



D.COM ALGORITHM STUDY

5강. 그-래 프

오늘의 목표

자료구조 중 하나인
그래프의 용어와 종류 그리고
구현하는 방법에 대해 알아보시다!

#관련 문제도 풀어봅시다!

가볍게,
읽을거리



링크

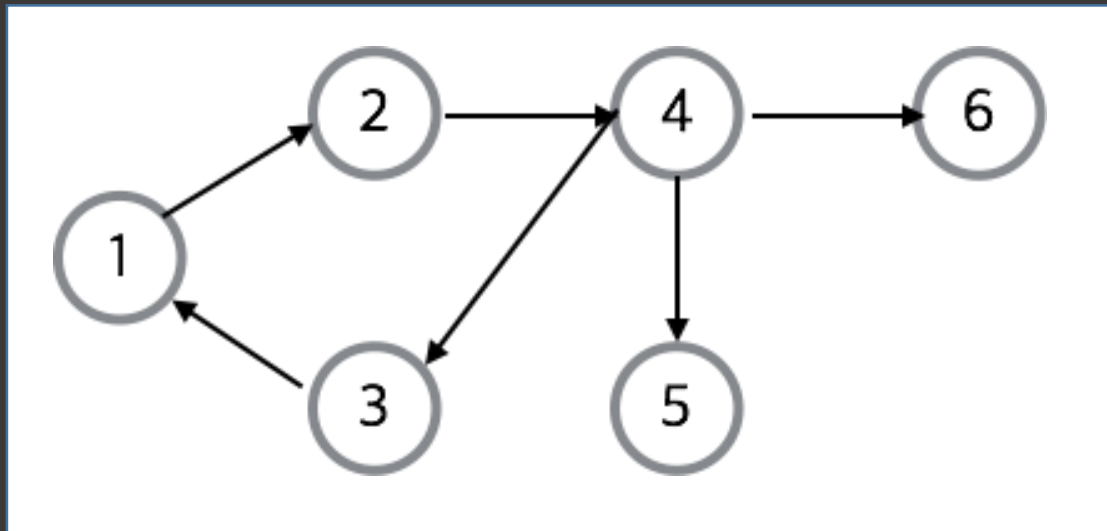
좋은 코딩을 위한 13가지 간단한 규칙[번역]

“간단하고 단순하게”

그래프

Graph

I 그래프

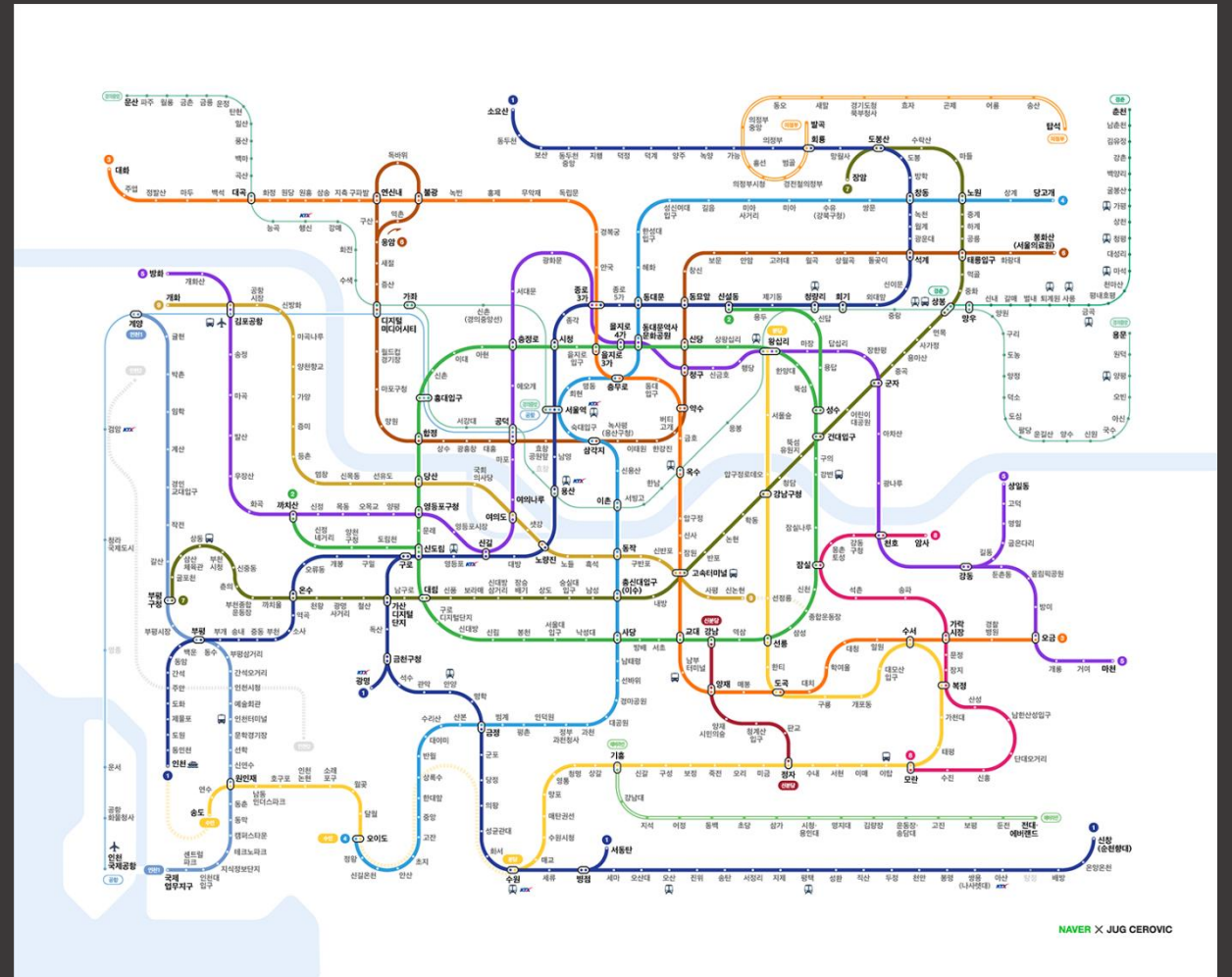


정점(Node, Vertex)와 그 노드를 연결하는 간선(Edge)를 하나로 모아 놓은 자료 구조(Data Structure)입니다.

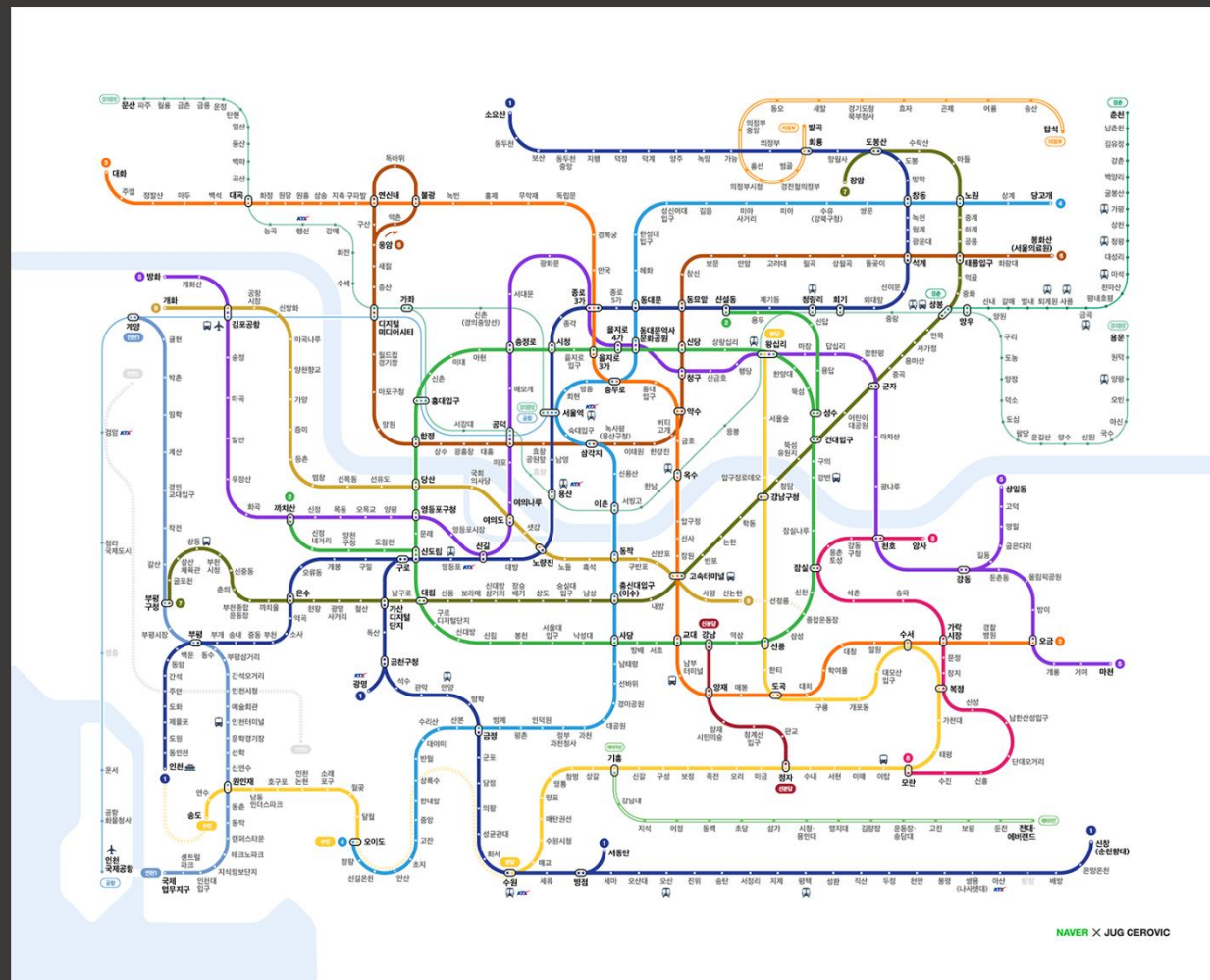
지하철 노선도도

그래프의 한 종류입니다.

전 슬라이드의 그래프 정의를
다시 생각해봅시다!



각 지하철 역이
정점(Node, Vertex)이며
이들을 연결하는 노선,
즉 **간선(Edge)**간의
관계로 그래프를 표현할 수 있습니다.



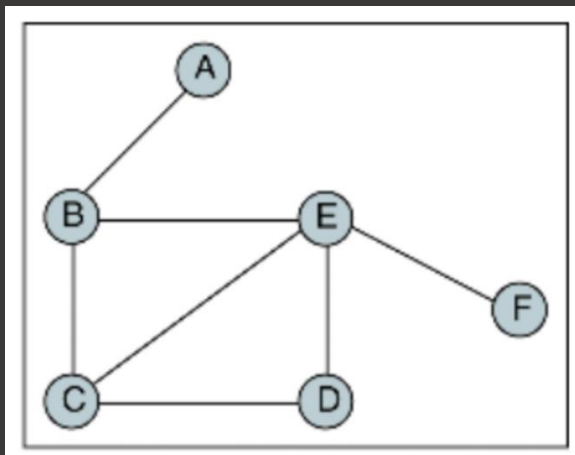
그래프에는 여러가지 **종류**가 있습니다.

—

방향이 있는가?
가중치가 있는가?
간선의 개수는 어떠한가?
등등

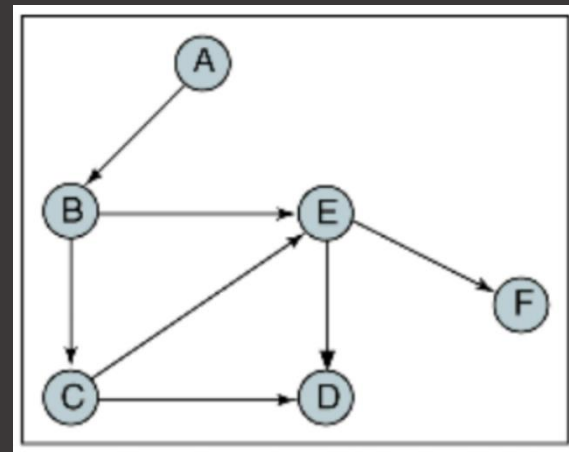
방향이 있는가?

—



무방향 그래프
Undirected Graph

두 정점을 연결하는 간선에 방향이 없습니다.
따라서, 정점 v_1 과 v_2 를 연결하는 간선을 $[v_1, v_2]$ 라 하면
 $[v_1, v_2]$ $[v_2, v_1]$ 은 같은 간선입니다.

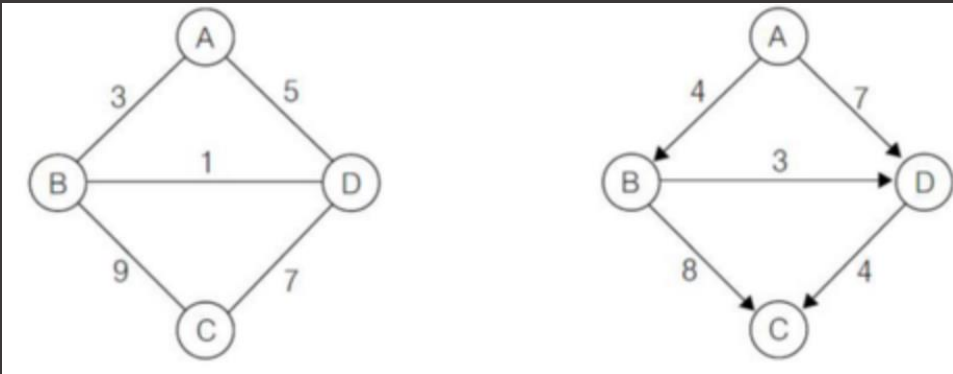


방향 그래프
Directed Graph

두 정점을 연결하는 간선에 방향이 있습니다.
따라서, $[v_1, v_2]$ $[v_2, v_1]$ 는 다른 간선입니다.

가중치가 있는가?

—

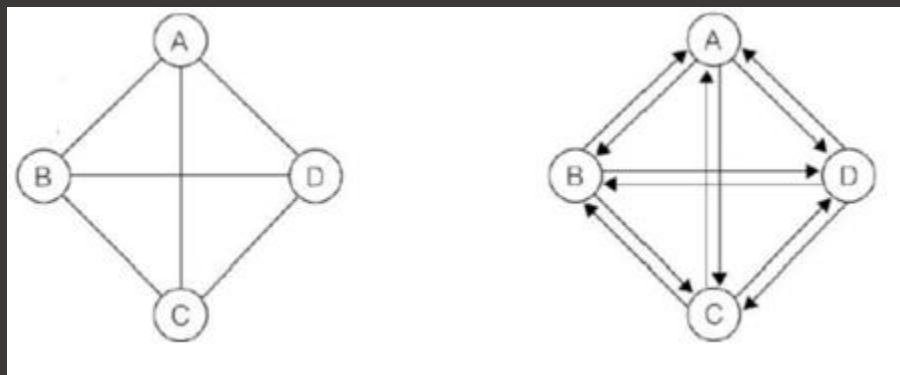


가중 그래프
Weighted Graph

간선에 가중치가 있는 그래프입니다.

가중치란 정점 이동 간에 드는 **비용**이라
생각하시면 됩니다!

간선의 개수는 어떠한가?



완전 그래프
Complete Graph

한 정점에서 모든 다른 정점과 연결되어
최대의 간선수를 가지는 그래프입니다.

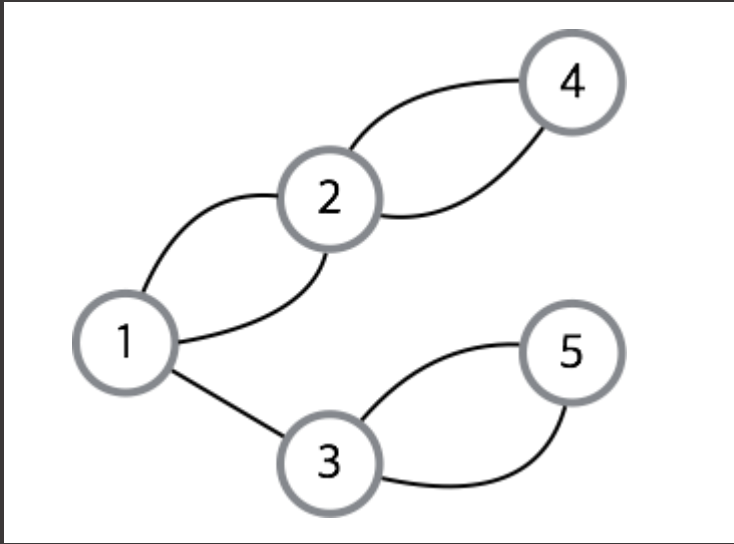
TIP!

무방향 그래프 최대 간선 수: $n(n-1)/2$

방향 그래프 최대 간선 수 : $n(n-1)$

간선의 개수는 어떠한가?

—



다중 그래프

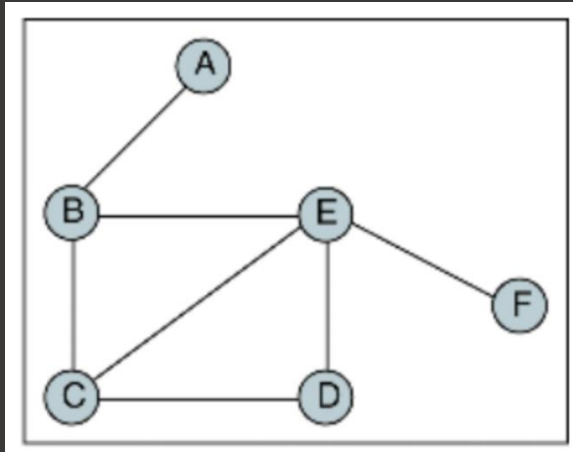
한 개의 정점에서 두 개 이상의
간선을 가질 수도 있습니다.
이때의 간선들은
서로 다른 간선입니다.

또한 간선의 양 끝점이
같은 루프(Loop)도 있습니다.

그래프에 쓰이는 여러가지 용어를 알아봅시다.

경로
사이클
단순 경로
단순 사이클
차수
가중치
루프
...

경로 Path

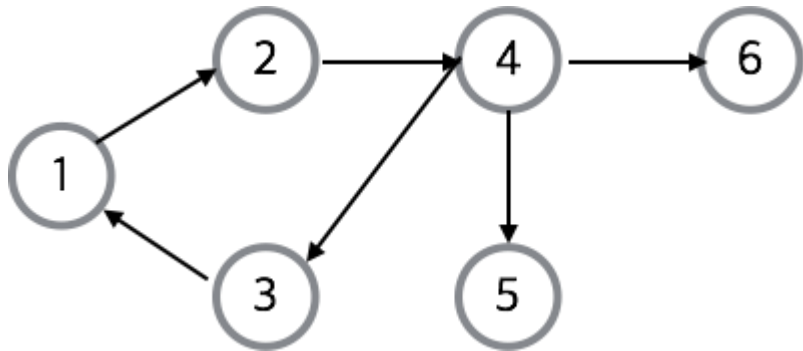


정점 A에서 정점 B까지 간선을 따라
갈 수 있는 길을 순서대로 나열한 것을 의미합니다.

예를 들어 왼쪽 그래프에서
A에서 E로 가는 경로(Path)는 두 가지가 있습니다.

A → B → E
A → B → C → E

사이클 Cycle



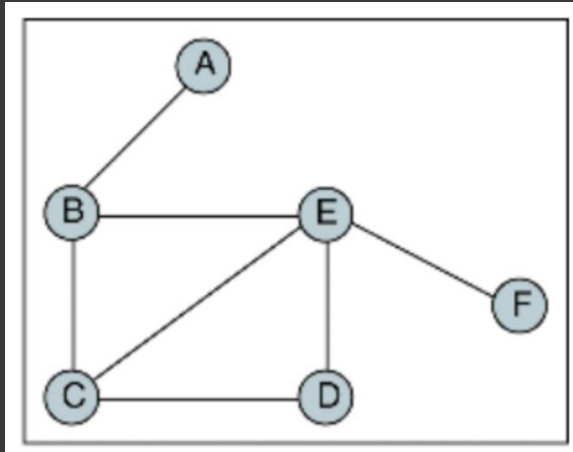
정점 A에서 다시 정점 A으로
돌아오는 경로를 의미합니다

예를 들어 왼쪽 그래프에서
1번 정점의 사이클은 다음과 같습니다.

1 → 2 → 4 → 3 → 1

차수

Degree

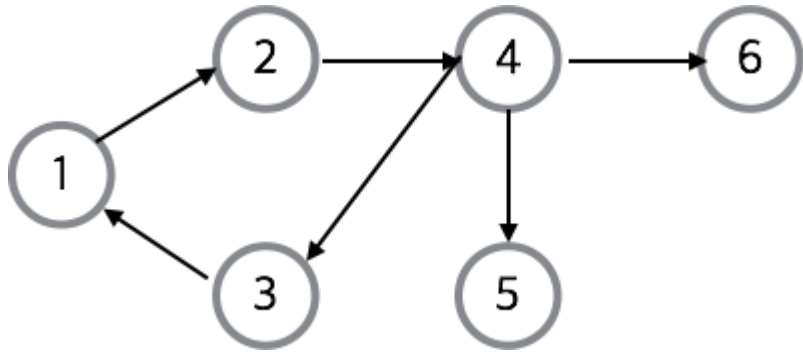


정점과 연결되어 있는
간선의 개수를 의미합니다.

예를 들어 왼쪽 그래프에서
E의 차수는 4입니다.

차수

Degree



방향 그래프의 경우

진입 차수(In-degree), 진출 차수(Out-degree)로
나누어 차수를 계산합니다.

예를 들어 왼쪽 그래프에서

4의 진입 차수는 1이고
진출 차수는 2입니다.

| 이제 그래프를 어떻게 **표현**하고
구현할 수 있을까요?

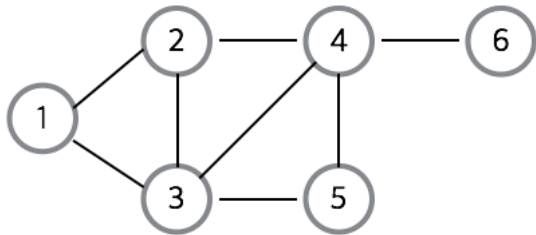
—
세가지 방법이 있습니다.

인접 행렬(Adjacent Matrix)

인접 리스트(Adjacent List)

간선 리스트(Edge List)

인접 행렬(Adjacent Matrix)



Ad	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	0	0
3	1	1	0	1	1	0
4	0	1	1	0	1	1
5	0	0	1	1	0	0
6	0	0	0	1	0	0

$$A[1][1] = 0$$

$$A[1][2] = 1$$

$$A[1][3] = 1$$

$$A[1][4] = 0$$

.

.

.

정점의 개수를 V 라고 했을 때,
 $V \times V$ 크기의 이차원 배열을 이용하여 표현합니다.

$$A[i][j] = 1 \text{ (} i \text{ 에서 } j \text{로 가는 간선이 있을 때), } 0 \text{ (없을 때)}$$

#만약 가중치 그래프라면 1대신 가중치 값을 넣어주면 됩니다!

#무방향 그래프라면 $A[i][j] = A[j][i]$ 입니다.

인접 행렬(Adjacent Matrix)



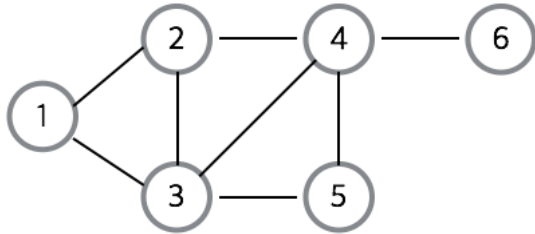
앞서 배운 개념을 어떻게 코드로 구현 할 수 있을까요?
만약 첫 줄에 정점의 개수 N, 간선의 개수 M
다음 M개의 줄에 간선이 연결하는 두 정점의 번호가
주어진다고 해봅시다.

인접 행렬은 이차원 배열을 이용하여 간단하게
구현할 수 있습니다.

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

```
bool a[100][100];
int n,m;
cin >> n >> m;
for(int i = 0; i < m; i++){
    cin >> from >> to;
    a[from][to] = a[to][from] = true;
}
```

인접 리스트(Adjacent List)



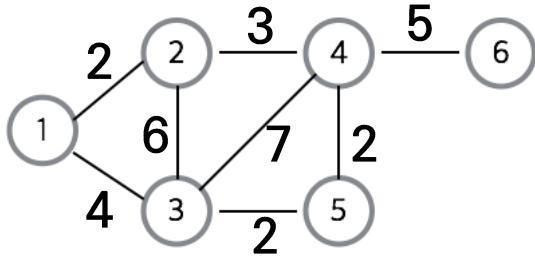
A[1]	2 3
A[2]	1 3 4
A[3]	1 2 4 5
A[4]	2 3 5 6
A[5]	3 4
A[6]	4

리스트를 이용하여 표현합니다.

A[i]에 i와 연결된 정점을 값을 넣습니다.

C++에서는 벡터 STL를 사용하여 구현하면 편리합니다!

인접 리스트(Adjacent List)



A[1] (2,2) (3,4)
A[2] (1,2) (3,6) (4,3)
A[3] (1,4) (2,6) (4,7) (5,2)
A[4] (2,3) (3,7) (5,2) (6,5)
A[5] (3,2) (4,2)
A[6] (4,5)

만약 **가중치 그래프**라면 다음과 같이 구현합니다.
C++에서 벡터 내 요소는
Pair STL을 사용하여 구현하면 편리합니다.

인접 리스트(Adjacent List)

인접 리스트는 Vector를 이용하면
간단하게 구현할 수 있습니다.

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



```
vector<int> a[100]
int n,m;
cin >> n >> m;
for(int i = 0; i < m; i++){
    cin >> from >> to;
    a[from].push_back(to);
    a[to].push_back(from);
}
```

인접 행렬(Adjacent Matrix) vs 인접 리스트(Adjacent List)



우선, 인접 행렬의 공간 복잡도는 간선의 개수와 무관하게 $O(V^2)$ 입니다.

다음으로, 인접 리스트의 공간 복잡도는 $O(V+E)$ (정점의 개수 + 간선의 개수)입니다.

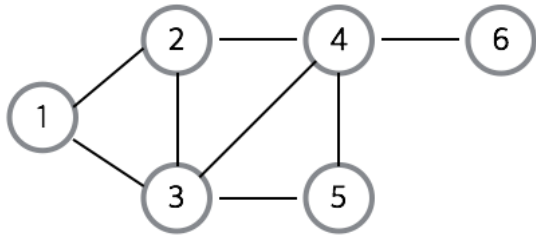
인접 행렬의 경우 공간 복잡도 측면에서 비효율적인 것을 알 수 있습니다.

하지만, 정점 u 와 v 가 주어졌을 때 단 한번의 배열의 접근으로만 연결 여부를 알 수 있습니다.

인접 리스트의 경우에는 연결 여부를 알기 위해 $ad[u]$ 의 첫 요소부터 각 요소를 일일이 검사해야 합니다.

따라서, 간선의 수가 적은 희소 그래프에서는 인접 리스트를 사용하는 것이 유리하고,
간선의 수가 V^2 에 수렴하는 밀집 그래프에서는 인접 행렬을 사용하는 것이 유리합니다.

간선 리스트(Edge List)



E[0] = 1 2

E[1] = 1 3

E[2] = 2 1

E[3] = 2 3

E[4] = 2 4

E[5] = 3 1

E[6] = 3 2

E[7] = 3 4

E[8] = 3 5

E[9] = 4 2

E[10] = 4 3

E[11] = 4 5

E[12] = 4 6

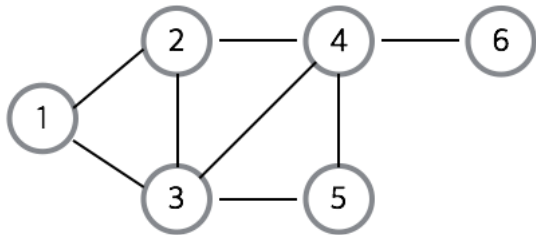
E[13] = 5 3

E[14] = 5 4

E[15] = 6 4

간선 각각을 저장하고 있는 배열을
통해 구현합니다.

간선 리스트(Edge List)



E[0] = 1 2

E[1] = 1 3

E[2] = 2 1

E[3] = 2 3

E[4] = 2 4

E[5] = 3 1

E[6] = 3 2

E[7] = 3 4

E[8] = 3 5

E[9] = 4 2

E[10] = 4 3

E[11] = 4 5

E[12] = 4 6

E[13] = 5 3

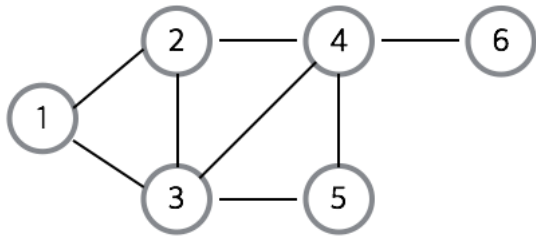
E[14] = 5 4

E[15] = 6 4

여기서 i 정점과 연결된 간선을 찾기 위해서
E 배열의 처음부터 끝까지 탐색해야 할까요?

바로 한번에 찾을 수 있는 방법이 있습니다!

간선 리스트(Edge List)



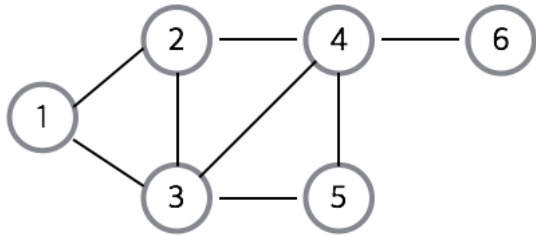
E[0] = 1 2	E[8] = 3 5
E[1] = 1 3	E[9] = 4 2
E[2] = 2 1	E[10] = 4 3
E[3] = 2 3	E[11] = 4 5
E[4] = 2 4	E[12] = 4 6
E[5] = 3 1	E[13] = 5 3
E[6] = 3 2	E[14] = 5 4
E[7] = 3 4	E[15] = 6 4

이를 위해서 `cnt[i]`라는 배열을 만들어봅시다!
우선 각 간선의 앞 정점을 기준으로 개수를 세서
`cnt` 배열에 저장합니다.
(`m`은 E배열의 크기입니다.)

```
for (int i = 0; i < m; i++) {  
    cnt[E[i][0]] += 1;  
}
```

i	0	1	2	3	4	5	6
Cnt[i]	0	2	3	4	4	2	1

간선 리스트(Edge List)



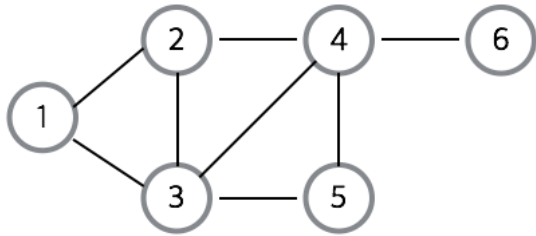
E[0] = 1 2	E[8] = 3 5
E[1] = 1 3	E[9] = 4 2
E[2] = 2 1	E[10] = 4 3
E[3] = 2 3	E[11] = 4 5
E[4] = 2 4	E[12] = 4 6
E[5] = 3 1	E[13] = 5 3
E[6] = 3 2	E[14] = 5 4
E[7] = 3 4	E[15] = 6 4

그리고 $i-1$ 정점의 개수와
 i 정점의 개수를 더합니다.
(n 은 cnt배열의 크기입니다.)

```
for (int i = 1; i <= n; i++) {  
    cnt[i] = cnt[i] + cnt[i-1];  
}
```

i	0	1	2	3	4	5	6
Cnt[i]	0	2	5	9	13	15	16

간선 리스트(Edge List)



E[0] = 1 2

E[1] = 1 3

E[2] = 2 1

E[3] = 2 3

E[4] = 2 4

E[5] = 3 1

E[6] = 3 2

E[7] = 3 4

E[8] = 3 5

E[9] = 4 2

E[10] = 4 3

E[11] = 4 5

E[12] = 4 6

E[13] = 5 3

E[14] = 5 4

E[15] = 6 4

그럼 이제 i 정점과 연결된 간선은
 E 배열에서 $\text{cnt}[i-1] \sim \text{cnt}[i] - 1$ 까지입니다.

예를 들어 4번 정점의 간선은
 $\text{cnt}[3] \sim \text{cnt}[4] - 1$ 이므로, 9 ~ 13 - 1
즉, $E[9] \sim E[12]$ 임을 알 수 있습니다.

실제로, 왼쪽 표에서도 $E[9] \sim E[13]$ 이
4번 정점의 간선을 나타냄을 알 수 있습니다.

i	0	1	2	3	4	5	6
Cnt[i]	0	2	5	9	13	15	16

간선 리스트(Edge List)



간선 리스트의 공간 복잡도는 $O(E)$ 입니다.

그래프를 구현할 때 주로 인접 리스트나 인접 행렬을 주로 사용하고,
간선 리스트를 잘 사용하지는 않습니다.

그럼에도 배운 이유는 나중에 배우게 될 벨만-포드 알고리즘이나
크루스칼 알고리즘 등의 일부 알고리즘을 구현할 때 필요하기 때문입니다.

따라서, 세가지 그래프 구현 방법의 개념을 모두 알고 있도록 합시다!



연습 문제
13023번 ABCDE

<https://www.acmicpc.net/problem/13023>

앞서 배운 개념들을 적절히 활용해보시다!
TBA



연습 문제

13023번 ABCDE

<https://www.acmicpc.net/problem/13023>

위 문제는 정말 여러가지 방법으로 풀 수 있습니다!

HINT

A와 B, C와 D가 연결 되어있는지 어떻게 알 수 있을까요?

그리고 B와 C가 연결되어 있나요?

마지막으로 D와 E가 연결되어 있나요?

■ 그래프의 **탐색**을 알아봅시다.



이번 단원의 하이라이트입니다!

탐색이란 하나의 정점으로부터 시작하여 차례대로 모든 정점을 한 번씩 방문하는 것입니다.



그래프를 **탐색**하는 대표적인 방법에는
두 가지가 있습니다!

BFS (Breadth-First Search) 너비 우선 탐색

DFS (Depth-First Search) 깊이 우선 탐색

DFS (Depth-First Search)

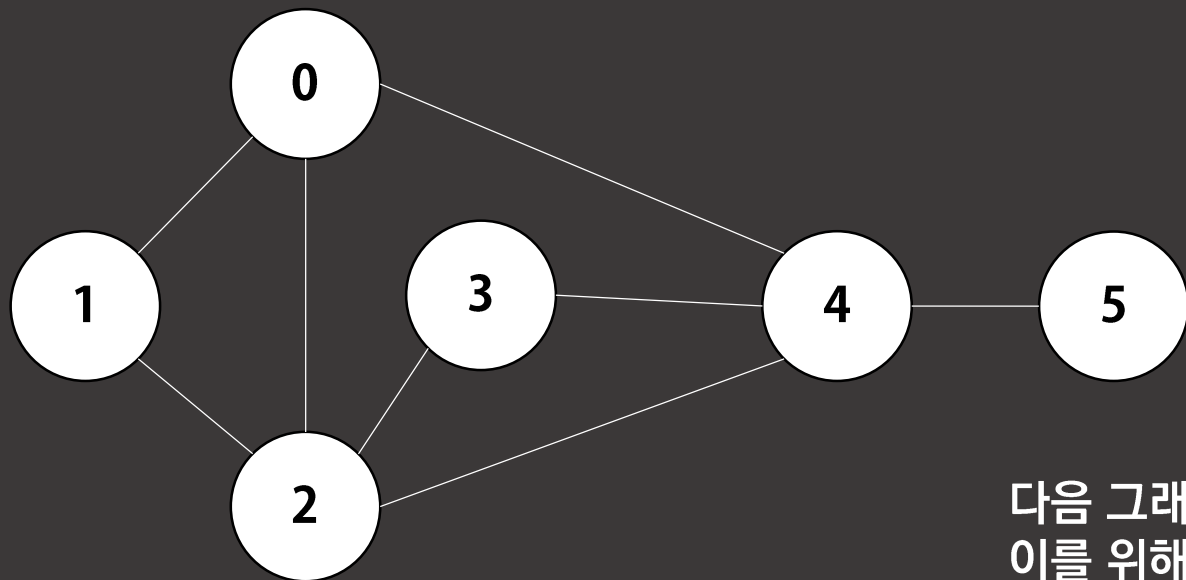
깊이 우선 탐색



이름 그대로 깊이를 우선하여 탐색하는 것을 의미합니다(?)
갈 수 있는 만큼 최대한 많이 가고, 갈 수 없으면 이전 정점으로 돌아갑니다.

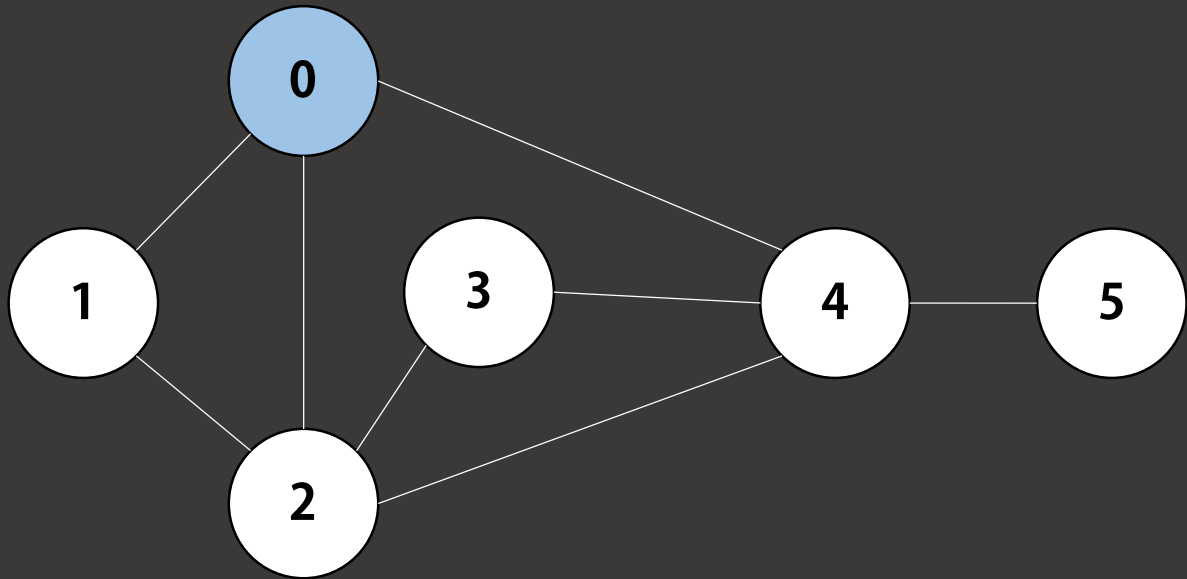
이 과정에서 **스택**을 이용합니다.

DFS



다음 그래프를 DFS로 탐색해봅시다!
이를 위해 정점의 방문 여부를 체크하는
check[i] 배열과 dfs를 위한 스택 s가 필요합니다!

DFS

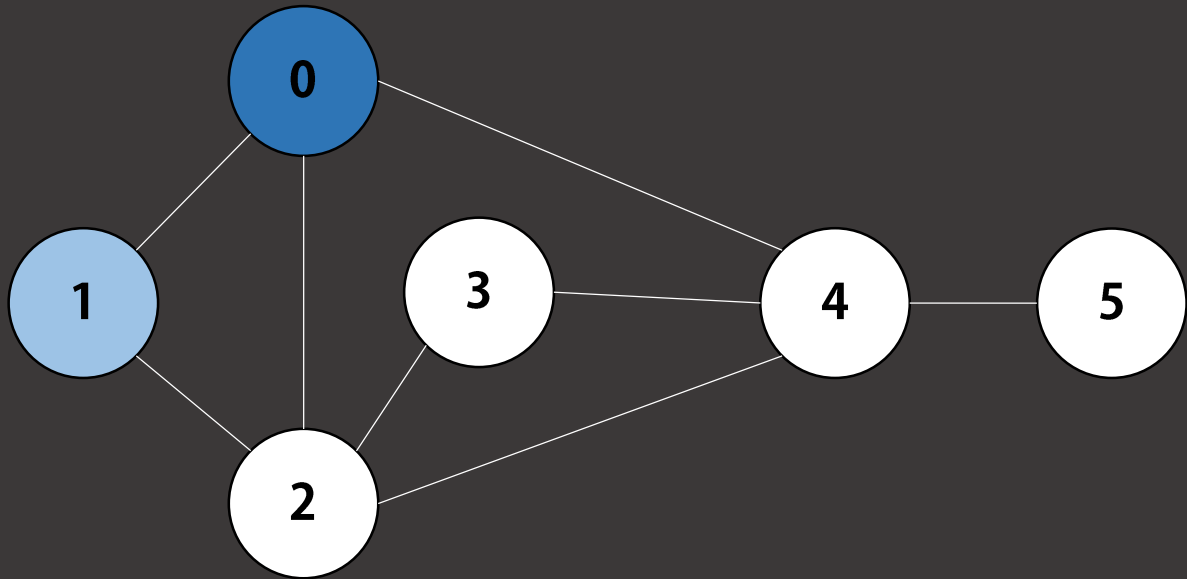


0부터 탐색을 시작하겠습니다!
현재 탐색하고 있는 정점이 0이므로
check[0] = 1로 방문 여부를 체크하고,
스택 S에 0을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	0	0	0	0	0

스택 S: 0

DFS

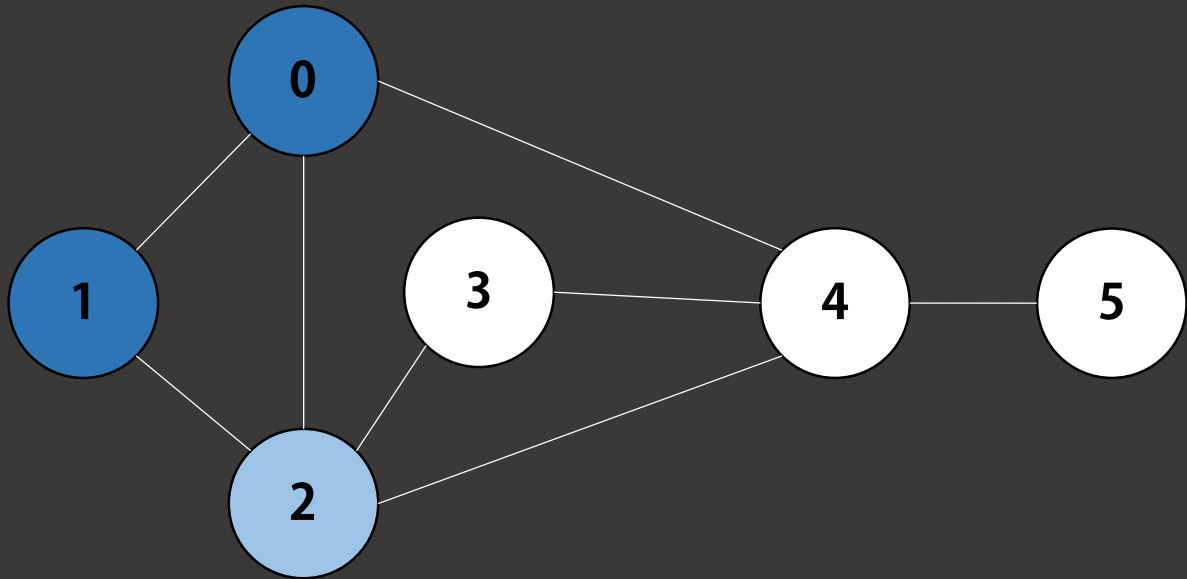


그리고 인접한 정점인 1로 이동합니다.
마찬가지로 check[1]에 기록하고
스택 S에 1을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	0	0	0	0

스택 S: 0 1

DFS

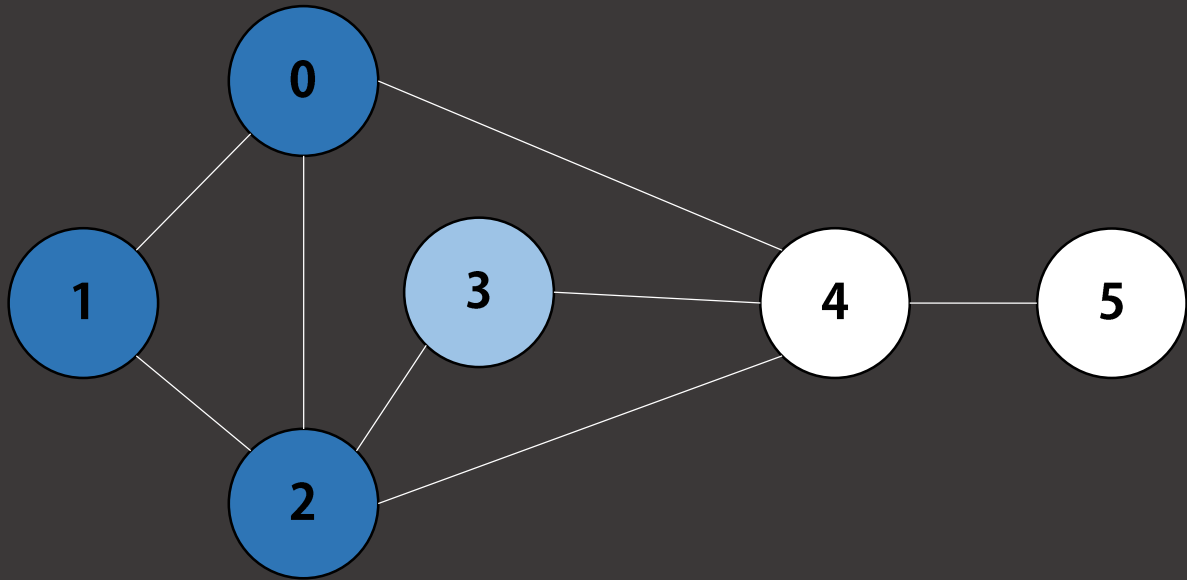


그리고 인접한 정점인 2로 이동합니다.
check[2]에 기록하고
스택 S에 2을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	0	0	0

스택 S: 0 1 2

DFS

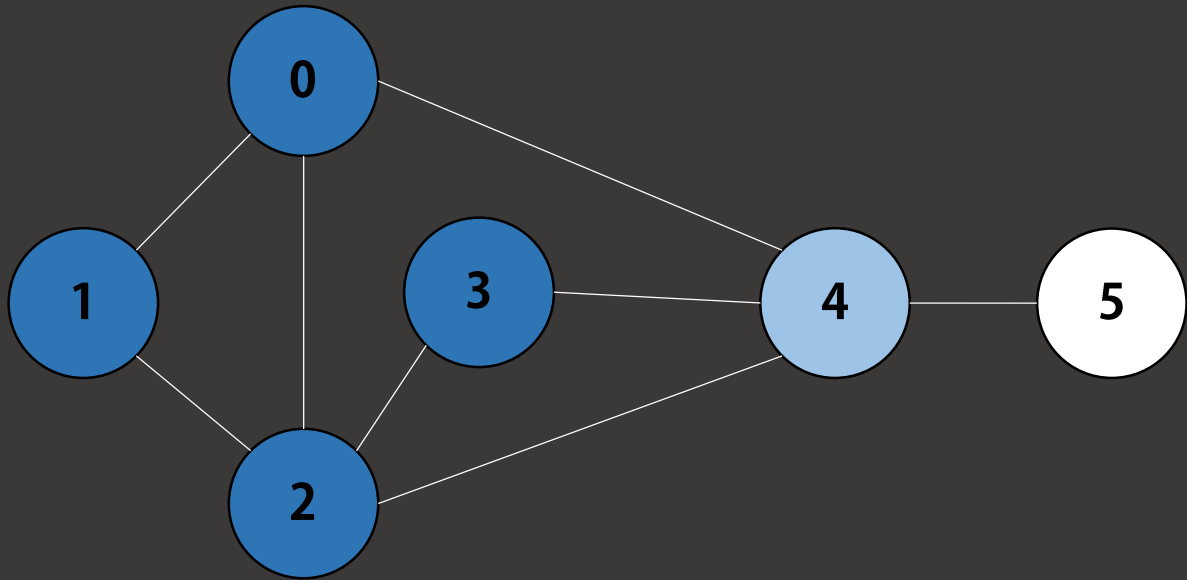


그리고 인접한 정점인 3로 이동합니다.
check[3]에 기록하고
스택 S에 3을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	0	0

스택 S: 0 1 2 3

DFS

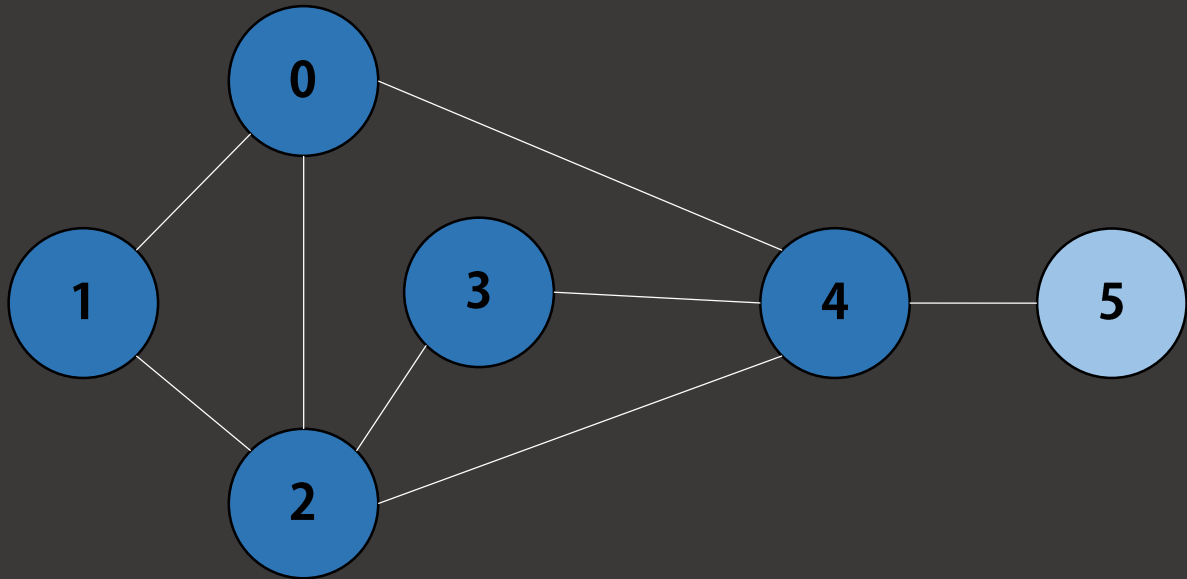


그리고 인접한 정점인 4로 이동합니다.
check[4]에 기록하고
스택 S에 4을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	0

스택 S: 0 1 2 3 4

DFS

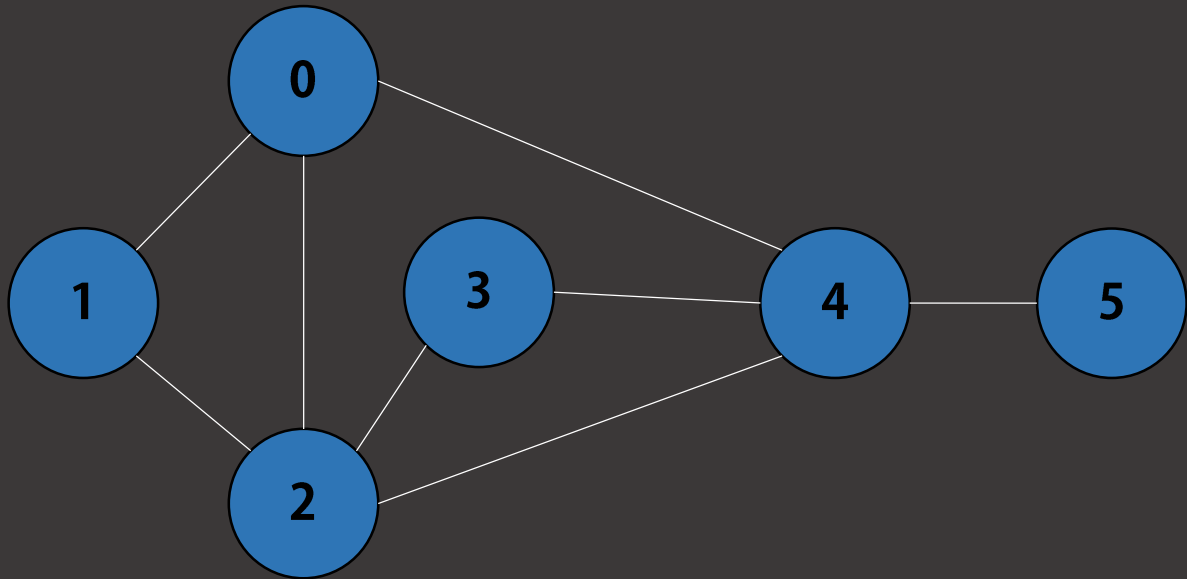


그리고 인접한 정점인 5로 이동합니다.
check[5]에 기록하고
스택 S에 5을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	1

스택 S: 0 1 2 3 4 5

DFS

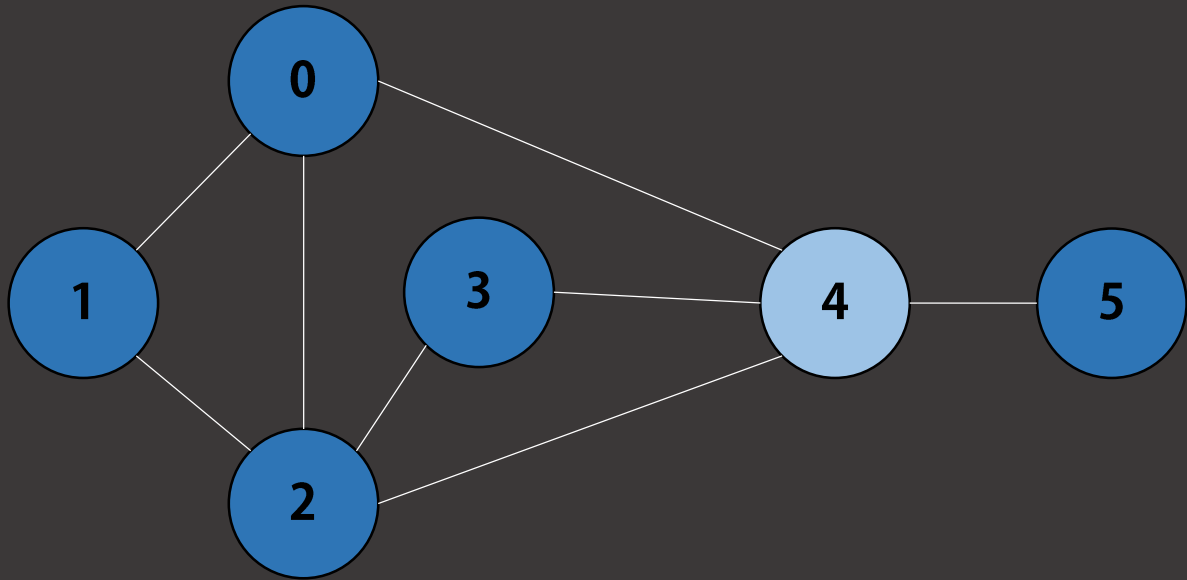


정점 5에 인접한 방문하지 않은 정점이 없기 때문에
스택 S를 한번 pop하고
스택 S의 top 값인 정점 4로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	1

스택 S: 0 1 2 3 4

DFS

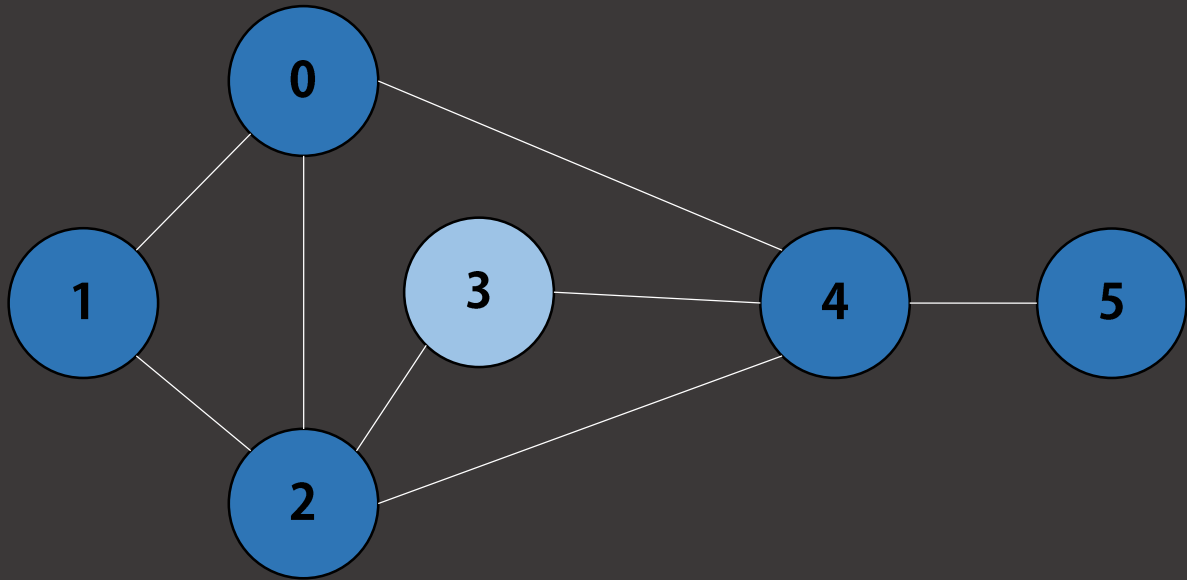


정점 4에 인접한 방문하지 않은 정점이 없기 때문에
스택 S를 한번 pop하고
스택 S의 top 값인 정점 3로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	1

스택 S: 0 1 2 3

DFS

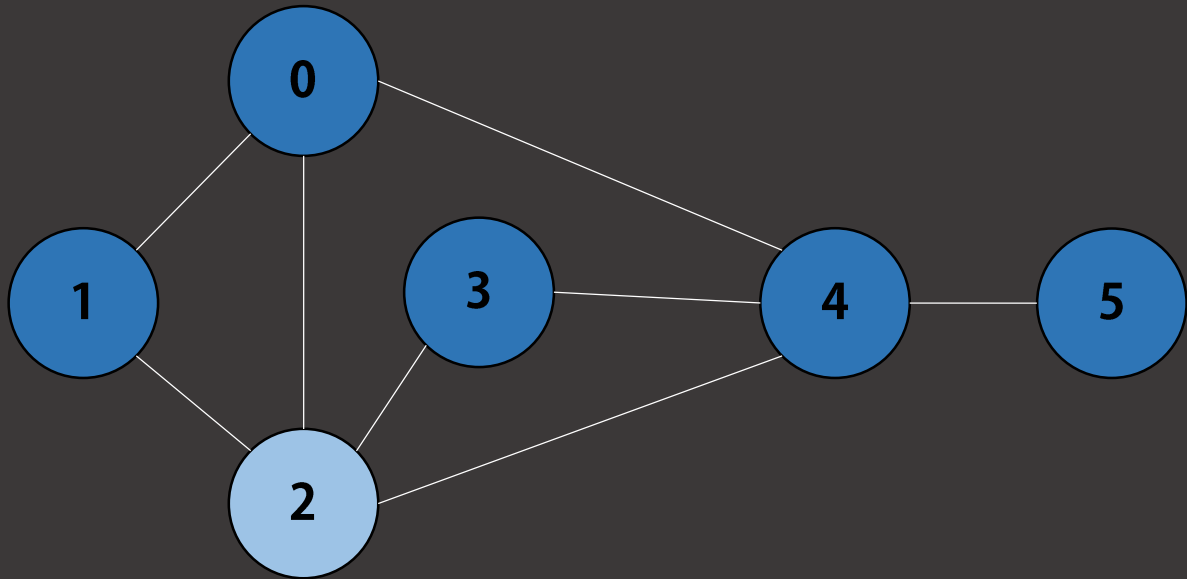


정점 3에 인접한 방문하지 않은 정점이 없기 때문에
스택 S를 한번 pop하고
스택 S의 top 값인 정점 2로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	1

스택 S: 0 1 2

DFS

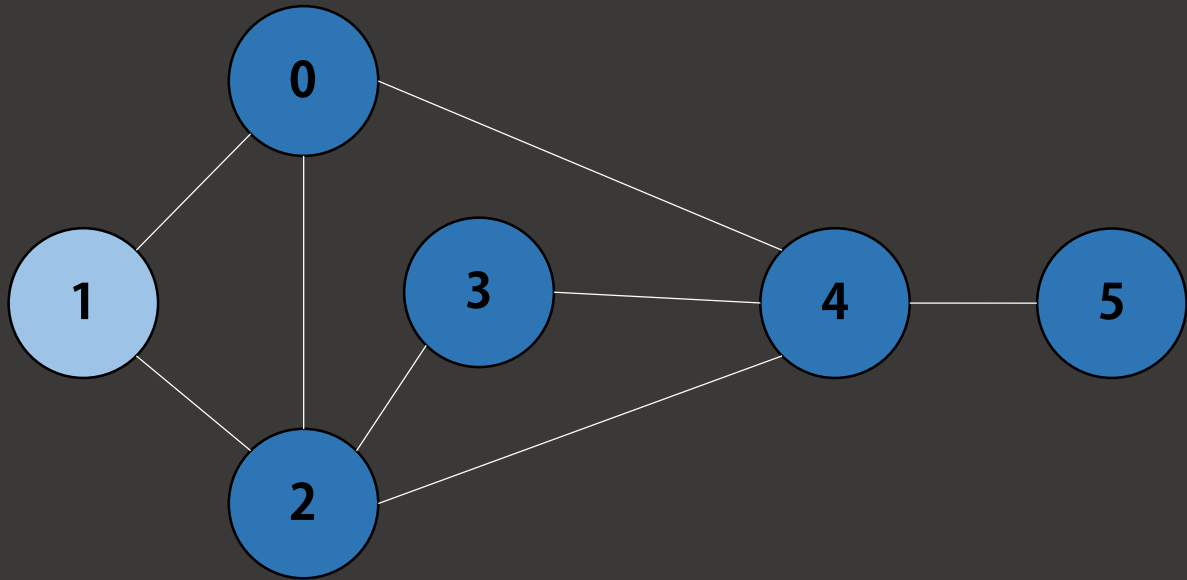


정점 2에 인접한 방문하지 않은 정점이 없기 때문에
스택 S를 한번 pop하고
스택 S의 top 값인 정점 1로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	1

스택 S: 0 1

DFS

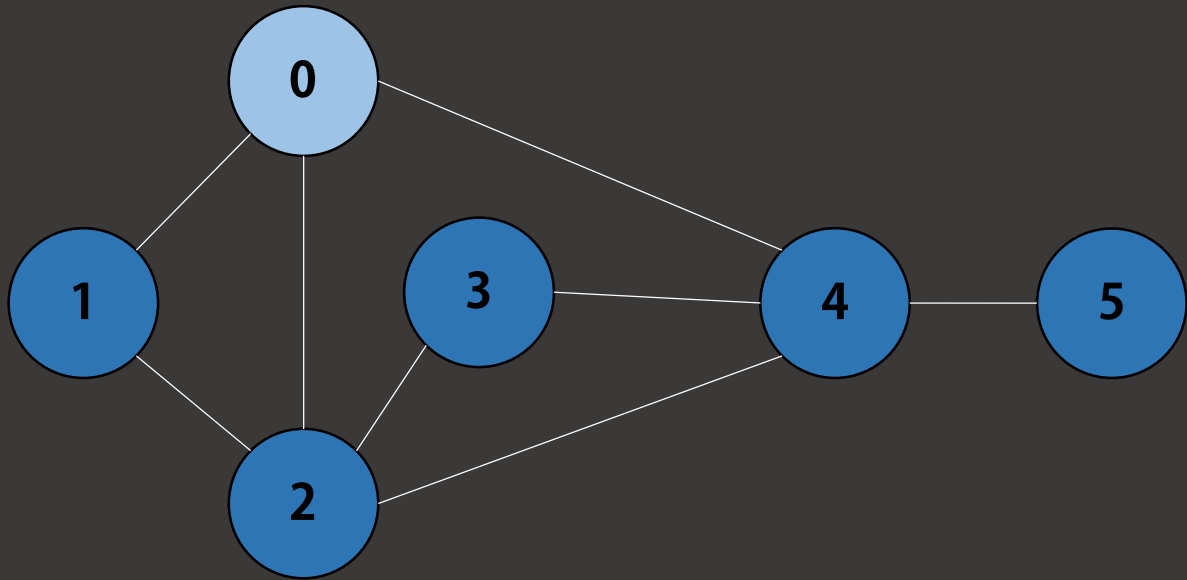


정점 1에 인접한 방문하지 않은 정점이 없기 때문에
스택 S를 한번 pop하고
스택 S의 top 값인 정점 0로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	1

스택 S: 0

DFS



정점 0에 인접한 방문하지 않은 정점이 없기 때문에
스택 S를 한번 pop합니다.
이때 스택 S가 empty이므로,
탐색을 종료합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	1

스택 S:

DFS의 작동 과정을 정리해봅시다.

1. 시작 정점을 스택에 Push하고 방문 처리합니다.
2. 스택의 TOP 정점에 방문하지 않은 인접한 정점이 하나라도 있으면
그 정점을 스택에 Push하고 방문 처리합니다.
만약 방문하지 않은 인접한 정점이 없으면 스택을 POP합니다.
3. 2번의 과정을 더 이상 수행할 수 없을 때까지(스택이 빌 때까지)
반복합니다.

DFS를 어떻게 구현할 수 있을까요?

DFS를 어떻게 구현할 수 있을까요?

재귀를 이용하여 구현하는 것이 포인트입니다.
앞서 배운 인접 행렬과 인접 리스트를 활용하여 구현해봅시다.

DFS




```
void dfs(int x){  
    check[x] = true;  
    for(int i = 1; i <= n; i++) {  
        if(a[x][i] == 1 && check[i] == false){  
            dfs(i);  
        }  
    }  
}
```

다음 코드는 **인접 행렬**을 이용하여
구현한 것입니다.
재귀 함수를 이용하여 구현할 수 있습니다.

Q. 아니 왜 스택 사용 안하나요?

A. 물론 명시적으로 스택을 사용해서 구현해도 됩니다.
하지만 코드의 가독성이나 편의를 위해 재귀 호출을 사용하였습니다.
왜냐하면 “재귀 호출이 결국 스택에 기반하기” 때문입니다.

DFS



```
void dfs(int x){  
    check[x] = true;  
    for(int i = 0; i < a[x].size(); i++){  
        int y = a[x][i];  
        if(check[y] == false):  
            dfs(y);  
    }  
}
```

다음 코드는 **인접 리스트**를 이용하여
구현한 것입니다.
인접 행렬 구현과 유사합니다.

BFS (Breadth-First Search)

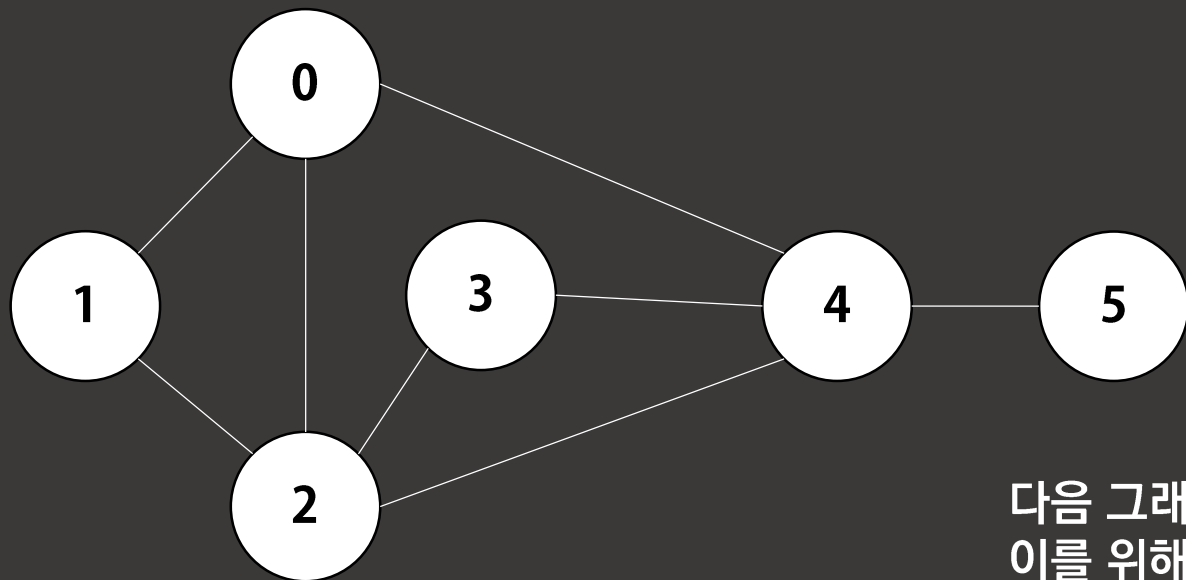
너비 우선 탐색



dfs와 다르게 너비를 우선하여 탐색하는 방식입니다.
현재 정점에서 인접한 정점을 모두 방문한 뒤, 다음 정점으로 넘어갑니다.

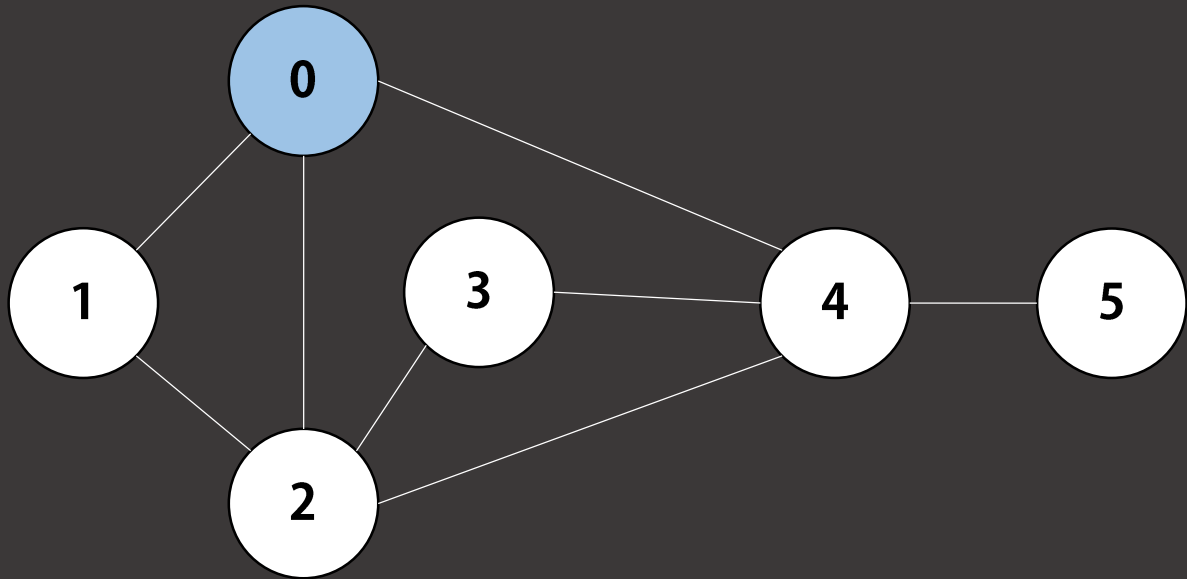
이 과정에서 **큐**를 이용합니다.

BFS



다음 그래프를 BFS로 탐색해보시다!
이를 위해 정점의 방문 여부를 체크하는
check[i] 배열과 bfs를 위한 큐 Q가 필요합니다!

BFS

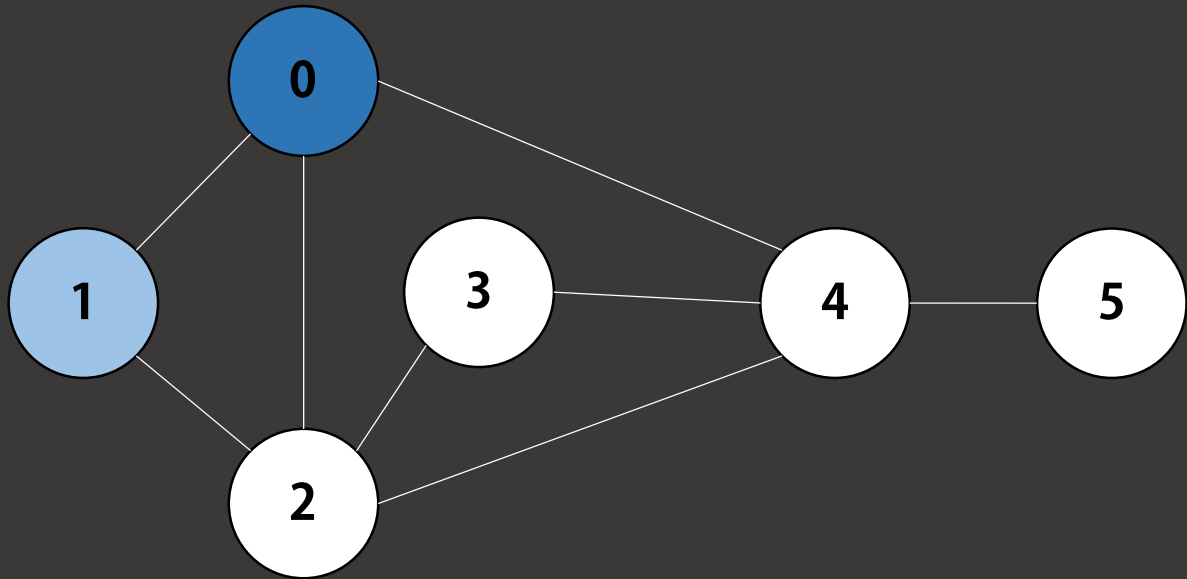


0부터 탐색을 시작하겠습니다!
현재 탐색하고 있는 정점이 0이므로
check[0] = 1로 방문 여부를 체크하고,
큐 Q에 0을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	0	0	0	0	0

큐 Q: 0

BFS

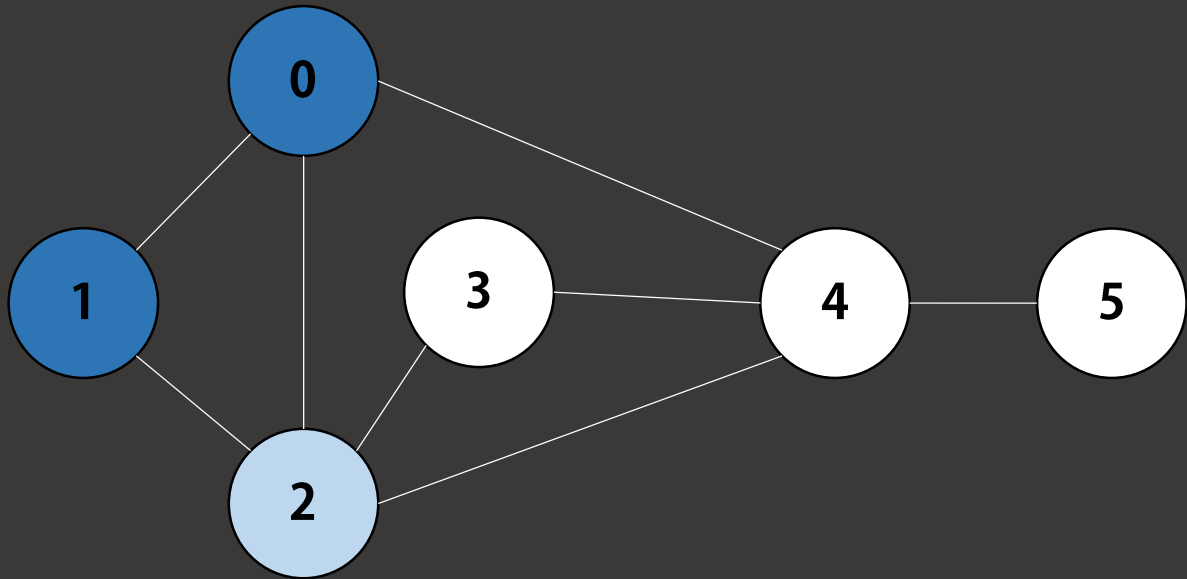


큐 Q를 pop하고
해당 값인 0에 인접한 정점인 1에 방문합니다.
check[1] = 1로 방문 여부를 체크하고,
큐 Q에 1을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	0	0	0	0

큐 Q: 1

BFS

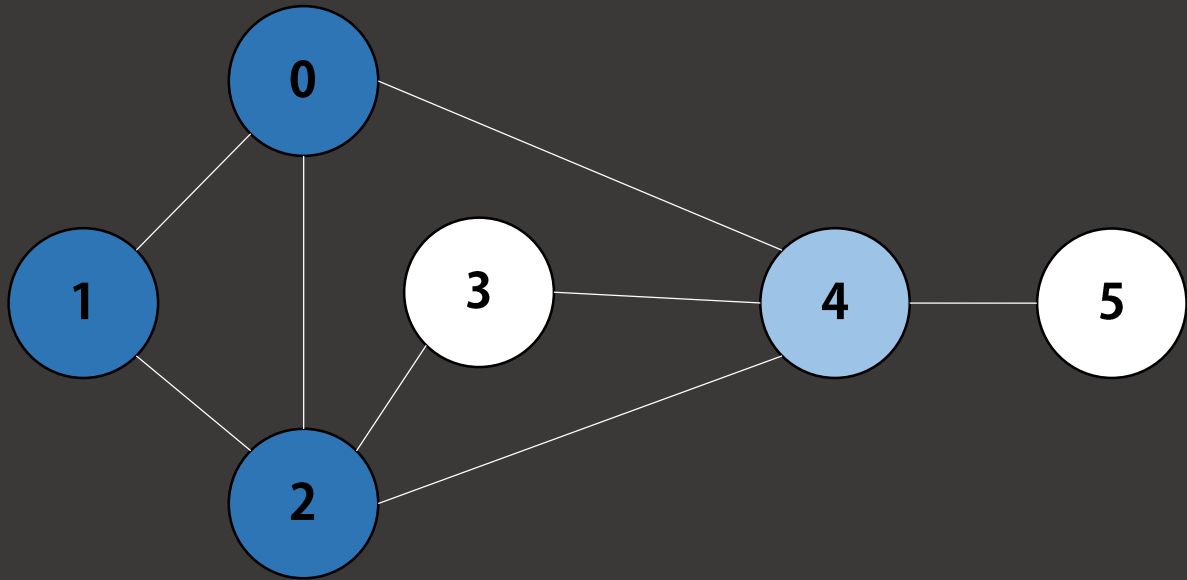


그리고 0에 인접한 정점인 2에 방문합니다.
check[2] = 1로 방문 여부를 체크하고,
큐 Q에 2을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	0	0	0

큐 Q: 1 2

BFS

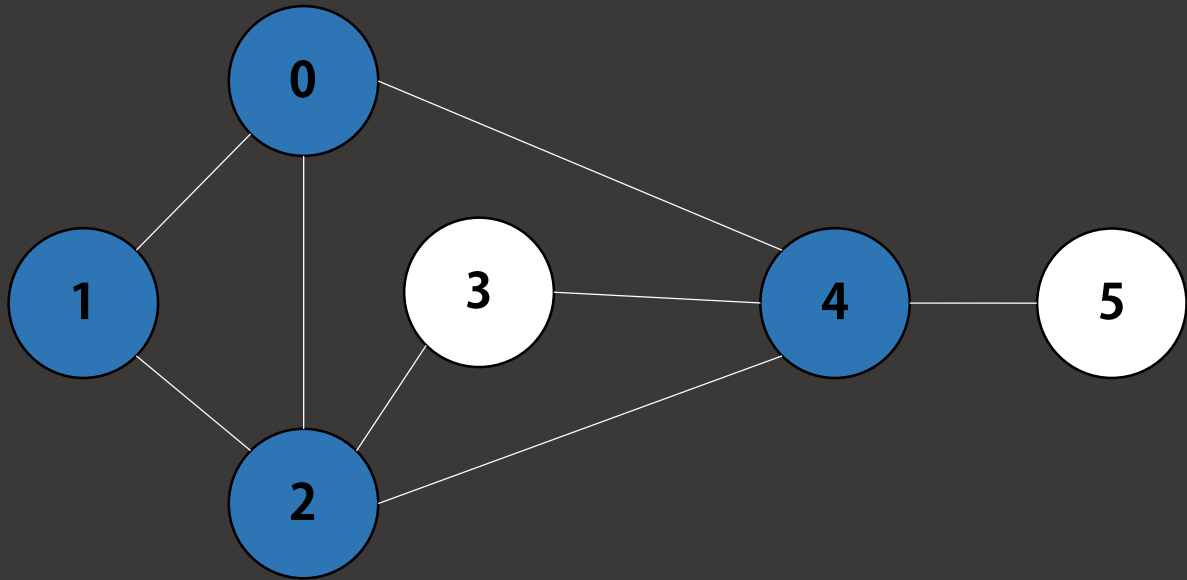


그리고 0에 인접한 정점인 4에 방문합니다.
check[4] = 1로 방문 여부를 체크하고,
큐 Q에 4을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	0	1	0

큐 Q: 1 2 4

BFS

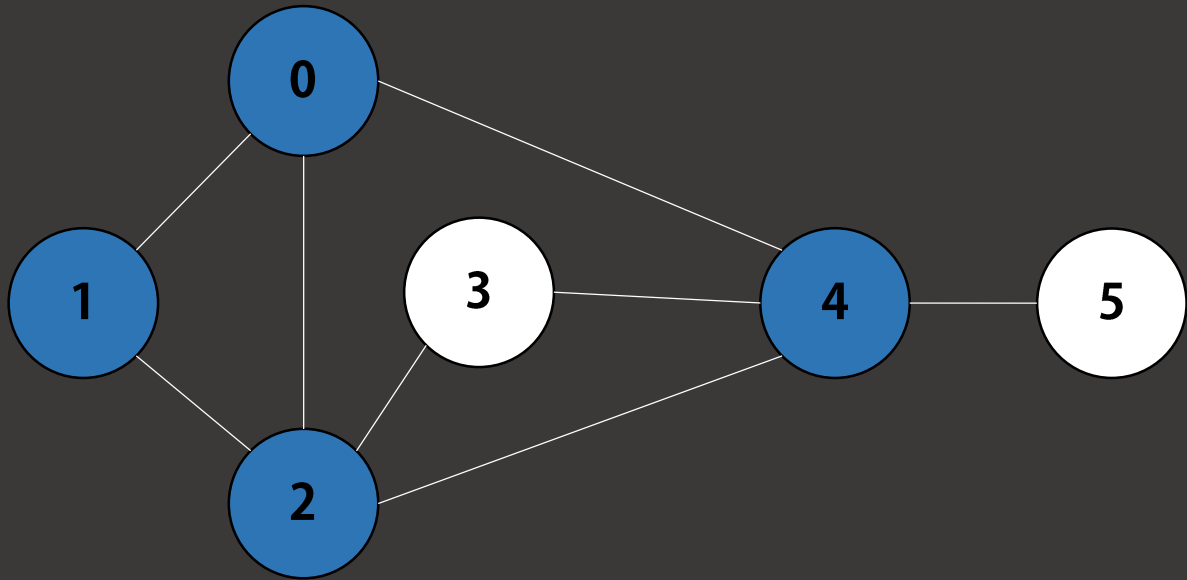


더 이상 0에 인접한 방문하지 않은 정점이 없으므로,
큐 Q를 pop하고
그 값인 1로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	0	1	0

큐 Q: 2 4

BFS

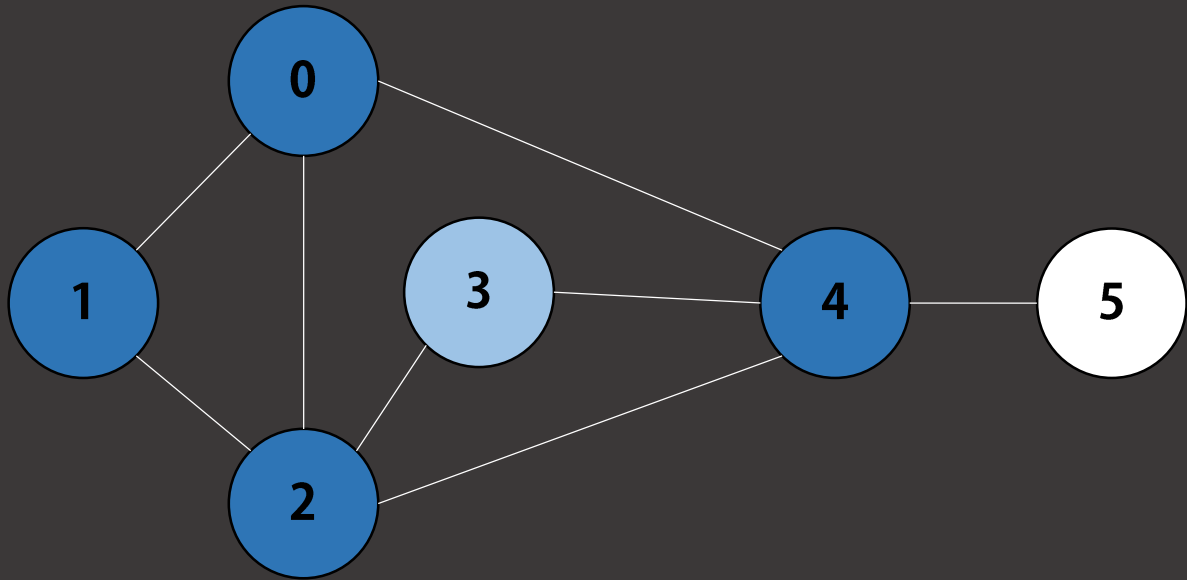


더 이상 1에 인접한 방문하지 않은 정점이 없으므로,
큐 Q를 pop하고
그 값인 2로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	0	1	0

큐 Q: 4

BFS

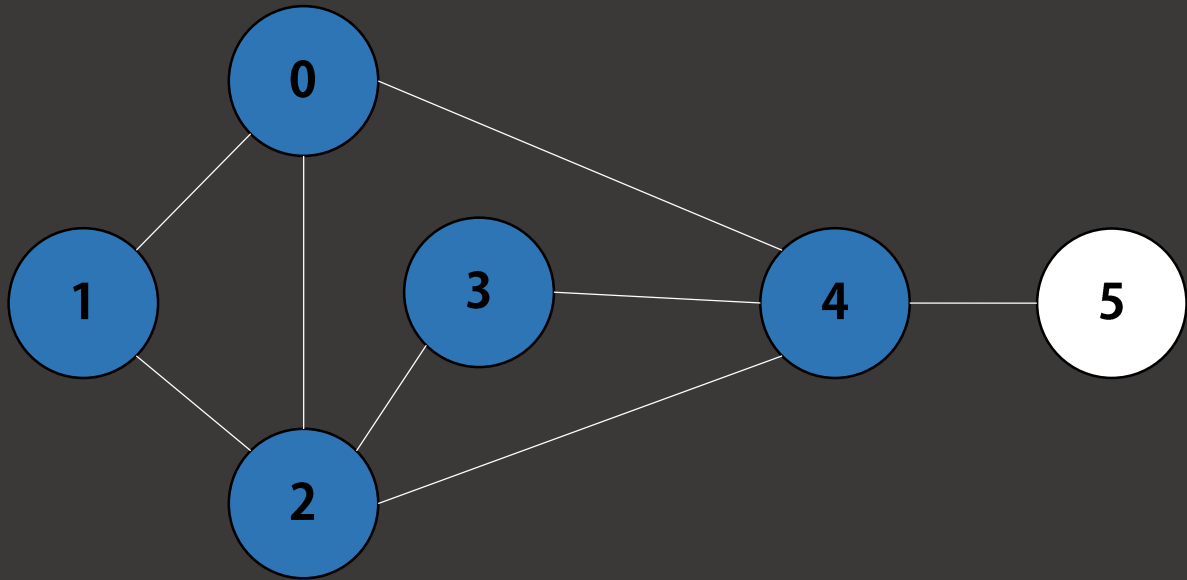


2에 인접한 정점인 3에 방문합니다.
check[3] = 1로 방문 여부를 체크하고,
큐 Q에 3을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	0

큐 Q: 4 3

BFS

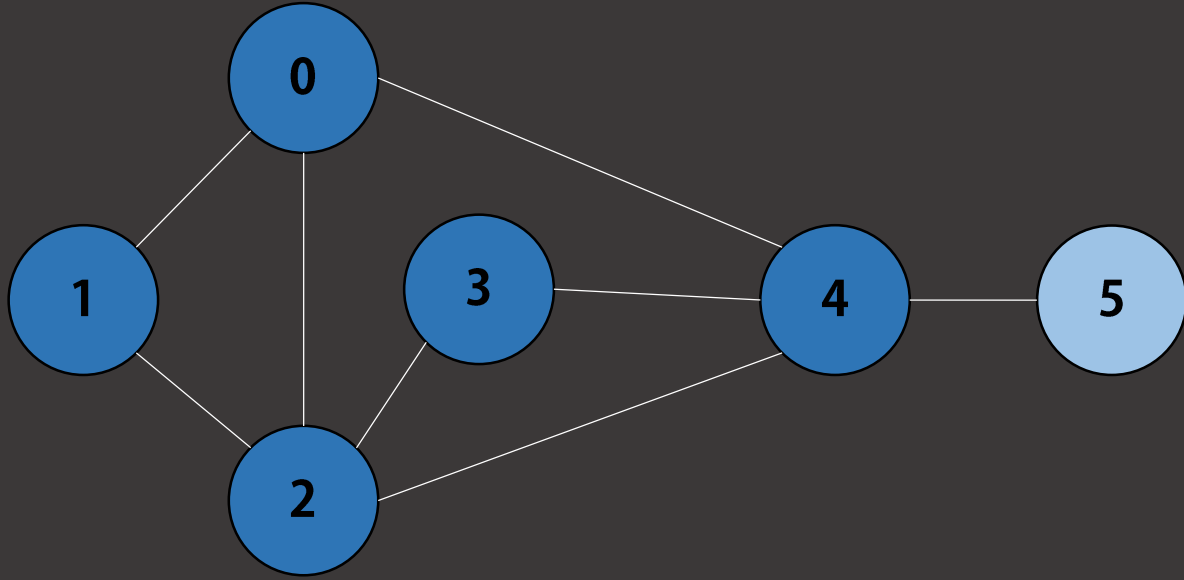


더 이상 2에 인접한 방문하지 않은 정점이 없으므로,
큐 Q를 pop하고
그 값인 4로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	0

큐 Q: 3

BFS

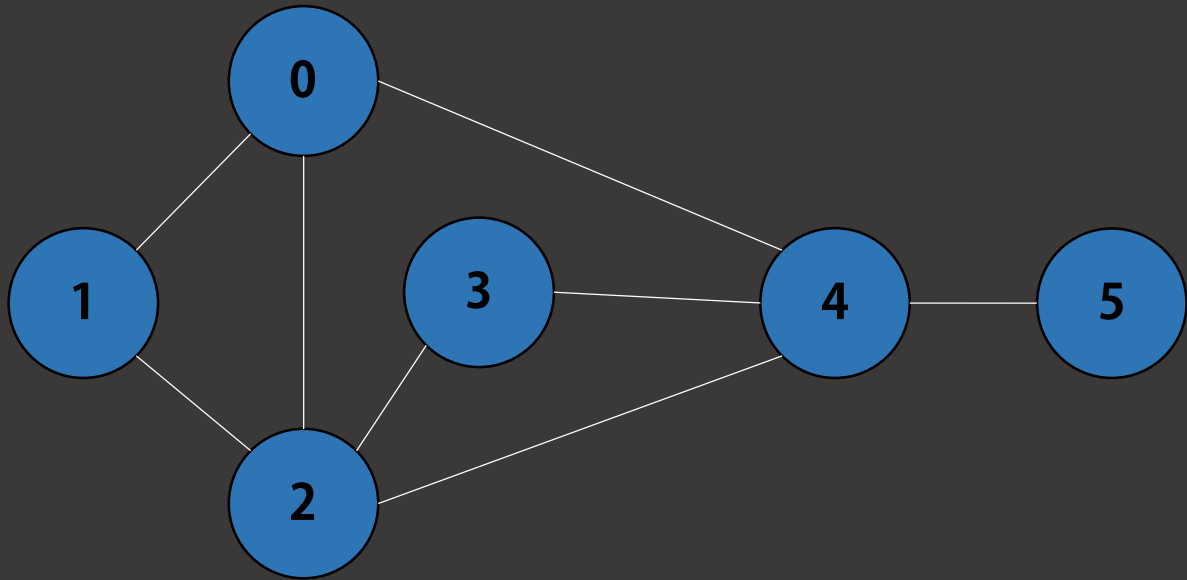


4에 인접한 정점인 5에 방문합니다.
check[5] = 1로 방문 여부를 체크하고,
큐 Q에 5을 Push 합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	1

큐 Q: 3 5

BFS

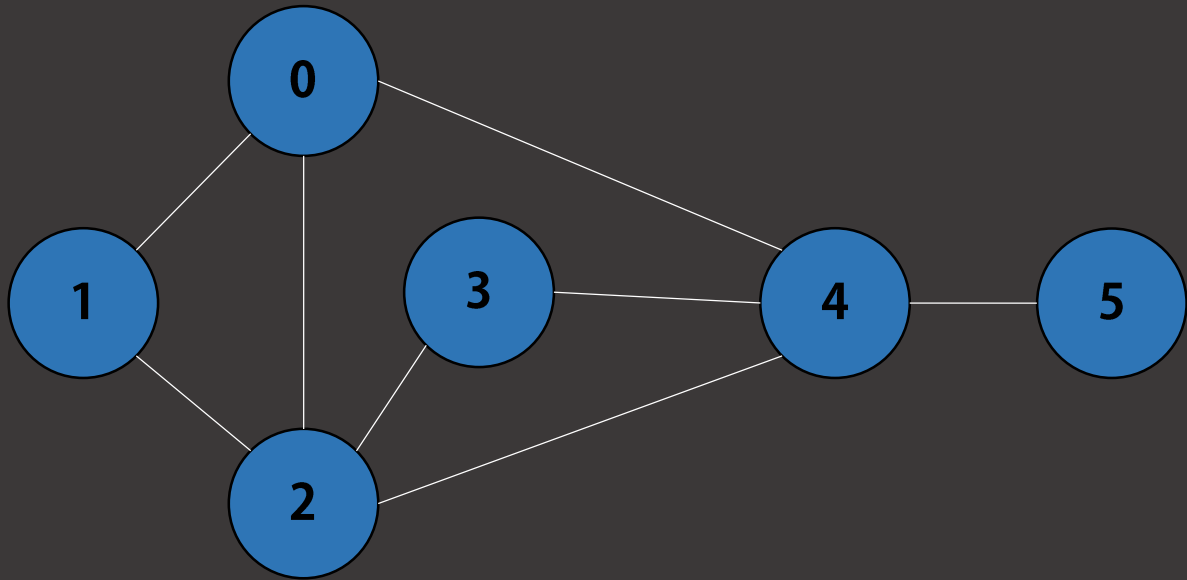


더 이상 3에 인접한 방문하지 않은 정점이 없으므로,
큐 Q를 pop하고
그 값인 5로 이동합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	0

큐 Q: 5

BFS



더 이상 5에 인접한 방문하지 않은 정점이 없으므로,
큐 Q를 pop합니다.
이때, 큐가 empty이므로
탐색을 종료합니다.

i	0	1	2	3	4	5
check[i]	1	1	1	1	1	0

큐 Q:

BFS의 작동 과정을 정리해봅시다.

1. 시작 정점을 큐에 Push하고 방문 처리합니다.
2. 큐에서 Pop한 정점에 방문하지 않은 인접한 정점이 하나라도 있으면
그 정점을 Q에 Push하고 방문 처리합니다.
만약 방문하지 않은 인접 정점이 없으면 큐를 POP합니다.
3. 2번의 과정을 더 이상 수행할 수 없을 때까지(큐가 빌 때까지)
반복합니다.

BFS를 어떻게 구현할 수 있을까요?

BFS를 어떻게 구현할 수 있을까요?

큐를 이용하여 구현하는 것이 포인트입니다.
앞서 배운 인접 행렬과 인접 리스트를 활용하여 구현해봅시다.

BFS

```
queue<int> q;
void bfs(int x){
    check[x] = true;
    q.push(x);
    while(!q.empty()){
        int y = q.front();
        q.pop();
        for(int i = 1; i <= n; i++) {
            if(a[y][i] == 1 && check[i] == false){
                check[i] = true;
                q.push(i);
            }
        }
    }
}
```

다음 코드는 **인접 행렬**을 이용하여
구현한 것입니다.
큐를 이용하여 구현할 수 있습니다.

BFS

```
queue<int> q;
void bfs(int x){
    check[x] = true;
    q.push(x);
    while(!q.empty()){
        int y = q.front();
        q.pop();
        for(int i = 0; i < a[y].size(); i++) {
            int z = a[y][i];
            if(check[z] == false) {
                check[z] = true;
                q.push(z);
            }
        }
    }
}
```

다음 코드는 **인접 리스트**를 이용하여
구현한 것입니다.
큐를 이용하여 구현할 수 있습니다.

깊이 우선 탐색
DFS

VS

너비 우선 탐색
BFS

우선, DFS와 BFS 모두
시간 복잡도는 인접 행렬을 사용하면 $O(V^2)$,
인접 리스트를 사용하면 $O(V+E)$ 입니다.
(왜 그럴까요? 한번 생각해봅시다!)

즉, 시간&공간 성능적인 측면에서 두 BFS, DFS 알고리즘은 유사한 것을 알 수 있습니다.

그렇다면 기능적인 측면에서의 장단점은 어떻게 될까요?

BFS의 경우 최단 거리 탐색, DFS의 경우 가중치 탐색 등에 유리합니다!

즉, 문제 유형에 따라 적절한 탐색 알고리즘을 사용하면 문제를 보다 수월하게 해결할 수 있습니다.

자세한 내용은 앞으로 문제를 풀어보며 알아보겠습니다!



연습 문제

1260번 DFS와 BFS

<https://www.acmicpc.net/problem/1260>

앞서 배운 개념들을 확인하는 간단한 문제입니다!



연습 문제

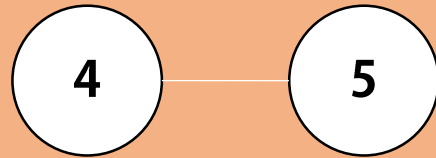
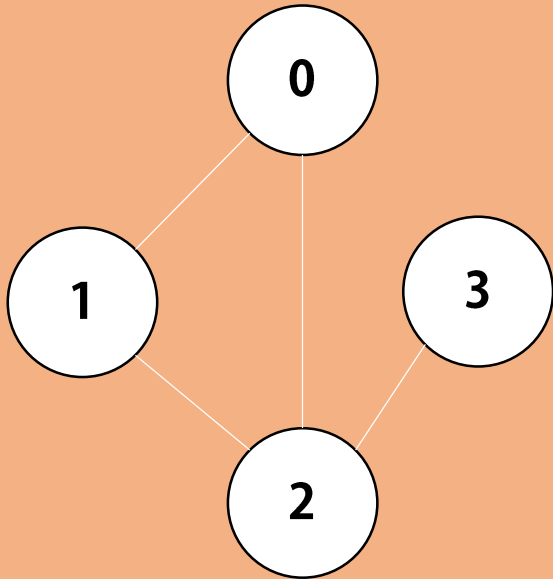
11724번 연결 요소의 개수

<https://www.acmicpc.net/problem/11724>

연결 요소(Connected Component)가 무엇인지는
다음 슬라이드에서 알아보시다!



연결 요소(Connected Component)란?



왼쪽과 같이 그래프가 나누어진 경우도 있습니다.
이때 각각의 그래프를 연결 요소라고 합니다!

자세한 정의는 다음 [링크](#)에서
Connected Component 설명을 참고해주세요!



연습 문제

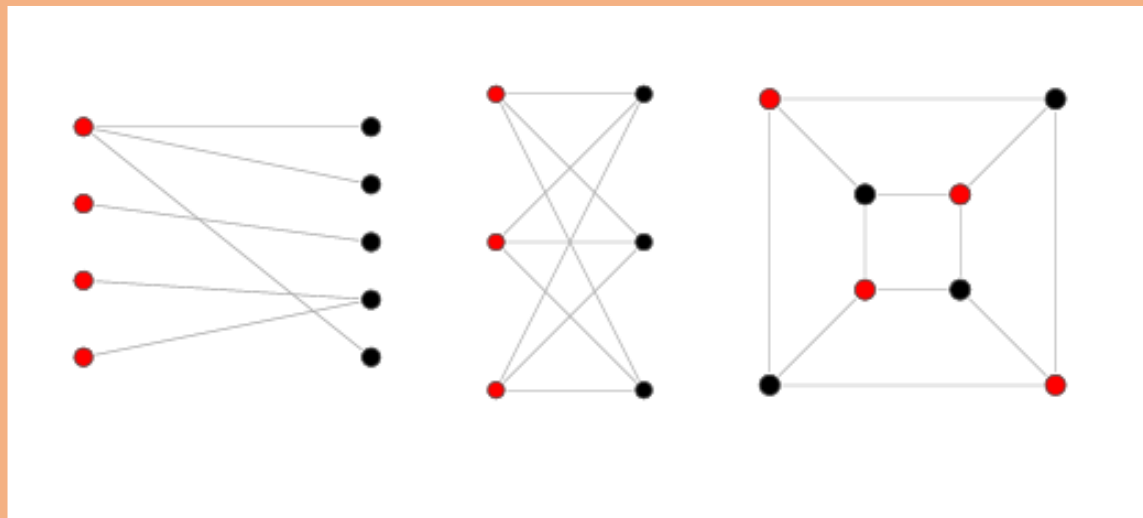
1707번 이분 그래프

<https://www.acmicpc.net/problem/1707>

이분 그래프(Bipartite Graph)가 무엇인지는
다음 슬라이드에서 알아보시다!



이분 그래프(Bipartite Graph)란?



왼쪽과 같이 그래프와 같이
인접한 정점끼리 서로 다른 색으로 칠해서
모든 정점을 두 가지 색으로 칠할 수 있는
그래프입니다!

정리

이번 시간에는

그래프에 대해 알아보았습니다!

그래프의 정의와 종류, 용어에 대해 알아보고,
두가지 그래프 탐색 방법 DFS, BFS에 대해 배웠습니다.
그래프는 DP와 더불어 PS에서 자주 출제되는 개념 중 하나입니다!
또한 '탐색'은 앞으로 알고리즘을 배우며 자주 보게 될 개념입니다.

문제들을 풀어보며, 개념을 탄탄하게 정리해 봅시다!

The End

수고하셨습니다!

다음 시간에는
그래프 친구

트리에 대해 알아보시다!