깊이 우선 탐색(DFS)**부터 시작해 보도록 하겠습니다!

- **깊이 우선 탐색(DFS)**는 **재귀**를 이용하여 구현할 수도 있고, **스택**을 이용하여 구현할 수도 있습니다. 원하시는 방식을 사용하시기 바랍니다.

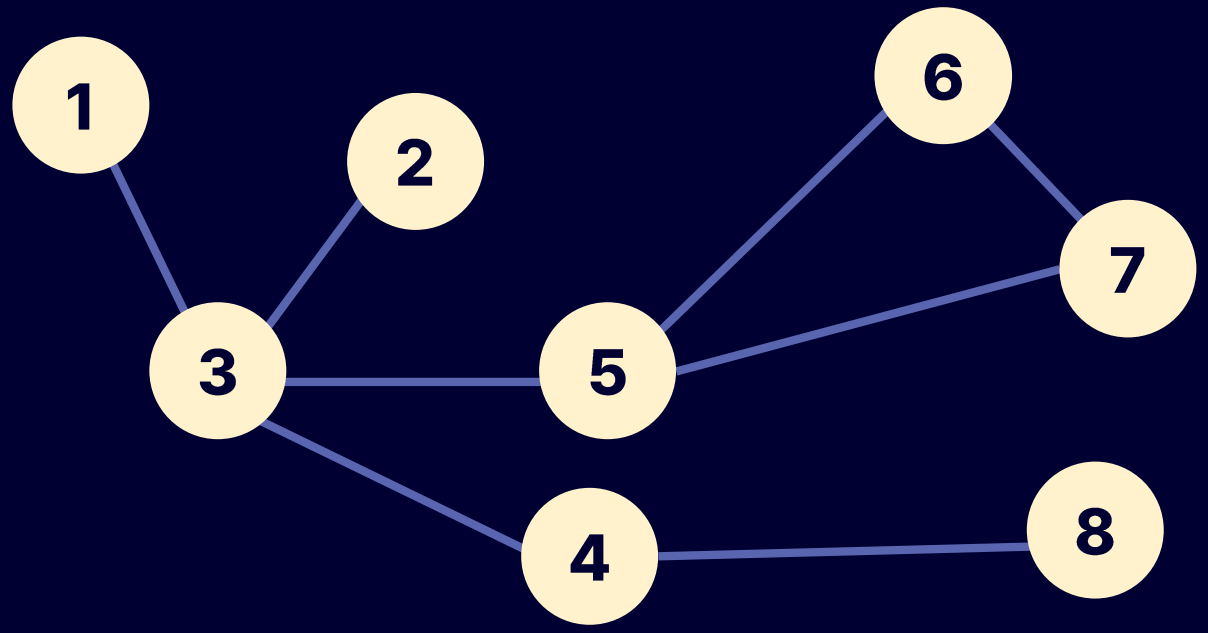
- 깊이 우선 탐색은 각 정점을 방문할 때마다 바로 눈앞에 보이는 인접한 정점을 찾아 계속해서 깊숙히 파내려가는 탐색 방식입니다.

- 다음 페이지에서부터는 본격적으로 그래프를 DFS 알고리즘을 이용해 탐색해 봅시다! 이제부터 깊이 우선 탐색을 이야기할 때는 편의상 DFS로 줄여 부르겠습니다.

깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

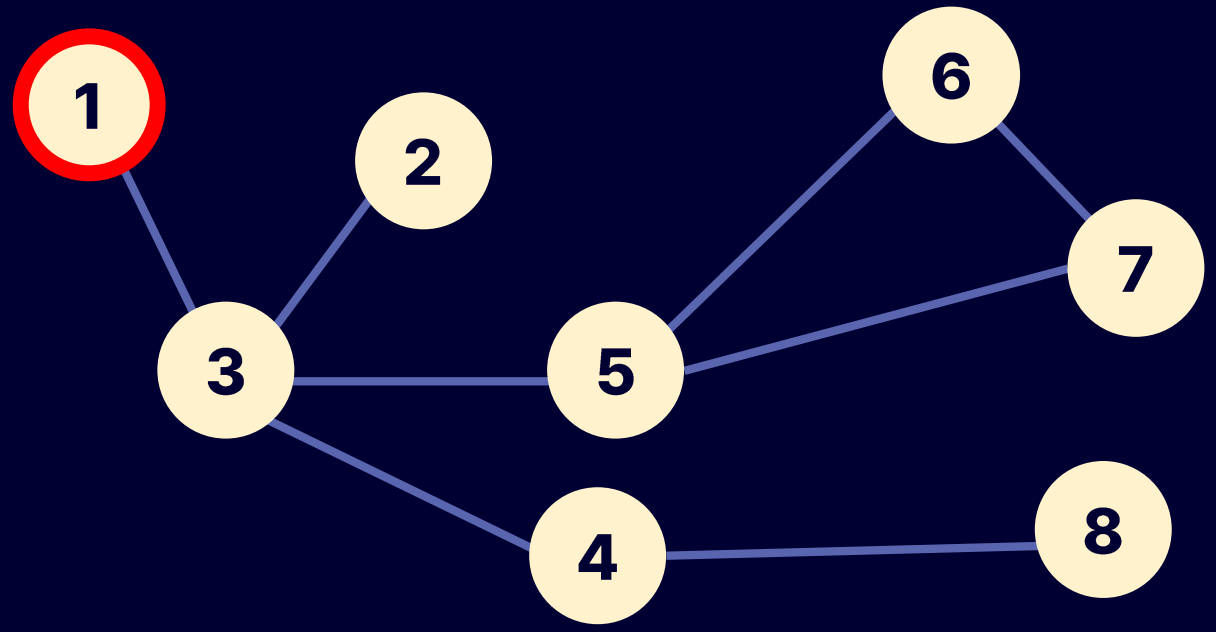
여기 우리가 방문해야 하는 무방향 그래프가 있습니다. 우선 **탐색을 시작할 정점**을 하나 고르겠습니다. 여기서는 **1번**을 골라보겠습니다.



깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

- 1번 정점의 방문을 진행합시다.
- 정점을 방문했을 때는, 가장 먼저 1번 정점을 방문했다는 사실을 기록하여야 합니다.



방문 처리를 한다는 것

- 왜 각 정점을 방문할 때마다 굳이 방문했다는 사실을 따로 기록해야 할까요? 이는 방문 처리를 하지 않을 경우 **정점의 방문이 무한히 반복, 즉 그래프 탐색이 무한히 진행되는 현상이 일어날 수 있기 때문**입니다. 아래의 예시를 참고하시면 될 것 같습니다.
- 방문 처리를 함으로써, **새로운 정점을 방문할 때마다 방문 여부를 먼저 검사**하여 이미 방문이 되었던 정점이라면 그 정점으로는 탐색을 하지 않게 할 수 있습니다. 이렇게 하면 **각 정점이 한 번씩만 방문**되겠죠?

1번 정점에서 2번 정점 방문... 2번 정점에서 1번 정점 방문...
1번 정점에서 2번 정점 방문... 2번 정점에서 1번 정점 방문...
...무한 반복

방문 처리를 한다는 것

- 방문 처리는 배열, 리스트 등의 자료구조를 사용하시면 됩니다. `visited[정점 수]`와 같이요. 아래는 예시입니다.

```
visited = [true, false, false, true];
```

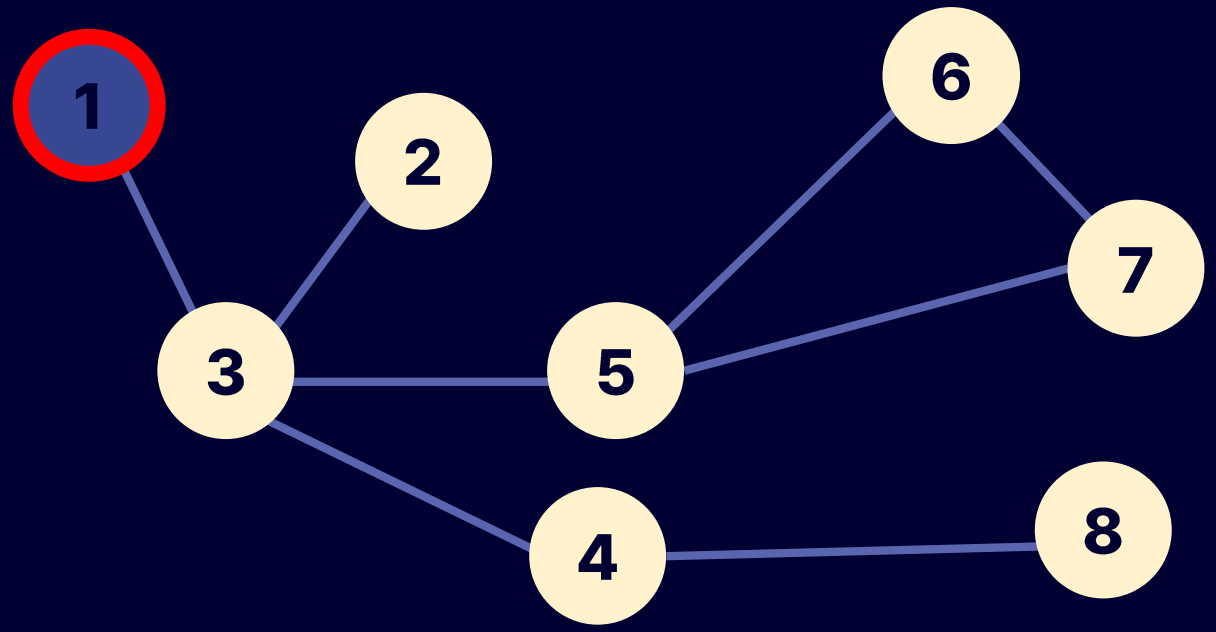
인덱스에 해당하는 정점의 방문 여부. boolean 값입니다.

깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1

- 방문 처리를 해야 하는 이유를 알았으니 계속해서 진행해 보죠. 1번 정점을 방문 처리했습니다.
- 이제 **1번 정점과 인접한 정점들을 다음 행선지로 삼읍시다.** 편의상 인접한 정점이 여러 개일 때는 정점의 번호가 작은 번호부터 방문한다고 해 보겠습니다(DFS의 규칙은 아닙니다).

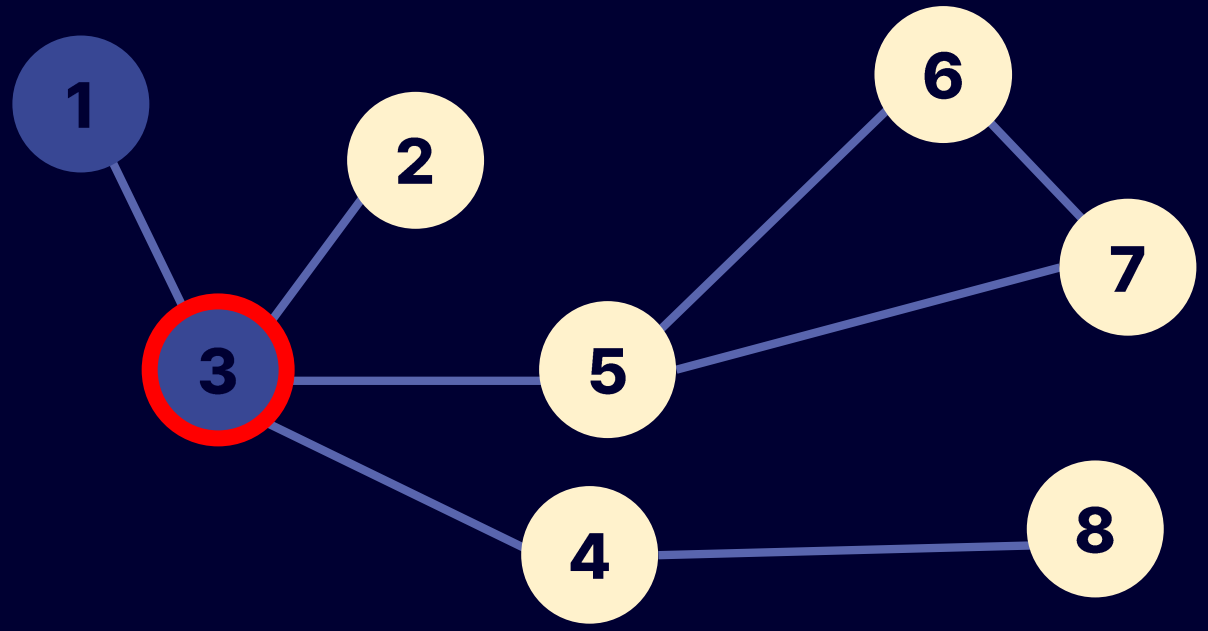


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 - 3

- 1번 정점과 인접한 정점은 3번 정점 뿐이므로, 3번 정점을 방문했습니다.
- 3번 정점을 **방문 처리**하고, **3번 정점과 인접한 정점들**을 찾아 봅시다. 1, 2, 4, 5번 정점이 3번 정점과 인접한 정점이군요. 이 정점들을 모두 방문해 보겠습니다.

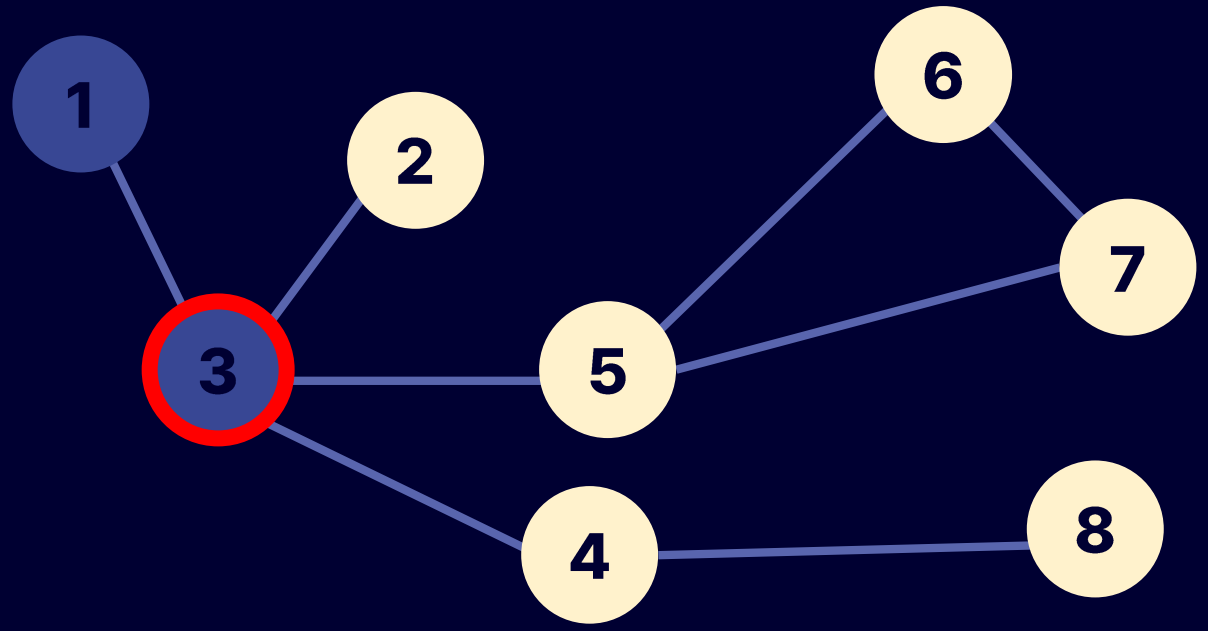


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 - 3

- 편의상 번호가 작은 정점부터 방문하자고 했으니 가장 번호가 작은 정점인 1번 정점부터 방문을 진행해야 하지만, 이미 방문 처리가 된 정점입니다.
- **이미 방문 처리된 정점은 방문하지 않겠습니다.** 그렇다면, 그 다음으로 작은 번호의 정점은 2번 정점이고, 이 정점은 방문이 되지 않았습니다. 따라서 2번 정점을 방문하겠습니다.



깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 - 2

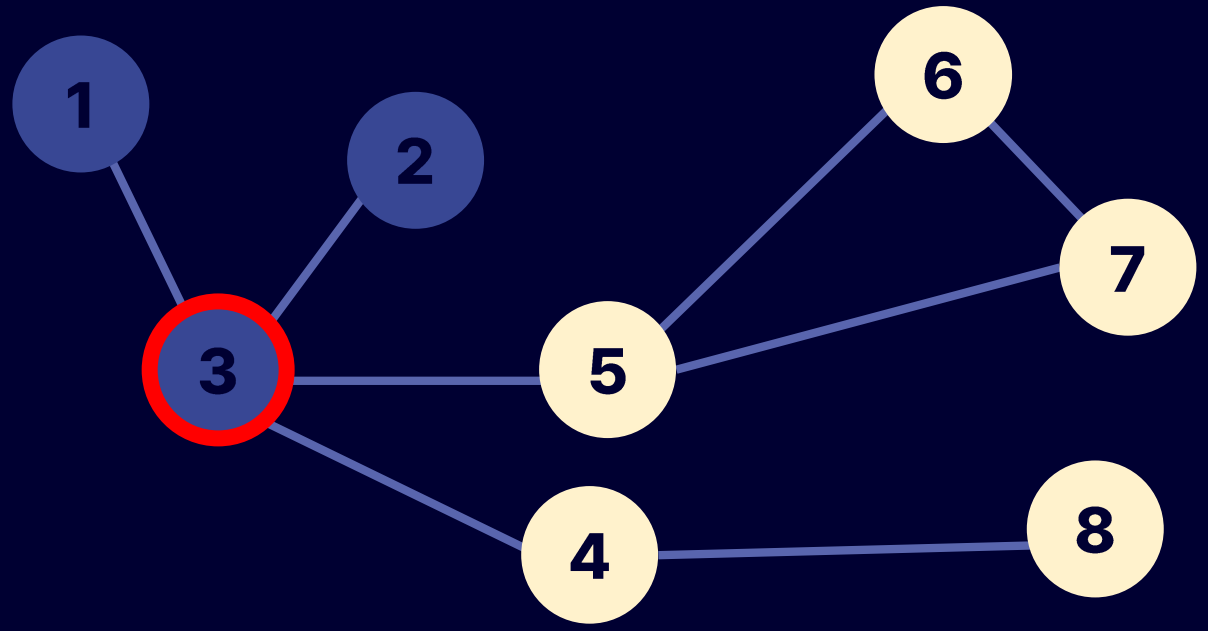
- 2번 정점을 방문했습니다.
- 2번 정점을 방문 처리하고 인접한 정점을 확인하려니 3번 정점만이 인접한 정점입니다. 이 경우 3번 정점으로의 탐색을 진행하여야 하지만, 3번 정점은 이미 방문된 정점입니다.
- **더 이상 방문할 정점이 없는 상황입니다.** 이 경우에는, 이전에 방문했던 정점으로 돌아가시면 됩니다. 이 때 방문 여부는 고려하지 않습니다.

깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 - 2

- 이제 다시 3번 정점 입장에서 생각해 봅시다. 1번 정점은 방문할 수 없었고, 2번 정점은 방문을 완료했습니다.
- 이제 남은 정점은 4, 5번 정점이며, 이 중 4번 정점으로의 방문을 진행합시다.

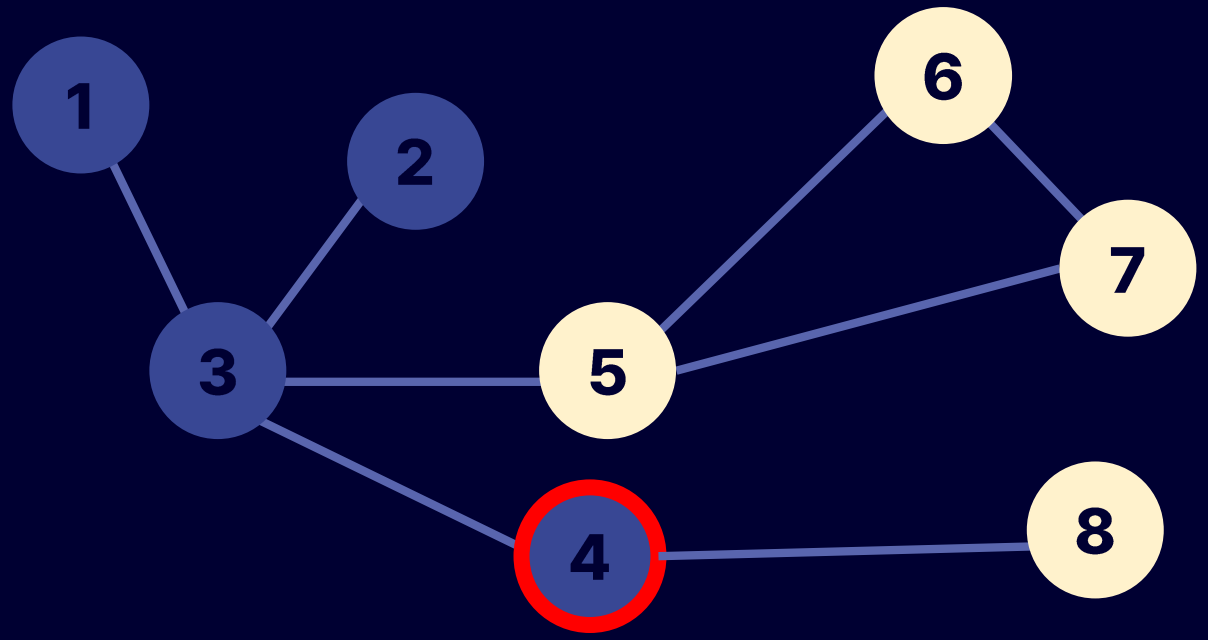


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 - 4

- 4번 정점을 방문했습니다. 인접한 정점은 3, 8번이며, 이 중 8번 정점만이 방문할 수 있는 정점입니다.
- 8번 정점으로의 방문을 진행합시다.

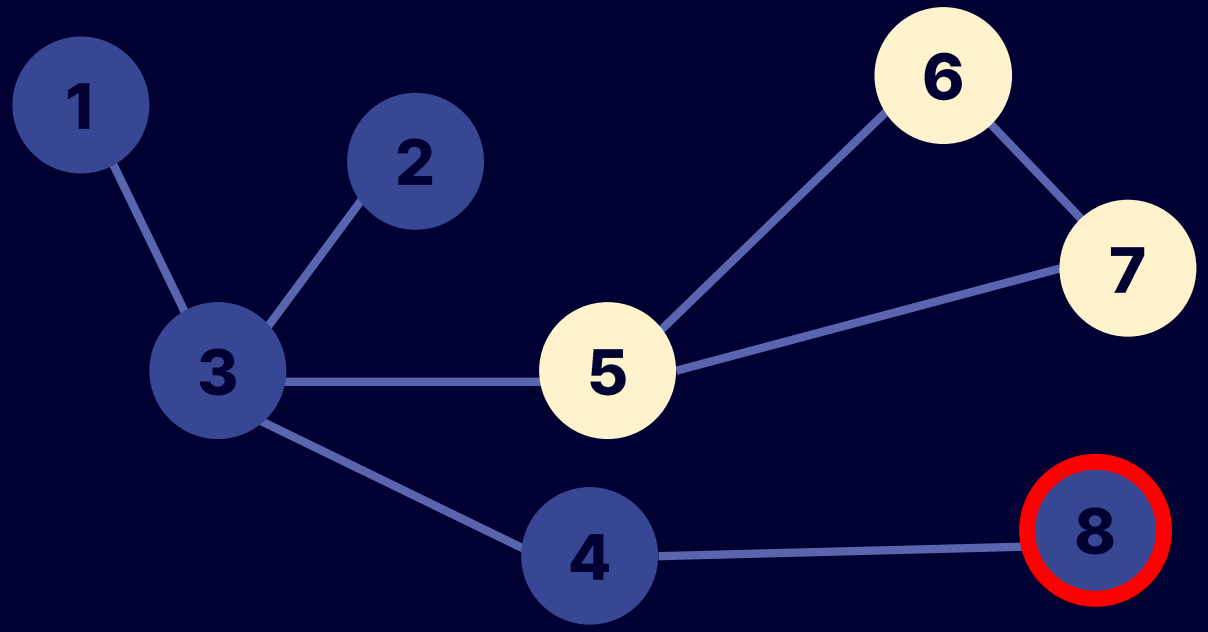


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 – 4 - 8

- 8번 정점을 방문했습니다. 더 이상 방문할 정점이 없으므로, 이전 정점인 4번으로 돌아가도록 합시다.

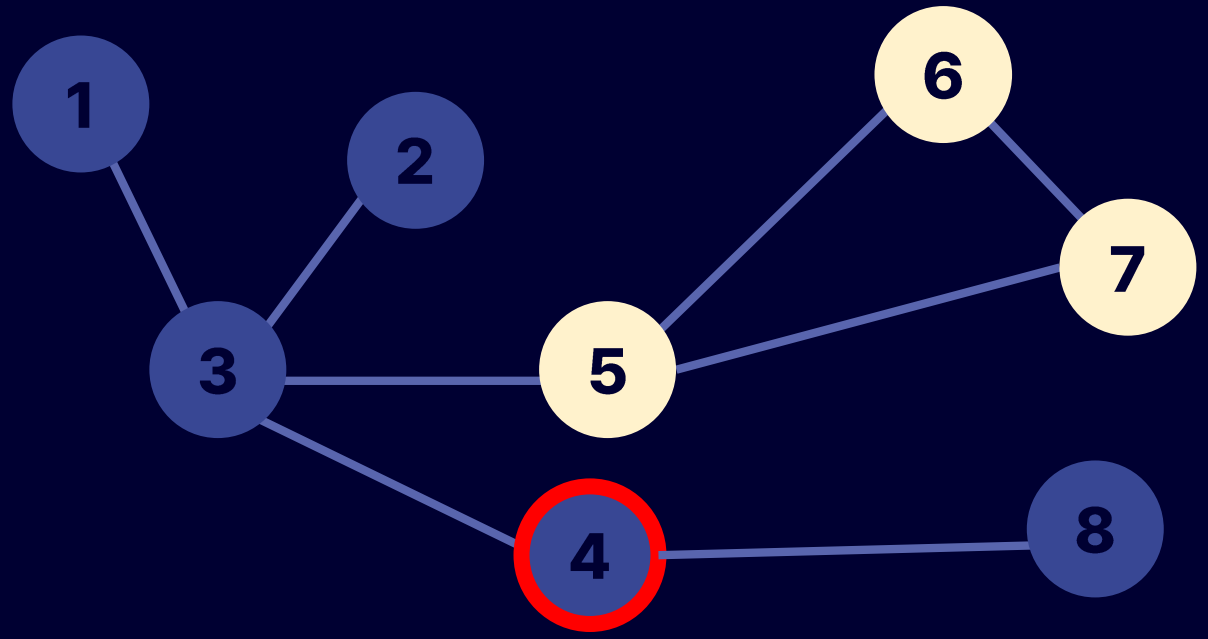


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 – 4 - 8

- 4번 정점으로 돌아왔습니다. 4번 정점 입장에서는 3번과 8번 정점이 인접한 정점인데, 두 정점 모두 방문 처리가 되어 있으므로 방문할 수 없습니다.
- 따라서 이전 정점인 3번 정점으로 돌아갑시다.

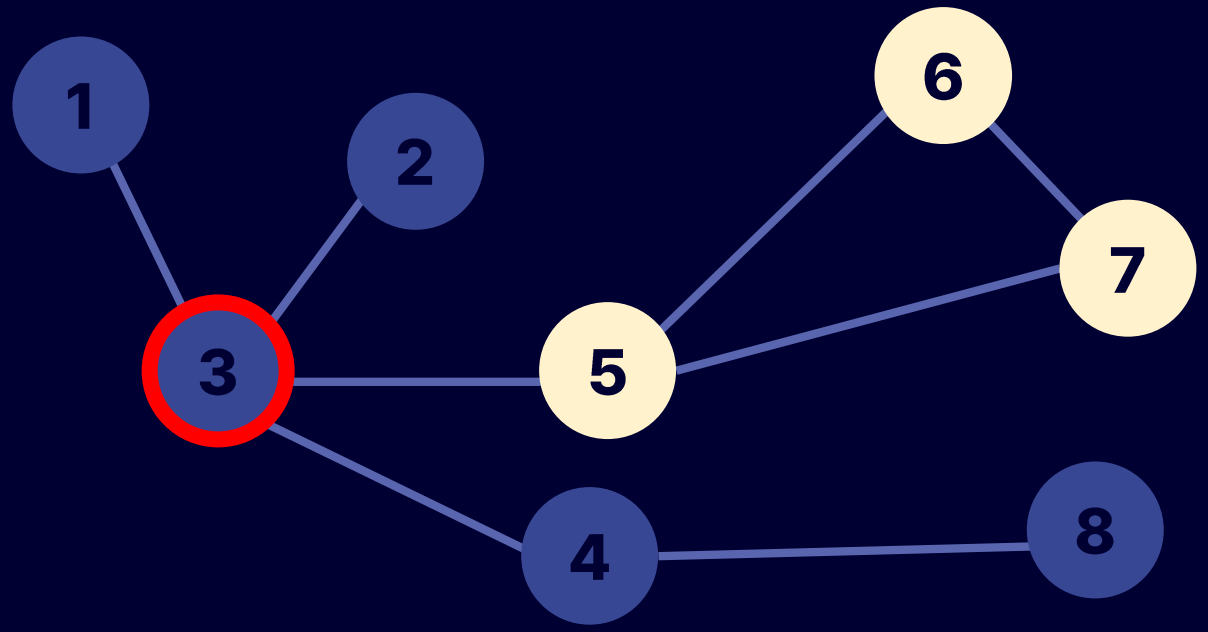


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 – 4 - 8

- 3번 정점으로 돌아왔습니다. 이제 3번 정점 입장에서는 5번 정점만 방문하면 끝이 납니다.
- 5번 정점으로의 탐색을 진행합시다.

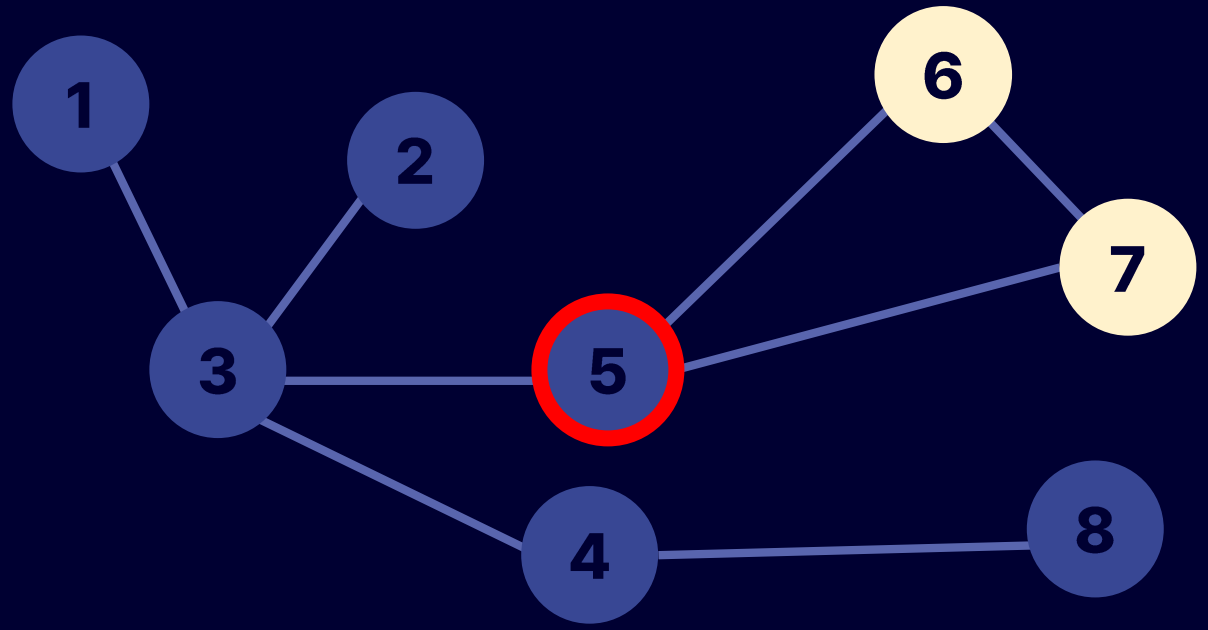


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 – 4 – 8 - 5

- 5번 정점을 방문했습니다. 5번 정점과 인접한 정점은 3, 6, 7번이며, 이 중 방문 처리가 되지 않은 6, 7번만 방문이 가능합니다. 6, 7번을 차례대로 방문합시다.
- 먼저 7번 정점을 방문해 보죠!

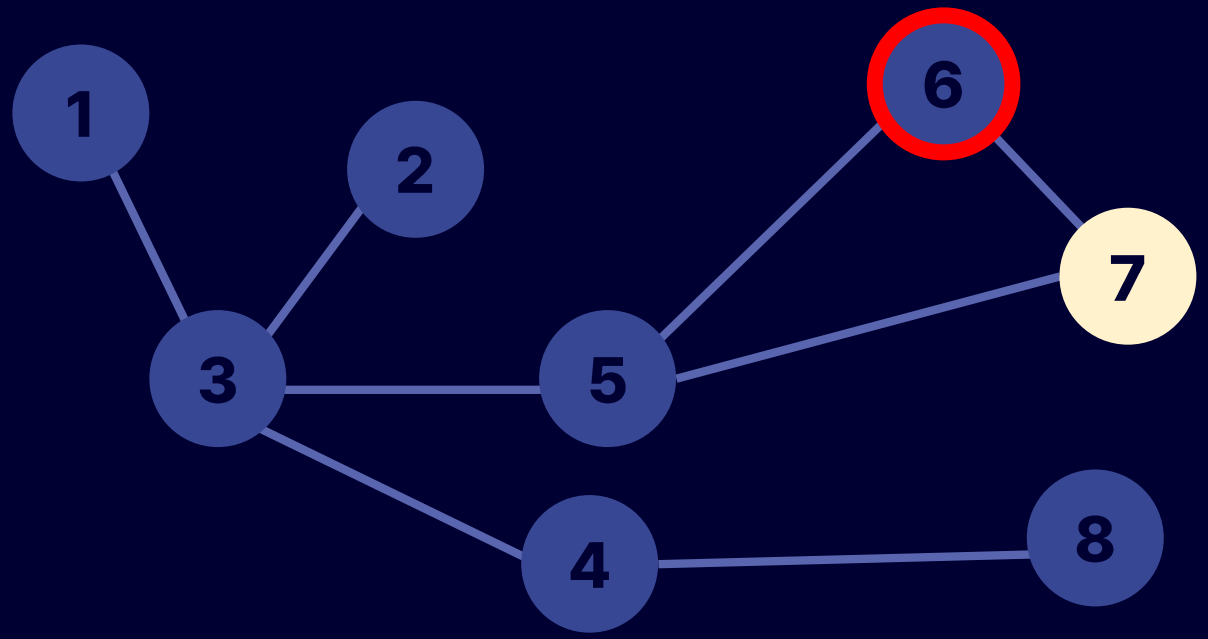


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 – 4 – 8 – 5 - 6

6번 정점을 방문했습니다. 인접한 정점 중 7번 정점만이 방문할 수 있는 정점입니다. 7번 정점을 방문합시다!

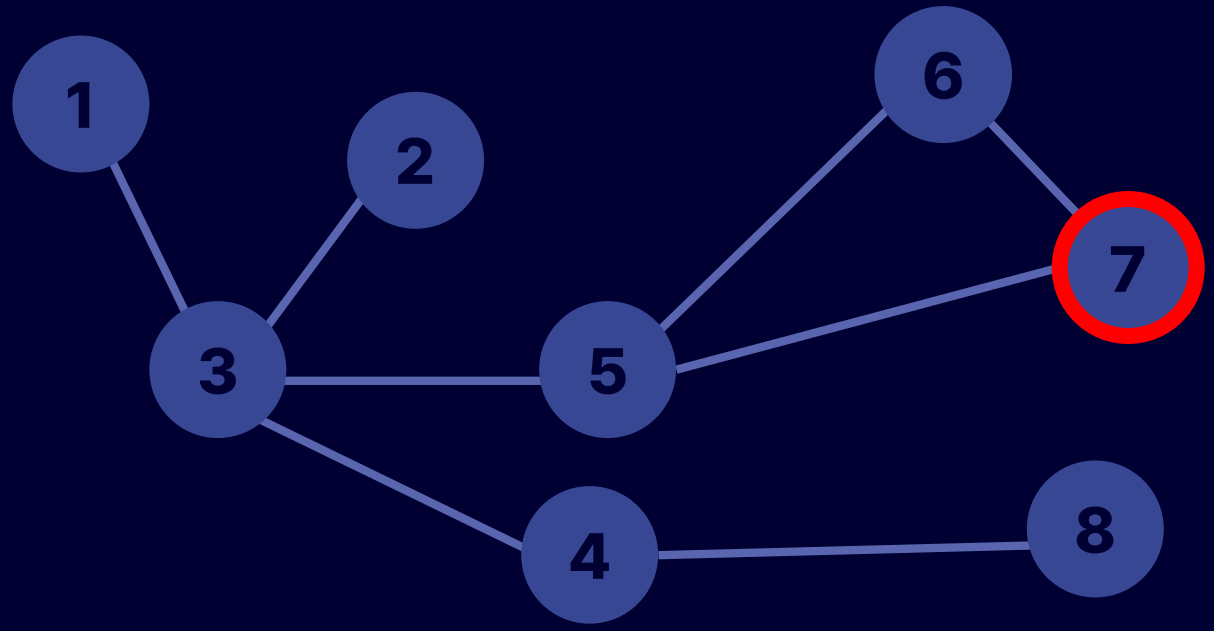


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 – 4 – 8 – 5 – 6 - 7

7번 정점을 방문했습니다. 이제 더 이상 방문할 수 있는 정점이 없습니다. 이전 단계로 돌아갑시다.

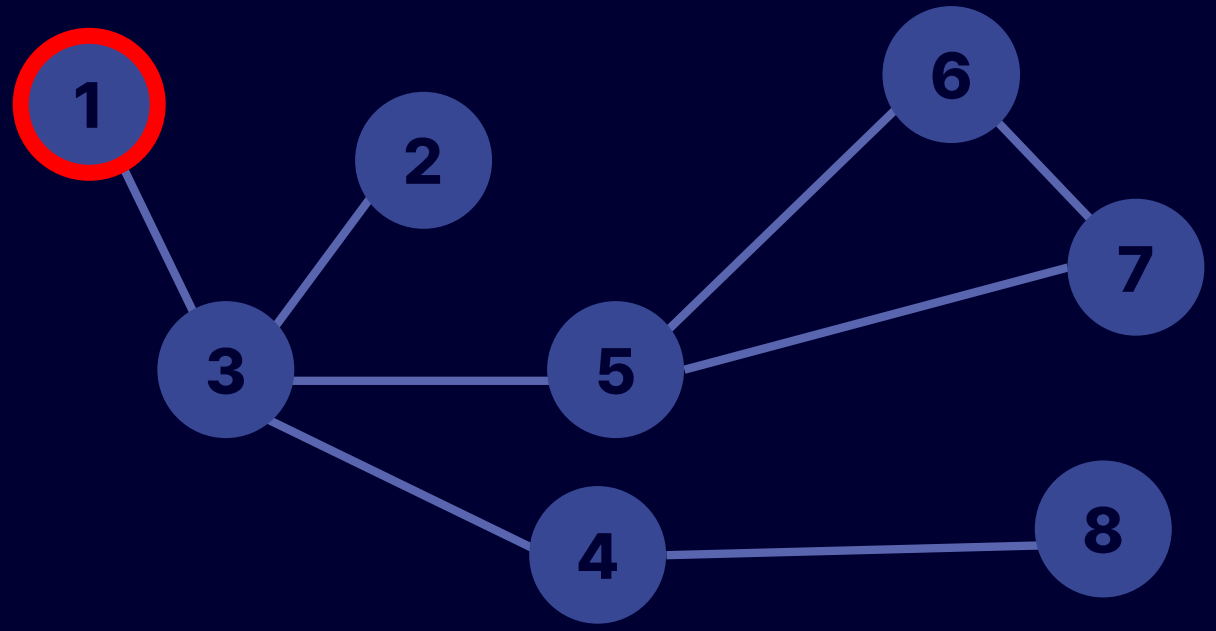


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 – 4 – 8 – 5 – 6 - 7

- 그렇게 7번 정점에서 6번 정점, 6번 정점에서 5번 정점, ... 이런 식으로 계속해서 이전 단계로 돌아가게 되며, 그렇게 1번 정점까지 돌아왔습니다.
- 1번 정점은 시작점이기에 이전 정점이 없습니다. 이렇게 되면 DFS는 끝이 나게 됩니다.

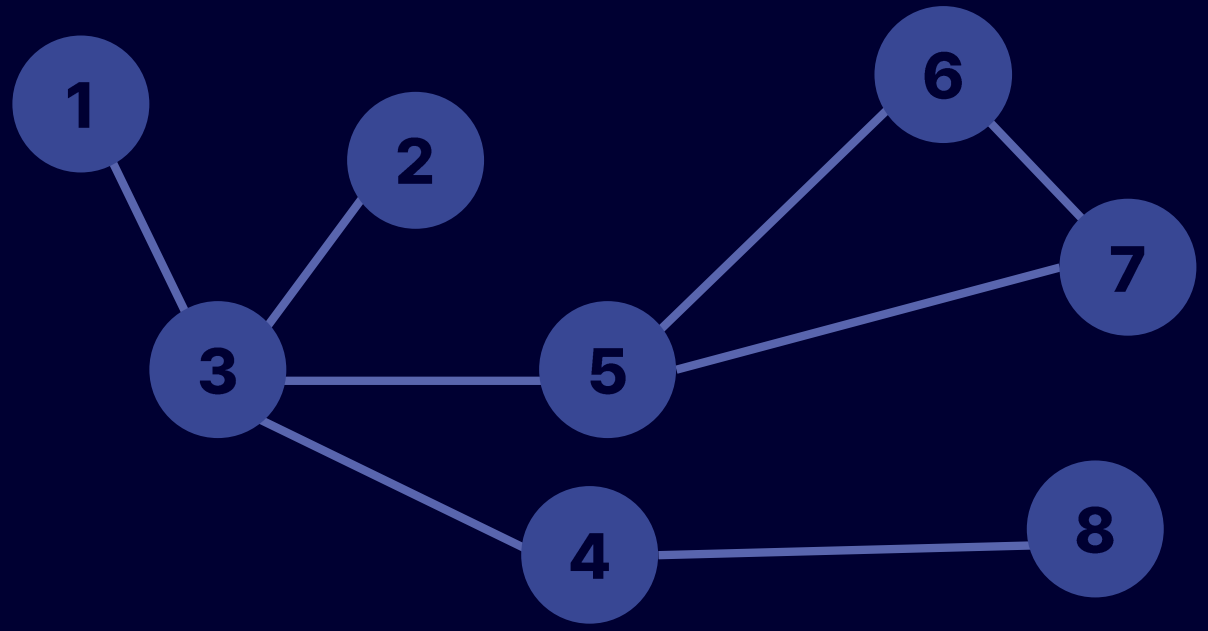


깊이 우선 탐색 (DFS)

미방문 정점 현재 정점 방문 정점

방문 순서: 1 – 3 – 2 – 4 – 8 – 5 – 6 - 7

- 방문 순서는 1 – 3 – 2 – 4 – 8 – 5 – 6 – 7 입니다. 방문 순서와 관련된 문제를 물어보는 문제도 있으므로 여러분은 그래프가 주어졌을 때 방문 순서가 어떨지도 아셔야 합니다. 코딩 테스트를 위해서도, 이론적인 학습을 위해서도요.
- DFS를 요약하자면 각 정점을 방문할 때 그 정점을 여러 번 방문하지 않도록 방문 처리하고, 인접한 정점들을 찾아 모두 방문해 보는 것입니다.



인접 행렬과 인접 리스트

- 이제부터 DFS를 본격적으로 코드로 구현해 보겠습니다만, 먼저 정점과 간선으로 이루어진 그래프를 코드상으로 표현하는 방법부터 알아야 합니다.
- 인접 행렬과 인접 리스트, 이 두 가지 방법을 이용하여 나타내 보겠습니다.

인접 행렬

- 인접 행렬은 2차원의 고정 크기 배열을 이용하여 그래프를 나타내는 방식입니다.
- 정점의 개수가 V라면 $V \times V$ 크기의 배열을 만드셔야 합니다. 각 칸은 boolean 타입의 값이며, 각 값의 의미는 이렇습니다:
  `graph[u][v]`: **u번 정점에서 v번 정점으로 이동할 수 있는 간선이 존재하는가?**
- 양방향 간선을 표현할 때에는 u번 정점에서 v번 정점으로 이동, v번 정점에서 u번 정점으로 이동하는 방법이 모두 가능하므로, 값도 두 개를 설정해 주셔야 함에 유의해 주세요!

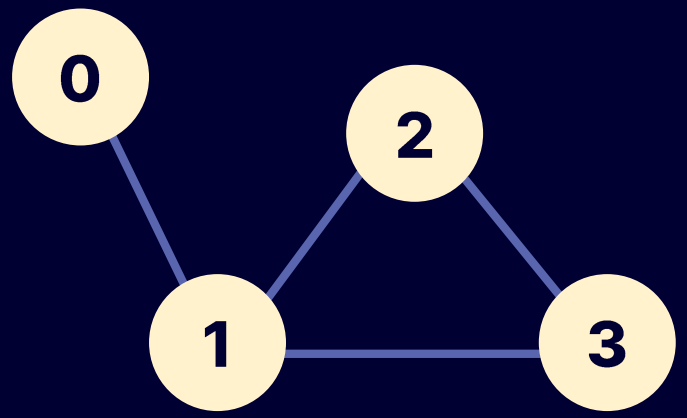
```
graph[u][v] = true
graph[v][u] = true
```

	0	1	2	3	…
0	true	false	true	true	
1	false	true	true	false	
2	false	false	true	false	
3	false	false	false	true	
…					

인접 리스트

- 인접 행렬은 2차원 리스트를 이용하여 그래프를 나타내는 방식입니다.
- 정점의 개수가 **V**라면 **V**개의 리스트를 만드셔야 합니다. 이후 **i**번째 리스트에는 **i**번 정점과 인접한 정점의 번호들을 저장해 주시면 됩니다.
  graph[i]: i번 정점과 인접한 정점의 번호들

```
graph = [
  [1],
  [0, 3, 2],
  [1, 3],
  [2, 1]
];
```



인접 리스트

- 눈치채셨겠지만 인접 리스트가 더 메모리를 적게 사용합니다. 그렇기에 정점의 수가 많을 때 (ex: 10만 개) 인접 리스트를 사용하는 것이 인접 행렬을 사용하는 것보다 유리합니다. 인접 리스트가 사용되는 경우가 훨씬 많다고 생각하시면 됩니다.
- 하지만 인접 리스트는 인접 행렬의 상위호환이 절대 아닙니다. 예를 들며 특정 정점을 연결하는 간선이 있는지를 알고 싶을 때 인접 행렬은 인접 리스트보다 훨씬 빠르고 간편하게 찾을 수 있습니다.
- 따라서, 여러분은 무조건 어느 한쪽만을 사용할 것이 아니라 문제와 부딪혀 보시면서 어느 쪽을 활용하여 구현할 지 스스로 판단하셔야 합니다.

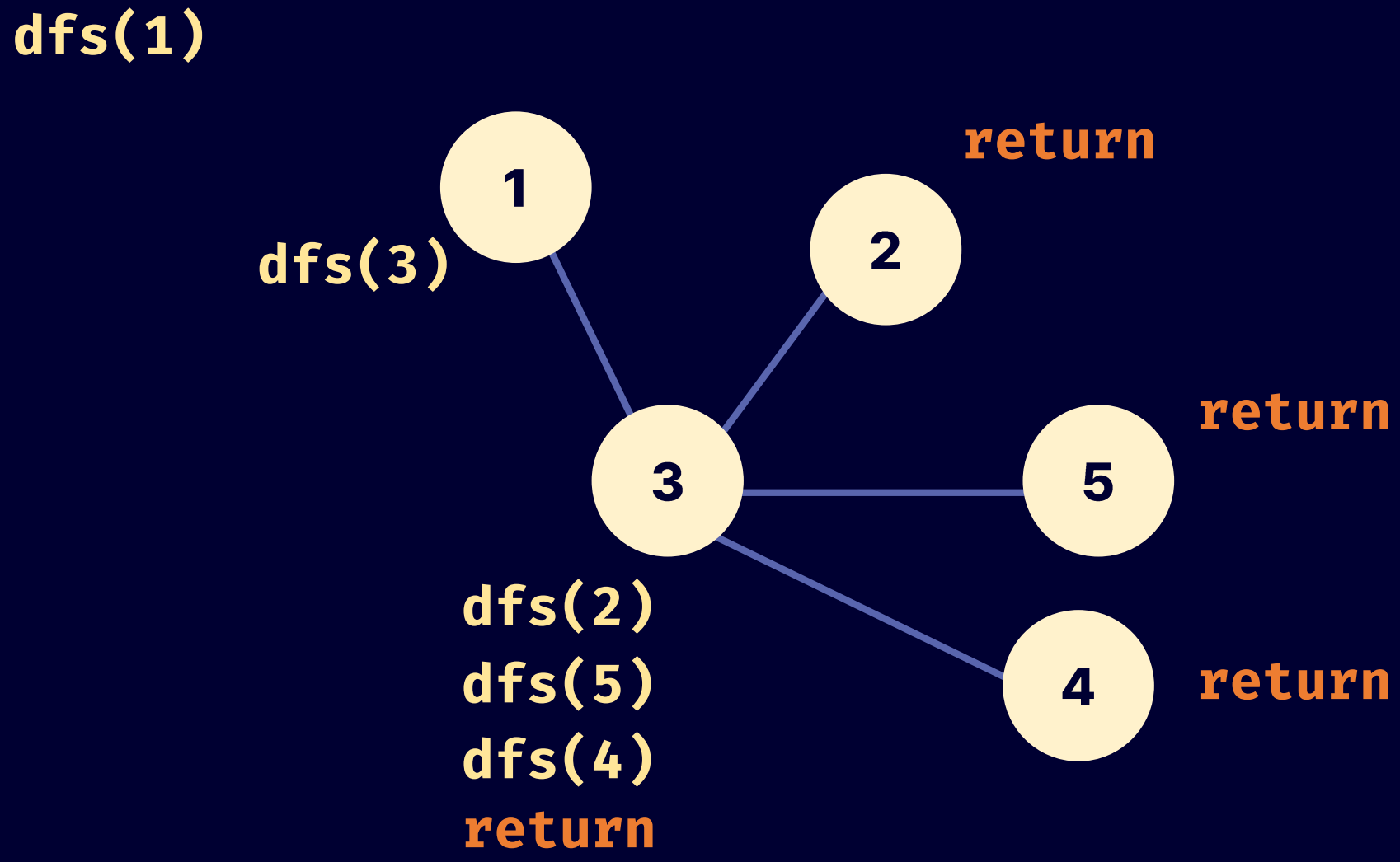
깊이 우선 탐색(DFS)의 구현

- 이제 여러분은 그래프 자료구조를 구현하실 수 있으시니, DFS를 본격적으로 구현해 보겠습니다.
- DFS는 재귀를 이용한 방법과 스택을 이용한 방법으로 나눌 수 있습니다.

깊이 우선 탐색(DFS)의 구현 - 재귀

- 재귀를 이용할 경우, **각 정점을 방문하는 작업** (방문 처리를 하고, 인접한 정점들을 차례로 방문)을 재귀함수로 나타내게 됩니다.
- 새로운 정점을 방문할 때 **`dfs(정점 번호)`** 와 같이 각 정점을 방문하도록 재귀함수를 호출하기만 하면 됩니다. 이 때 당연히 방문했는지의 여부를 검사한 후 호출해야 합니다. 당연히 dfs를 처음 시작할 때는 **`dfs(시작 정점)`**을 호출하게 됩니다.
- 더 이상 방문할 정점이 없을 경우에는 이전 단계로 되돌아간다고 설명하였습니다. 재귀를 사용한 방식이라면 그냥 그 정점에서 모든 작업을 마치고 return을 하는 것으로 자동으로 이전 단계로 되돌아갑니다. **새로운 정점을 방문할 때에 기존 정점을 다루는 재귀함수에서 새롭게 호출하기 때문입니다.**

깊이 우선 탐색(DFS)의 구현 - 재귀



깊이 우선 탐색(DFS)의 구현 - 재귀

```
const V = 10; // 정점의 개수

const visited = new Array(V).fill(false); // 방문 여부

const graph = [
  [],
  [3],
  [3],
  [1, 2, 4, 5],
  [3, 8],
  [3, 6, 7],
  [5, 7],
  [5, 6],
  [4],
];
```

```
const dfs = (vertexNo) => {
  visited[vertexNo] = true; // 방문 표시
  console.log(`Visit: ${vertexNo}`);

  for (let i = 0; i < graph[vertexNo].length; i++) { // 인접한 모든 정점 찾기
    const nextVertexNo = graph[vertexNo][i]; // 다음으로 방문할 정점의 번호
    if (!visited[nextVertexNo]) { // 방문을 안 했다면
      dfs(nextVertexNo); // 탐색 진행
    }
  }
};

dfs(1); // 1번 정점부터 방문 시작
```

stdout

```
Visit: 1
Visit: 3
Visit: 2
Visit: 4
Visit: 8
Visit: 5
Visit: 6
Visit: 7
```

DFS 설명 당시의 그래프의 탐색을 구현한 것입니다.

DFS에서 특정 방문을 방문한 이후 인접한 정점들을 방문할 때,
문제에서 명시하지 않은 한 어떤 정점을 먼저 방문해야 한다든지
그런 것은 없습니다.

따라서 올바른 DFS 코드더라도 방문 순서는 차이날 수 있습니다.

깊이 우선 탐색(DFS)의 구현 - 스택

- 스택을 이용할 경우, **반복문을 사용하며,** 현재 스택의 top에 있는 정점의 번호의 탐색을 진행하시면 됩니다.
- DFS를 처음 시작할 때는 `stack.push(시작 정점)`을 호출하는 것으로 시작하게 되며, 시작 정점에 대한 방문 처리를 진행합니다.
- 이후 반복문을 돌게 되는데, 아래의 과정을 반복하게 됩니다.
  - `stack.pop()`을 통해 스택의 top에 있는 정점의 번호를 얻습니다. 이 정점이 이번에 방문되는 정점입니다.
  - 재귀를 사용하는 방법과 마찬가지로 연결된 정점들을 돕니다. 이후 스택에 모두 push합니다.

깊이 우선 탐색(DFS)의 구현 - 스택

- 이해가 어려우시다면 스택을 일종의 대기열이라고 생각하셔도 좋습니다. 그러니까, 스택에 들어가는 정점들은 앞으로 방문할 정점들의 모음인 것이죠!

- 스택 자료구조의 특성상 마지막에 들어간 값일수록 일찍 나오기 때문에, 보다 깊이가 깊은 정점부터 방문하게 되는 효과가 있으며, 이로 인해 탐색을 깊숙하게 진행하는 DFS의 특성을 보이게 됩니다.

깊이 우선 탐색(DFS)의 구현 - 스택

```
const dfs = (startVertexNo) => {
  stack = [];
  stack.push(startVertexNo);
  visited[startVertexNo] = true;
  // 스택에 시작 정점을 스택에 넣음과 동시에 "방문 처리" 합니다
  // 여기서 당장 방문처리를 하지 않으면 이후에 다른 정점을 stack에 넣는 과정에서
  // 인접한 정점인 시작 정점이 또다시 방문되기 때문입니다

  while (stack.length > 0) {
    const currentVertexNo = stack.pop(); // 이번에 방문하게 될 정점입니다

    console.log(`Visit: ${currentVertexNo}`);

    for (let i = 0; i < graph[currentVertexNo].length; i++) {
      const nextVertexNo = graph[currentVertexNo][i]; // 다음으로 방문할 정점입니다

      if (!visited[nextVertexNo]) { // 아직 방문되지 않은 정점이라면
        visited[nextVertexNo] = true; // 방문 처리를 바로 하고
        stack.push(nextVertexNo); // 스택에 넣어줍니다
      }
    }
  }
};

dfs(1); // 1번 정점부터 방문 시작
// 와 저 스택으로 DFS 구현하는 건 이번이 두 번째인 것 같네요
```

깊이 우선 탐색(DFS)의 구현 - 스택

- 이것으로 깊이 우선 탐색(DFS)에 대한 모든 설명을 마칩니다.
- 인접 리스트를 사용했다고 가정했을 때 DFS의 시간복잡도는 정점의 개수가 $\mathbf{V}$개고 간선의 개수가 $\mathbf{E}$개일 경우 $\mathbf{O(V+E)}$ 입니다. 왜냐하면, 전체적인 탐색에 있어 총 $\mathbf{V}$ 번의 방문이 일어나고 각 정점마다 인접한 간선의 개수의 합은 $\mathbf{2E}$개이기 때문입니다. ($\mathbf{2E}$ 개라는 결론은 각 간선이 2개의 정점을 연결한다는 것을 눈치채신다면 이해하시기 수월하실 것입니다. 그리고 양방향 간선 기준입니다).
- 어느 방법을 사용하셔도 무방하지만, 재귀를 사용한 방법이 훨씬 직관적인 만큼 재귀를 이용한 구현 방법을 추천드립니다.
- 그 다음 페이지에서부터는 문제풀이를 다루지만, 개념 위주의 문제들이고, 개념은 앞서 설명드렸기에 제 설명은 굉장히 짧을 것입니다.

2 알고리즘 수업 - 깊이 우선 탐색 1

- 그래프가 주어지면 깊이 우선 탐색을 진행하고, 그 순서를 구할 수 있는지를 묻는 문제입니다.
- 개념을 설명드렸을 때와 똑같이 순서를 구해주시되, 다만 출력해야 하는 것이 깊이 우선 탐색의 순서가 아닌 **각 정점이 몇 번째로 방문되었는지이므로**, 약간 해야 하는 작업은 다를 것입니다. 이 점만 유의해 주시면 됩니다.
- 정답 배열을 따로 만들어두고, 각 정점이 방문될 때마다 그 정점의 답을 배열에 기록해 두시면 푸실 수 있을 것 같습니다.

3 바이러스

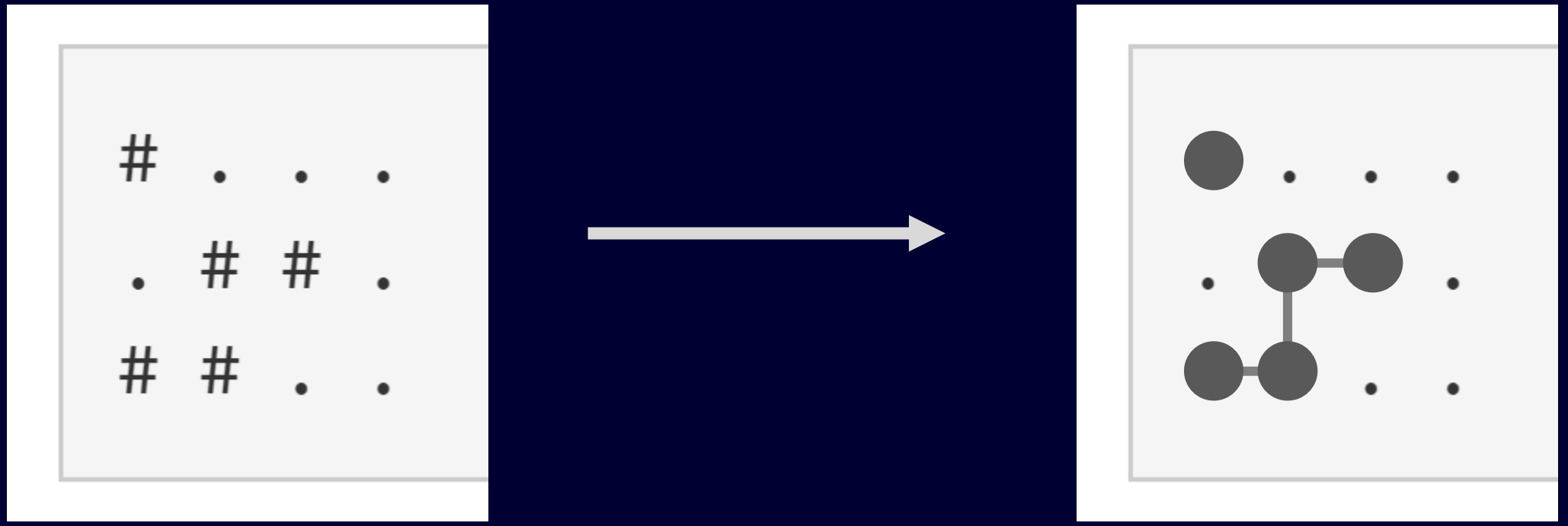
- 웜 바이러스에 걸리게 되는 컴퓨터의 수를 구해야 합니다. 이번에는 방문하게 되는 순서가 중요하지는 않지만, 그래프가 **몇 개의 정점으로 이루어져 있는지**를 알아야 합니다.
- **그래프 탐색을 거치면 연결된 정점들이 모두 방문**됨을 이용해 봅시다. 컴퓨터를 정점이라고 하고, 네트워크를 간선이라 했을 때, 1번 정점에서 탐색을 시작하고, 새로운 정점을 방문할 때마다 별도의 변수를 두고 1씩 방문 횟수를 더하는 것은 어떨까요?

1 음식물 피하기

- 바이러스 문제와 동일하게 그래프의 크기를 구하는 문제입니다. 문제는, 정점과 간선의 연결 관계를 친절하게 줬던 바이러스 문제와 달리 **2차원 격자 형태**로 음식물 쓰레기가 주어진다는 것입니다.
- 이 문제처럼 미로 찾기, 게임판 문제 등의 주제로 격자 형태로 데이터를 만들어야 하고, 이러한 형태의 데이터를 그래프 탐색하는 유형이 코딩 테스트, 알고리즘 대회 등에서 상당히 많이 나옵니다. **격자 형태로 주어지는 그래프**에서 탐색을 할 수 있는지를 묻는 문제입니다.

1 음식물 피하기

- 격자 형태의 데이터도 결국은 그래프로 치환할 수 있음을 잊지 마세요. 아래의 형태라면 주어진 문제를 그래프로 치환할 수 있지 않을까요?



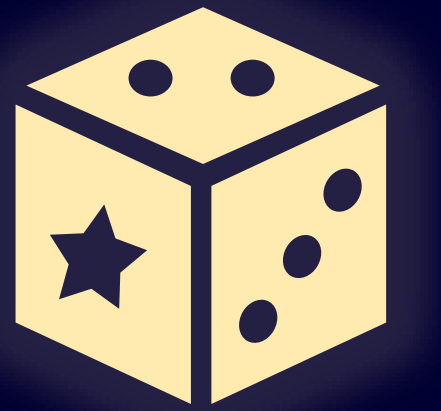
1 음식물 피하기

- 음식물 쓰레기는 어디에나 있을 수 있고, 얼마나 이어져 있을 지 알 수 없기 때문에 시작 정점을 하나로 잡을 수 없습니다.
- 격자의 각 칸을 정점으로 생각했을 때, 우리는 모든 정점에서 한 번씩 dfs를 돌려 보아야 합니다. 즉, 아래와 같이요.
- 각 칸에 있는 것이 음식물 쓰레기이고, 방문되지 않은 정점이라면 지금까지 우리가 탐색해 본 다른 음식물 쓰레기에 포함되지 않는 새로운 음식물 쓰레기이므로 dfs를 진행해 주어야 합니다.

```
for (let r = 1; r <= R; r++) {
  for (let c = 1; c <= C; c++) {
    if (!visited[r][c] && grid[r][c] === '#') {
      dfs(r, c);
    }
  }
}
```

1 음식물 피하기

- 새로운 음식물 쓰레기 조각(정점)을 방문했을 때는 **상하좌우를 확인하여 아직 방문되지 않은 음식물 쓰레기 조각이 있는지 확인**해 주시면 됩니다.
- 상하좌우를 확인하실 때는 **격자의 범위 바깥으로 갈 수 있음**에 유의하여 코드를 작성해 주세요.
- 다음 주차에 비슷한 문제를 한 문제 더 다룰테니, 지금 막히더라도 너무 걱정하지 마세요. 구현과 관련해서 궁금한 점이 있으시면 언제든 슬랙에 질문해 주세요!



The End