

# 청년 AI 아카데미 19기 알고리즘 실습

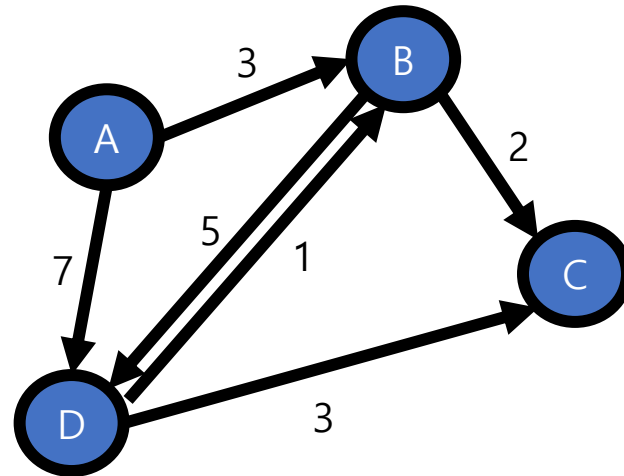
## 그래프

# Today

- **그래프 자료구조**
  - 인접 행렬 그래프
  - 인접 리스트 그래프
- **그래프 탐색**
  - BFS
  - DFS
- **그래프 심화**
  - 최단 경로 찾기

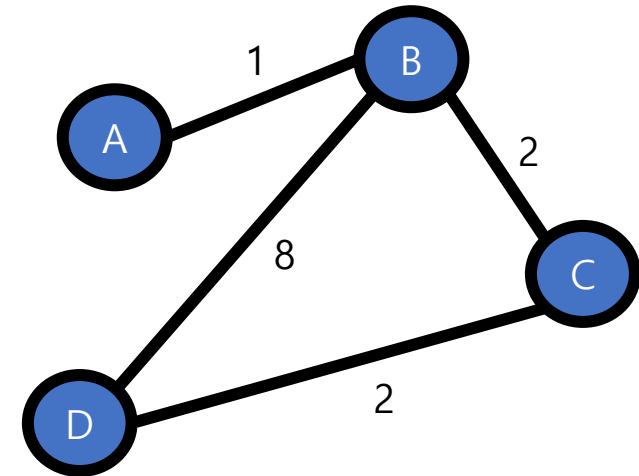
# 그래프

방향성 그래프 (Directed graph)

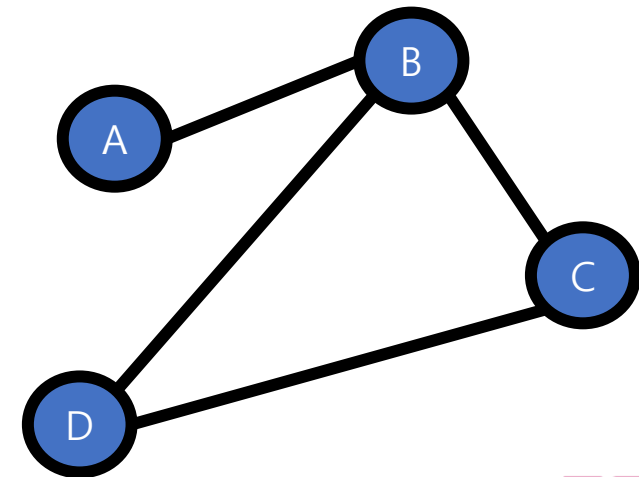
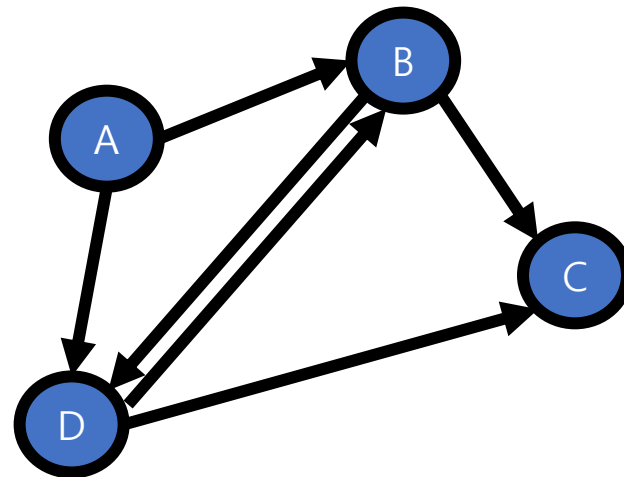


가중치 그래프  
(Weighted graph)

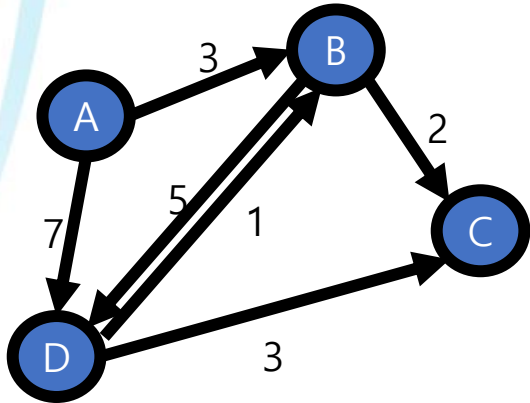
무방향성 그래프 (Undirected graph)



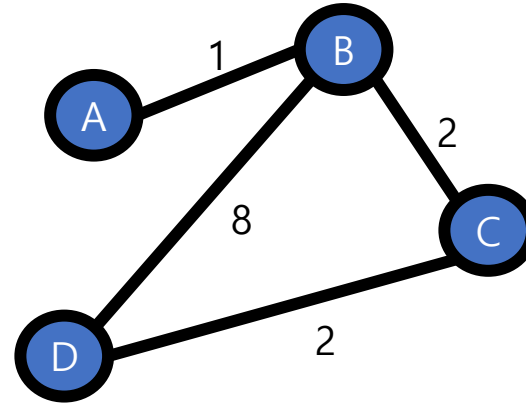
비가중치 그래프  
(Unweighted graph)



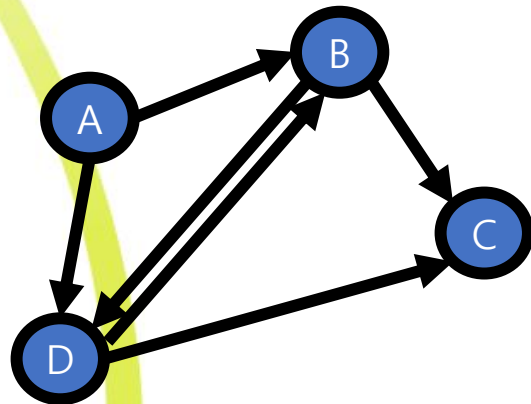
# 그래프 - 인접 행렬



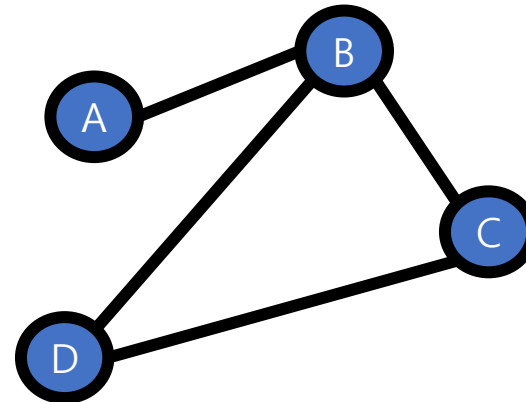
	A	B	C	D
A	0	3	0	7
B	0	0	2	5
C	0	0	0	0
D	0	1	3	0



	A	B	C	D
A	0	1	0	0
B	1	0	2	8
C	0	2	0	2
D	0	8	2	0

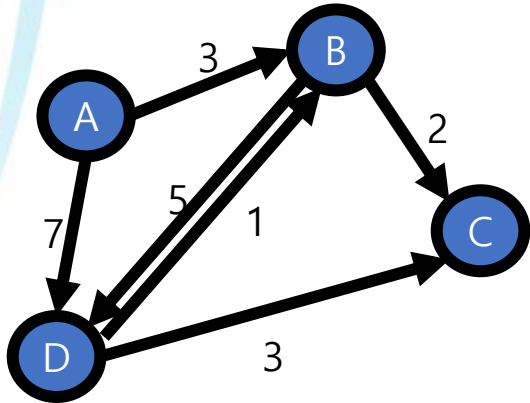


	A	B	C	D
A	0	1	0	1
B	0	0	1	1
C	0	0	0	0
D	0	1	1	0

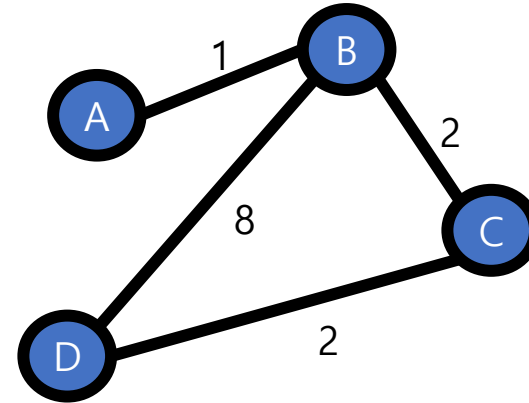


	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	1	0	1
D	0	1	1	0

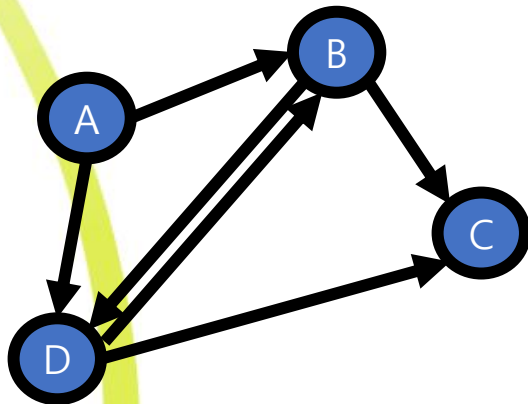
# 그래프 - 인접 리스트



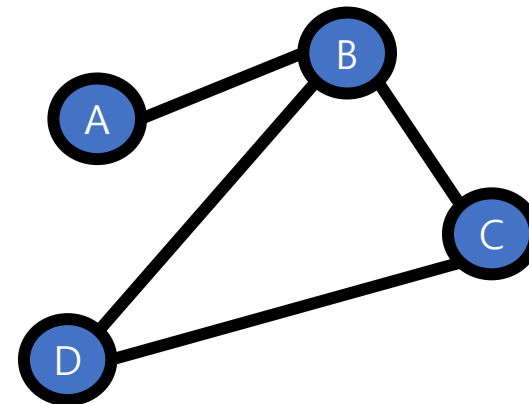
<b>A</b>	→	(B,3)	(D,7)
<b>B</b>	→	(C,2)	(D,5)
<b>C</b>	→		
<b>D</b>	→	(B,1)	(C,3)



<b>A</b>	→	(B,1)		
<b>B</b>	→	(A,1)	(C,2)	(D,8)
<b>C</b>	→	(B,2)	(D,2)	
<b>D</b>	→	(B,8)	(C,2)	



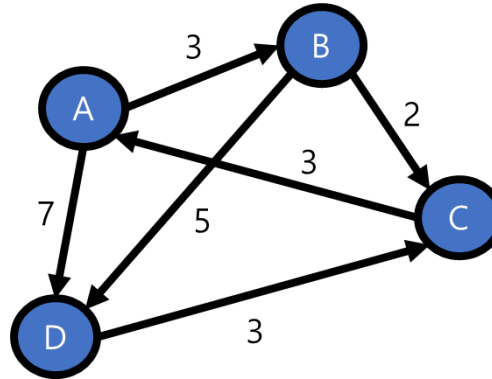
<b>A</b>	→	B	D
<b>B</b>	→	C	D
<b>C</b>	→		
<b>D</b>	→	B	C



<b>A</b>	→	B		
<b>B</b>	→	A	C	D
<b>C</b>	→	B	D	
<b>D</b>	→	B	C	

# 인접 행렬 vs 인접 리스트

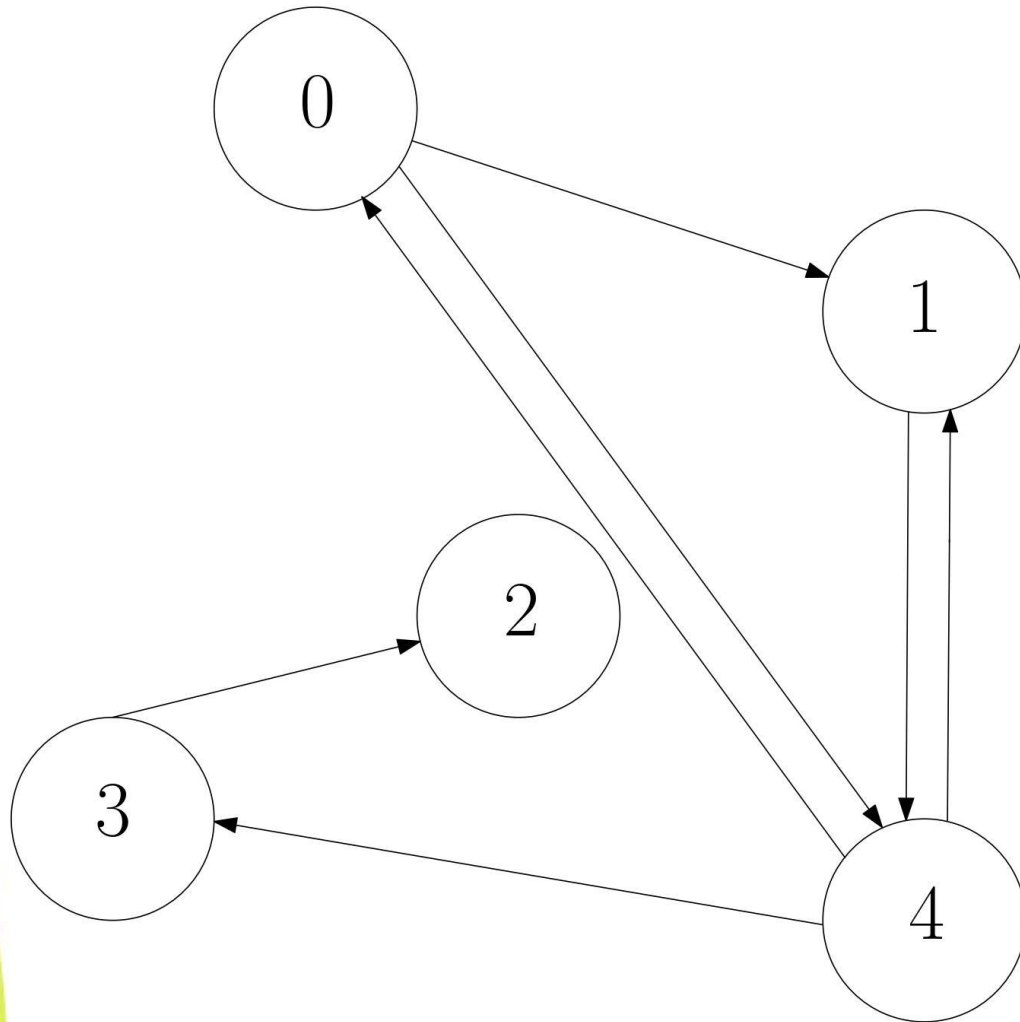
	A	B	C	D
A	0	3	0	7
B	0	0	2	5
C	3	0	0	0
D	0	0	3	0



A	B,3	D,7
B	C,2	D,5
C	A,3	
D	C,3	

인접 행렬	목록	인접 리스트
$O( V ^2)$	공간 복잡도	$O( V  +  E )$
$O(1)$	두 노드 사이의 간선 확인 시간	$O(\text{outdeg}(v))$
$O( V )$	한 노드의 모든 간선 확인 시간	$O(\text{outdeg}(v))$

# 01. 인접 행렬 구현하기

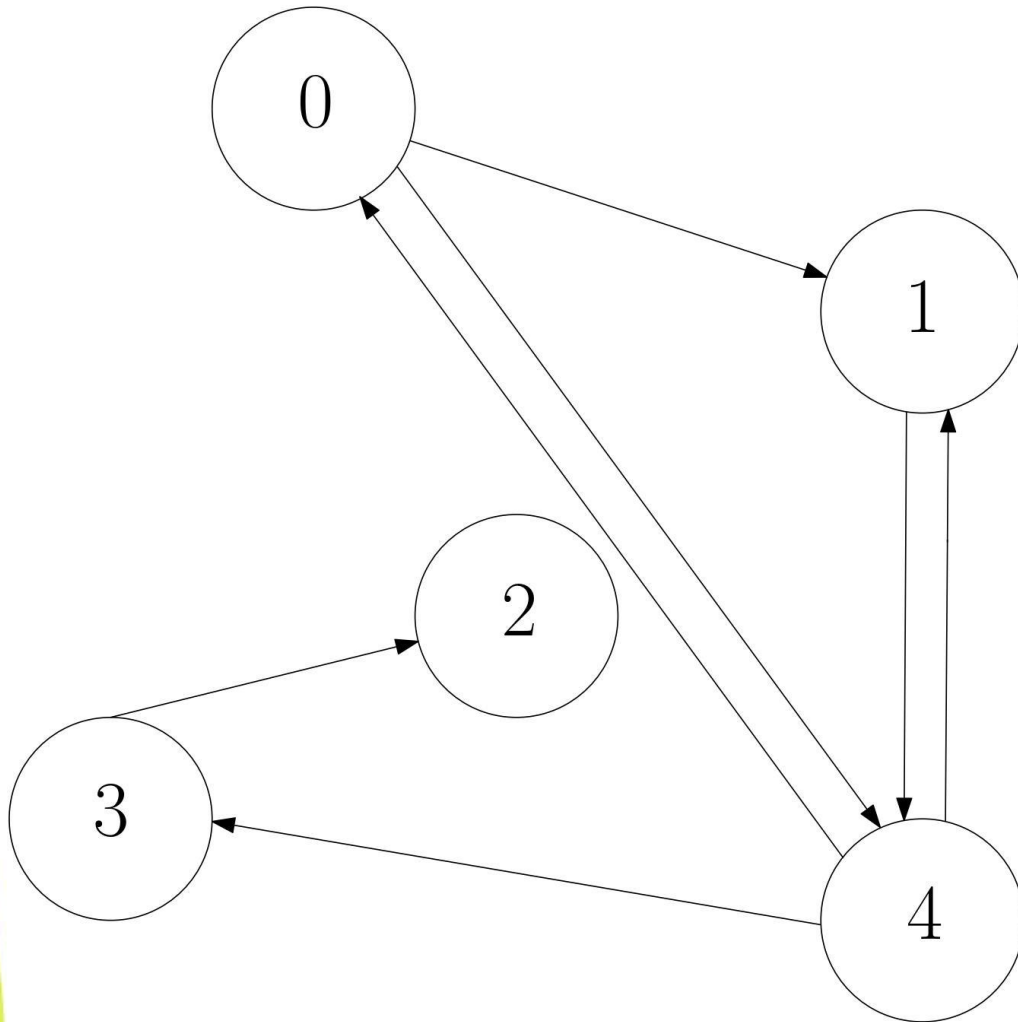


$N = 5, M = 7$

$E = (0,1), (1,4), (4,1), (0,4),$   
 $(4,0), (4,3), (3,2)$

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

# 01. 인접 행렬 구현하기



$N = 5, M = 7$

$E = (0,1), (1,4), (4,1), (0,4),$   
 $(4,0), (4,3), (3,2)$

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



# 01. 인접 행렬 구현하기

## 인접 행렬 구현하기

### 문제 정의

강의 때 배운 내용을 이용하여, 가중치가 있는 방향성 그래프가 주어졌을 때 이를 인접 행렬로 표현하는 프로그램을 작성하세요.

방향성 그래프에서 어떤 간선  $(u, v, c)$ 의 의미는  $u$ 에서  $v$ 로 가는 비용이  $c$ 인 간선이 있다는 것을 의미합니다.

### 입력 형식

- 입력의 첫 줄에 테스트 케이스의 숫자  $t$ 가 주어진다.
- 각 테스트 케이스마다 입력은 아래와 같다.
  - 첫 줄에 정점의 개수  $N$ 과 간선의 개수  $M$ 이 주어진다. ( $N \leq 1,000, M \leq 20,000$ )
  - 그 다음  $M$ 개의 줄에 걸쳐서 방향성 그래프의 간선  $(u, v, c)$ 가 공백을 사이에 두고 입력된다.
  - $u$ 와  $v$ 를 다 일치하는 간선은 여러 번 입력되지 않으며,  $u$ 와  $v$ 는 항상 0부터  $N - 1$  사이의 정수이고,  $c$ 는 자연수이다.

### 출력 형식

- 각 테스트 케이스에 대해 입력받은 그래프를 인접 행렬로 표시한 결과를 출력한다.
- 각 테스트 케이스에서 출력하는  $i$ 번째 줄의  $j$ 번째 숫자는 간선  $(i, j, c)$ 가 존재하면  $c$ 를 출력하고, 그렇지 않으면 0을 출력한다.

### 입력 예시

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

$N = 4, M = 6$

총  $M=6$ 개의 edge input을 받음

$i, j, c \rightarrow \text{Matrix}[i][j] = c$

(방향성 그래프이니  $\text{Matrix}[j][i]$ 는 업데이트 하지 않음!!)

### 출력 예시

0 3 0 7  
0 0 2 5  
3 0 0 0  
0 0 3 0  
0 1 1 0 1  
1 0 0 2 1  
0 0 0 0 1  
0 0 0 0 1  
0 1 2 0 0

	0	1	2	3
0	0	3	0	7
1	0	0	2	5
2	3	0	0	0
3	0	0	3	0

```
for i in range(N):
    print(*Matrix[i])
```



# 01. 인접 행렬 구현하기

```
t = int(input())
for _ in range(t):
    N,M = map(int,input().split())

    Matrix = [[0]*N for _ in range(N)]

    for i in range(M):
        u,v,c = map(int,input().split())
        Matrix[u][v] = c

    for i in range(N):
        print(*Matrix[i])
```

# 01. 인접 행렬 구현하기

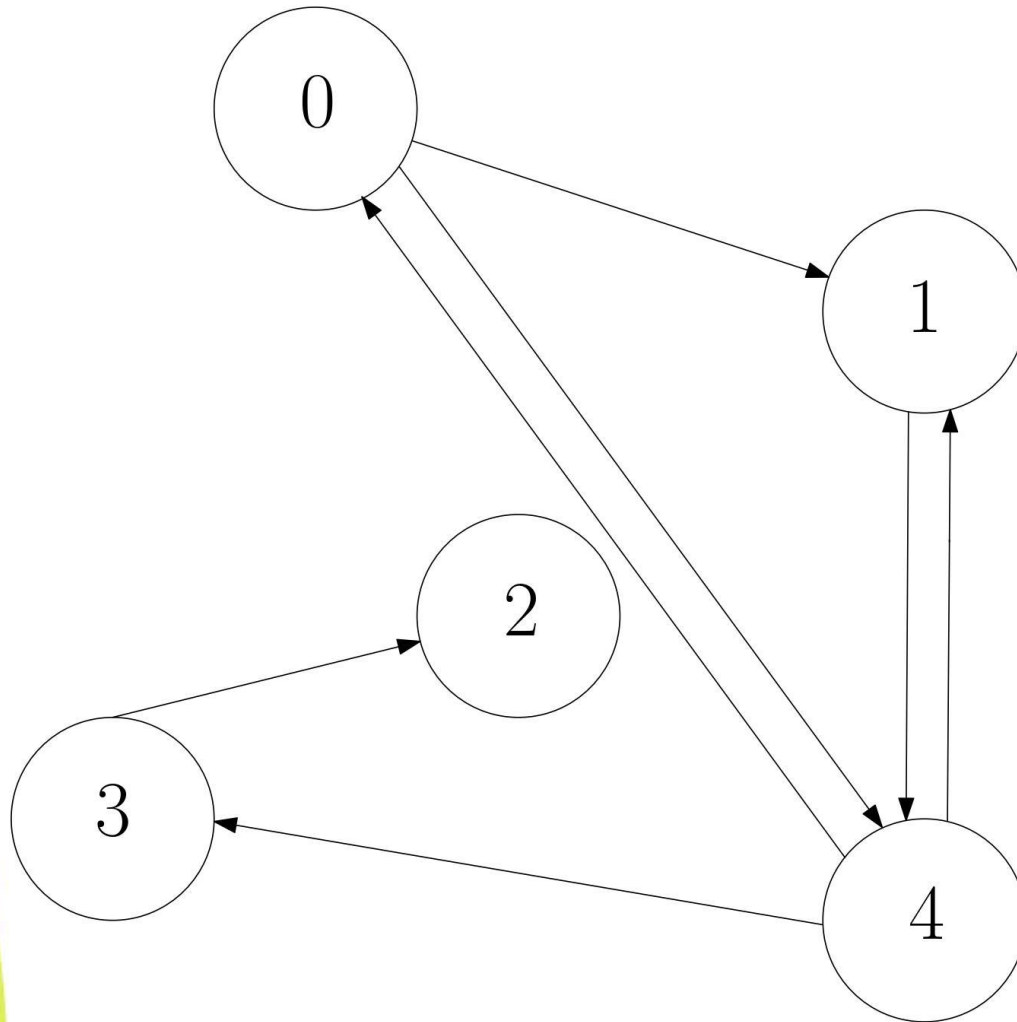
```
t = int(input())
for _ in range(t):
    N,M = map(int,input().split())

    Matrix = [[0]*N for _ in range(N)]

    for i in range(M):
        u,v,c = map(int,input().split())
        Matrix[u][v] = c

    for i in range(N):
        print(*Matrix[i])
```

## 02. 인접 리스트 구현하기

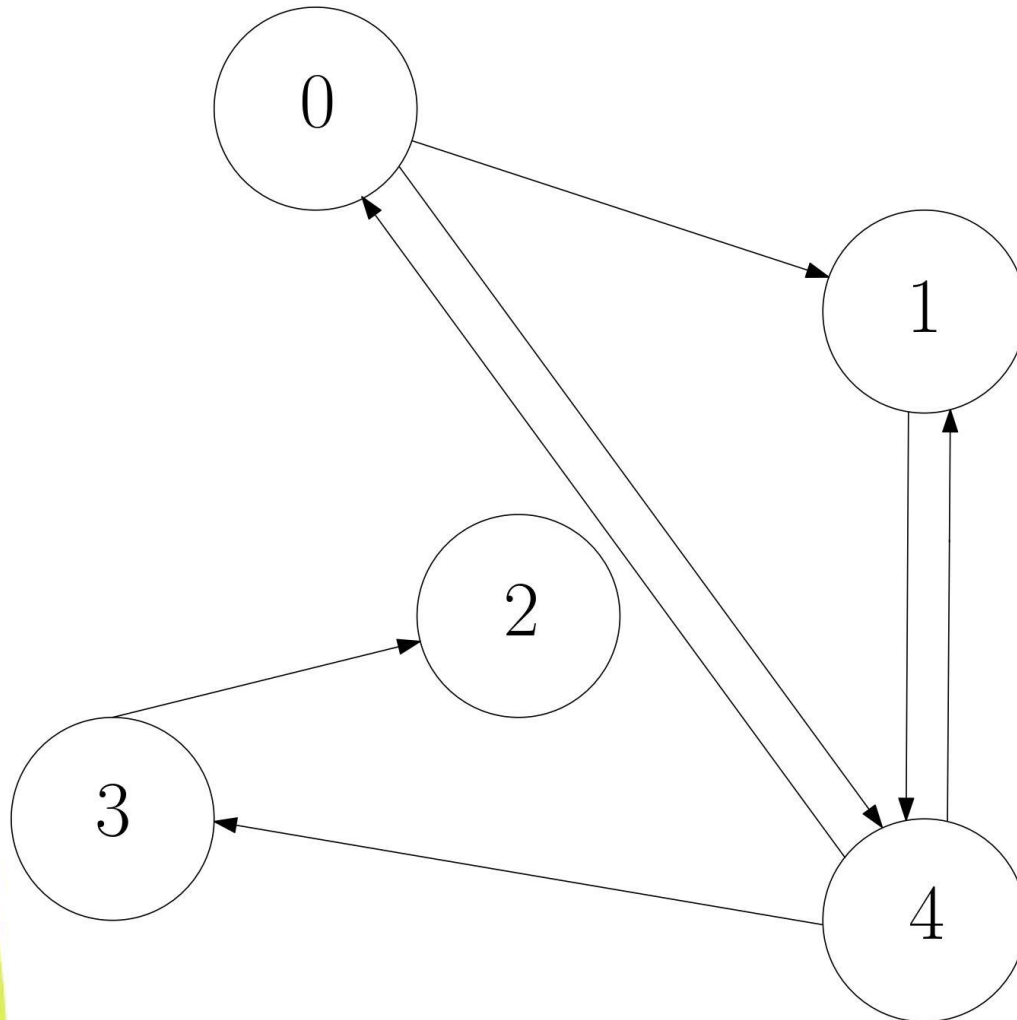


$N = 5, M = 7$

$E = (0,1), (1,4), (4,1), (0,4),$   
 $(4,0), (4,3), (3,2)$

0	
1	
2	
3	
4	

## 02. 인접 리스트 구현하기



$N = 5, M = 7$

$E = (0,1), (1,4), (4,1), (0,4),$   
 $(4,0), (4,3), (3,2)$

<b>0</b>	1,4
<b>1</b>	4
<b>2</b>	
<b>3</b>	2
<b>4</b>	0,1,3

## 02. 인접 리스트 구현하기

### 인접 리스트 구현하기

#### 문제 정의

강의 때 배운 내용을 이용하여, **가중치가 없는 무방향성 그래프**가 주어졌을 때 이를 인접 리스트로 표현하는 프로그램을 작성하세요.

무방향성 그래프에서 어떤 간선  $(u, v)$ 의 의미는  $u$ 에서  $v$ 로 가는 간선이 있다는 것을 의미하며,  $v$ 에서  $u$ 로 가는 간선 또한 존재함을 의미합니다.

#### 입력 형식

$u, v \rightarrow \text{List}[u].\text{append}(v), \text{List}[v].\text{append}(u)$

- 입력의 첫 줄에 테스트 케이스의 숫자  $t$ 가 주어진다.
- 각 테스트 케이스마다 입력은 아래와 같다.
  - 첫 줄에 정점의 개수  $N$ 과 간선의 개수  $M$ 이 주어진다. ( $N \leq 1,000, M \leq 20,000$ )
  - 그 다음  $M$ 개의 줄에 걸쳐서 무방향성 그래프의 간선  $(u, v)$ 가 공백을 사이에 두고 입력된다.
  - $u$ 와  $v$ 는 항상 0부터  $N - 1$  사이의 정수이고, 두 정점 간 간선은 유일하다.

#### 출력 형식

- 각 테스트 케이스에 대해 입력받은 그래프를 인접 리스트로 표시한 결과를 출력한다.
- 각 테스트 케이스에서 출력하는  $i$ 번째 줄은 정점  $i$ 에 연결된 정점들의 번호를 오름차순으로 하나씩 공백을 사이에 두고 출력한다.
- 정점  $i$ 에 연결된 어떠한 정점도 없다면 해당 줄은 빈 줄로 출력한다.

#### 입력 예시

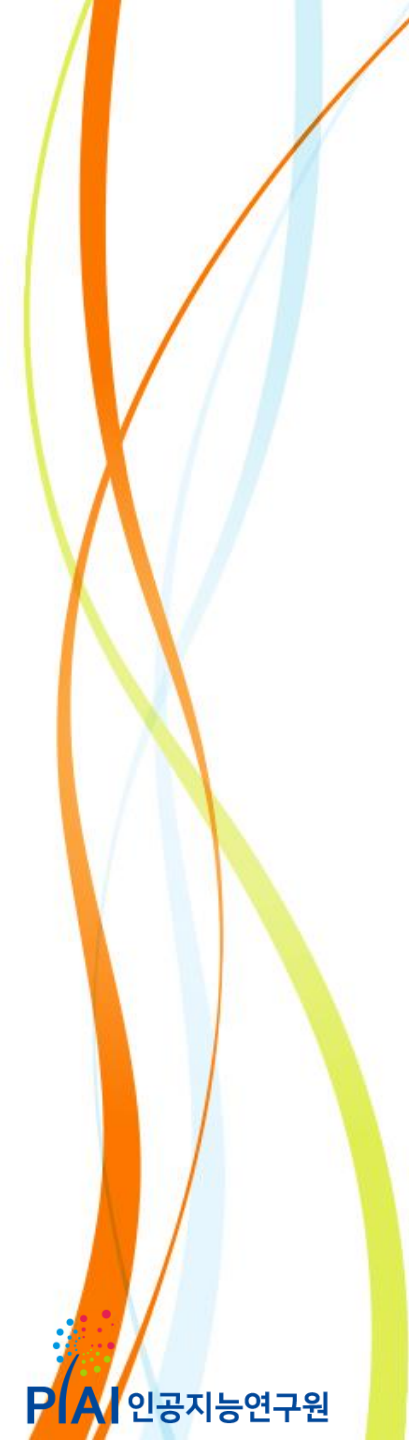
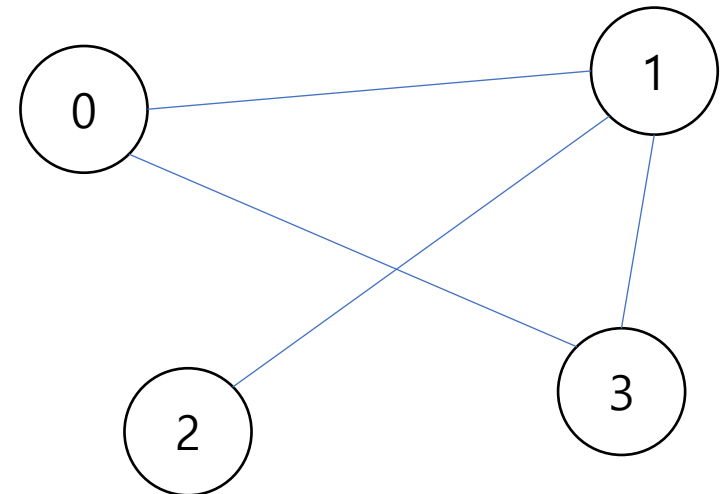
```
2
4 4
0 1
0 3
1 2
1 3
5 6
0 1
0 3
0 4
3 4
1 4
1 3
```

총  $M=4$ 개의 edge input을 받음

#### 출력 예시

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

0 1 4
0 1 3
```





## 02. 인접 리스트 구현하기

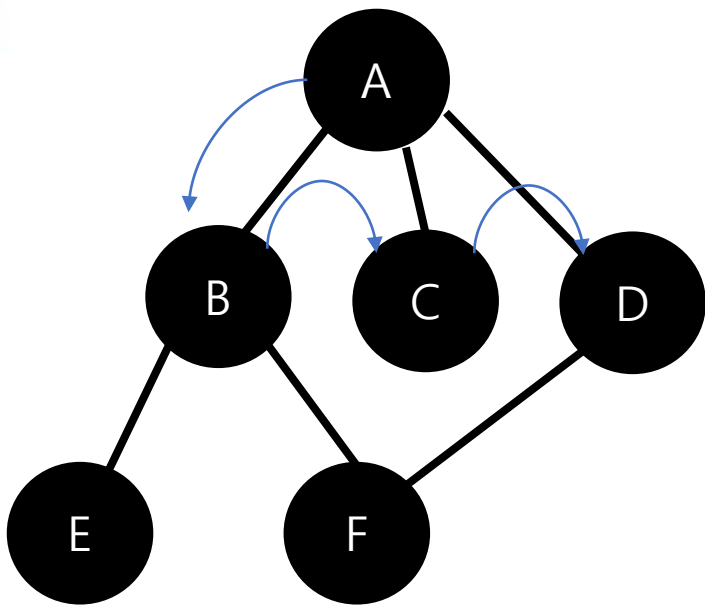
```
t = int(input())
for _ in range(t):
    N,M = map(int,input().split())

    List = [[] for _ in range(N)]

    for i in range(M):
        u,v = map(int,input().split())
        List[u].append(v)
        List[v].append(u)

    for i in range(N):
        List[i].sort()
        print(*List[i])
```

### 03. BFS (너비 우선 탐색)



Breadth-first search (BFS) 는 시작 노드에서 인접한 노드들을 먼저 탐색하는 방법

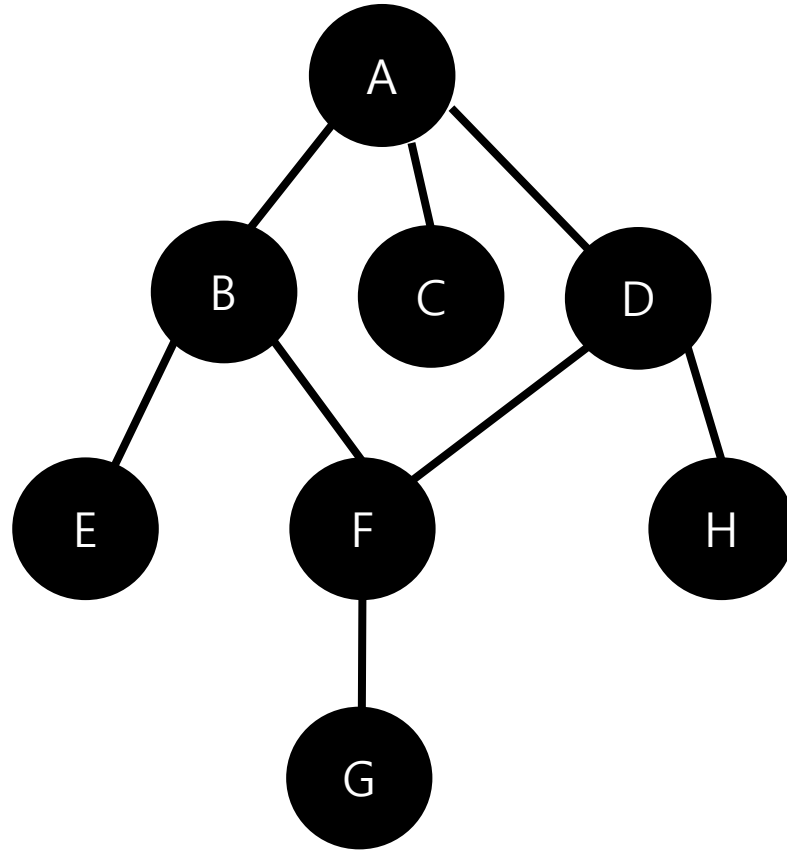
하나의 경로  $(u, v)$ 에 대해서:

1.  $u$  노드를 방문
2.  $u$  노드와 이웃한  $v$  방문
3.  $u$  노드와 이웃한  $v', v''...$  차례대로 방문
4.  $u$  노드와 이웃한 노드들을 모두 방문했다면  $v, v', v''...$  노드와 인접한 노드 중 방문하지 않은 이웃 노드들 방문

- 큐를 사용하는 방법



### 03. BFS (너비 우선 탐색)



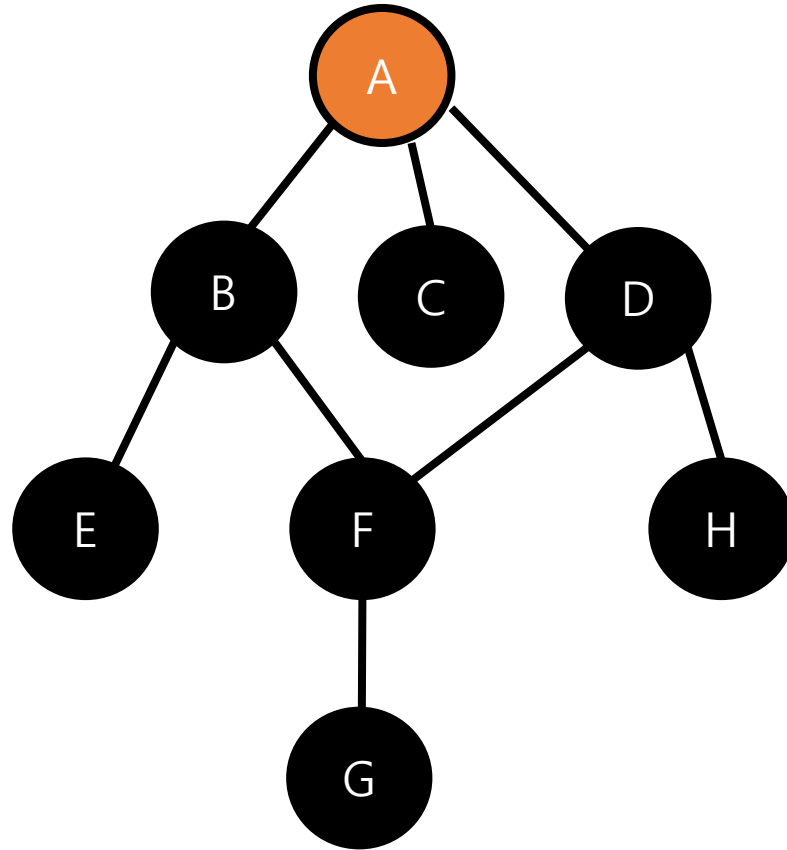
거리	
0	A
1	
2	
3	
결과	

초기 큐에 시작 노드인 A push

큐에서 pop된 A노드를 방문으로 저장

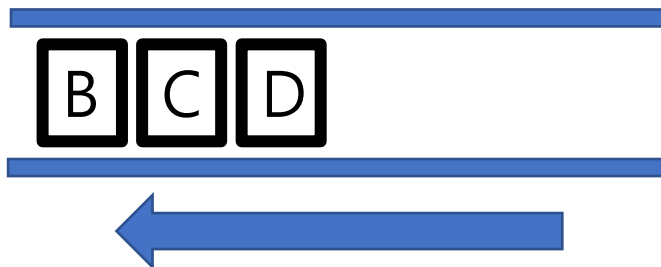


### 03. BFS (너비 우선 탐색)

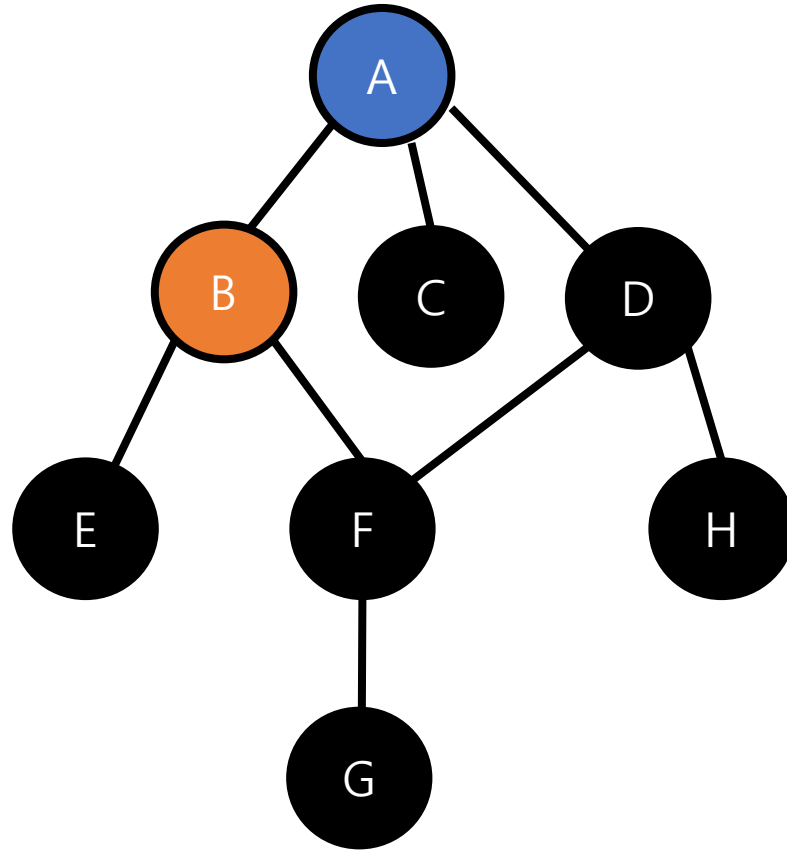


거리	
0	
1	B C D
2	
3	
결과	A

큐에 A 노드와 인접한 방문하지 않은 노드인 B, C, D push

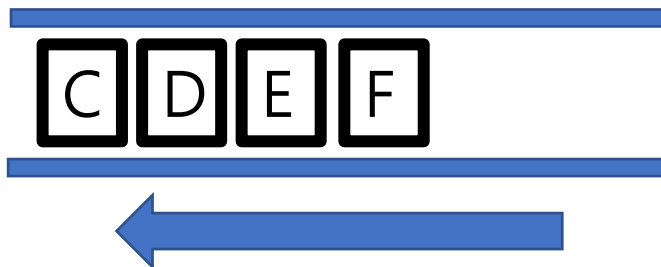


### 03. BFS (너비 우선 탐색)

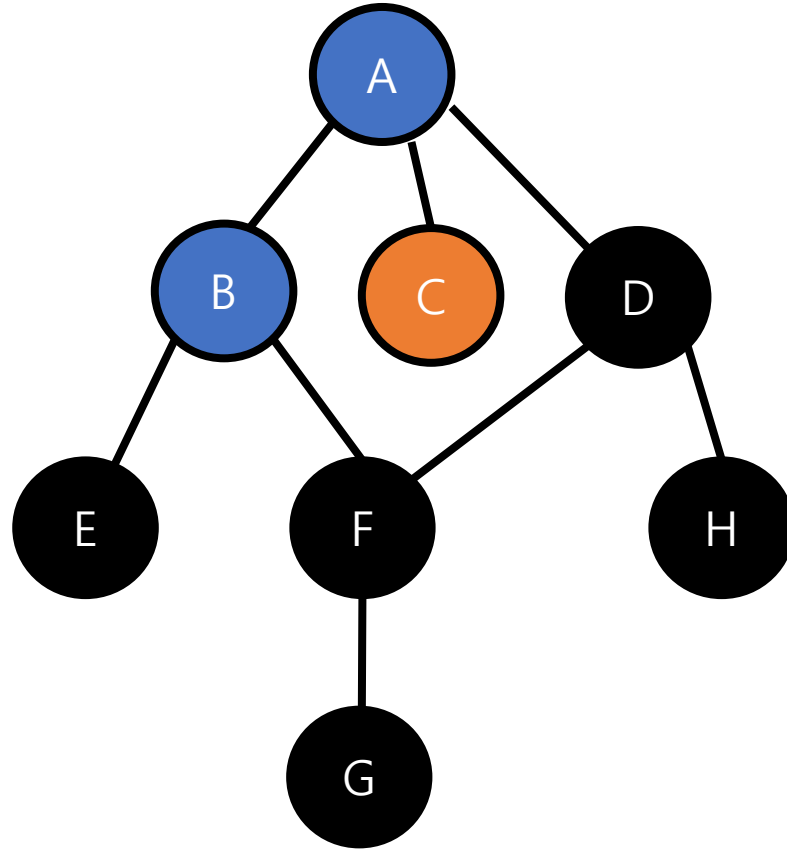


거리	
0	
1	C D
2	E F
3	
결과	A B

큐에서 pop된 B노드 방문  
B노드와 이웃한 방문하지 않은 노드인 E, F push

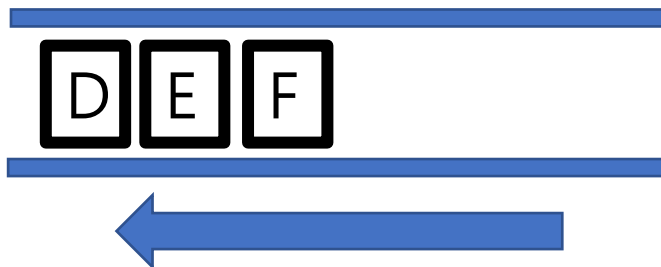


### 03. BFS (너비 우선 탐색)

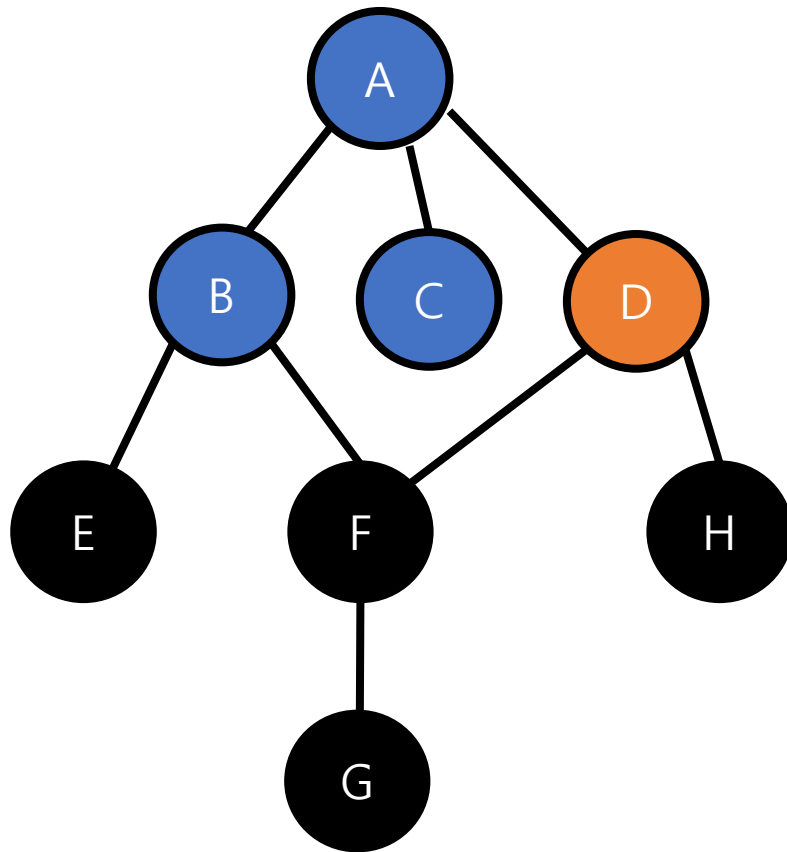


거리	
0	
1	D
2	E F
3	
결과	A B C

큐에서 pop된 C노드 방문  
C노드와 이웃한 방문하지 않은 노드 없음

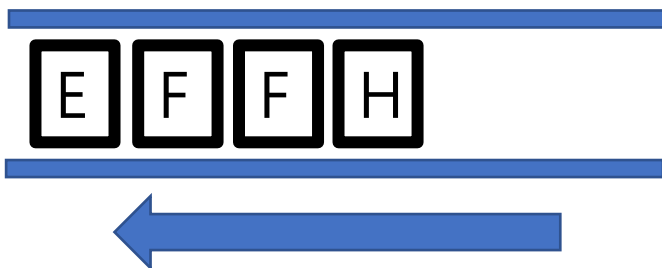


### 03. BFS (너비 우선 탐색)

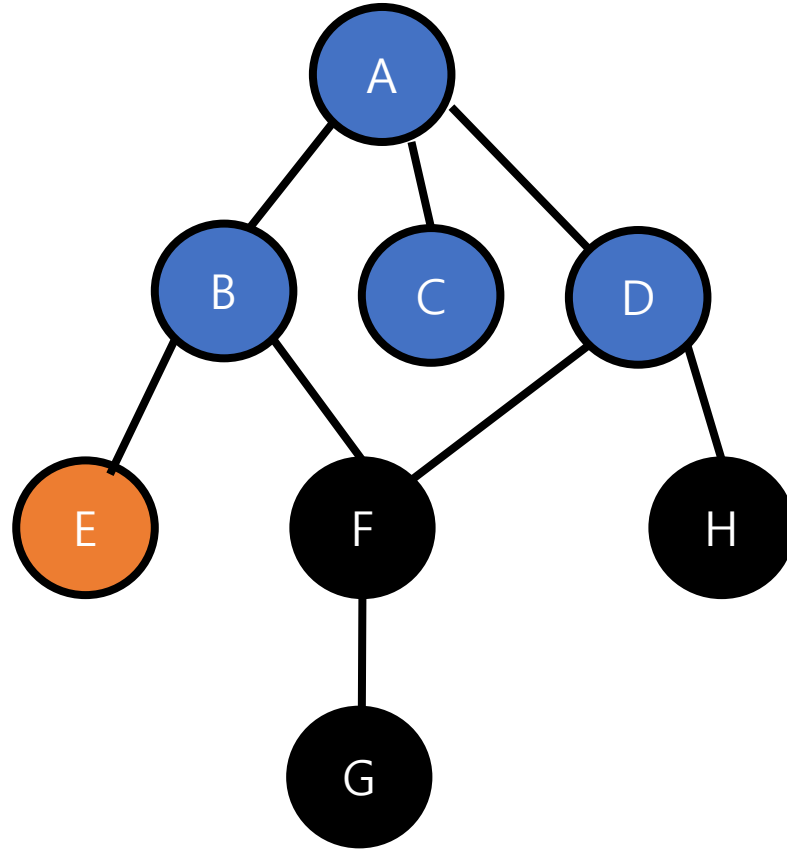


거리	
0	
1	
2	E F F H
3	
결과	A B C D

큐에서 pop된 D노드 방문  
D노드와 이웃한 방문하지 않은 노드 F, H push

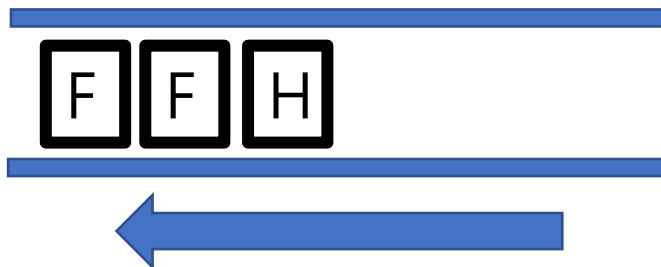


### 03. BFS (너비 우선 탐색)

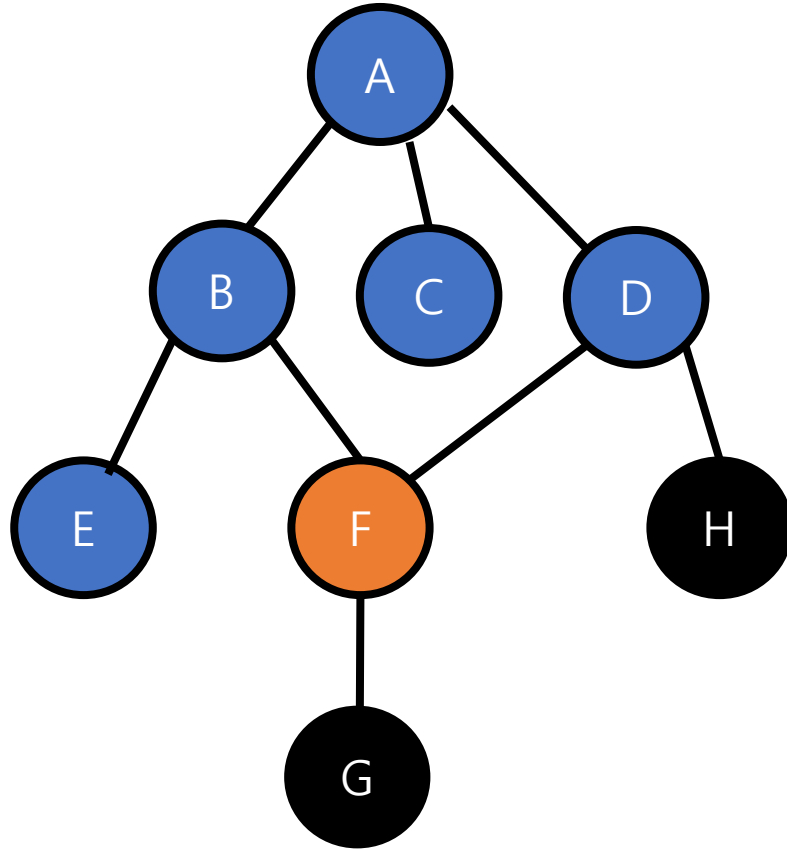


거리	
0	
1	
2	F F H
3	
결과	A B C D E

큐에서 pop된 E 노드 방문  
E노드와 이웃한 방문하지 않은 노드 없음

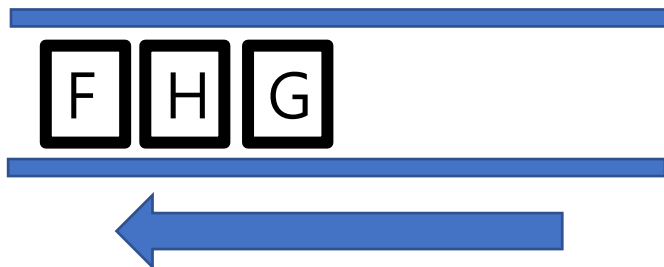


### 03. BFS (너비 우선 탐색)

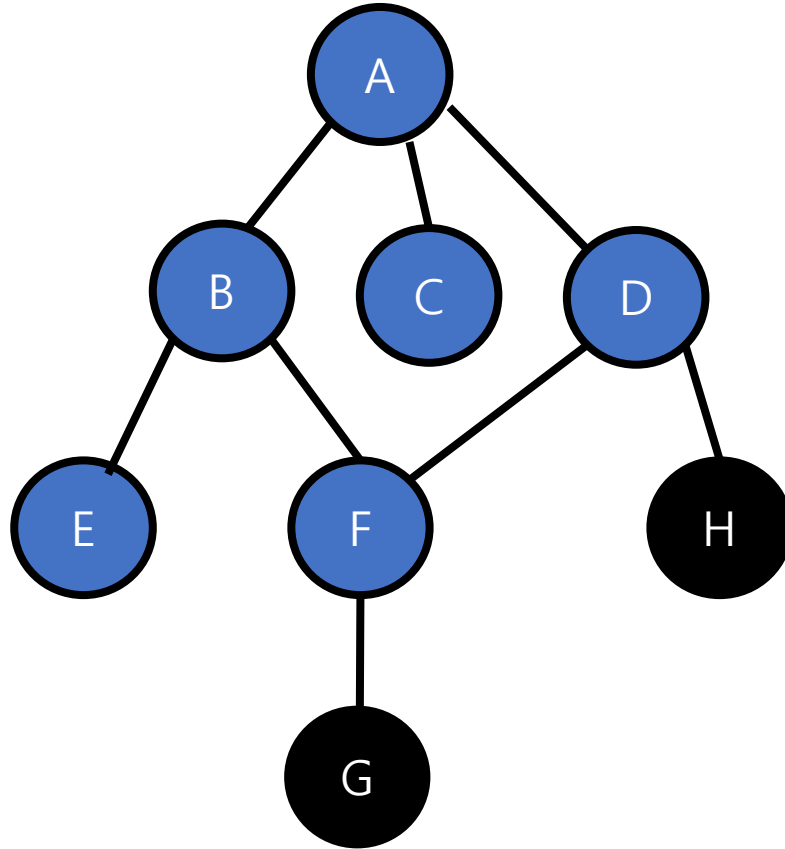


거리	
0	
1	
2	F H
3	G
결과	A B C D E F

큐에서 pop된 F 노드 방문  
E노드와 이웃한 방문하지 않은 노드 G push

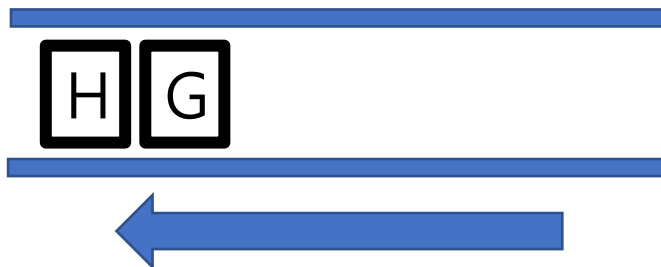


### 03. BFS (너비 우선 탐색)



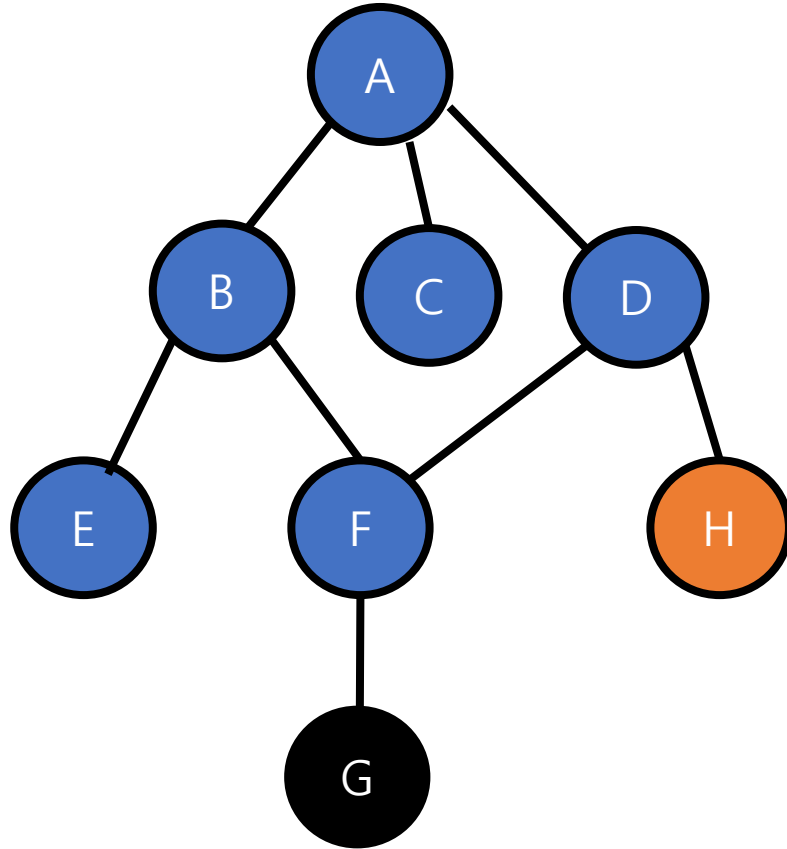
거리	
0	
1	
2	H
3	G
결과	A B C D E F

큐에서 pop된 F 노드 방문 X!  
이미 F노드를 방문하였기 때문





### 03. BFS (너비 우선 탐색)

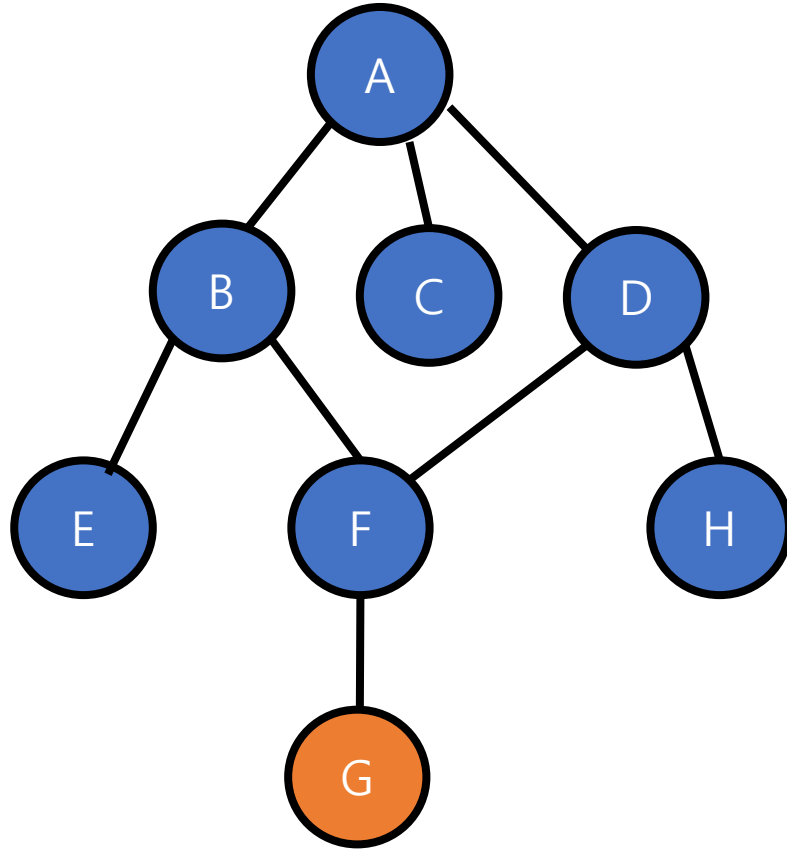


거리	
0	
1	
2	
3	G
결과	A B C D E F H

큐에서 pop된 H 노드 방문



### 03. BFS (너비 우선 탐색)



거리	
0	
1	
2	
3	
결과	A B C D E F H G



큐에서 pop된 H 노드 방문  
H노드와 이웃한 방문하지 않은 노드 없음  
큐에서 pop된 G 노드 방문  
G노드와 이웃한 방문하지 않은 노드 없음

큐에 노드가 없으므로 탐색 종료

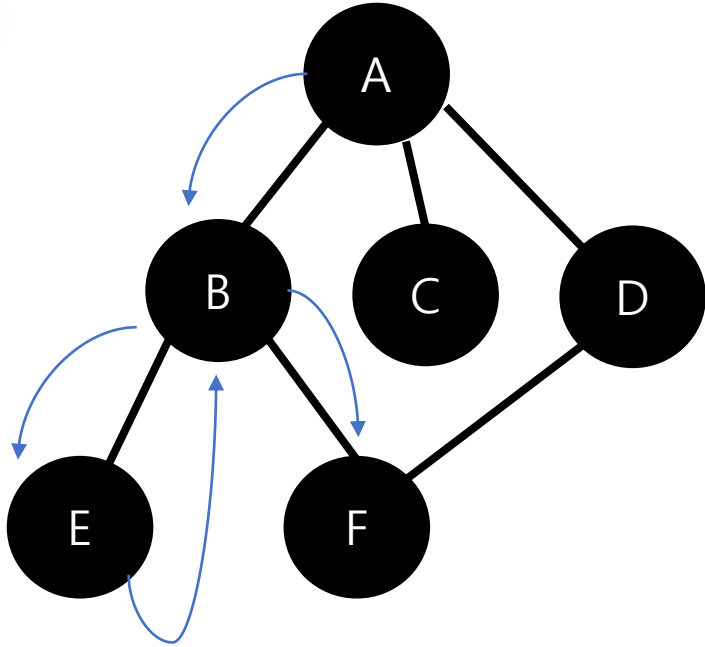
## 03. BFS

```
from collections import deque
T = int(input())
for _ in range(T):
    N,M = map(int,input().split())
    List = [[] for _ in range(N)]
    for i in range(M):
        u,v = map(int,input().split())
        List[u].append(v)
    for i in range(N):
        List[i].sort()

    Check = [False]*N # 방문했는지 안 했는지 확인
    Queue = deque([0]) # 큐
    while Queue:
        v = Queue.popleft() # 큐에서 나오는 v
        if Check[v] == True: # v가 이 시점에서 방문했다면
            continue
        Check[v] = True
        print(v,end=" ")
        for i in List[v]: # v와 연결된 모든 정점 중
            if Check[i] == False: # 아직 가지 않은 i가 있다면
                Queue.append(i) # 큐에 push

    print("")
```

## 04. DFS (깊이 우선 탐색)



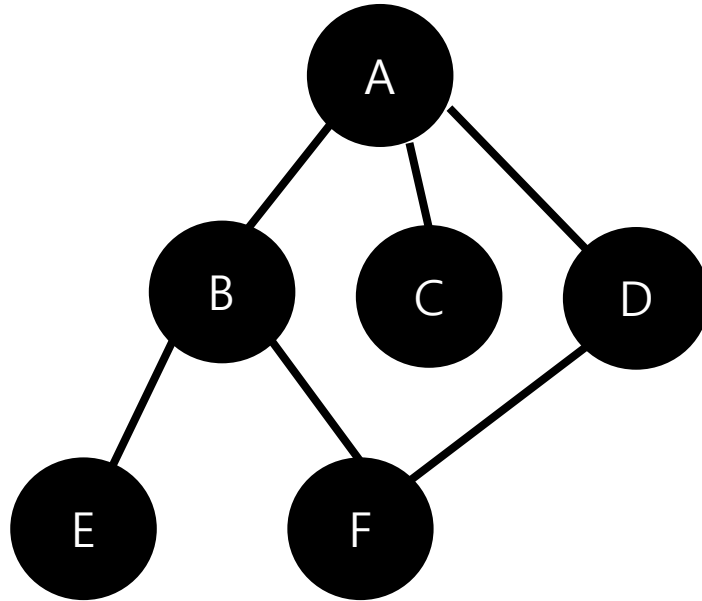
Depth-first search (DFS) 는 시작 노드에서 하나의 분기를 완벽하게 탐색한 후 다음 분기로 넘어가는 방법

하나의 경로  $(u, v)$ 에 대해서:

1.  $u$  노드를 방문
2.  $u$  노드와 연결된 노드 중  $v$  방문
3.  $v$  노드를 시작으로 하는 다른 경로의 노드 방문
4.  $v$  노드의 분기를 전부 탐색했다면, 다시  $u$ 에 인접한 노드 중 방문하지 않은 이웃 노드들 방문

- 재귀 함수를 사용하는 방법
- 명시적인 stack을 사용하는 방법

## 04. DFS (재귀)



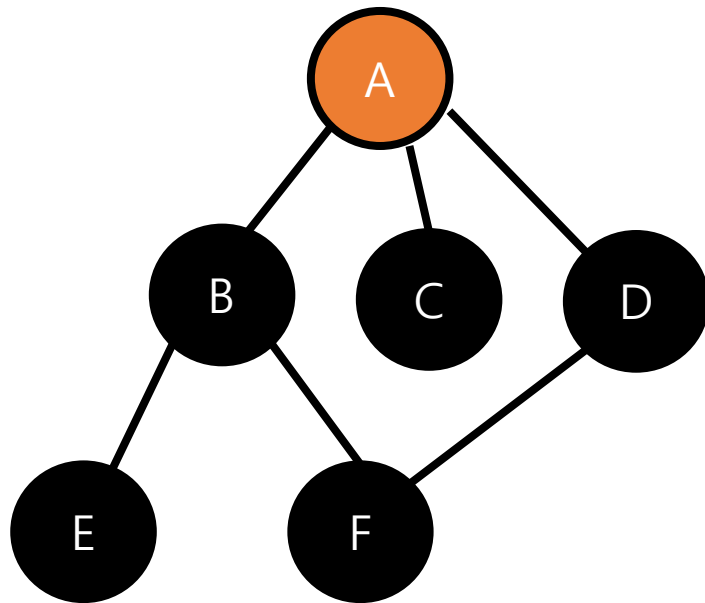
A	

DFS(A)

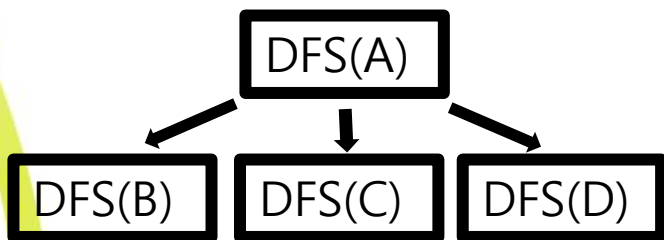
재귀 함수를 이용한 구현

각각의 함수에서 시작 노드를 매개변수로  
재귀적인 탐색을 진행한다.  
초기엔 시작 노드인 A

## 04. DFS (재귀)



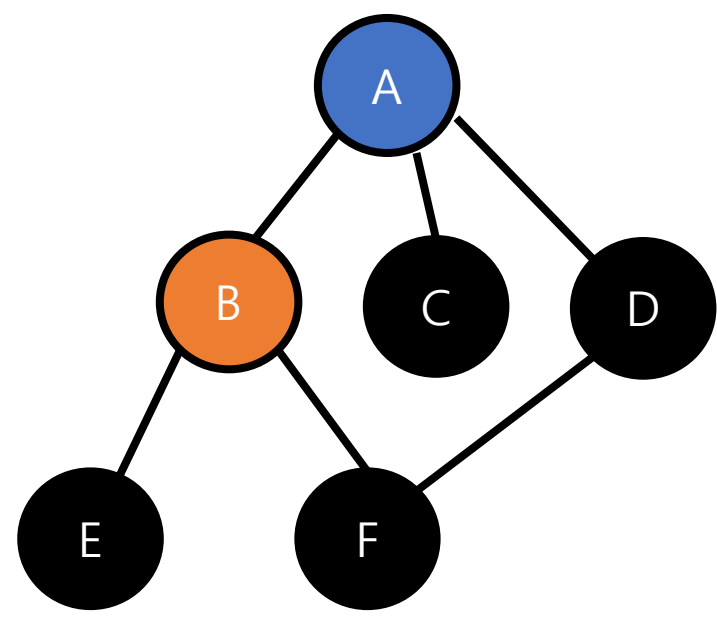
A	B	C	D



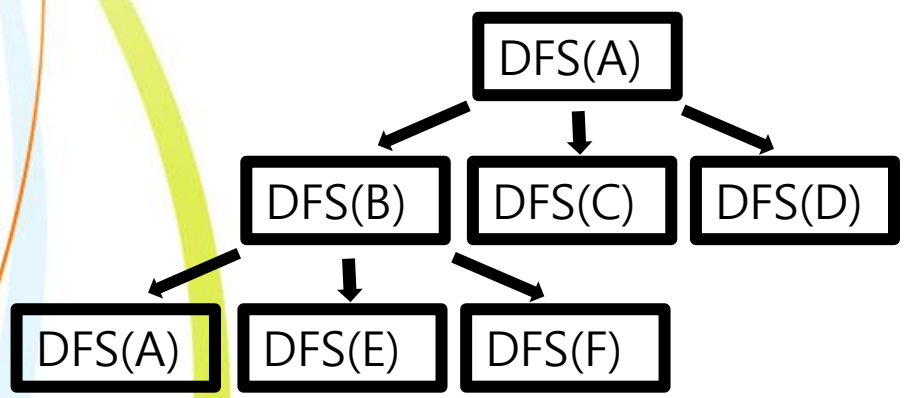
재귀 함수를 이용한 구현

A노드에 인접한 노드들에 대해 재귀호출  
A노드는 방문했음을 저장

# 04. DFS (재귀)



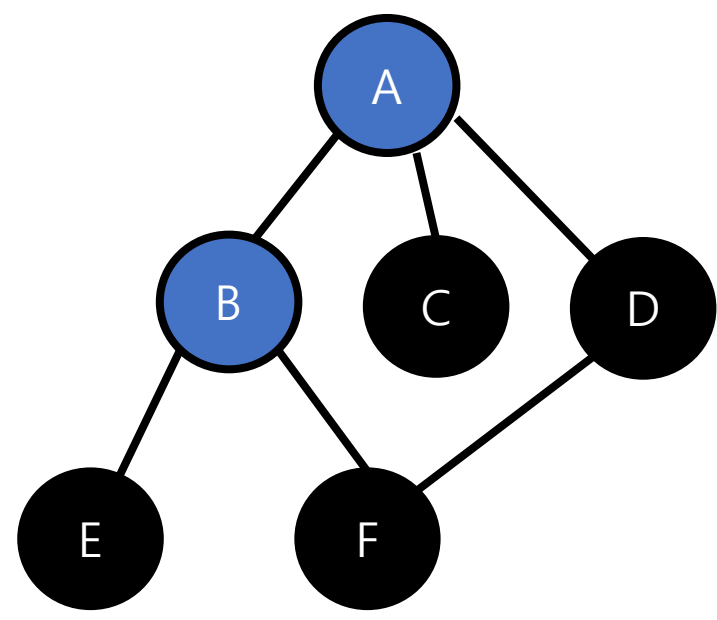
A	B C D
B	A E F



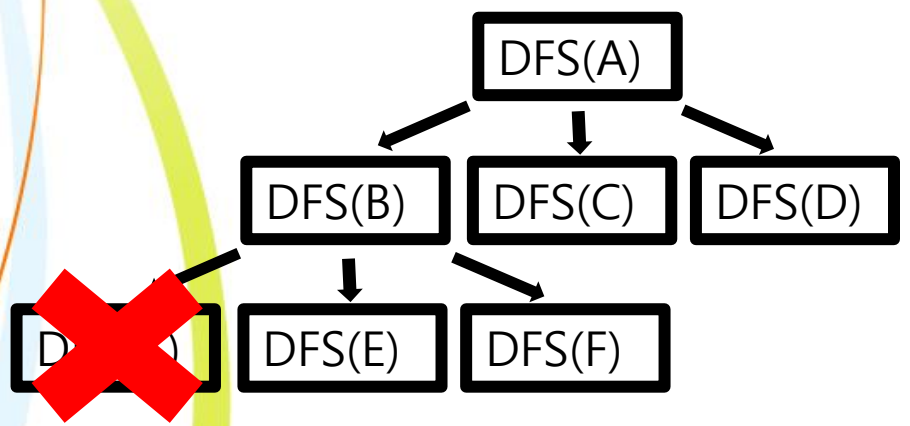
재귀 함수를 이용한 구현

현재 호출된 DFS(B)의 시작 노드는 B  
B노드에 인접한 노드들에 대해 재귀호출  
B노드는 방문했음을 저장

# 04. DFS (재귀)



A	B C D
B	E F

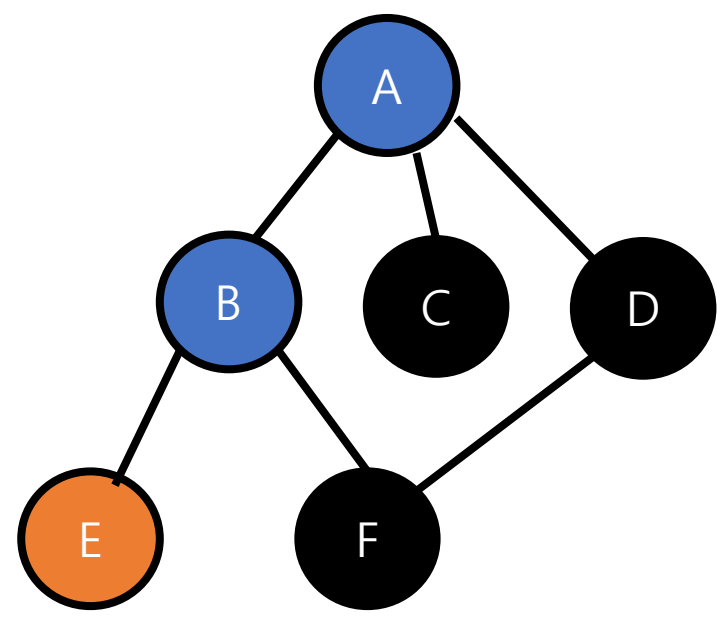


재귀 함수를 이용한 구현

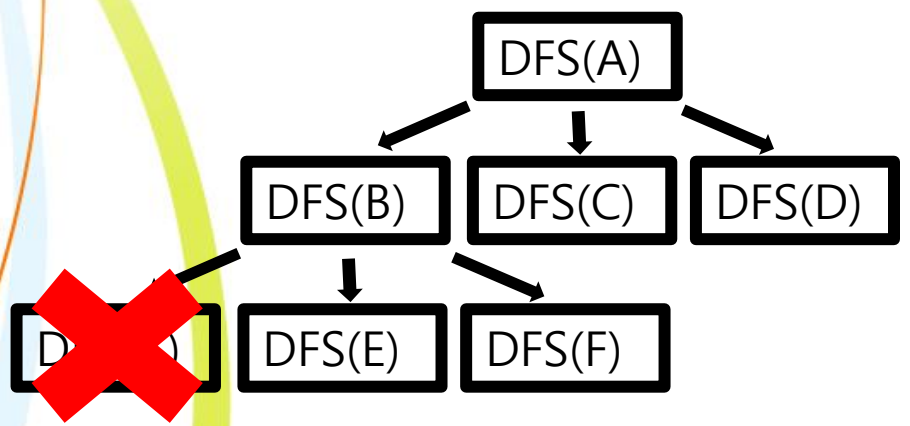
현재 호출된 DFS(B)의 시작 노드는 B  
여기서 방문한 것으로 표시된 A노드에 대  
해선 재귀호출을 안함



# 04. DFS (재귀)



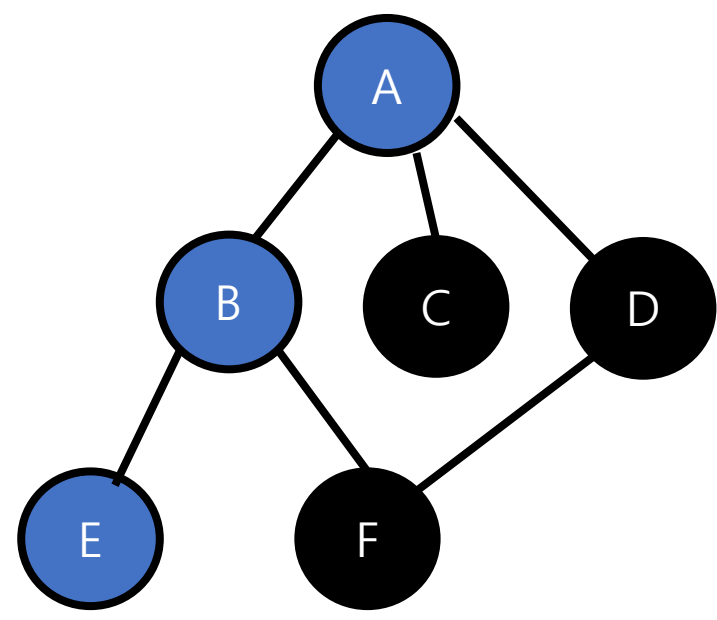
A	B C D
B	E F
E	B



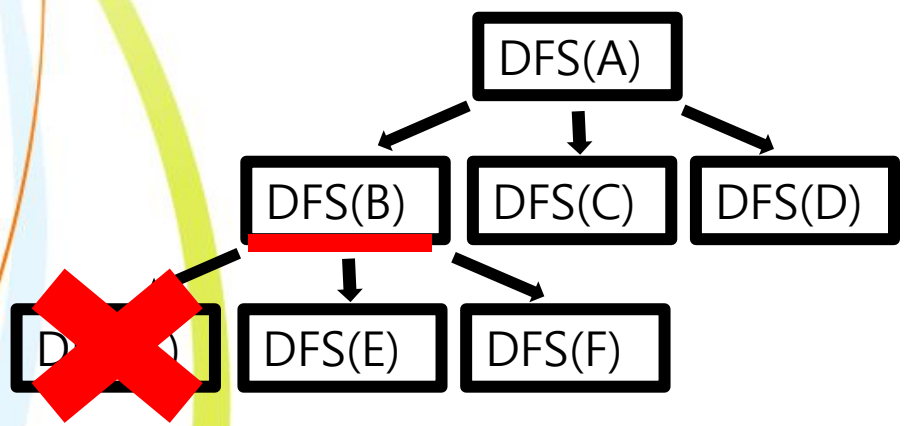
재귀 함수를 이용한 구현

현재 호출된 DFS(E)의 시작 노드는 E  
E노드에 인접한 노드들에 대해 재귀호출  
E노드는 방문했음을 저장

# 04. DFS (재귀)



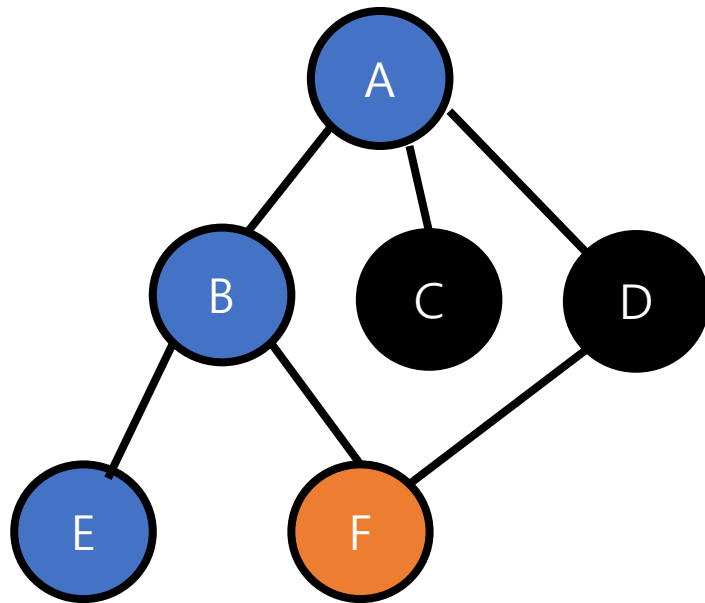
A	B	C	D
B	F		
E			



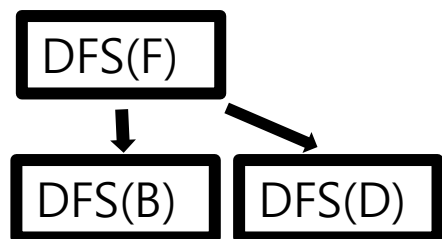
재귀 함수를 이용한 구현

현재 호출된 DFS(E)의 시작 노드는 E  
E노드에 인접한 방문되지 않은 노드는 없  
기 때문에 B노드로 돌아감

## 04. DFS (재귀)



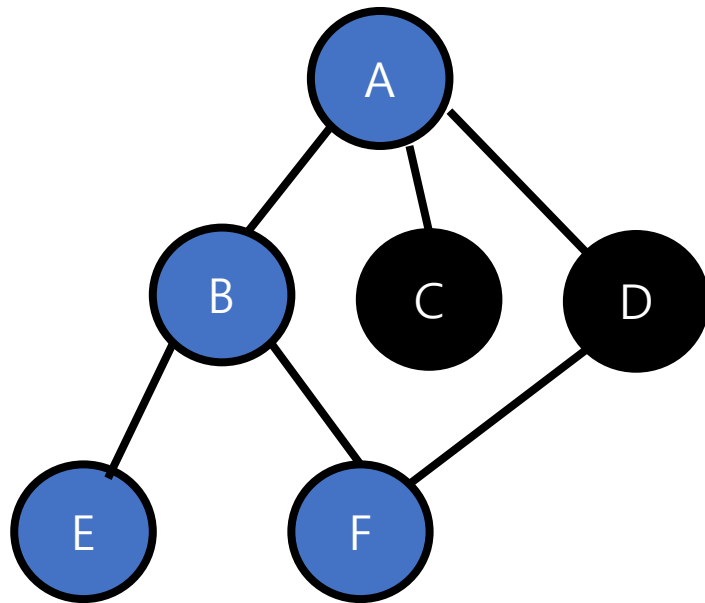
A	B C D
B	F
E	
F	B D



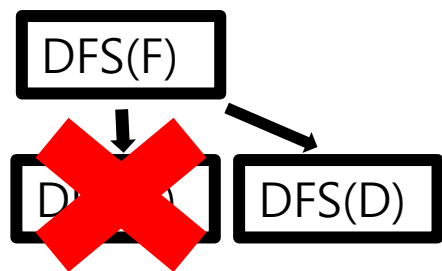
재귀 함수를 이용한 구현

B와 이웃한 다른 노드인 F를 시작점으로 하는 DFS(F)

## 04. DFS (재귀)



A	B	C	D
B	F		
E			
F	D		

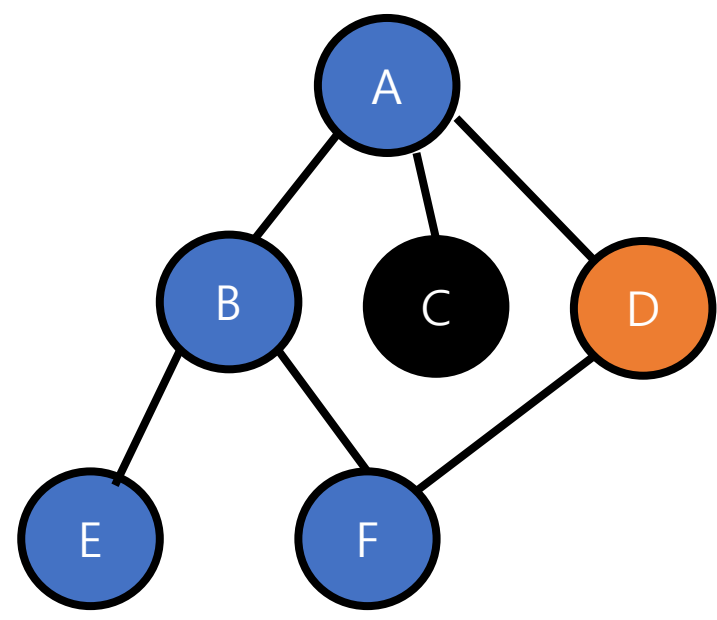


재귀 함수를 이용한 구현

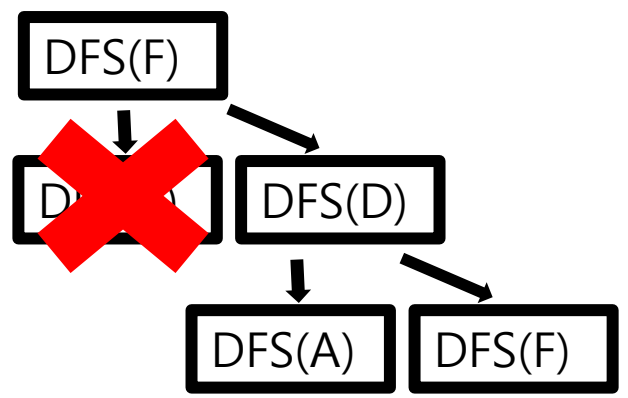
B와 이웃한 다른 노드인 F를 시작점으로  
하는 DFS(F)

이미 방문한 B노드는 호출하지 않음

# 04. DFS (재귀)



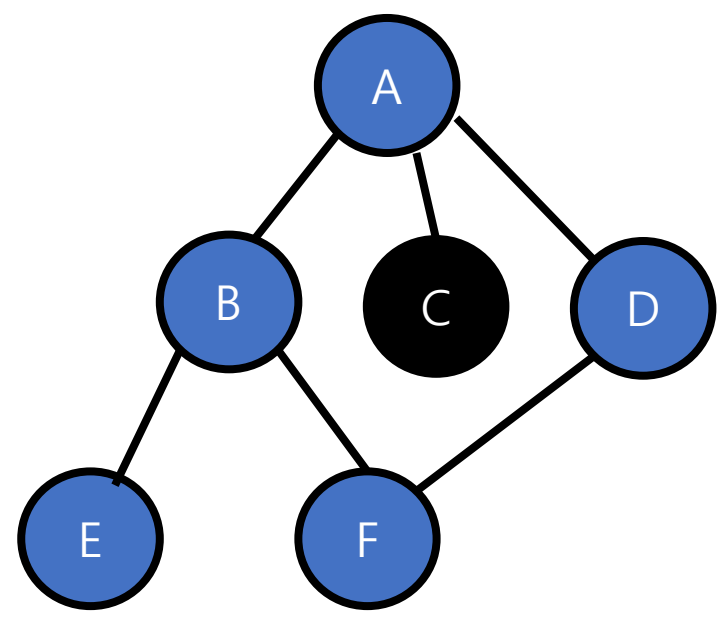
A	B C D
B	F
E	
F	D
D	A F



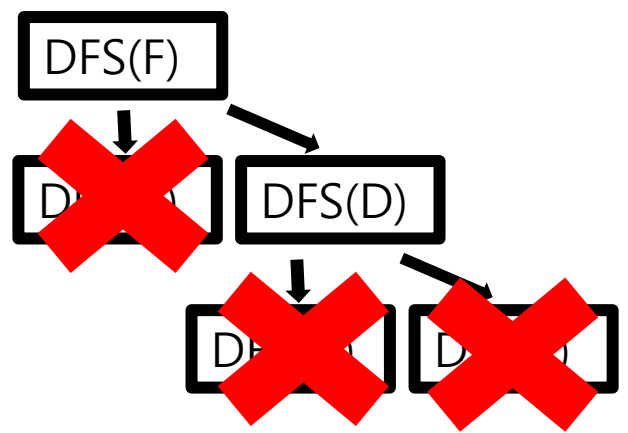
재귀 함수를 이용한 구현

현재 호출된 DFS(D)의 시작 노드는 D  
D노드에 인접한 노드들에 대해 재귀호출  
D노드는 방문했음을 저장

# 04. DFS (재귀)



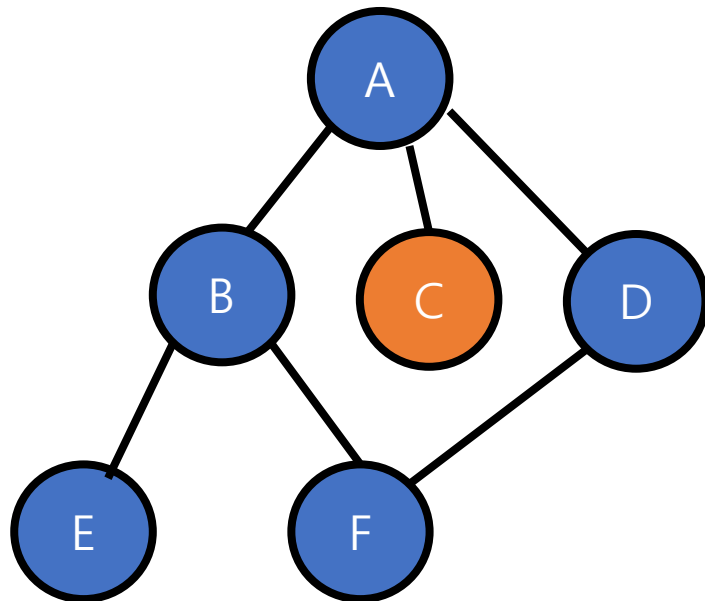
A	C D
B	
E	
F	
D	



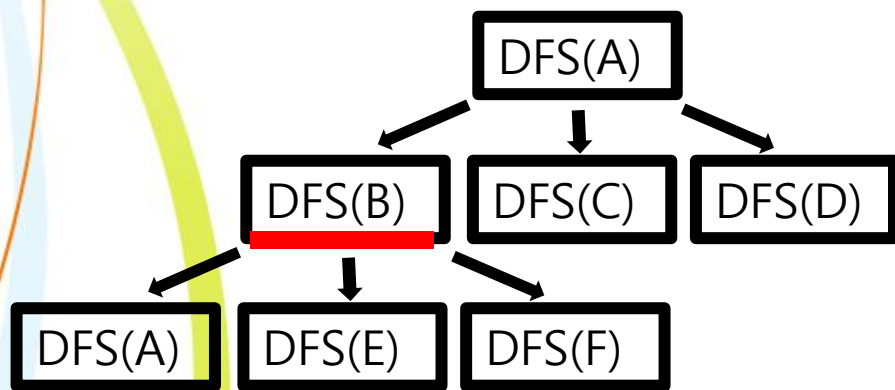
재귀 함수를 이용한 구현

D와 이웃한 다른 노드는 전부 방문하였음  
B → F 인 경로가 완전히 탐색되어 A노드  
에서 호출하였던 C와 D노드 탐색

## 04. DFS (재귀)



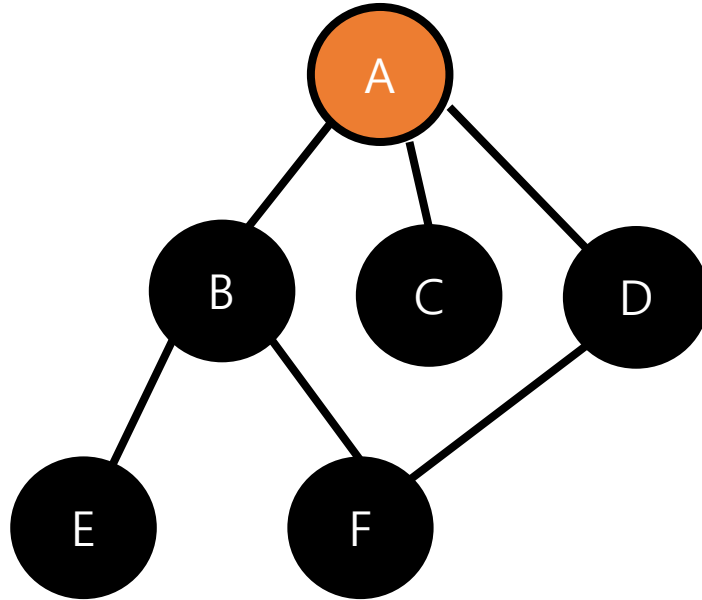
A	C D
B	
E	
F	
D	
C	A



재귀 함수를 이용한 구현

D노드는 이미 방문되었음  
 현재 호출된 DFS(C)의 시작 노드는 C  
 C노드에 인접한 노드들에 대해 재귀호출  
 C노드는 방문했음을 저장  
 모든 노드를 방문하였기 때문에 종료

## 04. DFS (스택)



A	D C B
st	D C B

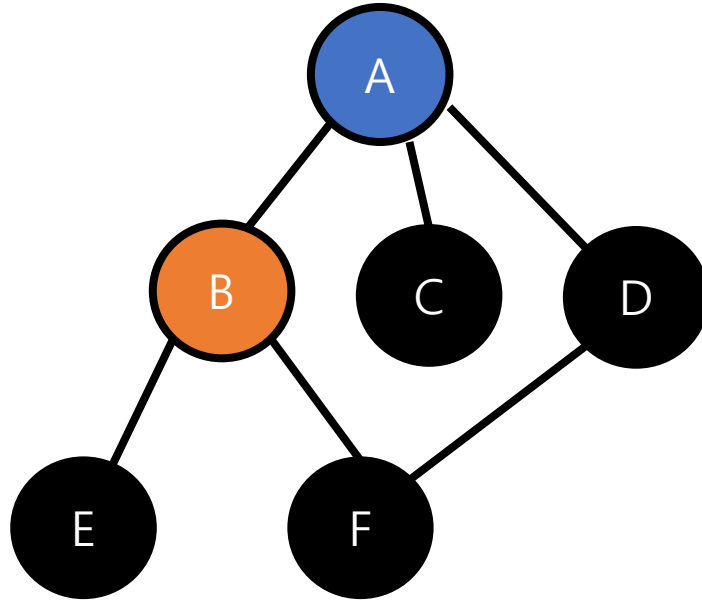
스택을 이용한 구현  
묵시적으로 스택을 통해 구현되는 재귀의 방식을,  
직접 스택을 만들어 명시적으로 탐색을 진행

초기 스택에 A노드의 이웃 (D, C, B) 넣음  
A노드를 방문으로 저장





## 04. DFS (스택)



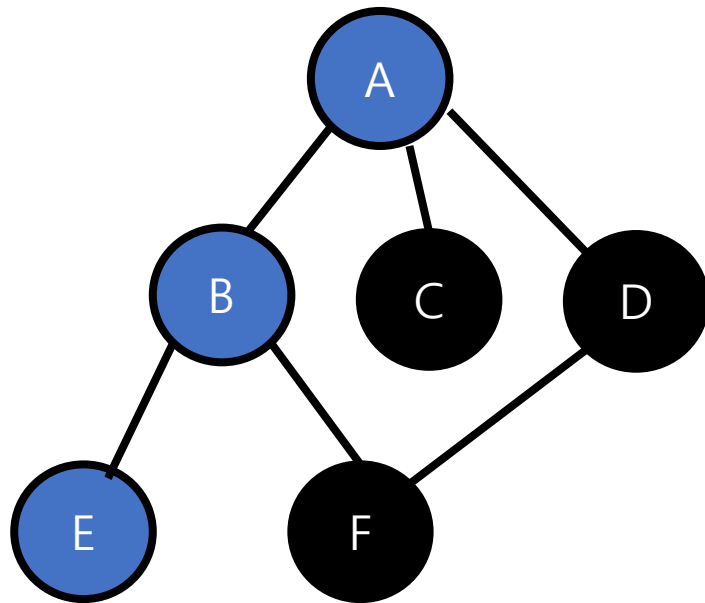
A	
B	F E
st	D C F E

스택을 이용한 구현  
스택에서 pop된 B노드 방문  
B노드를 방문으로 저장  
B노드와 이웃한 F, E노드 스택에 push

(값이 작은 노드를 먼저 탐색하기 위해  
push순서를 변경할 수 있음)



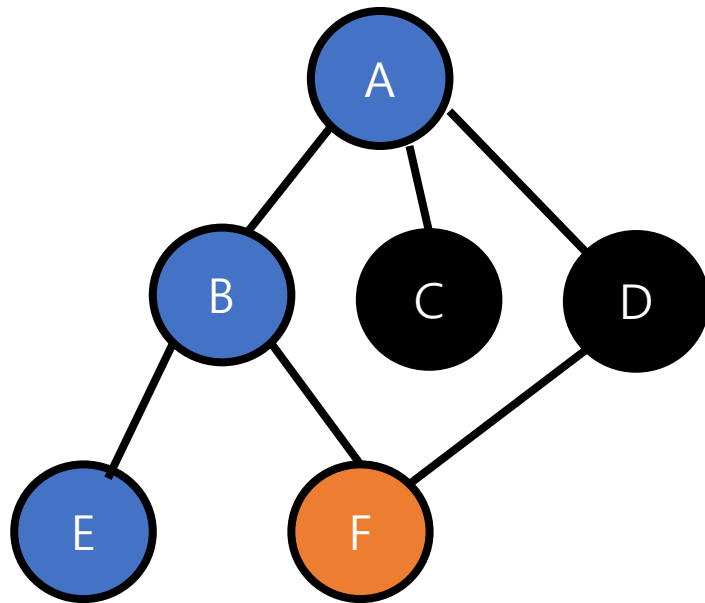
## 04. DFS (스택)



A	
B	
E	
st	D C F

스택을 이용한 구현  
스택에서 pop된 E노드 방문  
E노드를 방문으로 저장  
E노드와 이웃한 방문되지 않은 노드 없음

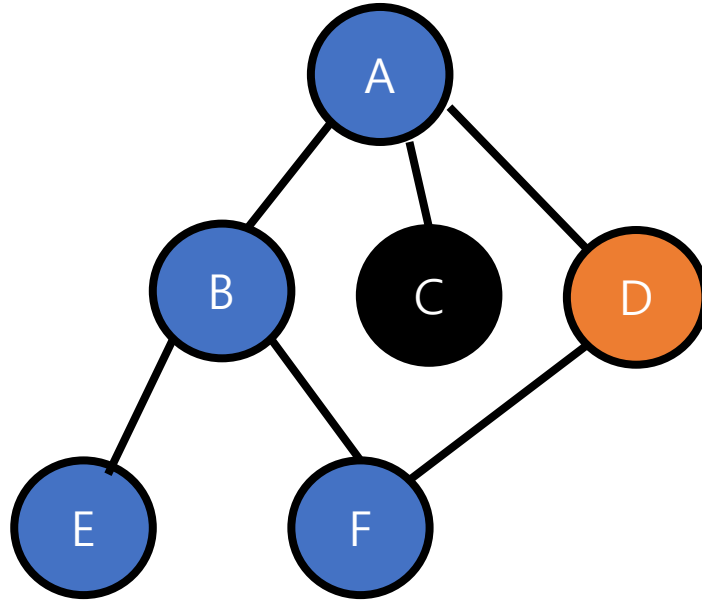
## 04. DFS (스택)



A	
B	
E	
F	D
st	D C D

스택을 이용한 구현  
스택에서 pop된 F노드 방문  
F노드를 방문으로 저장  
F노드와 이웃한 방문되지 않은 노드 D push

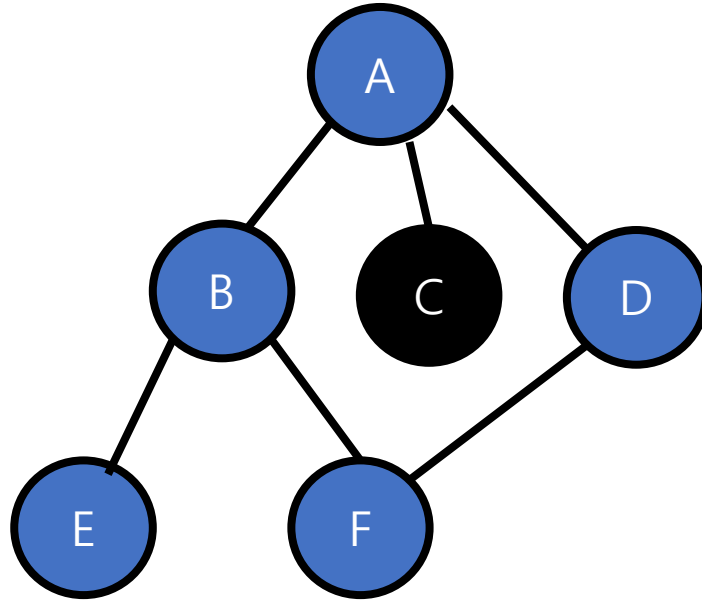
## 04. DFS (스택)



A	
B	
E	
F	
D	
st	D C

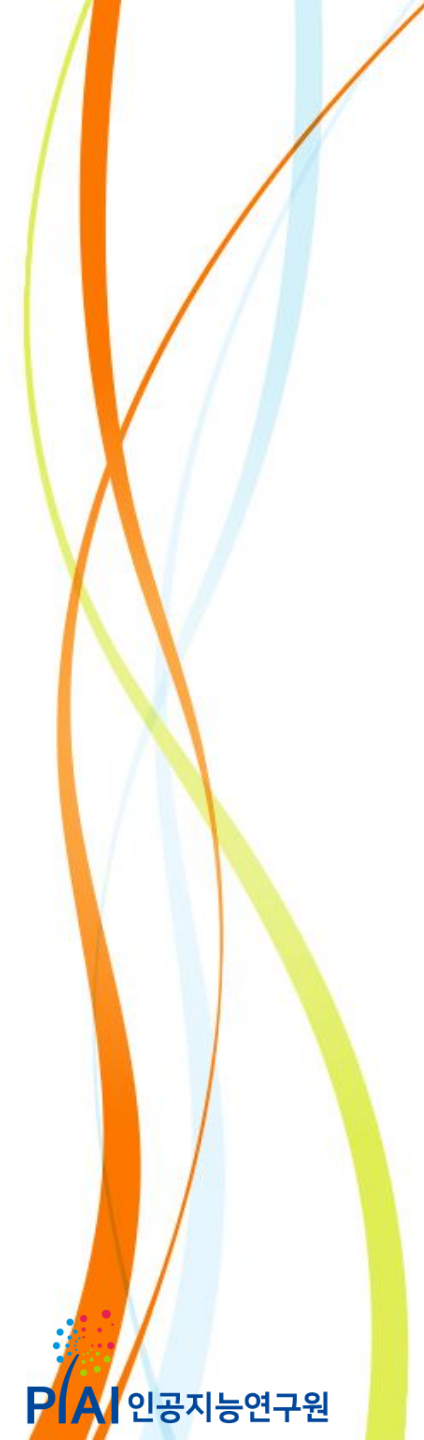
스택을 이용한 구현  
스택에서 pop된 D노드 방문  
D노드를 방문으로 저장  
D노드와 이웃한 방문되지 않은 node없음

## 04. DFS (스택)

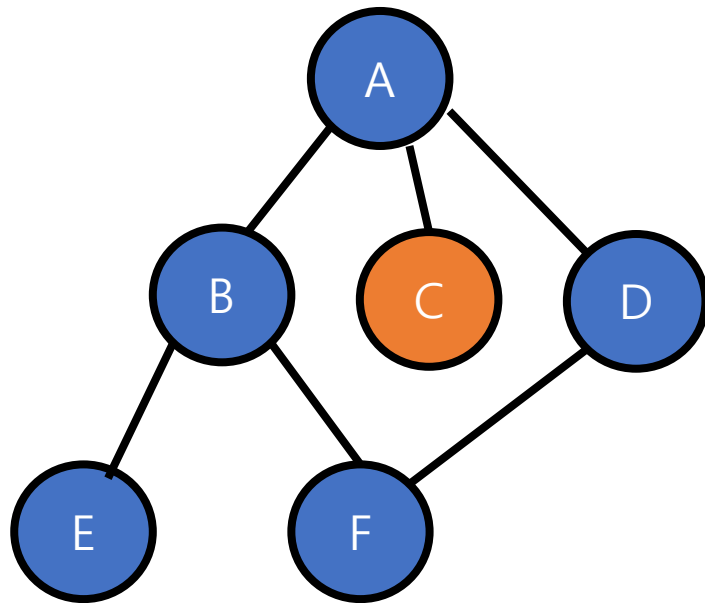


A	
B	
E	
F	
D	
C	
st	D

스택을 이용한 구현  
스택에서 pop된 C노드 방문  
C노드를 방문으로 저장  
C노드와 이웃한 방문되지 않은 node없음



## 04. DFS (스택)



A	
B	
E	
F	
D	
C	
st	

그 다음 스택에서 pop된 D노드는 방문하였기 때문에, 탐색 종료

## 04. DFS (재귀)

```
import sys
sys.setrecursionlimit(1000000)

def DFS(v, List, Check): # v에서 DFS 시작
    print(v, end=" ")
    Check[v] = True # v는 방문했으므로 표시
    for i in List[v]: # v와 연결된 모든 정점 중에
        if Check[i] == False: # 아직 가지 않은 i가 있다면
            DFS(i, List, Check) # 바로 i에서 DFS를 다시 시작

t = int(input())
for _ in range(t):
    N, M = map(int, input().split())
    List = [[] for _ in range(N)]
    for i in range(M):
        u, v = map(int, input().split())
        List[u].append(v)
        List[v].append(u)
    for i in range(N):
        List[i].sort()

    Check = [False]*N # 방문여부 확인을 위한 리스트
    DFS(0, List, Check) # 0부터 DFS 시작
    print("")
```

## 04. DFS (스택)

```
t = int(input())
for _ in range(t):
    N,M = map(int,input().split())
    List = [[] for _ in range(N)]
    for i in range(M):
        u,v = map(int,input().split())
        List[u].append(v)
        List[v].append(u)
    for i in range(N):
        List[i].sort(reverse=True)

    Check = [False]*N # 방문여부 확인을 위한 리스트
    Stack = [0] # 스택 사용
    while Stack:
        v = Stack.pop()
        if Check[v] == True: # v가 이미 방문되었다면
            continue
        Check[v] = True
        print(v,end=" ")
        for i in List[v]: # v와 연결된 모든 정점 중에
            if Check[i] == False: # 아직 가지 않은 i가 있다면
                Stack.append(i) # 스택에 i push

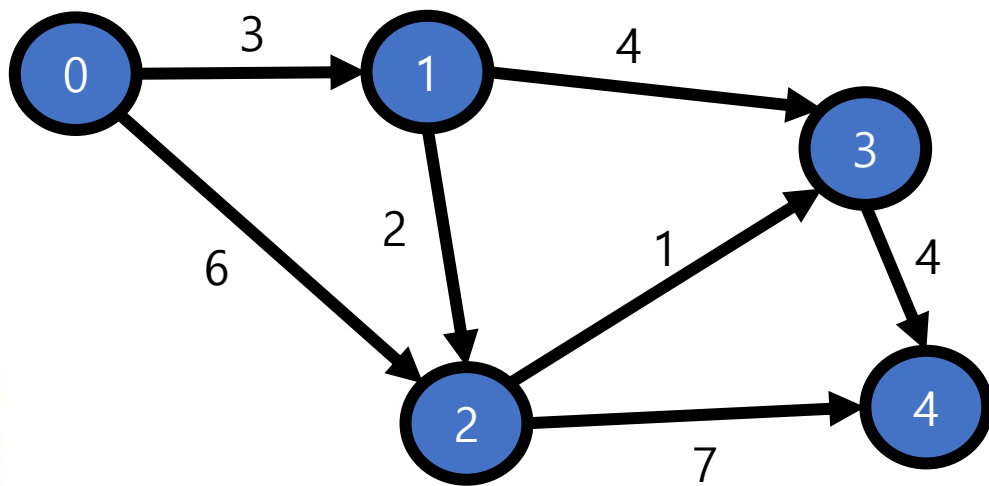
    print("")
```



## 05. 최단 경로 구하기 (Dijkstra)

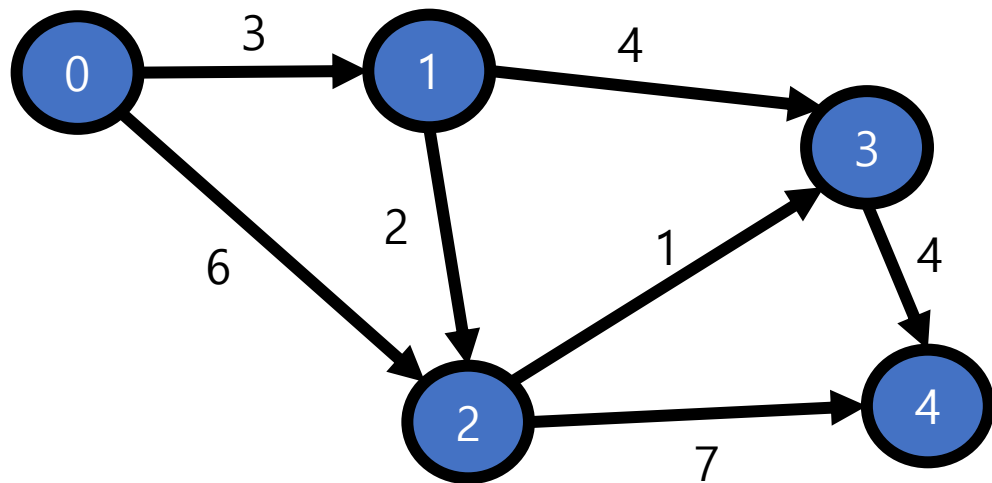
모든 간선의 가중치가 양수일 때, 최단 경로의 거리 값 찾기

※ 인접 행렬과 인접 리스트 두 형태로 모두 구현이 가능합니다.



0	1	2	3	4
0	3	5	6	10

## 05. 최단 경로 구하기 (Dijkstra)



Priority Queue(heapq)를 이용한 구현 vs 배열만 써서 구현

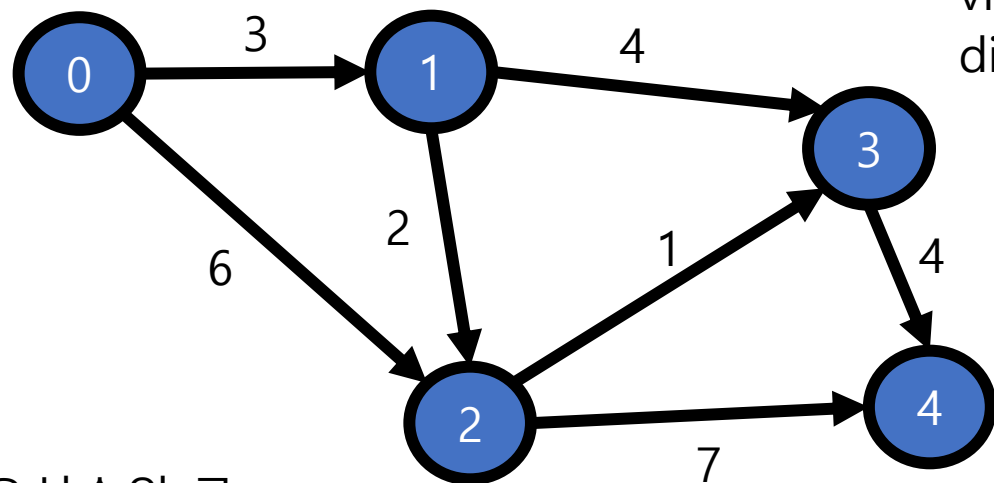
$$O(|E| \log |V|)$$

$$O(|V|^2)$$

※주의! heapq에는 decreasekey operation이 없습니다.

⇒ 각 정점의 거리가 확정되었는지 체크하고, 이미 확정된 정점이라면 무시!

## 05. 최단 경로 구하기 (Dijkstra)



visited=[False, False, False, False, False]  
dist=[-1, -1, -1, -1, -1]

우선순위 큐

(0, 0)

hq: 우선순위 큐

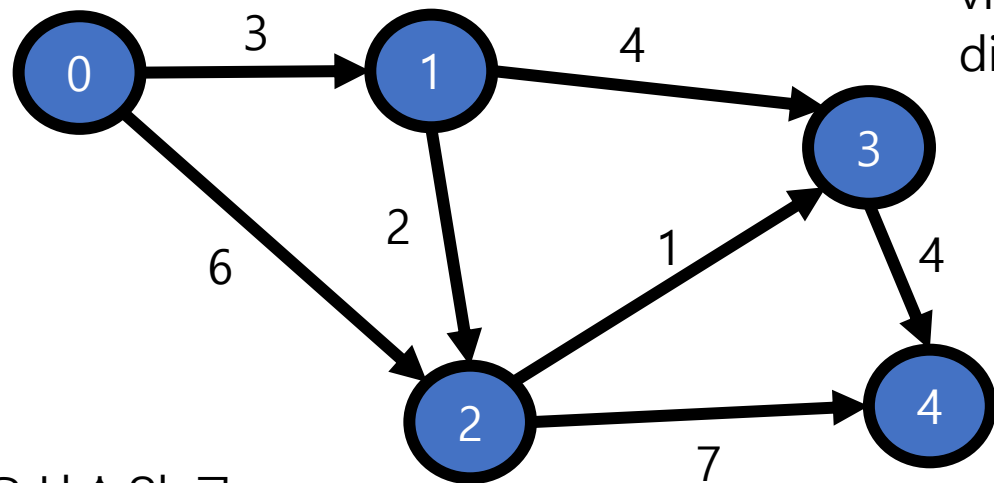
hq에 들어가는 원소: 숫자 튜플 (d, u)

- 0에서 u까지 d인 경로가 존재합니다.
- 우선순위: d 값이 가장 작은 튜플

최초의 hq: (0, 0)

- 0번 정점은 거리 0으로 0번에 도착할 수 있습니다.
- 그리고 이보다 더 좋은 경로는 없습니다. (곧 확정)

## 05. 최단 경로 구하기 (Dijkstra)



visited=[**True**, False, False, False, False]  
dist=[**0**, -1, -1, -1, -1]

우선순위 큐

(3, 1)

(6, 2)

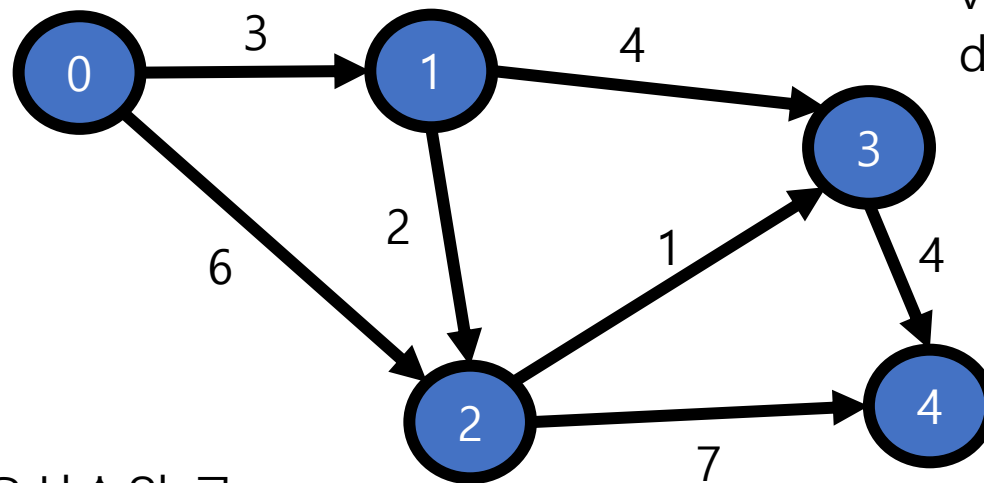
(0, 0)이 pop: 0에서 0으로 가는 최단 경로는 확정

visited[0]을 True로 바꿉니다. (확정 체크)

dist[0]을 0으로 저장: 확정된 최단 경로 길이 저장

0에서 방문할 수 있는 정점들의 정보를 hq에 push합니다. (0까지 온 경로 + 0과 정점 사이의 간선)

## 05. 최단 경로 구하기 (Dijkstra)



visited=[True, **True**, False, False, False]  
dist=[0, **3**, -1, -1, -1]

우선순위 큐

(5, 2)  
(7, 3)  
(6, 2)

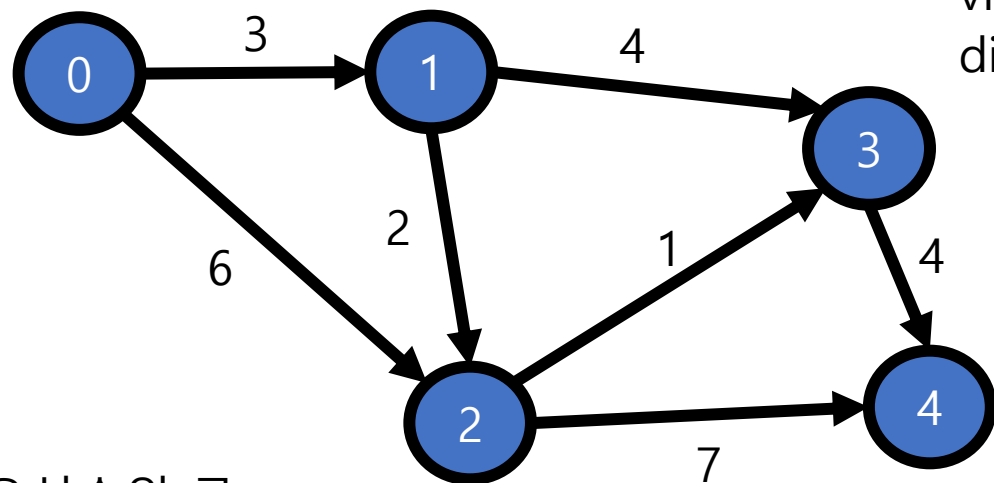
(3, 1)이 pop: 0에서 1으로 가는 최단 경로는 확정

visited[1] = True

dist[1] = 3: 확정된 최단 경로 길이

1에서 방문할 수 있는 정점들의 정보를 hq에 push합니다. (1까지 온 경로 + 1과 정점 사이의 간선)

## 05. 최단 경로 구하기 (Dijkstra)



visited=[True, True, **True**, False, False]  
dist=[0, 3, **5**, -1, -1]

우선순위 큐

**(6, 3)**

~~(7, 3)~~

(6, 2)

**(12, 4)**

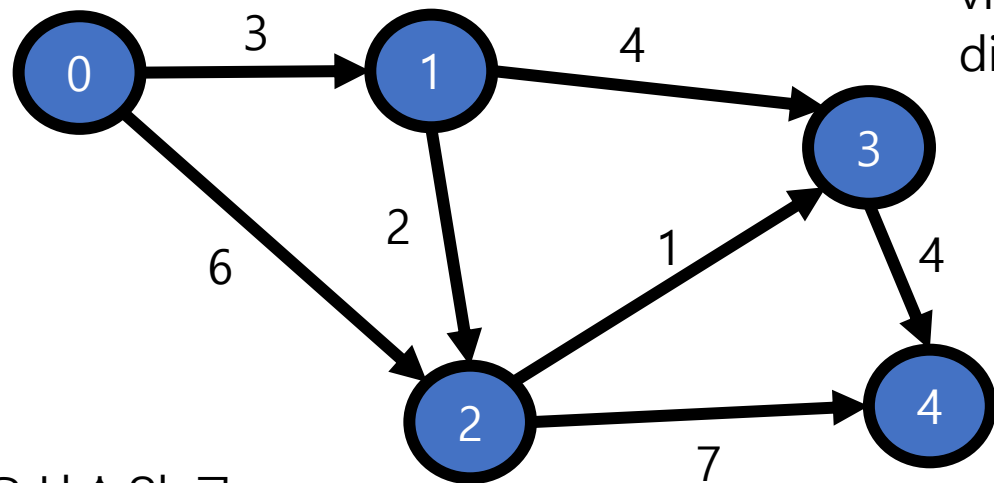
(5, 2)가 pop: 0에서 2으로 가는 최단 경로는 확정

visited[2] = True

dist[2] = 5: 확정된 최단 경로 길이

2에서 방문할 수 있는 정점들의 정보를 hq에 push합니다. (2까지 온 경로 + 2와 정점 사이의 간선)

## 05. 최단 경로 구하기 (Dijkstra)



visited=[True, True, True, False, False]  
dist=[0, 3, 5, -1, -1]

우선순위 큐

(6, 3)

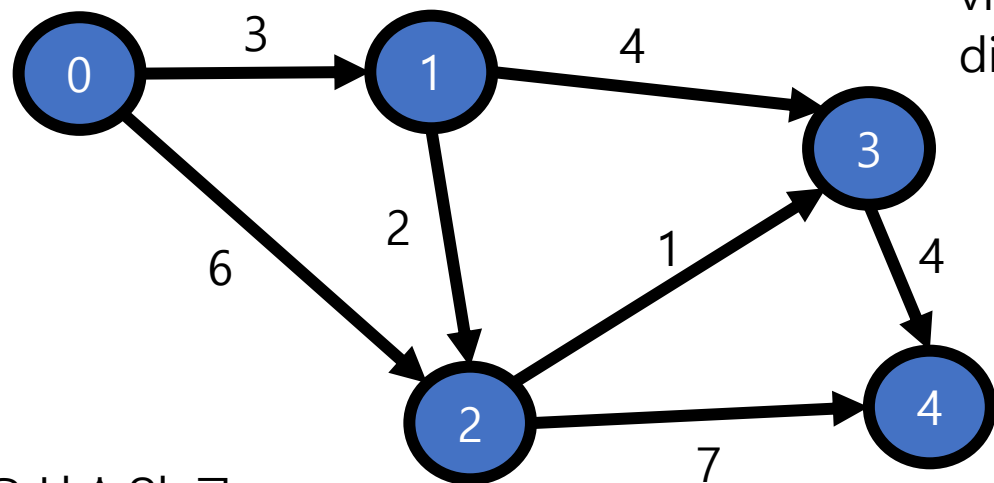
(12, 4)

(6, 2)가 pop

visited[2]는 이미 True

- 0에서 2로 가는 최단 경로는 이미 확정되었습니다.
- 이후에 나오는 경로가 5보다 작을 리 없음
- 그러므로 무시합니다.

## 05. 최단 경로 구하기 (Dijkstra)



visited=[True, True, True, **True**, False]  
dist=[0, 3, 5, **6**, -1]

우선순위 큐

**(10, 4)**

(12, 4)

(6, 3)이 pop: 0에서 3으로 가는 최단 경로는 확정

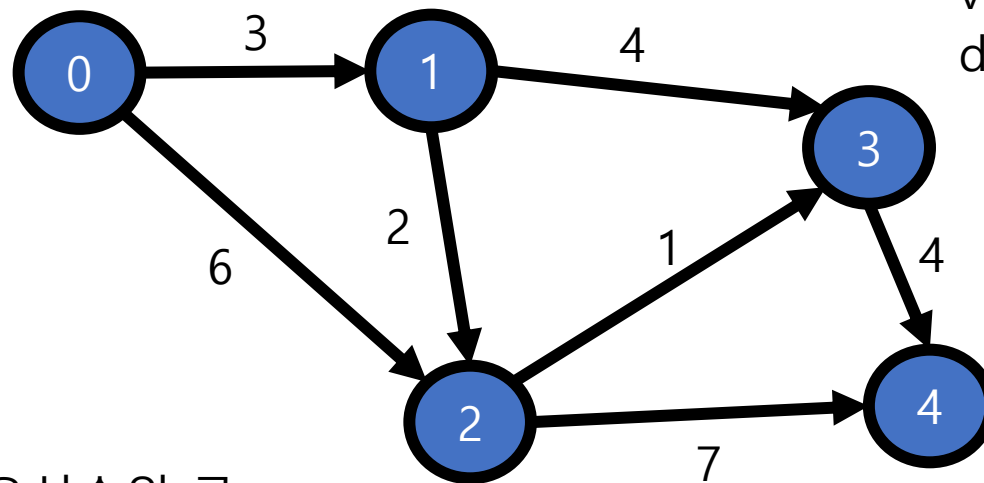
visited[3] = True

dist[3] = 6: 확정된 최단 경로 길이

3에서 방문할 수 있는 정점들의 정보를 hq에 push합니다. (3까지 온 경로 + 3과 정점 사이의 간선)



## 05. 최단 경로 구하기 (Dijkstra)



visited=[True, True, True, True, **True**]  
dist=[0, 3, 5, 6, **10**]

우선순위 큐

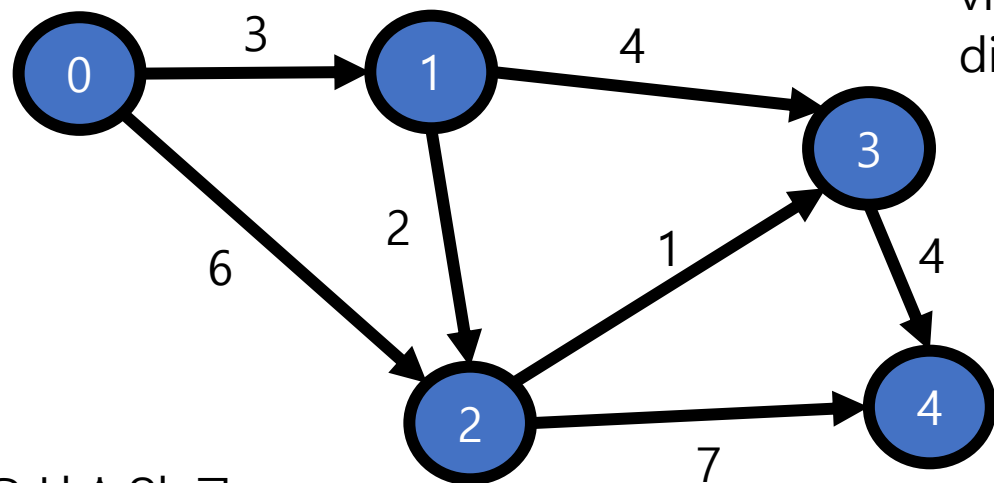
(12, 4)

(10, 4)이 pop: 0에서 4으로 가는 최단 경로는 확정

visited[4] = True

dist[4] = 10: 확정된 최단 경로 길이

## 05. 최단 경로 구하기 (Dijkstra)



visited=[True, True, True, True, **True**]  
dist=[0, 3, 5, 6, **10**]

우선순위 큐



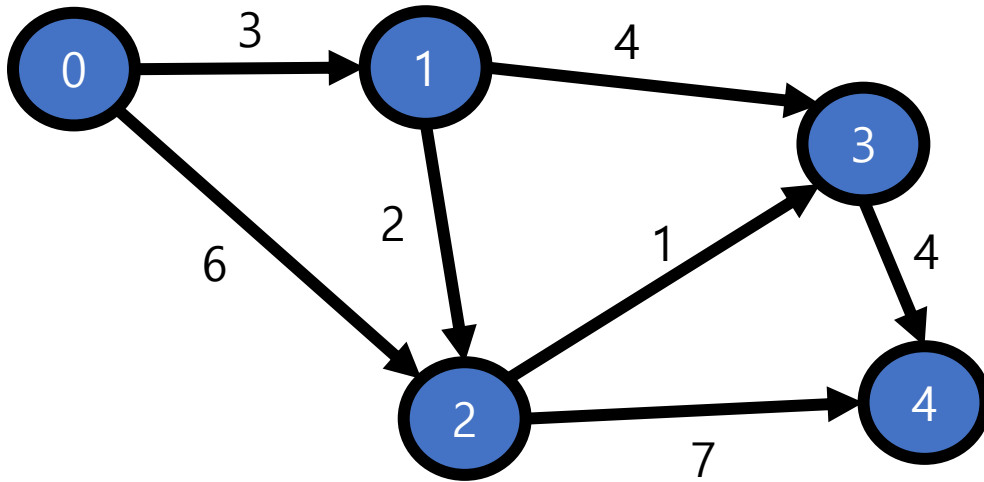
(12, 4)가 pop

visited[4]는 이미 True

- 0에서 4로 가는 최단 경로는 이미 확정되었습니다.
- 이후에 나오는 경로가 10보다 작을 리 없음
- 그러므로 무시합니다.

hq가 비었으므로 더 이상의 경우는 없습니다.

## 05. 최단 경로 구하기 (Dijkstra)



visited=[False, False, False, False, False]

dist=[0, ∞, ∞, ∞, ∞]

visited, dist 배열을 이용,  
visited가 False인 dist값 중 최소값을 지닌 정점을 확정  
이후 업데이트는 dist 배열에 바로 해줍니다.

0	1	2	3	4
0	?	?	?	?
0	1	2	3	4
0	3	6	?	?
0	1	2	3	4
0	3	5	7	?
0	1	2	3	4
0	3	5	6	12
0	1	2	3	4
0	3	5	6	10
0	1	2	3	4
0	3	5	6	10

## 05. 최단 경로 구하기 (Dijkstra)

```
import heapq

for t in range(int(input())):
    n, m = map(int, input().split())
    graph = [[] for _ in range(n)]           # 인접 리스트 정의
    for _ in range(m):
        u, v, c = map(int, input().split()) # 출발 정점, 도착 정점, 가중치 입력
        graph[u].append([v, c])             # 입력받은 정보를 인접리스트에 반영

    visited = [False]*n # 정점의 방문 여부를 저장하는 리스트
    dist = [-1]*n        # 0번 정점으로부터의 거리를 저장하는 리스트
    hq = []              # 우선순위 큐에 사용할 리스트 초기화
    heapq.heappush(hq, (0, 0)) # 0번 정점에서 0번 정점까지의 거리는 0, 0번 정점부터 시작 (순서 주의)
    while len(hq) > 0:
        d, u = heapq.heappop(hq)
        if visit[u] == False: # 현재 정점이 아직 방문되지 않았다면 진행
            visit[u] = True   # 현재 정점을 방문 처리
            dist[u] = d        # 0번 정점에서 현재 정점까지의 거리를 기록
            for i in graph[u]: # 현재 정점에서 출발하는 간선들에 대해 처리
                if visit[i[0]] == False: # 간선의 도착 정점이 방문되지 않았으면 거리 계산해서 heappush
                    heapq.heappush(hq, [d+i[1], i[0]])
    print(dist[-1]) # 목표 정점까지의 거리를 출력(도달 못할시에 -1 출력)
```

인접 리스트  
+  
우선 순위 큐

## 05. 최단 경로 구하기 (Dijkstra)

```
INF = float('inf') # 0에서 가는 경로가 아직 없음을 나타냄
T = int(input())
for _ in range(T):
    N,M = map(int, input().split())
    Matrix = [[0]*N for _ in range(N)]
    for i in range(M):
        u,v,c = map(int,input().split())
        Matrix[u][v] = c # 인접 행렬

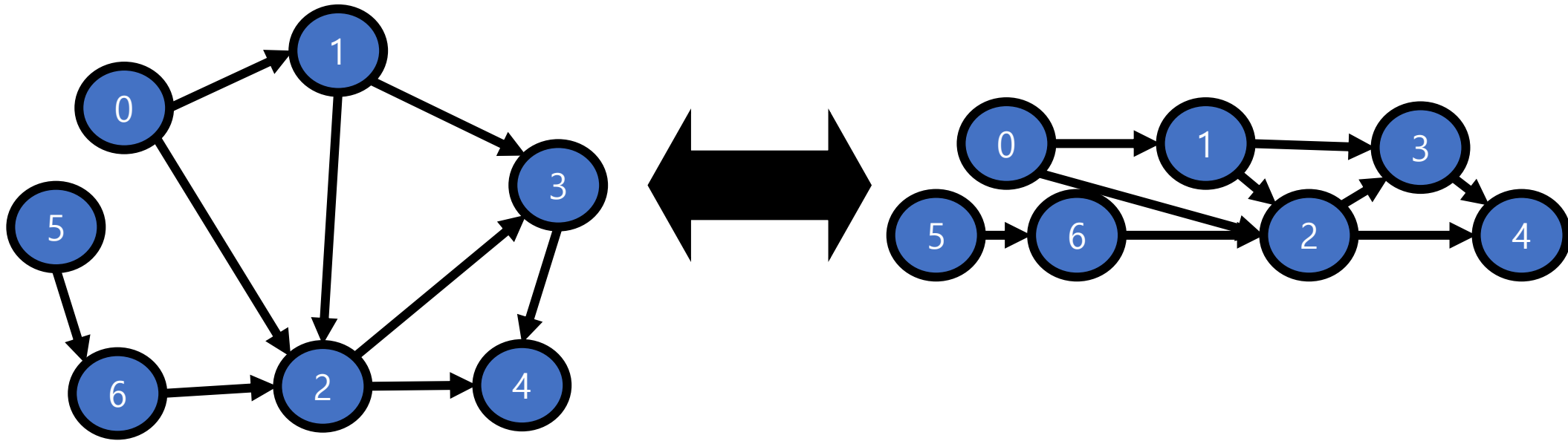
    D = [0]+[INF]*(N-1) # 0번 정점은 처음에 0번에서 0의 최솟값으로 이동
    Check = [False]*N
    for iteration in range(N):
        Min = INF
        for i in range(N): # 현재 확정되지 않은 경로 중 최솟값을 찾음
            if Check[i] == False and D[i] < Min:
                Min = D[i]
                idx = i # 그 최솟값을 지닌 정점을 저장
        if Min==INF: break
        Check[idx] = True # 그 정점까지의 최단 경로는 확정
        for i in range(N): # 그 정점에서 다른 정점까지 거리를 업데이트
            if Matrix[idx][i] != 0 and D[idx] + Matrix[idx][i] < D[i]:
                D[i] = D[idx] + Matrix[idx][i] # D에 바로 업데이트
    if Check[N-1] == False: # N-1번 정점까지 오는 경로가 없을 때
        print(-1)
    else:
        print(D[N-1])
```

인접 행렬  
+  
배열

# ADD01. 위상 정렬

비선형(Non-linear) 자료 구조인 그래프의 선형(Linear)화  
DAG(Directed Acyclic Graph)

- 방향성 그래프
- 사이클이 없음

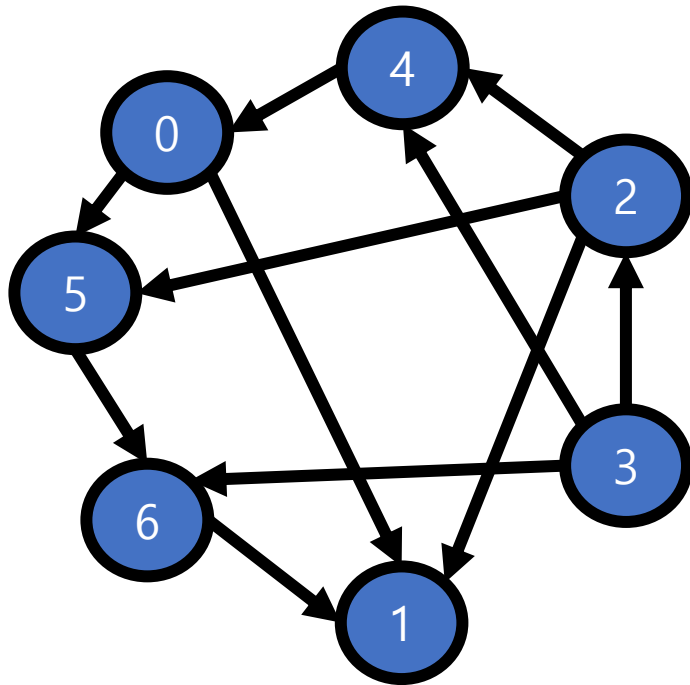


# ADD01. 위상 정렬

위상 정렬 결과를 출력하는 프로그램을 작성합니다.

Hint1 : Queue!

Hint2 : 간선을 직접 없애는 것도 좋지만, 정점으로 들어오는 간선의 숫자만 잘 조정한다면?

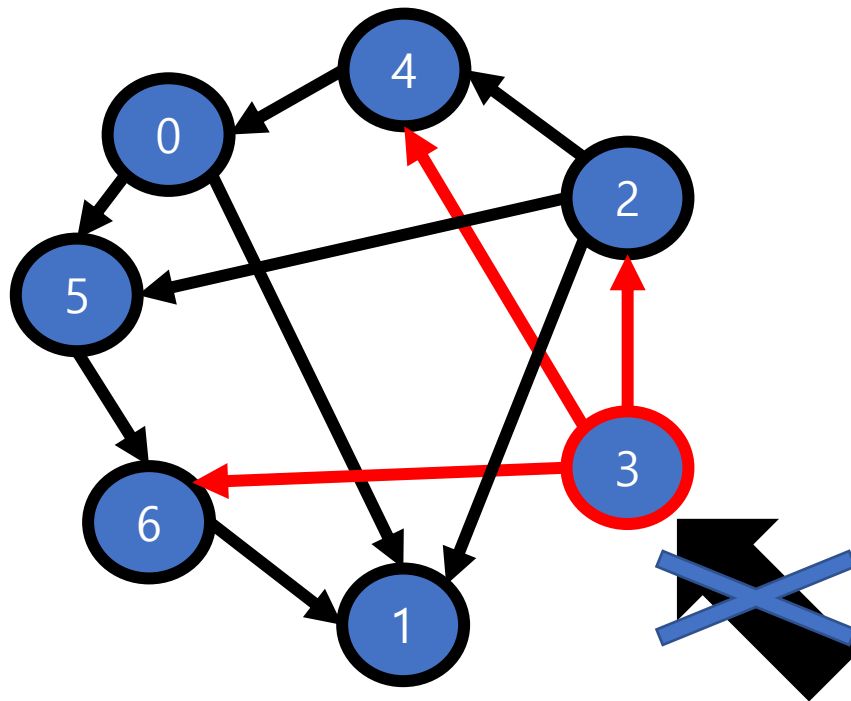


Indegree = [1, 3, 1, 0, 2, 2, 2]

DFS	3 2 1 4 0 5 6	O(V+E)
BFS	3 2 4 6 1 5 0	
Topological Sort	3 2 4 0 5 6 1	



# ADD01. 위상 정렬

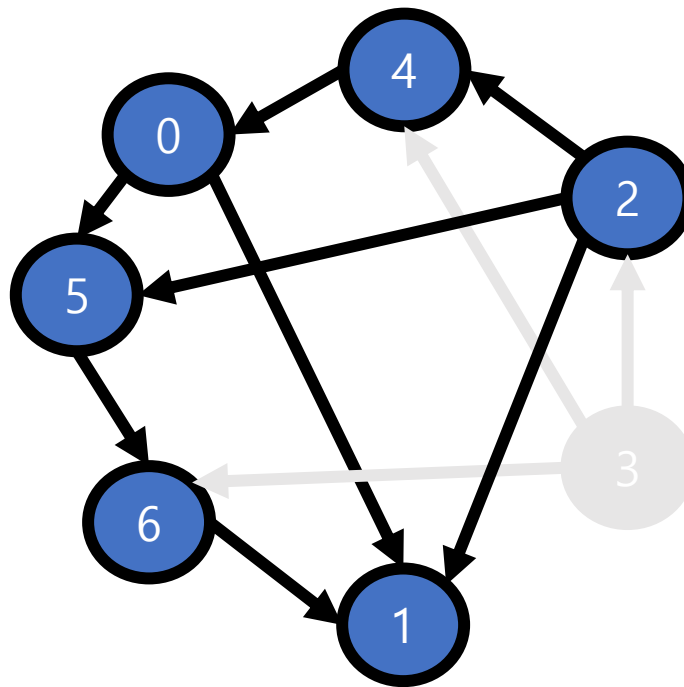


Indegree = [1, 3, 1, 0, 2, 2, 2]

3



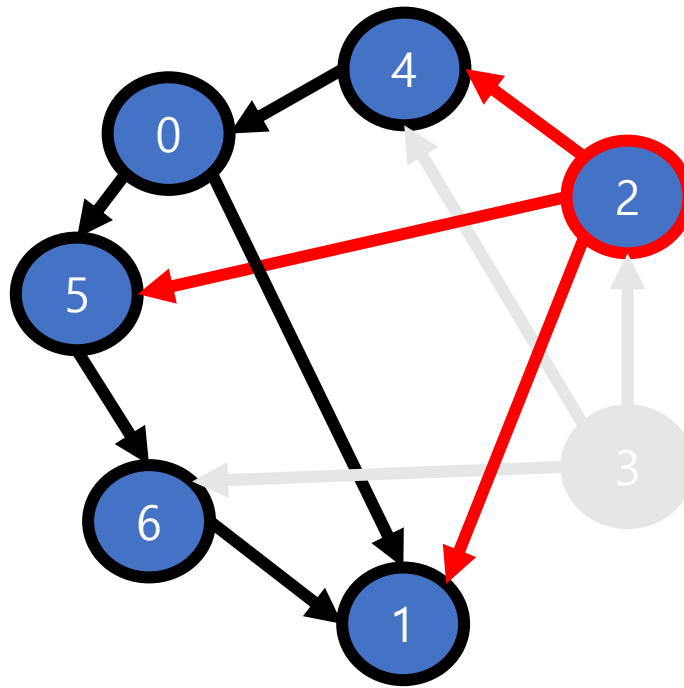
# ADD01. 위상 정렬



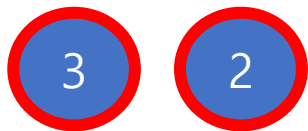
Indegree = [1, 3, 0, 0, 1, 2, 1]

3

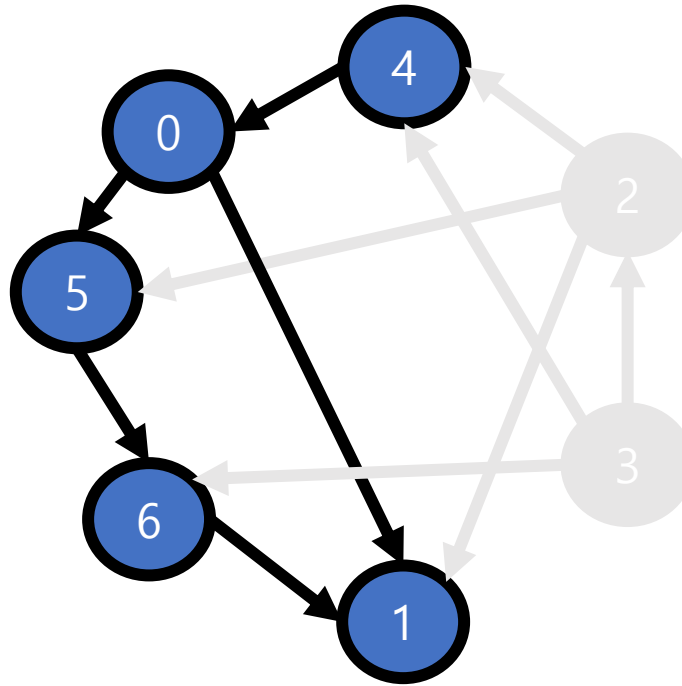
# ADD01. 위상 정렬



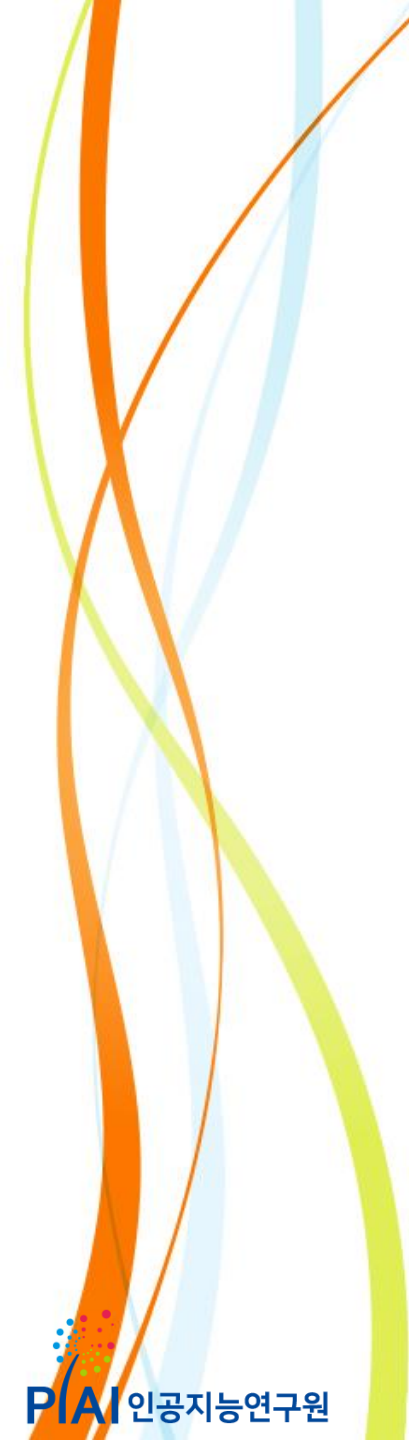
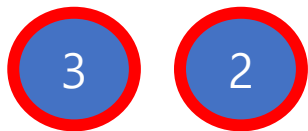
Indegree = [1, 3, 0, 0, 1, 2, 1]



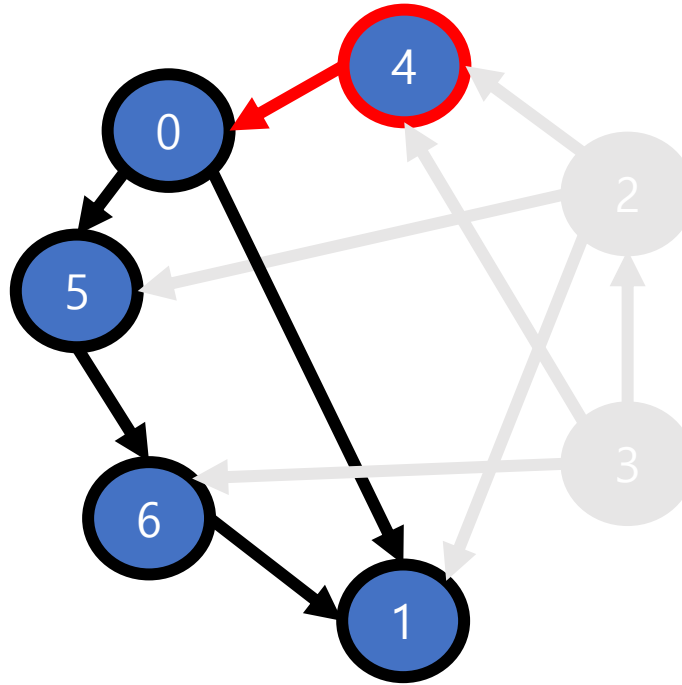
# ADD01. 위상 정렬



Indegree = [1, 2, 0, 0, 0, 1, 1]



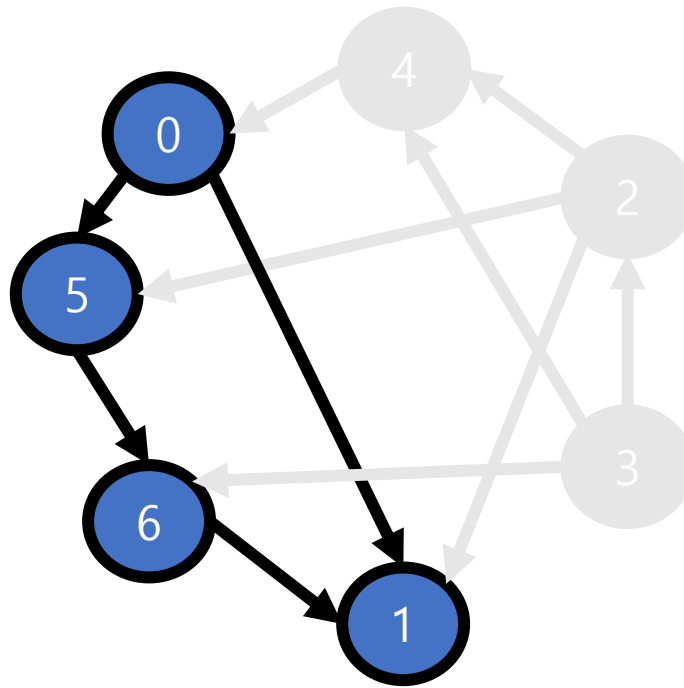
# ADD01. 위상 정렬



Indegree = [1, 2, 0, 0, 0, 1, 1]



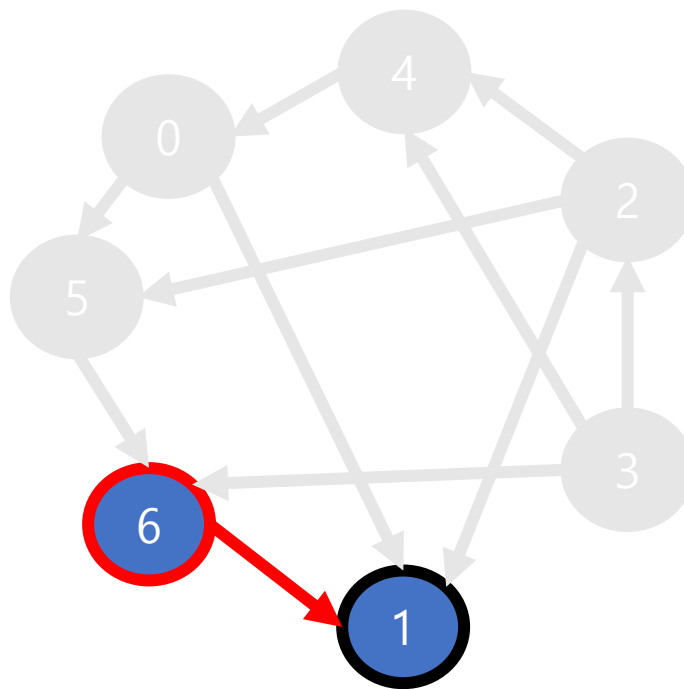
# ADD01. 위상 정렬



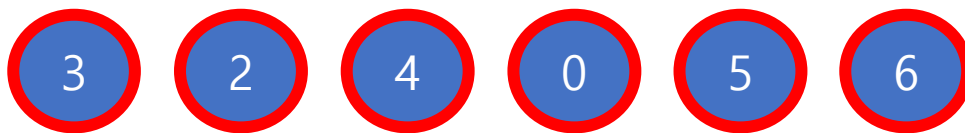
Indegree = [0, 2, 0, 0, 0, 1, 1]



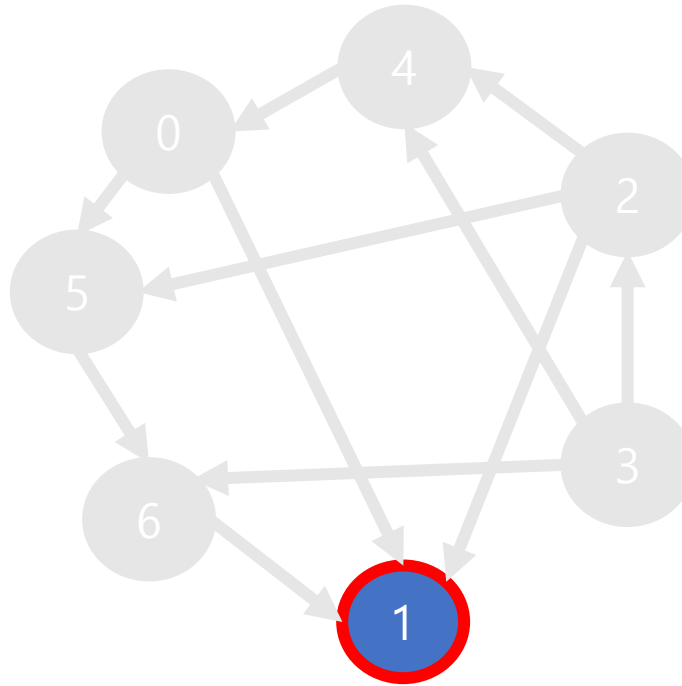
# ADD01. 위상 정렬



Indegree = [0, 1, 0, 0, 0, 0, 0]



# ADD01. 위상 정렬

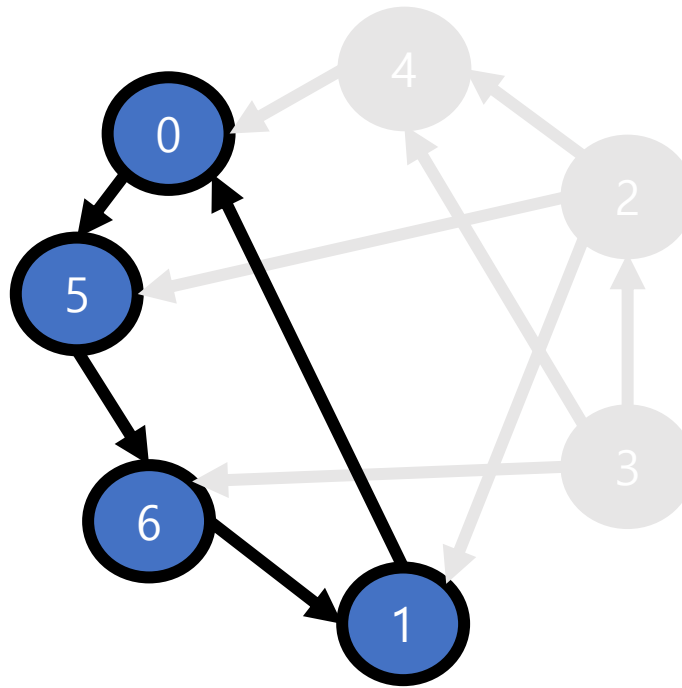


Indegree = [0, 0, 0, 0, 0, 0, 0]

3 2 4 0 5 6 1 = Result



# ADD01. 위상 정렬



Indegree = [1, 1, 0, 0, 0, 1, 1]



위상 정렬 불가 판별: 모든 정점을 탐색을 안 했는데, indegree가 0인 정점이 더 이상 없다!  
- Queue를 통한 판별! (indegree가 0인 정점들을 Queue에 넣고, pop할 때 Queue가 비면?)



# ADD01. 위상 정렬

```
from collections import deque
T = int(input())
for _ in range(T):
    N,M = map(int, input().split())
    List = [[] for _ in range(N)]
    Indegree = [0]*N # 들어오는 간선이 몇 개인지 각 정점에 대해 저장
    for i in range(M):
        a,b = map(int,input().split())
        List[a].append(b)
        Indegree[b] += 1 # a->b 간선이므로 b의 Indegree를 +1
    Queue = deque([]) # 큐의 선언
    # Indegree가 0인 정점을 찾는 과정
    for i in range(N):
        [REDACTED]

    result = [] # 결과가 저장되는 곳
    while Queue: # Queue가 빌 때까지 (더 이상 탐색을 못할때까지)
        x = Queue.popleft()
        result.append(x)
        [REDACTED]

    if [REDACTED]: # while문이 끝났을 때 정점의 개수가 입력과 일치하면 위상정렬 성공
        print(*result)
    else: # 아니면 실패 (*사이클이 존재함을 다른 방식으로 찾아도 됩니다)
        print(-1)
```