Minimum Spanning Tree (MST)

MST의 정의, 찾는 방법, 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. MST는 무엇이며, 어떻게 활용되는가?
03. MST의 성질 + Greedy 방법 개요
04. Kruskal's Algorithm
05. Prim's Algorithm Lazy Version
06. Prim's Algorithm Eager Version
07. 실습: Prim's Algorithm Eager Version 구현

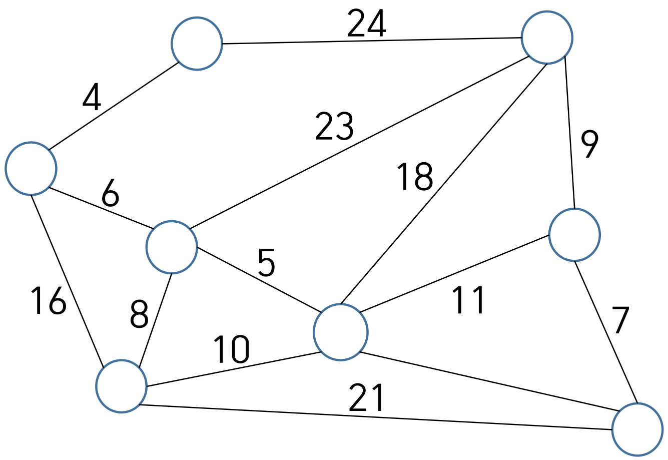
Greedy MST
Kruscal
Prim

(초점 1) 이 방법들이 왜 항상 MST를 잘 찾아내는가?
(초점 2) 우리가 배운 자료구조를 어떻게 잘 활용해 이들을 효율적으로 구현하는가?

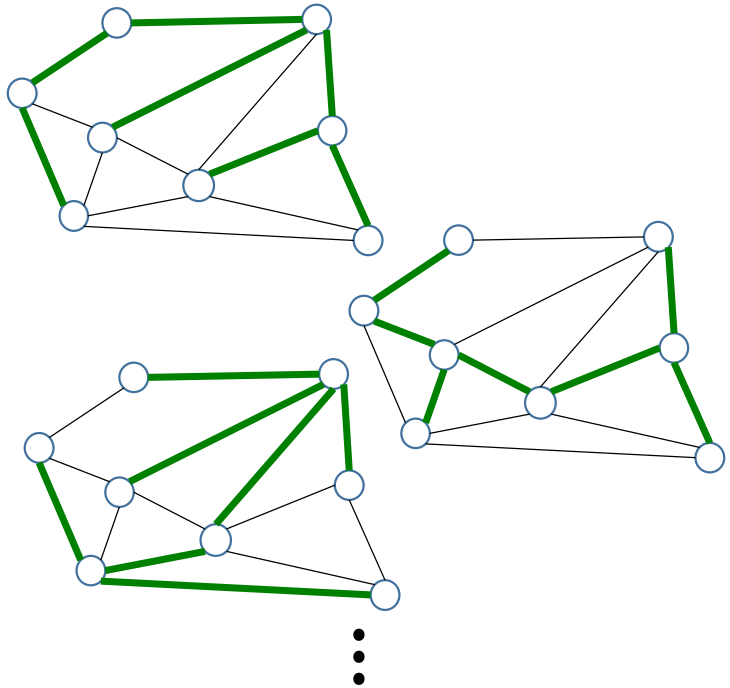
Greedy
Kruscal
Prim

Minimum Spanning Tree (MST): Weight 합 최소인 Spanning Tree

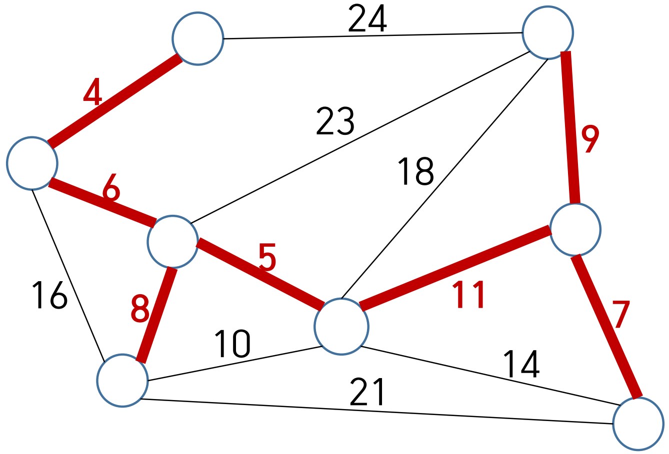
- <입력>
- 연결되었으며 (connected)
- edge weight 있는 (간선에 비용 有)
  weight가 다 달라야 해
- undirected graph G



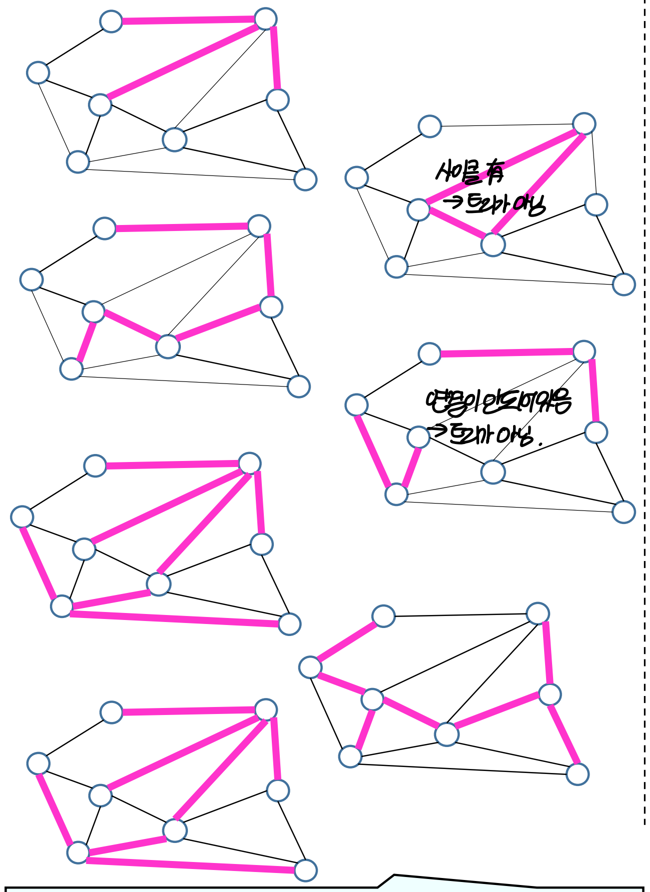
- Spanning Tree:
- G의 subgraph 중
- Tree 이며 (connected & acyclic)
  비순환
- Spanning (모든 정점 포함)



- <출력>
- Minimum Spanning Tree (MST):
- Spanning Tree 중 weight 합 최소인 Tree → 부분집합



① Tree (연결되어있다 & 사이클없다)
(~~connected & acyclic~~
subgraph of G)

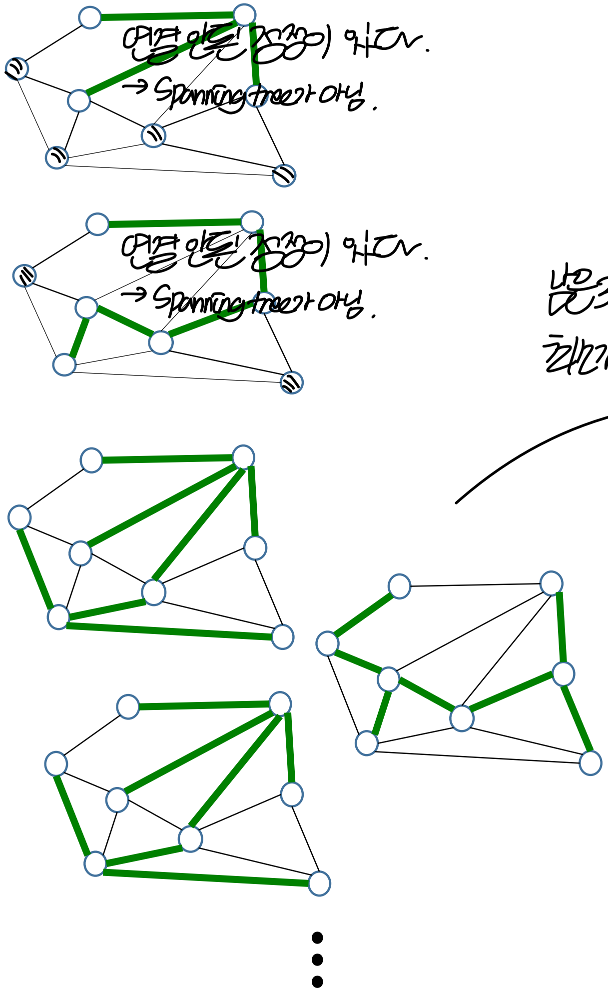


connected: subgraph 전체가 연결
(하나의 connected component)

acyclic: cycle 없음

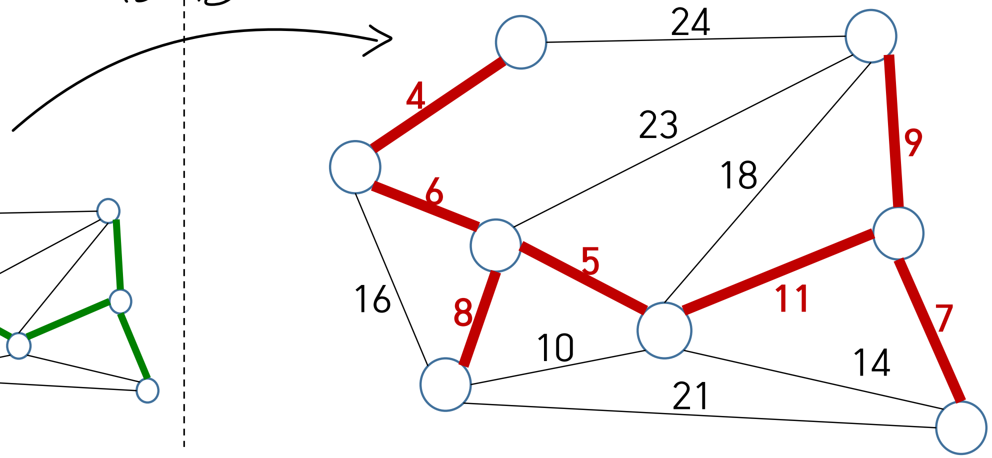
→ 그래프 상 정점을 하나도 빠짐없이 연결하는 트리

② Spanning Tree
(모든 정점 포함)



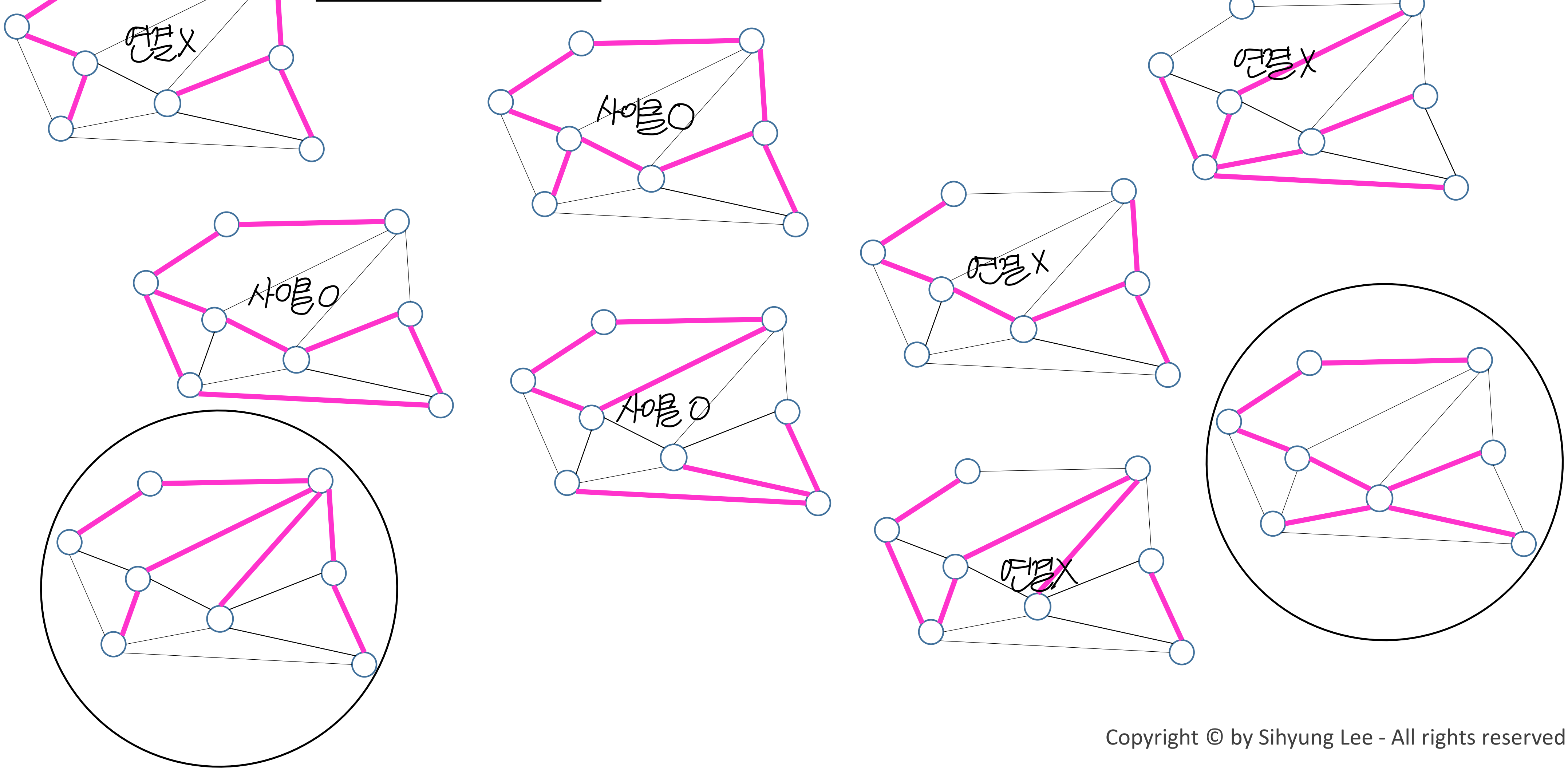
③ Minimum Spanning Tree (MST)
(간선의 weight 합 최소)

남은 3개 중 선택한 간선의 weight의 합이 최소가 되는 트리



[Q] 다음 중 그래프 G의 spanning tree인 것을 모두 고르시오.

G: 검은 실선으로 표기된 입력 그래프



Brute-force 알고리즘: 모든 가능한 spanning tree 탐색하며 weight 비교

완전탐색 알고리즘.
가능한 모든 경우의 수를 모두 탐색하면서 요구조건에 충족되는 결과만을 가져온다.

- V, E 커질수록 너무 많은 spanning tree 있어 시간 오래 걸림 → 앞의 방법은 비효율적

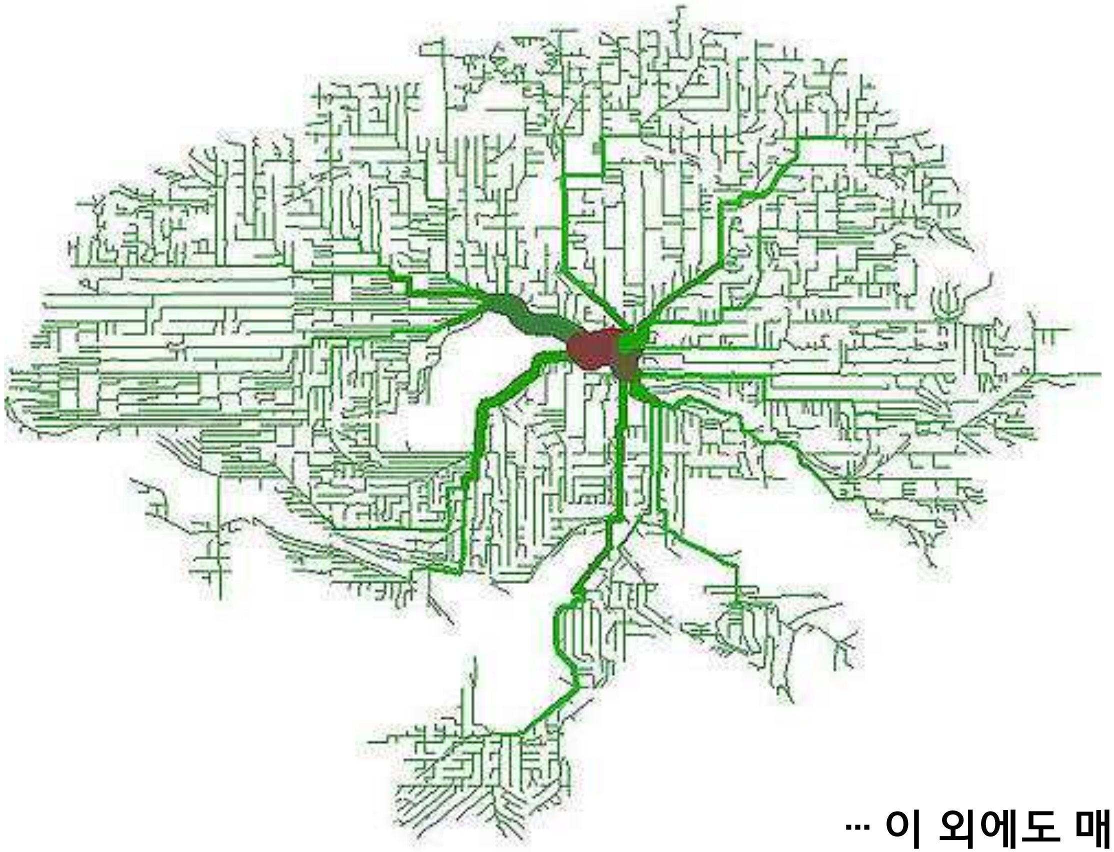
- 더 효율적인 방법 필요
  - Greedy Algorithm
  - Kruskal's Algorithm
  - Prim's Algorithm

MST 활용 예: 연결 자원 가능한 적게 쓰며 모든 지점 연결되도록 할 때 사용

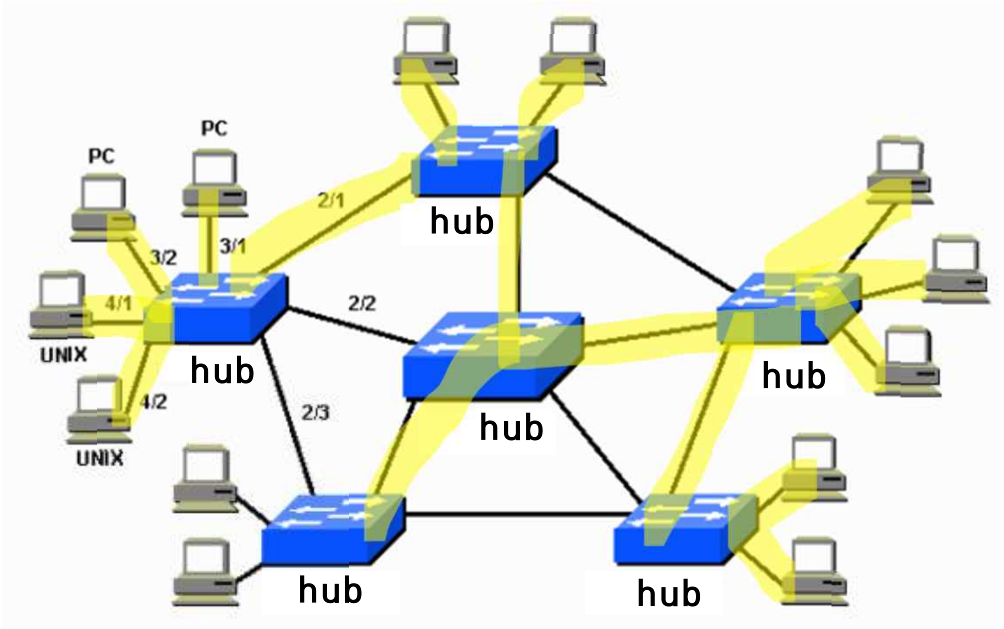
간선 합 최소 (minimum)

spanning

<North Seattle의 자전거 도로>



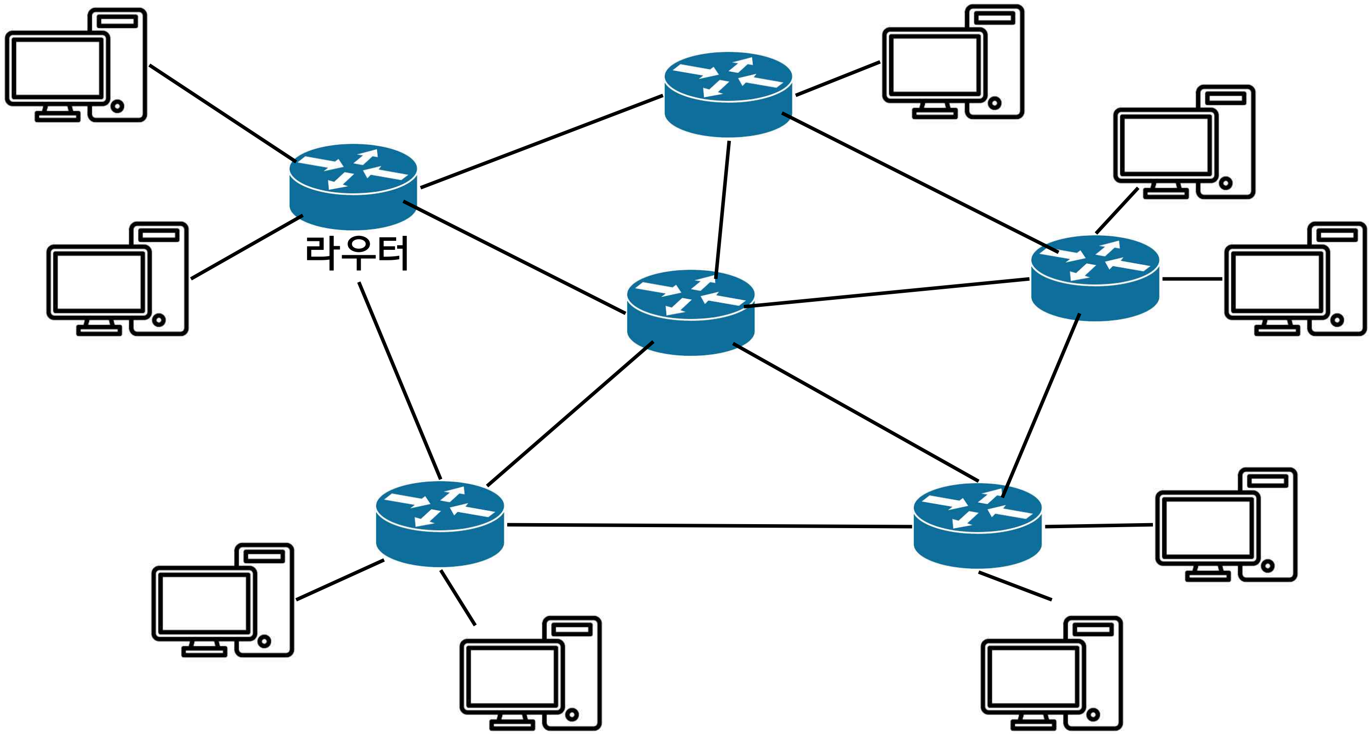
<컴퓨터망의 2계층 연결>
Broadcasting 시 모두에게 데이터 전달하되
무한 loop 생기지 않도록 spanning tree 구성



… 이 외에도 매우 많음 …

MST 활용 예: 연결 자원 가능한 적게 쓰며 모든 지점 연결되도록 할 때 사용

<컴퓨터망에서 Multicast or Broadcast>
여러 receiver에게 같은 데이터를 전송할 때 (예: streaming)
spanning tree 형태로 전송하면 같은 copy가 불필요하게 여러 번 재전송되거나 loop을 도는 것 방지할 수 있음



그 외 MST의 중요성

- **주요 자료 구조** 함께 잘 활용하는 좋은 예
- Union Find (with Connected Component 저장 구조)
- Priority Queue (with Binary Heap)
- Indexed Priority Queue: PQ에 저장한 key 값 변경 가능. 이를 위해 각 key를 unique한 index와 함께 저장하고, key 값 변경 시 index 사용해 변경하고자 하는 key 지정

Minimum Spanning Tree (MST)

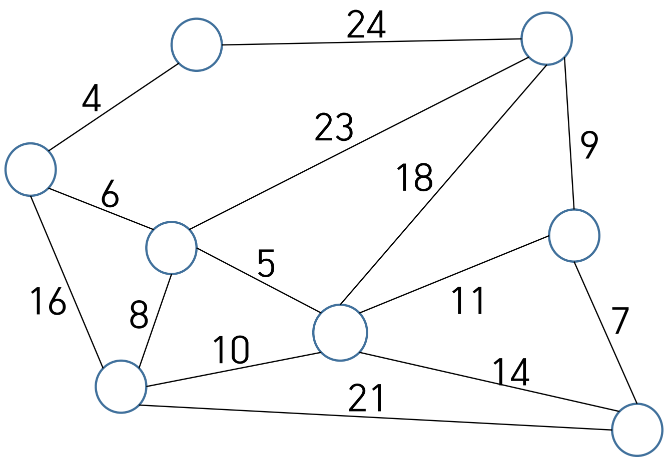
MST의 정의, 찾는 방법, 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. MST는 무엇이며, 어떻게 활용되는가?
03. MST의 성질 + Greedy 방법 개요
04. Kruskal's Algorithm
05. Prim's Algorithm Lazy Version
06. Prim's Algorithm Eager Version
07. 실습: Prim's Algorithm Eager Version 구현

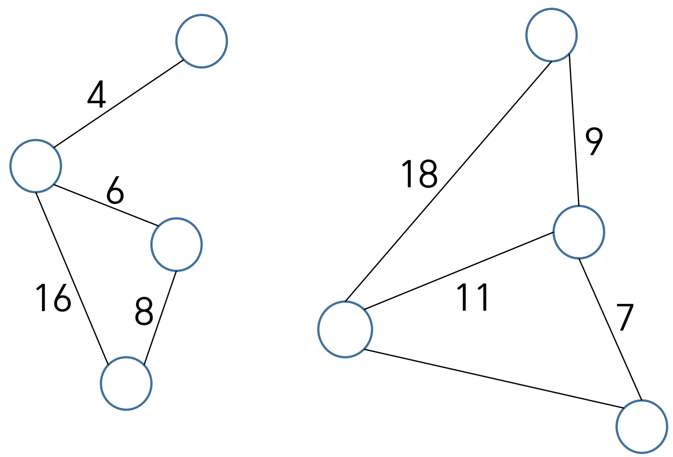
앞으로 볼 모든 MST 찾는 알고리즘에 공통 적용되는 기본 개념 & 성질 학습

[Q] MST의 입력 그래프 G는 왜 '연결'된 그래프여야 하나?

- **<입력>**
- **연결**되었으며 (connected)
- **edge weight** 있는 → 간선의 방향 ≠
- **undirected** graph G



<연결된 그래프>



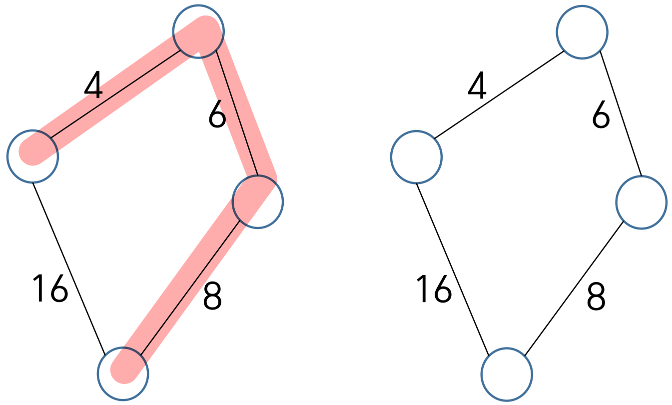
<연결 안 된 (비연결) 그래프>

간선 weight 모두 서로 다르다면
MST는 단 하나(유일)

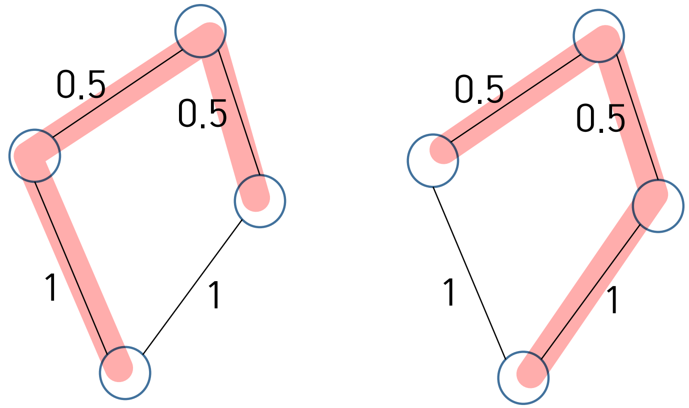
→답확인이 편리

Weight 같은 간선 있다면
MST 둘 이상 가능
(단 모든 MST의 weight sum은 같음)

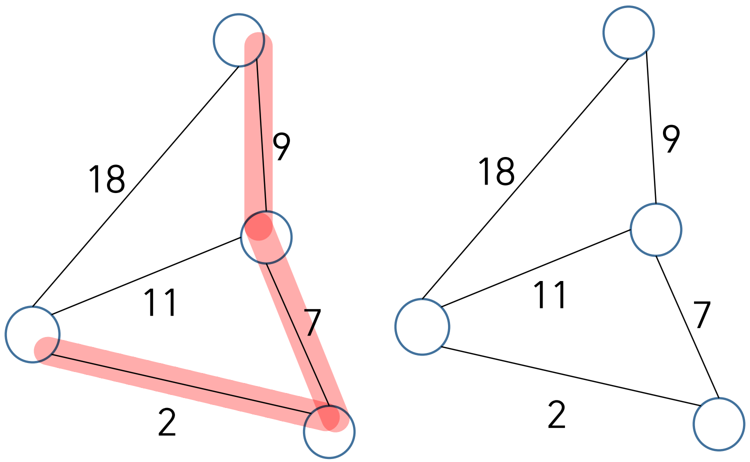
G1



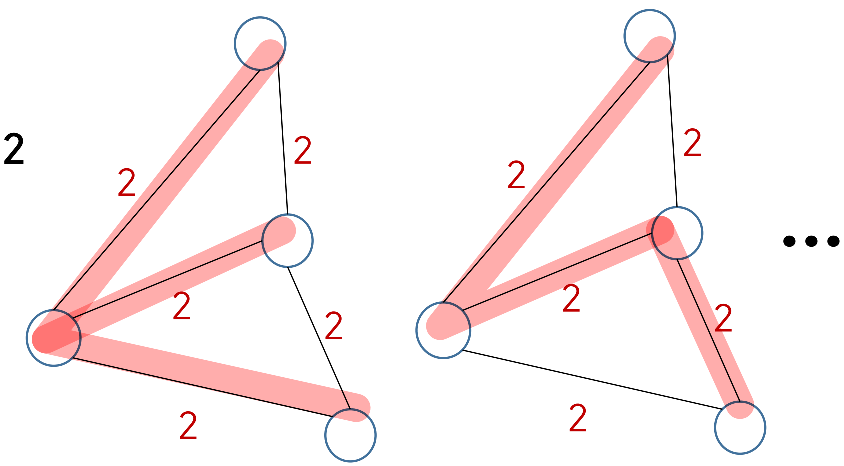
G11



G2

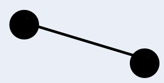
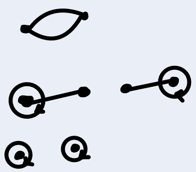
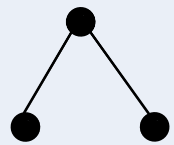
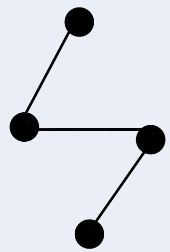
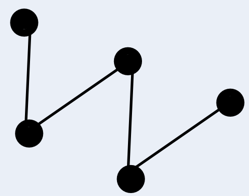


G22



[Q] 각 그래프에서 서로 다른 MST 다 찾아 보시오.

[Q] 입력 그래프 G에 V개 정점 있다면, Spanning Tree는 반드시 V-1개 간선 포함. Why?

V (G 상 정점 수)	1	2	3	4	5
간선 수	0	1	2	3	4
Spanning Tree		 정점개수=간선개수라면 사이클 生 → ∴ X 			

중요

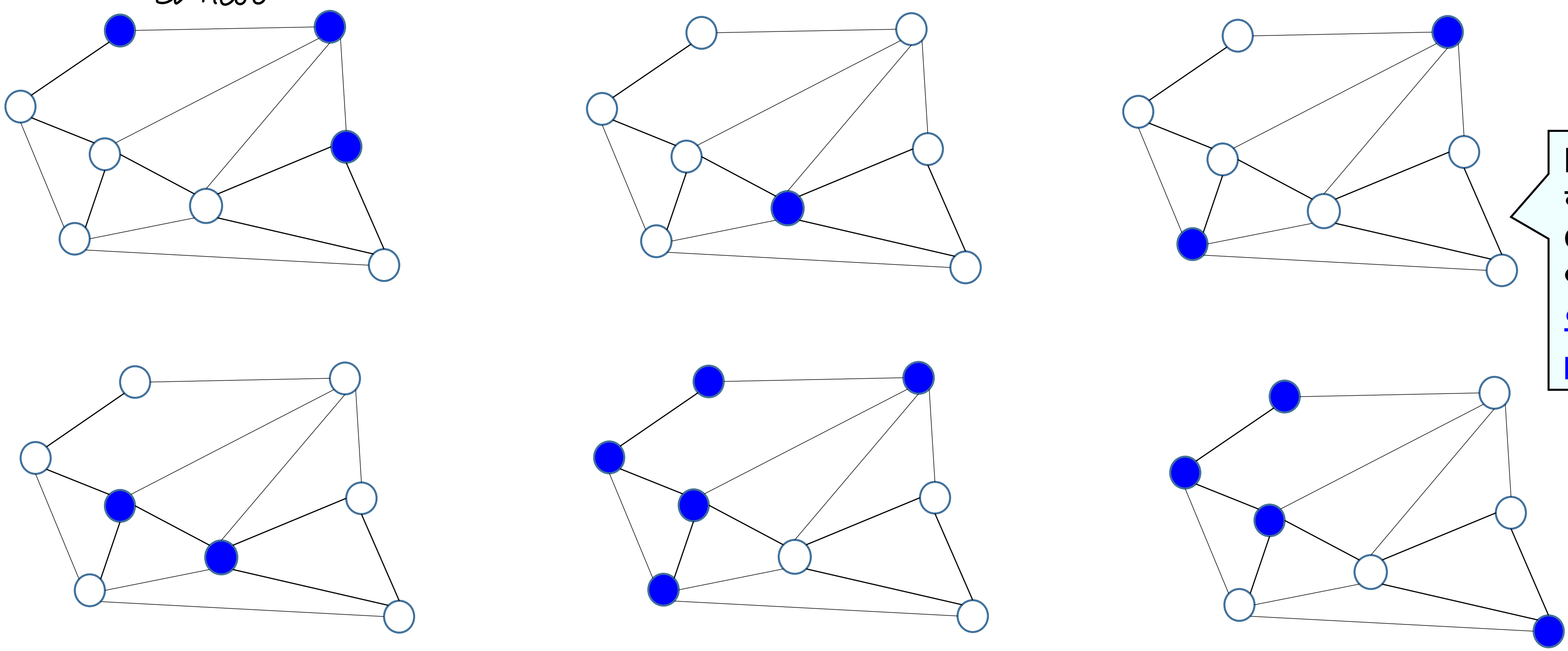
따라서 MST 찾으려면 연결 안 된 정점으로 간선 하나씩 더해가되,
V-1개 더했다면 멈추면 됨

Partition(cut) of graph G:
G의 정점 중 1 ~ (V-1)개 정점으로 이루어진 부분 집합

연결안된 부분을 정리하는 단어

- G의 $\boxed{\text{partition}}$의 예

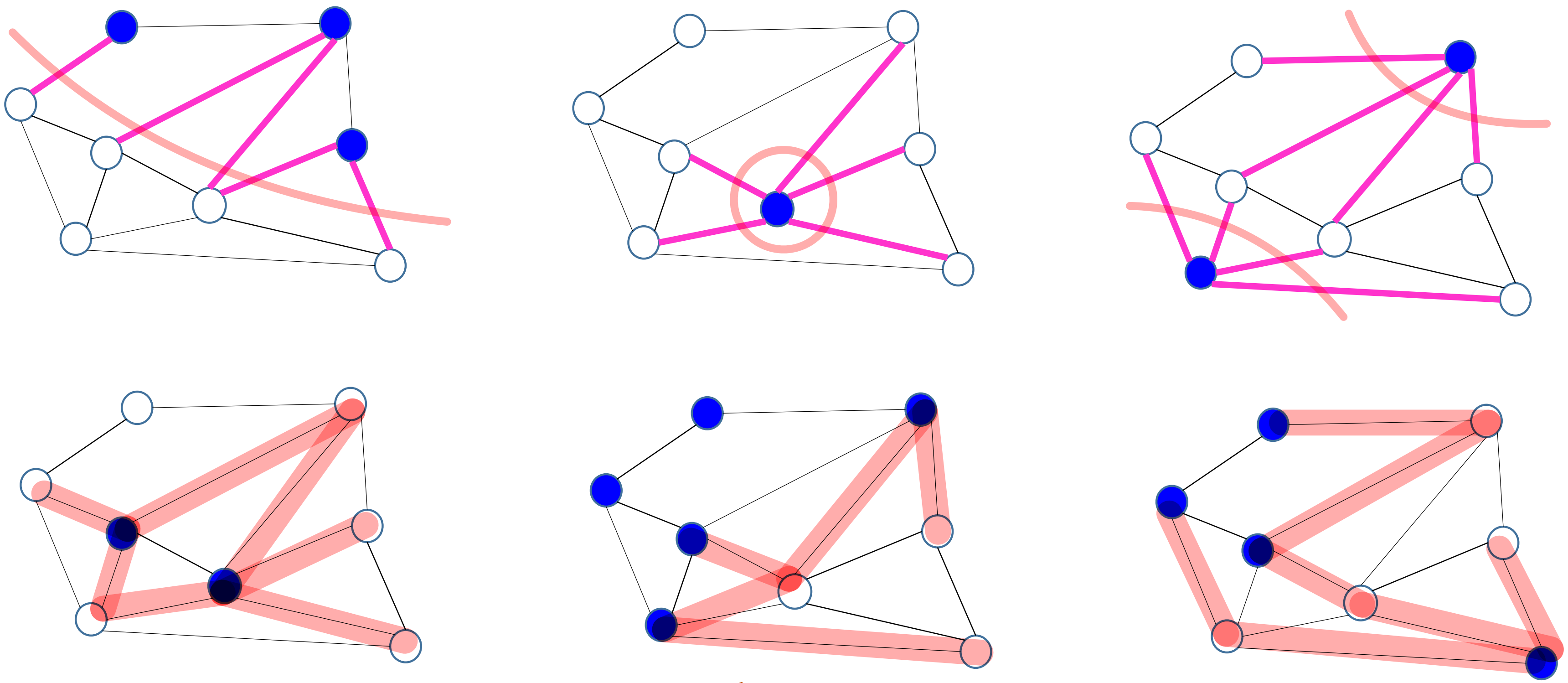
→ 정점의 부분집합



Partition은 인접한 정점으로만 이루어질 필요는 없으며 어떤 정점의 부분집합도 partition이 됨

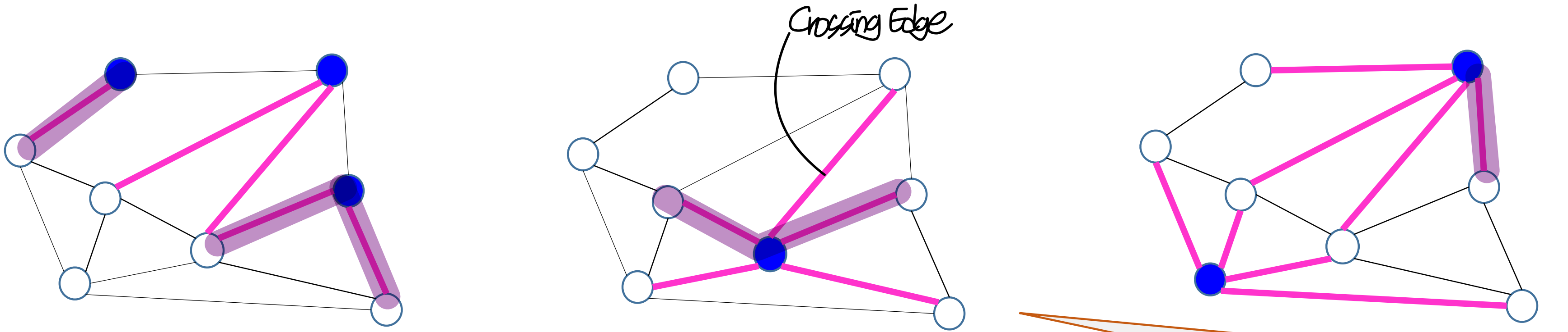
Crossing Edge: partition과 나머지 정점들 간 연결하는 간선

파란색이랑 흰색을 연결한 선

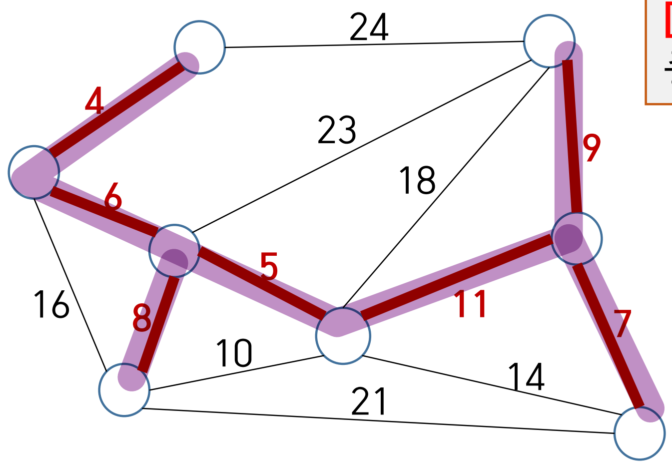


[Q] 아래 3가지 partition에 대한
crossing edge를 표시해 보시오.
Hint: 7개, 5개, 8개

어떤 partition에 대해서도 **최소 하나의 crossing edge는 반드시 MST에 포함**
Why? MST에서는 **모든 정점이 서로 연결**되어야 하므로

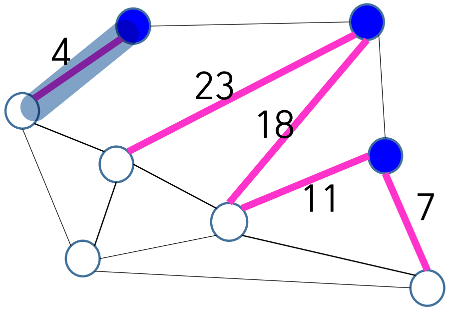


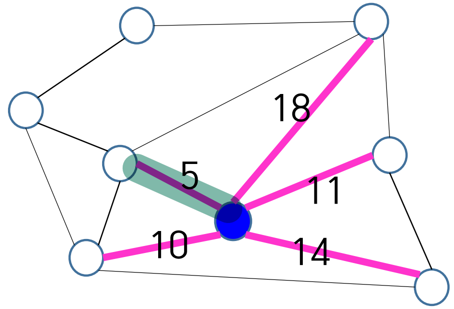
[Q] 어떤 partition에 대해서도 crossing edge 중 최소 하나는 MST에 포함됨 확인해 보시오.

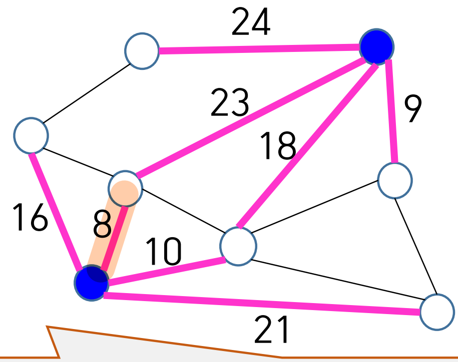


<위 그래프의 MST>

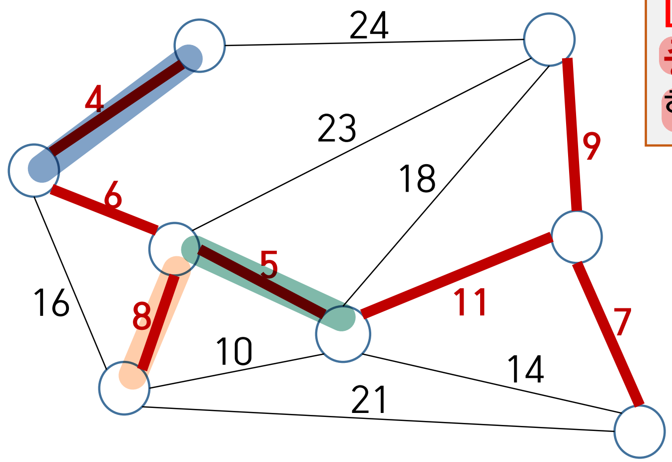
어떤 partition에 대해서도 weight 최소인 crossing edge는 반드시 MST에 포함
Why? "Minimum" Spanning Tree여야 하므로







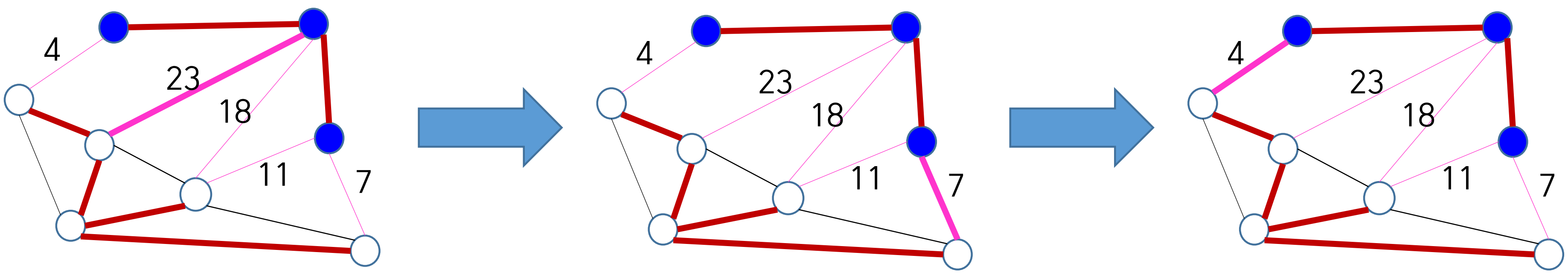
[Q] 어떤 partition에 대해서도 crossing edge 중 weight 가장 작은 간선은 반드시 MST에 포함됨 확인해 보시오.



<위 그래프의 MST>

어떤 partition에 대해서도 **weight 최소인 crossing edge는 반드시 MST에 포함**
Why? "**Minimum**" Spanning Tree여야 하므로

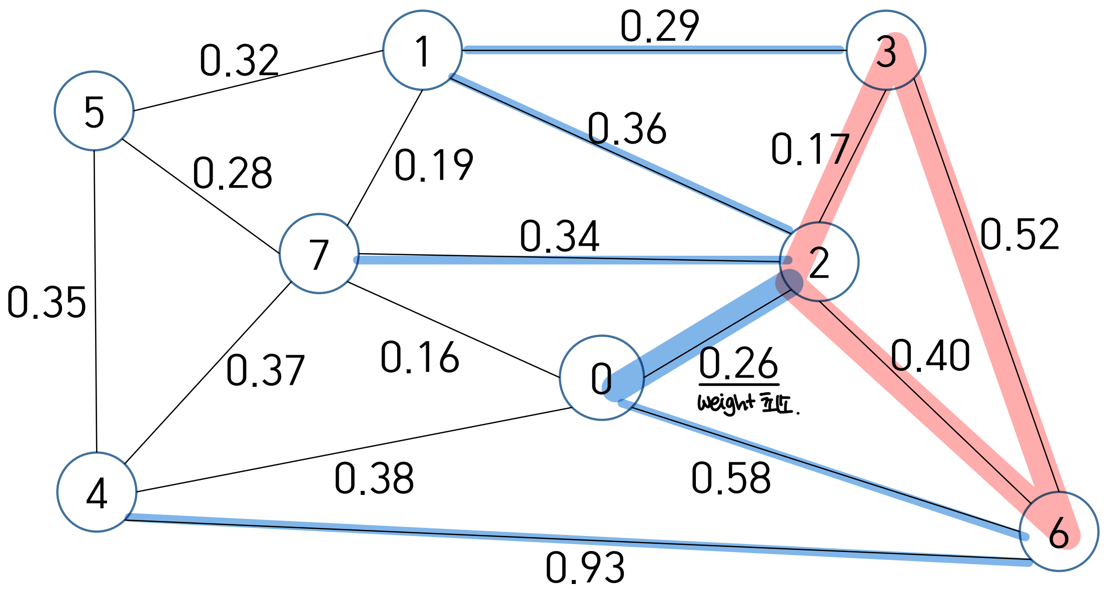
- 만약 ST에서 두 partition을 weight 최소 아닌 crossing edge e1으로 연결했다면
- e1을 weight 더 작은 crossing edge e2 (< e1)로 대체함으로써
- weight 합 더 작은 트리 만들 수 있음
- 따라서 MST라면 crossing edge 중 weight 최소인 edge를 반드시 사용할 것임



<weight 합 = 23 + k인 ST> **<weight 합 = 7 + k인 ST>** **<weight 합 = 4 + k인 ST>**

[Q] 아래 그래프에서 partition {2,3,6}을 생각해 보자.
Weight 최소인 crossing edge는 무엇인가?

- 0-7 (0.16)
- 2-3 (0.17)
- $\boxed{\text{0-2 (0.26)}}$
- 1-2 (0.36)

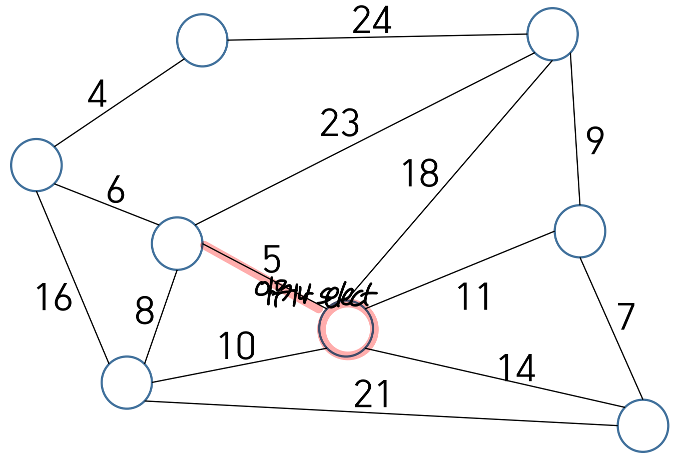


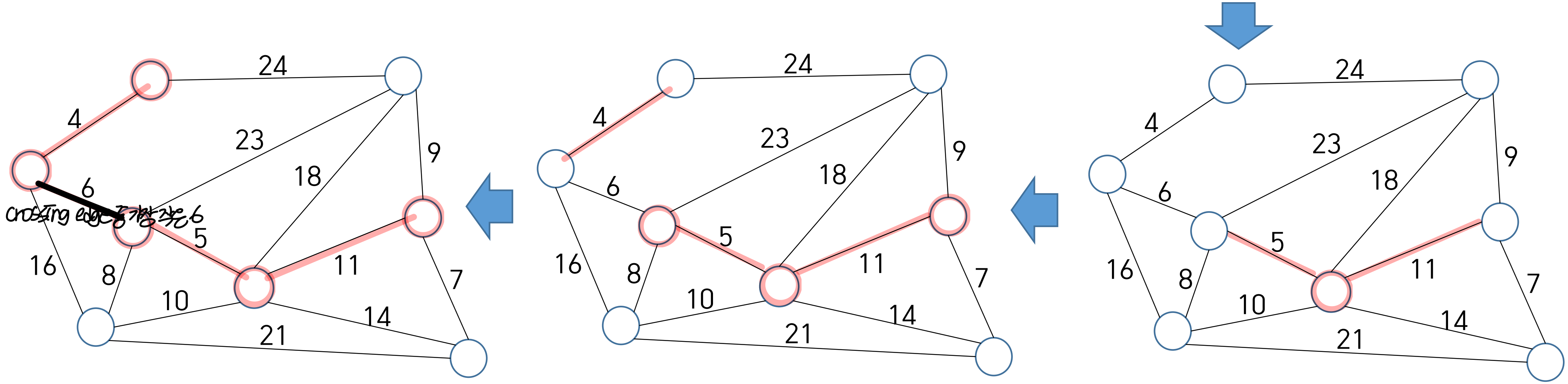
Greedy MST algorithm

← Kruscal/Prim 모두 동일

(지금 당장 최적인 선택지를 고르는 것)

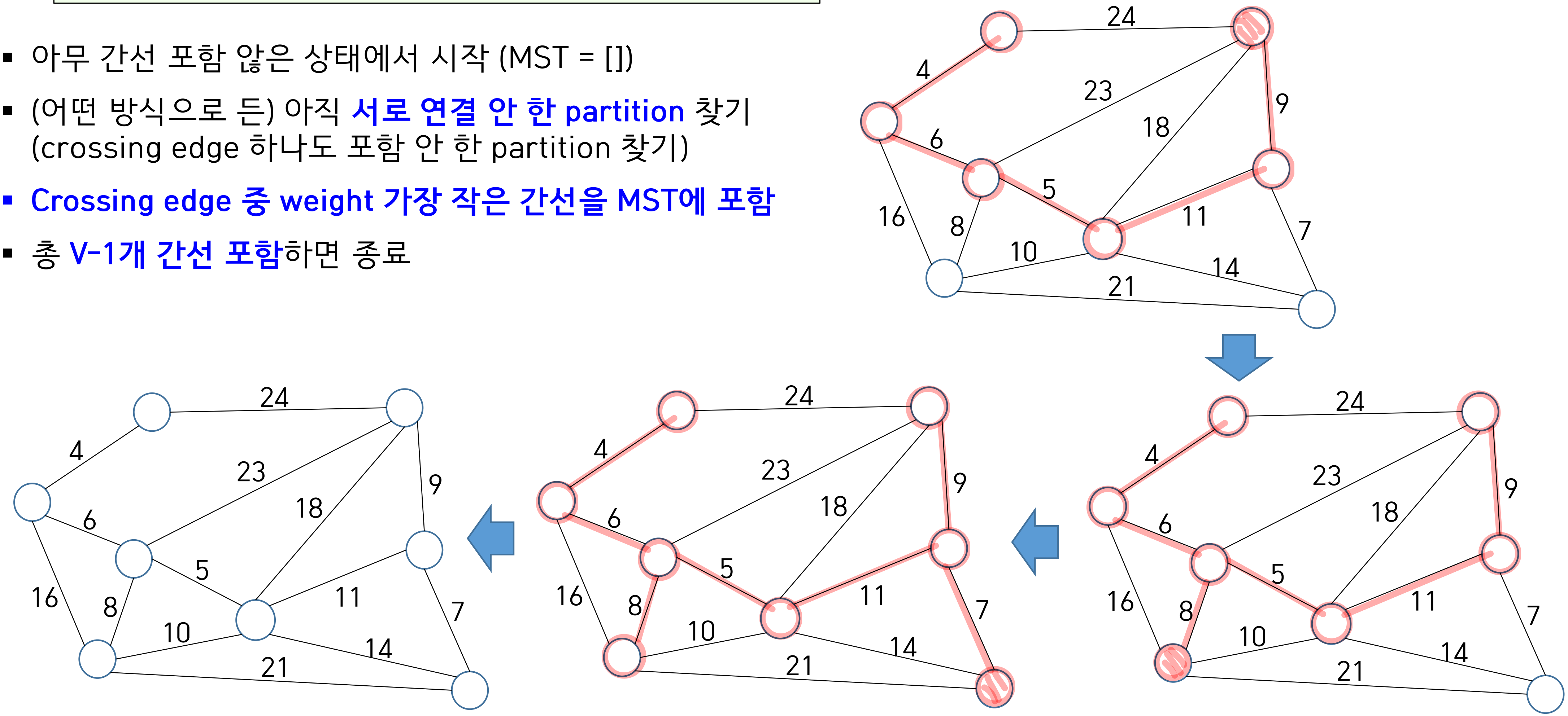
- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- (어떤 방식으로 든) 아직 **서로 연결 안 한** partition 찾기
  (crossing edge 하나도 포함 안 한 partition 찾기)
- **Crossing edge 중 weight 가장 작은 간선을 MST에 포함**
- 총 **V-1개 간선 포함**하면 종료





Greedy MST algorithm

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- (어떤 방식으로 든) 아직 **서로 연결 안 한 partition** 찾기 (crossing edge 하나도 포함 안 한 partition 찾기)
- **Crossing edge 중 weight 가장 작은 간선을 MST에 포함**
- 총 **V-1개 간선 포함**하면 종료

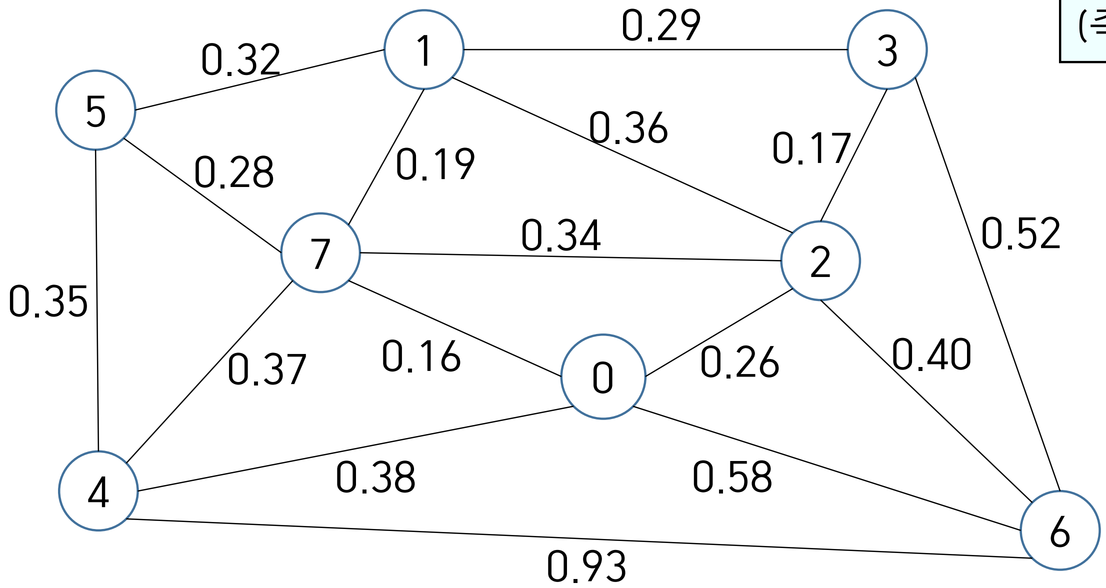


Greedy MST algorithm

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- (어떤 방식으로 든) 아직 **서로 연결 안 한 partition** 찾기 (crossing edge 하나도 포함 안 한 partition 찾기)
- **Crossing edge 중 weight 가장 작은 간선을 MST에 포함**
- 총 **V-1개 간선 포함**하면 종료

[Q] 아래 그래프에 Greedy MST algorithm 적용해 MST를 찾아보시오.

어떤 순서로 어떤 partition 선정했더라도 최종 결과는 같음 확인해 보자.
(즉 위 알고리즘은 올바름)



Greedy MST algorithm

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- (어떤 방식으로 든) 아직 **서로 연결 안 한 partition** 찾기 (crossing edge 하나도 포함 안 한 partition 찾기)
- **Crossing edge 중 weight 가장 작은 간선을 MST에 포함**
- 총 **V-1개 간선 포함**하면 종료

그래프 커지면 partition 수 매우 많아지며 이 중 연결 안 한 partition 잘 찾는 방법 필요

앞으로 볼 알고리즘은 이 부분이 다름

왜 잘 동작해야 하나?
(1) 반드시 MST에 포함되어야 하는 간선 포함해 가는 방식
(2) MST에는 총 V-1개 간선 있어야 하므로

Minimum Spanning Tree (MST)

MST의 정의, 찾는 방법, 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. MST는 무엇이며, 어떻게 활용되는가?
03. MST의 성질 + Greedy 방법 개요
04. Kruskal's Algorithm
05. Prim's Algorithm Lazy Version
06. Prim's Algorithm Eager Version
07. 실습: Prim's Algorithm Eager Version 구현

· 각각이 왜 Greedy에 포함되는가
어떻게 효율적으로 구현할까

Kruskal : 비용순

① → 모든 간선을 비용순으로 정렬

Kruskal's Algorithm

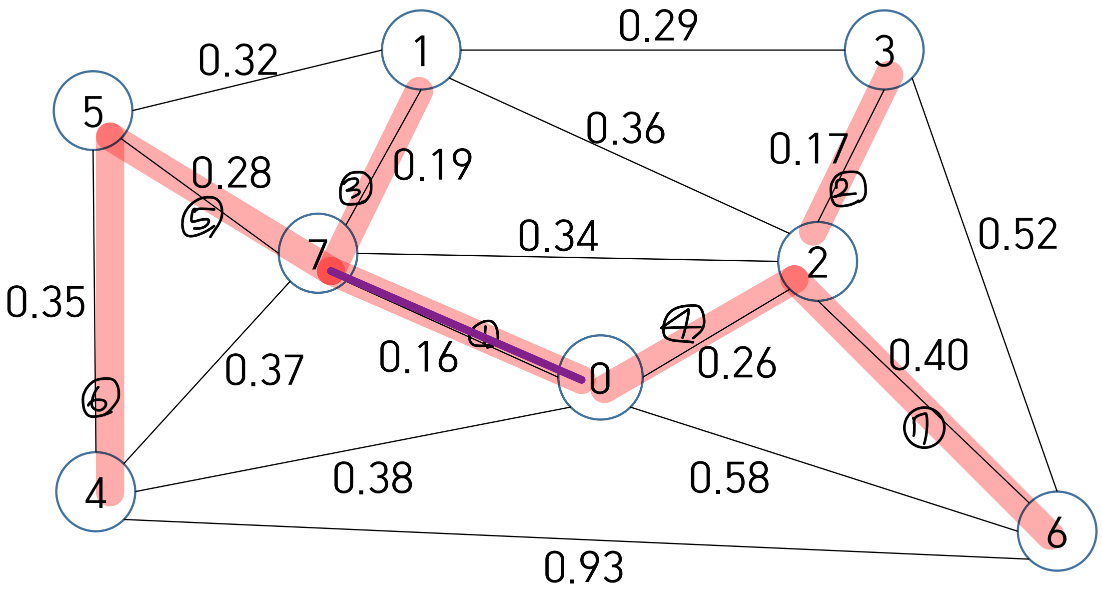
검은색: Greedy algorithm 그대로
푸른색: Crossing edge 추가할 partition 선정 방식만 특화한 부분

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- **간선을 weight의 오름차순에 따라 하나씩 검사**하며

② 간선을 비용순으로 추가하되..

- **추가했을 때 cycle 만들지 않는 간선이라면 MST에 추가**
- 총 V-1개 간선 포함하면 종료

(정점이 8개니까, 간선이 7개가 되면 종료)



<간선을 weight 오름차순으로 정렬한 결과>

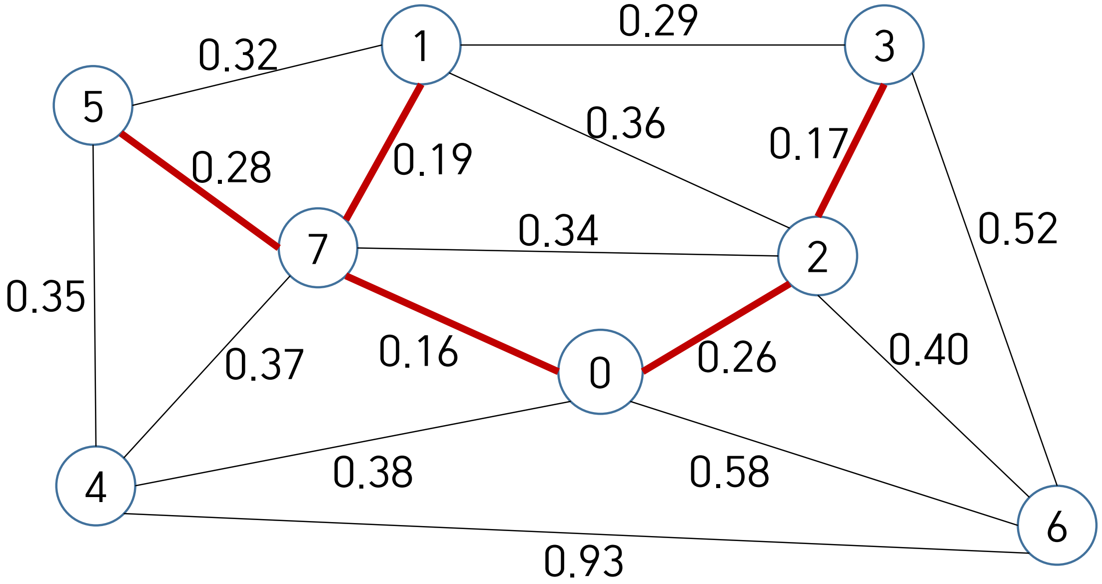
- 0-7 (0.16)
- 2-3 (0.17)
- 1-7 (0.19)
- 0-2 (0.26)
- 5-7 (0.28)
- ~~1-3 (0.29)~~ 사이클 생성
- ~~1-5 (0.32)~~
- ~~2-7 (0.34)~~
- 4-5 (0.35)
- ~~1-2 (0.36)~~
- ~~4-7 (0.37)~~
- ~~0-4 (0.38)~~
- 6-2 (0.40) 끝
- 3-6 (0.52)
- 6-0 (0.58)
- 6-4 (0.93)

Kruskal's Algorithm

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- **간선을 weight의 오름차순에 따라 하나씩 검사**하며
- **추가했을 때 cycle 만들지 않는 간선이라면 MST에 추가**
- 총 V-1개 간선 포함하면 종료

[Q] 왜 '**cycle 만들지 않는 간선**' 조건 필요한가?

↳ 끊어진 Partition을 발견



<간선을 weight 오름차순으로 $\boxed{\text{정렬}}$한 결과>

min PQ

- 0-7 (0.16)
- 2-3 (0.17)
- 1-7 (0.19)
- 0-2 (0.26)
- 5-7 (0.28)
- 1-3 (0.29)
- 1-5 (0.32)
- 2-7 (0.34)
- 4-5 (0.35)
- 1-2 (0.36)
- 4-7 (0.37)
- 0-4 (0.38)
- 6-2 (0.40)
- 3-6 (0.52)
- 6-0 (0.58)
- 6-4 (0.93)

왼쪽 상황에서
cycle 만드는 간선들

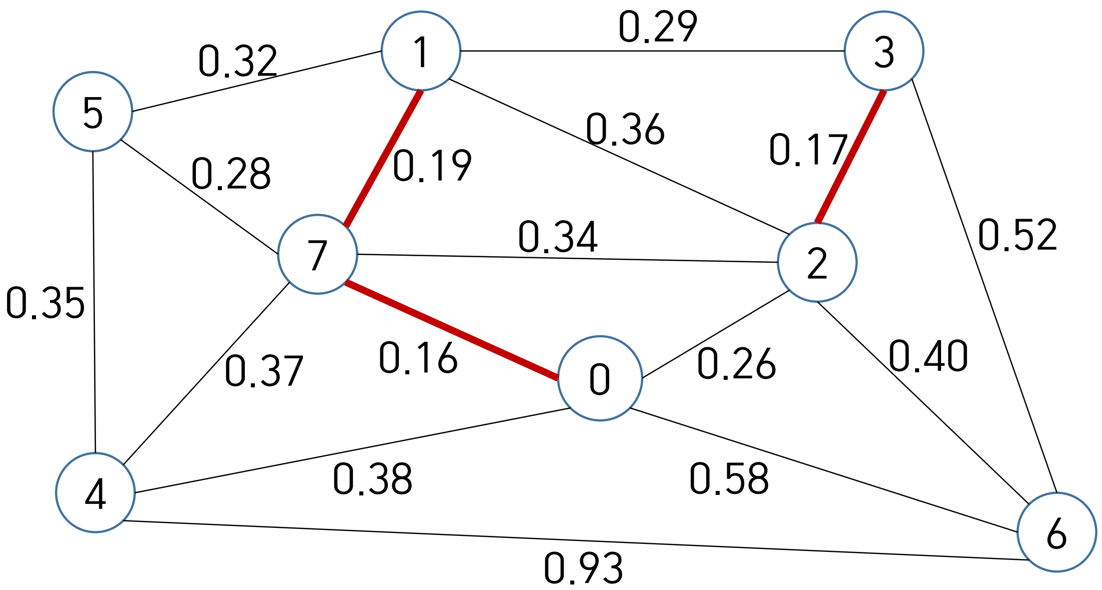
Kruskal's Algorithm ⊂ Greedy MST algorithm

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- **간선을 weight의 오름차순에 따라** 하나씩 검사하며
- **추가했을 때 cycle 만들지 않는 간선**이면 MST에 추가
- 총 V-1개 간선 포함하면 종료

[Q] Kruskal's Algorithm은 Greedy MST에 포함. 따라서 MST 만들어 냄. Why?

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- (어떤 방식으로 든) 아직 **서로 연결 안 한 partition** 찾기 (crossing edge 하나도 포함 안 한 partition 찾기)
- **Crossing edge 중 weight 가장 작은 간선** MST에 포함

총 V-1개 간선 포함하면 종료



<간선을 weight 오름차순으로 정렬한 결과>
- 0-7 (0.16)
- 2-3 (0.17)
- 1-7 (0.19)
- 0-2 (0.26)
- 5-7 (0.28)
- 1-3 (0.29)
- 1-5 (0.32)
- 2-7 (0.34)
- …

cycle 안 만듦. 따라서 {1,7,0} 포함 partition과 {2,3} 포함 partition은 아직 연결 안 됨

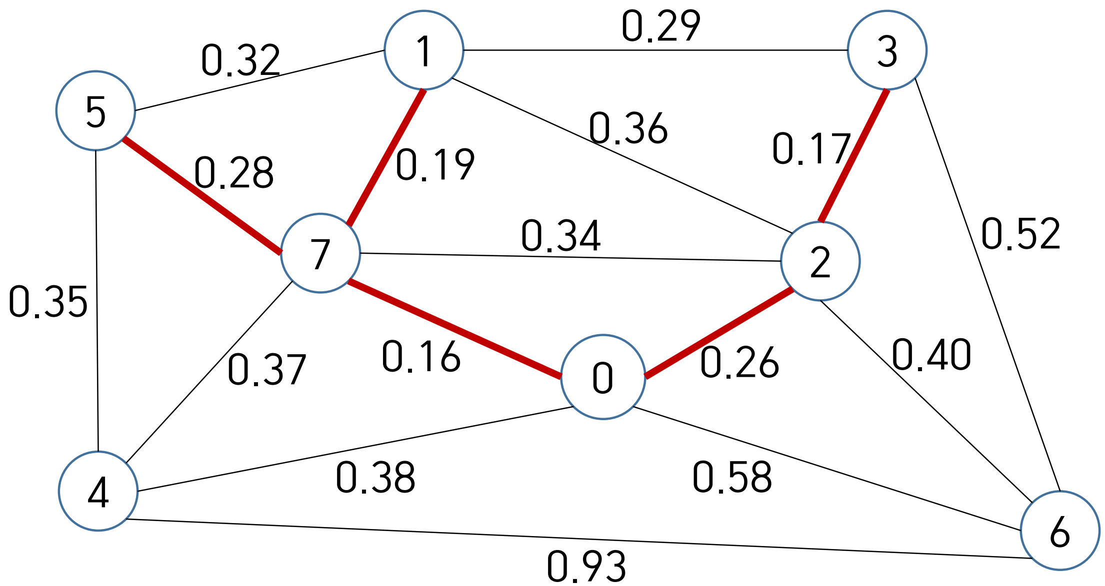
[Q] 'cycle 안 만드는 간선' 조건 어떻게 효율적으로 체크? [A1] DFS

①

[Q] 지금까지 MST에 포함한 정점이 V개, 간선이 E개라면 DFS로 cycle 탐지하는 데 걸리는 시간은? $\boxed{\sim V + E}$ $\leq V-1$

$E \sim V$

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- **간선을 weight의 오름차순에 따라 하나씩 검사**하며
- **추가했을 때 cycle 만들지 않는 간선이라면 MST에 추가**
- 총 V-1개 간선 포함하면 종료



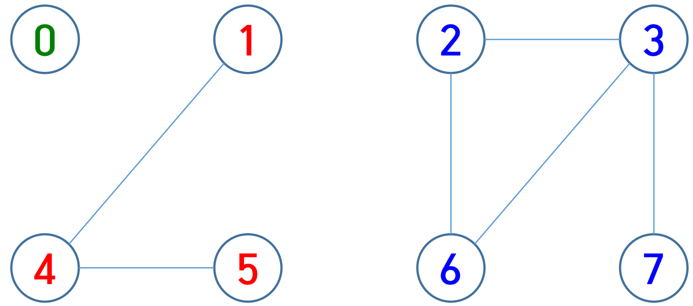
<간선을 weight 오름차순으로 정렬한 결과>
- 0-7 (0.16)
- 2-3 (0.17)
- 1-7 (0.19)
- 0-2 (0.26)
- 5-7 (0.28)
- 1-3 (0.29)
- 1-5 (0.32)
- 2-7 (0.34)
- …

②

Union Find (연결 상태 **변경** & **확인**)

- N개 정점 주어짐
  - 0 ~ (N-1) 까지 정점(vertex)으로 표현
  - 간선(edge) 없는 상태에서 시작
- 2개 명령 수행 필요
  - Union(a, b): 점 a와 b를 간선으로 연결
  - Connected(a, b): a와 b 연결하는 경로 존재하는지 True/False로 응답 (이를 Find 명령이라고도 함)

- 예제(N=10)



index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

각 정점 속한 component ID 저장

ids[a] == ids[b] 이면 a, b는 연결됨

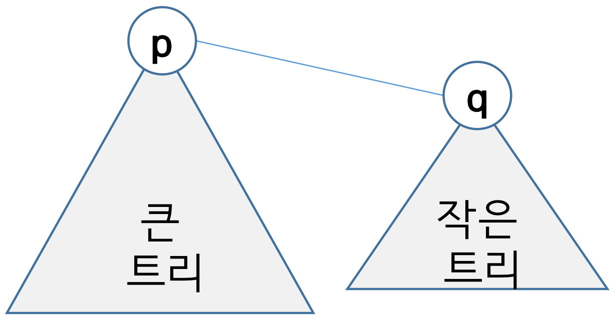
[Q] 간선 1-5 추가하면 cycle 생기는지 알고 싶다. 어떤 명령 쓰면 될까? Connected

WQU (Weighted Quick Union)의 비용

- 그래프 정점 수 V일 때
- union: $\log_2 V$
- connected: $\log_2 V$

- union(p,q): 작은 트리의 root를 큰 트리의 root 아래 연결
- 따라서 root 찾는 비용 $\log_2 V$가 union 비용

- connected(p,q): p,q의 root 비교
- 따라서 root 찾는 비용 $\log_2 V$가 connected 비용



‘cycle 안 만드는 간선' 조건 확인 방식: DFS (~V) vs. UF (~$\log_2 V$)

그래프가 고정된 상태에서 단 한 번 확인

그래프가 계속 변화하는 상태에서 여러 번 확인

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- **간선을 weight의 오름차순에 따라 하나씩 검사**하며
- **추가했을 때 cycle 만들지 않는 간선이라면 MST에 추가**
- 총 V-1개 간선 포함하면 종료

Kruskal's Algorithm 코드: UF + minPQ (혹은 list 정렬해도 됨)


```
def mstKruskal(g): # Constructor: finds an MST and stores it

    edgesInMST = [] # 지금까지 MST에 포함한 간선 저장
    weightSum = 0 # MST에 포함한 간선의 weight 합

    pq = PriorityQueue()
    for e in g.edges:
        pq.put(e)          모든 간선을 PQ에 넣어두기

    uf = UF(g.V)
    while not pq.empty() and len(edgesInMST) < g.V-1:
        e = pq.get()
        if not uf.connected(e.v, e.w): 사이클이 없다면
            uf.union(e.v, e.w)
            edgesInMST.append(e) 추가한다.
            weightSum += e.weight

    return edgesInMST, weightSum
```

<Kruskal's Algorithm>

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- 간선을 **weight의 오름차순에 따라 하나씩 검사**하며
- **추가했을 때 cycle 만들지 않는 간선이라면** MST에 추가
- 총 **V-1개 간선 포함하면 종료**

Kruskal's Algorithm의 비용


```
def mstKruskal(g): # Constructor: finds an MST and stores it

    edgesInMST = [] # 지금까지 MST에 포함한 간선 저장
    weightSum = 0 # MST에 포함한 간선의 weight 합

    pq = PriorityQueue()
    for e in g.edges:
        pq.put(e)

    uf = UF(g.V)
    while not pq.empty() and len(edgesInMST) < g.V-1:
        e = pq.get()
        if not uf.connected(e.v, e.w):
            uf.union(e.v, e.w)
            edgesInMST.append(e)
            weightSum += e.weight

    return edgesInMST, weightSum
```

[Q] Highlight한 부분 각각의 비용 생각해 보자.
그래프 g의 정점 수 V, 간선 수 E라 가정

↳ V << E임을 가정

UF+PQ

Operation	1회 비용	필요한 횟수
PQ, insert (heap (complete tree))	$\log \boxed{E}$ (간선들 넣으니까) ×	E
PQ, delete min	$\log E$ ×	E
UF, union (통과할 때만 (∵ V에 비례))	$\log \boxed{V}$ (정점개수가 원소개수) ×	V
UF, connected (꺼낼때마다 connected)	$\log V$ ×	E

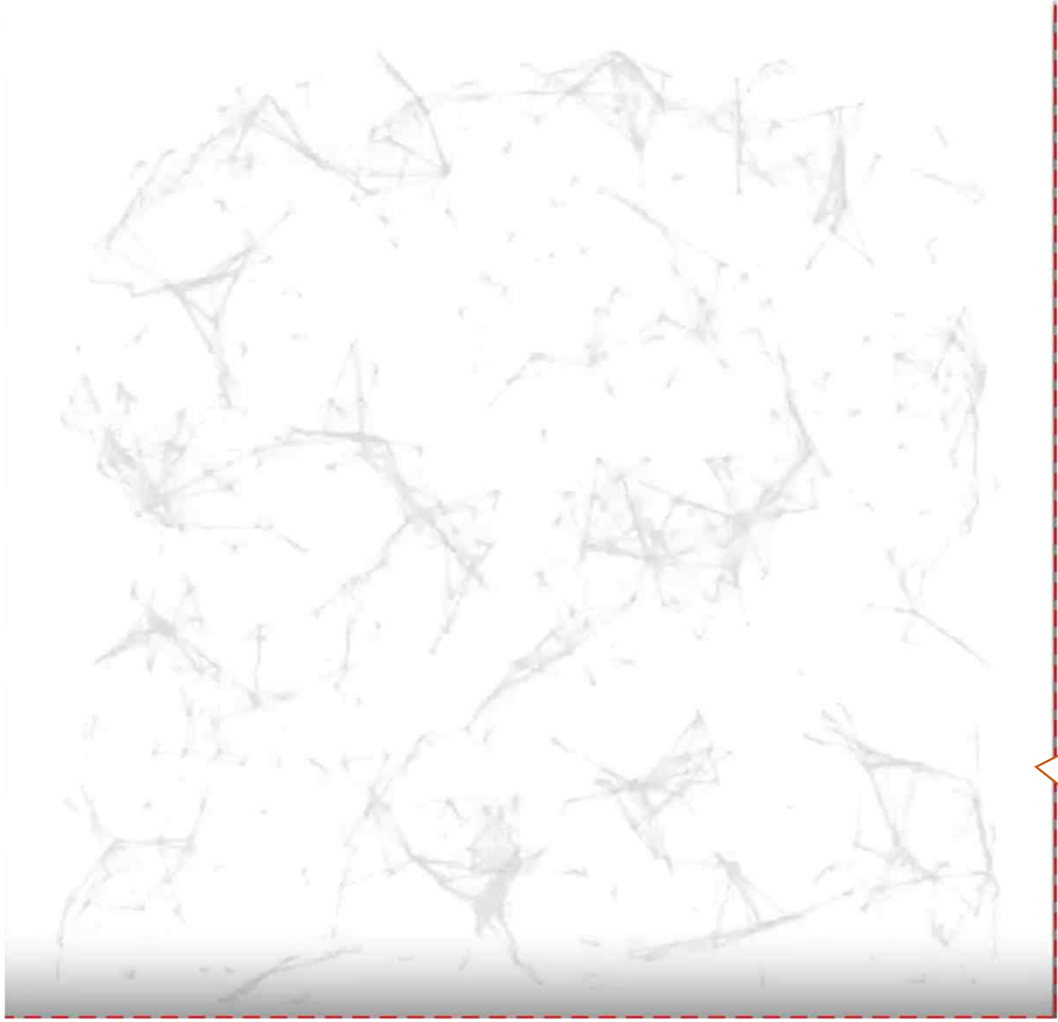
$\sim E\log E$

if DFS) $\sim EV$

[Q] 다음 중 그래프 G의 MST를 (그대로) 구하는 경우는?
(그래프의 edge weight > 0 이라 가정) 그래프 변경 전/후가 똑같이 나오는 경우

→ 대소관계는 계속 = → ∴ 답은 똑같이 나옴.

- G의 모든 edge weight에 17 더한 후 Kruskal 알고리즘 사용
- G의 모든 edge weight에 17 곱한 후 Kruskal 알고리즘 사용
- G의 모든 edge weight을 제곱한 후 Kruskal 알고리즘 사용
- ✓ 위 3가지 경우 모두



기계학습 등에서 속성 유사한 원소끼리 (그래프에서 서로 가까운 원소) 묶어주어야 할 때 (**clustering**) 원하는 cluster 개수 될 때까지 Kruskal 알고리즘 사용하기도 함

Minimum Spanning Tree (MST)

MST의 정의, 찾는 방법, 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)

02. MST는 무엇이며, 어떻게 활용되는가?

03. MST의 성질 + Greedy 방법 개요

04. Kruskal's Algorithm

05. Prim's Algorithm Lazy Version

06. Prim's Algorithm Eager Version

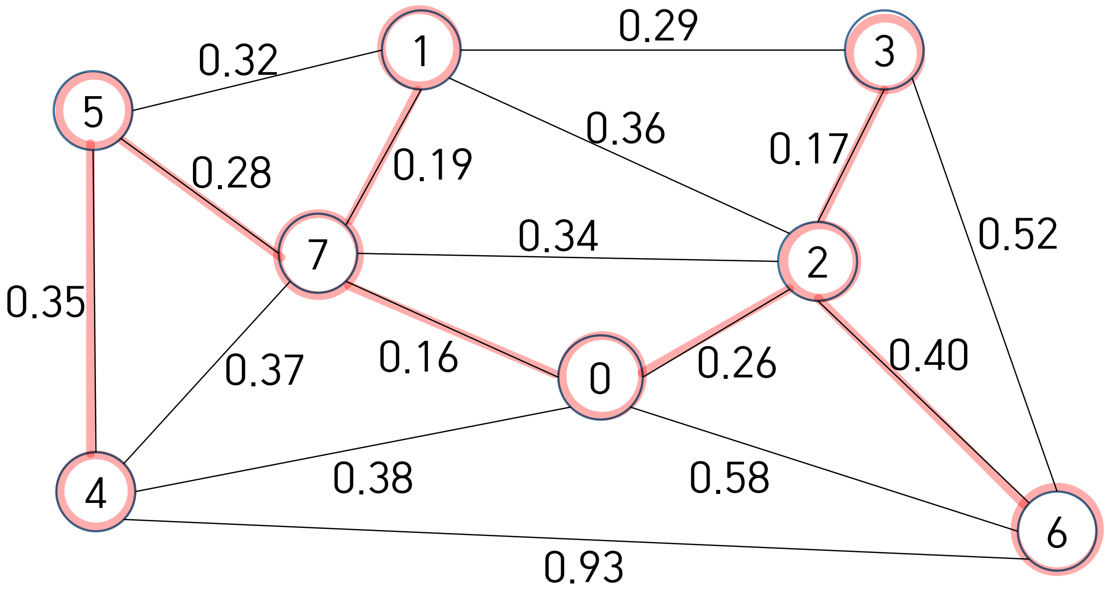
07. 실습: Prim's Algorithm Eager Version 구현

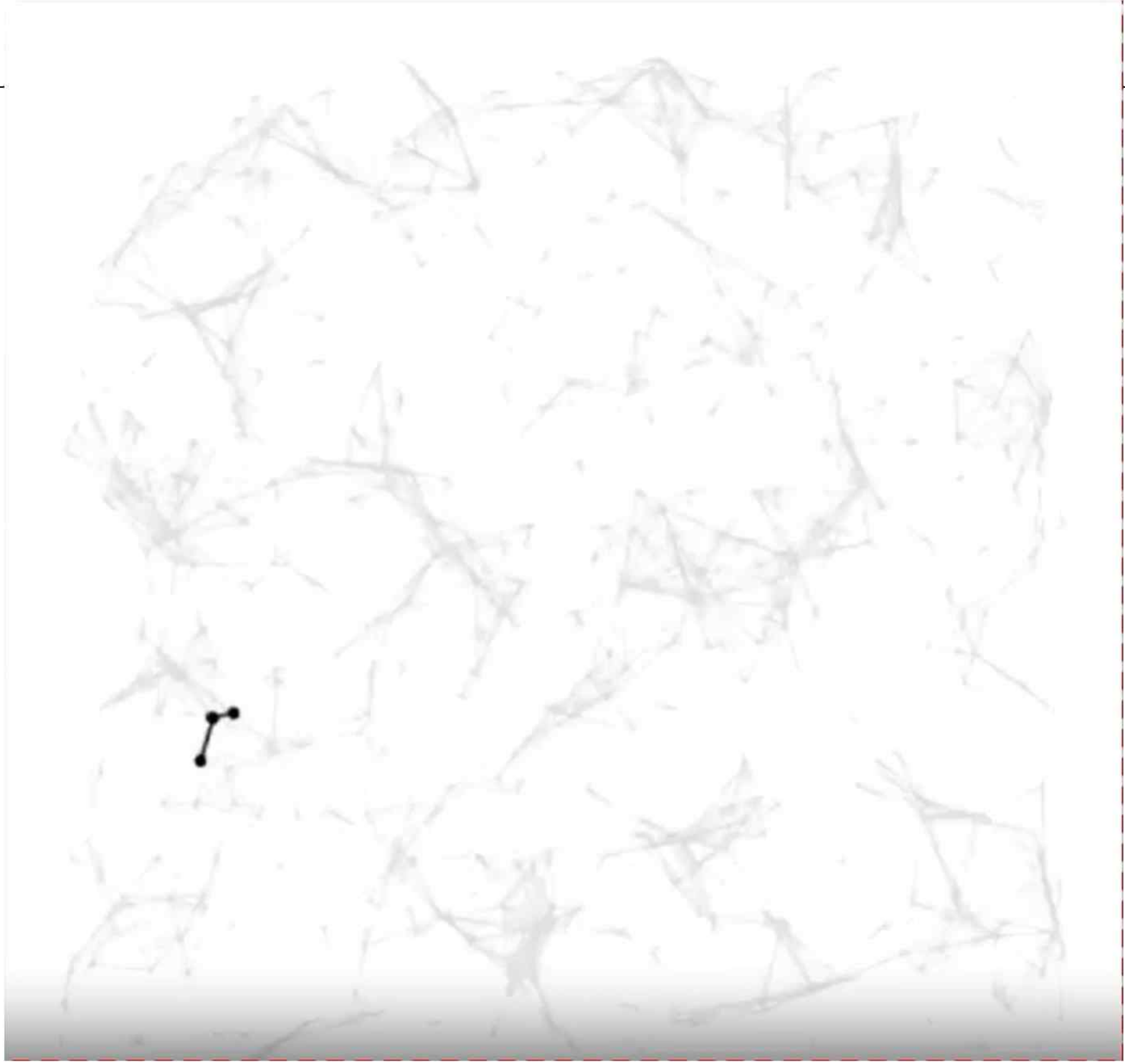
② 지금까지 만들어둔 MST / 나머지 정점 모두

Prim's Algorithm

검은색: Greedy algorithm 그대로
푸른색: Crossing edge 추가할 partition 선정 방식만 특화한 부분

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- **정점 0은 MST에 포함**된 상태라 봄
- MST와 나머지 정점 연결하는 간선 중
- weight 가장 작은 간선을 MST에 추가하는 것 반복
- 총 V-1개 간선 포함하면 종료





Kruskal: 분산된 여러 덩어리 만들기
Prim: 한 덩어리에서 계속 뻗어 나감
(간선 weight이 모두 다르다면) 결과로 얻은 MST는 같음

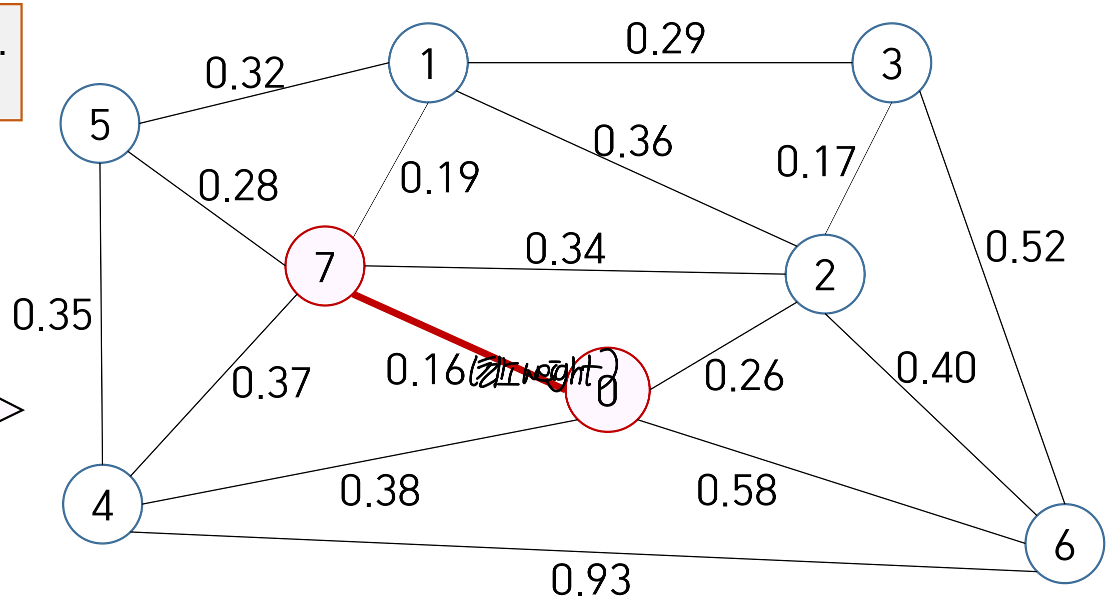
Prim's Algorithm ⊂ Greedy MST algorithm

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- **정점 0은 MST에 포함**된 상태라 봄
- **MST와 나머지 정점 연결하는 간선 중**
- **weight 가장 작은 간선**을 MST에 추가하는 것 반복
- 총 V-1개 간선 포함하면 종료

[Q] Prim's Algorithm은 Greedy MST에 포함. 따라서 MST 만들어 냄. Why?

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- (어떤 방식으로 든) 아직 **서로 연결 안 한 partition** 찾기 (crossing edge 하나도 포함 안 한 partition 찾기)
- **Crossing edge 중 weight 가장 작은 간선** MST에 포함
- 총 V-1개 간선 포함하면 종료

partition 1: 정점 0에서 시작해 지금까지 연결한 덩어리
partition 2: 나머지 정점



[Q] '두 partition 연결하는 crossing edge 중 최소 weight 간선' 어떻게 효율적으로 찾을까?

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- 정점 0은 MST에 포함된 상태라 봄
- **현재까지 만든 MST와 나머지 정점 연결하는 간선 중**
- **weight 가장 작은 간선**을 MST에 추가하는 것 반복
- 총 V-1개 간선 포함하면 종료

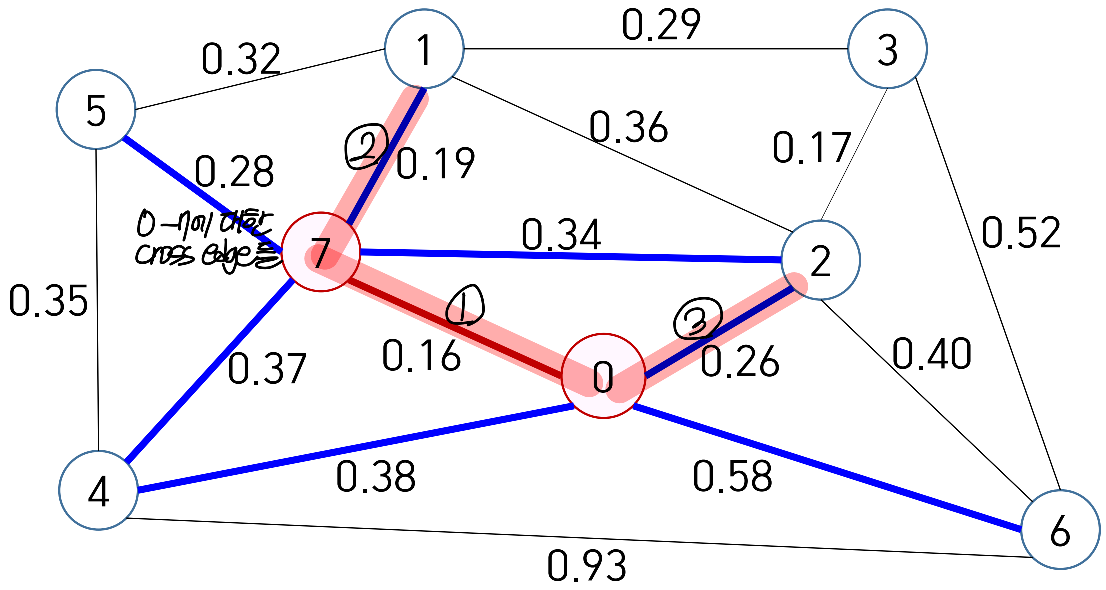
한 동아리만 유지하는 방식!

[A1] 모든 간선 차례로 다 확인해 보기

[Q] 간선 E개라면 걸리는 시간은? ~E

<간선과 weight>

- 0-7 (0.16)
- 2-3 (0.17)
- 1-7 (0.19)
- 0-2 (0.26)
- 5-7 (0.28)
- 1-3 (0.29)
- 1-5 (0.32)
- 2-7 (0.34)
- 4-5 (0.35)
- 1-2 (0.36)
- 4-7 (0.37)
- 0-4 (0.38)
- 6-2 (0.40)
- 3-6 (0.52)
- 6-0 (0.58)
- 6-4 (0.93)



#Prim이 왜 Greedy에 포함되는가
연결 안된 partition을 찾아야하는데, 그 연결 안된 partition이 지금까지 만든 MST와 나머지 정점이다
그래서 그 crossing edge 중에 최소 비중의 것들을 계속 포함하니까 여기에 딱 들어온다
-> 그러므로 Prim도 greedy MST이다
-> 그러므로 Prim도 올바른 MST를 잘 찾아낸다

minPQ (minimum Priority Queue) 활용 방법:
각 iteration마다 최소 weight 간선 찾는데 ~E 아닌 ~log(E)

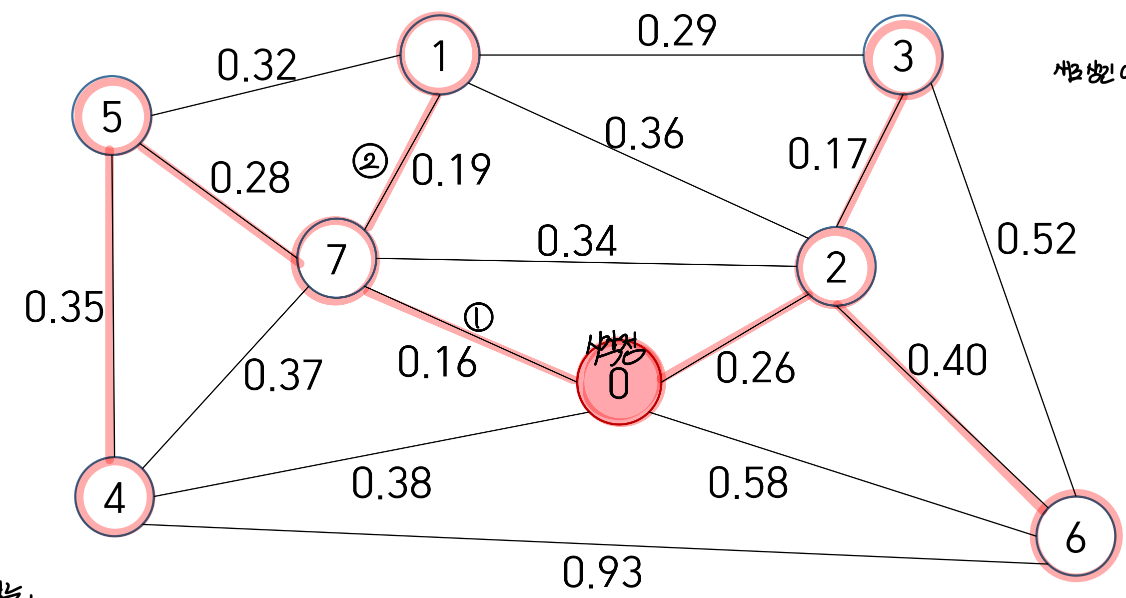
- 초기화: 0과 인접한 간선 모두 PQ에 추가
- V-1개 간선 추가할 때까지 아래 반복
  - PQ에서 weight 가장 작은 간선 v-w를 pop
    - v와 w 둘 다 MST 상에 있으면 이 간선 무시하고 다시 pop
  - v, w 중 v가 MST 상에, w가 외부에 있다고 가정
  - 간선 v-w와 새 정점 w를 MST에 추가
  - 새 정점 w와 인접한 간선 중 $\boxed{\text{MST 외부와 연결하는 간선}}$ 모두 PQ에 추가

← Cycle detection

각 정점의
MST 포함 여부
included[]

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

← Prim 알고리즘에서는 한정이라만 있기 때문에 T/F로 표기 가능



Priority Queue에서
(가장 비용이 작은 걸 계속 꺼내야 함. → Crossing Edge만 PQ에 넣음

<PQ에 저장된 간선>

0에서 나가는 간선

- ~~0-7 (0.16)~~ top ①
- ~~0-2 (0.26)~~ top ③
- 0-4 (0.38)
- 0-6 (0.58) → 시작점과 외부정점간의 Crossing Edge 먼저 PQ에 넣어

새로 생긴 cross edge (앞에서 이미 간선을 통해왔으니 이미 지나간 간선일 듯)

~~7-2 (0.34)~~ X (∵ Cycle 生)
~~7-1 (0.19)~~ top ②
~~7-5 (0.28)~~ ⑤
7-4 (0.37)

새로 생긴 cross edge

1-2 (0.36)
~~1-3 (0.29)~~ X
~~1-5 (0.32)~~ X

~~2-3 (0.17)~~ ④
2-6 (0.40)

3-6 (0.52)

~~5-4 (0.35)~~ ⑥

4-6 (0.93)

```
def mstPrimLazy(g):
    def include(v): # v를 MST에 추가 & 인접한 간선 중 MST 외부로 향하는 간선 모두 추가
        included[v] = True
        for e in g.adj[v]:
            if not included[e.other(v)]: pq.put(e)
```

정인 아무것도 포함 X → False로 초기화

초기화

```
    edgesInMST = [] # Stores edges selected as part of the MST
    included = [False] * g.V # included[v] == True if v is in the MST
    weightSum = 0  # Sum of edge weights in the MST
    pq = PriorityQueue() # Build a priority queue
    include(0)
```

: 정점0에 대해 호출하여 0에 인접한 간선이 다 PQ에 들어가게 하고나서 시작

초기화: 0을 MST에 추가 + 0과 인접한 간선 모두 PQ에 추가

```
    while not pq.empty() and len(edgesInMST) < g.V-1:
        e = pq.get()
        if included[e.v] and included[e.w]: continue # v-w 모두 MST 상에 있는 간선 무시
        edgesInMST.append(e)
        weightSum += e.weight
        if not included[e.v]: include(e.v) # v,w 중 아직 MST에 포함 안 한 정점과 간선 포함
        if not included[e.w]: include(e.w)

    return edgesInMST, weightSum
```

```
def mstPrimLazy(g):
    def include(v): # v를 MST에 추가 & 인접한 간선 중 MST 외부로 향하는 간선 모두 추가
        included[v] = True
        for e in g.adj[v]:
            if not included[e.other(v)]: pq.put(e)

    edgesInMST = [] # Stores edges selected as part of the MST
    included = [False] * g.V # included[v] == True if v is in the MST
    weightSum = 0  # Sum of edge weights in the MST
    pq = PriorityQueue() # Build a priority queue
    include(0)

    while not pq.empty() and len(edgesInMST) < g.V-1:
        e = pq.get()  PQ에서 꺼내서, 사이클을 만드는지/안만드는지 확인
Kruskal's UF의 Connected에 해당
        if included[e.v] and included[e.w]: continue # v-w 모두 MST 상에 있는 간선 무시
        edgesInMST.append(e)
        weightSum += e.weight   새로운 정점 포함시, 한 정점은 외부정점, 한 정점은 내부정점임. → 외부 정점만 포함하면 됨.
누가 외부정점인지 찾아보는 과정은
        if not included[e.v]: include(e.v) # v,w 중 아직 MST에 포함 안 한 정점과 간선 포함
        if not included[e.w]: include(e.w)

    return edgesInMST, weightSum
```

최소 weight 간선 v-w를 PQ에서 pop 한 후
v, w 모두 MST 상에 있지 않다면

minPQ (minimum Priority Queue) 활용 방법 성능: ~E log(E)

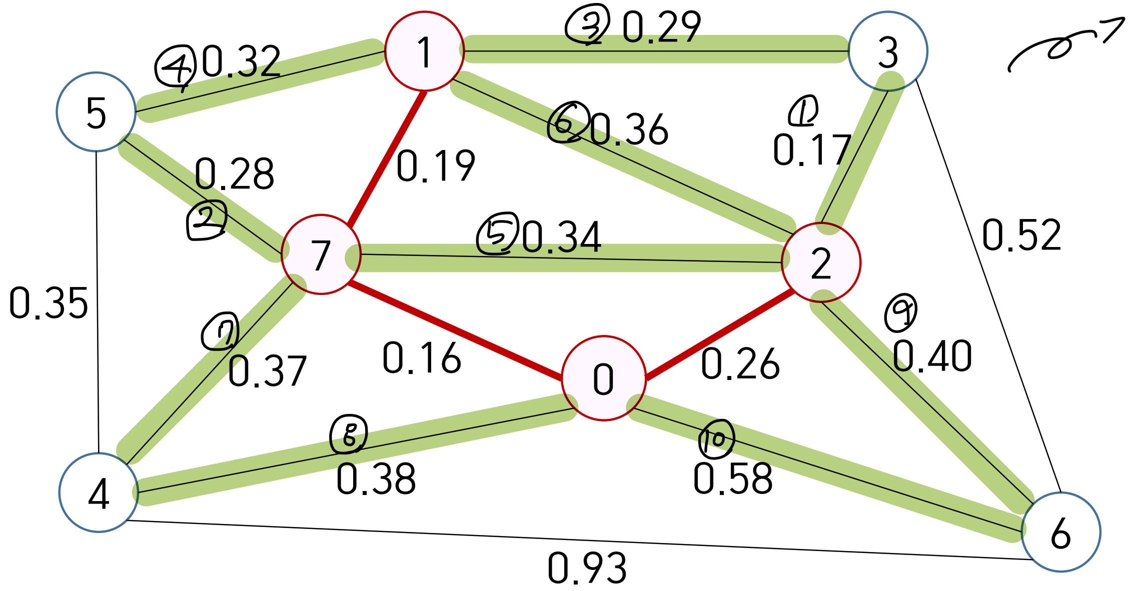
- 초기화: 0과 인접한 간선 모두 PQ에 추가
- V-1개 간선 추가할 때까지 아래 반복
  - **PQ에서 weight 가장 작은 간선 v-w를 pop**
    - **v와 w 둘 다 MST 상에 있으면 이 간선 무시하고 다시 pop**
  - v, w 중 v가 MST 상에, w가 외부에 있다고 가정
  - 간선 v-w와 새 정점 w를 MST에 추가
  - **새 정점 w와 인접한 간선 중 MST 외부와 연결하는 간선 모두 PQ에 추가**

	Operation	1회 비용	필요한 횟수	
	PQ, delete min	~logE	E	뽑을때마다 cycle 확인
	PQ, insert	~logE	E	
uf.connected( )	included[v] 확인	~1	E	
uf.union( )	included[v] 변경	~1	V	

ElogE

→ Kruscal과 똑같이 나오긴한데,
Prim이 Kruscal보다 상수시간만큼 빠름.
(UF를 쓰지 않아주기 때문)

[Q] 다음 그래프에 Prim의 알고리즘을 적용해 간선 0-7, 1-7, 0-2를 추가했다. 0-2를 추가한 후에는 PQ에 어떤 key 값이 들어있는가?



→ 외부로 나가는 Crossing Edge만 PQ에 들어있다

- 0.17 0.26 0.28 0.29 0.38 0.40
- 0.17 0.28 0.29 0.38 0.40
- 0.17 0.28 0.29 0.32 0.37 0.38 0.58
- 0.17 0.28 0.29 0.32 0.34 0.36 0.37 0.38 0.40 0.58

Minimum Spanning Tree (MST)

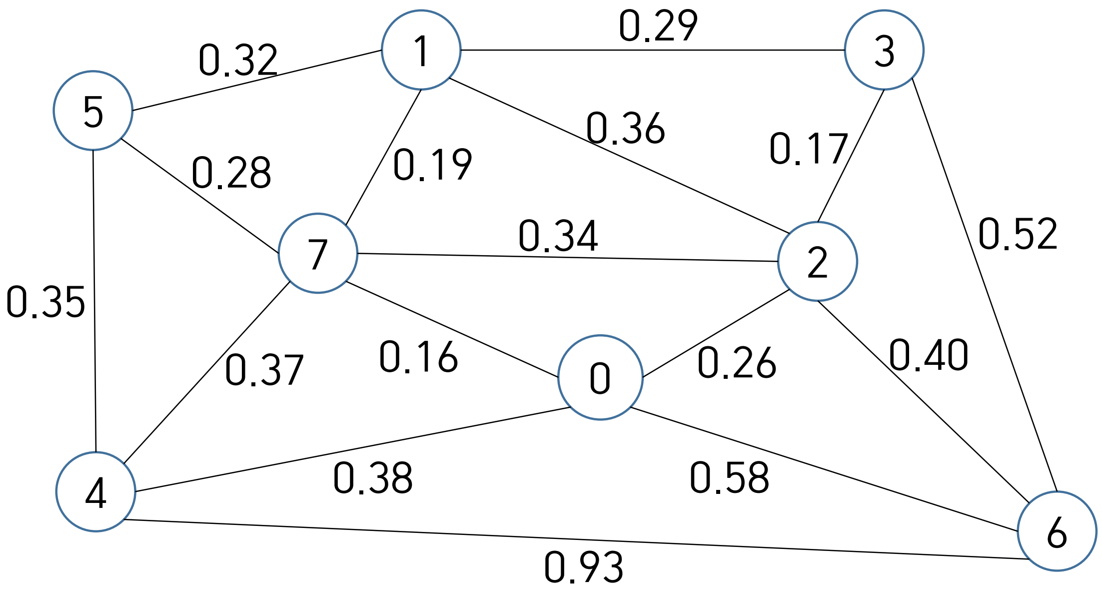
MST의 정의, 찾는 방법, 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. MST는 무엇이며, 어떻게 활용되는가?
03. MST의 성질 + Greedy 방법 개요
04. Kruskal's Algorithm
05. Prim's Algorithm Lazy Version
06. Prim's Algorithm Eager Version
07. 실습: Prim's Algorithm Eager Version 구현

Prim's Algorithm

검은색: Greedy algorithm 그대로
푸른색: Crossing edge 추가할 partition 선정 방식만 특화한 부분

- 아무 간선 포함 않은 상태에서 시작 (MST = [])
- **정점 0은 MST에 포함**된 상태라 봄
- MST와 나머지 정점 연결하는 간선 중
- weight 가장 작은 간선을 MST에 추가하는 것 반복
- 총 V-1개 간선 포함하면 종료



① Crossing Edge 아닌 것 제거

Prim′s Algorithm Lazy Version

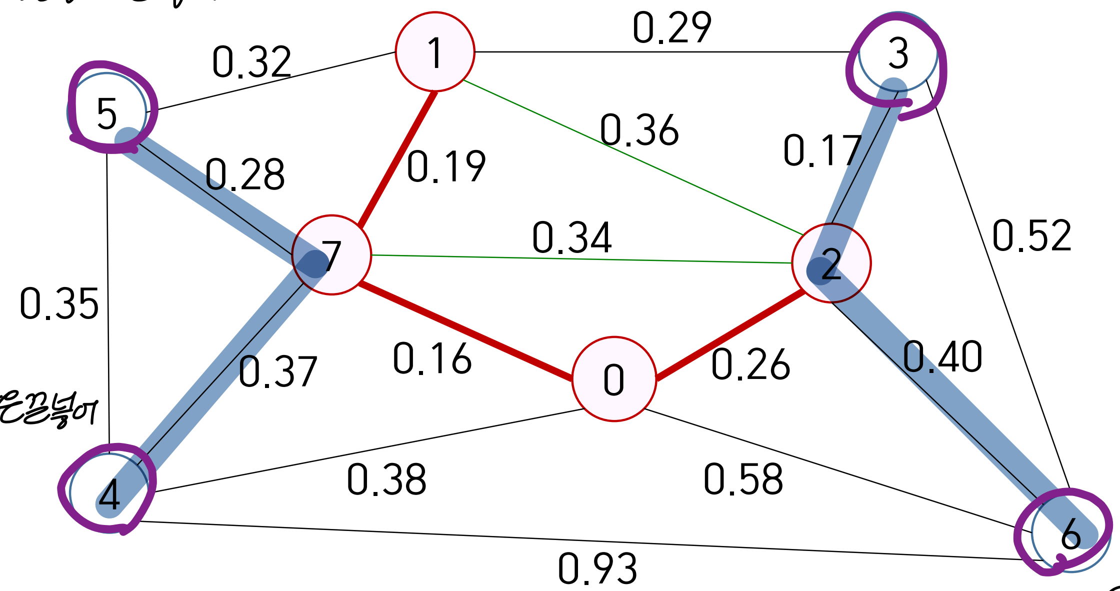
Prim′s Algorithm Eager Version

(최소비용이 아닌 것 제거m) 정점별로 최소 비용 간선 하나만 남겨둠.

<PQ에 저장된 간선>

2-3 (0.17)
5-7 (0.28)
1-3 (0.29)
1-5 (0.32)
7-2 (0.34)
1-2 (0.36)
4-7 (0.37)
0-4 (0.38)
2-6 (0.40)
0-6 (0.58)

둘 중 작은 걸 넣어



<PQ에 저장된 간선>

3 2-3 (0.17)
5 5-7 (0.28)
4 4-7 (0.37)
6 2-6 (0.40)

- MST와 나머지 정점 연결하는 간선 모두 포함
- MST 내부 연결하는 간선도 일부 포함 (기존에 추가되었으나 weight 높아 pop되지 않은 것)

MST 만드는데 불필요한 간선도 일단 PQ에 쌓아 두었다 나중에 pop 할 때 검사해보고 제거하므로 '**Lazy**'

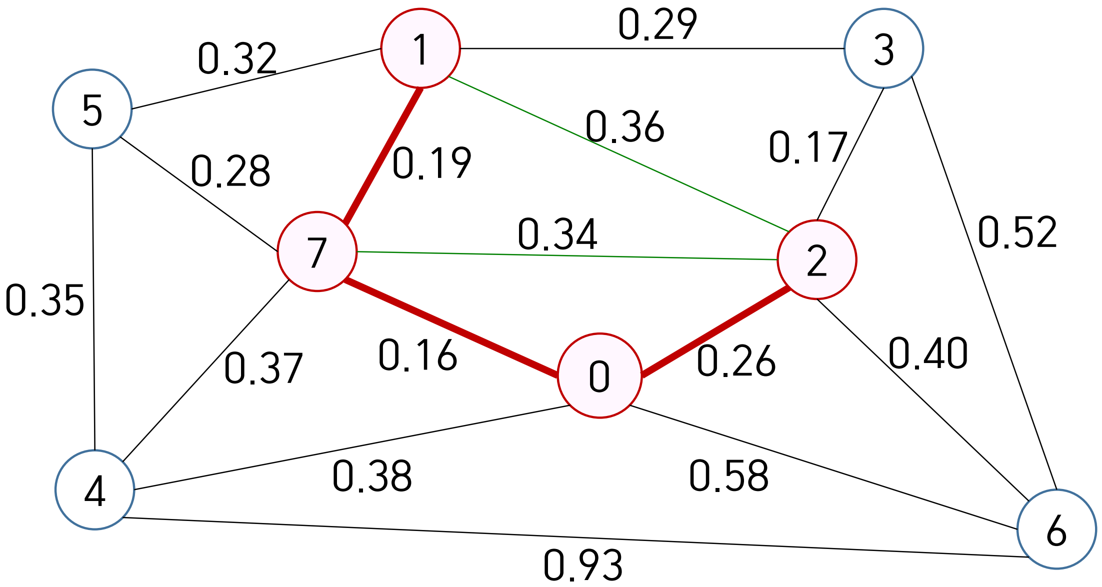
②

- MST에 포함되지 않은 나머지 정점별로 (한 번에 갈 수 있는 점들만) 최소 weight 간선 하나씩만 포함
- MST에 포함된 정점에 대한 간선 (MST 내부 연결하는 간선)은 포함하지 않음 (이미 최소 weight 간선이 pop 되었음)

MST 만드는데 꼭 필요한 간선만 PQ에 저장하므로 '**Eager**'

Prim's Algorithm Lazy Version

<PQ에 저장된 간선>
2-3 (0.17)
5-7 (0.28)
1-3 (0.29)
1-5 (0.32)
7-2 (0.34)
1-2 (0.36)
4-7 (0.37)
0-4 (0.38)
2-6 (0.40)
0-6 (0.58)



- PQ가 간선 수 E에 비례한 수의 간선 포함하므로
- insert, delete 비용 ~log E
- insert, delete 횟수 ~E

Prim's Algorithm Eager Version

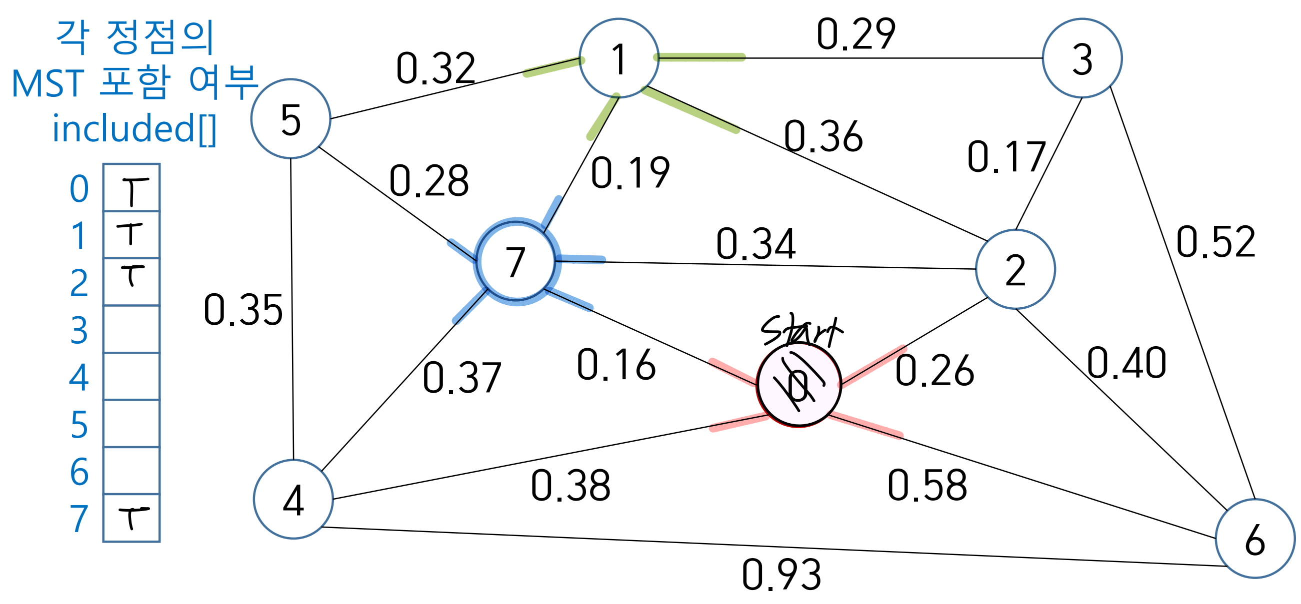
<PQ에 저장된 간선>	
3	2-3 (0.17)
5	5-7 (0.28)
4	4-7 (0.37)
6	2-6 (0.40)

정점별로 하나씩 저장

- PQ에 최대로 정점 수 V만큼의 값만 저장하므로
- insert, delete 비용 ~log V
- insert, delete 횟수 ~V
- 보통 V << E 이므로 비용 절감

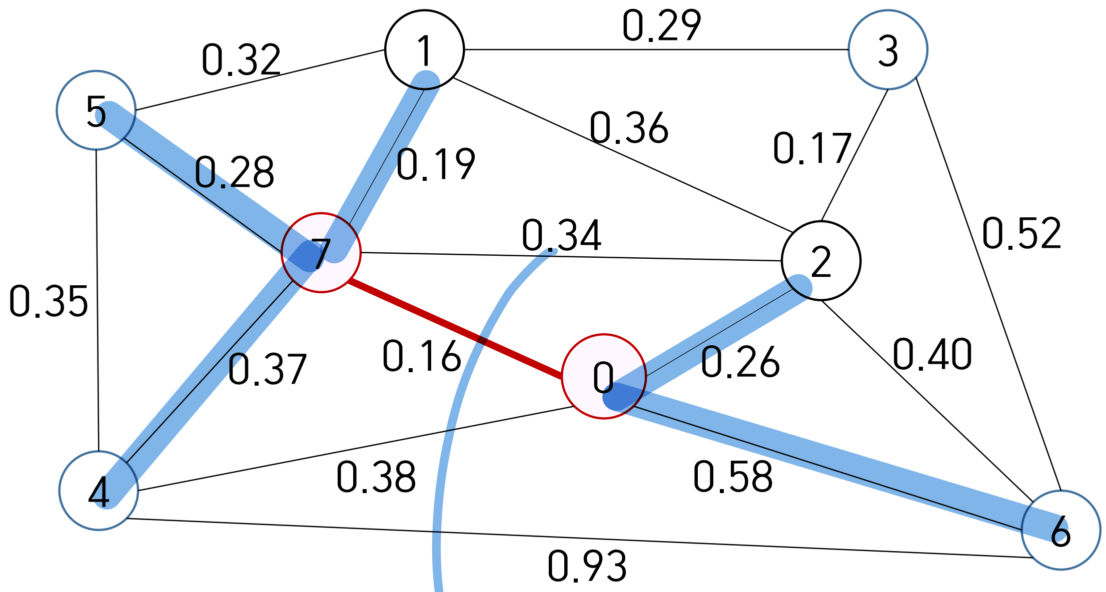
Prim′s Algorithm Eager Version: Indexed minPQ 활용 방법

- 초기화: 0과 **인접한 정점과 최소 weight 간선** PQ에 추가
- V-1개 간선 추가할 때까지 아래 반복
  - PQ에서 weight 가장 작은 간선 v-w를 pop해서 MST에 추가
  - v, w 중 w가 새로 MST에 연결된 정점이라고 가정
  - 새 정점 w와 인접한 간선 중 MST 외부 정점 x와 연결하는 간선에 w-x 대해
    - **x에 대한 간선이 아직 PQ에 없다면 PQ에 추가**
    - **x에 대한 간선 e가 이미 PQ에 있다면, w-x의 weight이 e보다 작은 경우 e를 대체 (decreaseKey)**



정점 번호 (index)	PQ에 저장된 간선[객체] (key)
0	Never used
② 1 (1로가는간선)	② 7-1 (0.19)
③ 2	① 0-2 (0.26)
3	1-3 (0.29)
4	① ~~0-4 (0.38)~~ ② ~~7-4 (0.37)~~ 5-4 (0.35)
5 (5로가는간선)	③ 7-5 (0.28)
6	① ~~0-6 (0.58)~~ 3-6 (0.52)
① 7 (7로가는간선)	① 0-7 (0.16)

[Q] 다음 그래프에 Prim's Algorithm Eager Version을 적용해 간선 0-7을 추가했다. 0-7를 추가한 후에는 indexed minPQ에 어떤 key 값이 들어있는가?



7-2는 안들어가나?

정점 번호 (index)	PQ에 저장된 간선 (key)
0	
1	7-1 (0.19)
2	0-2 (0.26)
3	
4	~~0-4 (0.38)~~ 7-4 (0.37)
5	7-5 (0.28)
6	0-6 (0.58)
~~7~~	~~0-7 (0.16)~~

Indexed minPQ (minimum Priority Queue) 활용 방법 성능

- 초기화: 0과 **인접한 정점과 최소 weight 간선** PQ에 추가
- V-1개 간선 추가할 때까지 아래 반복
  - PQ에서 **weight 가장 작은 간선 v-w를 pop**해서 MST에 추가
  - v, w 중 w가 새로 MST에 연결된 정점이라고 가정
  - 새 정점 w와 인접한 간선 중 MST 외부 정점 x와 연결하는 간선에 w-x 대해
    - **x에 대한 간선이 아직 PQ에 없다면 PQ에 추가**
    - **x에 대한 간선 e가 이미 PQ에 있다면, w-x의 weight이 e보다 작은 경우 e를 대체 (decreaseKey)**

Operation	1회 비용	필요한 횟수
PQ, delete min	logV	V
PQ, insert	logV	V (꺼낼때마다 담임)
PQ, decreaseKey	logV	E
included[v] 확인	~1	~E
included[v] 변경	~1	~V

~ElogV

Minimum Spanning Tree (MST)

MST의 정의, 찾는 방법, 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. MST는 무엇이며, 어떻게 활용되는가?
03. MST의 성질 + Greedy 방법 개요
04. Kruskal's Algorithm
05. Prim's Algorithm Lazy Version
06. Prim's Algorithm Eager Version
07. 실습: Prim's Algorithm Eager Version 구현

실습 목표: Prim's Algorithm의 Eager Version 구현

- 이번 시간에 배운 Prim's Algorithm Eager Version과 다른 알고리즘의 차이점 이해
- Indexed minPQ 적절하게 활용해 보기

프로그램 구현 조건

- Prim's algorithm eager version 수행하는 함수 구현

  def mstPrimEager(**g**):

- 입력 **g**: WUGraph 객체 (Weighted Undirected Graph 객체)
  - 입력은 항상 WUGraph 객체가 들어온다고 가정

- 반환 값: 2-tuple (**Edge 객체 리스트**, **weight 합계**)
  - (1) MST에 포함한 간선(Edge 객체) 리스트. Prim's algorithm에서 선정하는 순서대로 포함해야 함
  - **(2) MST에 포함한 간선의 weight 합계**

- 이번 시간에 제공한 코드 UndirectedWeightedGraph.py에 위 함수 작성해 제출
  - 위 코드에 포함된 IndexMinPQ 반드시 사용해야 함
  - mstPrimEager() 외 이미 작성된 코드는 변경하거나 삭제하면 안 됨
    - mstPrimEager() 함수 입력이 WUGraph 객체이므로 WUGraph와 Edge 클래스 반드시 필요
    - 위 코드에 포함된 mstKruskal(), mstPrimLazy() 함수는 결과 및 속도 비교 위해 필요
    - UF 클래스는 mstKrusal() 실행 위해 필요

이들 두 함수 참조해 작성하세요.

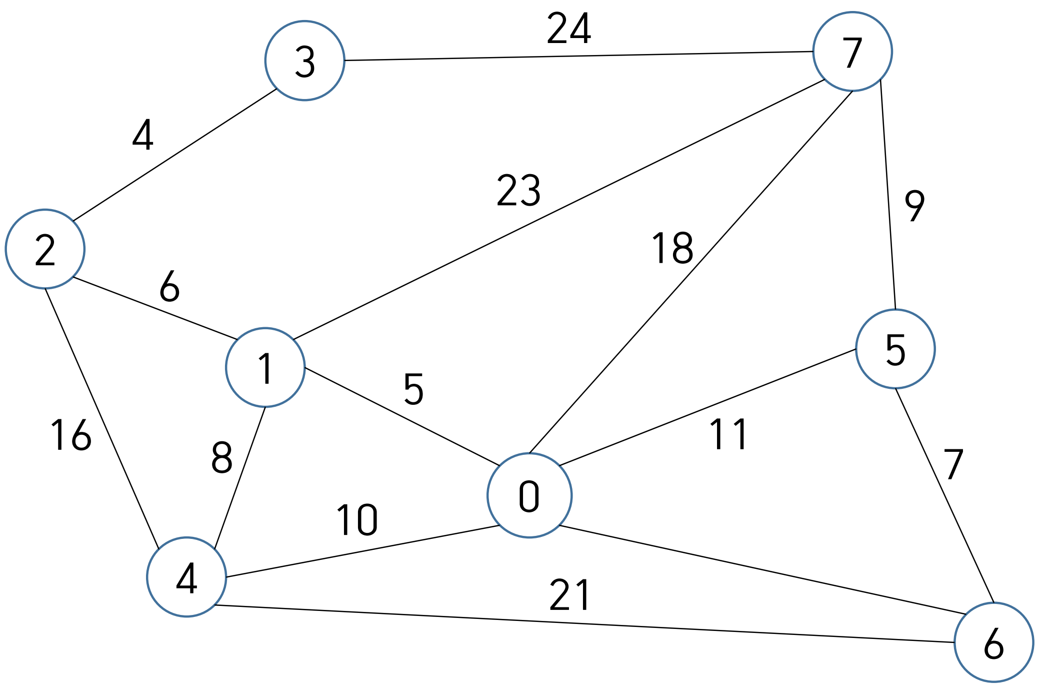
프로그램 구현 조건

- 최종 결과물로 UndirectedWeightedGraph.py 파일 하나만 제출하며, 이 파일만으로 코드가 동작해야 함
- import는 원래 UndirectedWeightedGraph.py 파일에서 import하던 3개 패키지 외에는 추가로 할 수 없음 (Path, PriorityQueue, timeit)
- 각자 테스트에 사용하는 모든 코드는 반드시 if __name__ == "__main__": 아래에 넣어
- 제출한 파일을 import 했을 때는 실행되지 않도록 할 것

프로그램 입출력 예: __main__ 아래 테스트 코드에 있음

```
g8a = WUGraph.fromFile("wugraph8a.txt")
print(mstPrimEager(g8a))
```

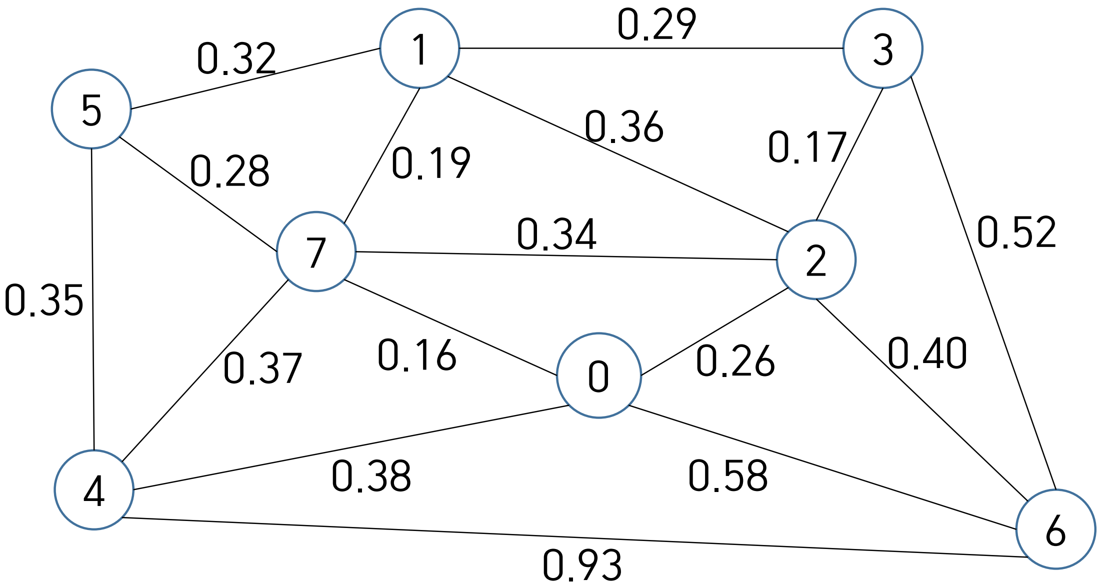
```
([0-1 (5.0), 1-2 (6.0), 2-3 (4.0), 1-4 (8.0), 0-5 (11.0), 5-6 (7.0), 5-7 (9.0)], 50.0)
```



프로그램 입출력 예: __main__ 아래 테스트 코드에 있음

```
g8 = WUGraph.fromFile("wugraph8.txt")
print(mstPrimEager(g8))
```

```
([0-7 (0.16), 1-7 (0.19), 0-2 (0.26), 2-3 (0.17), 5-7 (0.28), 4-5 (0.35), 2-6 (0.4)], 1.81)
```



그 외 입출력 예제 필요하면
임의의 그래프를 입력으로 주고
mstPrimLazy()의 반환값과 같은지 비교해 보세요.

프로그램 구현 조건 - 성능

- V << E인 그래프의 경우 mstKruskal, mstPrimLazy보다 mstPrimEager가 더 빠르게 결과 찾아야 함
- 채점 시에는 작성한 mstPrimEager의 실행 시간을 mstKruskal, mstPrimLazy과 비교해 더 빠른지 확인하며 (즉 상대 시간을 비교함) 절대 시간을 측정하지는 않음

- 실행 결과로 반환한 MST가 올바르지 않다면 성능 측정도 fail한 것으로 봄
- 답이 올바르지 않다면 성능 측정은 의미가 없으므로

프로그램 실행시간 출력 예: __main__ 아래 테스트 코드에 있음

```
g8 = WUGraph.fromFile("wugraph8.txt")
n = 100
print(timeit.timeit(lambda: mstKruskal(g8), number=n)/n)
print(timeit.timeit(lambda: mstPrimLazy(g8), number=n)/n)
print(timeit.timeit(lambda: mstPrimEager(g8), number=n)/n)
```

```
0.00013905000000000002
8.065799999999999e-05
4.093300000000008e-05
```

```
g8a = WUGraph.fromFile("wugraph8a.txt")
n = 100
print(timeit.timeit(lambda: mstKruskal(g8a), number=n)/n)
print(timeit.timeit(lambda: mstPrimLazy(g8a), number=n)/n)
print(timeit.timeit(lambda: mstPrimEager(g8a), number=n)/n)
```

```
6.279300000000002e-05
5.888400000000002e-05
4.669800000000002e-05
```

구현된 API 정리

```
# 이미 구현된 기능
class Edge: # Weight 있는 방향성 없는 간선 나타내는 클래스 (예습자료 참조)
            # WUGraph 객체 내부의 간선이 Edge 클래스 객체이므로 mstPrimEager(g) 작성 시 사용됨

class WUGraph: # Weight 있는 방향성 없는 그래프 나타내는 클래스 (예습자료 참조)
            # mstPrimEager(g) 작성 시 사용됨 (함수 입력 g가 WUGraph 객체이므로)

class UF: # Union Find를 수행하고 결과를 저장하는 클래스
WGU         # Kruskal's Algorithm 구현에 활용되며, Prim's Algorithm 구현에는 사용되지 않음

class IndexMinPQ: # Indexed minPQ를 나타내는 클래스 (예습자료 참조)
            # mstPrimEager(g) 작성 시 반드시 사용

def mstKruskal(g): # WUGraph 객체 g에 대해 Kruskal's Algorithm 수행하고 결과 반환하는 함수
            # 성능 테스트에 사용되며, Prim's Algorithm 구현에는 사용되지 않음

def mstPrimLazy(g): WUGraph 객체 g에 대해 Prim's Algorithm Lazy Version 수행하고 결과 반환하는 함수
            # 이 코드 참조해서 mstPrimEager(g) 작성하기
```

<IndexMinPQ에 대해 유의할 부분>
index: 정점 번호
key: 간선 (Edge 클래스 객체)
insert(index, key) 하는데
delMin() 함수는 2-tuple (key, index) 반환함에 유의

구현할 API 정리: def mstPrimEager(g)

```
# 구현해야 하는 기능
def mstPrimEager(g): # WUGraph 객체 g에 Prim's Algorithm Eager Version 수행 후 결과 반환
    def include(w): # mstPrimEager() 내부에서 호출하는 함수로
        # 정점 w를 MST에 포함할 때 수행해야 하는 일 기술
        # included[w] = True          contain(m)
        # w에 인접한 각 간선 e = w-x에 대해:
        #     정점 x가 아직 pq에 없다면 pq.insert(x, e)
        #     x가 이미 pq에 있고 pq에 저장된 간선보다 e의 weight이 더 작다면 pq.decreaseKey(x, e)
                                    keyof(m)

    # 필요한 자료구조 초기화
    # 결과 저장할 리스트(MST에 포함할 간선 저장하는 리스트)를 비어있는 리스트 []로 초기화
    # 각 정점의 포함 여부 결정하는 included[] 리스트를 모두 False로 초기화
    # pq = IndexMinPQ(g.V)
    # include(0)  # include(0) 호출해 정점 0에 인접한 정점을 모두 pq에 추가함

    # while 결과 리스트에 V-1개의 간선을 포함하지 않았다면:
    #     e, w = pq.delMin()
    #     e를 결과 리스트에 추가
    #     include(w)
    #
    # 결과 리스트와 이 리스트에 포함된 간선의 weight 합을 2-tuple로 반환
```

Hint: UndirectedWeightedGraph.py 파일에서
Prim's Algorithm Lazy Version 코드를 참조해 작성하세요.

스마트 출결

수업 전

문서 형태
개요/이론 학습

수업 전

개요 관련 퀴즈 수행
스스로 학습 & 시행착오

40~50분

수업(화/수)

퀴즈 풀이
개요 복습

수업(화/수)

응용/심화 내용 학습
질의 응답

수업(화/수)

실습 과제 (프로그래밍)
질의 응답

~3시간

이론 학습 (1.5~2시간) + 실습(1~1.5 시간) = 총 ~3시간

실습과제 수업 중 완료 못한 경우
당일 저녁 or 다음날까지
각자 수행해 제출

12:00까지 실습 & 질의응답

- 작성한 코드는 lms > 강의 콘텐츠 > 오늘 수업 > 실습 과제 제출함에 제출

- 시간 내 제출 못한 경우 내일 11:59pm까지 제출 마감
- 마감 시간 후에는 제출 불가하므로 그때까지 작성한 코드 꼭 제출하세요.