그래프- 2

-탐욕적 알고리즘과 최소 신장 트리-

HaRim Jung, Ph.D.
Visiting Professor / Senior Researcher
SKKU Institute for Convergence / Convergence Research Institute
Sungkyunkwan University, Korea

탐욕적 알고리즘 개요 (1/3)

❑ 탐욕적(greedy) 알고리즘이란?

- 일련의 연속적인 부분 해(local solution or current solution) 선택이 필요한(혹은 가능한) 문제가 주어졌을 때, 선택을 해야 할 순간마다 **그 순간에 최적이라고 생각되는 것을 부분 해로 선택**함으로써 최종 해(global solution or final solution)를 도출하는 알고리즘
- 선택을 해야 하는 매 순간 최적의 부분 해를 선택하지만 최종 해가 최적이라는 보장(정확하다는 보장)이 없음
  - 따라서 탐욕적 알고리즘은 최종 해가 항상 최적인지 이론적으로 검증해야 하지만 매우 어려운 경우가 많음
  - 이론적인 검증이 불가능한 알고리즘 혹은 근사 해(approximate solution), i.e., 거의 최적에 가까운 최종 해를 도출하는 알고리즘을 **휴리스틱(heuristics)**이라고 함

❑ (일반적인) 탐욕적 알고리즘 설계 과정

- **선정 과정(selection procedure)**: 현재 가장 최적이라 생각되는 부분 해를 선택
- **적정성 점검(feasibility check)**: 선택한 부분 해를 해 집합(solution set)에 포함시키는 것이 적절한지를 확인하고 적절하다면 해 집합에 포함시킴
- **해답 점검(solution check)**: 새로 얻은 해 집합이 주어진 문제의 최종 해인지 확인

❑ 탐욕적 알고리즘의 적용 예: 거스름돈 계산하기 문제(coin change problem)

- **문제**: 액면가가 {1, 5, 10, 50, 100, 500}인 동전을 가지고 거스름돈 x를 만들기 위한 최소 동전 개수는 몇 개인가? 단, 각 액면가에 해당하는 동전의 개수는 무한하다고 가정

탐욕적 알고리즘 개요 (2/3)

❑ **탐욕적 알고리즘의 적용 예**: **거스름돈 계산하기 문제**(coin change problem) contd.

- **설계 과정**
  - **선정 과정**: 현재 선택할 수 있는 동전 중 **액면가가 가장 높은 동전**을 부분 해로 선택
    - 부분 해 **선택 기준**(예: 액면가가 가장 높은 동전 혹은 액면가가 가장 낮은 동전)의 타당 여부를 보이기 위한 방법 중 하나는 최종 해가 항상 최적이 아님을 보이는 **반례(counterexample)**를 찾는 것
    - 탐욕적 알고리즘 설계 과정에서 **부분 해 선택 기준 선정**은 가장 중요한 고려 사항 중 하나
  - **적정성 점검**: 해당 동전을 거스름돈에 추가 시 거스름돈 총액을 초과하는지 확인하여 초과하지 않았다면 해 집합에 추가하고 초과했다면 선정 과정으로 되돌아감
  - **해답 점검**: 현재까지의 금액(해 집합의 총액)이 거스름돈 총액에 도달했는지 확인

```python
def coin_change(x):
    d = [1, 5, 10, 50, 100, 500]
    result = []
    i = len(d) - 1
    while True:
        while x >= d[i]:
            x = x - d[i]
            result.append(d[i])
        i -= 1
        if i < 0:
            break
    for i in range(len(result)):
        print(result[i], end=" ")

x = 16
coin_change(x)
```

탐욕적 알고리즘 개요 (3/3)

❑ 탐욕적 알고리즘의 적용 예: 거스름돈 계산하기 문제(coin change problem) contd.

- 액면가가 {1, 5, 10, 50, 100, 500}인 동전 시스템(coin system) 상 Slide 3의 탐욕적 알고리즘을 이용하여 거스름돈 x = 16을 만들면 항상 동전의 개수는 최소 → 최종 해가 항상 최적임을 보장

- 동전 시스템에 액면가가 12인 동전을 추가한다면?
  - 알고리즘의 결과는 12, 1, 1, 1, 1이므로 최종적으로 최적의 해(i.e., 10, 5, 1)를 보장하지 못함
- 탐욕적 알고리즘을 이용하여 거스름돈 계산하기 문제에 대한 최종 해를 도출 시 동전 시스템에 의해 최종 해의 최적 여부가 결정

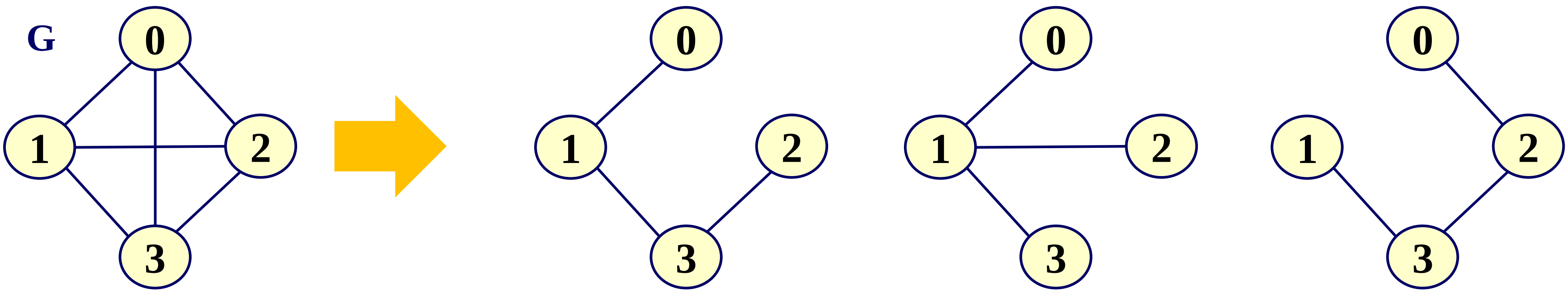
  → **항상 최적인 최종 해를 도출할 수 없음**

- 위와 같이 탐욕적 알고리즘을 이용하여 항상 최적인 최종 해를 도출해 낼 수 없는 문제가 많이 존재하지만, 시간·공간 효율성의 이유로 탐욕적 알고리즘을 이용하여 최적 해(optimal solution) 대신 근사 해(approximate solution)를 도출하는 경우도 많음
  - 주어진 문제에 대해 시간·공간 효율성이 낮은 알고리즘을 이용하여 최적 해를 도출하는 것보다 시간·공간 효율성이 높은 알고리즘을 이용하여 적정한 수준의 근사 해를 도출하는 것이 더 유용할 수 있음

최소 신장 트리 (1/2)

- **신장 트리**(Spanning Tree)
  - 주어진 그래프 G가 하나의 연결성분으로 구성되어 있을 때, G의 모든 정점들을 포함하되 트리가 되는(i.e., 싸이클이 없는) 부분 연결 그래프
    - 하나의 연결성분으로 구성된 그래프 G = (V, E)가 주어졌을 때, 사이클이 존재하지 않고 V’ = V이고, E’ ⊆ E인 그래프 G’ = (V’, E’) ⟶ **정점이 N 개인 그래프의 신장 트리는 N - 1 개의 간선을 가짐**

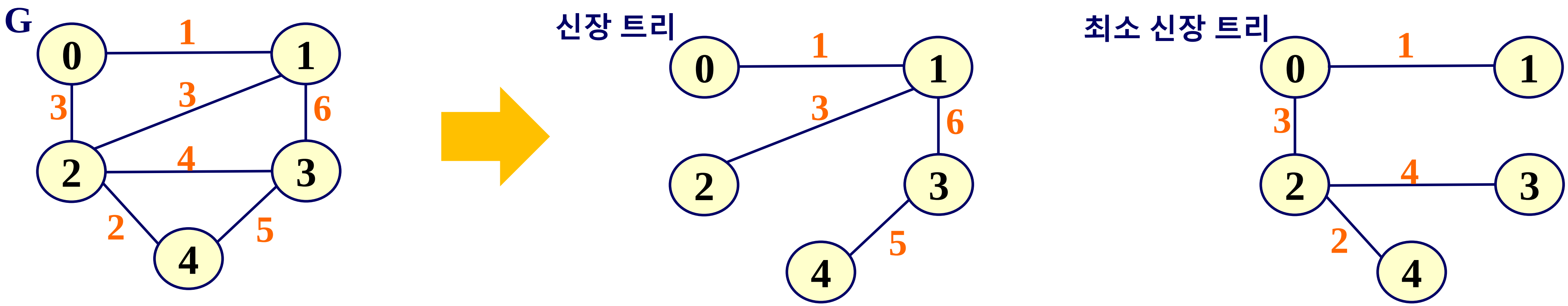
그래프 G가 주어졌을 때 신장 트리의 예:



- **최소 신장 트리**(MST: Minimum Spanning Tree)
  - 하나의 연결성분으로 이루어진 **무방향 가중치 그래프**에서 **간선들의 가중치 합이 최소**인 부분그래프(신장 트리)
  - 다음과 같은 이유로 간선들의 가중치 합이 최소인 부분그래프는 당연히 트리가 되어야 함
    - 만약 트리가 아니라면 **반드시 싸이클이 존재**함
    - 따라서 싸이클 상의 한 간선을 제거하면 더 작은 가중치 합을 가지는 (신장)트리가 생성됨
  - 모든 신장 트리가 최소 신장 트리는 아니며, 최소 신장 트리는 하나 이상 존재할 수 있음

최소 신장 트리 (2/2)

가중치 그래프 G가 주어졌을 때 신장 트리, 최소 신장 트리의 예:



- MST **찾기**
  - **문제**: **무방향 그래프** G = (V,E)**가 주어졌을 때**, E' ⊆ E**를 만족하면서**, (*V*, E')**가** *G***의** MST**가 되는** E'**을 찾는 문제**
    - MST를 찾기 위한 단순한 방법은 주어진 그래프 G의 모든 신장 트리를 고려하여 가중치가 최소인 신장 트리를 선택 → 비효율적
    - MST를 찾는 대표적인 알고리즘인 **Prim 알고리즘**과 **Kruskal 알고리즘**은 모두 **탐욕적(Greedy) 알고리즘**
    - 탐욕적 알고리즘은 최적해(최솟값 또는 최댓값)를 찾는 문제를 해결하기 위한 알고리즘 방식들 중 하나로서, 알고리즘의 선택이 항상 '욕심 내어' (1) **지역적인(부분적인) 최적해**를 선택하며, (2) 이러한 부분적인 최적해 선택을 축적하여 최종 최적해를 찾음

- MST **찾기의 활용 혹은 응용**(applications)
  - **통신망:** 전화선의 길이가 최소가 되도록 전화 케이블 망을 구성하는 문제
  - **도로망**: 도시들을 모두 연결하면서 도로의 길이가 최소가 되도록 하는 문제
  - **배관 작업**: 파이프의 총 길이가 최소가 되도록 연결하는 문제
  - **전기 회로**: 단자들을 모두 연결하면서 전선의 길이를 최소가 되도록 하는 문제

최소 신장 트리 – Kruskal 알고리즘 (1/18)

❑ Kruskal 알고리즘

- 그래프에 속한 간선들 중 가중치가 가장 작은 간선을 선택 후, 싸이클을 만들지 않으면 MST 간선에 추가하고, 싸이클을 만들면 제외시키는 연산을 반복하여 N-1 개의 간선이 MST에 추가 되었을 때 알고리즘을 종료, 여기서 N은 정점의 수
- Kruskal 알고리즘은 간선을 기반으로 동작
- 그래프 G = (V, E)가 주어졌을 때. Kruskal 알고리즘에서 초기의 최소 신장 트리 MST는 공집합이며, (i) 가중치가 가장 작은 간선 e (∈ E)를 선택하기 위해 E에 포함된 간선들을 가중치를 기준으로 오름차순 정렬 후 저장할 수 있는 리스트 L과 (ii) e를 MST에 추가하였을 때 싸이클이 생성되는지 여부를 판별하기 위한 추가적인 자료구조를 사용

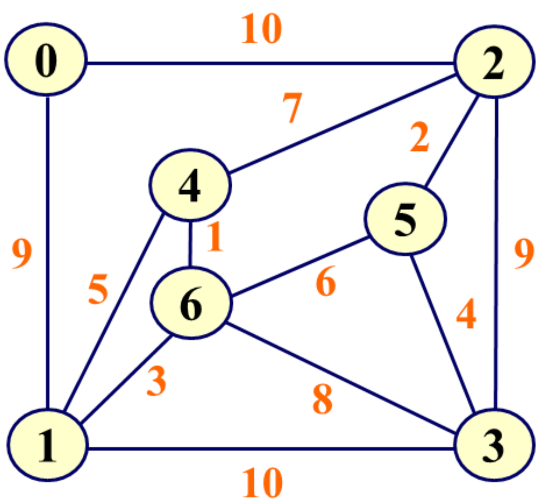
-Kruskal 알고리즘 설계 과정-

- 그래프 G = (V, E)가 주어졌을 때, 초기 최소 신장 트리 MST = { }를 생성하고 E에 포함된 간선들을 가중치를 기준으로 오름차순 정렬 후 리스트 L에 저장
- 서로소(disjoint)가 되는 V의 부분집합들을 생성하는데, 각 부분집합마다 하나의 정점만 원소로 포함하게 함(예: V = {0, 1, 2, 3}이라면 부분집합 {0}, {1}, {2}, {3} 생성)
- 최종해를 얻지 못하는 동안 다음 절차를 계속 반복:
  - **선정과정:** L에서 다음 간선(가중치의 값이 최소인 간선) e = (i, j)를 선정 후 L에서 제거
  - **적정성 검사**: 만약 e = (i, j)를 추가 시 싸이클이 생성되지 않는다면(혹은 i를 포함하는 부분집합과 j를 포함하는 부분집합이 서로소이면)
    - i를 포함하는 부분집합과 j를 포함하는 부분집합을 합한 후, e를 MST에 추가
  - **해답 점검**: 만약 |MST| = |V| – 1이라면 (혹은 만약 모든 정점들의 부분집합이 하나의 집합으로 합하여 지면), MST는 최소 신장 트리임

- 가중치 기준 오름차순으로 간선 E에 대한 리스트 L을 생성
- **repeat**:
  - L에서 가장 작은 가중치를 가진 간선 e를 선택 후, e를 L에서 제거
  - **if** MST에 간선 e를 추가 시 싸이클이 생성되지 않는다면:
    - 간선 e를 MST에 추가
- **until** |MST| == |V| – 1

❑ Kruskal 알고리즘 contd.

[예제] 아래의 그래프에서 Kruskal 알고리즘 수행 과정



E

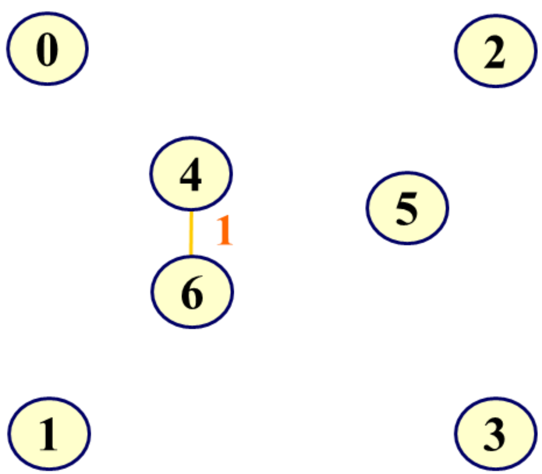
(0, 1)	9
(0, 2)	10
(1, 3)	10
(1, 4)	5
(1, 6)	3
(2, 3)	9
(2, 4)	7
(2, 5)	2
(3, 5)	4
(3, 6)	8
(4, 6)	1
(5, 6)	6

L 가중치 기준 오름차순으로 정렬

(4, 6)	1
(2, 5)	2
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10

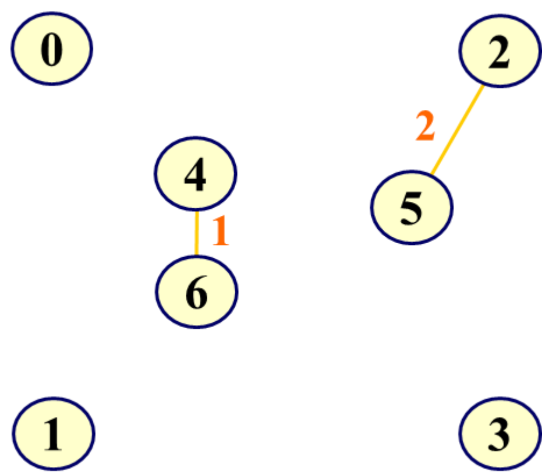
L

(4, 6)	1
(2, 5)	2
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



L

(2, 5)	2
(1, 6)	3
(3, 5)	4
(1, 4)	5
(5, 6)	6
(2, 4)	7
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



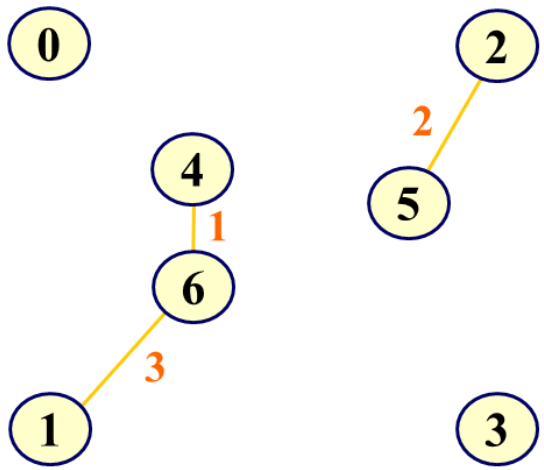
최소 신장 트리 – Kruskal 알고리즘 (3/18)

❑ Kruskal 알고리즘 contd.

[예제] 아래의 그래프에서 Kruskal 알고리즘 수행 과정 contd.

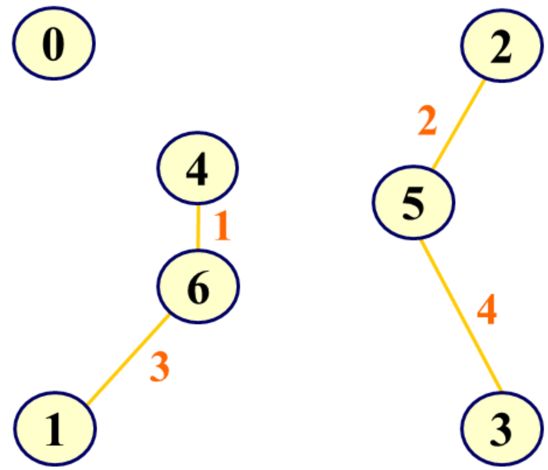
L

(1, 6) 3
(3, 5) 4
(1, 4) 5
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



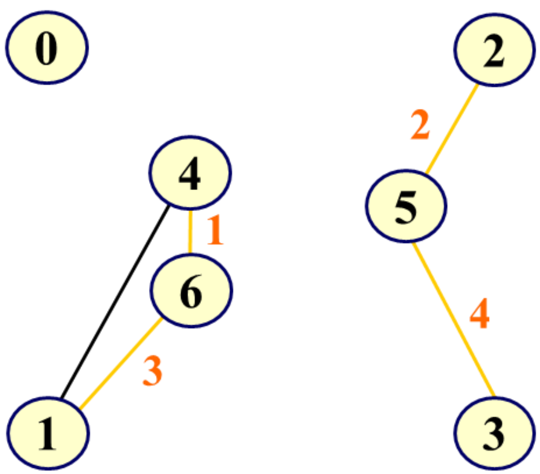
L

(3, 5) 4
(1, 4) 5
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



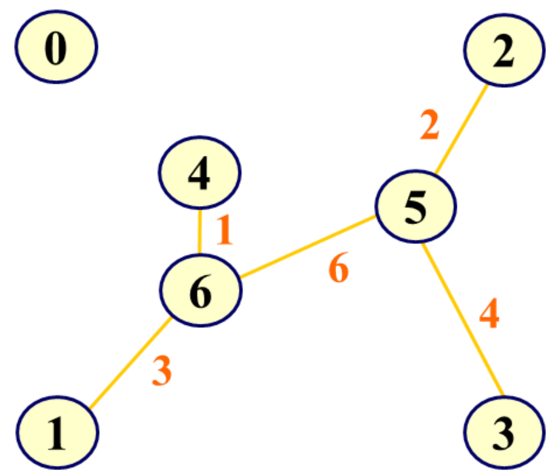
L

(1, 4) 5
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



L

(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



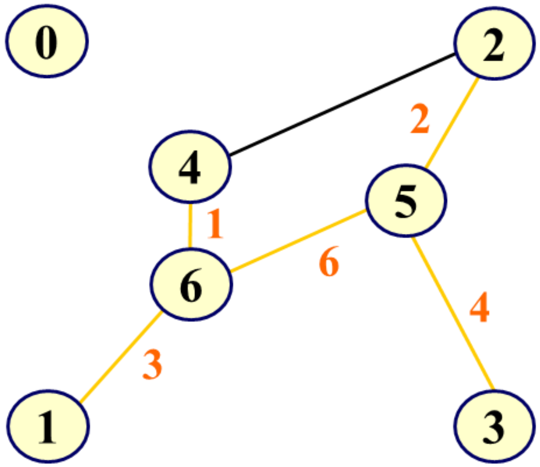
최소 신장 트리 – Kruskal 알고리즘 (4/18)

❑ Kruskal **알고리즘** contd.

[예제] 아래의 그래프에서 Kruskal 알고리즘 수행 과정 contd.

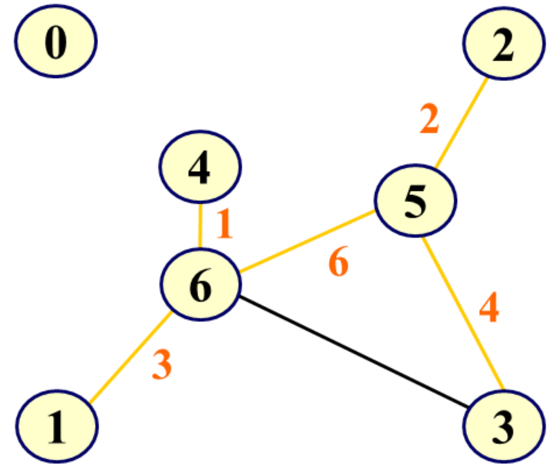
L

(2, 4)	**7**
(3, 6)	8
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



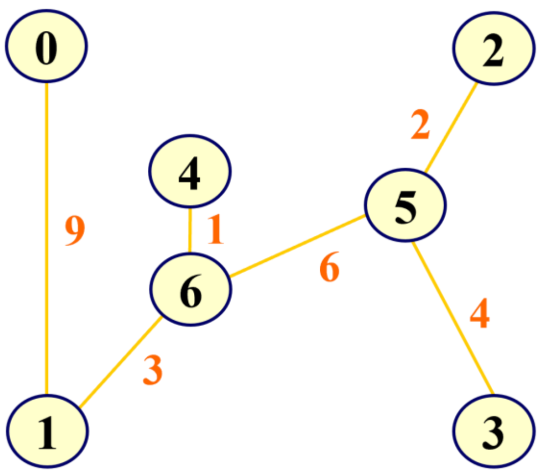
L

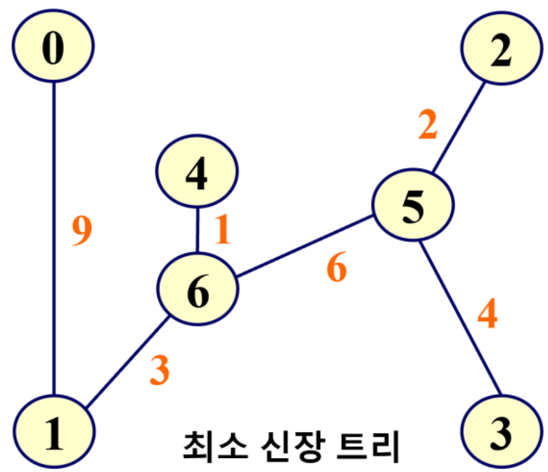
(3, 6)	**8**
(0, 1)	9
(2, 3)	9
(0, 2)	10
(1, 3)	10



L

(0, 1)	**9**
(2, 3)	9
(0, 2)	10
(1, 3)	10





최소 신장 트리

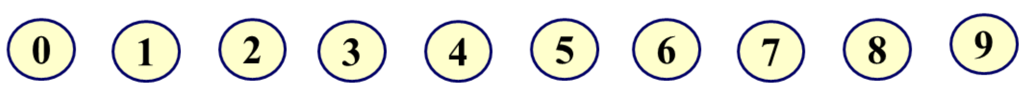
최소 신장 트리 – Kruskal 알고리즘 (5/18)

Kruskal **알고리즘** contd.

- **서로소 집합(disjoint sets)과 Union & Find 연산**
  - MST에 간선 추가 시 싸이클 생성 여부를 어떻게 판단해야 할까?
  - Kruskal 알고리즘에서 MST에 추가하려는 간선이 싸이클을 생성하는지의 여부는 (1) **서로소 집합**, (2) **서로소 집합과 관련된 연산인 union(합집합) 연산** 및 (3) 주어진 원소에 대해 어느 집합에 속해 있는지를 찾는 **find 연산**을 활용
  - **서로소 집합**
    - 집합이 주어졌을 때 서로 중복된 원소를 포함하지 않는 부분 집합
    - 서로소 집합은 (논리적으로)**트리의 형태로 표현** 가능하며, **리스트로 저장**할 수 있음
    - Kruskal 알고리즘은 그래프의 각 정점을 유일한 원소로 하는 N 개의 (서로소) 집합을 생성, 여기서 N은 정점의 수
      - » 그 후, union & find 연산으로 서로소 집합을 유지하는 MST를 찾음
    - 각 정점을 유일한 원소로 하는 N(= 10) 개의 **(서로소) 집합**을 10 개의 (논리적) **트리로 표현**하고 (물리적) Python 리스트에 저장한 예:

      왜? 각 집합 s마다 트리로 표현되고 s는 원소가 하나이므로
      - » 리스트의 각 인덱스(**INDEX**)는 트리의 각 노드(그래프의 각 정점)에 해당
      - » 리스트의 각 인덱스 위치에는 (1) 루트 노드의 경우 루트 자신이 저장되며, (2) 루트 노드가 아닌 노드는 자신의 부모 노드가 저장(아래의 그림은 **각 원소를 루트로 하는 10 개의 트리**를 저장)

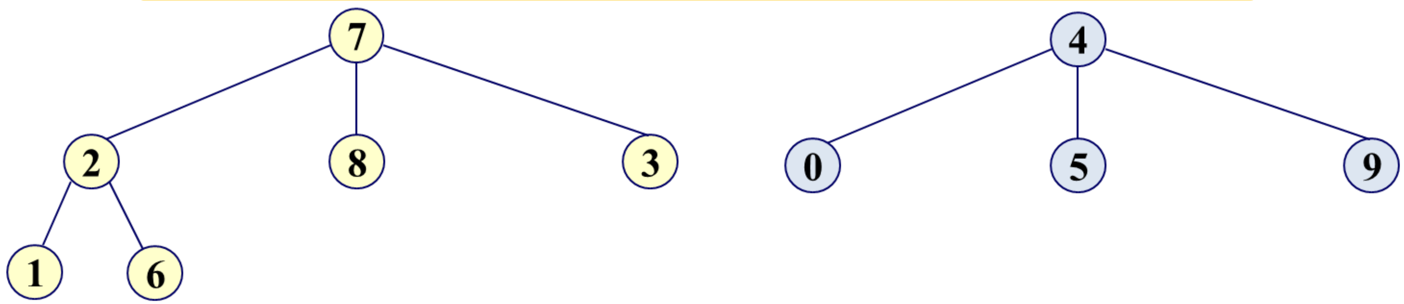
INDEX:	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9



최소 신장 트리 – Kruskal 알고리즘 (6/18)

- Kruskal 알고리즘 contd.
  - **서로소 집합(disjoint sets)과 Union & Find 연산 contd.**
    - **서로소 집합 contd.**
      - 두 개의 서로소 집합 {7, 2, 8, 3, 1, 6}과 {4, 0, 5, 9}를 (논리적) 트리로 표현하고 물리적 Python 리스트에 저장한 예:

INDEX:	0	1	2	3	4	5	6	7	8	9
	4	2	7	7	4	4	2	7	7	4



    - **union(A, B)**
      - 집합 A와 집합 B의 합집합 연산 → **대표 값(서로소 집합을 트리로 표현할 경우 루트 노드)**
      - given sets: {3, 5, 7} , {4, 2, 8}, {9}, {1, 6} → **각 집합은 대표 값을 이름으로 함: 집합 5, 집합 8, 집합 9, 집합 1**
      - union(5, 1): {3, 5, 7, 1, 6}, {4, 2, 8}, {9}
        - » 대표 값을 사용하는 이유는 Kruskal 알고리즘에서 두 노드(정점) u와 v로 구성된 간선을 선택 했을 때, u와 v가 **같은 집합에 속하는지 아닌지를 판단(i.e., 트리에서 동일한 루트 노드를 가지는지 아닌지를 판단)**하기 위함임(만약 대표 값이 같다면 서로 같은 집합에 속한다는 사실을 알 수 있음)

최소 신장 트리 – Kruskal 알고리즘 (7/18)

- Kruskal 알고리즘 contd.
  - **서로소 집합(disjoint sets)과 Union & Find 연산 contd.**
    - **find(u)**
      - u를 포함하는 집합의 이름(대표 값 혹은 트리의 루트 노드)을 검색하는 연산
      - given sets: {7, 2, 8, 3, 1, 6}과 {4, 0, 5, 9}
      - find(6): 7 반환, find(9): 4 반환

INDEX:

0	1	2	3	4	5	6	7	8	9
4	2	7	7	4	4	2	7	7	4

» find(6)은 p[6] = 2를 통해 6의 부모인 2를 찾고,p[2] = 7로 2의 부모인 7을 찾으며, 마지막으로 p[7] = 7이기 때문에 7을 반환함(즉, **“6은 7이 대표인 집합에 속해 있다”**는 사실을 반환함) **NOTE**: 리스트는 단지 각 트리 노드의 부모 노드 정보만 저장함 → 따라서 재귀호출을 통해 특정 노드의 루트 노드에 접근

→ : 호출 → : 반환

find(6) 호출 ↑ **7 반환**

6 != 2이므로 ↓ ↑ **7 반환**

find(2) 호출

2 != 7이므로 ↓ ↑ **7 반환**

find(7) 호출

7 == 7이므로 7 반환

```
def find(u):
    if u != p[u]: # p는 서로소 집합을 담고 있는 Python 리스트
        return find(p[u])
    return p[u]
```

» find(3)도 7을 반환하므로, 6과 3은 동일한 집합에 속함

» find(9) = 4이므로, 6과 9는 서로 다른 집합에 속함

- 평균적으로 특정 노드의 루트를 접근하기 위해서는 O(logN), 최악의 경우는?

INDEX:

0	1	2	3	4	5	6	7	8	9
4	2	7	7	4	4	2	7	7	4

최소 신장 트리 – Kruskal 알고리즘 (8/18)

- Kruskal 알고리즘 contd.
  - **서로소 집합(disjoint sets)과 Union & Find 연산 contd.**
    - **find(u) contd.**

INDEX:	0	1	2	3	4	5
	0	0	1	2	3	4

- 최악의 경우 find(5): O(N)

→ : 호출 → : 반환

↑ 0 반환

find(5) 호출

5 != 4이므로 ↓ ↑ 0 반환

find(4) 호출

4 != 3이므로 ↓ ↑ 0 반환

find(3) 호출

3 != 2 이므로 ↓ ↑ 0 반환

find(2) 호출

2 != 1이므로 ↓↑ 0 반환

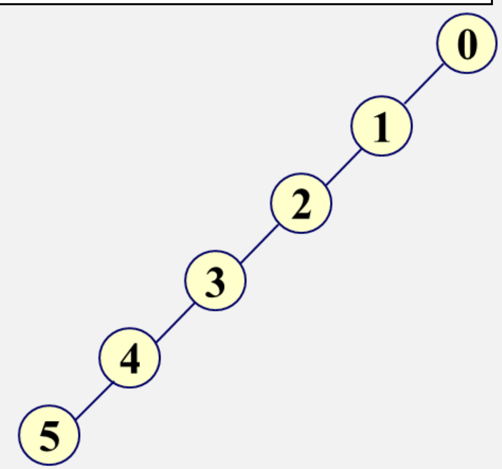
find(1) 호출

1 != 0이므로 ↓↑ 0 반환

find(0) 호출

0 == 0이므로 0 반화

```
def find(u):
    if u != p[u]: # p는 서로소 집합을 담고 있는 Python 리스트
        return find(p[u])
    return p[u]
```



최소 신장 트리 – Kruskal 알고리즘 (9/18)

- Kruskal 알고리즘 contd.
  - **서로소 집합(disjoint sets)과 Union & Find 연산 contd.**

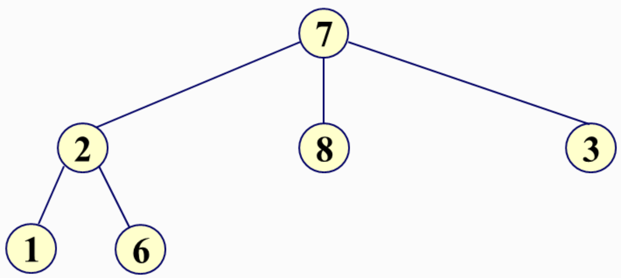
- 참고: find 연산에서 경로 압축 -

- find 연산 시 경로 압축을 수행할 수 있는데, 이는 나중에 수행되는 find 연산을 더 빠르게 수행하기 위해서임
- find 연산을 수행하면서 루트 노드까지 올라가는 경로 상의 각 노드의 부모를 루트 노드로 갱신하는 것을 **경로 압축(path compression)**이라고 함

-경로 압축의 예-

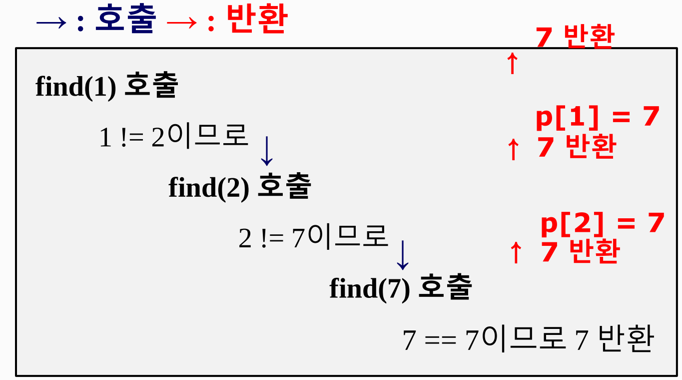
find(1) 연산 수행 전

INDEX:	0	1	2	3	4	5	6	7	8	9
	4	2	7	7	4	4	2	7	7	4



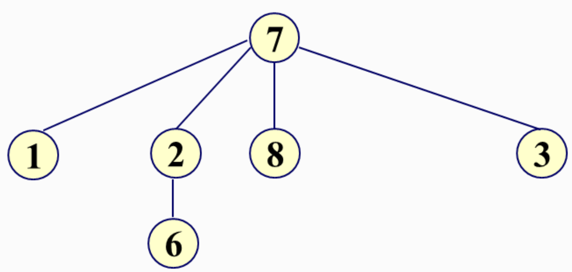
```
def find(u):
    if u != p[u]:
        p[u] = find(p[u])
    return p[u]
```

→ : 호출 → : 반환



find(1) 연산 수행 후

INDEX:	0	1	2	3	4	5	6	7	8	9
	4	7	7	7	4	4	2	7	7	4

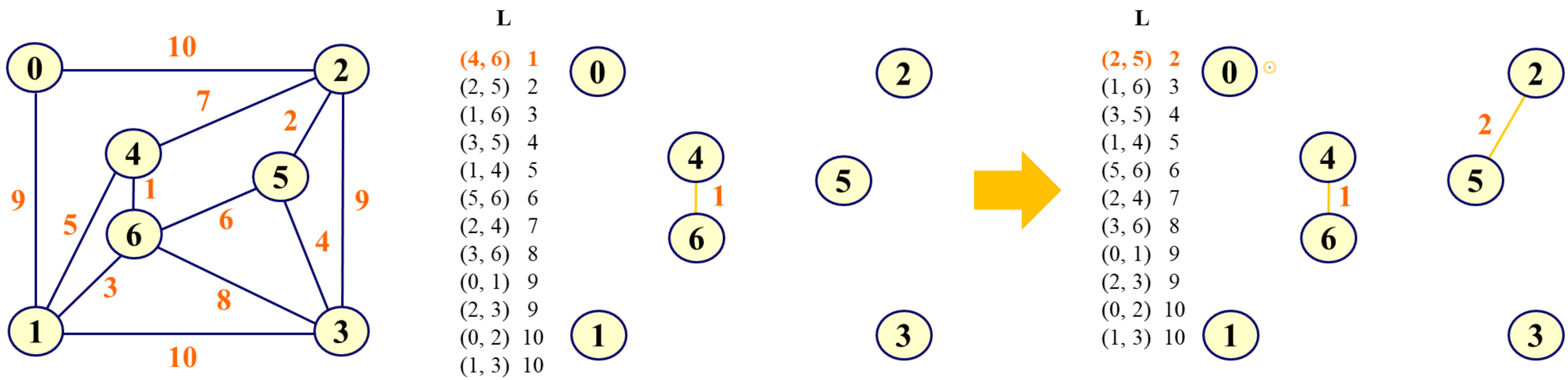


find(1) 수행 시 경로 압축은 당장 find(1)의 수행 시간이 줄어들지는 않으나, 추후의 find(1)의 수행 시간을 단축

최소 신장 트리 – Kruskal 알고리즘 (10/18)

❑ Kruskal 알고리즘 contd.

- **Kruskal 알고리즘에서 서로소 집합 적용**
  - 그래프 G = (V, E)가 주어졌을 때 알고리즘 초기에 서로소가 되는 V의 부분집합들을 생성하는데, 각 부분집합마다 하나의 정점만 원소로 포함하게 함(아래의 예에서 V = {0, 1, 2, 3, 4, 5, 6}이므로 부분집합 {0}, {1}, {2}, {3}, {4}, {5}, {6} 생성)



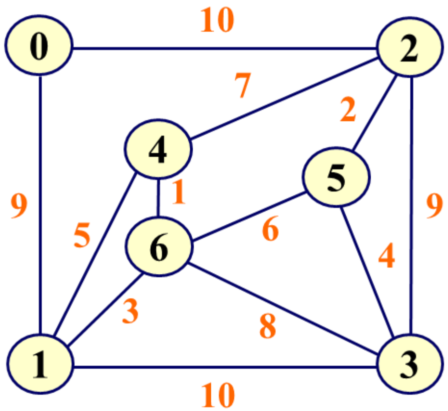
  - 위의 그림에서 간선 (4, 6)을 MST에 추가 시, 정점 4가 포함된 집합(트리) 4(대표 값)와 정점 6이 포함된 집합 6이 동일한지 확인, 동일하지 않으면(find(4) != find(6)) 집합 4와 집합 6을 합한 후, i.e., union(4, 6), 간선 (4, 6)을 MST에 추가. 동일한 방법으로 간선 (2, 5)를 MST에 추가 시, find(2) != find(5)므로 union(2, 5) 후, 간선 (2, 5)를 MST에 추가
    - 현재 MST: { (4, 6), (2, 5) }, 현재 부분 집합: {0}, {1}, {2, 5}, {3}, {4, 6}

최소 신장 트리 – Kruskal 알고리즘 (11/18)

❑ Kruskal 알고리즘 contd.

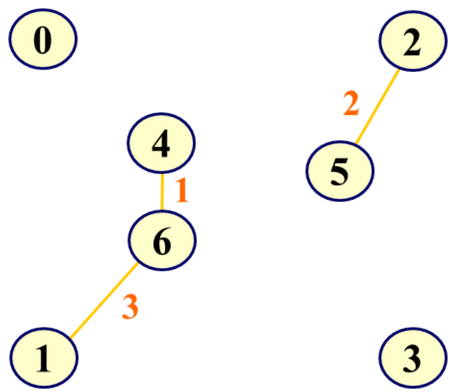
- **Kruskal 알고리즘에서 서로소 집합 적용 contd.**

이전 부분 집합: **{0}, {1}, {2, 5}, {3}, {4, 6}**



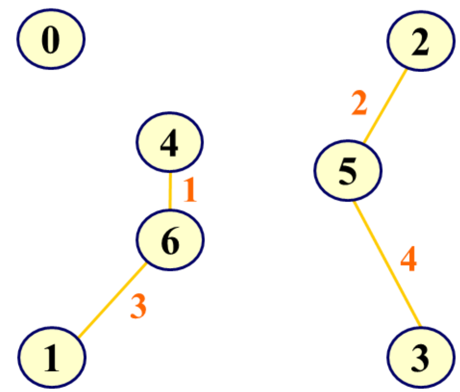
L

(1, 6) 3
(3, 5) 4
(1, 4) 5
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



L

(3, 5) 4
(1, 4) 5
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



– 위의 그림에서 간선 (1, 6)을 MST에 추가 시, 정점 1이 포함된 집합(트리) 1과 정점 6이 포함된 집합 4(대표값)가 동일한지 확인, find(1) != find(6)이므로 집합 1과 집합 4를 합한 후, i.e., union(1, 6), 간선 (1, 6)을 MST에 추가. 동일한 방법으로 간선 (3, 5)를 MST에 추가 시, find(3) != find(5)므로 집합 3과 집합 2를 합한 후, i.e., union(3, 5) 후, 간선 (3, 5)를 MST에 추가

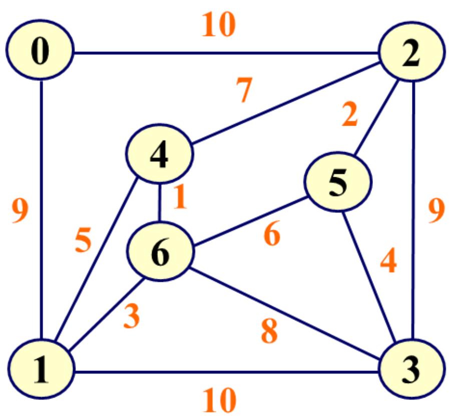
  - 현재 MST: { (4, 6), (2, 5), (1, 6), (3, 5) }, 현재 부분 집합: {0}, {1, 4, 6}, {2, 3, 5}

```
def union(u, v): # 노드 u가 속한 집합(트리)과 노드 v가 속한 집합을 합함
    root1 = find(u)
    root2 = find(v)
    p[root2] = root1
```

최소 신장 트리 – Kruskal 알고리즘 (12/18)

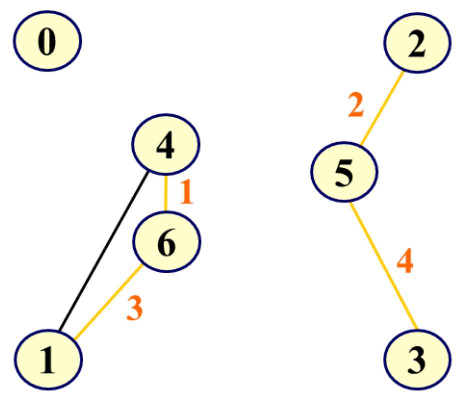
❑ Kruskal **알고리즘** contd.

- **Kruskal 알고리즘에서 서로소 집합 적용 contd.**



L

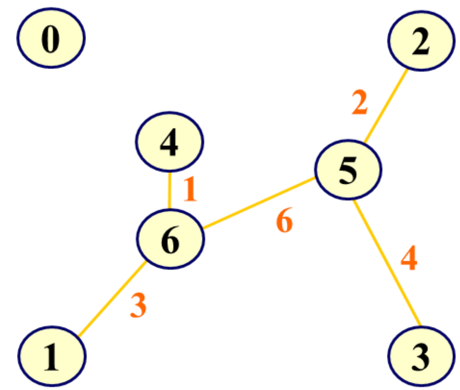
(1, 4) 5
(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



이전 부분 집합: **{0}, {1, 4, 6}, {2, 3, 5}**

L

(5, 6) 6
(2, 4) 7
(3, 6) 8
(0, 1) 9
(2, 3) 9
(0, 2) 10
(1, 3) 10



– 위의 그림에서 간선 (1, 4)를 MST에 추가 시, 정점 1이 포함된 집합(트리) 1과 정점 4가 포함된 집합 1이 동일한지 확인, find(1) == find(4)이므로 정점 1과 정점 4는 동일한 집합에 포함되어 있음 → 위의 그림처럼 **이미 동일한 집합에 포함된 정점들을 연결하는 간선을 새로 추가하게 되면 반드시 싸이클이 생성**될 수밖에 없으므로 **간선 (1, 4)는 MST에 추가하지 않음**. 그 후, 간선 (5, 6)을 MST에 추가 시, find(5) != find(6)므로 집합 1과 집합 2를 합한 후, i.e., union(5, 6) 후, 간선 (5, 6)을 MST에 추가

  - 현재 MST: { (4, 6), (2, 5), (1, 6), (3, 5), (5, 6) }, 현재 부분 집합: {0}, {1, 2, 3, 4, 5, 6}

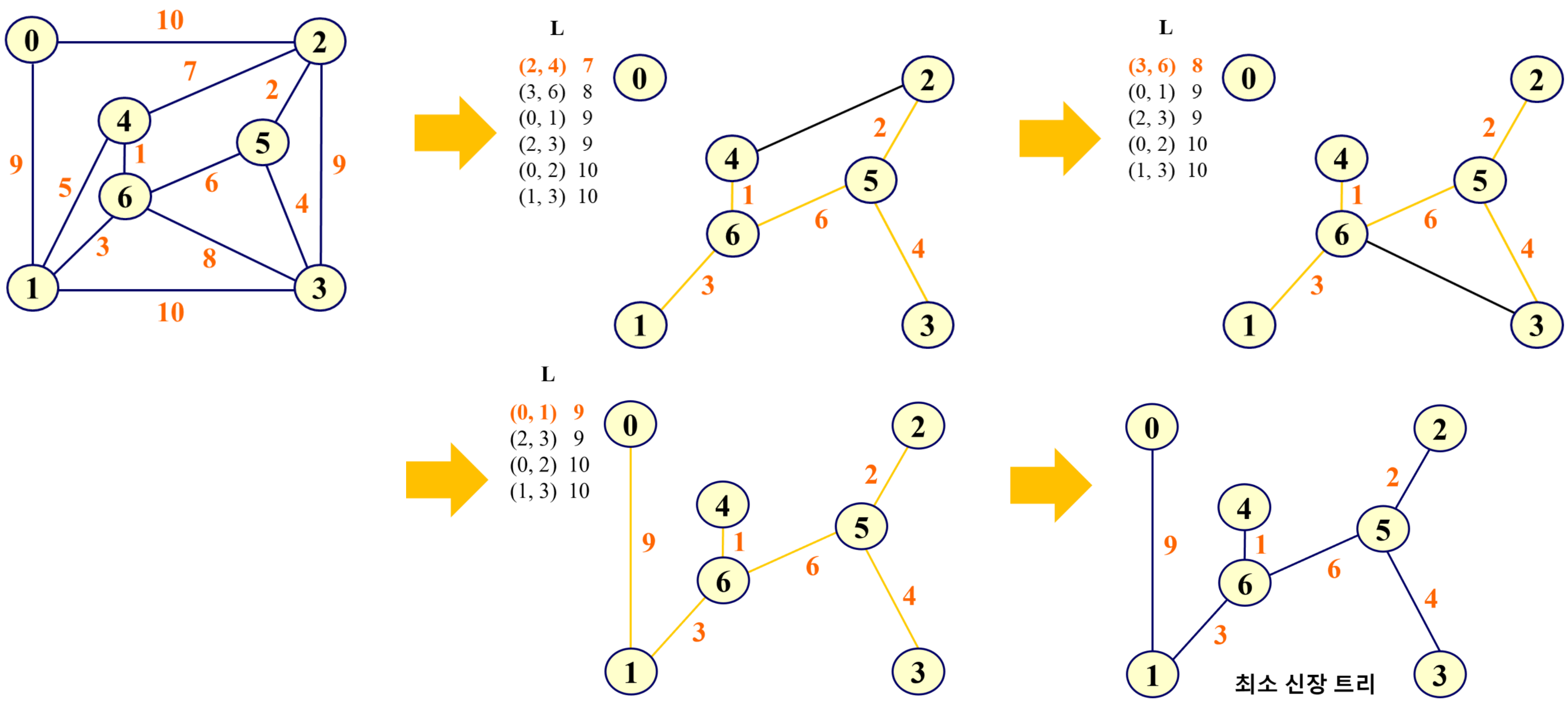
```
def union(u, v): # 노드 u가 속한 집합(트리)과 노드 v가 속한 집합을 합함
    root1 = find(u)
    root2 = find(v)
    p[root2] = root1
```

최소 신장 트리 – Kruskal 알고리즘 (13/18)

❑ Kruskal 알고리즘 contd.

- **Kruskal 알고리즘에서 서로소 집합 적용 contd.**

이전 부분 집합: **{0}, {1, 2, 3, 4, 5, 6}**



– 위의 그림에서 간선 (2, 4)를 MST에 추가 시, 정점 2가 포함된 집합(트리) 1과 정점 4가 포함된 집합 1이 동일한지 확인, find(2) == find(4)이므로 **간선 (2, 4)는 MST에 추가하지 않음**. 그 후, 간선 (3, 6)을 MST에 추가 시, find(3) == find(6)이므로 간선 (3, 6)도 **MST에 추가하지 않음**. 마지막으로 간선 (0, 1)을 MST에 추가 시 find(0) != find(1)이므로 집합 0과 집합 1을 합한 후, i.e., union(0, 1) 후, 간선 (0, 1)을 MST에 추가. MST에 포함된 간선의 수가 N – 1이므로(혹은 모든 정점들의 부분집합이 하나의 집합으로 합하여졌으므로) 알고리즘 종료

  - 최종 MST: { (4, 6), (2, 5), (1, 6), (3, 5), (5, 6), (0, 1) }, 최종 집합: {0, 1, 2, 3, 4, 5, 6}

최소 신장 트리 – Kruskal 알고리즘 (14/18)

- Kruskal 알고리즘 contd.

  - **Kruskal 알고리즘에 대한 Python 코드**

```
1  weights = [(0, 1, 9), (0, 2, 10), (1, 3, 10), (1, 4, 5),
2             (1, 6, 3), (2, 3, 9), (2, 4, 7), (2, 5, 2),
3             (3, 5, 4), (3, 6, 8), (4, 6, 1), (5, 6, 6)  ]
4  weights.sort(key = lambda t: t[2])
5  mst = []
6  N = 7
7  p = []
8  for i in range(N):
9      p.append(i)
10
11 def find(u):
12     if u != p[u]:
13         p[u] = find(p[u])
14     return p[u]
15
16 def union(u, v):
17     root1 = find(u)
18     root2 = find(v)
19     p[root2] = root1
```

계속

```
21 tree_edges = 0
22 mst_cost = 0
23 while True:
24     if tree_edges == N - 1:
25         break
26     u, v, wt = weights.pop(0)
27     if find(u) != find(v):
28         union(u, v)
29         mst.append((u, v))
30         mst_cost += wt
31         tree_edges += 1
32
33 print('최소신장트리: ', end='')
34 print(mst)
35 print('최소신장트리 가중치:', mst_cost)
```

- **라인 1-3:** 그래프 간선들의 집합(간선의 두 정점과 가중치)
- **라인 4:** 그래프 간선들의 집합을 가중치 기준으로 정렬(sort 함수에 key 매개변수 사용,- sort 메소드와 lambda 함수 참조)
- **라인** 5: MST를 구성하는 간선들을 담을 빈 Python 리스트 생성
- **라인 7:** 서로소 집합들을 담을 빈 Python 리스트 생성

최소 신장 트리 – Kruskal 알고리즘 (15/18)

❑ Kruskal 알고리즘 contd.

- **Kruskal 알고리즘에 대한 Python 코드 contd.**

```python
 1 weights = [(0, 1, 9), (0, 2, 10), (1, 3, 10), (1, 4, 5),
 2            (1, 6, 3), (2, 3, 9), (2, 4, 7), (2, 5, 2),
 3            (3, 5, 4), (3, 6, 8), (4, 6, 1), (5, 6, 6)  ]
 4 weights.sort(key = lambda t: t[2])
 5 mst = []
 6 N = 7
 7 p = []
 8 for i in range(N):
 9     p.append(i)
10 
11 def find(u):
12     if u != p[u]:
13         p[u] = find(p[u])
14     return p[u]
15 
16 def union(u, v):
17     root1 = find(u)
18     root2 = find(v)
19     p[root2] = root1
```

계속

```python
21 tree_edges = 0
22 mst_cost = 0
23 while True:
24     if tree_edges == N - 1:
25         break
26     u, v, wt = weights.pop(0)
27     if find(u) != find(v):
28         union(u, v)
29         mst.append((u, v))
30         mst_cost += wt
31         tree_edges += 1
32 
33 print('최소신장트리: ', end='')
34 print(mst)
35 print('최소신장트리 가중치:', mst_cost)
```

-find(u)-

- **라인 12: if** 노드(정점) v가 루트 노드가 아니라면,
  - **라인 13**: (루트 노드에 도달할 때까지) v의 현재 부모 노드 vp의 부모 노드va를 v의 부모 노드로 변경 (**경로 압축**)
  - **라인 14**: 루트 노드인 대표 값 반환

최소 신장 트리 – Kruskal 알고리즘 (16/18)

- Kruskal 알고리즘 contd.

  - **Kruskal 알고리즘에 대한 Python 코드 contd.**

```python
1  weights = [(0, 1, 9), (0, 2, 10), (1, 3, 10), (1, 4, 5),
2             (1, 6, 3), (2, 3, 9), (2, 4, 7), (2, 5, 2),
3             (3, 5, 4), (3, 6, 8), (4, 6, 1), (5, 6, 6)  ]
4  weights.sort(key = lambda t: t[2])
5  mst = []
6  N = 7
7  p = []
8  for i in range(N):
9      p.append(i)
10
11 def find(u):
12     if u != p[u]:
13         p[u] = find(p[u])
14     return p[u]
15
16 def union(u, v):
17     root1 = find(u)
18     root2 = find(v)
19     p[root2] = root1
```

계속

```python
21 tree_edges = 0
22 mst_cost = 0
23 while True:
24     if tree_edges == N - 1:
25         break
26     u, v, wt = weights.pop(0)
27     if find(u) != find(v):
28         union(u, v)
29         mst.append((u, v))
30         mst_cost += wt
31         tree_edges += 1
32
33 print('최소신장트리: ', end='')
34 print(mst)
35 print('최소신장트리 가중치:', mst_cost)
```

-union(u, v)-

- **라인 17-18:** u와 v의 루트 노드(대표 값)를 찾아서 각각 root1, root2에 할당
- **라인 19**: (임의로) root2의 부모를 root1로 설정

최소 신장 트리 – Kruskal 알고리즘 (17/18)

- Kruskal 알고리즘 contd.

  - **Kruskal 알고리즘에 대한 Python 코드 contd.**

```python
1  weights = [(0, 1, 9), (0, 2, 10), (1, 3, 10), (1, 4, 5),
2             (1, 6, 3), (2, 3, 9), (2, 4, 7), (2, 5, 2),
3             (3, 5, 4), (3, 6, 8), (4, 6, 1), (5, 6, 6)  ]
4  weights.sort(key = lambda t: t[2])
5  mst = []
6  N = 7
7  p = []
8  for i in range(N):
9      p.append(i)
10
11 def find(u):
12     if u != p[u]:
13         p[u] = find(p[u])
14     return p[u]
15
16 def union(u, v):
17     root1 = find(u)
18     root2 = find(v)
19     p[root2] = root1
```

계속

```python
21 tree_edges = 0
22 mst_cost = 0
23 while True:
24     if tree_edges == N - 1:
25         break
26     u, v, wt = weights.pop(0)
27     if find(u) != find(v):
28         union(u, v)
29         mst.append((u, v))
30         mst_cost += wt
31         tree_edges += 1
32
33 print('최소신장트리: ', end='')
34 print(mst)
35 print('최소신장트리 가중치:', mst_cost)
```

- **라인 21-22:** 현재 MST 간선의 수와 MST 가중치를 저장하기 위한 변수 tree_edges, mst_cost 할당 및 0으로 초기화
- **라인 23**: 현재 MST 간선의 수가 N – 1 개가 될 때까지 while-루프 실행, 여기서 N은 그래프 정점의 수
- **라인 24-25**: 현재 MST 간선의 수가 N – 1 개라면 break
- **라인 26**: 그래프 간선들의 집합에서 최소의 가중치를 가진 간선을 pop 후 각각 u, v, wt에 할당(여기서 u, v는 두 정점이고 wt는 가중치)

최소 신장 트리 – Kruskal 알고리즘 (18/18)

- Kruskal 알고리즘 contd.
  - **Kruskal 알고리즘에 대한 Python 코드 contd.**

```python
1  weights = [(0, 1, 9), (0, 2, 10), (1, 3, 10), (1, 4, 5),
2             (1, 6, 3), (2, 3, 9), (2, 4, 7), (2, 5, 2),
3             (3, 5, 4), (3, 6, 8), (4, 6, 1), (5, 6, 6)  ]
4  weights.sort(key = lambda t: t[2])
5  mst = []
6  N = 7
7  p = []
8  for i in range(N):
9      p.append(i)
10
11 def find(u):
12     if u != p[u]:
13         p[u] = find(p[u])
14     return p[u]
15
16 def union(u, v):
17     root1 = find(u)
18     root2 = find(v)
19     p[root2] = root1
```

계속

```python
21 tree_edges = 0
22 mst_cost = 0
23 while True:
24     if tree_edges == N - 1:
25         break
26     u, v, wt = weights.pop(0)
27     if find(u) != find(v):
28         union(u, v)
29         mst.append((u, v))
30         mst_cost += wt
31         tree_edges += 1
32
33 print('최소신장트리: ', end='')
34 print(mst)
35 print('최소신장트리 가중치:', mst_cost)
```

```
최소신장트리: [(4, 6), (2, 5), (1, 6), (3, 5), (5, 6), (0, 1)]
최소신장트리 가중치: 25
```

결과

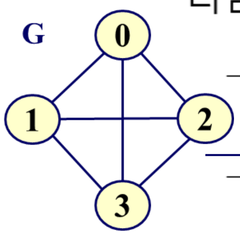
- **라인** 27: **if** u와 v의 루트 노드(대표 값)가 서로 다르다면,
  - **라인** 28: u가 속한 집합과 v가 속한 집합에 대해 합집합 연산 수행
  - **라인** 29-31: 간선 (u, v)를 MST에 추가, 현재 MST 가중치에 간선 (u, v)의 가중치를 합한 후 간선의 수 1 증가

Kruskal 알고리즘 분석

❑ Kruskal 알고리즘 분석

대소관계 비교를 통한 정렬의 이론적 하한

- 그래프 G = (V, E)가 주어졌을 때 입력(정점과 간선) 사이즈를 N (= |V|)과 M (= |E|)이라고 한다면, 간선들의 정렬에 **O(MlogN)**, 슬라이드 14-18의 알고리즘 라인 23의 반복문에 의해 find 및 union 연산을 M 번 수행하는데 find 및 union 연산의 수행 시간은 **O(logM)**, N 개의 서로소 집합을 초기화하는데 걸리는 시간 **O(N)** (슬라이드 20 의 알고리즘 라인 8의 반복문)



  - 그런데 G는 그래프이므로 M ≥ N – 1이므로 수행 시간은 O(MlogM)
  - G는 모든 정점이 다른 모든 정점과 연결이 될 때, $M = (N-1) + (N-2) + \cdots + 2 + 1 = \frac{N(N-1)}{2}$이므로 최악의 경우 Kruskal 알고리즘의 시간 복잡도는 $O(N^2 \log N^2) = O(2N^2 \log N) = O(N^2 \log N)$
  - (일반적으로) G가 밀집 그래프(dense graph)면, i.e., 간선의 수가 최대 간선의 수에 가까운 그래프면 Kruskal 알고리즘의 시간 복잡도는 $O(N^2 \log N)$, 희소 그래프(sparse graph)면, i.e., 간선의 수가 적은 그래프면 M ≈ N 이므로 Kruskal 알고리즘의 시간 복잡도는 O(NlogN) Note: 연결 그래프 G에서 $N - 1 \le M \le \frac{N(N-1)}{2}$

❑ Prim 알고리즘과의 비교

	최악의 경우 시간복잡도	sparse graph	dense graph
Prim Kruskal	$O(N^2)$ $O(N^2 \log N)$	$O(N^2)$ O(^{2}logN)	$O(N^2)$ $O(N^2 \log N)$

최소 신장 트리 – Prim 알고리즘 (1/7)

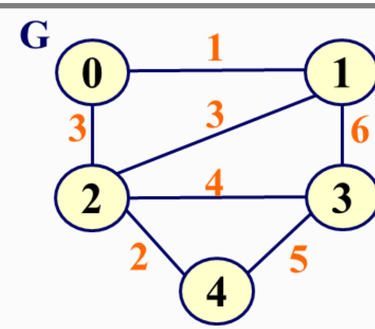
❑ Prim 알고리즘

- 임의의 시작 정점에서 가장 가까운 정점을 추가하여 간선이 하나인 MST를 만들고, 만들어진 MST에 인접한 가장 가까운 정점을 하나씩 추가하는 방식을 반복하여 N-1 개의 간선이 생성되었을 때 알고리즘을 종료, 여기서 N은 정점의 수
- **Prim 알고리즘은 정점을 기반**으로 동작(**Kruskal 알고리즘은 간선을 기반**으로 동작)
- 그래프 G = (V, E)가 주어졌을 때 Prim 알고리즘에서 초기의 최소 신장 트리 MST는 하나의 임의 정점 s(∈ V)만을 가지며, 현재 MST를 구성하는 정점(들)과 인접한 (MST에 속하지 않은) 정점들 중에서 간선의 가중치가 가장 작은 정점을 선택하기 위해 **리스트 dist**를 사용(dist[i]는 현재 **MST부터 정점 i까지의 최소 거리(최소 가중치)**를 저장)하며, 특정 정점이 MST에 이미 포함되었는지 여부를 위해 **리스트 visited**를 사용
  - 리스트 dist와 visited의 사이즈는 N

-Prim 알고리즘 설계 과정-

- 주어진 그래프에 G = (V, E)에서 임의의 시작 정점 s (∈ V)를 선택하여 dist[s] = 0으로 초기화 하고, dist[s]를 제외한 모든 dist의 항목 값을 ∞로 초기화
- **선정과정&적정성 검사**: MST에 속하지 않은 각 정점 v에 대하여 (1) **dist[v]가 최소**인 정점 u (i.e., 현재 MST에 가장 가까운 정점 u)를 찾아 MST에 추가 후 (2) MST에 속하지 않은 정점들(V – MST) 중 u와 인접한 각 정점 w에 대해 dist[w] 값 갱신 (NOTE: 적정성 **검사는 u를 MST에 추가했을 때 싸이클 존재 유무**에 대한 검사인데 Prim 알고리즘은 적정성 검사가 필요 없음)
- **해답 점검**: MST와 V가 동일한지 확인(MST = V면 최소 신장 트리 발견 아니면 선전과정&적정성 검사로 되돌아감)

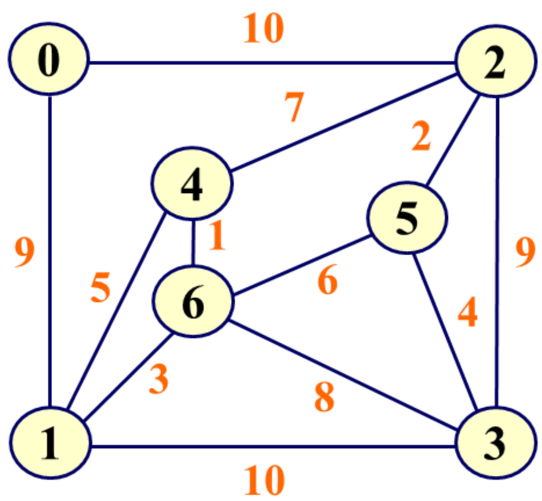
- **repeat**
  - MST에 속하지 않은 각 정점 v에 대하여 dist[v]가 최소인 정점 u를 찾아 MST에 추가
  - **for** MST에 속하지 않은 각 정점 w에 대하여: # **정점 u의 추가로 인한 dist의 항목 값 갱신을 위한 과정**
    - **if** 간선 (u,w)의 가중치 < dist[w]:
      - dist[w] = 간선 (u,w)의 가중치
- **until** |MST| == N

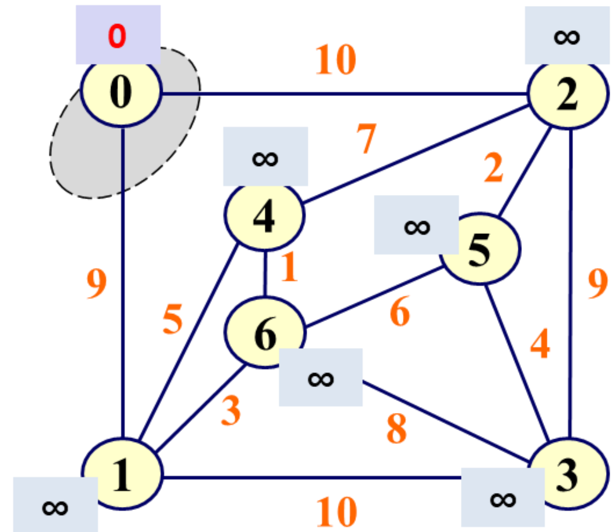


최소 신장 트리 – Prim 알고리즘 (2/7)

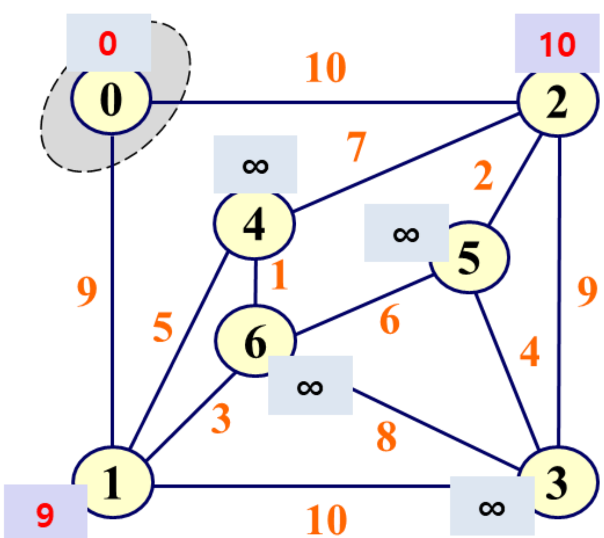
❑ Prim 알고리즘 contd.

[예제] Prim 알고리즘 수행 과정

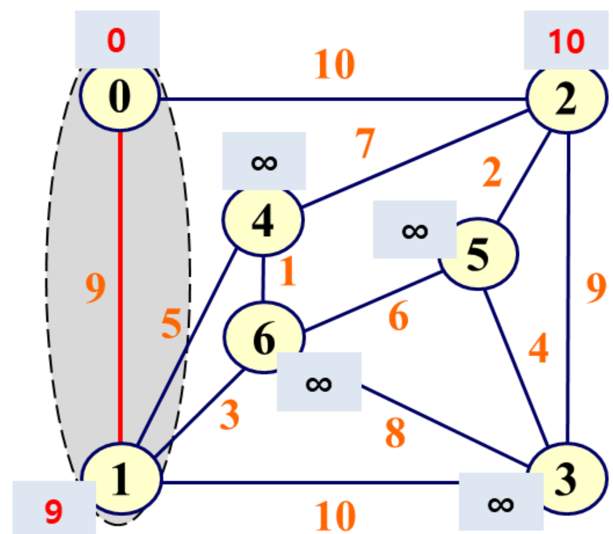




dist[0] = 0으로 초기화 하고, dist[0] 를 제외한 모든 dist의 항목 값을 ∞로 초기화 후 0을 MST에 포함



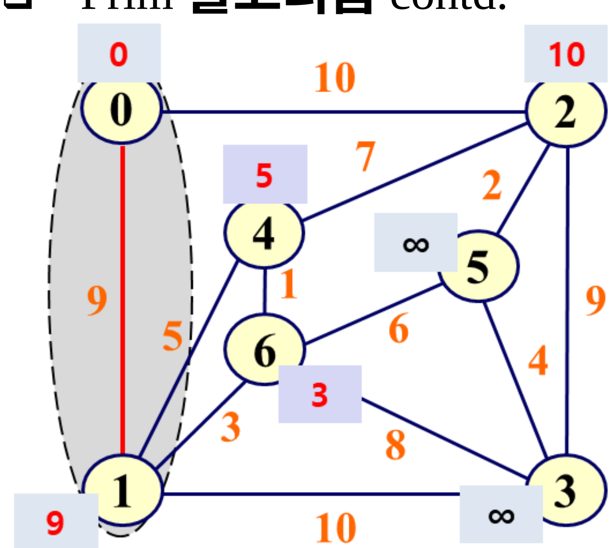
MST =<0>와 인접한 정점 1과 2에 대해 dist[1] = 9, dist[2] = 10으로 갱신



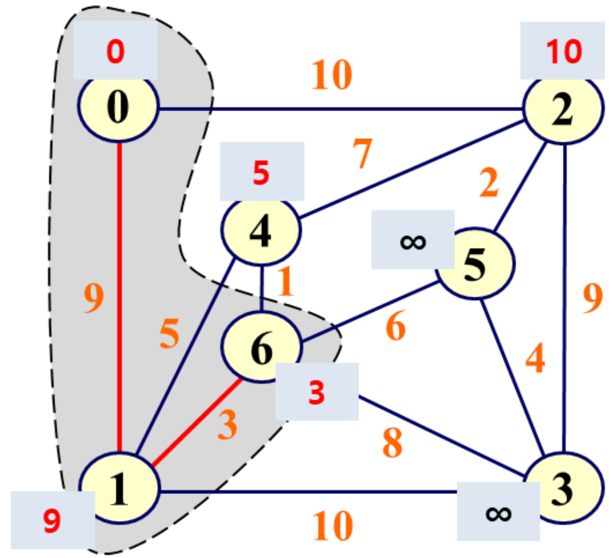
MST에 포함되지 않은 정점 중 dist[1]의 값(=9)이 최소이므로 1을 MST에 추가

최소 신장 트리 – Prim 알고리즘 (3/7)

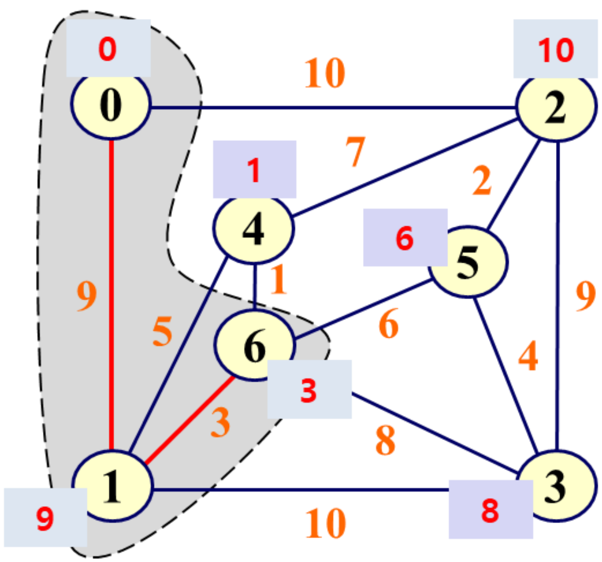
❑ Prim 알고리즘 contd.



MST =<0, 1>와 인접한 정점 2, 4, 6 중 4와 6에 대해 dist[4] = 5, dist[6] = 3으로 갱신

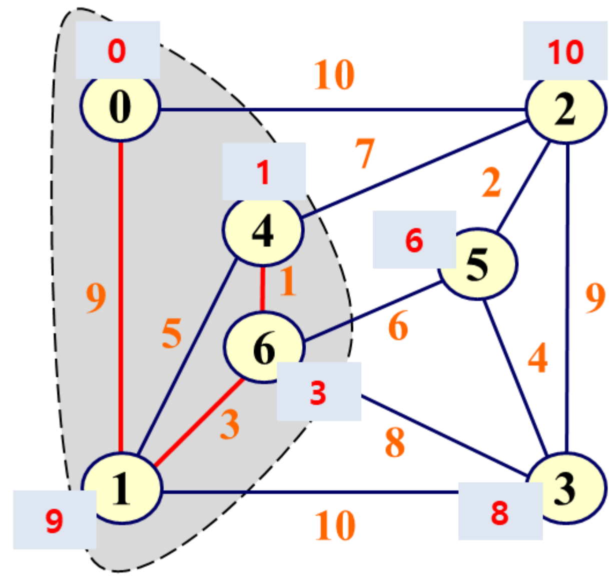


MST에 포함되지 않은 정점 중 dist[6]의 값(=3)이 최소이므로 6을 MST에 추가



MST =<0,1,6>와 인접한 정점 2, 4, 5, 3 중

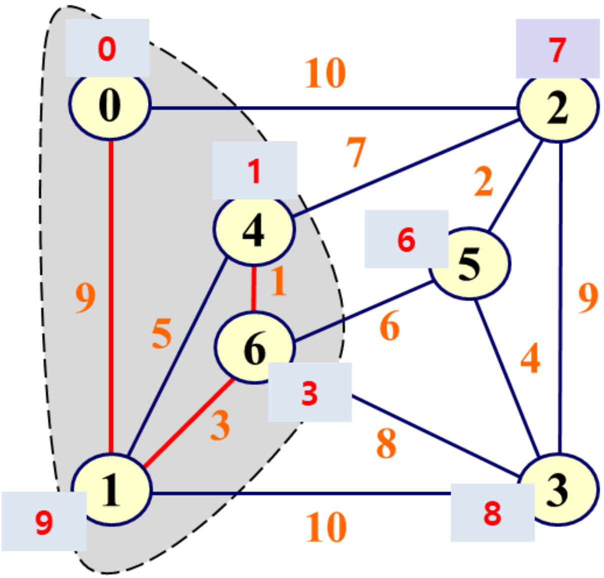
4, 5, 3에 대해 dist[4] = 1, dist[5] =6, dist[3] = 8으로 갱신



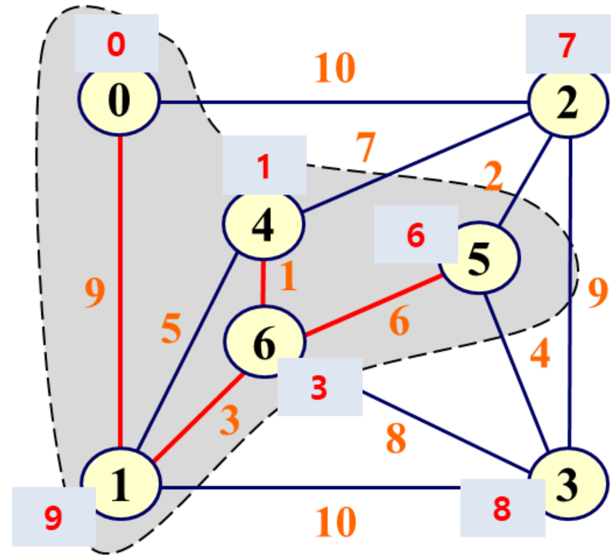
MST에 포함되지 않은 정점 중 dist[4]의 값(=1)이 최소이므로 4를 MST에 추가

최소 신장 트리 – Prim 알고리즘 (4/7)

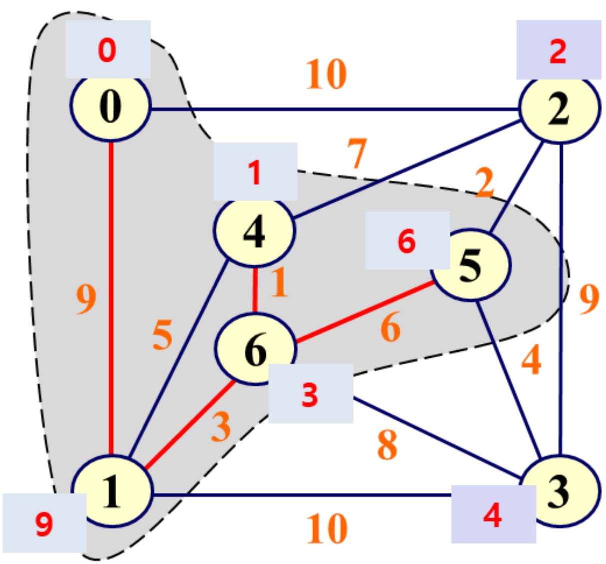
- Prim 알고리즘 contd.



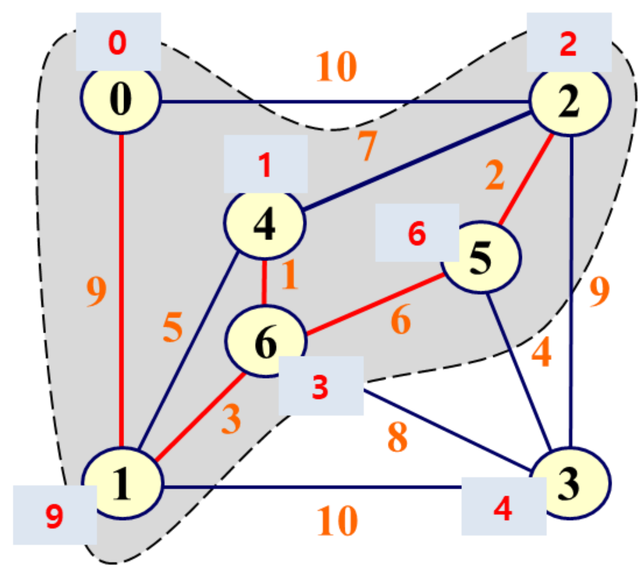
MST = <0,1,6,4>와 인접한 정점 2, 5, 3 중 2에 대해 dist[2] = 7로 갱신



MST에 포함되지 않은 정점 중 dist[5]의 값(=6)이 최소이므로 5를 MST에 추가



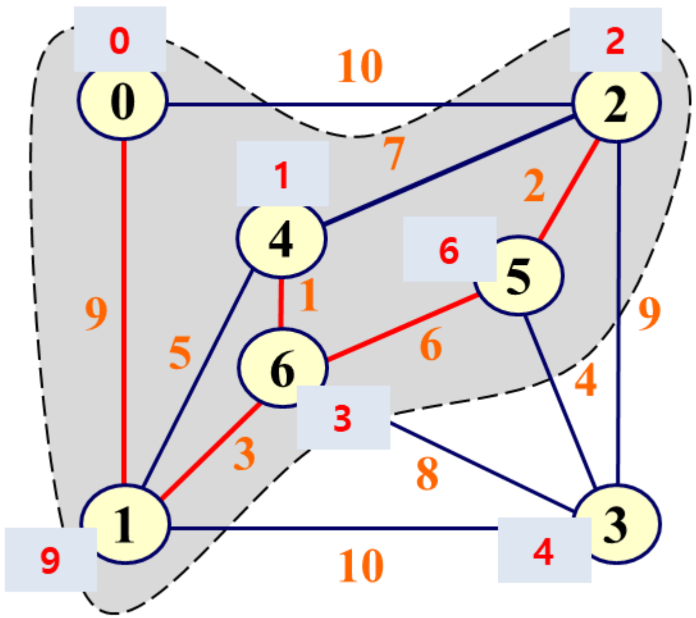
MST =<0,1,6,4,5>와 인접한 정점 2와 3 중 2와 3에 대해 dist[2] = 2, dist[3] =4로 갱신



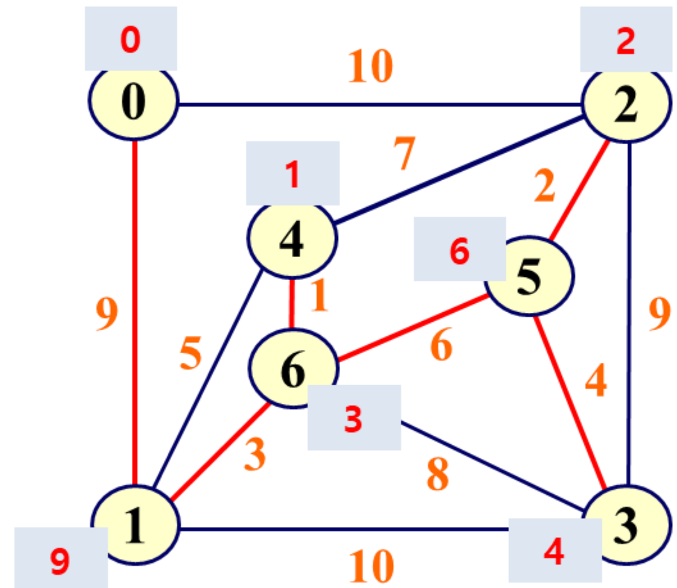
MST에 포함되지 않은 정점 중 dist[2]의 값(=2)이 최소이므로 2를 MST에 추가

최소 신장 트리 – Prim 알고리즘 (5/7)

❑ Prim 알고리즘 contd.

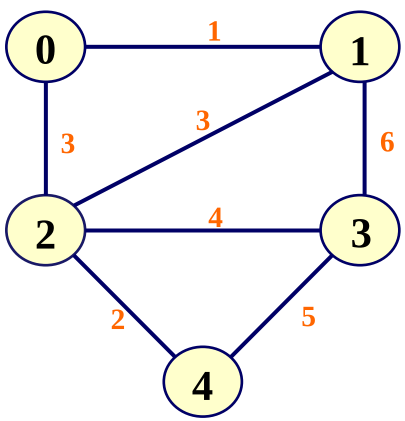


MST =<0,1,6,4,5,2>와 인접한 정점 3에 대해서는 dist[3]을 갱신할 필요가 없으므로 Pass

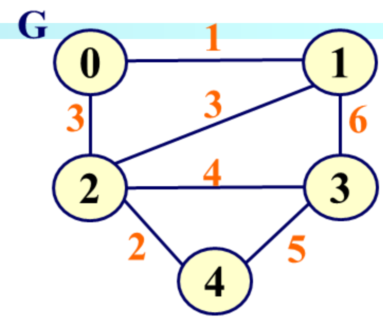


MST에 포함되지 않은 정점 중 3을 MST에 추가. 즉, MST =<0,1,6,4,5,2,3>

[연습] 아래와 같은 그래프가 주어졌을 때, Prim 의 알고리즘에 의해 선택된 처음 2 개의 트리 간선의 가중치의 합은? 단, 시작 정점은 4이다.



최소 신장 트리 – Prim 알고리즘 (6/7)



- Prim 알고리즘 contd.
  - **Prim 알고리즘에 대한 Python 코드**

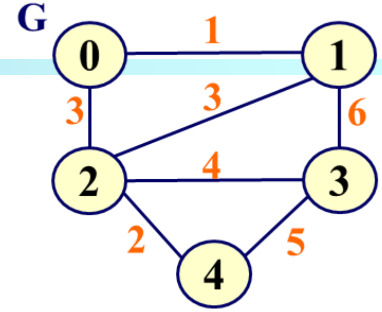
```python
1  import sys
2  N = 7
3  s = 0
4  g = [None for x in range(N)]
5  g[0] = [(1, 9), (2, 10)]
6  g[1] = [(0, 9), (3, 10), (4, 5), (6, 3)]
7  g[2] = [(0, 10), (3, 9), (4, 7), (5, 2)]
8  g[3] = [(1, 10), (2, 9), (5, 4), (6, 8)]
9  g[4] = [(1, 5),  (2, 7), (6, 1)]
10 g[5] = [(2, 2),  (3, 4), (6, 6)]
11 g[6] = [(1, 3),  (3, 8), (4, 1), (5, 6)]
12
13
14 visited = [False for x in range(N)]
15 dist = [sys.maxsize for x in range(N)]
16 dist[s] = 0
17 previous = [None for x in range(N)]
18 previous[s] = s
```

계속

```python
20 for v in range(N):
21     u = -1
22     mindist = sys.maxsize
23     for i in range(N):
24         if not visited[i] and dist[i] < mindist:
25             mindist = dist[i]
26             u = i
27     visited[u] = True
28     for w, wt in g[u]:
29         if not visited[w]:
30             if wt < dist[w]:
31                 dist[w] = wt
32                 previous[w] = u
33
34 print('최소신장트리: ', end='')
35 mst_cost = 0
36 for i in range(1,N):
37     print('(%d, %d)'% (i, previous[i]), end='')
38     mst_cost += dist[i]
39 print('\n최소신장트리 가중치: ', mst_cost)
```

- **라인 1:** 무한대 값을 표현하기 위해 sys 모듈 import
- **라인 2-3:** 그래프 정점의 수 N을 7로 가정하고 시작 정점 s를 0이라고 가정
- **라인 5-11:** 인접리스트를 이용하여 그래프 구성
- **라인 13-17:** Prim 알고리즘을 구현하기 위한 Python 리스트 visited, dist, previous 초기화
  - 여기서 리스트 previous는 MST의 간선을 생성하기 위해 사용(예를 들어 previous[i] = j는 간선 (i, j) 혹은 간선 (j, i)를 의미함)

최소 신장 트리 – Prim 알고리즘 (7/7)



❑ Prim 알고리즘 contd.

- **Prim 알고리즘에 대한 Python 코드 contd.**

```
 1  import sys
 2  N = 7
 3  s = 0
 4  g = [None for x in range(N)]
 5  g[0] = [(1, 9), (2, 10)]
 6  g[1] = [(0, 9), (3, 10), (4, 5), (6, 3)]
 7  g[2] = [(0, 10), (3, 9), (4, 7), (5, 2)]
 8  g[3] = [(1, 10), (2, 9), (5, 4), (6, 8)]
 9  g[4] = [(1, 5),  (2, 7), (6, 1)]
10  g[5] = [(2, 2),  (3, 4), (6, 6)]
11  g[6] = [(1, 3),  (3, 8), (4, 1), (5, 6)]
12
13
14  visited = [False for x in range(N)]
15  dist = [sys.maxsize for x in range(N)]
16  dist[s] = 0
17  previous = [None for x in range(N)]
18  previous[s] = s
```

계속

```
20  for v in range(N):
21      u = -1
22      mindist = sys.maxsize
23      for i in range(N):
24          if not visited[i] and dist[i] < mindist:
25              mindist = dist[i]
26              u = i
27      visited[u] = True
28      for w, wt in g[u]:
29          if not visited[w]:
30              if wt < dist[w]:
31                  dist[w] = wt
32                  previous[w] = u
33
34  print('최소신장트리: ', end='')
35  mst_cost = 0
36  for i in range(1,N):
37      print('(%d, %d)'% (i, previous[i]), end='')
38      mst_cost += dist[i]
39  print('\n최소신장트리 가중치: ', mst_cost)
```

- **라인 20-26:** 방문하지 않은 정점들에 대한 dist의 원소들 중에서 최솟값을 가진 정점 u을 찾음(N 개의 정점을 방문할 때까지 반복 후 종료)
- **라인 28-32:** u에 인접하면서 트리에 속하지 않은 정점 w에 대하여 필요 시 dist의 원소 갱신 후, (만약 갱신 했다면) w에 대한 dist의 원소 갱신이 u로 인해 이루어졌음을 기록하기 위해 u를 w의 previous로 설정(즉, 간선 (u, w) 생성)

수행시간 분석

입력 사이즈(그래프 정점의 수)를 N이라고 한다면, 라인 20의 주 반복문에 의해 N 번의 반복을 통해 (1) 정점 u를 찾고, (2) u와 인접하면서 트리에 속하지 않은 각 정점 w에 해당하는 dist의 원소 값을 갱신함

- 라인 20의 주 반복문: N 번 반복
- 라인 23과 라인 28의 내부 반복문: 2n 번 반복

따라서 O(n²)