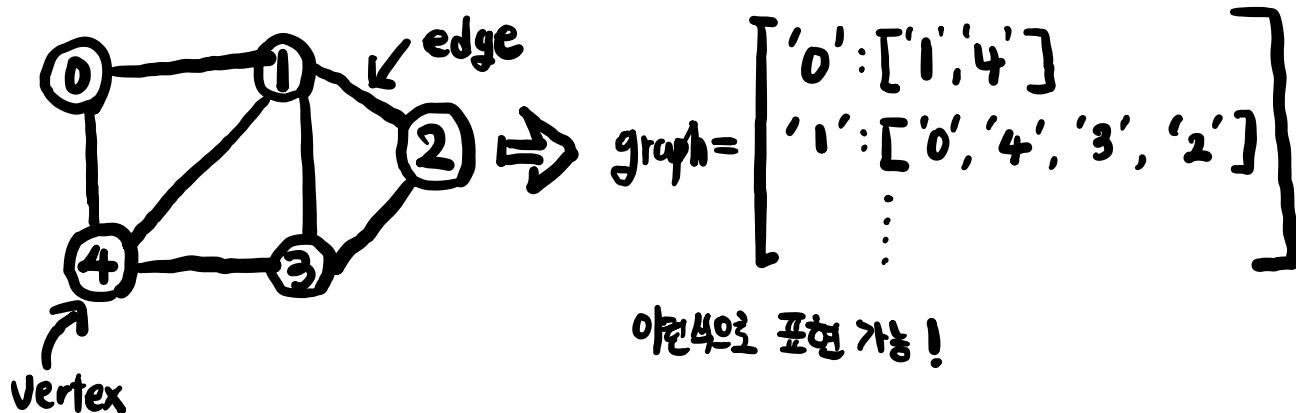


Graph

→ 그래프는 비선형 자료구조로, 정점(vertex or node)과 정점을 잇는 간선(edge)로 이루어져 있다. 위 간선은 cost/weight/value등의 가중치로 주어진다.



- **Undirected graph**: vertex \leftrightarrow edge 간의 방향성이 존재 X
- **directed graph**: vertex \leftrightarrow edge 간의 방향성이 존재 O
- **Weighted graph**: 가중치 graph (edge에 가중치 생김)
- **Subgraph**: 기존 그래프의 일부 정점 및 간선으로 이루어진 부분 그래프
- **Degree**: undirected graph에서 각 정점에 연결된 edge의 개수
 - [outdegree : vertex에서 다른 vertex로出去하는 edge 개수]
 - [indegree : 다른 vertex에서 해당 vertex로 들어오는 edge 개수]

⇒ DFS, BFS에 주로 활용! or.. 최단 노드 거리 구하기... 디스크스트라... 등등!

1. 디스크스트라 알고리즘 (Dijkstra)

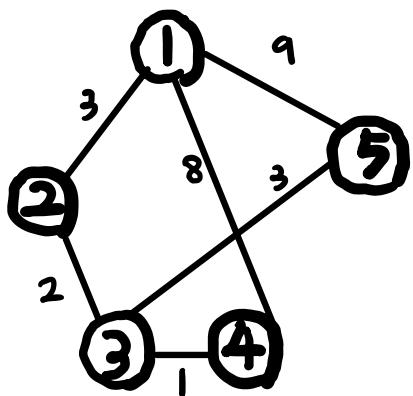
⇒ 가중치가 0인 edge가 있는 경우에 최소가 되는 경로를 찾는 알고리즘이다.

Greedy랑 DP가 섞여있는데, 현재 위치한 노드에서 최선의 경로를 반복적으로 찾으면서 계산해둔 경로를 활용해 중복된 하위 문제들을 푸다.

But 다익스트라는 그래프의 가중치가 음의 값이 존재하면 사용할 수 없다.

⇒ 최단경로라고 여겨진 경로 비용을 DP 테이블에 저장 후 재방문하지 않으려,
음의 가중치가 있다면, 이러한 규칙이 어긋나기 때문이다...
'플로이드 와샬', '벨만포드' 알고리즘은 음의 가중치도 사용 가능하다!

<example>

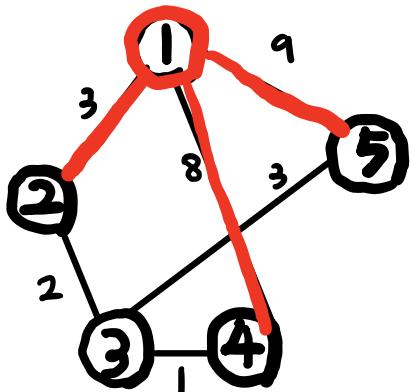


자기자신은 '0'으로 표현!
갈수없는 vertex는 'INF'로 표현!

0	3	INF	8	9
3	0	2	INF	INF
INF	2	0	1	3
8	INF	1	0	INF
9	INF	3	INF	0

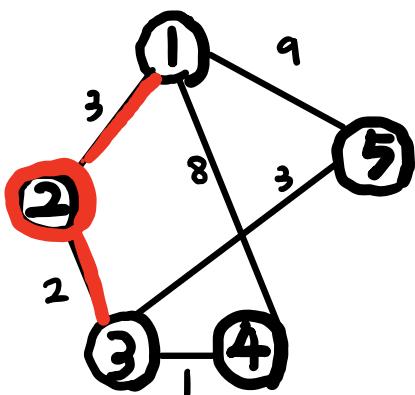
Dijkstra 알고리즘은 특정 한 노드에서 다른 모든 노드의 최단 거리를 구한다

예를 들어 ①을 특정 노드로 지정하면, table [0, 3, INF, 8, 9]는 다음과 같이 업데이트 된다.



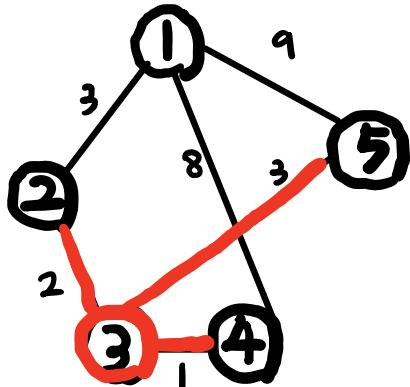
0	3	INF	8	9
0	3	INF	8	9

다른 vertex로 가는 가중치가 2번 vertex가 가장 작기 때문에,
2번 노드로 이동하고, 노드 1번을 방문처리한다. (테이블 변화 x)



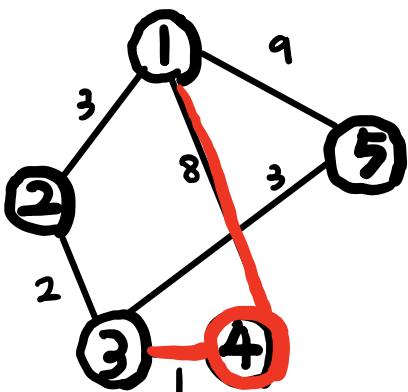
0	3	INF	8	9
---	---	-----	---	---

① 번노드는 이미 방문 했고 가중치도 초기 때문에 봐제!
 3번 노드(vertex)가 제일 가중치 값이 작기 때문에
 3번 노드로 옮긴다... 하지만! $1 \rightarrow 2 \rightarrow 3$ ($1 \rightarrow 3$)으로
 가는 비용이 $INF > 5$ 이므로 (최적의 상태를 계속 유지하므로)
 비용을 바꿔준다. (다른 노드를 탐색하다 3을 방문한 후로 있기 때문)



0	3	5	8	9
---	---	---	---	---

② 번노드 봐제, 4번 노드 가중치가 제일 작기 때문에
 ④ 번노드로 가기전! $\rightarrow 4$ 보다 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ 의
 가중치 6이 더 작은 것을 확인!
 또한 ⑤ 번노드 거리 또한 $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$: 8이 9보다 적음



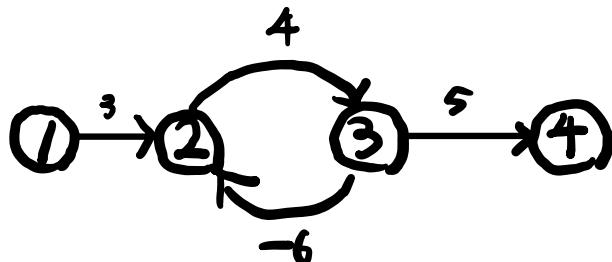
0	3	5	6	8
---	---	---	---	---

여기선 ①, ③ 노드가 모두 방문처리 됐으므로 업데이트 종료!
 위 데이블이 최종 ① 노드의 표이다.

1. 벨만포드 (Bellman Ford) 알고리즘

⇒ 다익스트라의 보완점으로 제시된 알고리즘이다. 기존 다익스트라 알고리즘은 음의 가중치일 때, 최단 경로를 구할 수 없다는 것 때문이다.

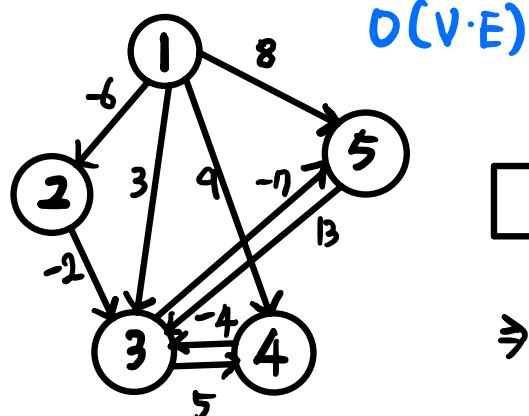
사실, 정확하게 말하면.. 사이클 형성 여부에 따라 달라지게 된다.



위 그림에서 $① \rightarrow ④$ 로 가는 최단거리를 구하고자 할 때,

$3 \rightarrow 4 \rightarrow 5$ (가중치) 가 아닌, $3 \rightarrow 4 \rightarrow -6 \rightarrow 4 \rightarrow -6 \dots$ 이렇게 비용이 무한히 작아지게 된다.

위 문제를 해결하기 위해, 다익스트라는 출발노드에서만 연결된 노드를 반복적으로 탐색한 것과는 달리 **벨만포드는 모든 노드가 출발점이 되어 다른 노드까지의 최소비용을 구한다.** (대신, 시간 복잡도가 늘어나게 된다.) → 사실 말이 해결이지...

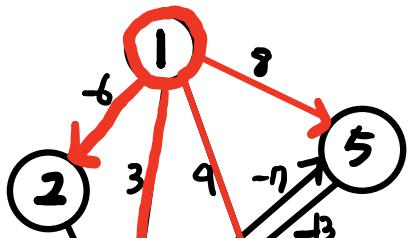


확인만 가능하다! 음의 사이클이 있으면 경로 못 구함!

INF	INF	INF	INF	INF
-----	-----	-----	-----	-----

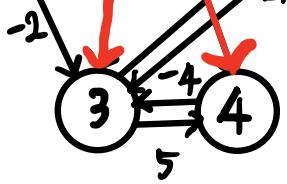
⇒ 모든 노드가 한 번씩 출발노드가 될 것이다.
위 그림과 표는 초기 상태의 그림이다.

[Iteration 1] 첫 번째로, 노드 1번을 먼저 탐색해보자. (차례대로!)

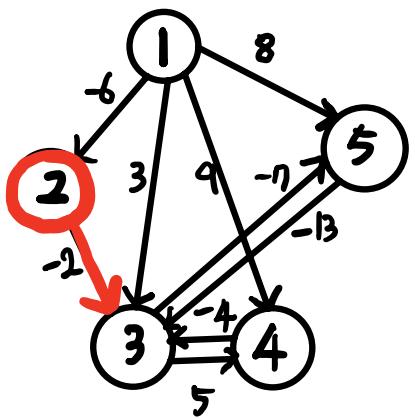


1번 노드에 연결된 인접 edge의 가중치를 업데이트한다.

0	-6	3	9	8
---	----	---	---	---

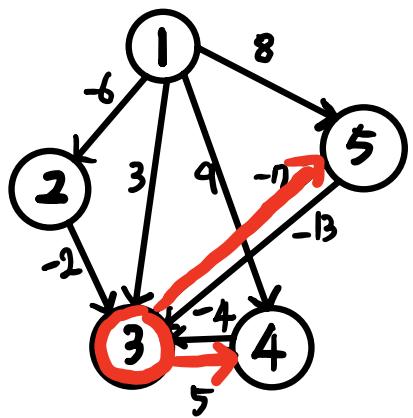


→ 사이클이 없으므로 그냥 0



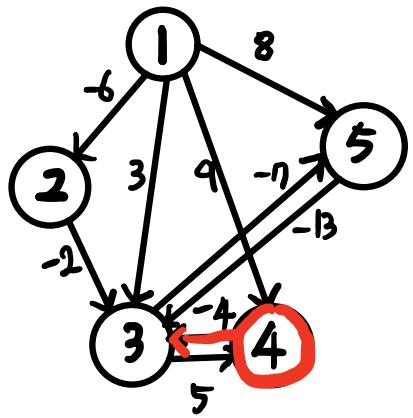
$(1 \rightarrow 3)$ 경로보다, $(1 \rightarrow 2 \rightarrow 3)$ 경로 비용이 더 작으므로 값을 고려한다.

0	-6	-8	9	8
---	----	----	---	---



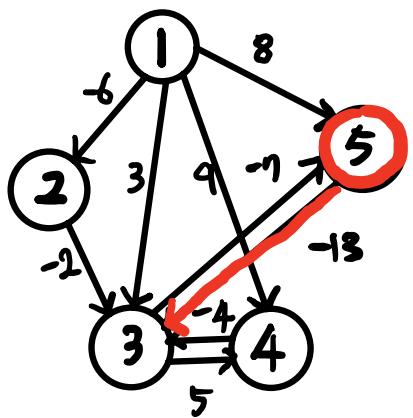
' $(1 \rightarrow 5)$ 보다, $(1 \rightarrow 2 \rightarrow 3 \rightarrow 5)$ ', ' $(1 \rightarrow 4)$ 보다 $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4)$ '

0	-6	-8	-3	-15
---	----	----	----	-----



$(1 \rightarrow 2 \rightarrow 3)$ 보다 $(1 \rightarrow 4 \rightarrow 3)$ 이 큼! (번동없음)

0	-6	-8	-3	-15
---	----	----	----	-----



$(1 \rightarrow 5)$ 가 0이 놀랄정도.

$(1 \rightarrow 3)$ 보다 $(1 \rightarrow 5 \rightarrow 3)$ 이 더 작아!

0	-6	-18	-3	-15
---	----	-----	----	-----

위 과정은 “노드개수 (v) - 1” 까지 반복한다.

음의 사이클이 존재 유무를 확인하고 싶은 경우에, 마지막 한 번 반복문을 돌려서 또 값이 갱신된다면, 음의 사이클이 존재한다고 보면 되겠다! (위 예시는 음의 사이클 존재...!)

3. 플로이드 워셜 (Floyd Warshall) 알고리즘

⇒ 모든 노드간의 최단거리를 구할 수 있다. (음의 가중치도 가능, but 음의 사이클 x)

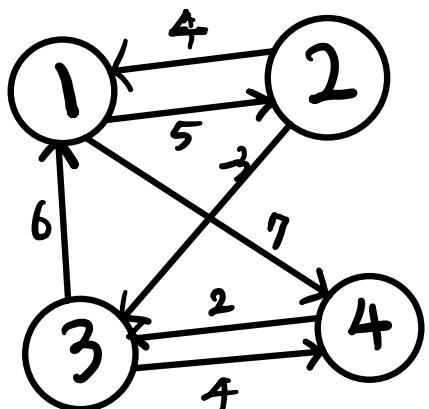
최단거리를 구할 때, 점화식이 사용되므로 동적 계획법에 해당된다.

이를 업데이트할 테이블이 필요하고 (모든 노드번호로 2차원 배열)

특정 노드에서 출발하는 디익스트라 알고리즈다 (1차원 배열 사용) 차이가 있다.

이 알고리즘의 핵심은, 각 단계마다 특정한 노드 K 를 거쳐가는 경우를 확인한다.

$$D_{ij} = \min(D_{ij}, D_{ik} + D_{kj}) \Rightarrow (i \rightarrow j) \text{ 와 } (i \rightarrow k \rightarrow j) \text{ 중 어느 곳이 최소비용인지 확인}$$



INF	5	INF	7
4	INF	-3	INF
6	INF	INF	4
INF	INF	2	INF

[Iteration 1] 거쳐가는 노드 $k=1$ (노드 1에서 노드 1을 거쳐 노드 1~4 탐색)

⑥ 자기자신 0값 넣음.

$$\textcircled{1} \quad \text{graph}[1][1] = \min (\text{graph}[1][1], \text{graph}[2][1] + \text{graph}[1][1]) \Rightarrow \text{변화} \times$$

$$\textcircled{2} \quad \text{graph}[1][2] = \min (\text{graph}[1][2], \text{graph}[2][1] + \text{graph}[1][2]) \\ = (5 = 5) \text{ 이므로 } \text{변화} \times$$

⋮

0	5	INF	7
4	INF	-3	INF
6	INF	INF	4
INF	INF	2	INF

[Iteration 1] $k=1$ 때로, 노드 2 \rightarrow 1을 거쳐 노드 1~4 탐색

$$\textcircled{1} \quad \text{graph}[2][1] = \min (\text{graph}[2][1], \text{graph}[2][1] + \text{graph}[1][1]) \\ = \min (4, 4 + 0) \Rightarrow \text{변화} \times$$

$$\textcircled{2} \quad \text{graph}[2][2] = \min (\text{graph}[2][2], \text{graph}[2][2] + \text{graph}[1][2]) \\ = \min (0, 4 + 5) = 0 \quad (\text{변화} \times)$$

$$\textcircled{3} \quad \text{graph}[2][3] = \min (\text{graph}[2][3], \text{graph}[2][3] + \text{graph}[1][3]) \\ = \min (-3, 4 + INF) = -3 \quad (\text{변화} \times)$$

$$\textcircled{4} \quad \text{graph}[2][4] = \min (\text{graph}[2][4], \text{graph}[2][4] + \text{graph}[1][4]) \\ = \min (INF, 4 + 7) = 11 \quad (\text{변화} \times)$$

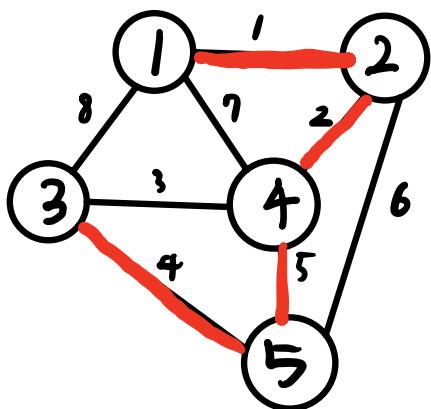
0	5	INF	7
4	0	-3	11
6	INF	INF	4
INF	INF	2	INF

⋮ (3, 4번째 생략)
⋮

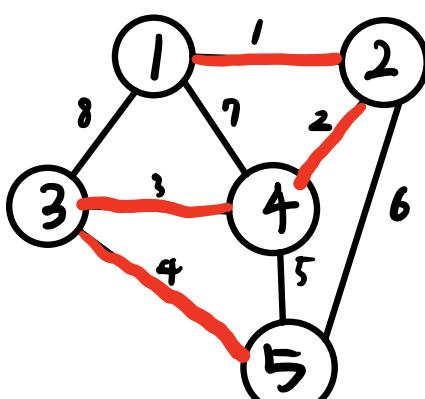
이렇게 Iterate 2, 3, 4도 반복! (각 노드 2, 3, 4를 거쳐 각 1~4를
또 둘씩 끼워 넣으므로)

4. 최소신장트리 (Minimum Spanning Tree)

→ 신장트리 (spanning tree)란, 그래프내의 모든 노드를 포함하면서 Cycle이 없는 부분 graph를 의미하는데 ... 그 중 간선에 가중치의 합이 최소가 되는 신장트리를 최소 신장 트리라고 한다.

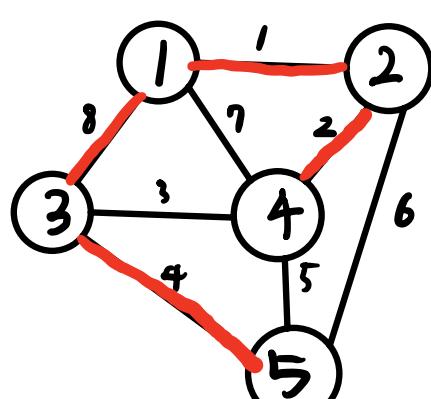


가중치합: 12



가중치합: 10

MST..!



가중치합: 15

4-1) 크루스칼(Kruskal) 알고리즘

⇒ 크루스칼 알고리즘은 그리디 알고리즘(greedy algorithm)으로 MST를 구한다.

간선의 가중치가 작은 것부터 하나하나 그래프에 포함시켜 그래프가 신장트리가 되게 만든다. (이때, cycle이 생기면 안되므로 뭉기전에 확인 작업을 한다!)

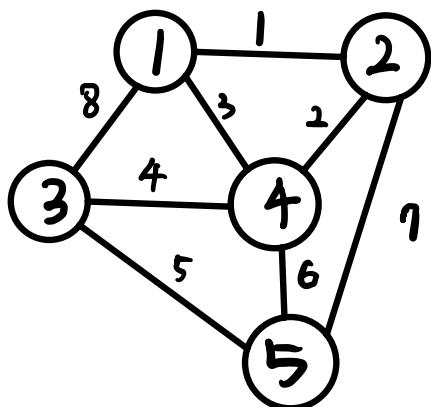
↳ 같은 루트노드를 가지고 있는지 확인하는, union-find 알고리즘 사용!



더 작은 값이 루트노드가 됨!

(밑에서 설명)

[example]



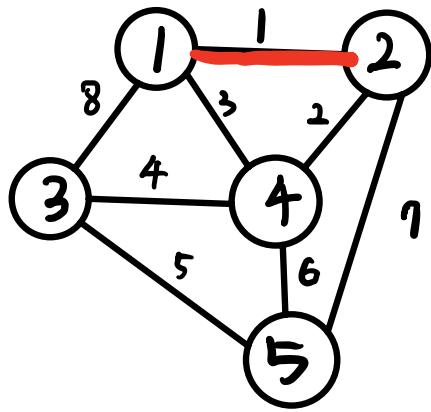
노드 1	노드 2	간선 가중치
1	2	1
2	4	2
1	4	3
3	4	4
3	5	5
4	5	6
2	5	7
1	3	8

간선의 가중치가
→ 가장 작은 것을 선택
해야 하므로!
오름차순으로 먼저 정렬!

노드	1	2	3	4	5
부모노드	1	2	3	4	5

total weight
= 0

- 1) 위의 간선 테이블을 (undirected graph) 오름차순으로 정렬하고 Cycle을 방지하기 위한 root table을 준비시킨다.

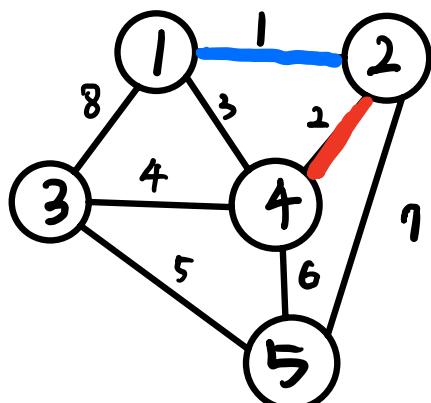


노드 1	노드 2	전체 가중치
1	2	1
2	4	2
1	4	3
3	4	4
3	5	5
4	5	6
2	5	7
1	3	8

노드	1	2	3	4	5
부모노드	1	2 → 1	3	4	5

total weight
= 1

- 2) 먼저, 제일 첫 번째에서 두 노드(노드 1, 노드 2)가 연결되어 있는지 확인 후 (root table에서 같은 부모노드를 가지면 연결된 것!) 처음 1, 2는 다르므로 연결후 노드 2의 부모노드를 더 작은 1로 설정한다.

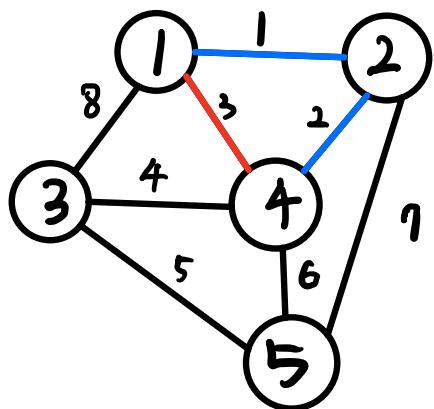


노드 1	노드 2	전체 가중치
1	2	1
2	4	2
1	4	3
3	4	4
3	5	5
4	5	6
2	5	7
1	3	8

노드	1	2	3	4	5

total weight

3) 마찬가지로 정렬된 graph에서 두 번째에 있는 0번, 4번 노드가 서로 연결되어 있는지 확인 후, 안되어 있으므로 union(결합) 후 부모노드를 업데이트 시켜준다. 아래, 2번 노드의 부모노드 값이 직접 4번 노드의 부모노드 값을, 2번 노드의 부모노드로 설정한다.

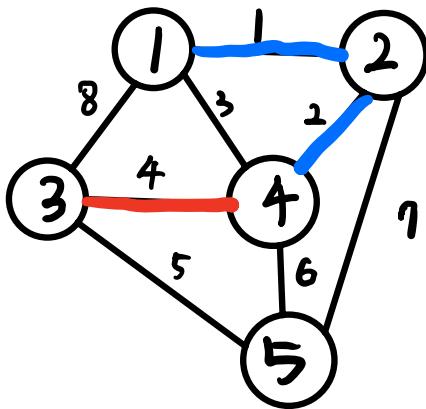


노드 1	노드 2	전선 가중치
1	2	1
2	4	2
1	4	3
3	4	4
3	5	5
4	5	6
2	5	7
1	3	8

노드	1	2	3	4	5
부모노드	1	1	3	1	5

total weight
= 3

4) 3번째를 거쳐와서 연결유무를 확인하는데.. 1번노드의 부모노드와 4번노드의 부모노드가 같으므로 연결하지 않고 건너뛴다! (\because cycle 방지)

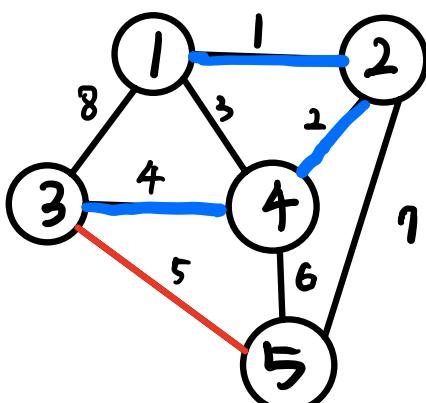


노드 1	노드 2	전체 가중치
1	2	1
2	4	2
1	4	3
3	4	4
3	5	5
4	5	6
2	5	7
1	3	8

노드	1	2	3	4	5
부모노드	1	1	3 → 1	1	5

total weight
= 7

5) 4번째) 간선 정보를 들고오면, 3번 노드와 4번 노드가 연결되어 있는지 확인한다. 둘의 부모노드가 다르므로 결합후 3번 노드의 부모노드를 업데이트한다.



노드 1	노드 2	전체 가중치
1	2	1
2	4	2
1	4	3
3	4	4
3	5	5
4	5	6
2	5	7
1	3	8

노드	1	2	3	4	5
부모노드	1	1	1	1	5 → 1

total weight
= 12

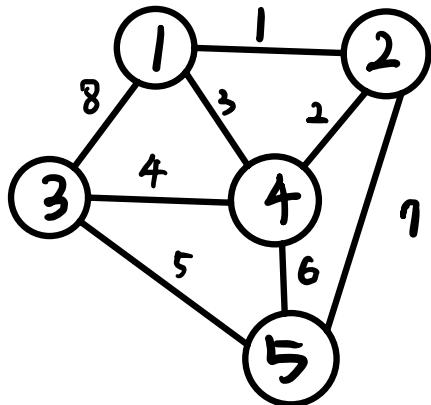
6) 5번째 값을 끄내 연결되어 있는지 확인 후, 연결되어 있지 않으면 부모노드값을 갱신한다.

7) 간선의 개수가 $V-1$ 개가 됐으면, (MST의 간선) 종료한다.

4-2) 프림(Prim) 알고리즘

→ 크루스칼 알고리즘과 더불어 그리디 알고리즘을 기본으로 MST를 구하는 알고리즈다.

[Example]



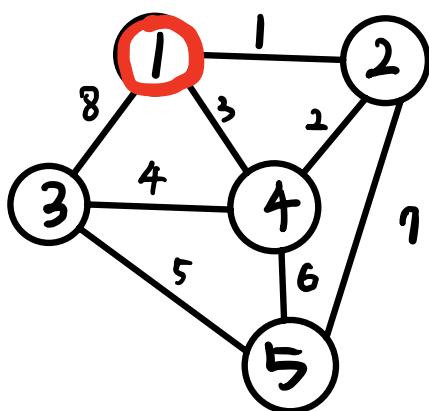
minimum heap (최소힙)	
<heap> for 무한순위 Queue	
노드	간선 가중치

연결된 Graph
(connected graph)

Node	1	2	3	4	5
Bool	False	False	False	False	False

Total weight =

- 1) Connected graph에 속하지 않은 연결된 노드 중, 간선의 가중치가 제일 작은 순으로 MST를 연결시켜야 하므로 우선순위 queue를 사용한다.



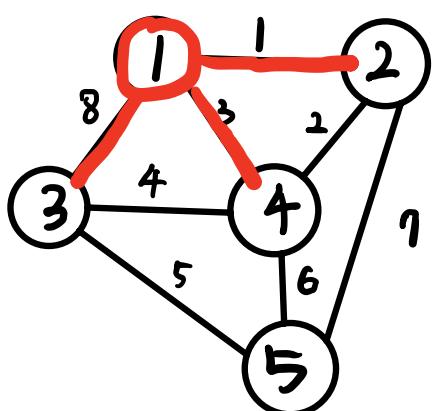
노드	전선 가중치
1	0

이진트리 (Graph)

Node	1	2	3	4	5
Bool	False	False	False	False	False

Total weight =

2) 노드 1을 시작으로 삼았을 때, 우선순위 큐에 담는다. (시작 노드이므로 가중치 0)



노드	전선 가중치
2	1
4	3
3	8

이진트리 (Graph)

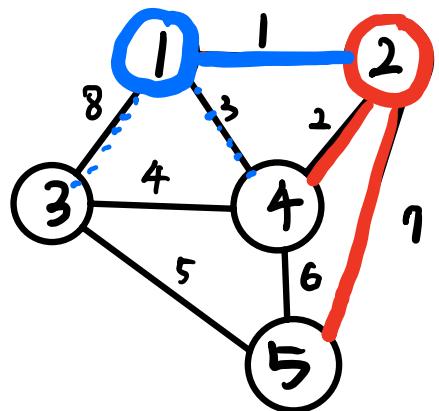
Node	1	2	3	4	5
Bool	False	False	False	False	False

Total weight =

True

3) Queue에서 값을 꺼낸 다음 connected graph에 포함 시킨다.

그리고 노드 1과 연결되어 있으면서 connected graph가 False인 노드를 우선순위 Queue에 넣어준다.



노드	전선 가중치
4	2
4	3
5	7
3	8

연결된 Graph

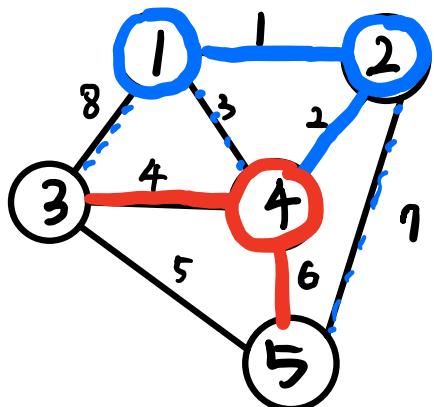
Node	1	2	3	4	5
Bool	True	False	False	False	False

Total weight = 1

True

4) Queue에서 값을 꺼낸 후, ($\text{노드 } 2, 1$) 해당 노드를 graph[] (connected)

뺀 후, 해당 노드와 연결된 노드 중, connected graph[] 속하지 않은 노드를 우선순위 Queue에 추가한다.



노드	전선 가중치
4	3
3	4
5	6
5	7
3	8

연결도 Graph

Node	1	2	3	4	5
Bool	True	True	False	False	False

Total weight = 3

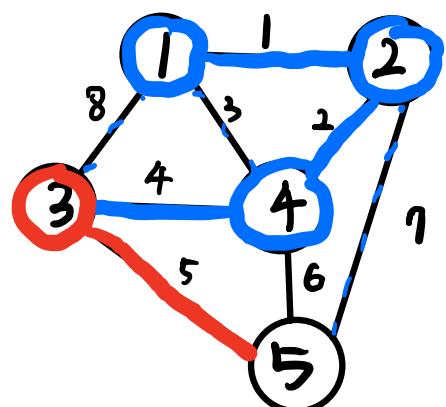
↓
True

5) [4, 2]를 Queue에서 뺀 후, connected graph에 추가시킨다.

이후 4번 노드와 연결되어 있으면서, connected graph에 속하지 않는 노드라면, [노드, 간선 가중치]를 우선순위 큐에 삽입한다.

5-1) 그 다음 [4, 3]을 꺼내고 할 때, 이미 3번 노드가 connected graph에 속하도록 그냥 삭제한다.

노드	간선 가중치
5	5
5	6
5	7
3	8



연결도 Graph

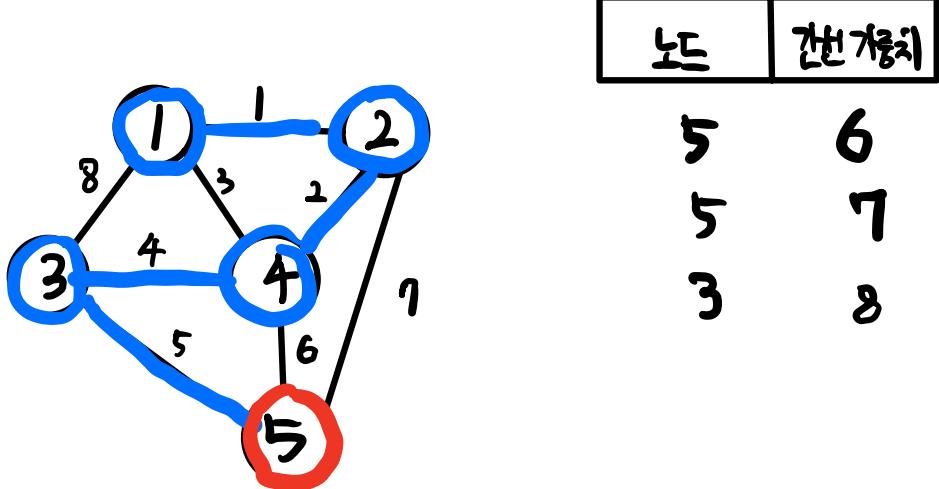
Node	1	2	3	4	5
Bool	True	True	False	True	False

Total weight = 7

↓
True

6) [3, 4]를 우선순위 Queue에서 꺼낸 후, connected graph에 속한 노드가 아니라면, 추가한다. 그리고 3번 노드와 연결된 노드 중,

(connected graph)에 속하지 않은 [노드, 간선 가중치]를 추가한다.



연결된 Graph

Node	1	2	3	4	5
Bool	True	True	True	True	False

Total weight = 12

True

7) [5, 5]를 Queue에서 제거후, (connected graph) 추가시킨다.

이때, (connected graph) False가 있다면 중지한다!