

이것이 취업을 위한 코딩 테스트다 with 파이썬

최단 경로 & 기타 그래프 이론

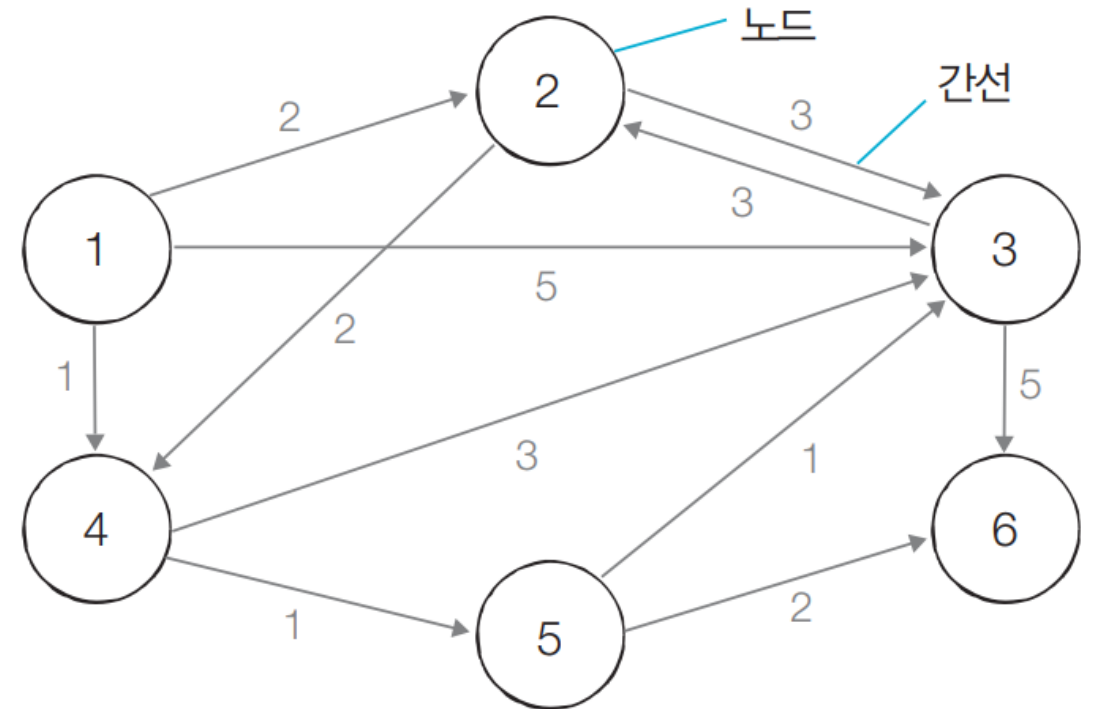
나동빈(dongbinna@postech.ac.kr)

Pohang University of Science and Technology

최단 경로 알고리즘

최단 경로 문제

- 최단 경로 알고리즘은 가장 짧은 경로를 찾는 알고리즘을 의미합니다.
- 다양한 문제 상황
 - 한 지점에서 다른 한 지점까지의 최단 경로
 - 한 지점에서 다른 모든 지점까지의 최단 경로
 - 모든 지점에서 다른 모든 지점까지의 최단 경로
- 각 지점은 그래프에서 **노드**로 표현
- 지점 간 연결된 도로는 그래프에서 **간선**으로 표현



다익스트라 최단 경로 알고리즘 개요

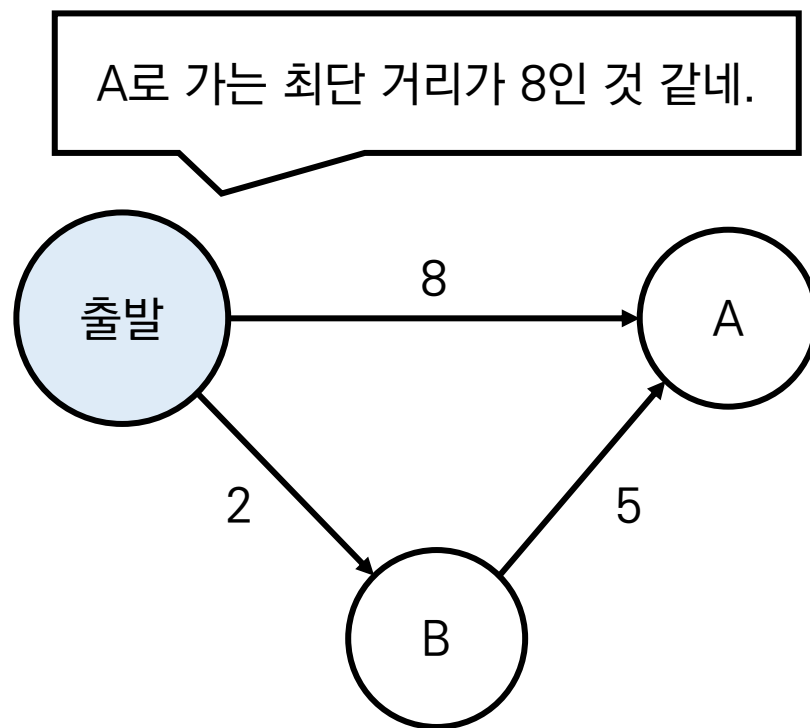
- **특정한 노드에서 출발하여 다른 모든 노드로 가는 최단 경로를 계산합니다.**
- 다익스트라 최단 경로 알고리즘은 음의 간선이 없을 때 정상적으로 동작합니다.
 - 현실 세계의 도로(간선)는 음의 간선으로 표현되지 않습니다.
- 다익스트라 최단 경로 알고리즘은 그리디 알고리즘으로 분류됩니다.
 - **매 상황에서 가장 비용이 적은 노드를 선택해 임의의 과정을 반복합니다.**

다익스트라 최단 경로 알고리즘

- 알고리즘의 **동작 과정**은 다음과 같습니다.
 1. 출발 노드를 설정합니다.
 2. 최단 거리 테이블을 초기화합니다.
 3. 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택합니다.
 4. 해당 노드를 거쳐 다른 노드로 가는 비용을 계산하여 최단 거리 테이블을 갱신합니다.
 5. 위 과정에서 3번과 4번을 반복합니다.

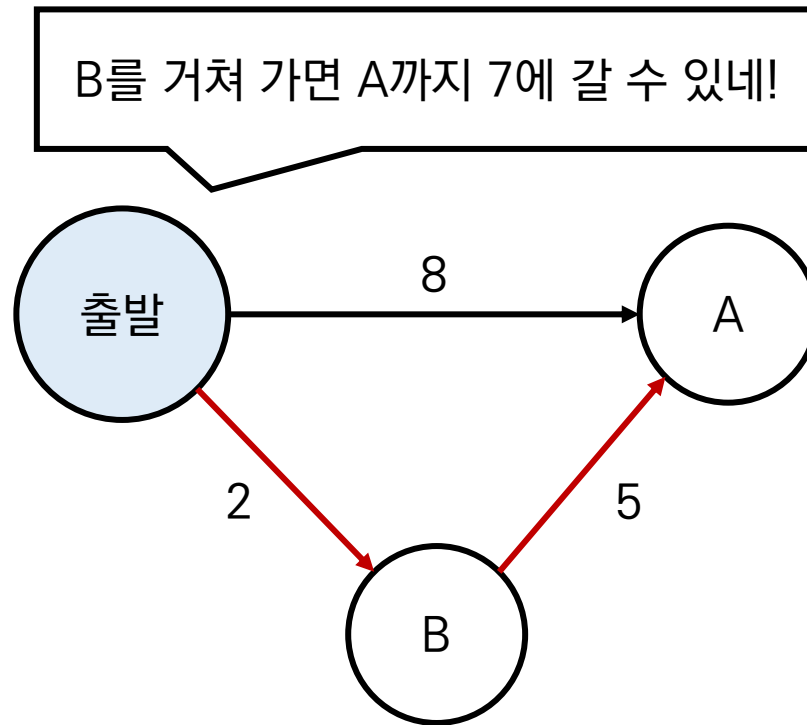
다익스트라 최단 경로 알고리즘

- 알고리즘 동작 과정에서 최단 거리 테이블은 각 노드에 대한 현재까지의 최단 거리 정보를 가지고 있습니다.
- 처리 과정에서 더 짧은 경로를 찾으면 '이제부터는 이 경로가 제일 짧은 경로야'라고 갱신합니다.



다익스트라 최단 경로 알고리즘

- 알고리즘 동작 과정에서 최단 거리 테이블은 각 노드에 대한 현재까지의 최단 거리 정보를 가지고 있습니다.
- 처리 과정에서 더 짧은 경로를 찾으면 '이제부터는 이 경로가 제일 짧은 경로야'라고 갱신합니다.



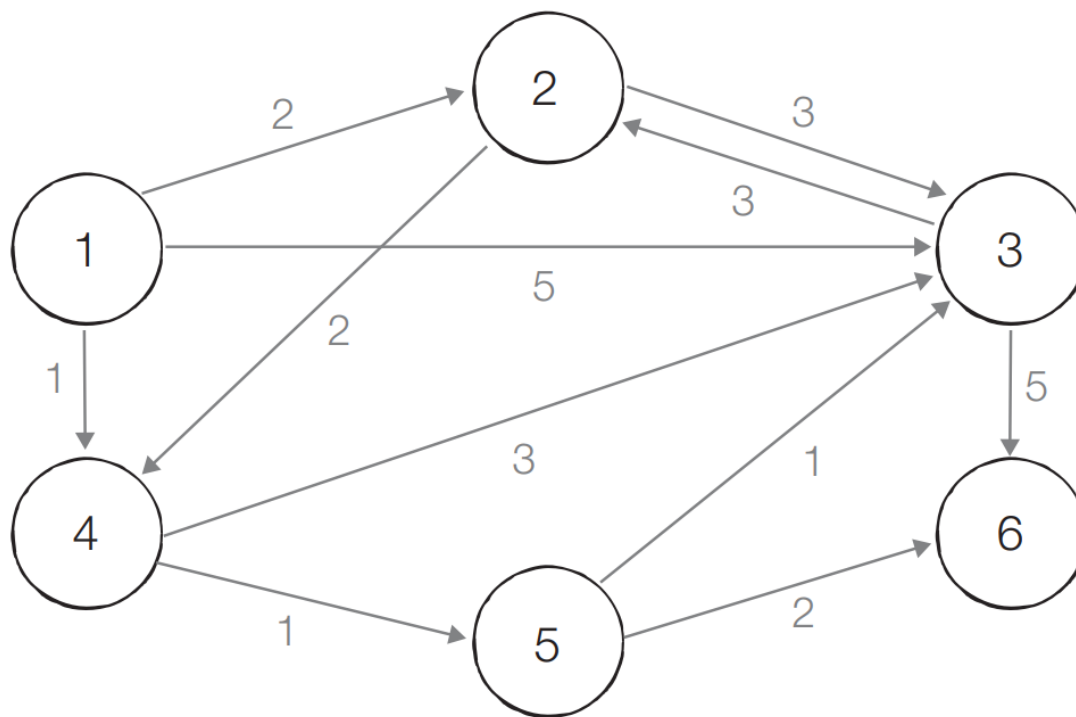
다익스트라 알고리즘: 동작 과정 살펴보기

- [초기 상태] 그래프를 준비하고 출발 노드를 설정합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드



노드 번호	1	2	3	4	5	6
거리	0	무한	무한	무한	무한	무한

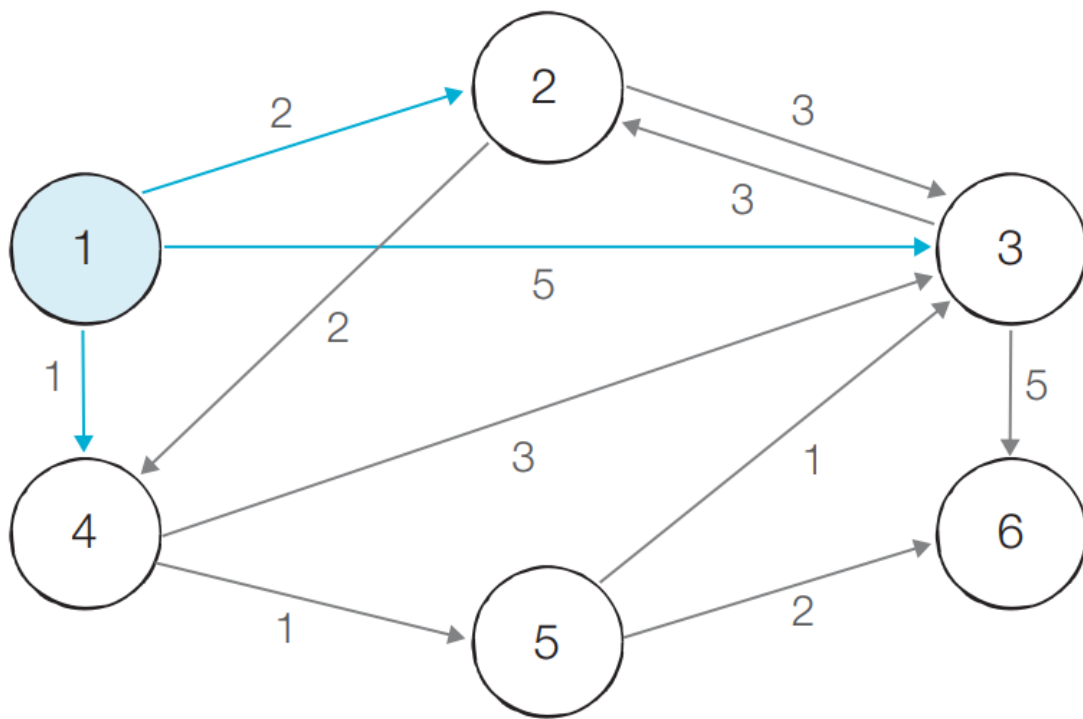
다익스트라 알고리즘: 동작 과정 살펴보기

- [Step 1] 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드인 1번 노드를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드



인접 노드	현재 값	거쳐갈 때	갱신 여부
2번	무한	$0+2$	True
3번	무한	$0+5$	True
4번	무한	$0+1$	True

노드 번호	1	2	3	4	5	6
거리	0	2	5	1	무한	무한



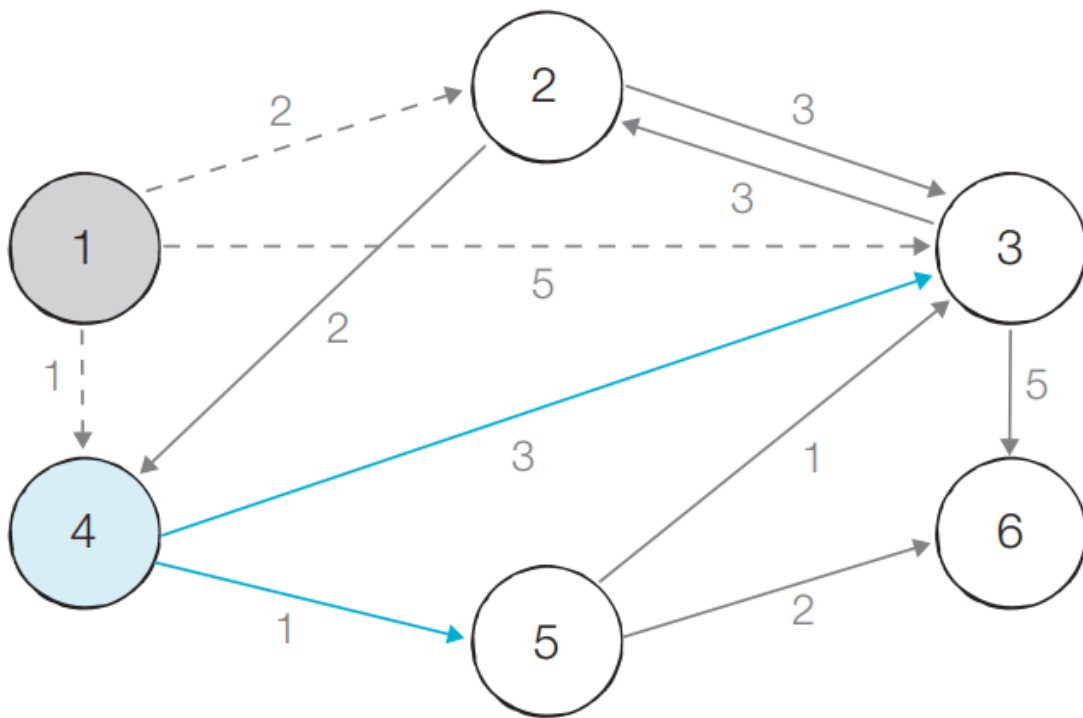
다익스트라 알고리즘: 동작 과정 살펴보기

- [Step 2] 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드인 4번 노드를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드



인접 노드	현재 값	거쳐갈 때	갱신 여부
3번	5	1+3	True
5번	무한	1+1	True

노드 번호	1	2	3	4	5	6
거리	0	2	4	1	2	무한



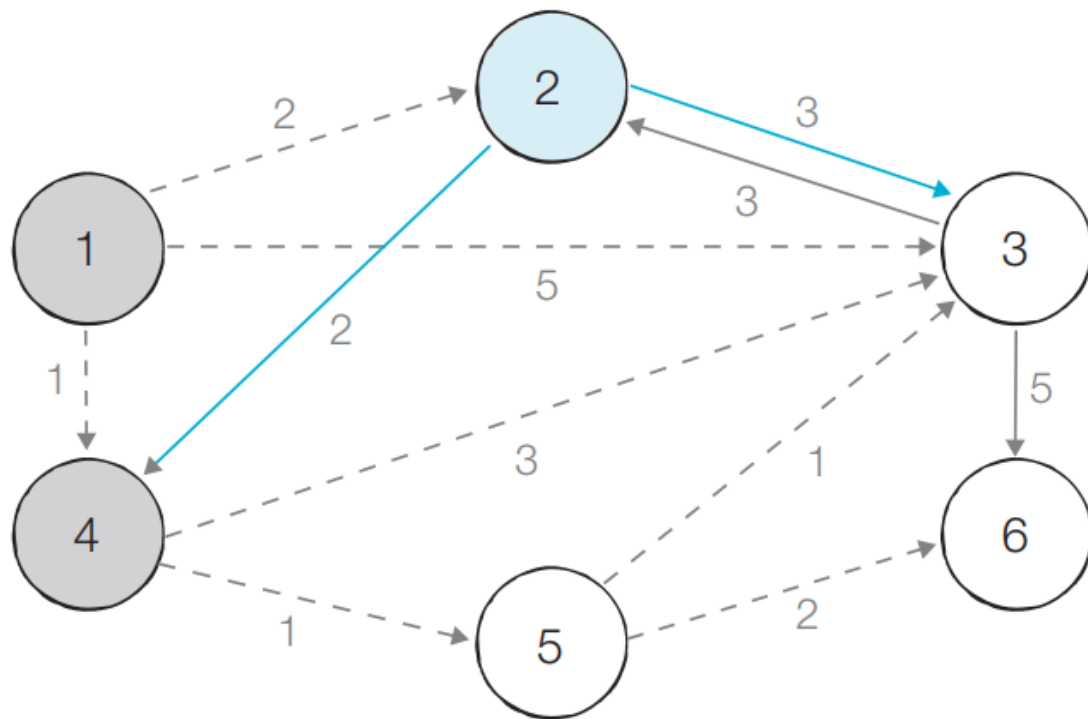
다익스트라 알고리즘: 동작 과정 살펴보기

- [Step 3] 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드인 2번 노드를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드



인접 노드	현재 값	거쳐갈 때	갱신 여부
3번	4	2+3	False
4번	1	2+2	False

노드 번호	1	2	3	4	5	6
거리	0	2	4	1	2	무한



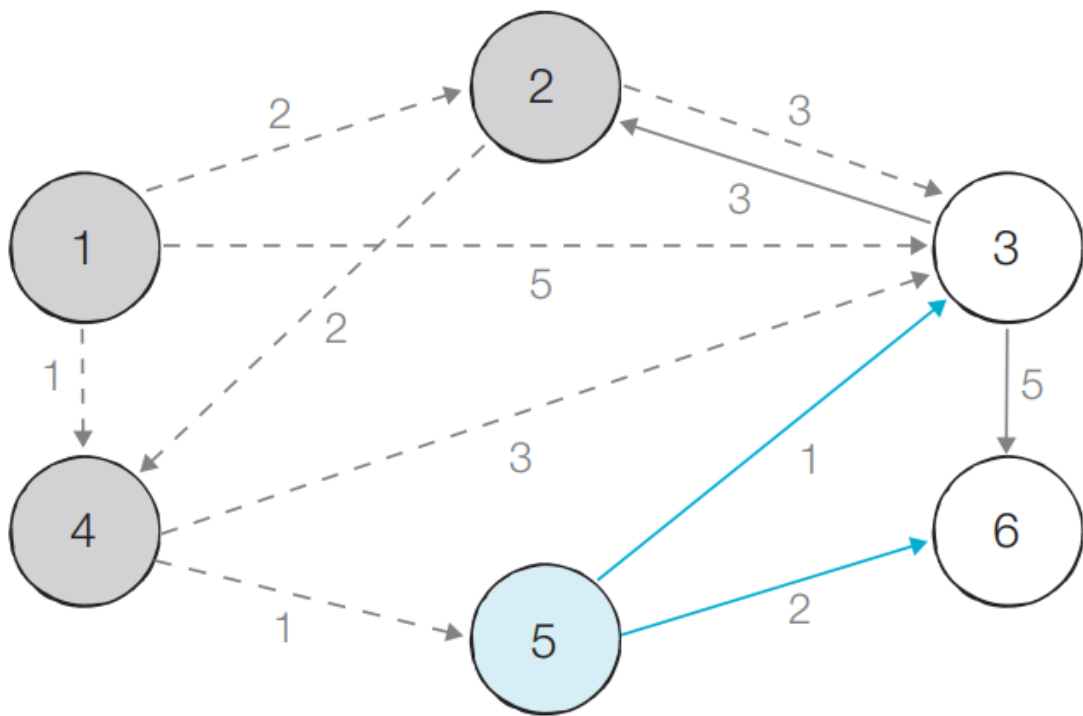
다익스트라 알고리즘: 동작 과정 살펴보기

- [Step 4] 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드인 5번 노드를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드



인접 노드	현재 값	거쳐갈 때	갱신 여부
3번	4	2+1	True
6번	무한	2+2	True

노드 번호	1	2	3	4	5	6
거리	0	2	3	1	2	4



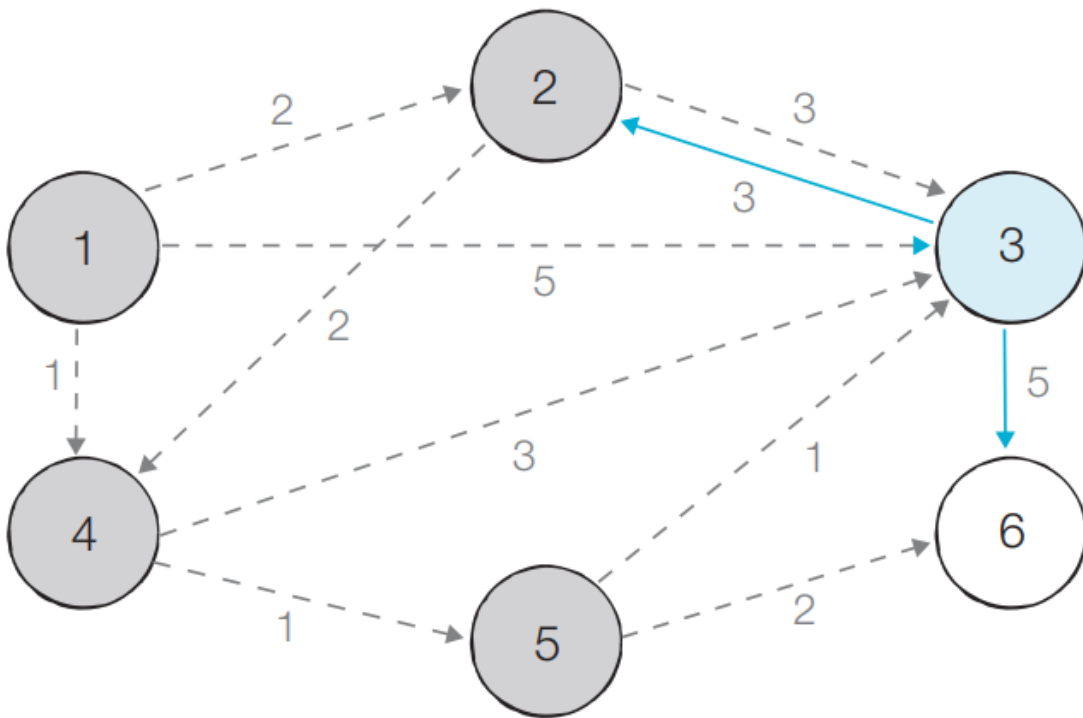
다익스트라 알고리즘: 동작 과정 살펴보기

- [Step 5] 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드인 3번 노드를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드



인접 노드	현재 값	거쳐갈 때	갱신 여부
2번	2	3+3	False
6번	4	3+5	False

노드 번호	1	2	3	4	5	6
거리	0	2	3	1	2	4



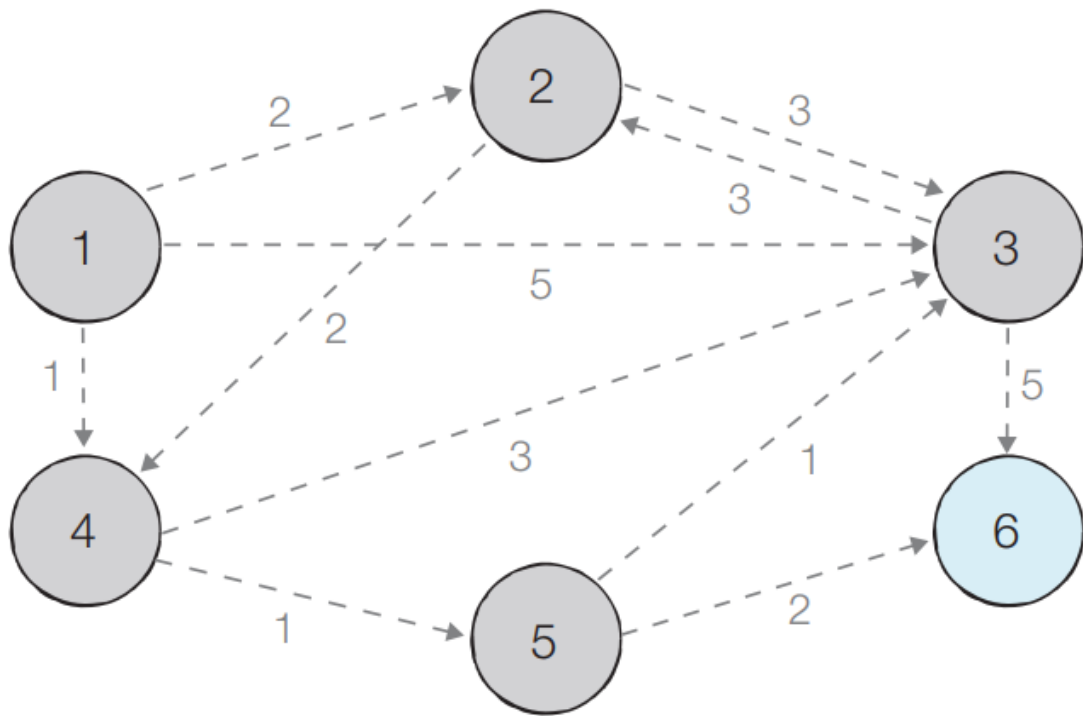
다익스트라 알고리즘: 동작 과정 살펴보기

- [Step 6] 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드인 6번 노드를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드



인접 노드	현재 값	거쳐갈 때	갱신 여부
이동할 수 있는 인접 노드가 없습니다.			

노드 번호	1	2	3	4	5	6
거리	0	2	3	1	2	4

다익스트라 알고리즘의 특징

- 그리디 알고리즘: 매 상황에서 방문하지 않은 가장 비용이 적은 노드를 선택해 임의의 과정을 반복합니다.
- 단계를 거치며 **한 번 처리된 노드의 최단 거리는 고정**되어 더 이상 바뀌지 않습니다.
 - 한 단계당 하나의 노드에 대한 최단 거리를 확실히 찾는 것으로 이해할 수 있습니다.
- 다익스트라 알고리즘을 수행한 뒤에 테이블에 각 노드까지의 최단 거리 정보가 저장됩니다.
 - 완전한 형태의 최단 경로를 구하려면 소스코드에 추가적인 기능을 더 넣어야 합니다.

다익스트라 알고리즘: 간단한 구현 방법

- 단계마다 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택하기 위해 매 단계마다 1차원 테이블의 모든 원소를 확인(순차 탐색)합니다.

다익스트라 알고리즘: 간단한 구현 방법 (Python)

```
import sys
input = sys.stdin.readline
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

# 노드의 개수, 간선의 개수를 입력받기
n, m = map(int, input().split())
# 시작 노드 번호를 입력받기
start = int(input())
# 각 노드에 연결되어 있는 노드에 대한 정보를 담는 리스트를 만들기
graph = [[] for i in range(n + 1)]
# 방문한 적이 있는지 체크하는 목적의 리스트를 만들기
visited = [False] * (n + 1)
# 최단 거리 테이블을 모두 무한으로 초기화
distance = [INF] * (n + 1)

# 모든 간선 정보를 입력받기
for _ in range(m):
    a, b, c = map(int, input().split())
    # a번 노드에서 b번 노드로 가는 비용이 c라는 의미
    graph[a].append((b, c))

# 방문하지 않은 노드 중에서, 가장 최단 거리가 짧은 노드의 번호를 반환
def get_smallest_node():
    min_value = INF
    index = 0 # 가장 최단 거리가 짧은 노드(인덱스)
    for i in range(1, n + 1):
        if distance[i] < min_value and not visited[i]:
            min_value = distance[i]
            index = i
    return index
```

```
def dijkstra(start):
    # 시작 노드에 대해서 초기화
    distance[start] = 0
    visited[start] = True
    for j in graph[start]:
        distance[j[0]] = j[1]
    # 시작 노드를 제외한 전체 n - 1개의 노드에 대해 반복
    for i in range(n - 1):
        # 현재 최단 거리가 가장 짧은 노드를 꺼내서, 방문 처리
        now = get_smallest_node()
        visited[now] = True
        # 현재 노드와 연결된 다른 노드를 확인
        for j in graph[now]:
            cost = distance[now] + j[1]
            # 현재 노드를 거쳐서 다른 노드로 이동하는 거리가 더 짧은 경우
            if cost < distance[j[0]]:
                distance[j[0]] = cost

# 다익스트라 알고리즘을 수행
dijkstra(start)

# 모든 노드로 가기 위한 최단 거리를 출력
for i in range(1, n + 1):
    # 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
    if distance[i] == INF:
        print("INFINITY")
    # 도달할 수 있는 경우 거리를 출력
    else:
        print(distance[i])
```

다익스트라 알고리즘: 간단한 구현 방법 성능 분석

- 총 $O(V)$ 번에 걸쳐서 최단 거리가 가장 짧은 노드를 매번 선형 탐색해야 합니다.
- 따라서 전체 시간 복잡도는 $O(V^2)$ 입니다.
- 일반적으로 코딩 테스트의 최단 경로 문제에서 전체 노드의 개수가 5,000개 이하라면 이 코드로 문제를 해결할 수 있습니다.
 - 하지만 노드의 개수가 10,000개를 넘어가는 문제라면 어떻게 해야 할까요?

우선순위 큐(Priority Queue)

- 우선순위가 가장 높은 데이터를 가장 먼저 삭제하는 자료구조입니다.
- 예를 들어 여러 개의 물건 데이터를 자료구조에 넣었다가 가치가 높은 물건 데이터부터 꺼내서 확인해야 하는 경우에 우선순위 큐를 이용할 수 있습니다.
- Python, C++, Java를 포함한 대부분의 프로그래밍 언어에서 **표준 라이브러리 형태로 지원합니다.**

자료구조	추출되는 데이터
스택(Stack)	가장 나중에 삽입된 데이터
큐(Queue)	가장 먼저 삽입된 데이터
우선순위 큐(Priority Queue)	가장 우선순위가 높은 데이터

힙(Heap)

- 우선순위 큐(Priority Queue)를 구현하기 위해 사용하는 자료구조 중 하나입니다.
- **최소 힙(Min Heap)과 최대 힙(Max Heap)**이 있습니다.
- 다익스트라 최단 경로 알고리즘을 포함해 다양한 알고리즘에서 사용됩니다.

우선순위 큐 구현 방식	삽입 시간	삭제 시간
리스트	$O(1)$	$O(N)$
힙(Heap)	$O(\log N)$	$O(\log N)$

힙 라이브러리 사용 예제: 최소 힙

```
import heapq

# 오름차순 힙 정렬(Heap Sort)
def heapsort(iterable):
    h = []
    result = []
    # 모든 원소를 차례대로 힙에 삽입
    for value in iterable:
        heapq.heappush(h, value)
    # 힙에 삽입된 모든 원소를 차례대로 꺼내어 담기
    for i in range(len(h)):
        result.append(heapq.heappop(h))
    return result

result = heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
print(result)
```

실행 결과

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

힙 라이브러리 사용 예제: 최대 힙

```
import heapq

# 내림차순 힙 정렬(Heap Sort)
def heapsort(iterable):
    h = []
    result = []
    # 모든 원소를 차례대로 힙에 삽입
    for value in iterable:
        heapq.heappush(h, -value)
    # 힙에 삽입된 모든 원소를 차례대로 꺼내어 담기
    for i in range(len(h)):
        result.append(-heapq.heappop(h))
    return result

result = heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
print(result)
```

실행 결과

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

다익스트라 알고리즘: 개선된 구현 방법

- 단계마다 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택하기 위해 힙(Heap) 자료구조를 이용합니다.
- 다익스트라 알고리즘이 동작하는 **기본 원리는 동일**합니다.
 - 현재 가장 가까운 노드를 저장해 놓기 위해서 힙 자료구조를 추가적으로 이용한다는 점이 다릅니다.
 - 현재의 최단 거리가 가장 짧은 노드를 선택해야 하므로 최소 힙을 사용합니다.

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

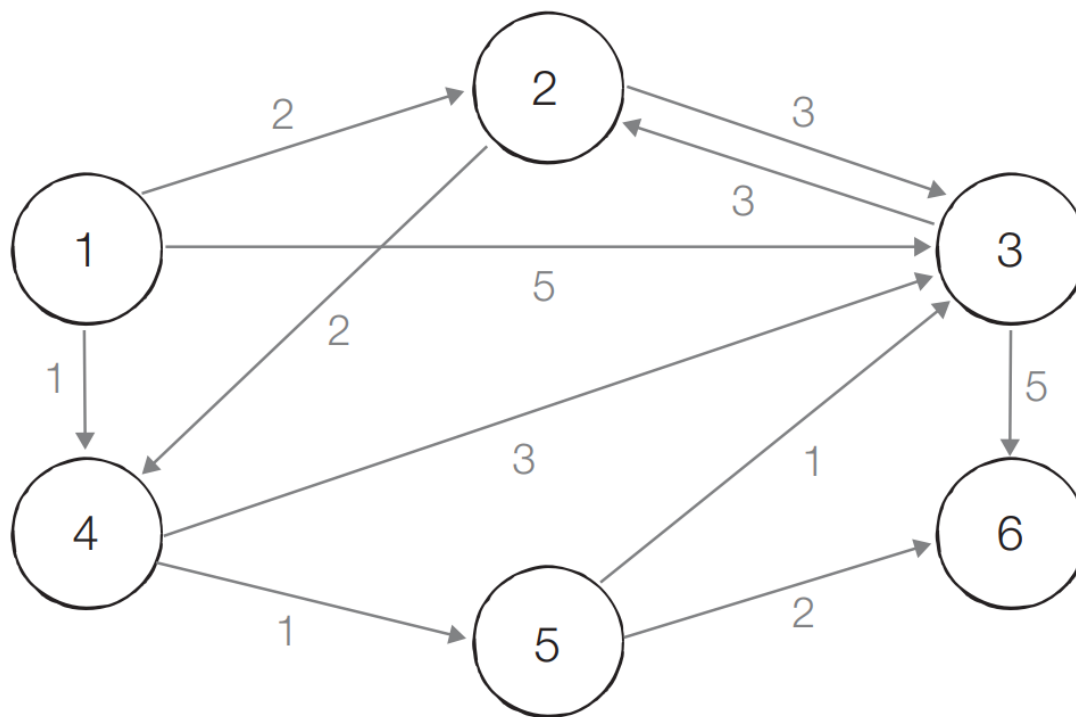
- [초기 상태] 그래프를 준비하고 출발 노드를 설정하여 우선순위 큐에 삽입합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드

우선순위 큐
(거리: 0, 노드: 1)



노드 번호	1	2	3	4	5	6
거리	0	무한	무한	무한	무한	무한

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

- [Step 1] 우선순위 큐에서 원소를 꺼냅니다. 1번 노드는 아직 방문하지 않았으므로 이를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

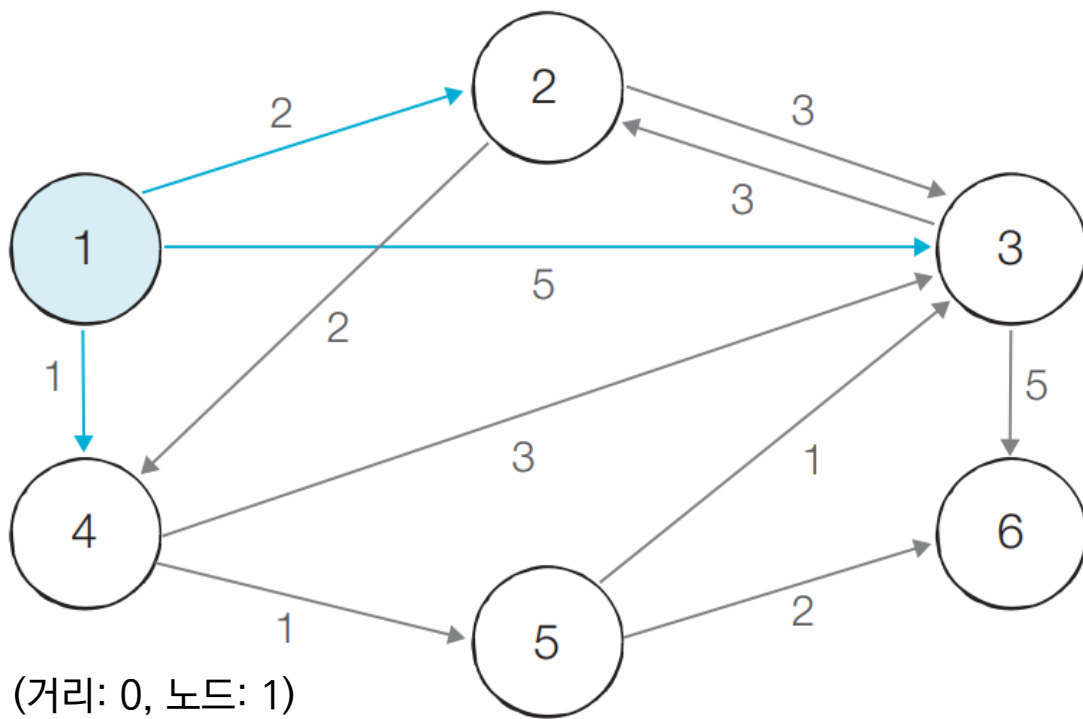
■ : 이미 방문한 노드

우선순위 큐

(거리: 1, 노드: 4)

(거리: 2, 노드: 2)

(거리: 5, 노드: 3)



현재 꺼낸 원소: (거리: 0, 노드: 1)

인접 노드	현재 값	거쳐갈 때	갱신 여부
2번	무한	0+2	True
3번	무한	0+5	True
4번	무한	0+1	True

노드 번호	1	2	3	4	5	6
거리	0	2	5	1	무한	무한

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

- [Step 2] 우선순위 큐에서 원소를 꺼냅니다. 4번 노드는 아직 방문하지 않았으므로 이를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드

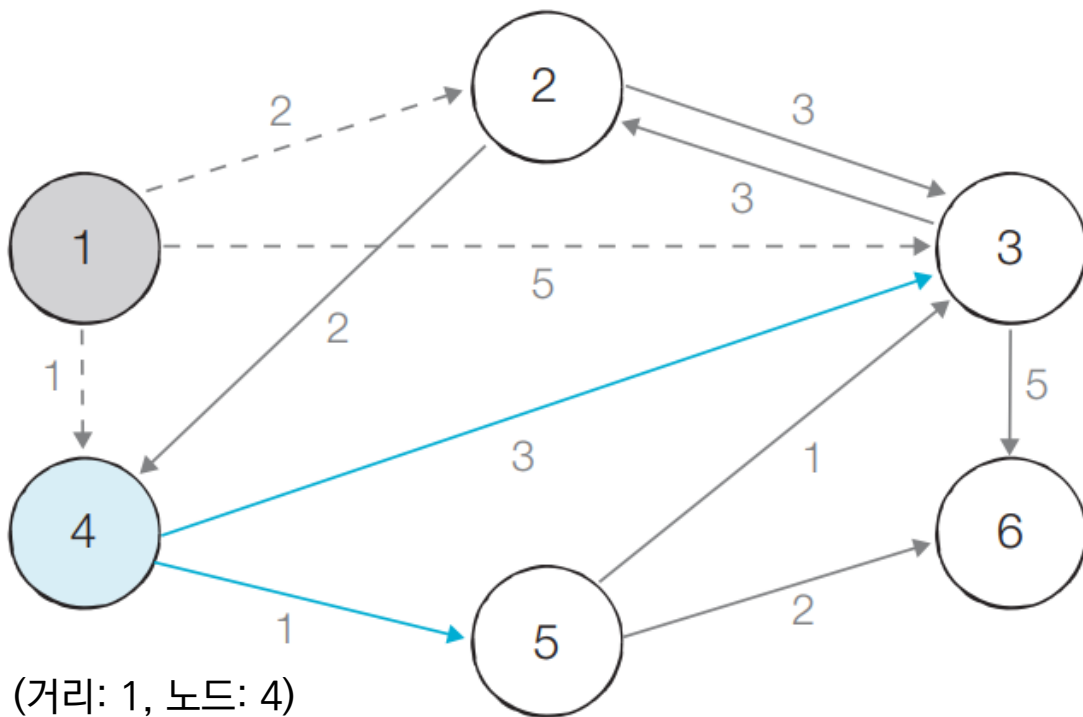
우선순위 큐

(거리: 2, 노드: 2)

(거리: 2, 노드: 5)

(거리: 4, 노드: 3)

(거리: 5, 노드: 3)



현재 꺼낸 원소: (거리: 1, 노드: 4)

인접 노드	현재 값	거쳐갈 때	갱신 여부
3번	5	1+3	True
5번	무한	1+1	True

노드 번호	1	2	3	4	5	6
거리	0	2	4	1	2	무한

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

- [Step 3] 우선순위 큐에서 원소를 꺼냅니다. 2번 노드는 아직 방문하지 않았으므로 이를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

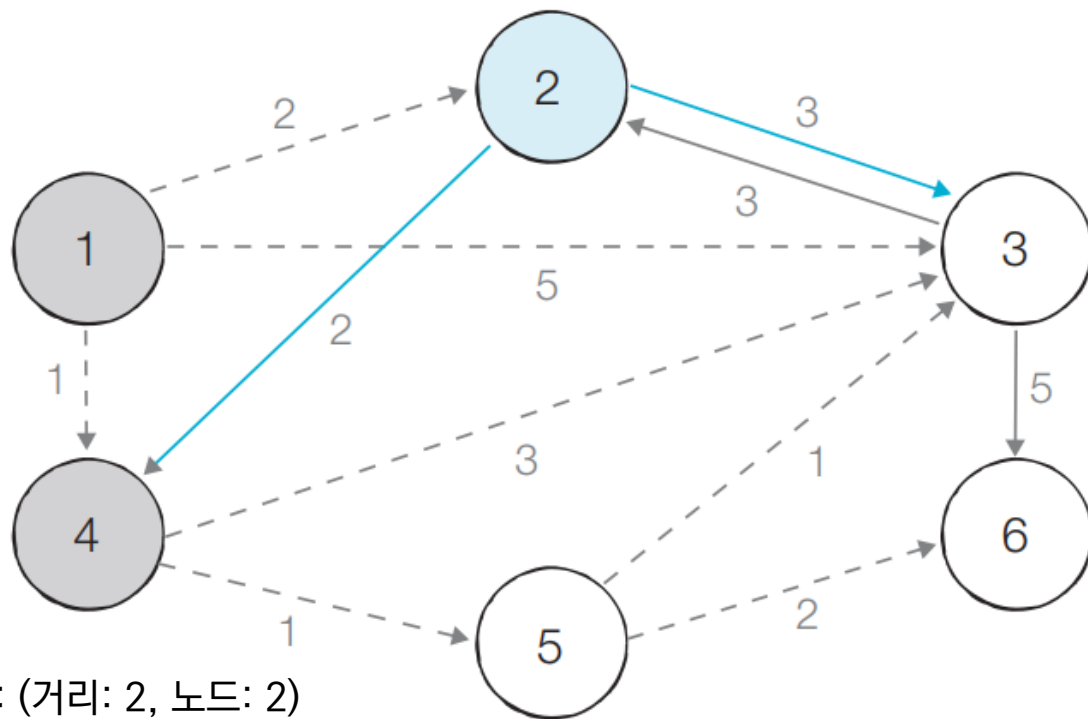
■ : 이미 방문한 노드

우선순위 큐

(거리: 2, 노드: 5)

(거리: 4, 노드: 3)

(거리: 5, 노드: 3)



현재 꺼낸 원소: (거리: 2, 노드: 2)

인접 노드	현재 값	거쳐갈 때	갱신 여부
3번	4	2+3	False
4번	1	2+2	False

노드 번호	1	2	3	4	5	6
거리	0	2	4	1	2	무한

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

- [Step 4] 우선순위 큐에서 원소를 꺼냅니다. 5번 노드는 아직 방문하지 않았으므로 이를 처리합니다.

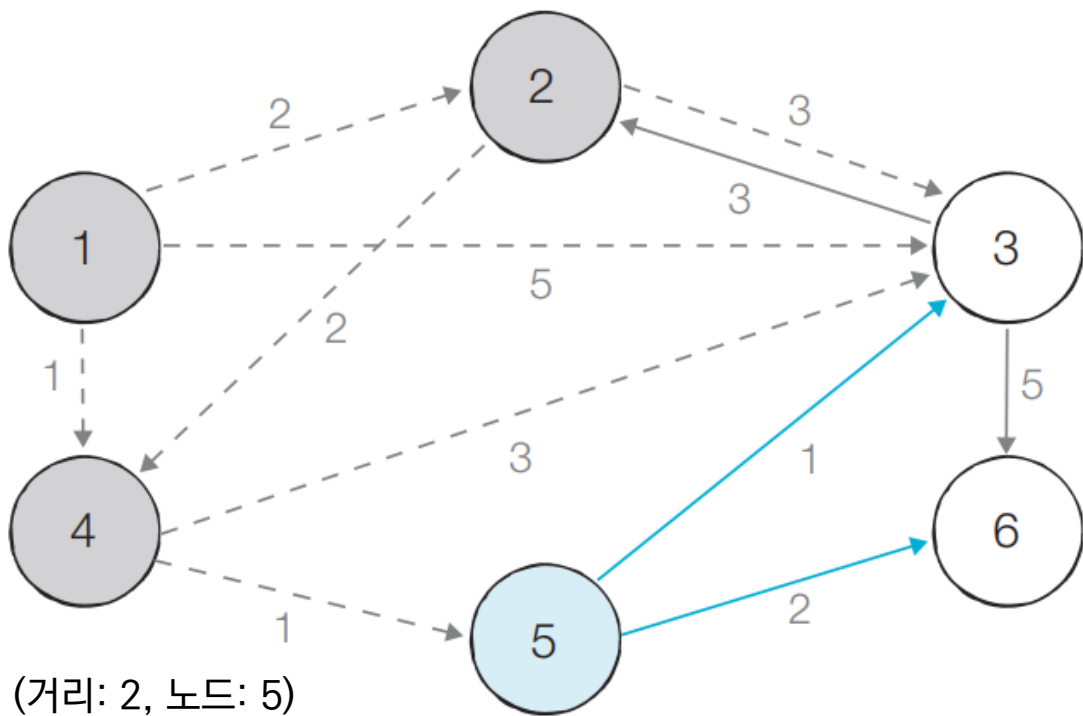
출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드

우선순위 큐

(거리: 3, 노드: 3)
(거리: 4, 노드: 3)
(거리: 4, 노드: 6)
(거리: 5, 노드: 3)



현재 꺼낸 원소: (거리: 2, 노드: 5)

인접 노드	현재 값	거쳐갈 때	갱신 여부
3번	4	2+1	True
6번	무한	2+2	True

노드 번호	1	2	3	4	5	6
거리	0	2	3	1	2	4

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

- [Step 5] 우선순위 큐에서 원소를 꺼냅니다. 3번 노드는 아직 방문하지 않았으므로 이를 처리합니다.

출발: 1번 노드

■ : 처리 중인 노드

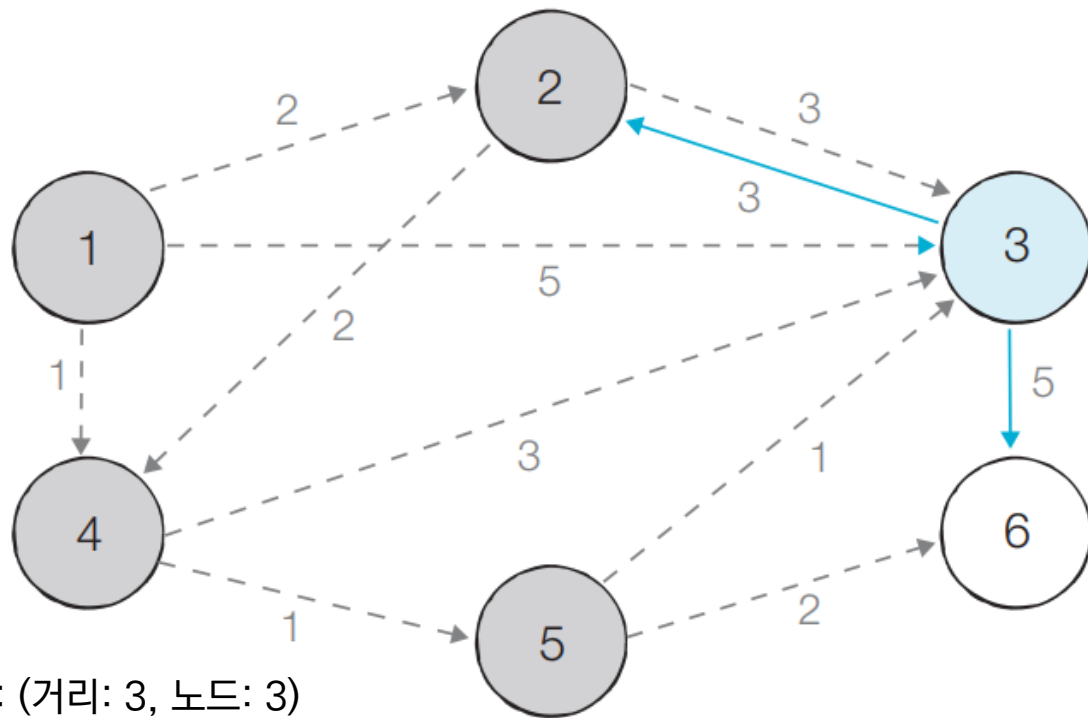
■ : 이미 방문한 노드

우선순위 큐

(거리: 4, 노드: 3)

(거리: 4, 노드: 6)

(거리: 5, 노드: 3)



현재 꺼낸 원소: (거리: 3, 노드: 3)

인접 노드	현재 값	거쳐갈 때	갱신 여부
2번	2	3+3	False
6번	4	3+5	False

노드 번호	1	2	3	4	5	6
거리	0	2	3	1	2	4

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

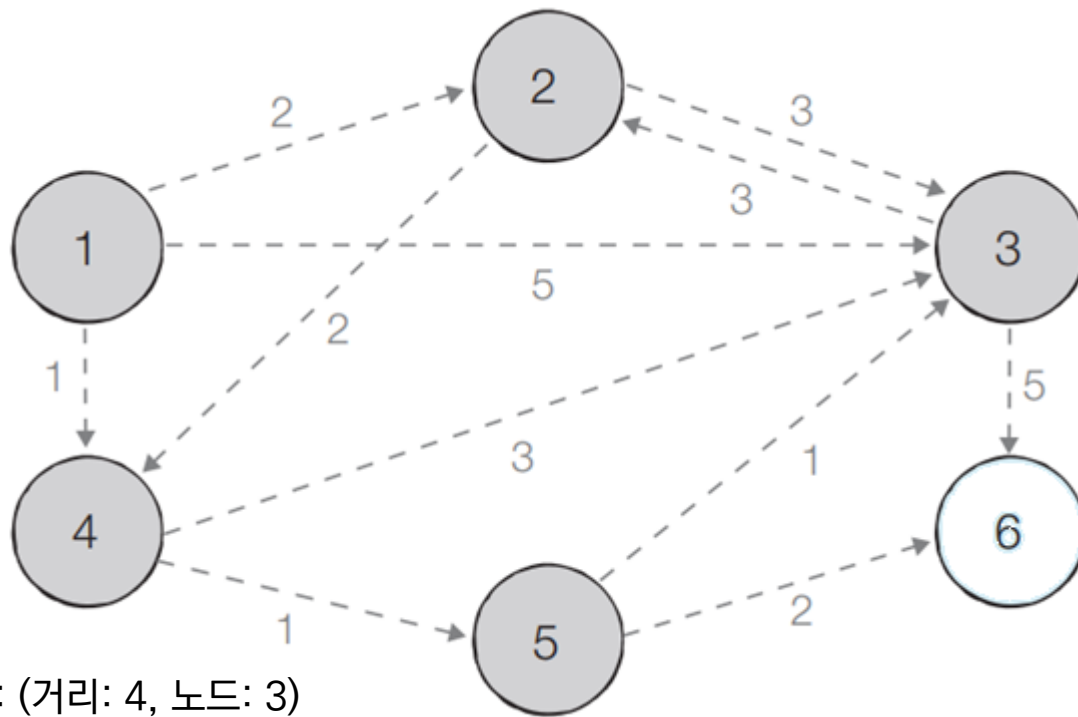
- [Step 6] 우선순위 큐에서 원소를 꺼냅니다. 3번 노드는 이미 방문했으므로 무시합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드

우선순위 큐
(거리: 4, 노드: 6)
(거리: 5, 노드: 3)



현재 꺼낸 원소: (거리: 4, 노드: 3)

노드 번호	1	2	3	4	5	6
거리	0	2	3	1	2	4

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

- [Step 7] 우선순위 큐에서 원소를 꺼냅니다. 6번 노드는 아직 방문하지 않았으므로 이를 처리합니다.

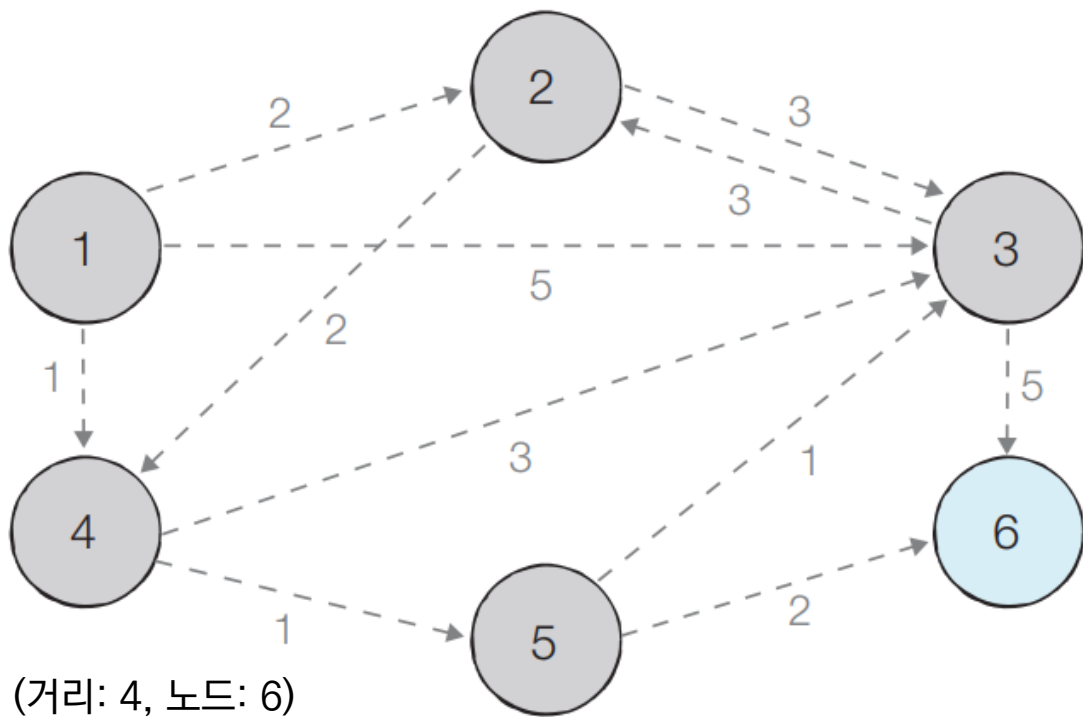
출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드

우선순위 큐

(거리: 5, 노드: 3)



현재 꺼낸 원소: (거리: 4, 노드: 6)

노드 번호	1	2	3	4	5	6
거리	0	2	3	1	2	4

인접 노드	현재 값	거쳐갈 때	갱신 여부
이동할 수 있는 인접 노드가 없습니다.			

다익스트라 알고리즘: 동작 과정 살펴보기 (우선순위 큐)

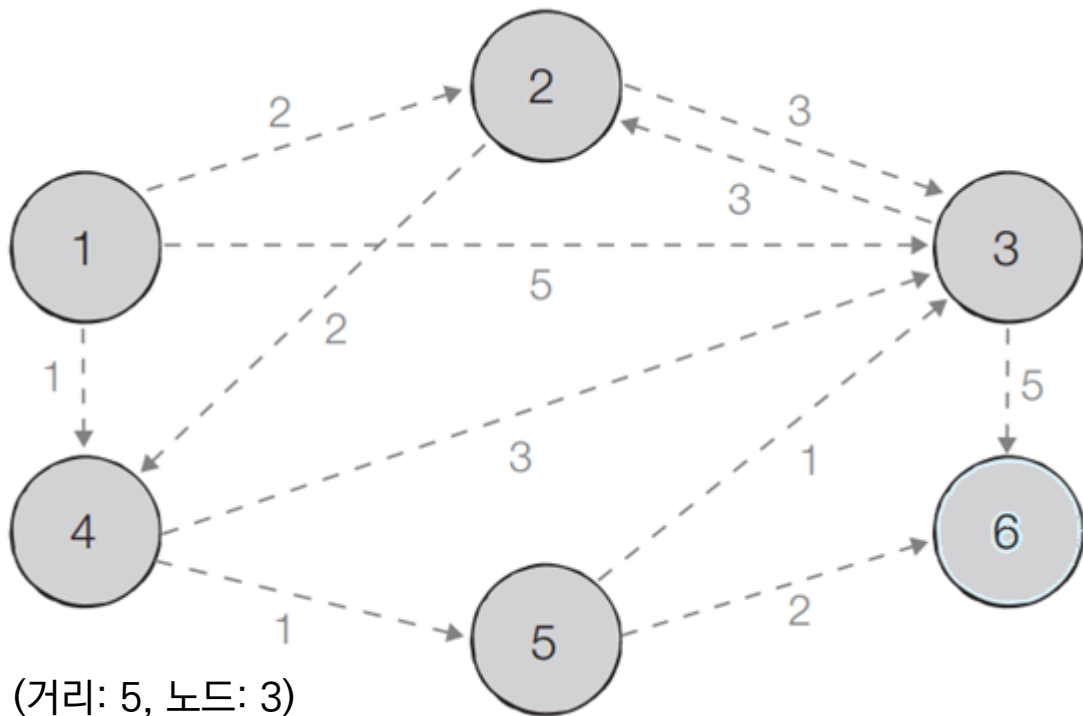
- [Step 8] 우선순위 큐에서 원소를 꺼냅니다. 3번 노드는 이미 방문했으므로 무시합니다.

출발: 1번 노드

■ : 처리 중인 노드

■ : 이미 방문한 노드

우선순위 큐



현재 꺼낸 원소: (거리: 5, 노드: 3)

노드 번호	1	2	3	4	5	6
거리	0	2	3	1	2	4

다익스트라 알고리즘: 개선된 구현 방법 (Python)

```
import heapq
import sys
input = sys.stdin.readline
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

# 노드의 개수, 간선의 개수를 입력받기
n, m = map(int, input().split())
# 시작 노드 번호를 입력받기
start = int(input())
# 각 노드에 연결되어 있는 노드에 대한 정보를 담는 리스트를 만들기
graph = [[] for i in range(n + 1)]
# 최단 거리 테이블을 모두 무한으로 초기화
distance = [INF] * (n + 1)

# 모든 간선 정보를 입력받기
for _ in range(m):
    a, b, c = map(int, input().split())
    # a번 노드에서 b번 노드로 가는 비용이 c라는 의미
    graph[a].append((b, c))
```

```
def dijkstra(start):
    q = []
    # 시작 노드로 가기 위한 최단 거리는 0으로 설정하여, 큐에 삽입
    heapq.heappush(q, (0, start))
    distance[start] = 0
    while q: # 큐가 비어있지 않다면
        # 가장 최단 거리가 짧은 노드에 대한 정보 꺼내기
        dist, now = heapq.heappop(q)
        # 현재 노드가 이미 처리된 적이 있는 노드라면 무시
        if distance[now] < dist:
            continue
        # 현재 노드와 연결된 다른 인접한 노드들을 확인
        for i in graph[now]:
            cost = dist + i[1]
            # 현재 노드를 거쳐서, 다른 노드로 이동하는 거리가 더 짧은 경우
            if cost < distance[i[0]]:
                distance[i[0]] = cost
                heapq.heappush(q, (cost, i[0]))

# 다익스트라 알고리즘을 수행
dijkstra(start)

# 모든 노드로 가기 위한 최단 거리를 출력
for i in range(1, n + 1):
    # 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
    if distance[i] == INF:
        print("INFINITY")
    # 도달할 수 있는 경우 거리를 출력
    else:
        print(distance[i])
```

다익스트라 알고리즘: 개선된 구현 방법 성능 분석

- 힙 자료구조를 이용하는 다익스트라 알고리즘의 시간 복잡도는 $O(E \log V)$ 입니다.
- 노드를 하나씩 꺼내 검사하는 반복문(while문)은 노드의 개수 V 이상의 횟수로는 처리되지 않습니다.
 - 결과적으로 현재 우선순위 큐에서 꺼낸 노드와 연결된 다른 노드들을 확인하는 총횟수는 최대 간선의 개수(E)만큼 연산이 수행될 수 있습니다.
- 직관적으로 전체 과정은 E 개의 원소를 우선순위 큐에 넣었다가 모두 빼내는 연산과 매우 유사합니다.
 - 시간 복잡도를 $O(E \log E)$ 로 판단할 수 있습니다.
 - 중복 간선을 포함하지 않는 경우에 이를 $O(E \log V)$ 로 정리할 수 있습니다.
 - $O(E \log E) \rightarrow O(E \log V^2) \rightarrow O(2E \log V) \rightarrow O(E \log V)$

플로이드 워셜 알고리즘 개요

- 모든 노드에서 다른 모든 노드까지의 최단 경로를 모두 계산합니다.
- 플로이드 워셜(Floyd-Warshall) 알고리즘은 다익스트라 알고리즘과 마찬가지로 단계별로 거쳐 가는 노드를 기준으로 알고리즘을 수행합니다.
 - 다만 매 단계마다 방문하지 않은 노드 중에 최단 거리를 갖는 노드를 찾는 과정이 필요하지 않습니다.
- 플로이드 워셜은 2차원 테이블에 최단 거리 정보를 저장합니다.
- 플로이드 워셜 알고리즘은 다이나믹 프로그래밍 유형에 속합니다.

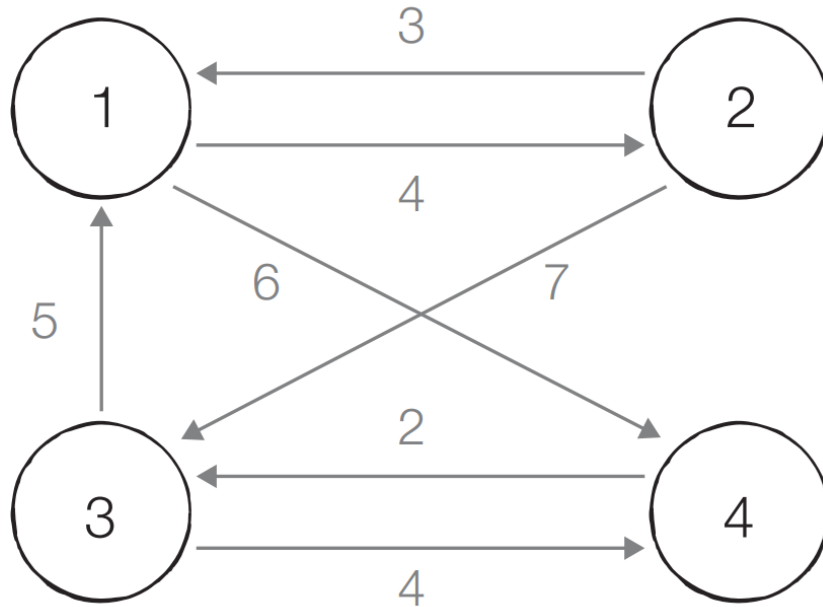
플로이드 워셜 알고리즘

- 각 단계마다 특정한 노드 k 를 거쳐 가는 경우를 확인합니다.
 - a 에서 b 로 가는 최단 거리보다 a 에서 k 를 거쳐 b 로 가는 거리가 더 짧은지 검사합니다.
- 점화식은 다음과 같습니다.

$$D_{ab} = \min(D_{ab}, D_{ak} + D_{kb})$$

플로이드 워셜 알고리즘: 동작 과정 살펴보기

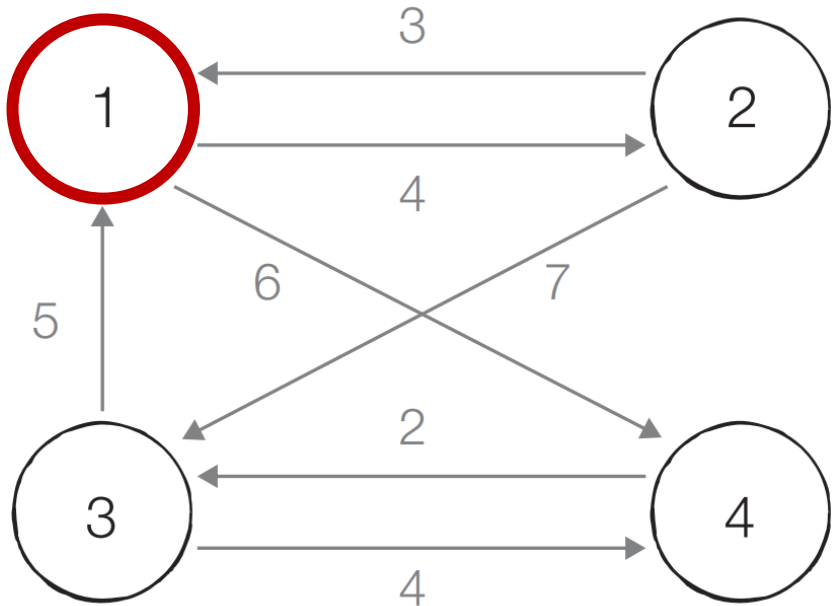
- [초기 상태] 그래프를 준비하고 최단 거리 테이블을 초기화합니다.
 - 기본 점화식: $D_{ab} = \min(D_{ab}, D_{ak} + D_{kb})$



도착 \ 출발	1번	2번	3번	4번
1번	0	4	무한	6
2번	3	0	7	무한
3번	5	무한	0	4
4번	무한	무한	2	0

플로이드 워셜 알고리즘: 동작 과정 살펴보기

- [Step 1] 1번 노드를 거쳐 가는 경우를 고려하여 테이블을 갱신합니다.
 - 점화식: $D_{ab} = \min(D_{ab}, D_{a1} + D_{1b})$



$$D_{23} = \min(D_{23}, D_{21} + D_{13})$$

$$D_{24} = \min(D_{24}, D_{21} + D_{14})$$

$$D_{32} = \min(D_{32}, D_{31} + D_{12})$$

$$D_{34} = \min(D_{34}, D_{31} + D_{14})$$

$$D_{42} = \min(D_{42}, D_{41} + D_{12})$$

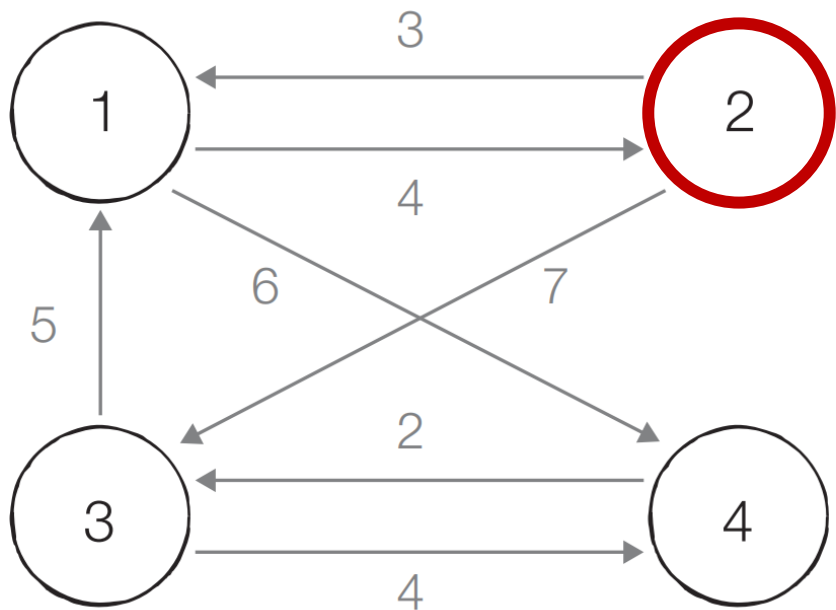
$$D_{43} = \min(D_{43}, D_{41} + D_{13})$$

갱신된 최단 거리 테이블

0	4	무한	6
3	0	7	9
5	9	0	4
무한	무한	2	0

플로이드 워셜 알고리즘: 동작 과정 살펴보기

- [Step 2] 2번 노드를 거쳐 가는 경우를 고려하여 테이블을 갱신합니다.
 - 점화식: $D_{ab} = \min(D_{ab}, D_{a2} + D_{2b})$



$$D_{13} = \min(D_{13}, D_{12} + D_{23})$$

$$D_{14} = \min(D_{14}, D_{12} + D_{24})$$

$$D_{31} = \min(D_{31}, D_{32} + D_{21})$$

$$D_{34} = \min(D_{34}, D_{32} + D_{24})$$

$$D_{41} = \min(D_{41}, D_{42} + D_{21})$$

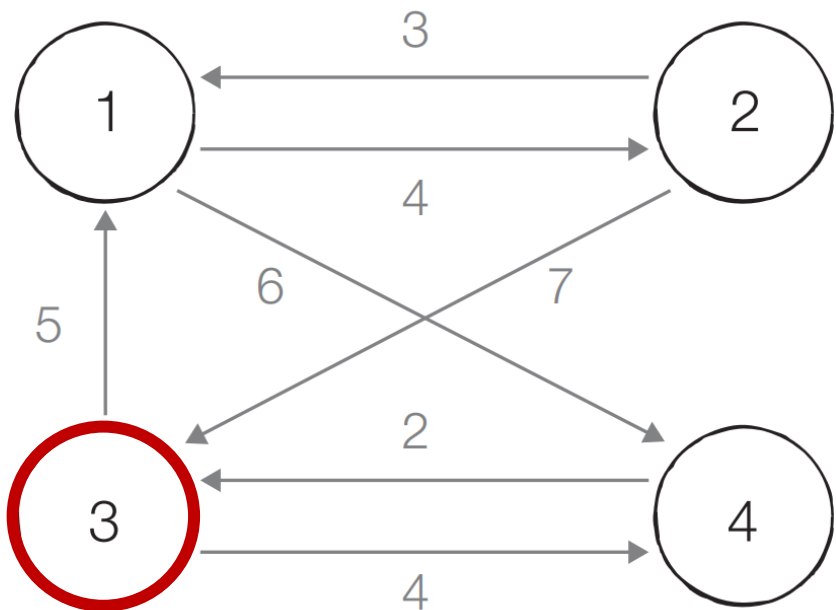
$$D_{43} = \min(D_{43}, D_{42} + D_{23})$$

갱신된 최단 거리 테이블

0	4	11	6
3	0	7	9
5	9	0	4
무한	무한	2	0

플로이드 워셜 알고리즘: 동작 과정 살펴보기

- [Step 3] 3번 노드를 거쳐 가는 경우를 고려하여 테이블을 갱신합니다.
 - 점화식: $D_{ab} = \min(D_{ab}, D_{a3} + D_{3b})$



$$D_{12} = \min(D_{12}, D_{13} + D_{32})$$

$$D_{14} = \min(D_{14}, D_{13} + D_{34})$$

$$D_{21} = \min(D_{21}, D_{23} + D_{31})$$

$$D_{24} = \min(D_{24}, D_{23} + D_{34})$$

$$D_{41} = \min(D_{41}, D_{43} + D_{31})$$

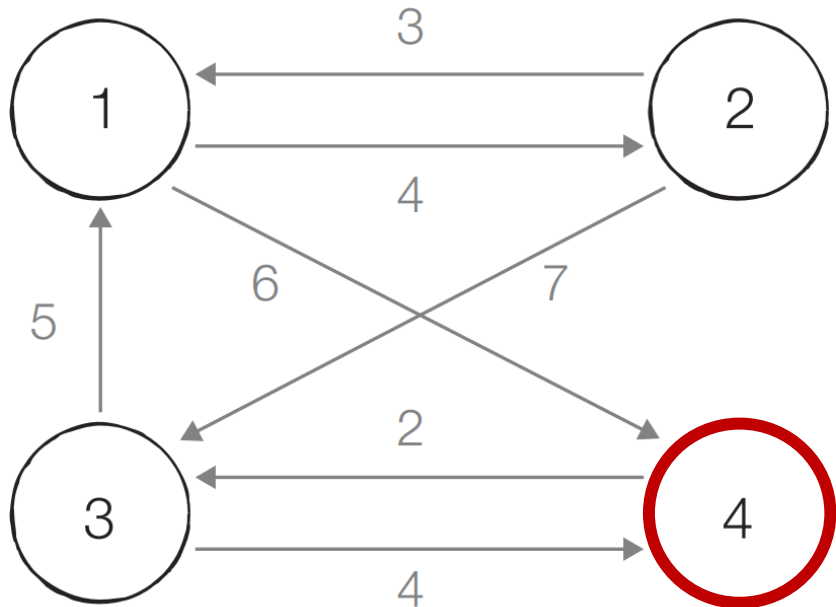
$$D_{42} = \min(D_{42}, D_{43} + D_{32})$$

갱신된 최단 거리 테이블

0	4	11	6
3	0	7	9
5	9	0	4
7	11	2	0

플로이드 워셜 알고리즘: 동작 과정 살펴보기

- [Step 4] 4번 노드를 거쳐 가는 경우를 고려하여 테이블을 갱신합니다.
 - 점화식: $D_{ab} = \min(D_{ab}, D_{a4} + D_{4b})$



$$D_{12} = \min(D_{12}, D_{14} + D_{42})$$

$$D_{13} = \min(D_{13}, D_{14} + D_{43})$$

$$D_{21} = \min(D_{21}, D_{24} + D_{41})$$

$$D_{23} = \min(D_{23}, D_{24} + D_{43})$$

$$D_{31} = \min(D_{31}, D_{34} + D_{41})$$

$$D_{32} = \min(D_{32}, D_{34} + D_{42})$$

갱신된 최단 거리 테이블

0	4	8	6
3	0	7	9
5	9	0	4
7	11	2	0

플로이드 워셜 알고리즘

```
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

# 노드의 개수 및 간선의 개수를 입력받기
n = int(input())
m = int(input())
# 2차원 리스트(그래프 표현)를 만들고, 무한으로 초기화
graph = [[INF] * (n + 1) for _ in range(n + 1)]

# 자기 자신에서 자기 자신으로 가는 비용은 0으로 초기화
for a in range(1, n + 1):
    for b in range(1, n + 1):
        if a == b:
            graph[a][b] = 0

# 각 간선에 대한 정보를 입력 받아, 그 값으로 초기화
for _ in range(m):
    # A에서 B로 가는 비용은 c라고 설정
    a, b, c = map(int, input().split())
    graph[a][b] = c
```

```
# 점화식에 따라 플로이드 워셜 알고리즘을 수행
for k in range(1, n + 1):
    for a in range(1, n + 1):
        for b in range(1, n + 1):
            graph[a][b] = min(graph[a][b], graph[a][k] + graph[k][b])

# 수행된 결과를 출력
for a in range(1, n + 1):
    for b in range(1, n + 1):
        # 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
        if graph[a][b] == INF:
            print("INFINITY", end=" ")
        # 도달할 수 있는 경우 거리를 출력
        else:
            print(graph[a][b], end=" ")
    print()
```

플로이드 워셜 알고리즘 성능 분석

- 노드의 개수가 N 개일 때 알고리즘상으로 N 번의 단계를 수행합니다.
 - 각 단계마다 $O(N^2)$ 의 연산을 통해 현재 노드를 거쳐 가는 모든 경로를 고려합니다.
- 따라서 플로이드 워셜 알고리즘의 총 시간 복잡도는 $O(N^3)$ 입니다.

〈문제〉 전보: 문제 설명

- 어떤 나라에는 **N개의 도시**가 있다. 그리고 각 도시는 보내고자 하는 메시지가 있는 경우, 다른 도시로 전보를 보내서 다른 도시로 해당 메시지를 전송할 수 있다.
- 하지만 X라는 도시에서 Y라는 도시로 전보를 보내고자 한다면, 도시 X에서 Y로 향하는 통로가 설치되어 있어야 한다. 예를 들어 X에서 Y로 향하는 통로는 있지만, Y에서 X로 향하는 통로가 없다면 Y는 X로 메시지를 보낼 수 없다. 또한 통로를 거쳐 메시지를 보낼 때는 일정 시간이 소요된다.
- 어느 날 C라는 도시에서 위급 상황이 발생했다. 그래서 최대한 많은 도시로 메시지를 보내고자 한다. 메시지는 도시 C에서 **출발하여 각 도시 사이에 설치된 통로를 거쳐, 최대한 많이 퍼져나갈 것이다.**
- 각 도시의 번호와 통로가 설치되어 있는 정보가 주어졌을 때, 도시 C에서 보낸 메시지를 받게 되는 도시의 개수는 총 몇 개이며 도시들이 모두 메시지를 받는 데까지 걸리는 시간은 얼마인지 계산하는 프로그램을 작성하시오.

〈문제〉 전보: 문제 조건

난이도 ●●● | 풀이 시간 60분 | 시간 제한 1초 | 메모리 제한 128MB

입력 조건

- 첫째 줄에 도시의 개수 N , 통로의 개수 M , 메시지를 보내고자 하는 도시 C 가 주어진다.
($1 \leq N \leq 30,000$, $1 \leq M \leq 200,000$, $1 \leq C \leq N$)
- 둘째 줄부터 $M + 1$ 번째 줄에 걸쳐서 통로에 대한 정보 X, Y, Z 가 주어진다. 이는 특정 도시 X 에서 다른 특정 도시 Y 로 이어지는 통로가 있으며, 메시지가 전달되는 시간이 Z 라는 의미다.
($1 \leq X, Y \leq N$, $1 \leq Z \leq 1,000$)

출력 조건

- 첫째 줄에 도시 C 에서 보낸 메시지를 받는 도시의 총 개수와 총 걸리는 시간을 공백으로 구분하여 출력한다.

입력 예시

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

출력 예시

```
2 4
```

〈문제〉 전보: 문제 해결 아이디어

- **핵심 아이디어:** 한 도시에서 다른 도시까지의 **최단 거리 문제**로 치환할 수 있습니다.
- N 과 M 의 범위가 충분히 크기 때문에 우선순위 큐를 활용한 다익스트라 알고리즘을 구현합니다.

〈문제〉 전보: 답안 예시 (Python)

```
import heapq
import sys
input = sys.stdin.readline
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

def dijkstra(start):
    q = []
    # 시작 노드로 가기 위한 최단 거리는 0으로 설정하여, 큐에 삽입
    heapq.heappush(q, (0, start))
    distance[start] = 0
    while q: # 큐가 비어있지 않다면
        # 가장 최단 거리가 짧은 노드에 대한 정보를 꺼내기
        dist, now = heapq.heappop(q)
        if distance[now] < dist:
            continue
        # 현재 노드와 연결된 다른 인접한 노드들을 확인
        for i in graph[now]:
            cost = dist + i[1]
            # 현재 노드를 거쳐서, 다른 노드로 이동하는 거리가 더 짧은 경우
            if cost < distance[i[0]]:
                distance[i[0]] = cost
                heapq.heappush(q, (cost, i[0]))
```

```
# 노드의 개수, 간선의 개수, 시작 노드를 입력받기
n, m, start = map(int, input().split())
# 각 노드에 연결되어 있는 노드에 대한 정보를 담는 리스트를 만들기
graph = [[] for i in range(n + 1)]
# 최단 거리 테이블을 모두 무한으로 초기화
distance = [INF] * (n + 1)

# 모든 간선 정보를 입력받기
for _ in range(m):
    x, y, z = map(int, input().split())
    # x번 노드에서 y번 노드로 가는 비용이 z라는 의미
    graph[x].append((y, z))

# 다익스트라 알고리즘을 수행
dijkstra(start)

# 도달할 수 있는 노드의 개수
count = 0
# 도달할 수 있는 노드 중에서, 가장 멀리 있는 노드와의 최단 거리
max_distance = 0
for d in distance:
    # 도달할 수 있는 노드인 경우
    if d != 1e9:
        count += 1
        max_distance = max(max_distance, d)

# 시작 노드는 제외해야 하므로 count - 1을 출력
print(count - 1, max_distance)
```

〈문제〉 미래 도시: 문제 설명

- 미래 도시에는 1번부터 N번까지의 회사가 있는데 특정 회사끼리는 서로 도로를 통해 연결되어 있다. 방문 판매원 A는 현재 1번 회사에 위치해 있으며, X번 회사에 방문해 물건을 판매하고자 한다.
- 미래 도시에서 특정 회사에 도착하기 위한 방법은 회사끼리 연결되어 있는 도로를 이용하는 방법이 유일하다. 또한 연결된 2개의 회사는 **양방향**으로 이동할 수 있다. 공중 미래 도시에서 특정 회사와 다른 회사가 도로로 연결되어 있다면, 정확히 1만큼의 시간으로 이동할 수 있다.
- 또한 오늘 방문 판매원 A는 기대하던 소개팅에도 참석하고자 한다. 소개팅의 상대는 K번 회사에 존재한다. 방문 판매원 A는 X번 회사에 가서 물건을 판매하기 전에 먼저 소개팅 상대의 회사에 찾아가서 함께 커피를 마실 예정이다. 따라서 방문 판매원 A는 **1번 회사에서 출발하여 K번 회사를 방문한 뒤에 X번 회사로 가는 것이 목표**다. 이때 방문 판매원 A는 가능한 한 빠르게 이동하고자 한다.
- 방문 판매원이 회사 사이를 이동하게 되는 **최소 시간**을 계산하는 프로그램을 작성하시오.

〈문제〉 미래 도시: 문제 조건

난이도 ●●○ | 풀이 시간 40분 | 시간 제한 1초 | 메모리 제한 128MB

입력 조건

- 첫째 줄에 전체 회사의 개수 N 과 경로의 개수 M 이 공백으로 구분되어 차례대로 주어진다. ($1 \leq N, M \leq 100$)
- 둘째 줄부터 $M + 1$ 번째 줄에는 연결된 두 회사의 번호가 공백으로 구분되어 주어진다.
- $M + 2$ 번째 줄에는 X 와 K 가 공백으로 구분되어 차례대로 주어진다. ($1 \leq K \leq 100$)

출력 조건

- 첫째 줄에 방문 판매원 A 가 K 번 회사를 거쳐 X 번 회사로 가는 최소 이동 시간을 출력한다.
- 만약 X 번 회사에 도달할 수 없다면 -1 을 출력한다.

입력 예시 1

```
5 7
1 2
1 3
1 4
2 4
3 4
3 5
4 5
4 5
```

출력 예시 1

```
3
```

〈문제〉 미래 도시: 문제 해결 아이디어

- **핵심 아이디어:** 전형적인 최단 거리 문제이므로 **최단 거리 알고리즘을 이용해 해결합니다.**
- N의 크기가 최대 100이므로 플로이드 워셜 알고리즘을 이용해도 효율적으로 해결할 수 있습니다.
- 플로이드 워셜 알고리즘을 수행한 뒤에 **(1번 노드에서 X까지의 최단 거리 + X에서 K까지의 최단 거리)**를 계산하여 출력하면 정답 판정을 받을 수 있습니다.

〈문제〉 미래 도시: 답안 예시 (Python)

```
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

# 노드의 개수 및 간선의 개수를 입력받기
n, m = map(int, input().split())
# 2차원 리스트(그래프 표현)를 만들고, 모든 값을 무한으로 초기화
graph = [[INF] * (n + 1) for _ in range(n + 1)]

# 자기 자신에서 자기 자신으로 가는 비용은 0으로 초기화
for a in range(1, n + 1):
    for b in range(1, n + 1):
        if a == b:
            graph[a][b] = 0

# 각 간선에 대한 정보를 입력 받아, 그 값으로 초기화
for _ in range(m):
    # A와 B가 서로에게 가는 비용은 1이라고 설정
    a, b = map(int, input().split())
    graph[a][b] = 1
    graph[b][a] = 1
```

```
# 거쳐 갈 노드 x와 최종 목적지 노드 k를 입력받기
x, k = map(int, input().split())

# 점화식에 따라 플로이드 워셜 알고리즘을 수행
for k in range(1, n + 1):
    for a in range(1, n + 1):
        for b in range(1, n + 1):
            graph[a][b] = min(graph[a][b], graph[a][k] + graph[k][b])

# 수행된 결과를 출력
distance = graph[1][k] + graph[k][x]

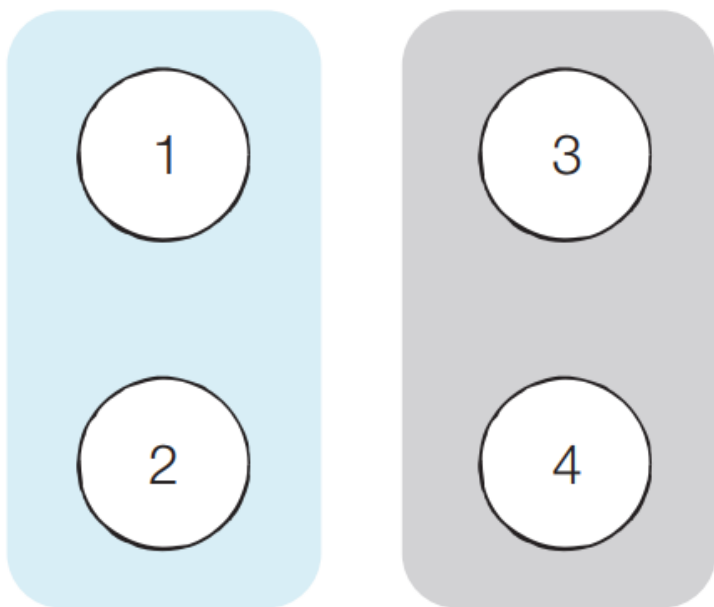
# 도달할 수 없는 경우, -1을 출력
if distance >= INF:
    print("-1")
# 도달할 수 있다면, 최단 거리를 출력
else:
    print(distance)
```

기타 그래프 이론

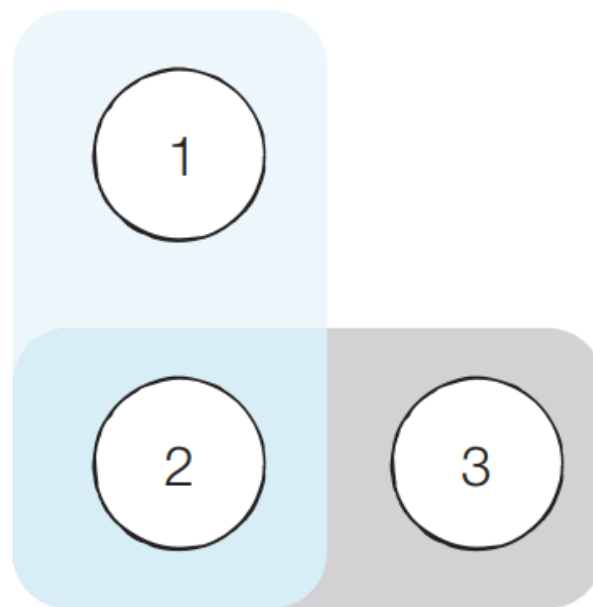
서로소 집합

- 서로소 집합(Disjoint Sets)란 공통 원소가 없는 두 집합을 의미합니다.

$\{1, 2\}$ 와 $\{3, 4\}$ 는 서로소 관계이다.



$\{1, 2\}$ 와 $\{2, 3\}$ 은 서로소 관계가 아니다.



서로소 집합 자료구조

- 서로소 부분 집합들로 나뉘어진 원소들의 데이터를 처리하기 위한 자료구조입니다.
- 서로소 집합 자료구조는 두 종류의 연산을 지원합니다.
 - **합집합(Union):** 두 개의 원소가 포함된 집합을 하나의 집합으로 합치는 연산입니다.
 - **찾기(Find):** 특정한 원소가 속한 집합이 어떤 집합인지 알려주는 연산입니다.
- 서로소 집합 자료구조는 **합치기 찾기(Union Find) 자료구조**라고 불리기도 합니다.

서로소 집합 자료구조

- 여러 개의 합치기 연산이 주어졌을 때 서로소 집합 자료구조의 동작 과정은 다음과 같습니다.
 1. 합집합(Union) 연산을 확인하여, 서로 연결된 두 노드 A, B를 확인합니다.
 - 1) A와 B의 루트 노드 A', B'를 각각 찾습니다.
 - 2) A'를 B'의 부모 노드로 설정합니다.
 2. 모든 합집합(Union) 연산을 처리할 때까지 1번의 과정을 반복합니다.

서로소 집합 자료구조: 동작 과정 살펴보기

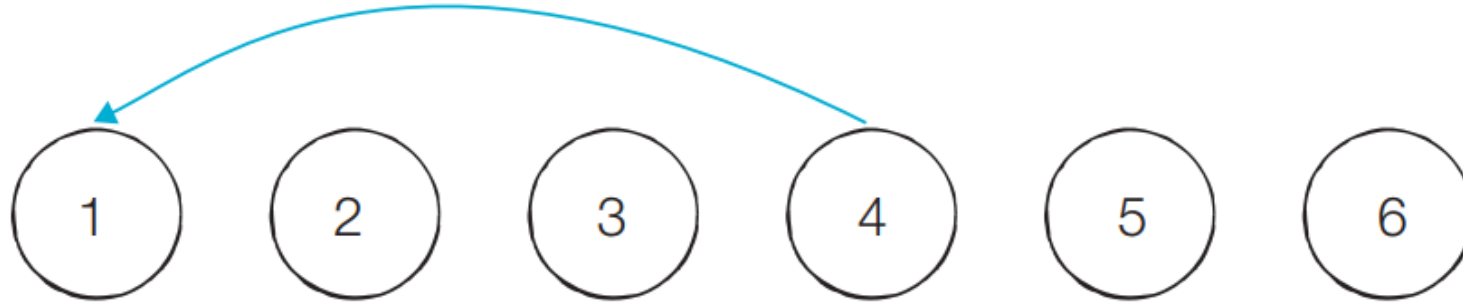
- 처리할 연산들: $Union(1, 4)$, $Union(2, 3)$, $Union(2, 4)$, $Union(5, 6)$
- [초기 단계] 노드의 개수 크기의 부모 테이블을 초기화합니다.



노드 번호	1	2	3	4	5	6
부모	1	2	3	4	5	6

서로소 집합 자료구조: 동작 과정 살펴보기

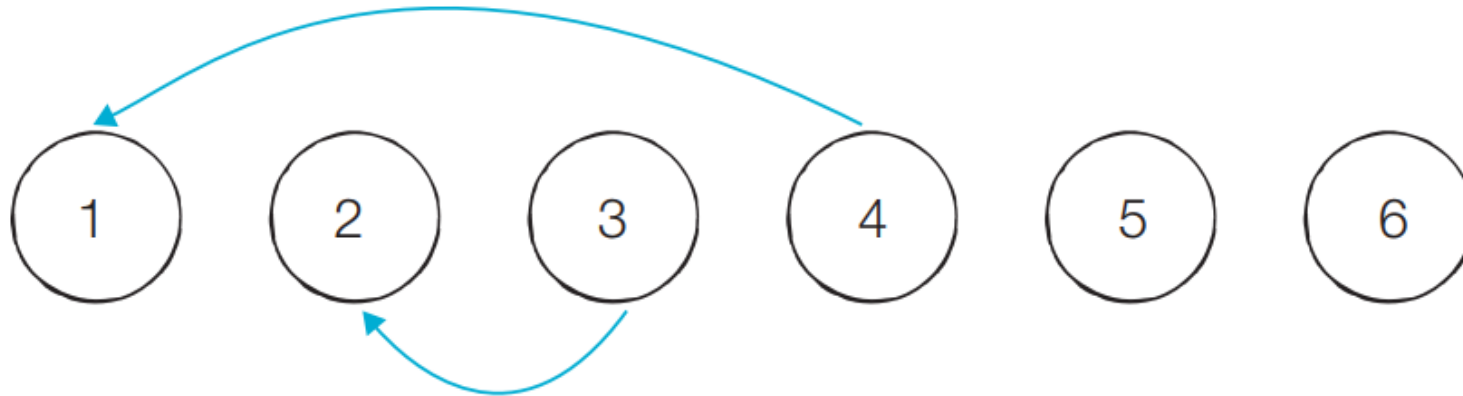
- 처리할 연산들: $Union(1, 4)$, $Union(2, 3)$, $Union(2, 4)$, $Union(5, 6)$
- [Step 1] 노드 1과 노드 4의 루트 노드를 각각 찾습니다. 현재 루트 노드는 각각 1과 4이므로 더 큰 번호에 해당하는 루트 노드 4의 부모를 1로 설정합니다.



노드 번호	1	2	3	4	5	6
부모	1	2	3	1	5	6

서로소 집합 자료구조: 동작 과정 살펴보기

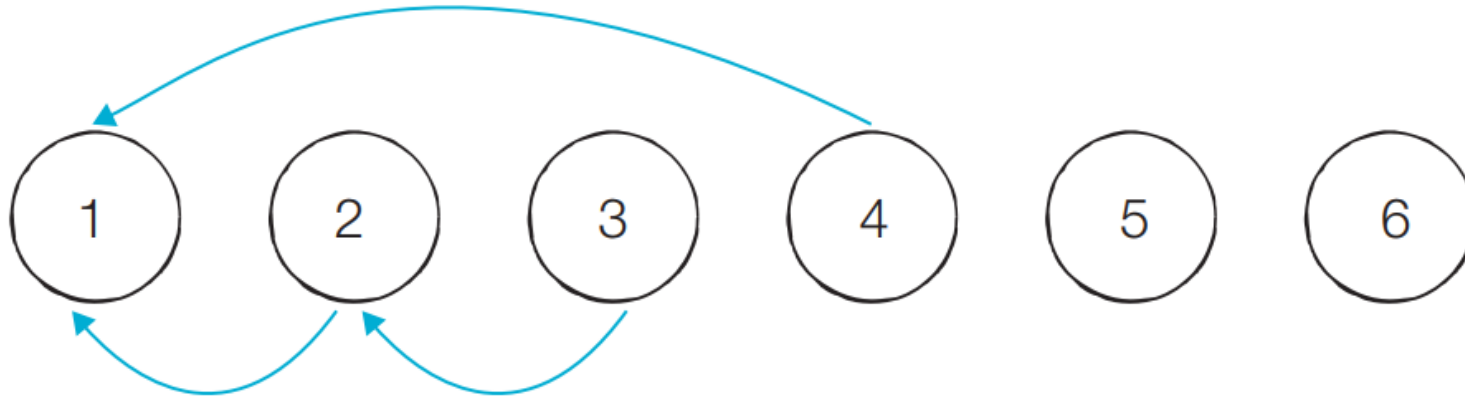
- 처리할 연산들: $Union(1, 4)$, $Union(2, 3)$, $Union(2, 4)$, $Union(5, 6)$
- [Step 2] 노드 2과 노드 3의 루트 노드를 각각 찾습니다. 현재 루트 노드는 각각 2와 3이므로 더 큰 번호에 해당하는 루트 노드 3의 부모를 2로 설정합니다.



노드 번호	1	2	3	4	5	6
부모	1	2	2	1	5	6

서로소 집합 자료구조: 동작 과정 살펴보기

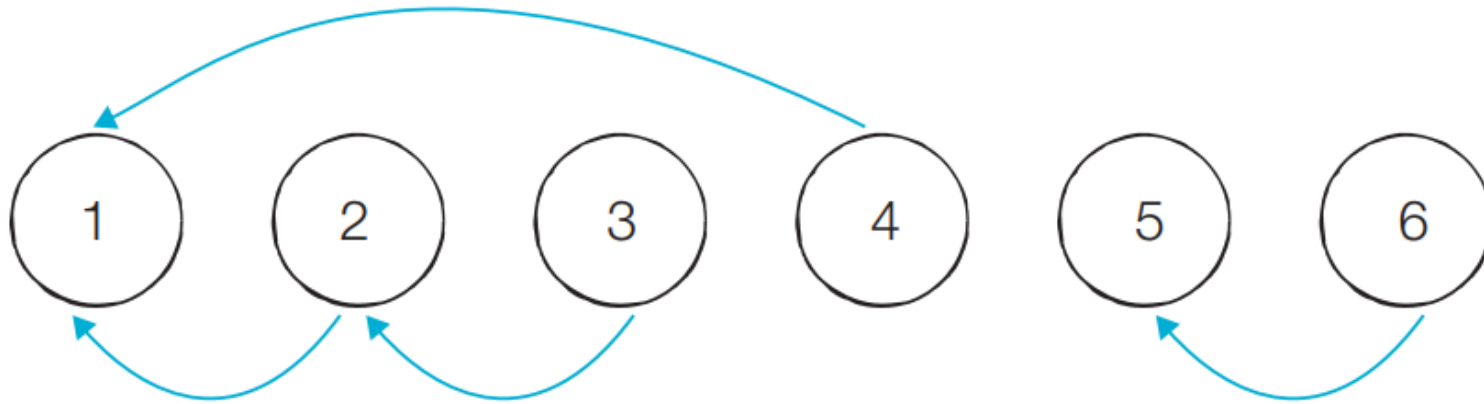
- 처리할 연산들: $Union(1, 4)$, $Union(2, 3)$, $Union(2, 4)$, $Union(5, 6)$
- [Step 3] 노드 2와 노드 4의 루트 노드를 각각 찾습니다. 현재 루트 노드는 각각 2와 1이므로 더 큰 번호에 해당하는 루트 노드 2의 부모를 1로 설정합니다.



노드 번호	1	2	3	4	5	6
부모	1	1	2	1	5	6

서로소 집합 자료구조: 동작 과정 살펴보기

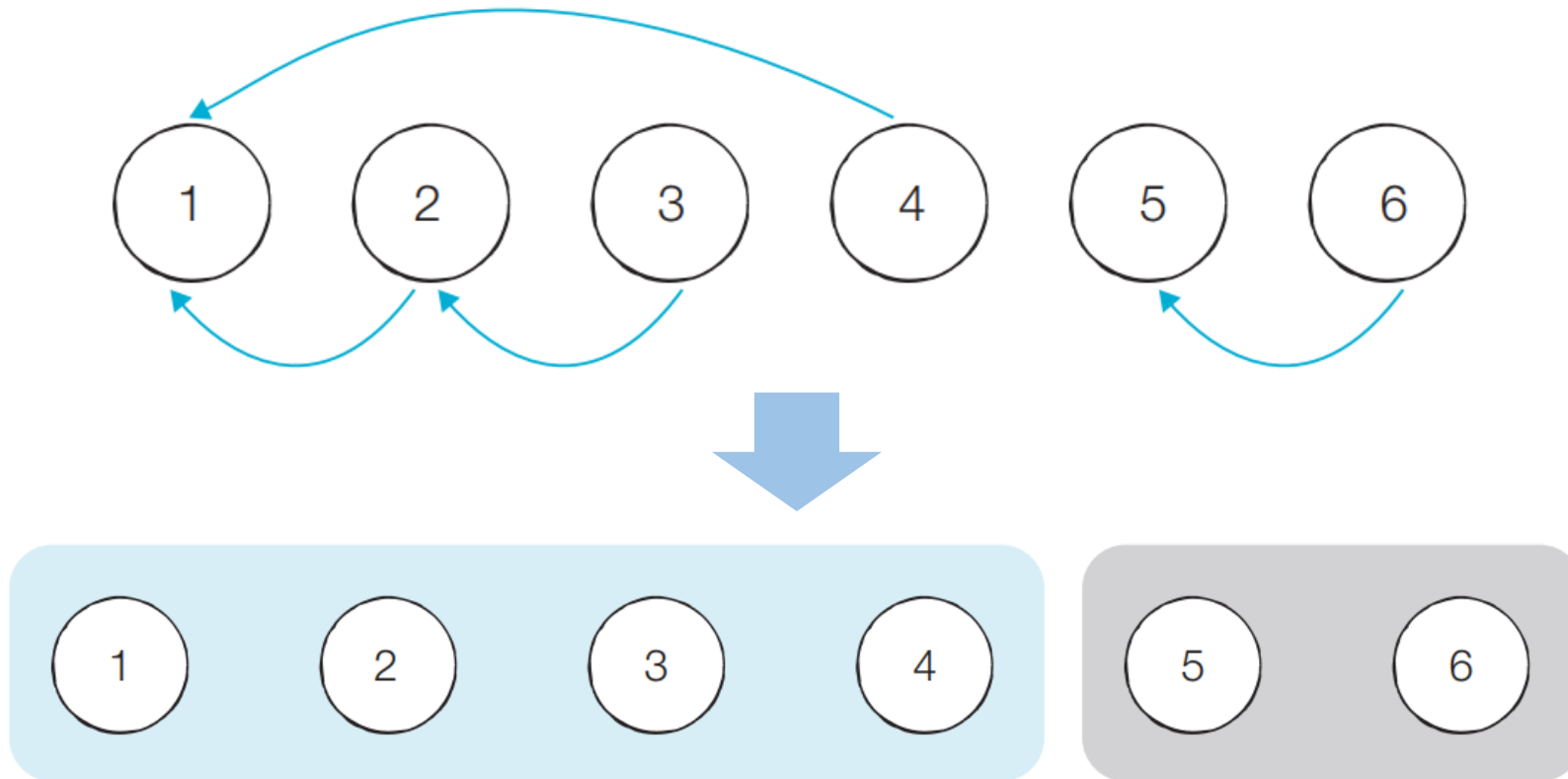
- 처리할 연산들: $Union(1, 4)$, $Union(2, 3)$, $Union(2, 4)$, $Union(5, 6)$
- [Step 4] 노드 5와 노드 6의 루트 노드를 각각 찾습니다. 현재 루트 노드는 각각 5와 6이므로 더 큰 번호에 해당하는 루트 노드 6의 부모를 5로 설정합니다.



노드 번호	1	2	3	4	5	6
부모	1	1	2	1	5	5

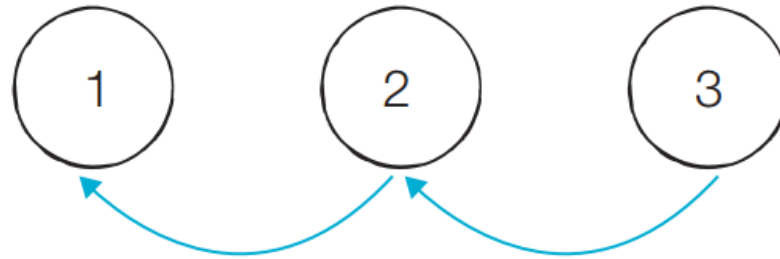
서로소 집합 자료구조: 연결성

- 서로소 집합 자료구조에서는 **연결성**을 통해 손쉽게 집합의 형태를 확인할 수 있습니다.



서로소 집합 자료구조: 연결성

- 기본적인 형태의 서로소 집합 자료구조에서는 루트 노드에 즉시 접근할 수 없습니다.
 - 루트 노드를 찾기 위해 부모 테이블을 계속해서 확인하며 거슬러 올라가야 합니다.
- 다음 예시에서 노드 3의 루트를 찾기 위해서는 노드 2를 거쳐 노드 1에 접근해야 합니다.



노드 번호	1	2	3
부모	1	1	2

서로소 집합 자료구조: 기본적인 구현 방법 (Python)

```
# 특정 원소가 속한 집합을 찾기
def find_parent(parent, x):
    # 루트 노드를 찾을 때까지 재귀 호출
    if parent[x] != x:
        return find_parent(parent, parent[x])
    return x

# 두 원소가 속한 집합을 합치기
def union_parent(parent, a, b):
    a = find_parent(parent, a)
    b = find_parent(parent, b)
    if a < b:
        parent[b] = a
    else:
        parent[a] = b
```

```
# 노드의 개수와 간선(Union 연산)의 개수 입력 받기
v, e = map(int, input().split())
parent = [0] * (v + 1) # 부모 테이블 초기화하기

# 부모 테이블상에서, 부모를 자기 자신으로 초기화
for i in range(1, v + 1):
    parent[i] = i

# Union 연산을 각각 수행
for i in range(e):
    a, b = map(int, input().split())
    union_parent(parent, a, b)

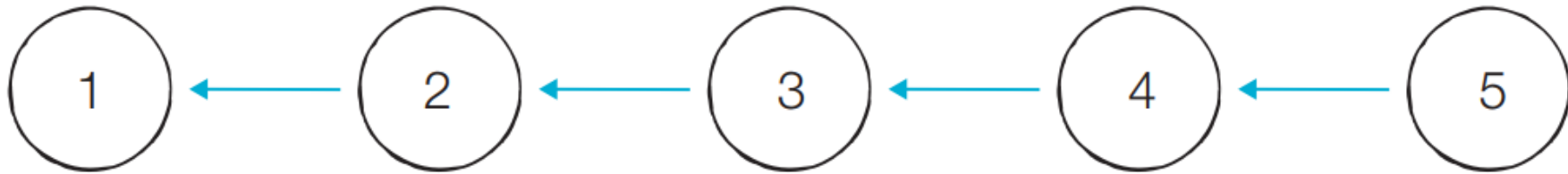
# 각 원소가 속한 집합 출력하기
print('각 원소가 속한 집합: ', end='')
for i in range(1, v + 1):
    print(find_parent(parent, i), end=' ')

print()

# 부모 테이블 내용 출력하기
print('부모 테이블: ', end='')
for i in range(1, v + 1):
    print(parent[i], end=' ')
```

서로소 집합 자료구조: 기본적인 구현 방법의 문제점

- 합집합(Union) 연산이 편향되게 이루어지는 경우 찾기(Find) 함수가 비효율적으로 동작합니다.
- 최악의 경우에는 찾기(Find) 함수가 모든 노드를 다 확인하게 되어 시간 복잡도가 $O(V)$ 입니다.
 - 다음과 같이 {1, 2, 3, 4, 5}의 총 5개의 원소가 존재하는 상황을 확인해 봅시다.
 - 수행된 연산들: $Union(4,5)$, $Union(3,4)$, $Union(2,3)$, $Union(1,2)$



노드 번호	1	2	3	4	5
부모	1	1	2	3	4

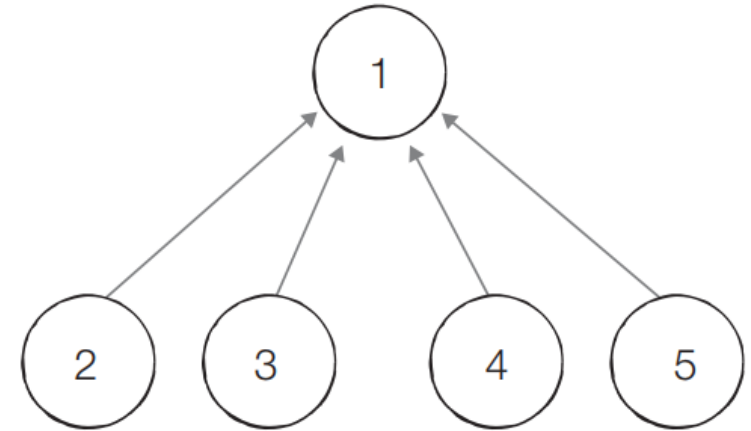
서로소 집합 자료구조: 경로 압축

- 찾기(Find) 함수를 최적화하기 위한 방법으로 **경로 압축(Path Compression)**을 이용할 수 있습니다.
 - 찾기(Find) 함수를 재귀적으로 호출한 뒤에 부모 테이블 값을 바로 갱신합니다.

```
# 특정 원소가 속한 집합을 찾기
def find_parent(parent, x):
    # 루트 노드가 아니라면, 루트 노드를 찾을 때까지 재귀적으로 호출
    if parent[x] != x:
        parent[x] = find_parent(parent, parent[x])
    return parent[x]
```

서로소 집합 자료구조: 경로 압축

- 경로 압축 기법을 적용하면 각 노드에 대하여 찾기(Find) 함수를 호출한 이후에 해당 노드의 루트 노드가 바로 부모 노드가 됩니다.
- 동일한 예시에 대해서 모든 합집합(Union) 함수를 처리한 후 각 원소에 대하여 찾기(Find) 함수를 수행하면 다음과 같이 부모 테이블이 갱신됩니다.
- 기본적인 방법에 비하여 시간 복잡도가 개선됩니다.



노드 번호	1	2	3	4	5
부모	1	1	1	1	1

서로소 집합 자료구조: 경로 압축 (Python)

```
# 특정 원소가 속한 집합을 찾기
def find_parent(parent, x):
    # 루트 노드를 찾을 때까지 재귀 호출
    if parent[x] != x:
        parent[x] = find_parent(parent, parent[x])
    return parent[x]

# 두 원소가 속한 집합을 합치기
def union_parent(parent, a, b):
    a = find_parent(parent, a)
    b = find_parent(parent, b)
    if a < b:
        parent[b] = a
    else:
        parent[a] = b
```

```
# 노드의 개수와 간선(Union 연산)의 개수 입력 받기
v, e = map(int, input().split())
parent = [0] * (v + 1) # 부모 테이블 초기화하기

# 부모 테이블상에서, 부모를 자기 자신으로 초기화
for i in range(1, v + 1):
    parent[i] = i

# Union 연산을 각각 수행
for i in range(e):
    a, b = map(int, input().split())
    union_parent(parent, a, b)

# 각 원소가 속한 집합 출력하기
print('각 원소가 속한 집합: ', end='')
for i in range(1, v + 1):
    print(find_parent(parent, i), end=' ')

print()

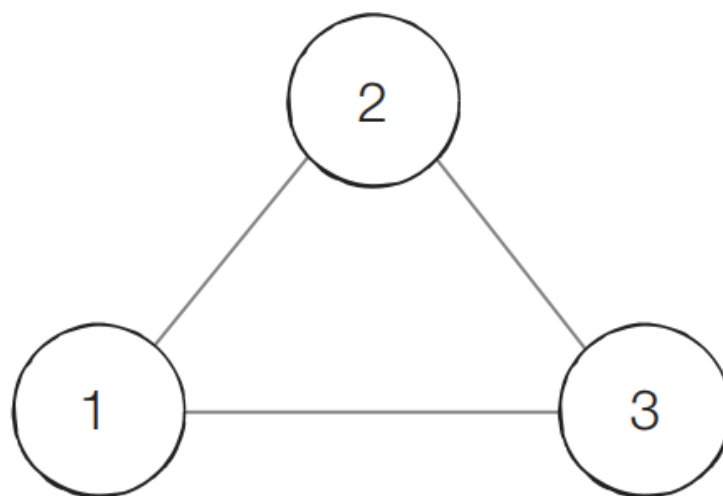
# 부모 테이블 내용 출력하기
print('부모 테이블: ', end='')
for i in range(1, v + 1):
    print(parent[i], end=' ')
```

서로소 집합을 활용한 사이클 판별

- 서로소 집합은 무방향 그래프 내에서의 사이클을 판별할 때 사용할 수 있습니다.
 - 참고로 방향 그래프에서의 사이클 여부는 DFS를 이용하여 판별할 수 있습니다.
- 사이클 판별 알고리즘은 다음과 같습니다.
 1. 각 간선을 하나씩 확인하며 두 노드의 루트 노드를 확인합니다.
 - 1) 루트 노드가 서로 다르다면 두 노드에 대하여 합집합(Union) 연산을 수행합니다.
 - 2) 루트 노드가 서로 같다면 사이클(Cycle)이 발생한 것입니다.
 2. 그래프에 포함되어 있는 모든 간선에 대하여 1번 과정을 반복합니다.

서로소 집합을 활용한 사이클 판별: 동작 과정 살펴보기

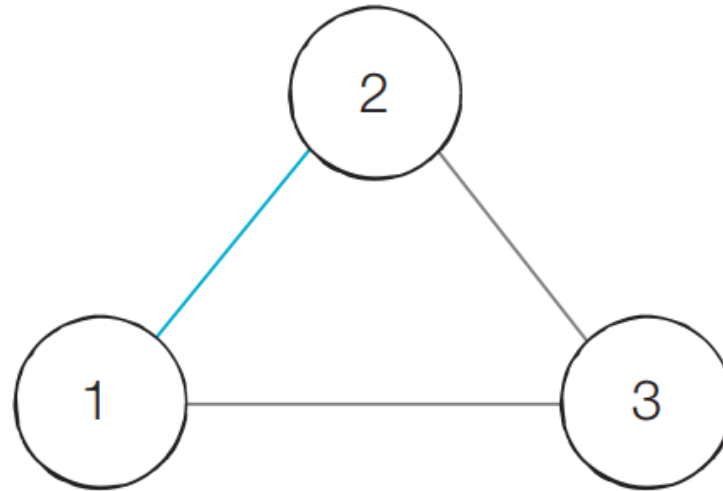
- **[초기 단계]** 모든 노드에 대하여 자기 자신을 부모로 설정하는 형태로 부모 테이블을 초기화합니다.



인덱스	1	2	3
부모	1	2	3

서로소 집합을 활용한 사이클 판별: 동작 과정 살펴보기

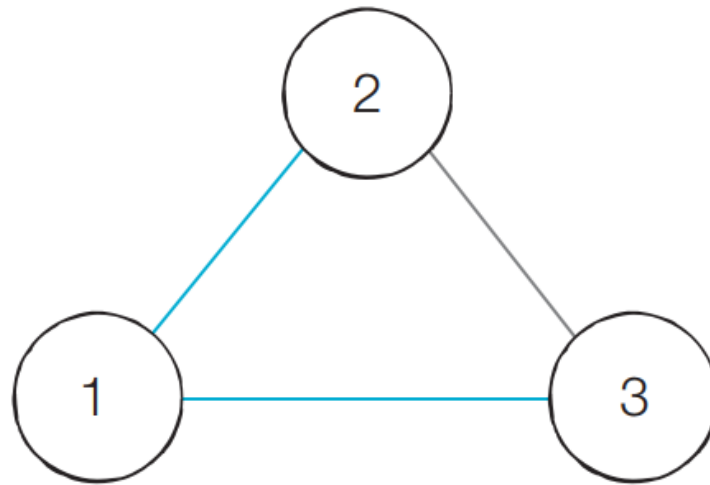
- **[Step 1]** 간선 (1, 2)를 확인합니다. 노드 1과 노드 2의 루트 노드는 각각 1과 2입니다. 따라서 더 큰 번호에 해당하는 노드 2의 부모 노드를 1로 변경합니다.



인덱스	1	2	3
부모	1	1	3

서로소 집합을 활용한 사이클 판별: 동작 과정 살펴보기

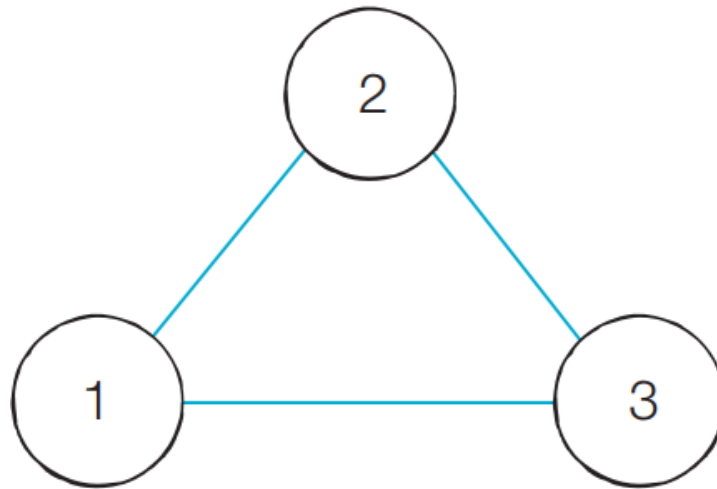
- **[Step 2]** 간선 (1, 3)을 확인합니다. 노드 1과 노드 3의 루트 노드는 각각 1과 3입니다. 따라서 더 큰 번호에 해당하는 노드 3의 부모 노드를 1로 변경합니다.



인덱스	1	2	3
부모	1	1	1

서로소 집합을 활용한 사이클 판별: 동작 과정 살펴보기

- [Step 3] 간선 (2, 3)을 확인합니다. 이미 노드 2과 노드 3의 루트 노드는 모두 1입니다. 다시 말해 **사이클이 발생**한다는 것을 알 수 있습니다.



인덱스	1	2	3
부모	1	1	1

서로소 집합을 활용한 사이클 판별

```
# 특정 원소가 속한 집합을 찾기
def find_parent(parent, x):
    # 루트 노드를 찾을 때까지 재귀 호출
    if parent[x] != x:
        parent[x] = find_parent(parent, parent[x])
    return parent[x]

# 두 원소가 속한 집합을 합치기
def union_parent(parent, a, b):
    a = find_parent(parent, a)
    b = find_parent(parent, b)
    if a < b:
        parent[b] = a
    else:
        parent[a] = b
```

```
# 노드의 개수와 간선(Union 연산)의 개수 입력 받기
v, e = map(int, input().split())
parent = [0] * (v + 1) # 부모 테이블 초기화하기

# 부모 테이블상에서, 부모를 자기 자신으로 초기화
for i in range(1, v + 1):
    parent[i] = i

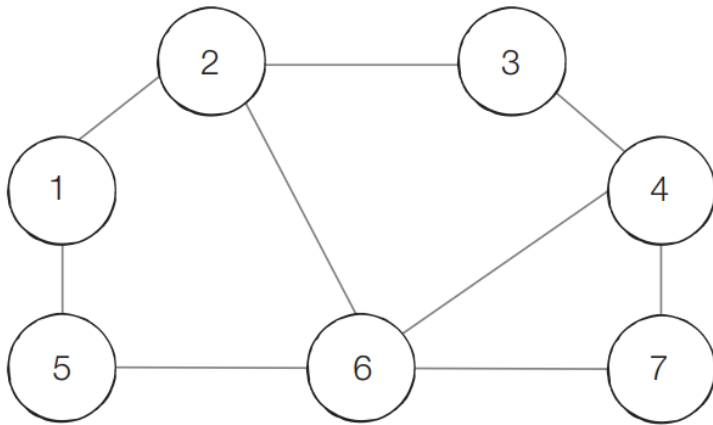
cycle = False # 사이클 발생 여부

for i in range(e):
    a, b = map(int, input().split())
    # 사이클이 발생한 경우 종료
    if find_parent(parent, a) == find_parent(parent, b):
        cycle = True
        break
    # 사이클이 발생하지 않았다면 합집합(Union) 연산 수행
    else:
        union_parent(parent, a, b)

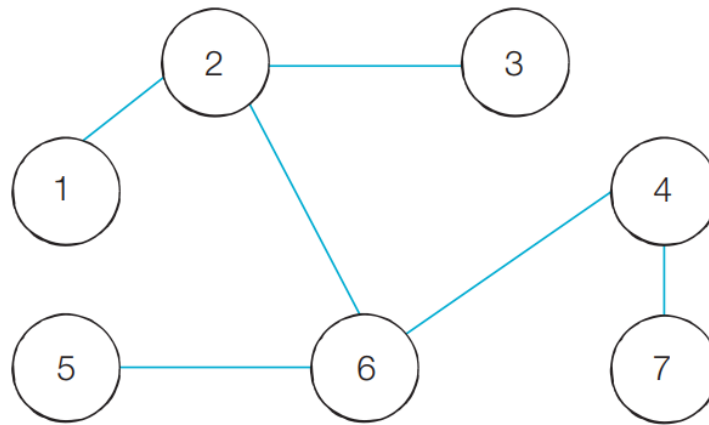
if cycle:
    print("사이클이 발생했습니다.")
else:
    print("사이클이 발생하지 않았습니다.")
```

신장 트리

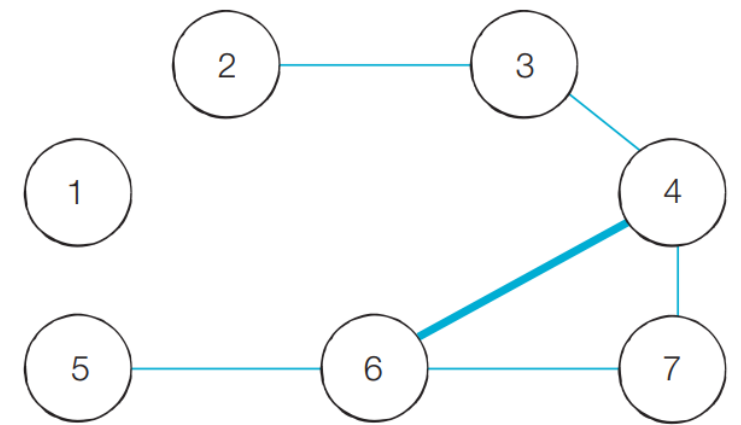
- 그래프에서 모든 노드를 포함하면서 사이클이 존재하지 않는 부분 그래프를 의미합니다.
 - 모든 노드가 포함되어 서로 연결되면서 사이클이 존재하지 않는다는 조건은 **트리의 조건**이기도 합니다.



원본 그래프



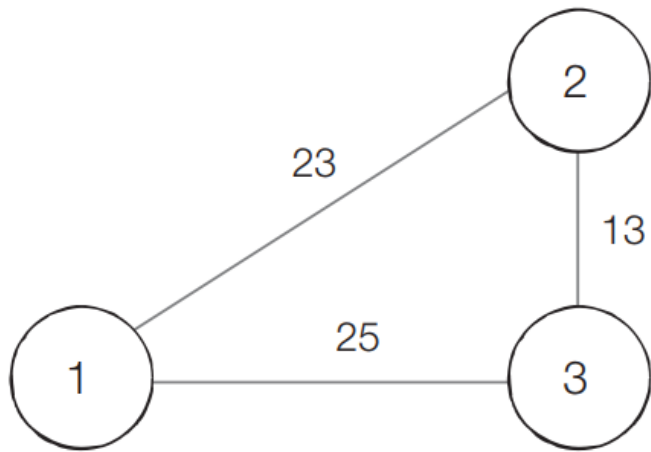
가능한 신장 트리 예시



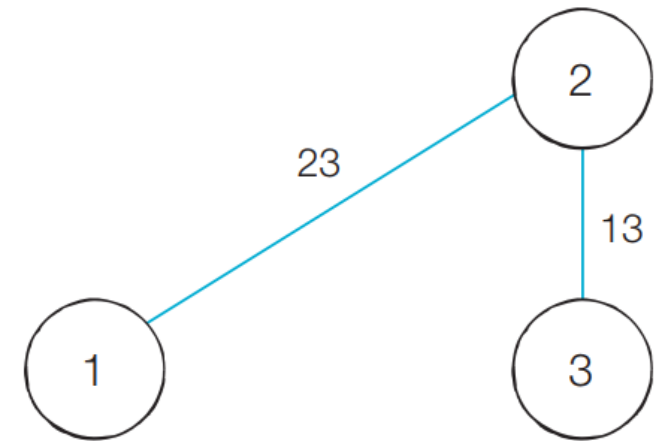
신장 트리가 아닌 부분 그래프 예시

최소 신장 트리

- 최소한의 비용으로 구성되는 신장 트리를 찾아야 할 때 어떻게 해야 할까요?
- 예를 들어 N개의 도시가 존재하는 상황에서 두 도시 사이에 도로를 놓아 **전체 도시가 서로 연결될 수 있게** 도로를 설치하는 경우를 생각해 봅시다.
 - 두 도시 A,B를 선택했을 때 A에서 B로 이동하는 경로가 반드시 존재하도록 도로를 설치합니다.



최소 신장 트리 찾기

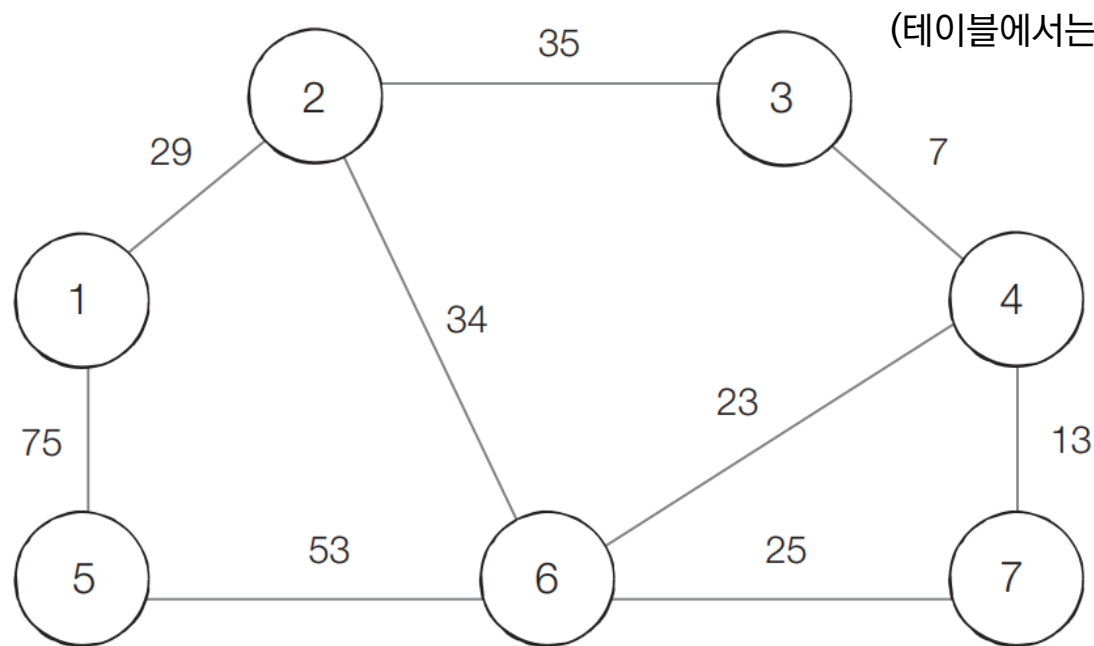


크루스칼 알고리즘

- 대표적인 최소 신장 트리 알고리즘입니다.
- 그리디 알고리즘으로 분류됩니다.
- 구체적인 동작 과정은 다음과 같습니다.
 1. 간선 데이터를 비용에 따라 오름차순으로 정렬합니다.
 2. 간선을 하나씩 확인하며 현재의 간선이 사이클을 발생시키는지 확인합니다.
 - 1) 사이클이 발생하지 않는 경우 최소 신장 트리에 포함시킵니다.
 - 2) 사이클이 발생하는 경우 최소 신장 트리에 포함시키지 않습니다.
 3. 모든 간선에 대하여 2번의 과정을 반복합니다.

크루스칼 알고리즘: 동작 과정 살펴보기

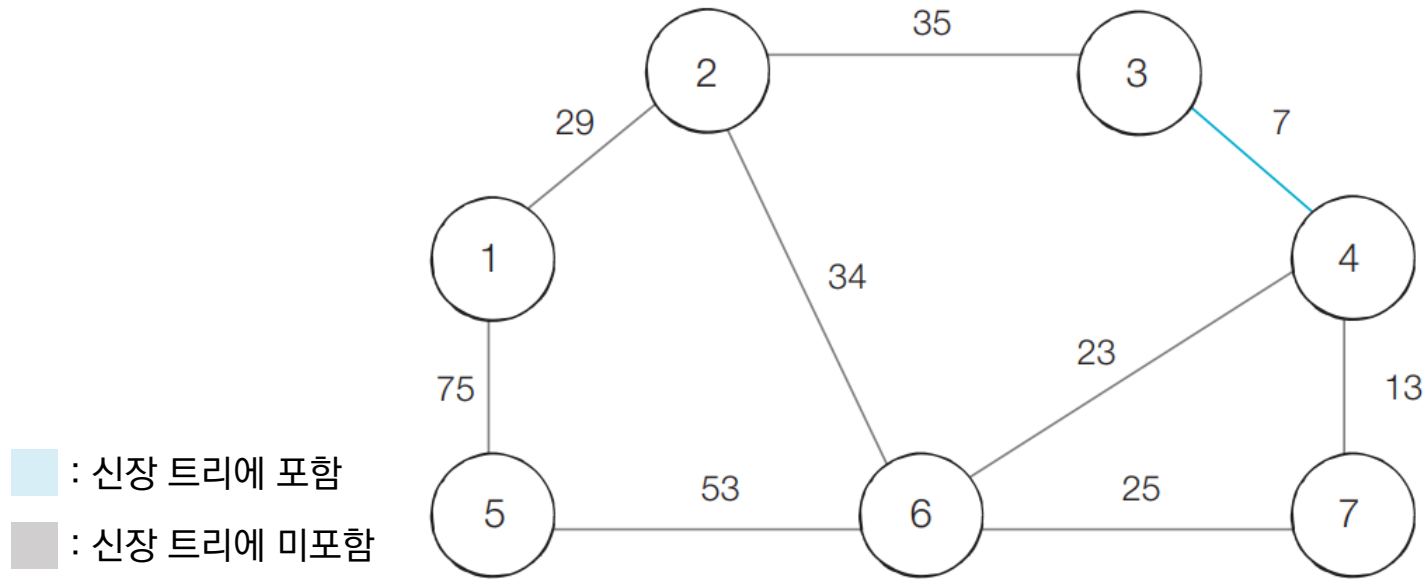
- [초기 단계] 그래프의 모든 간선 정보에 대하여 오름차순 정렬을 수행합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25

크루스칼 알고리즘: 동작 과정 살펴보기

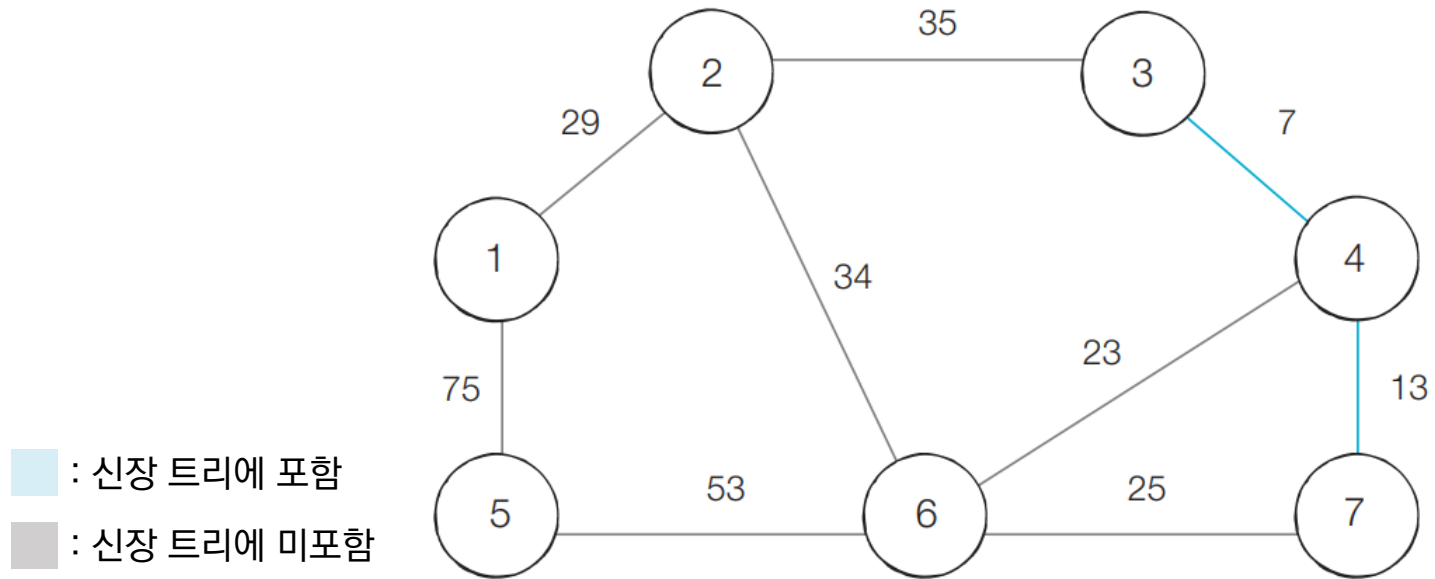
- [Step 1] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (3, 4)를 선택하여 처리합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1				

크루스칼 알고리즘: 동작 과정 살펴보기

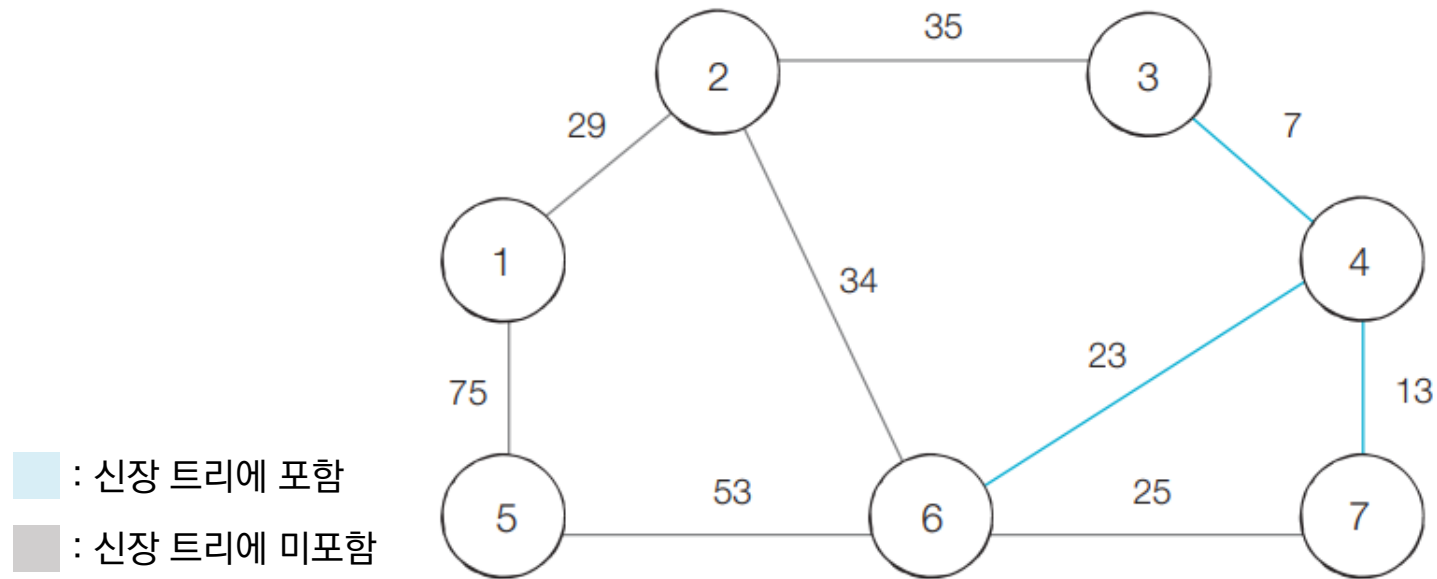
- [Step 2] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (4, 7)을 선택하여 처리합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1		step 2		

크루스칼 알고리즘: 동작 과정 살펴보기

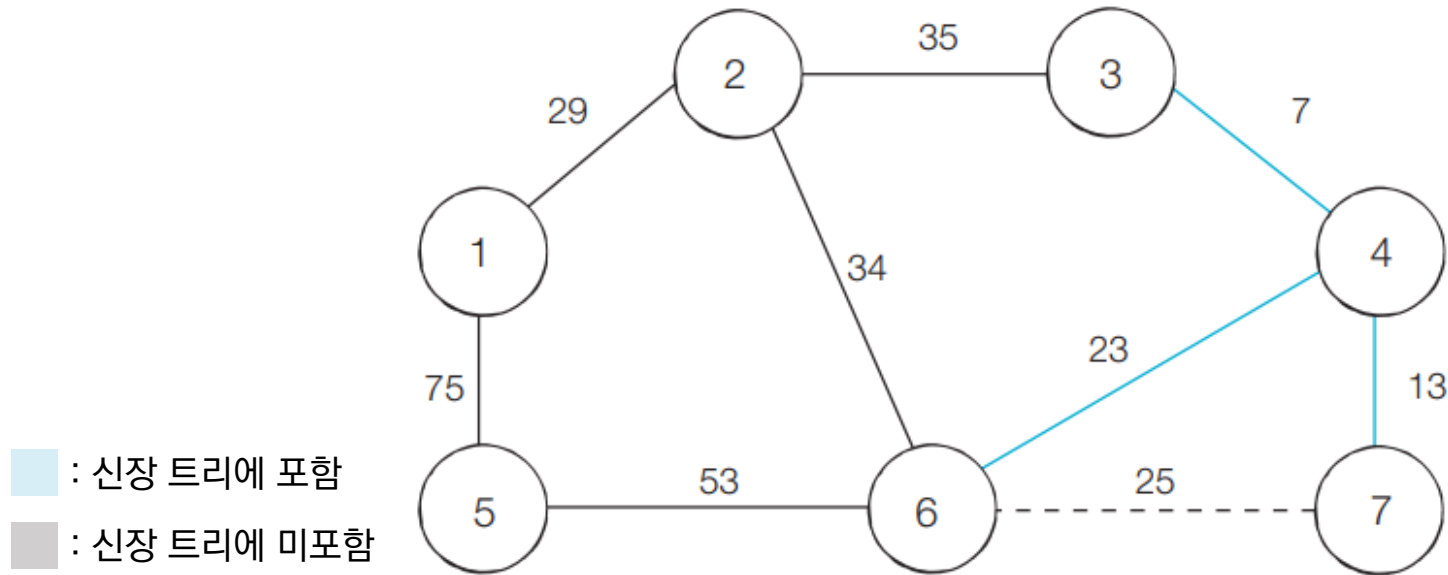
- [Step 3] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (4, 6)을 선택하여 처리합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1	step 3	step 2		

크루스칼 알고리즘: 동작 과정 살펴보기

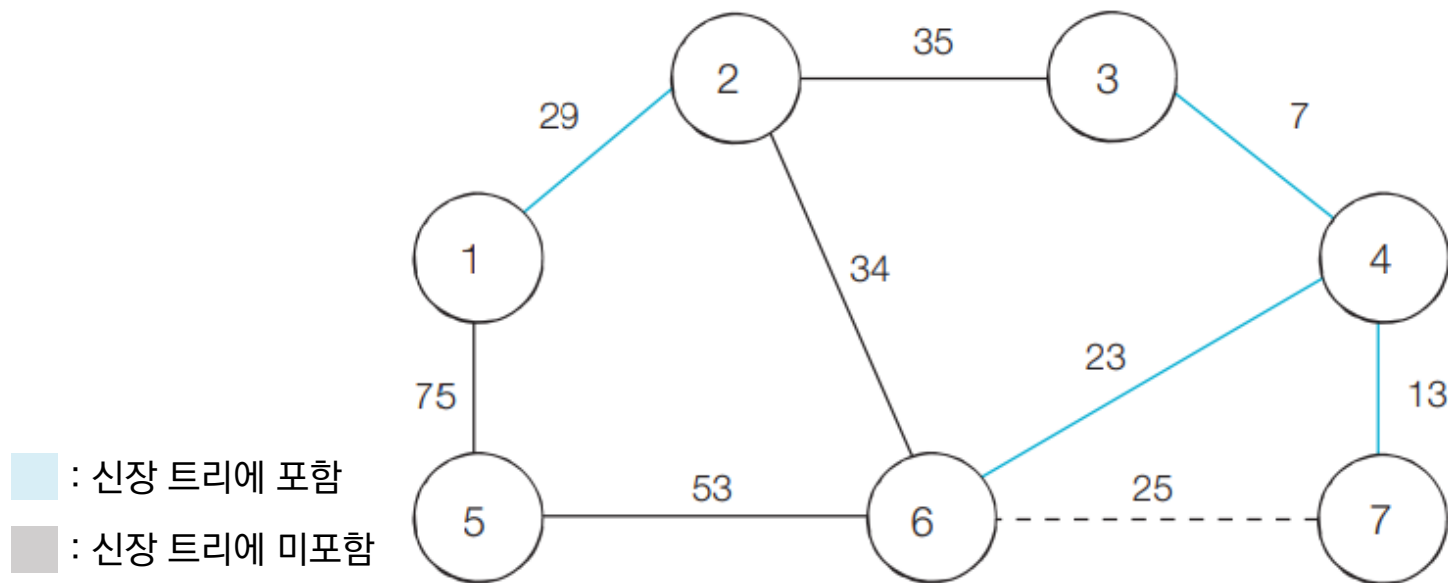
- [Step 4] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (6, 7)을 선택하여 처리합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1	step 3	step 2		step 4

크루스칼 알고리즘: 동작 과정 살펴보기

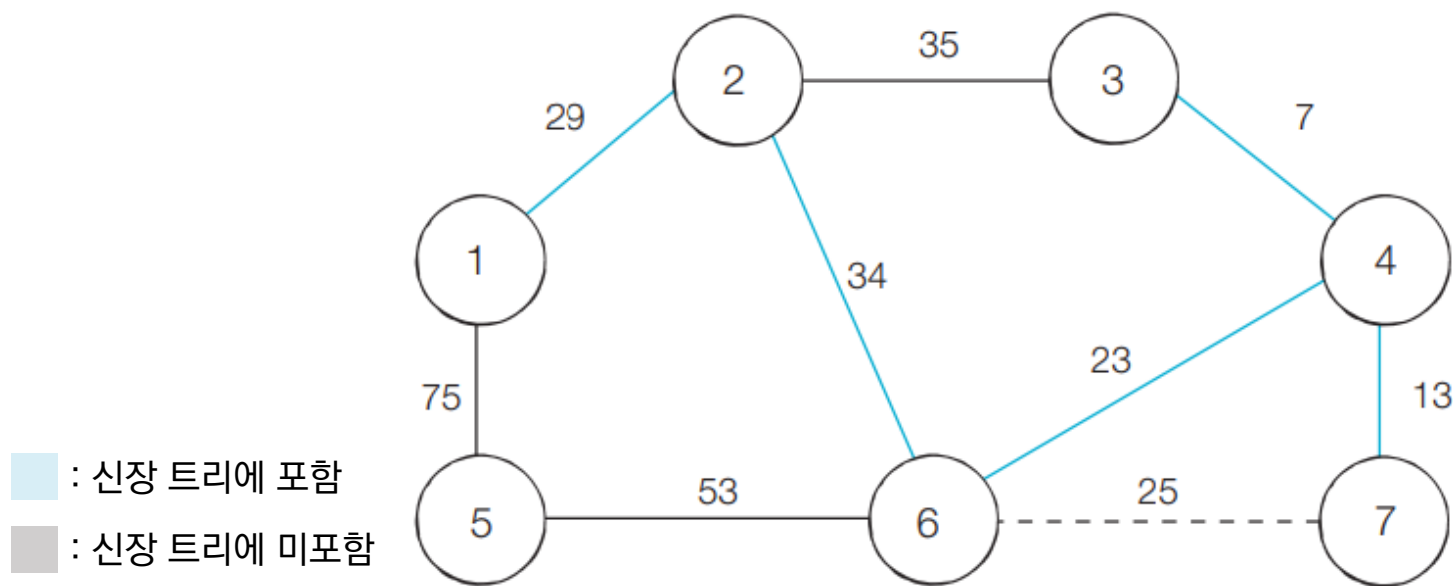
- [Step 5] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (1, 2)를 선택하여 처리합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서	step 5				step 1	step 3	step 2		step 4

크루스칼 알고리즘: 동작 과정 살펴보기

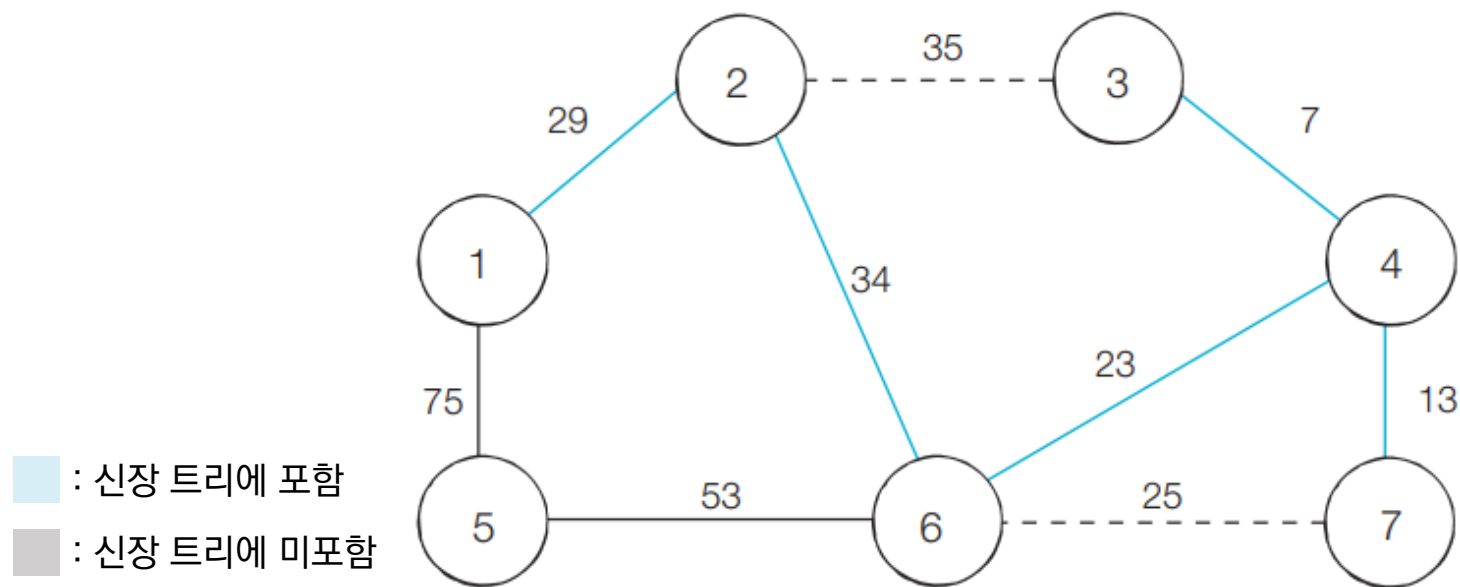
- [Step 6] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (2, 6)을 선택하여 처리합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서	step 5			step 6	step 1	step 3	step 2		step 4

크루스칼 알고리즘: 동작 과정 살펴보기

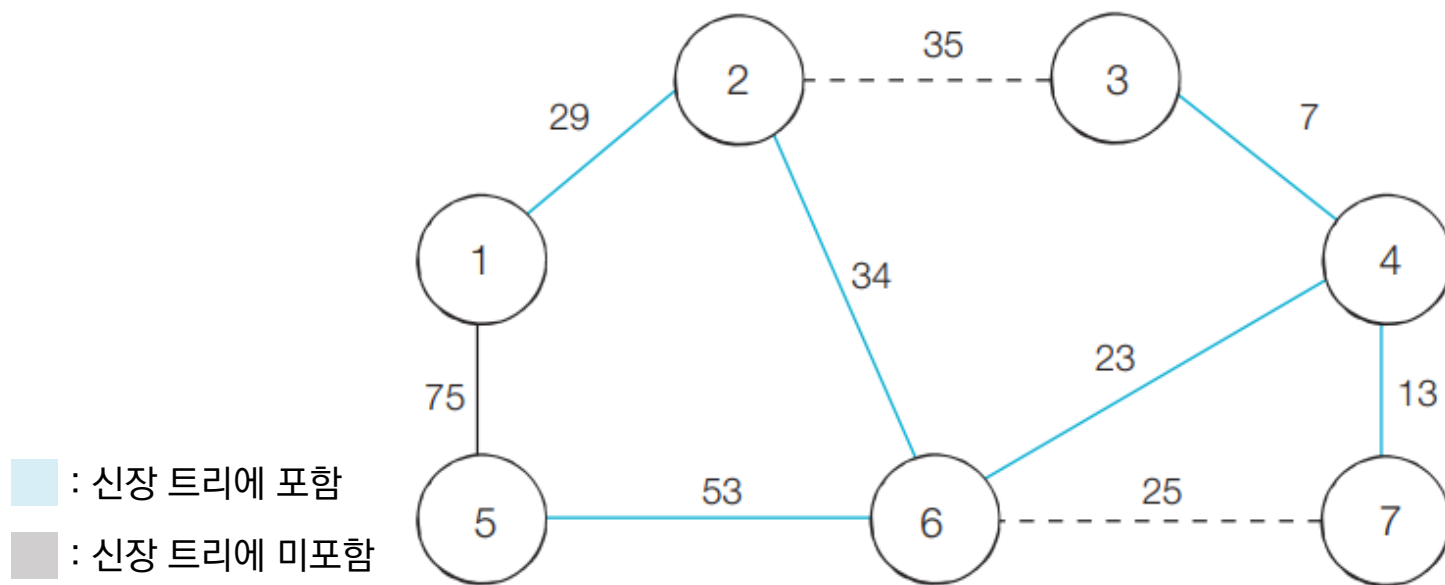
- [Step 7] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (2, 3)을 선택하여 처리합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서	step 5		step 7	step 6	step 1	step 3	step 2		step 4

크루스칼 알고리즘: 동작 과정 살펴보기

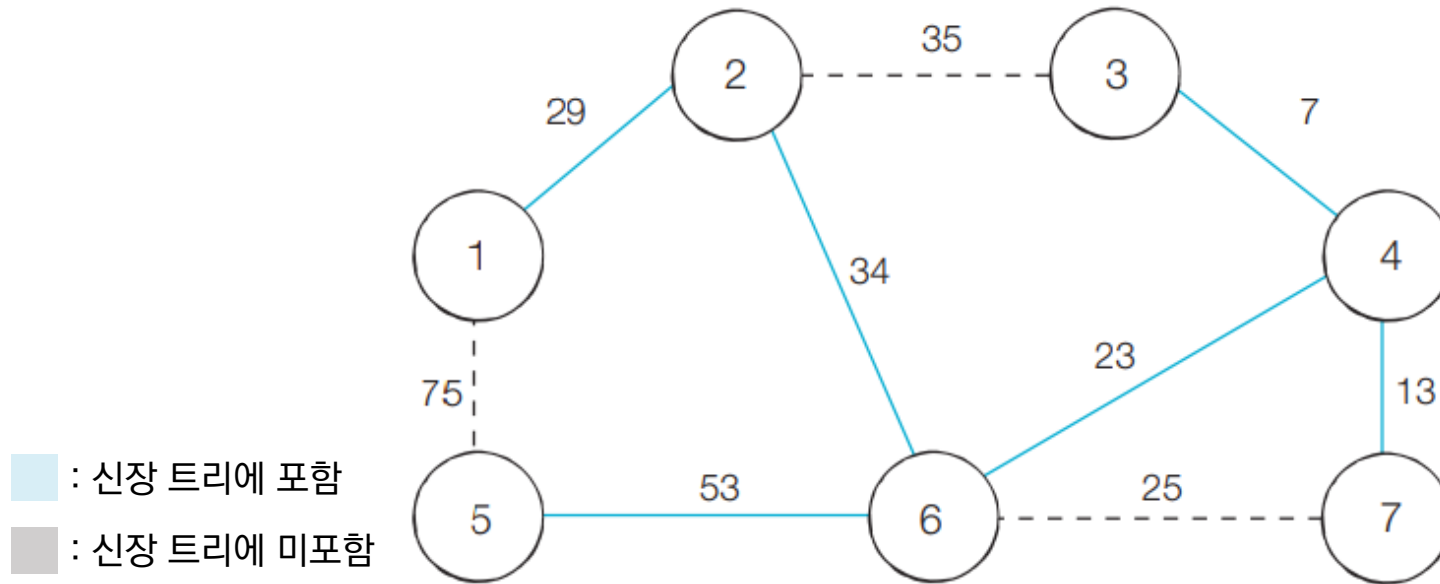
- [Step 8] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (5, 6)을 선택하여 처리합니다.



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서	step 5		step 7	step 6	step 1	step 3	step 2	step 8	step 4

크루스칼 알고리즘: 동작 과정 살펴보기

- [Step 9] 아직 처리하지 않은 간선 중에서 가장 짧은 간선인 (1, 5)를 선택하여 처리합니다.

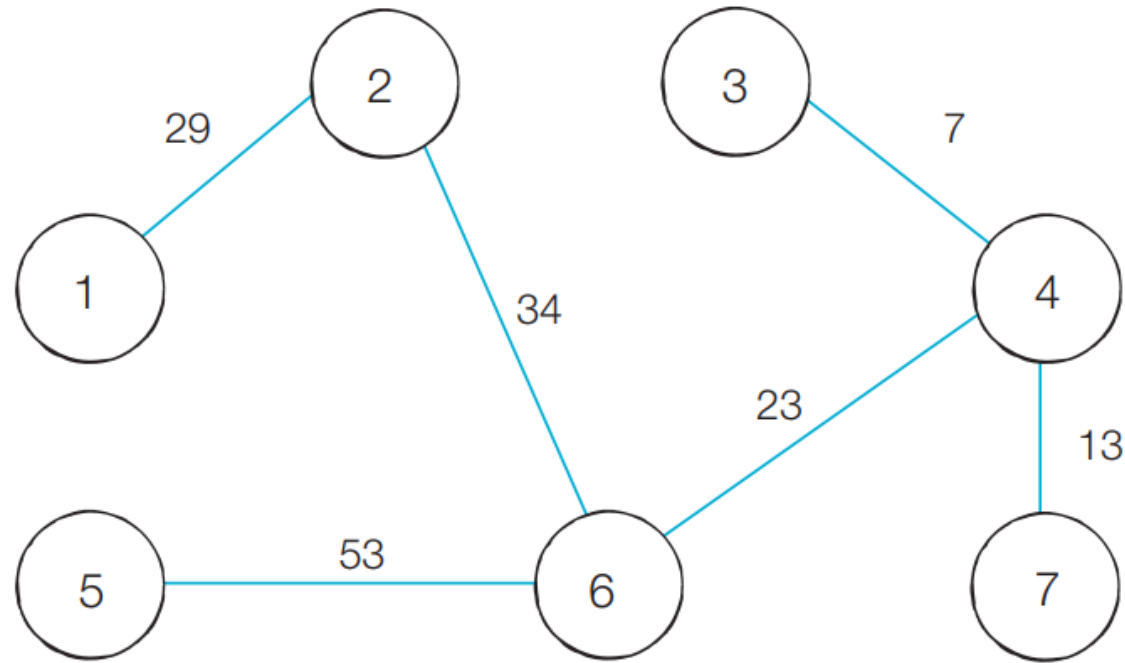


간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서	step 5	step 9	step 7	step 6	step 1	step 3	step 2	step 8	step 4

크루스칼 알고리즘: 동작 과정 살펴보기

- [알고리즘 수행 결과]

- 최소 신장 트리에 포함되어 있는 간선의 비용만 모두 더하면, 그 값이 최종 비용에 해당합니다.



크루스칼 알고리즘

```
# 특정 원소가 속한 집합을 찾기
def find_parent(parent, x):
    # 루트 노드를 찾을 때까지 재귀 호출
    if parent[x] != x:
        parent[x] = find_parent(parent, parent[x])
    return parent[x]

# 두 원소가 속한 집합을 합치기
def union_parent(parent, a, b):
    a = find_parent(parent, a)
    b = find_parent(parent, b)
    if a < b:
        parent[b] = a
    else:
        parent[a] = b

# 노드의 개수와 간선(Union 연산)의 개수 입력 받기
v, e = map(int, input().split())
parent = [0] * (v + 1) # 부모 테이블 초기화하기

# 모든 간선을 담을 리스트와, 최종 비용을 담을 변수
edges = []
result = 0
```

```
# 부모 테이블상에서, 부모를 자기 자신으로 초기화
for i in range(1, v + 1):
    parent[i] = i

# 모든 간선에 대한 정보를 입력 받기
for _ in range(e):
    a, b, cost = map(int, input().split())
    # 비용순으로 정렬하기 위해서 튜플의 첫 번째 원소를 비용으로 설정
    edges.append((cost, a, b))

# 간선을 비용순으로 정렬
edges.sort()

# 간선을 하나씩 확인하며
for edge in edges:
    cost, a, b = edge
    # 사이클이 발생하지 않는 경우에만 집합에 포함
    if find_parent(parent, a) != find_parent(parent, b):
        union_parent(parent, a, b)
        result += cost

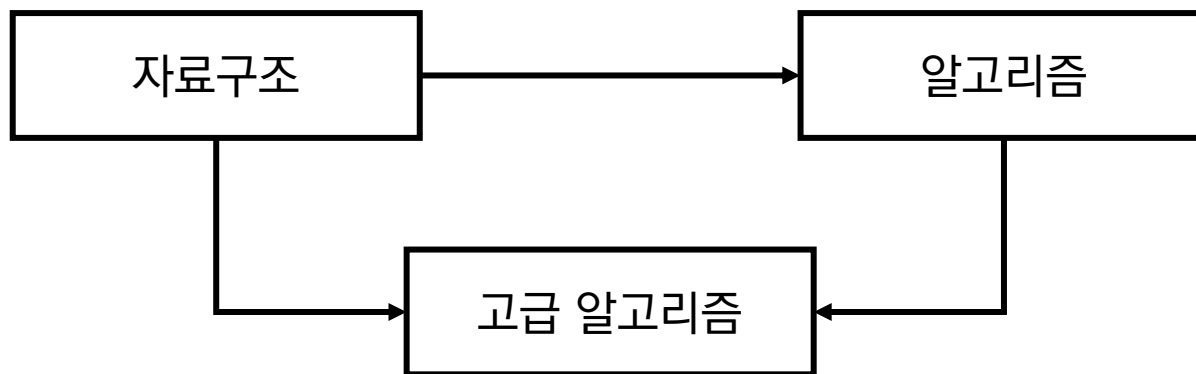
print(result)
```


크루스칼 알고리즘 성능 분석

- 크루스칼 알고리즘은 간선의 개수가 E 개일 때, $O(E \log E)$ 의 시간 복잡도를 가집니다.
- 크루스칼 알고리즘에서 가장 많은 시간을 요구하는 곳은 간선을 정렬을 수행하는 부분입니다.
 - 표준 라이브러리를 이용해 E 개의 데이터를 정렬하기 위한 시간 복잡도는 $O(E \log E)$ 입니다.

위상 정렬

- **사이클이 없는 방향 그래프**의 모든 노드를 **방향성에 거스르지 않도록** 순서대로 나열하는 것을 의미합니다.
- 예시) 선수과목을 고려한 학습 순서 설정



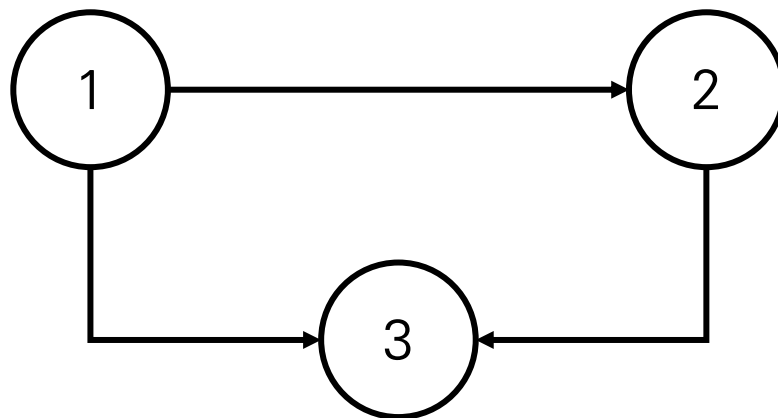
- 위 세 과목을 모두 듣기 위한 **적절한 학습 순서**는?
 - 자료구조 → 알고리즘 → 고급 알고리즘 (O)
 - 자료구조 → 고급 알고리즘 → 알고리즘 (X)

진입차수와 진출차수

- 진입차수(Indegree): 특정한 노드로 들어오는 간선의 개수
- 진출차수(Outdegree): 특정한 노드에서 나가는 간선의 개수

진입차수 = 0

진출차수 = 2




진입차수 = 1

진출차수 = 1

진입차수 = 2

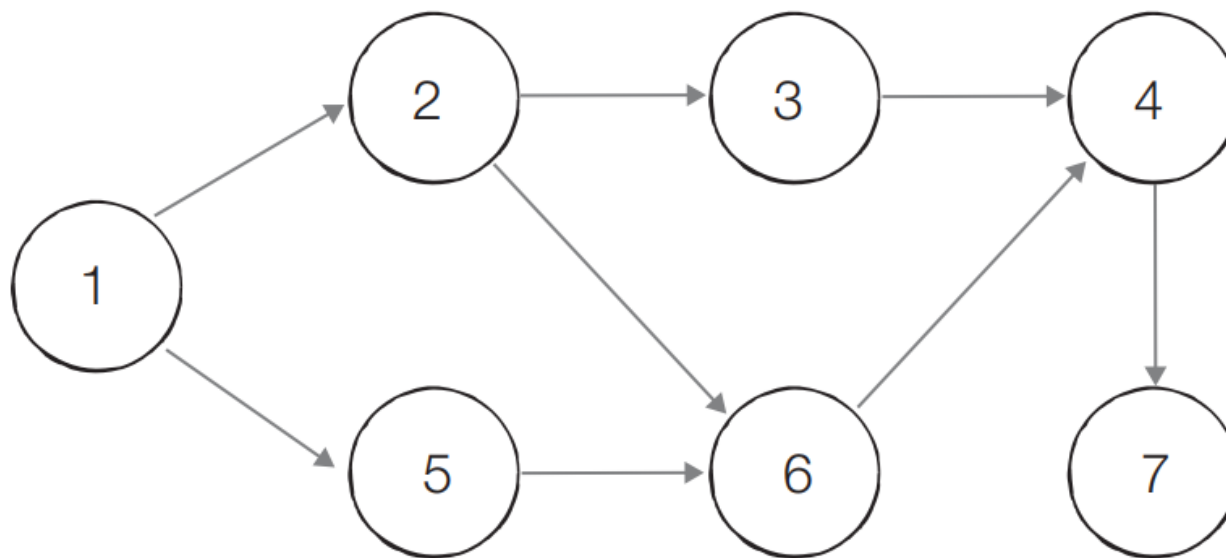
진출차수 = 0

위상 정렬 알고리즘

- 큐를 이용하는 위상 정렬 알고리즘의 동작 과정은 다음과 같습니다.
 1. 진입차수가 0인 모든 노드를 큐에 넣는다.
 2. 큐가 빌 때까지 다음의 과정을 반복한다.
 - 1) 큐에서 원소를 꺼내 해당 노드에서 나가는 간선을 그래프에서 제거한다.
 - 2) 새롭게 진입차수가 0이 된 노드를 큐에 넣는다.
-  결과적으로 각 노드가 큐에 들어온 순서가 위상 정렬을 수행한 결과와 같습니다.

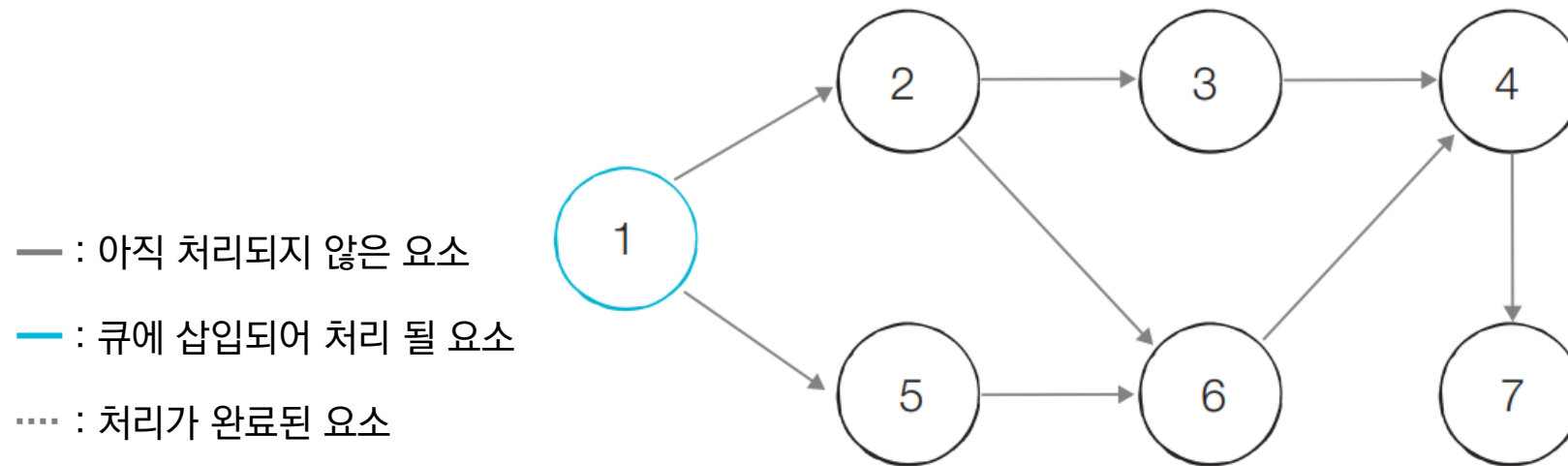
위상 정렬 동작 예시

- 위상 정렬을 수행할 그래프를 준비합니다.
 - 이때 그래프는 사이클이 없는 방향 그래프 (DAG)여야 합니다.



위상 정렬 동작 예시

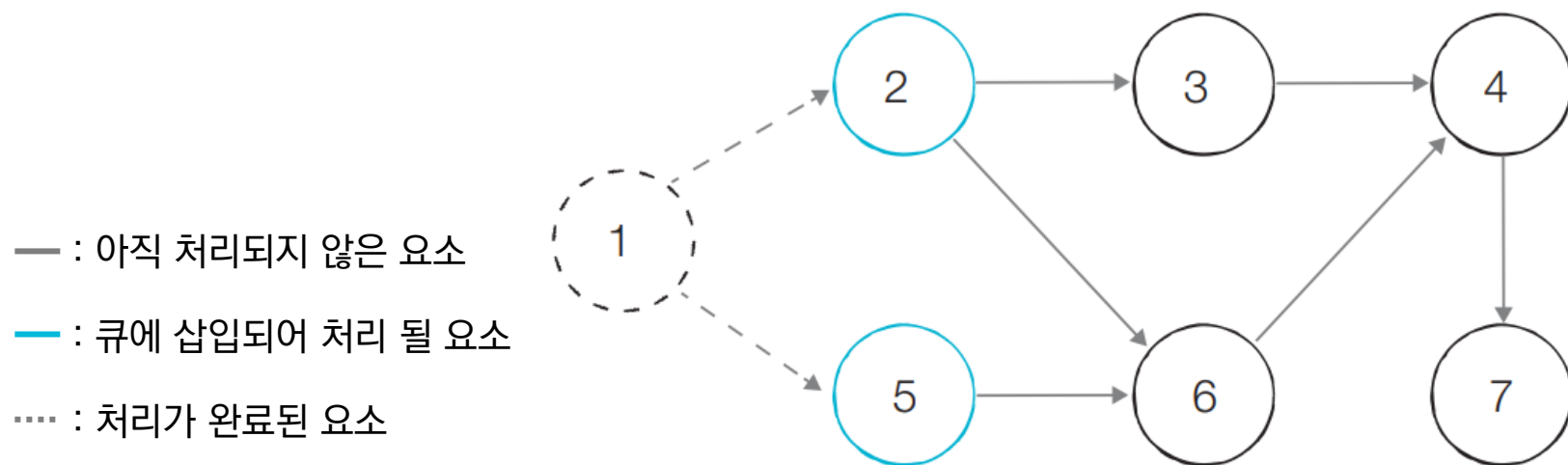
- [초기 단계] 초기 단계에서는 진입차수가 0인 모든 노드를 큐에 넣습니다.
 - 처음에 노드 1이 큐에 삽입됩니다.



노드	1	2	3	4	5	6	7
진입차수	0	1	1	2	1	2	1
큐	노드 1						

위상 정렬 동작 예시

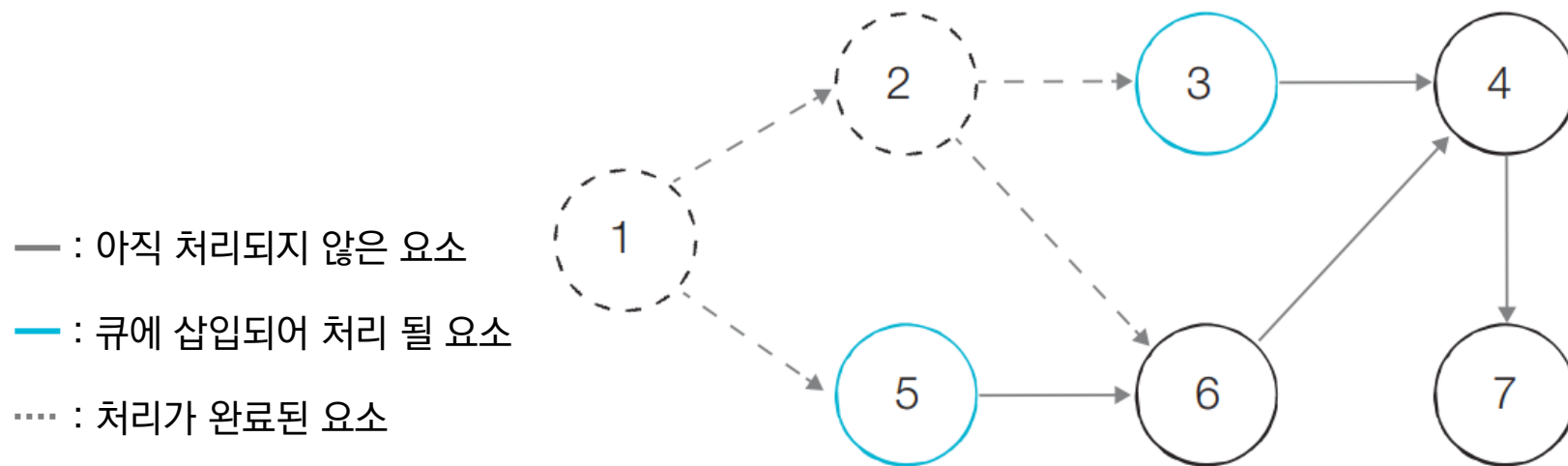
- [Step 1] 큐에서 노드 1을 꺼낸 뒤에 노드 1에서 나가는 간선을 제거합니다.
 - 새롭게 진입차수가 0이 된 노드들을 큐에 삽입합니다.



노드	1	2	3	4	5	6	7
진입차수	0	0	1	2	0	2	1
큐	노드 2, 노드 5						

위상 정렬 동작 예시

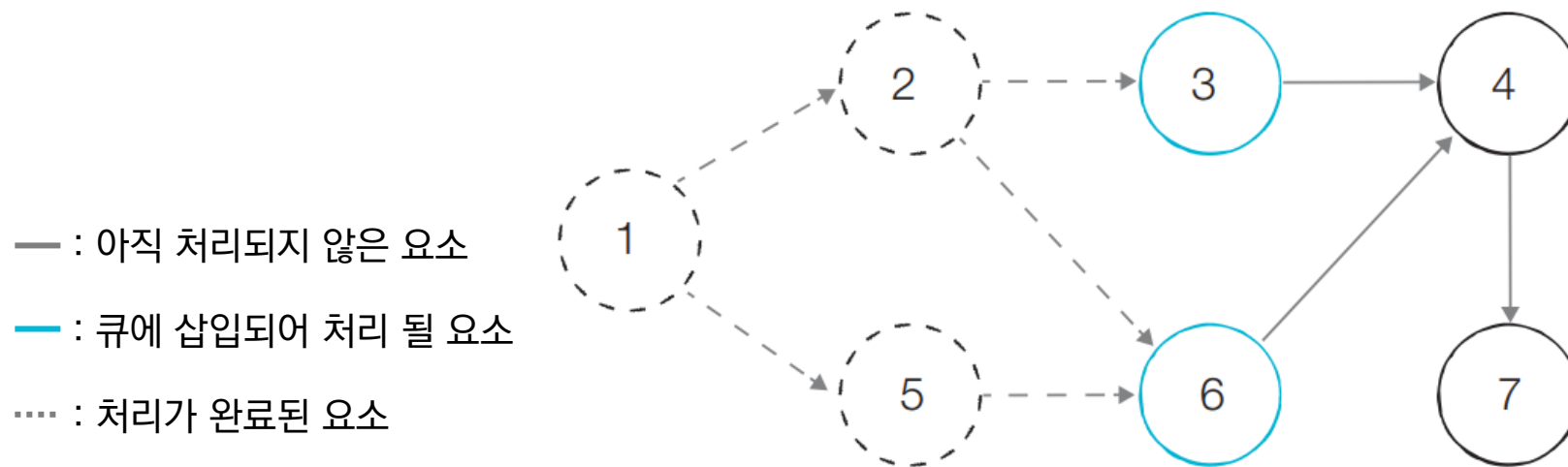
- [Step 2] 큐에서 노드 2를 꺼낸 뒤에 노드 2에서 나가는 간선을 제거합니다.
 - 새롭게 진입차수가 0이 된 노드를 큐에 삽입합니다.



노드	1	2	3	4	5	6	7
진입차수	0	0	0	2	0	1	1
큐	노드 5, 노드 3						

위상 정렬 동작 예시

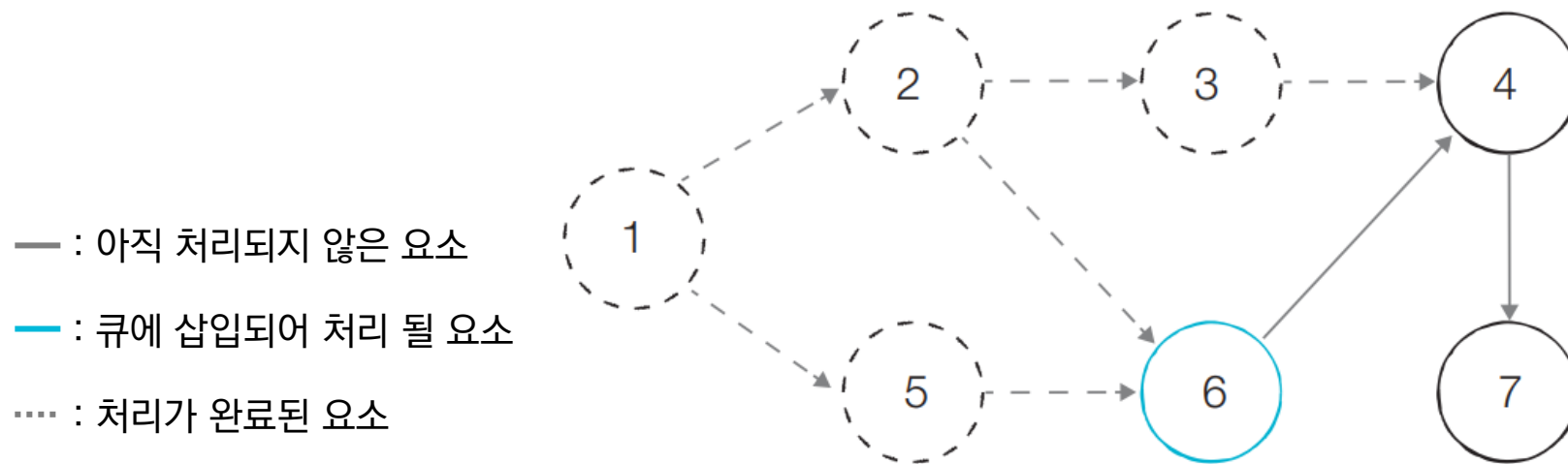
- [Step 3] 큐에서 노드 5를 꺼낸 뒤에 노드 5에서 나가는 간선을 제거합니다.
 - 새롭게 진입차수가 0이 된 노드를 큐에 삽입합니다.



노드	1	2	3	4	5	6	7
진입차수	0	0	0	2	0	0	1
큐	노드 3, 노드 6						

위상 정렬 동작 예시

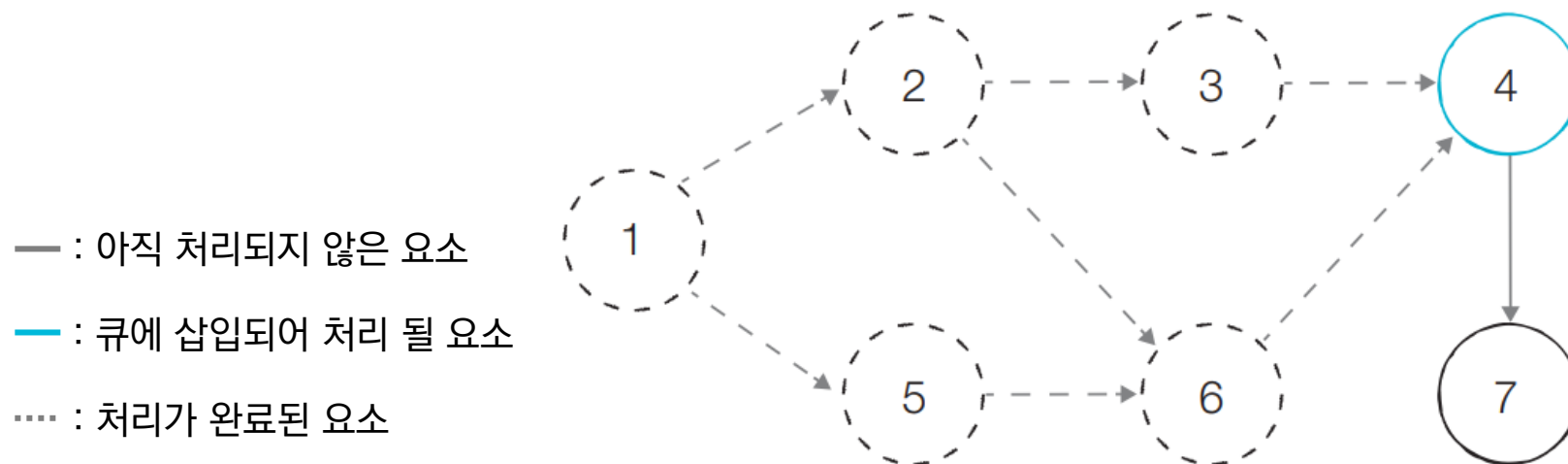
- [Step 4] 큐에서 노드 3를 꺼낸 뒤에 노드 3에서 나가는 간선을 제거합니다.
 - 새롭게 진입차수가 0이 된 노드가 없으므로 그냥 넘어갑니다.



노드	1	2	3	4	5	6	7
진입차수	0	0	0	1	0	0	1
큐	노드 6						

위상 정렬 동작 예시

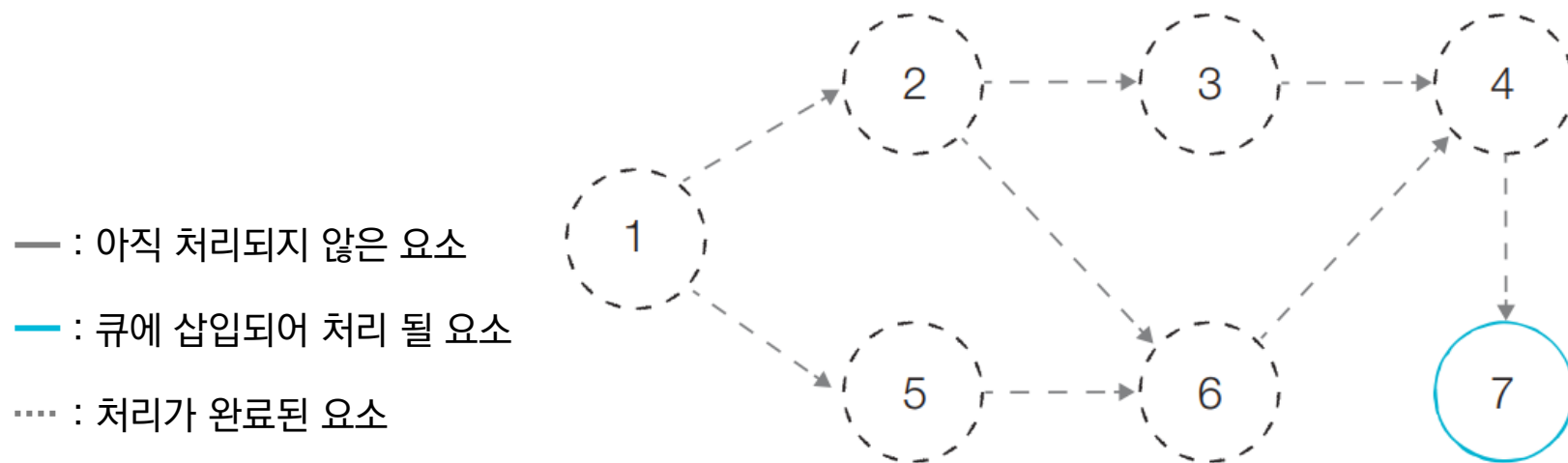
- [Step 5] 큐에서 노드 6를 꺼낸 뒤에 노드 6에서 나가는 간선을 제거합니다.
 - 새롭게 진입차수가 0이 된 노드를 큐에 삽입합니다.



노드	1	2	3	4	5	6	7
진입차수	0	0	0	0	0	0	1
큐	노드 4						

위상 정렬 동작 예시

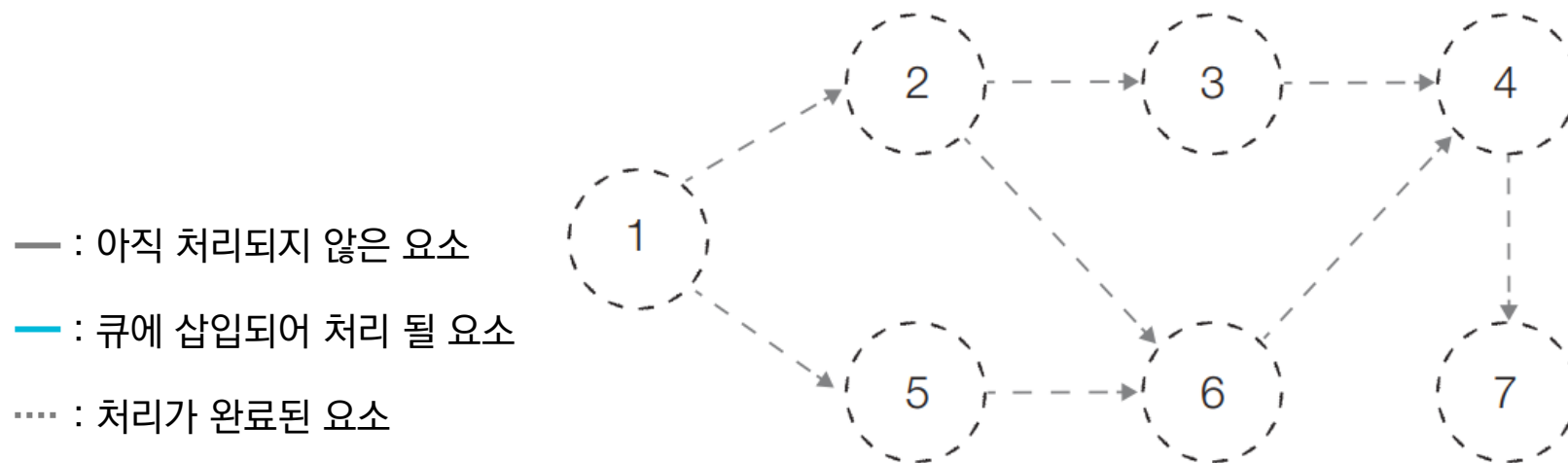
- [Step 6] 큐에서 노드 4를 꺼낸 뒤에 노드 4에서 나가는 간선을 제거합니다.
 - 새롭게 진입차수가 0이 된 노드를 큐에 삽입합니다.



노드	1	2	3	4	5	6	7
진입차수	0	0	0	0	0	0	0
큐	노드 7						

위상 정렬 동작 예시

- [Step 7] 큐에서 노드 7을 꺼낸 뒤에 노드 7에서 나가는 간선을 제거합니다.
 - 새롭게 진입차수가 0이 된 노드가 없으므로 그냥 넘어갑니다.

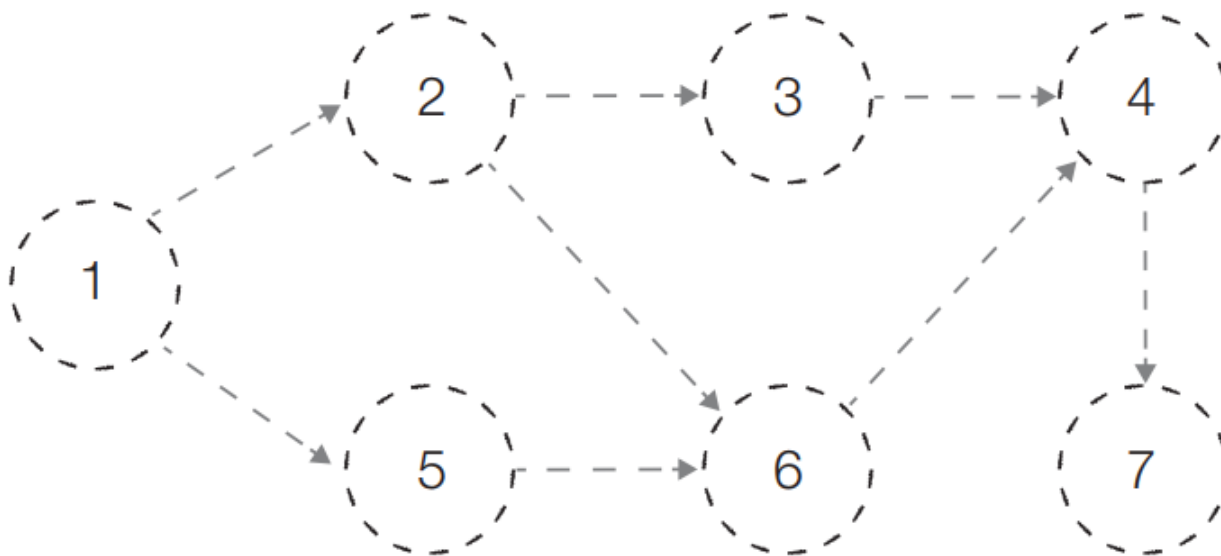


노드	1	2	3	4	5	6	7
진입차수	0	0	0	0	0	0	0
큐							

위상 정렬 동작 예시

- [위상 정렬 결과]

- 큐에 삽입된 전체 노드 순서: $1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 4 \rightarrow 7$



위상 정렬의 특징

- 위상 정렬은 DAG에 대해서만 수행할 수 있습니다.
 - DAG (Direct Acyclic Graph): 순환하지 않는 방향 그래프
- 위상 정렬에서는 여러 가지 답이 존재할 수 있습니다.
 - 한 단계에서 큐에 새롭게 들어가는 원소가 2개 이상인 경우가 있다면 여러 가지 답이 존재합니다.
- 모든 원소를 방문하기 전에 큐가 빈다면 사이클이 존재한다고 판단할 수 있습니다.
 - 사이클에 포함된 원소 중에서 어떠한 원소도 큐에 들어가지 못합니다.
- 스택을 활용한 DFS를 이용해 위상 정렬을 수행할 수도 있습니다.

위상 정렬 알고리즘

```
from collections import deque

# 노드의 개수와 간선의 개수를 입력 받기
v, e = map(int, input().split())
# 모든 노드에 대한 진입차수는 0으로 초기화
indegree = [0] * (v + 1)
# 각 노드에 연결된 간선 정보를 담기 위한 연결 리스트 초기화
graph = [[] for i in range(v + 1)]

# 방향 그래프의 모든 간선 정보를 입력 받기
for _ in range(e):
    a, b = map(int, input().split())
    graph[a].append(b) # 정점 A에서 B로 이동 가능
    # 진입 차수를 1 증가
    indegree[b] += 1
```

입력 예시

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

출력 예시

```
1 2 5 3 6 4 7
```

```
# 위상 정렬 함수
def topology_sort():
    result = [] # 알고리즘 수행 결과를 담을 리스트
    q = deque() # 큐 기능을 위한 deque 라이브러리 사용
    # 처음 시작할 때는 진입차수가 0인 노드를 큐에 삽입
    for i in range(1, v + 1):
        if indegree[i] == 0:
            q.append(i)
    # 큐가 빌 때까지 반복
    while q:
        # 큐에서 원소 꺼내기
        now = q.popleft()
        result.append(now)
        # 해당 원소와 연결된 노드들의 진입차수에서 1 빼기
        for i in graph[now]:
            indegree[i] -= 1
            # 새롭게 진입차수가 0이 되는 노드를 큐에 삽입
            if indegree[i] == 0:
                q.append(i)
    # 위상 정렬을 수행한 결과 출력
    for i in result:
        print(i, end=' ')
```

```
topology_sort()
```


위상 정렬 알고리즘 성능 분석

- 위상 정렬을 위해 차례대로 모든 노드를 확인하며 각 노드에서 나가는 간선을 차례대로 제거해야 합니다.
 - 위상 정렬 알고리즘의 시간 복잡도는 $O(V + E)$ 입니다.