

- 

## 제 8 장 그래프

- **그래프**는 인터넷, 도로, 운송, 전력, 상하수도망, 신경망, 화학성분 결합, 단백질 네트워크, 금융 네트워크, 소셜네트워크 분석(Social Network Analysis) 등의 광범위한 분야에서 활용되는 자료구조이다.

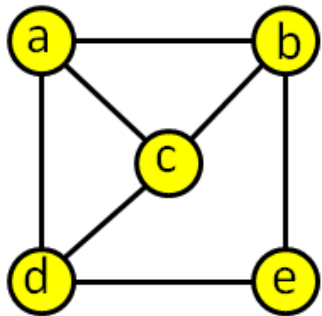
# 내용

- 그래프 용어
- 깊이우선탐색(DFS)
- 너비우선탐색(BFS)
- 연결성분(Connected Components)
- 이중연결성분(Doubly Connected Components)
- 강연결성분(Strongly Connected Components)
- 위상정렬(Topological Sort)
- 최소신장트리(Minimum Spanning Tree)
- 최단경로(Shortest Paths)

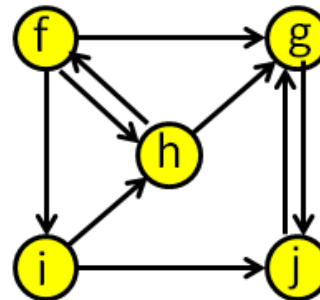
## 8.1 그래프

### 8.1.1 그래프 용어

- 그래프는 정점(Vertex)과 간선(Edge)의 집합으로 하나의 간선은 두 개의 정점을 연결
- 그래프는  $G=(V, E)$ 로 표현,  $V$ =정점의 집합,  $E$ =간선의 집합
- 방향 그래프(Directed Graph): 간선에 방향이 있는 그래프
- 무방향 그래프(Undirected Graph): 간선에 방향이 없는 그래프

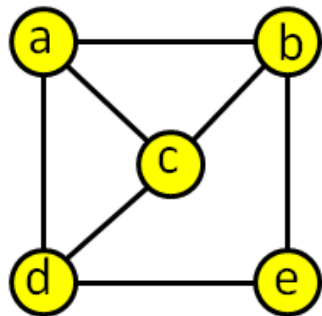


(a) 무방향그래프

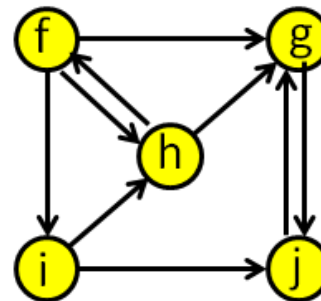


(b) 방향 그래프

- 정점 a와 b를 연결하는 간선을 (a, b)로 표현
- 정점 a에서 b로 간선의 방향이 있는 경우  $\langle a, b \rangle$ 로 표현
- 차수(Degree): 정점에 인접한 정점의 수
- 방향 그래프에서는 차수를 진입 차수(In-degree)와 진출 차수(Out-degree)로 구분
- 그림(a) 정점 a의 차수 = 3, 정점 e의 차수 = 2.
- 그림(b) 정점 g의 진입 차수 = 3, 진출 차수 = 1.

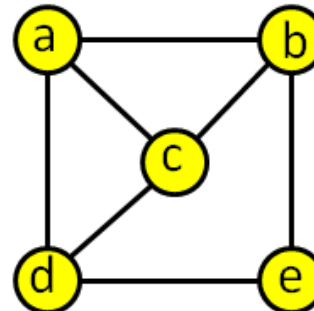


(a) 무방향그래프

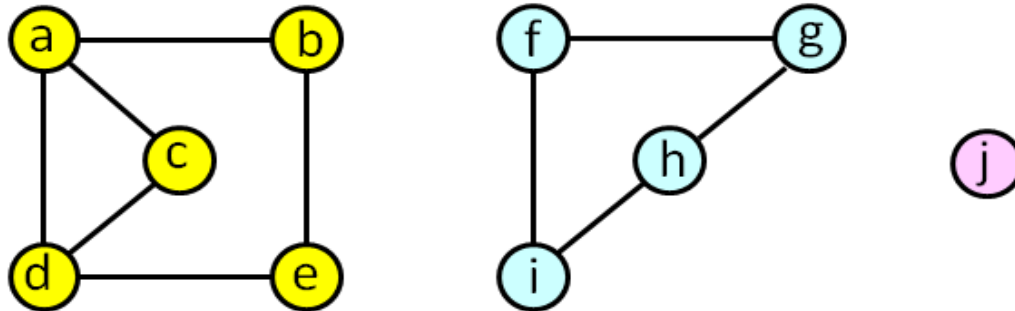


(b) 방향 그래프

- **경로(Path)**는 시작 정점  $u$ 부터 도착점  $v$ 까지의 정점들을 나열하여 표현
  - $[a, c, b, e]$ : 정점  $a$ 로부터 도착점  $e$ 까지의 여러 경로들 중 하나
- **단순 경로(Simple Path)**: 경로 상의 정점들이 모두 다른 경로
- ‘일반적인’ 경로: 동일한 정점을 중복하여 방문하는 경우를 포함
  - $[a, b, c, b, e]$ : 정점  $a$ 로부터 도착점  $e$ 까지의 경로
- **싸이클(Cycle)**: 시작 정점과 도착점이 동일한 단순 경로
  - $[a, b, e, d, c, a]$



- 연결성분(Connected Component): 그래프에서 정점들이 서로 연결되어 있는 부분 아래의 그래프는 3개의 연결성분,  $[a, b, c, d, e]$ ,  $[f, g, h, i]$ ,  $[j]$ 로 구성



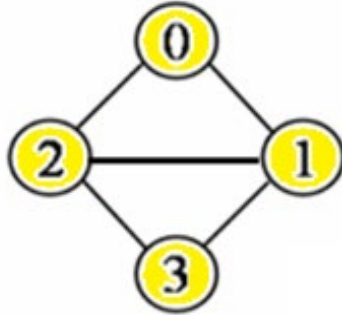
- **가중치(Weighted) 그래프**: 간선에 가중치가 부여된 그래프
  - 가중치는 두 정점 사이의 거리, 지나는 시간이 될 수도 있다. 또한 음수인 경우도 존재
- **부분그래프(Subgraph)**: 주어진 그래프의 정점과 간선의 일부분(집합)으로 이루어진 그래프
  - 부분그래프는 원래의 그래프에 없는 정점이나 간선을 포함하지 않음
- **트리(Tree)**: 사이클이 없는 그래프
- **신장트리(Spanning Tree)**: 주어진 그래프가 하나의 연결성분으로 구성되어 있을 때, 그래프의 모든 정점들을 사이클 없이 연결하는 부분그래프



## 8.1.2 그래프 자료구조

- 그래프를 자료구조로서 저장하는 방법
  - 인접 행렬(Adjacency Matrix)
  - 인접 리스트(Adjacency List)
- $N$ 개의 정점을 가진 그래프의 인접 행렬은 2차원  $N \times N$  리스트에 저장
- 리스트가  $a$ 라면, 정점들을  $0, 1, 2, \dots, N-1$ 로 하여, 정점  $i$ 와  $j$  사이에 간선이 없으면  $a[i][j] = 0$ , 간선이 있으면  $a[i][j] = 1$ 로 표현
- 가중치 그래프는 1 대신 가중치 저장

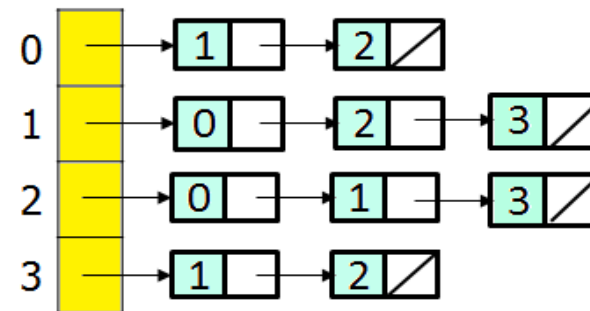
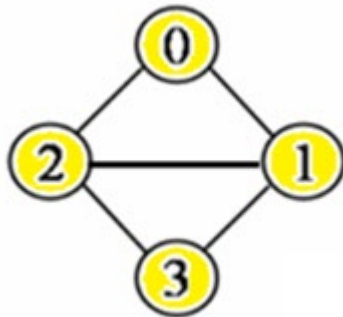
## 인접 행렬



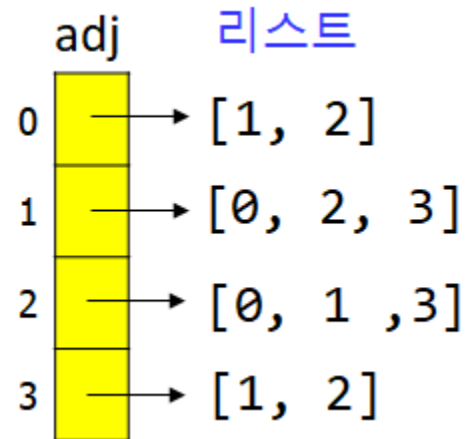
|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |

- 인접 리스트는 각 정점마다 1 개의 단순연결리스트를 이용하여 인접한 각 정점을 노드에 저장

## 인접 리스트



$\text{adj} = [[1, 2], [0, 2, 3], [0, 1, 3], [1, 2]]$



- 실세계의 그래프는 대부분 정점의 평균 차수가 작은 **희소 그래프(Sparse Graph)**이다.
- 희소그래프의 간선 수는 최대 간선 수인  $N(N-1)/2$ 보다 훨씬 작으므로 인접리스트에 저장하는 것이 매우 적절
  - 무방향 그래프를 인접리스트를 사용하여 저장할 경우 간선 1 개당 2개의 Edge 객체를 저장하고, 방향 그래프의 경우 간선 1 개당 1개의 Edge 객체만 저장하기 때문
- **조밀 그래프(Dense Graph):** 간선의 수가 최대 간선 수에 근접한 그래프

## 8.2 그래프 탐색

- 그래프에서는 두 가지 방식으로 모든 정점을 방문
- 깊이우선탐색(DFS; Depth First Search)
- 너비우선탐색(BFS; Breadth First)

A maze diagram illustrating a search path. The maze consists of several interconnected paths and dead ends. A green arrow indicates the initial path taken from the start point (green dot) towards the goal point (red dot). The path eventually leads to the goal point, which is marked by a red dot. Red arrows indicate the final segment of the path leading to the goal.

[핵심 아이디어] DFS는 실타래를 가지고 미로에서 출구를 찾는 것과 유사하다. 새로운 곳으로 갈 때는 실타래를 풀면서 진행하고, 길이 막혀 진행할 수 없을 때에는 실타래를 되감으며 왔던 길을 되돌아가 같은 방법으로 다른 경로를 탐색하여 출구를 찾는다.

- 그래프에서의 DFS는 임의의 정점에서 시작하여 이웃하는 하나의 정점을 방문하고,
- 방금 방문한 정점의 이웃 정점을 방문하며,
- 이웃하는 정점들을 모두 방문한 경우에는 이전 정점으로 되돌아가서 탐색을 수행하는 방식으로 진행

```

01 adj_list = [[2, 1], [3, 0], [3, 0], [9, 8, 2, 1],
02             [5], [7, 6, 4], [7, 5], [6, 5], [3], [3]]
03 N = len(adj_list)
04 visited = [None] * N
05
06 def dfs(v):
07     visited[v] = True
08     print(v, ' ', end='')
09     for w in adj_list[v]:
10         if not visited[w]:
11             dfs(w)
12
13 print('DFS 방문 순서:')
14 for i in range(N):
15     if not visited[i]:
16         dfs(i)

```

그래프 인접리스트

정점 방문 여부 확인 용

정점 v 방문

정점 v에 인접한 정점  
으로 dfs() 재귀호출

dfs() 호출

[프로그램 8-1] dfs.py

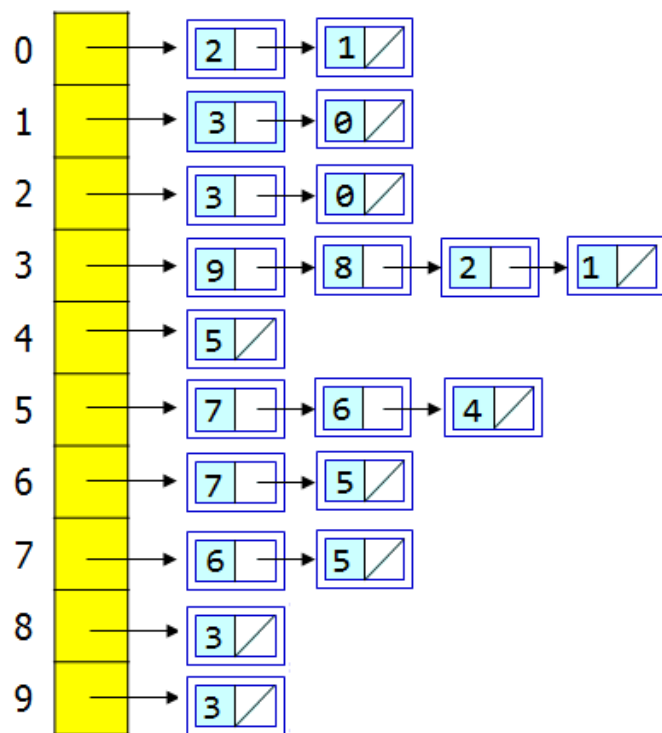
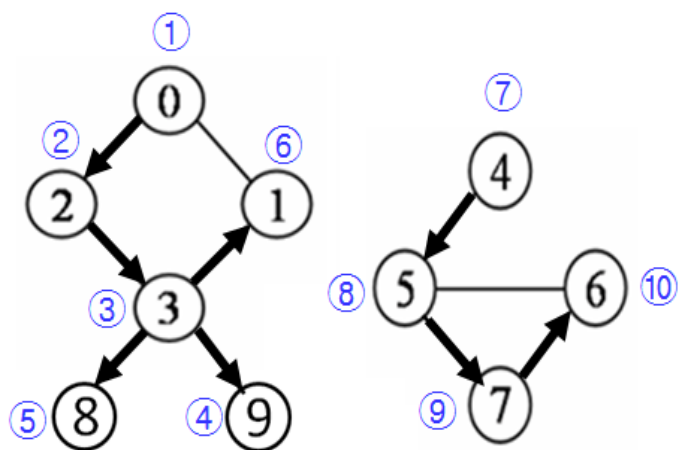
Console PyUnit

<terminated> dfs.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32

DFS 방문 순서:

0 2 3 9 8 1 4 5 7 6

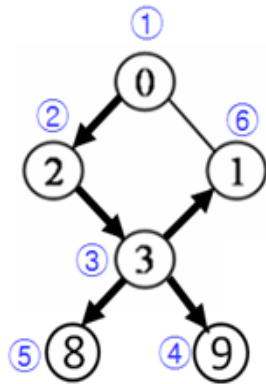
# DFS 수행 과정



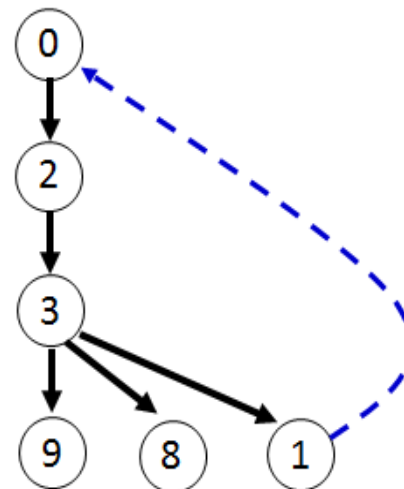
| 방문순서 | dfs()호출 | visited[]         | 출력 |
|------|---------|-------------------|----|
| ①    | dfs(0)  | visited[0] = True | 0  |
| ②    | dfs(2)  | visited[2] = True | 2  |
| ③    | dfs(3)  | visited[3] = True | 3  |
| ④    | dfs(9)  | visited[9] = True | 9  |
| ⑤    | dfs(8)  | visited[8] = True | 8  |
| ⑥    | dfs(1)  | visited[1] = True | 1  |
| ⑦    | dfs(4)  | visited[4] = True | 4  |
| ⑧    | dfs(5)  | visited[5] = True | 5  |
| ⑨    | dfs(7)  | visited[7] = True | 7  |
| ⑩    | dfs(6)  | visited[6] = True | 6  |



- (a)의 DFS 방문순서대로 정점 0부터 위에서 아래방향으로 정점들을 그리면 (b)와 같은 트리가 만들어진다.
- 실선은 탐색하며 처음 방문할 때 사용된 간선이고, 점선은 **뒷간선(Back Edge)**으로서 탐색 중 이미 방문된 정점에 도달한 경우를 나타낸다.
- 그래프가 1개의 연결성분으로 되어있을 때 DFS를 수행하며 만들어지는 트리를 **깊이우선 신장 트리(Depth First Spanning Tree)**라고 한다.



(a)



(b)

# 수행 시간

- DFS의 수행 시간은 탐색이 각 정점을 한번씩 방문하며, 각 간선을 한번씩만 사용하여 탐색하기 때문에  $O(N+M)$
- $N$ 은 그래프의 정점의 수이고,  $M$ 은 간선의 수

## 8.2.2 너비우선탐색



[핵심 아이디어] BFS는 연못에 돌을 던져서 만들어지는 동심원의 물결이 퍼져나가는 것 같이 정점들을 방문한다.

- BFS는 임의의 정점  $s$ 에서 시작하여  $s$ 의 모든 이웃하는 정점들을 방문하고, 방문한 정점들의 이웃 정점들을 모두 방문하는 방식으로 그래프의 모든 정점을 방문
- BFS는 이진트리에서의 레벨순회와 유사

```

01 adj_list = [[2, 1], [3, 0], [3, 0], [9, 8, 2, 1],
02             [5], [7, 6, 4], [7, 5], [6, 5], [3], [3]]
03 N = len(adj_list)
04 visited = [None] * N
05
06 def bfs(i):
07     queue = []
08     visited[i] = True
09     queue.append(i)
10     while len(queue) != 0:
11         v = queue.pop(0)
12         print(v, ' ', end='')
13         for w in adj_list[v]:
14             if not visited[w]:
15                 visited[w] = True
16                 queue.append(w)
17
18 print('BFS 방문 순서:')
19 for i in range(N):
20     if not visited[i]:
21         bfs(i)

```

그래프 인접리스트

정점 방문 여부 확인 용

큐를 리스트로 구현

큐의 맨 앞에서 제거된 정점을  
v가 참조하게 함

정점 v 방문

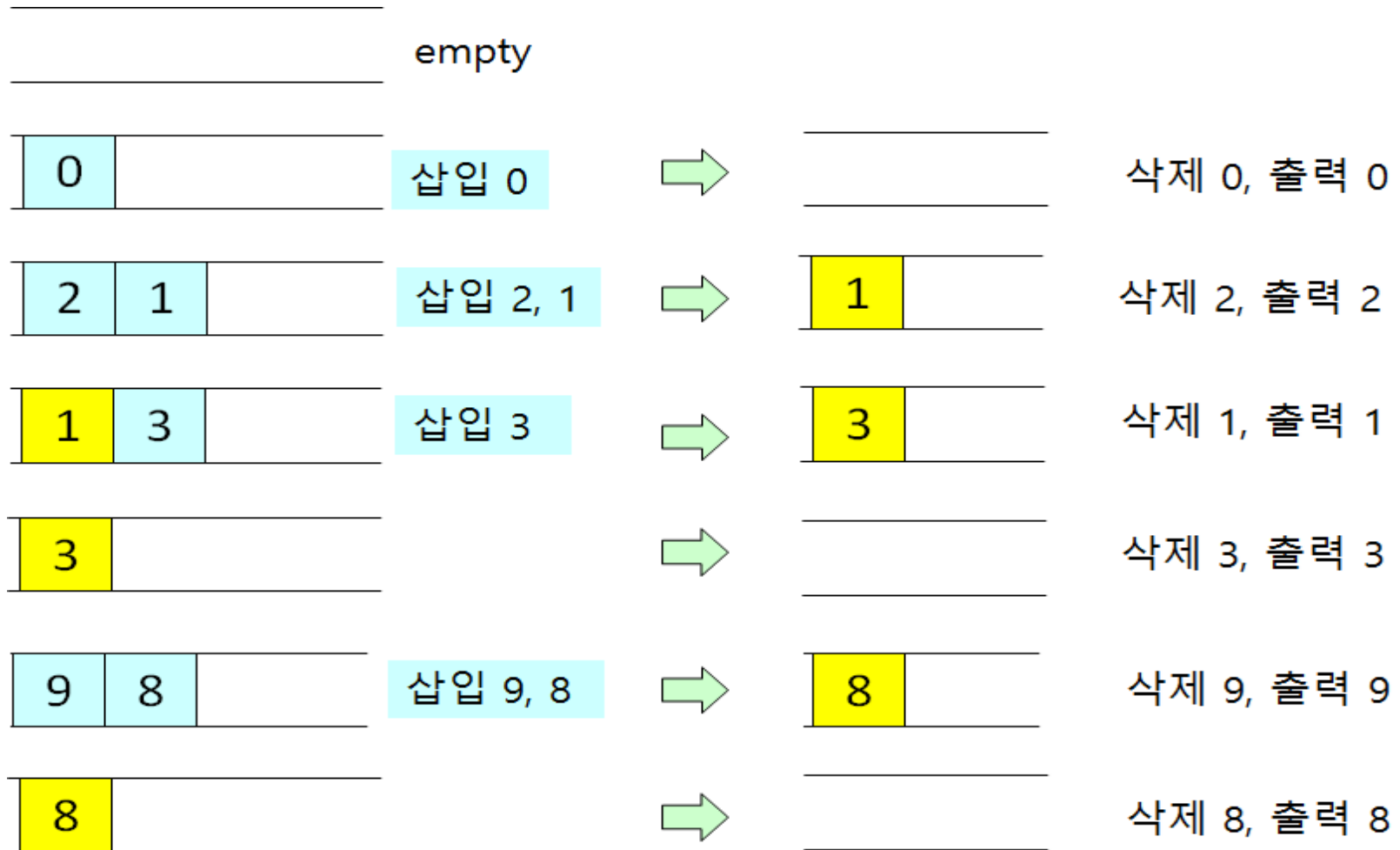
v에 인접하면서 방문  
안된 정점 큐에 삽입

bfs() 호출

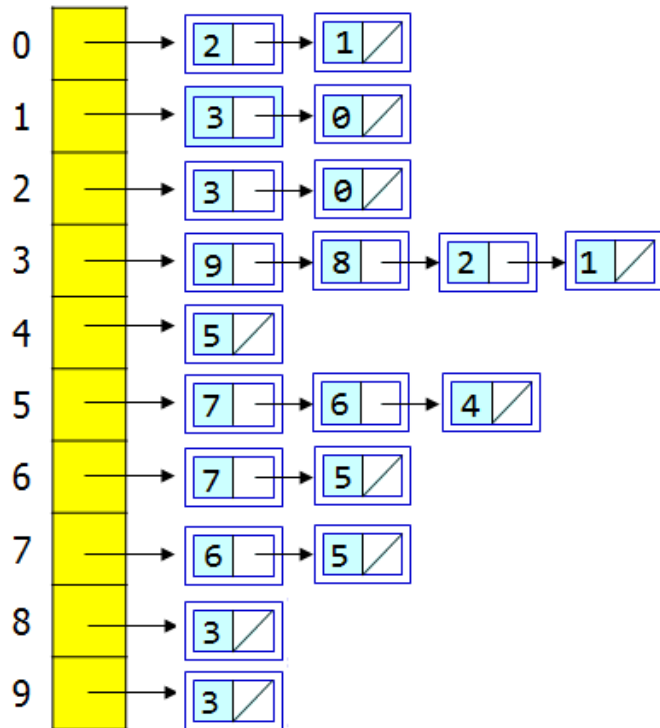
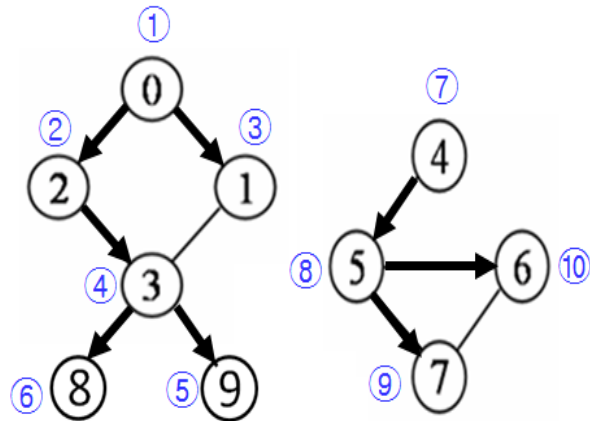
[프로그램 8-2] bfs.py

- Line 19의 for-루프는 0부터 N-1까지의 정점에 대해 bfs() 함수를 호출하여 그래프의 모든 정점들을 방문한다.
- Line 06의 bfs() 함수는 line 08~09에서 visited[i]를 True로 만들고, i를 큐에 삽입한다.
- Line 10의 while-루프는 큐가 empty가 되면 종료되고, 루프가 처음 시작하여 끝날 때까지 연속적으로 방문된 정점들은 하나의 연결성분을 구성한다.

- bfs(0)부터 수행되며 첫 번째 연결성분의 정점들을 모두 방문할 때까지 큐에 정점들이 삽입, 삭제되며 정점들이 출력(방문)될 때의 큐의 상태




## BFS 수행 과정



| 방문순서 | visited[]         | 출력 |
|------|-------------------|----|
| ①    | visited[0] = True | 0  |
| ②    | visited[2] = True | 2  |
| ③    | visited[1] = True | 1  |
| ④    | visited[3] = True | 3  |
| ⑤    | visited[9] = True | 9  |
| ⑥    | visited[8] = True | 8  |
| ⑦    | visited[4] = True | 4  |
| ⑧    | visited[5] = True | 5  |
| ⑨    | visited[7] = True | 7  |
| ⑩    | visited[6] = True | 6  |

## 프로그램 수행 결과

Console  PyUnit

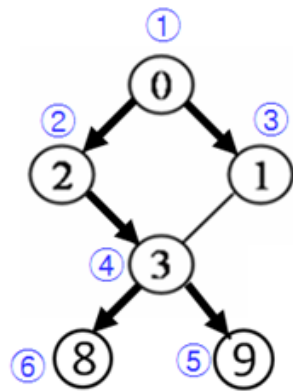
<terminated> bfs.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\

**BFS** 방문 순서:

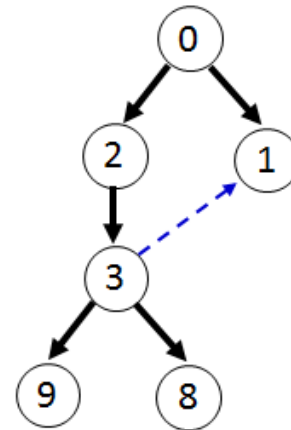
0 2 1 3 9 8 4 5 7 6



- (a)의 그래프에서 BFS 방문순서대로 정점 0부터 위에서 아래방향으로 그려보면 (b)와 같은 트리가 만들어짐
- 실선은 탐색하며 처음 방문할 때 사용된 그래프의 간선이고, 점선은 **교차 간선(Cross Edge)**으로서 탐색 중 이미 방문된 정점에 도달한 경우를 나타냄
- 그래프가 1개의 연결성분으로 되어 있을 때 BFS를 수행하며 만들어지는 트리: **너비우선 신장 트리 (Breadth First Spanning Tree)**



(a)



(b)

# 수행 시간

- BFS는 각 정점을 한번씩 방문하며, 각 간선을 한 번씩만 사용하여 탐색하기 때문에  $O(N+M)$ 의 수행시간이 소요
- BFS와 DFS는 정점의 방문 순서나 간선을 사용하는 순서만 다를 뿐이다.



## DFS와 BFS로 수행 가능한 그래프 응용

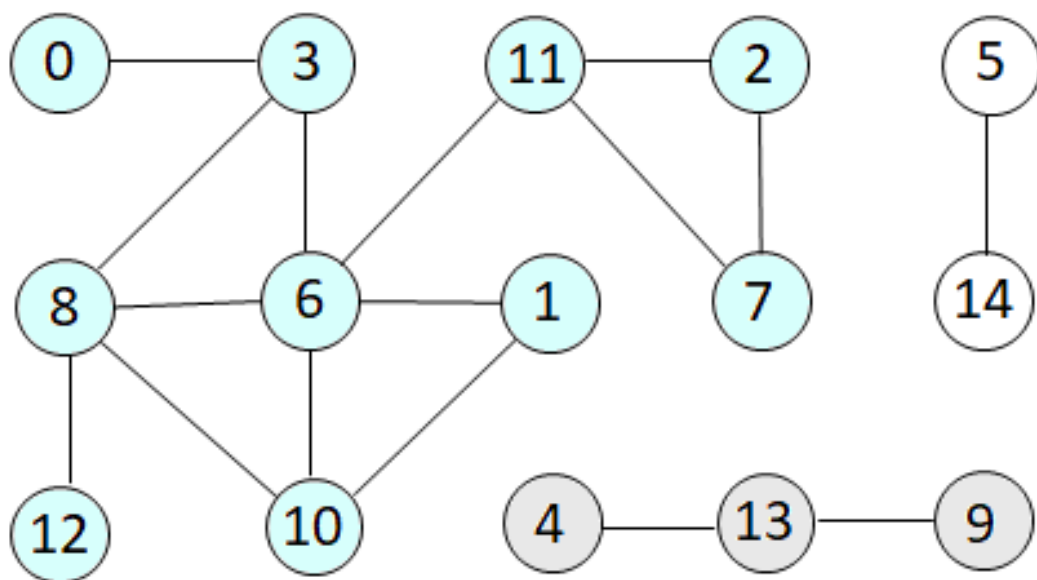
| 응용                    | DFS | BFS |
|-----------------------|-----|-----|
| 신장트리, 연결성분, 경로, 싸이클   | ✓   | ✓   |
| 최소 선분을 사용하는 경로        |     | ✓   |
| 위상 정렬, 이중 연결성분, 강연결성분 | ✓   |     |

## 8.3 기본적인 그래프 알고리즘

### 8.3.1 연결성분 찾기

[핵심 아이디어] 주어진 정점으로부터 DFS를 수행하여 방문되는 정점들을 리스트로 모아서 1개의 연결성분을 만든다. 다음엔 방문 안된 정점으로부터 같은 과정을 반복하여 다른 연결성분들을 추출한다.





## 프로그램 수행 결과

Console PyUnit

```
<terminated> cc.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\
```

연결성분 리스트:

```
[[0, 3, 6, 1, 10, 8, 12, 11, 2, 7], [4, 13, 9], [5, 14]]
```

- [프로그램 8-3]은 정점 0으로부터 DFS를 수행하여 [0, 3, 6, 1, 10, 8, 12, 11, 2, 7]을 차례로 방문하여 첫 번째 연결성분을 찾고,
- 이 정점들은 방문될 때 visited[]의 해당 원소가 True가 되었으므로, line 15의 for-루프에서는 다음 방문할 정점은 정점 4가 됨
- 왜냐하면 정점 4의 visited[] 원소가 False이기 때문
- 따라서 정점 4로부터 DFS를 수행하여 [4, 13, 9]를 두 번째 연결성분으로 찾고, 마지막으로 [5, 14]를 찾고 종료

## 수행 시간

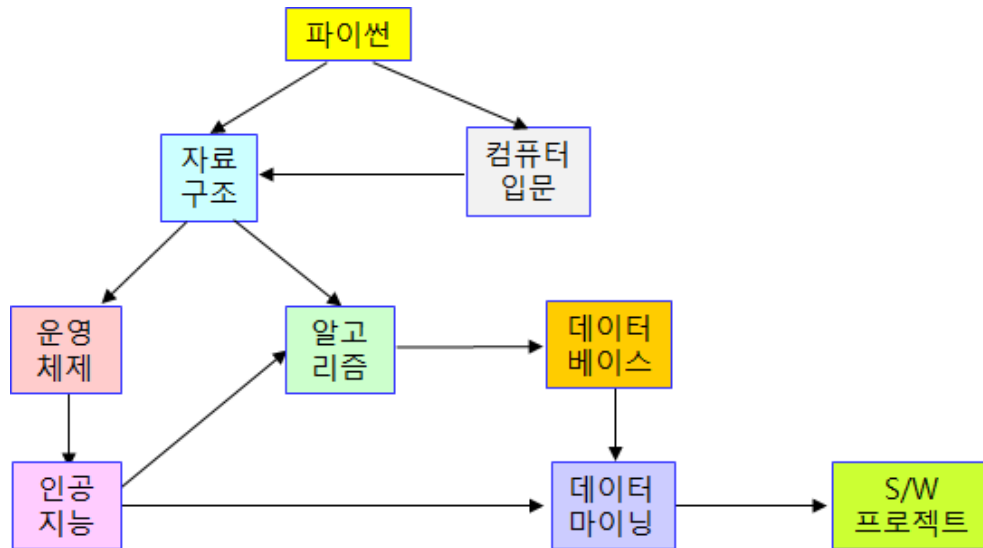
- 연결성분 찾기는 기본적으로 DFS를 수행하는 것이므로 각 정점을 1번씩 방문하며, 각 간선을 1번씩만 사용하여 탐색하기 때문에  $O(N+M)$ 의 수행시간이 소요된다. 여기서  $N$ 은 그래프의 정점의 수이고,  $M$ 은 간선의 수이다.



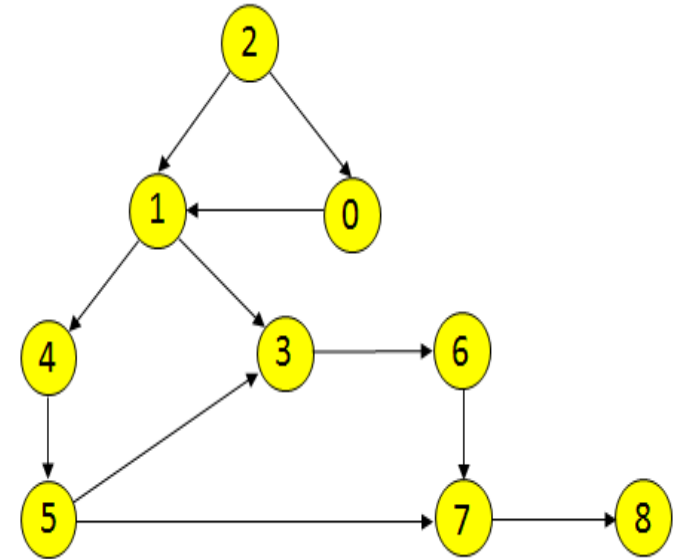
## 8.3.2 위상 정렬

- 위상 정렬(Topological Sort)이란 사이클이 없는 방향그래프(Directed Acyclic Graph, DAG)에서 정점을 선형순서(즉, 정점들을 일렬)로 나열하는 것
- 위상정렬 결과는 그래프의 각 간선  $\langle u, v \rangle$ 에 대해  $u$ 가  $v$ 보다 반드시 앞서 나열되어야 함

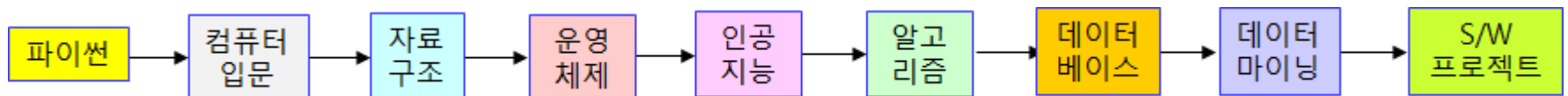
[예제] (a)는 교과과정의 선수과목 관계도이고, (c)는 교과목 수강순서도이다.



(a) 선수과목 관계도



(b) 그래프



(c) 교과목 수강 순서

- 주어진 그래프에 따라 여러 개의 위상 정렬이 존재할 수 있음
- 일반적으로 작업(Task)들 사이에 의존관계가 존재할 때 수행 가능한 작업 순서를 도식화하는데에 위상 정렬을 사용
- 위상 정렬 찾기
  1. 그래프에서 진입 차수가 0인 정점  $v$ 로부터 시작하여  $v$ 를 출력하고  $v$ 를 그래프에서 제거하는 과정을 반복하는 순방향 방법
  2. 진출 차수가 0인 정점  $v$ 를 출력하고  $v$ 를 그래프에서 제거하는 과정을 반복하여 얻은 출력 리스트를 역순으로 만들어 결과를 얻는 역방향 방법

- 순방향 방법은 각 정점의 진입 차수를 알아야 하므로 인접리스트를 각 정점으로 진입하는 정점들의 리스트로 바꾸어야
- 역방향 방법은 주어진 인접리스트를 입력에 대해 변형된 DFS를 수행하여 출력 리스트를 작성한 후에 리스트를 역순으로 만들어 위상 정렬 결과를 얻음

[핵심 아이디어] DFS를 수행하며 각 정점  $v$ 의 인접한 모든 정점들의 방문이 끝나자마자  $v$ 를 리스트에 추가한다. 리스트가 완성되면 리스트를 역순으로 만든다

- “ $v$ 의 인접한 모든 정점들의 방문이 끝나자마자  $v$  를 리스트에 추가한다”

⇒  $v$ 가 추가되기 전에  $v$ 에 인접한 모든 정점들이 이미 리스트에 추가되어 있음을 뜻함

- 따라서 리스트가 완성되어 이를 역순으로 만들면 위상 정렬 결과를 얻음

```

01 adj_list = [[1], [3, 4], [0, 1],
02             [6], [5], [7], [7], [8], []]
03 N = len(adj_list)
04 visited = [None] * N
05 s = []
06
07 def dfs(v):
08     visited[v] = True
09     for w in adj_list[v]:
10         if not visited[w]:
11             dfs(w)
12     s.append(v)
13
14 for i in range(N):
15     if not visited[i]:
16         dfs(i)
17 s.reverse()
18 print('위상정렬: ')
19 print(s)

```

그래프 인접리스트

정점 방문 여부 확인 용

위상정렬 결과  
리스트 초기화

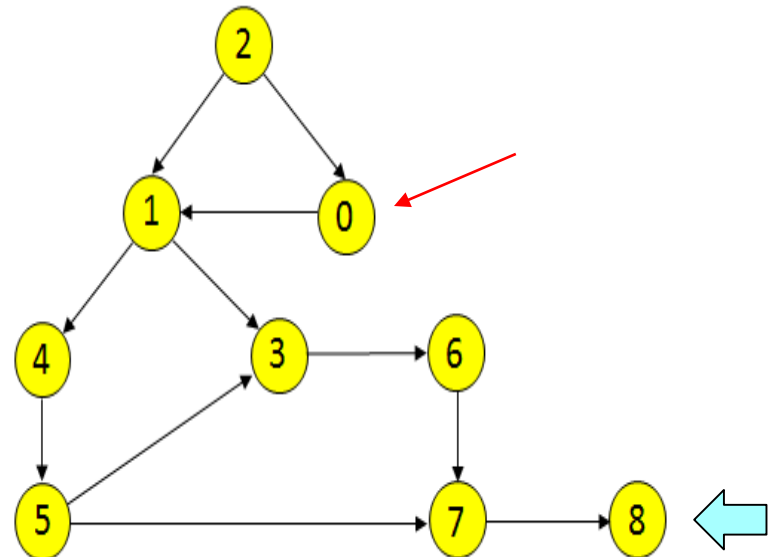
정점 v의 모든 인접한 정점들이  
방문되었으므로 정점 v 추가

dfs() 호출로 시작

s의 역순으로 위상정렬  
결과 얻음

[프로그램 8-4] topological\_sort.py


먼저 line 16의 **dfs(0)**으로 시작하여, dfs(1), dfs(3), dfs(6), dfs(7)을 차례로 호출한 후에, dfs(8)이 호출된다. 이 때 정점 8에선 더 이상 인접한 정점이 없으므로 line 09의 for-루프가 더 이상 수행되지 않고 바로 line 12의 s.append(8)이 수행되어 **정점 8**이 s에 가장 먼저 저장된다. 즉, 위상정렬순서의 가장 마지막 정점을 찾아서 s에 저장한 것이다.



- s.append(8)이 수행된 후 dfs() 함수가 리턴된 뒤, 정점 7에 대해 line 09의 for-루프에서 정점 7의 인접한 모든 정점들을 이미 방문했으므로, line 12에서 '7'을 s에 추가
- 이를 계속하여 line 14의 for-루프에서 더 이상 방문 안된 인접한 정점이 없으면 line 17에서 s를 역순으로 만들어 line 19에서 위상정렬을 출력



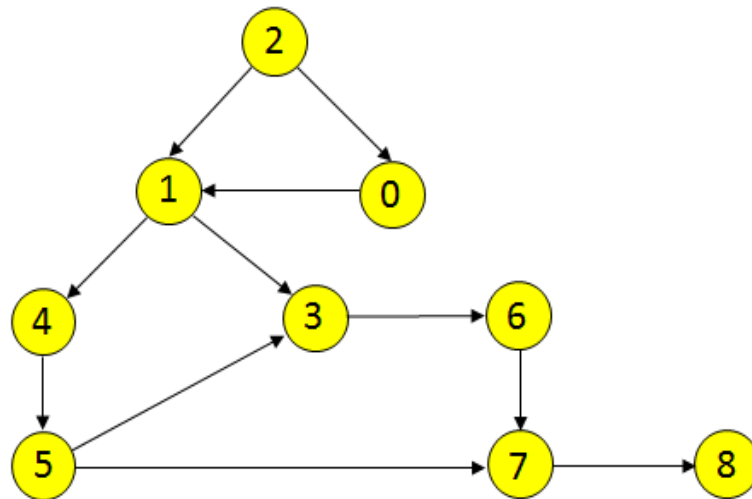
## 프로그램 수행 결과

Console  PyUnit

<terminated> topological\_sort.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\

위상정렬:

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

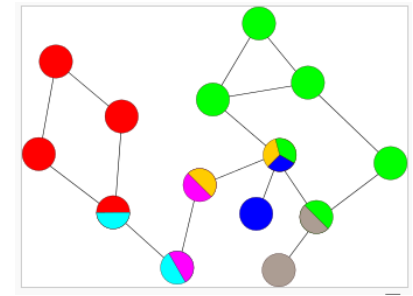


# 수행 시간

- 위상 정렬 알고리즘의 수행 시간은 DFS의 수행 시간과 동일한  $O(N+M)$
- 기본적으로 DFS를 수행하며 추가로 소요되는 시간은 line 12에서 정점을 리스트에 저장하고, 모든 탐색이 끝나면 리스트를 역순으로 만드는 시간으로 이는  $O(N)$
- 따라서 위상 정렬 알고리즘의 수행 시간은

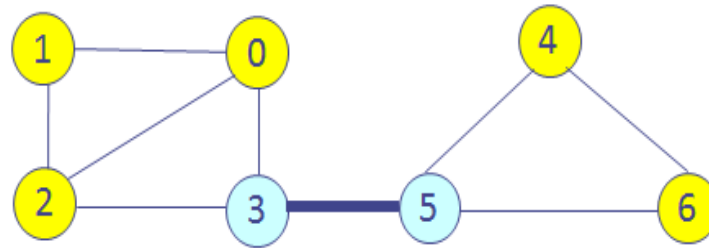
$$O(N+M) + O(N) = O(N+M)$$

# 이중 연결성분



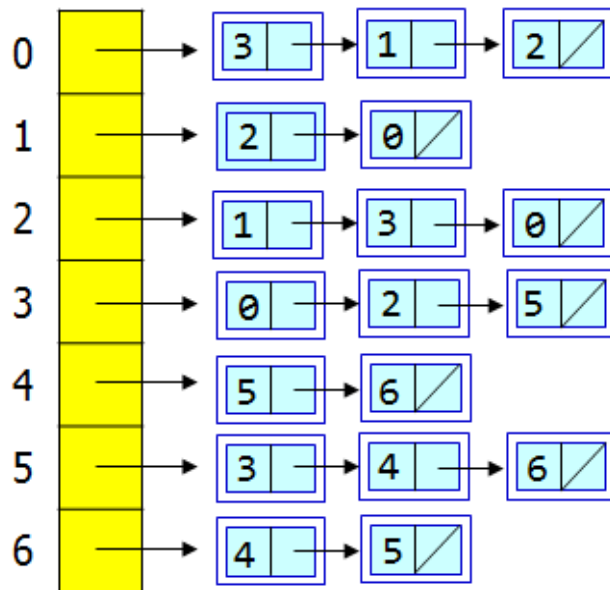
- 이중 연결성분(Biconnected Component): 무방향 그래프의 연결성분에서 임의의 두 정점들 사이에 적어도 두 개의 단순 경로가 존재하는 연결성분
- 따라서 하나의 단순 경로 상의 어느 정점 하나가 삭제되더라도 삭제된 정점을 거치지 않는 또 다른 경로가 존재하므로 연결성분내에서 정점들 사이의 연결이 유지
- 이중연결성분은 통신 네트워크 보안, 전력 공급 네트워크 등에서 네트워크의 견고성(Robustness)을 분석하는 주된 방법

- 단절 정점(Articulation Point 또는 Cut Point):  
연결성분의 정점들 중 하나의 정점을 삭제했을 때, 두 개 이상의 연결성분들로 분리될 때 삭제된 정점
- 다리 간선(Bridge): 간선을 제거했을 때 두 개 이상의 연결성분들로 분리될 때 삭제된 간선

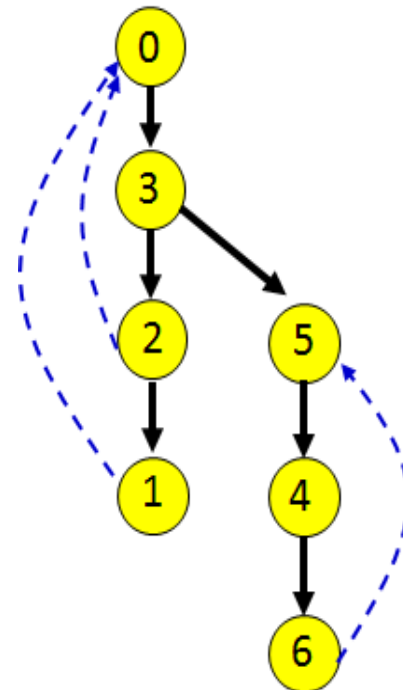


- 정점 3과 5는 각각 단절 정점
- 간선 (3, 5)는 다리 간선
- 위 그래프는 3 개의 이중연결성분,  $[0, 1, 2, 3]$ ,  $[3, 5]$ ,  $[4, 5, 6]$ 으로 구성
- 단절 정점은 이웃한 이중연결성분들에 동시에 속하고, 다리 간선은 그 자체로 하나의 이중연결성분

- 이중 연결성분을 찾는 알고리즘을 알아보기 전에 DFS를 수행하며 만들어지는 DFS 신장 트리와 이중연결성분과의 관계를 살펴보자.
- 그래프에 대한 인접리스트가 (a)와 같다면, 정점 0에서부터 DFS를 수행하면 (b)와 같은 신장 트리를 얻음



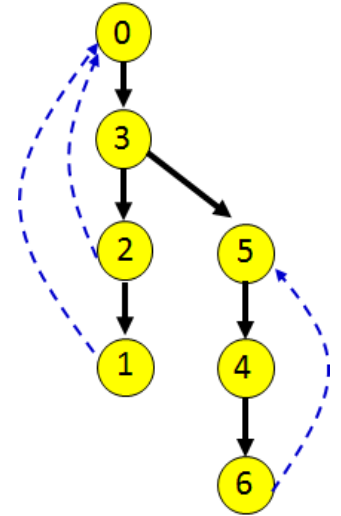
(a)



(b)

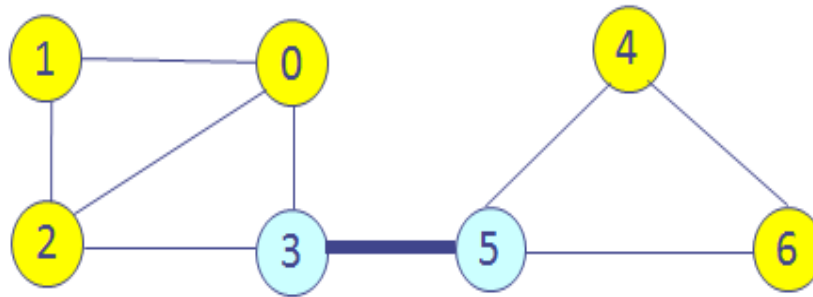
- (b)의 트리에서 점선으로 표시된 각각의 뒷간선은 싸이클을 만든다.

- 뒷간선 (2, 0)을 신장트리에 추가하면 [0-3-2-0]의 싸이클을 만들고,
- 뒷간선 (1, 0)은 [0-3-2-1-0]의 싸이클을 만들며,
- 뒷간선 (6, 5)는 [5-4-6-5]의 싸이클 형성



- 이중연결성분은 성분내의 정점들 사이에 적어도 2개의 단순 경로가 있어야 하므로, 뒷간선으로 만들어지는 싸이클 상의 정점들은 하나의 이중연결성분에 속함

- 다음 그래프는  $[(0, 3), (3, 2), (2, 1), (1, 0), (2, 0)]$ ,  $[(3, 5)]$ ,  $[(5, 4), (4, 6), (6, 5)]$ 의 **3개의 이중 연결성분**으로 구성되고 정점 3과 5가 단절정점이다.

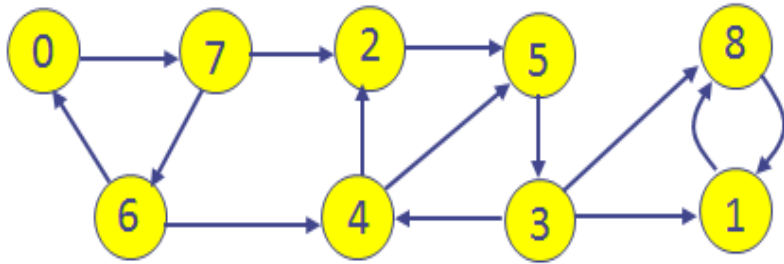


이중연결성분을 찾는 알고리즘의 수행시간은  
깊이우선탐색의 수행시간과 동일한  $O(N+M)$ 이다.

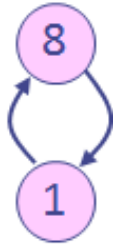
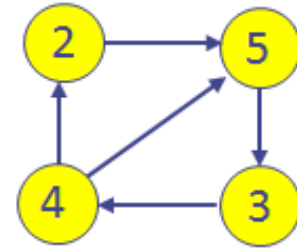
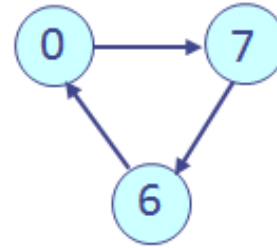
# 강연결성분

- 강연결성분(Strongly Connected Component):  
방향그래프에서 연결성분 내의 임의의 두 정점  $u$ 와  $v$ 에 대해 정점  $u$ 에서  $v$ 로 가는 경로가 존재하고 동시에  $v$ 에서  $u$ 로 돌아오는 경로가 존재하는 연결성분
- 강연결성분은 단절정점이나 다리 간선을 포함하지 않는다.
- 강연결성분은 소셜네트워크에서 커뮤니티(Community)를 분석하는데 활용되며, 인터넷의 웹 페이지 분석에도 사용된다.





그래프



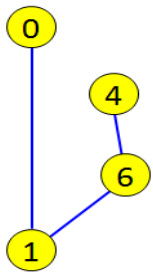
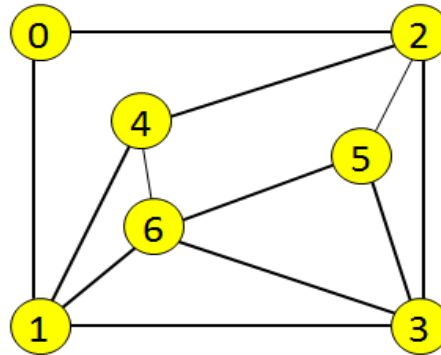
3개의 강연결성분

- 강연결성분은 소셜네트워크에서 커뮤니티(Community)를 분석하는데 활용되며, 인터넷의 웹 페이지 분석에도 사용
- 강연결성분 찾는 알고리즘의 수행시간은 깊이우선탐색의 수행시간과 동일한  $O(N+M)$

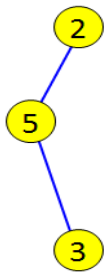
## 8.4 최소신장트리

- 최소 신장 트리(Minimum Spanning Tree, MST): 하나의 연결성분으로 이루어진 무방향 가중치 그래프에서 간선의 가중치의 합이 최소인 신장 트리
- MST를 찾는 대표적인 알고리즘은 Kruskal, Prim, Sollin 알고리즘 - 모두 그리디 (Greedy) 알고리즘
- 그리디 알고리즘은 최적해(최솟값 또는 최댓값)를 찾는 문제를 해결하기 위한 알고리즘 방식들 중 하나로서, 알고리즘의 선택이 항상 '욕심내어' 지역적인 최솟값(또는 최댓값)을 선택하며, 이러한 부분적인 선택을 축적하여 최적해를 찾음

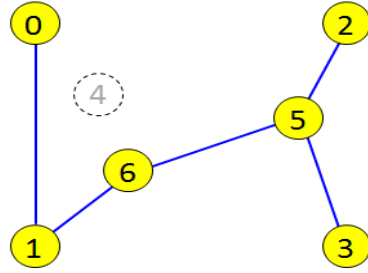
어느 그래프가 신장 트리일까?



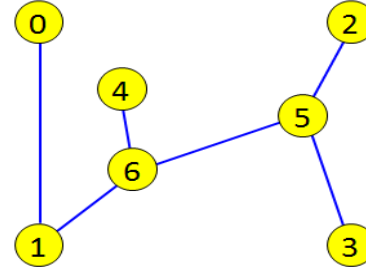
(b)



(c)



(d)



(e)

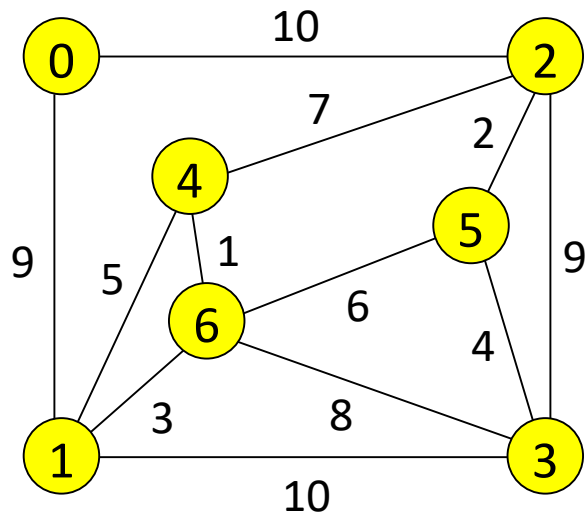
## 8.4.1 Kruskal 알고리즘

- 간선들을 가중치가 감소하지 않는 순서로 정렬한 후 가장 가중치가 작은 간선을 트리에 추가하여 싸이클을 만들지 않으면 트리 간선으로 선택하고, 싸이클을 만들면 버리는 일을 반복하여  $N-1$ 개의 간선이 선택되었을 때 알고리즘을 종료
  - $N$ 은 그래프 정점의 수
- Kruskal 알고리즘이 그리디 알고리즘인 이유: 남아있는 (정렬된) 간선들 중에서 항상 '욕심 내어' 가중치가 가장 작은 간선을 가져오기 때문

## Kruskal 알고리즘

- [1] 가중치가 감소하지 않는 순서로 간선 리스트  $L$ 을 만든다.
- [2] **while** 트리의 간선 수  $< N-1$ :
- [3]     $L$ 에서 가장 작은 가중치를 가진 간선  $e$ 를 가져오고,  
       $e$ 를  $L$ 에서 제거
- [4]    **if** 간선  $e$ 가  $T$ 에 추가하여 사이클을 만들지 않으면:
- [5]        간선  $e$ 를  $T$ 에 추가

# [예제]



## 정렬된 L

(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

(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

(4, 6) 1

(2, 5) 2

(1, 6) 3

(3, 5) 4

(1, 4) 5

(5, 6) 6

(2, 4) 7

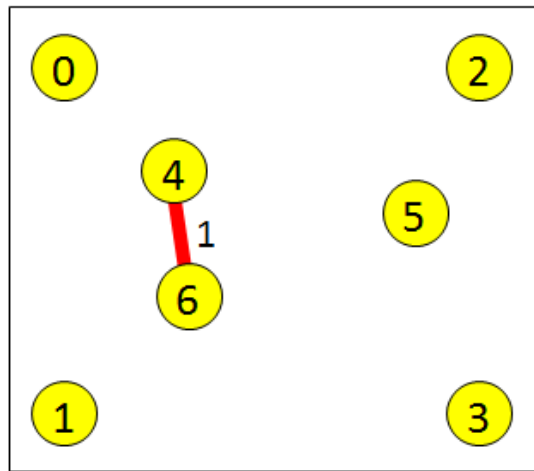
(3, 6) 8

(0, 1) 9

(2, 3) 9

(0, 2) 10

(1, 3) 10



(2, 5) 2

(1, 6) 3

(3, 5) 4

(1, 4) 5

(5, 6) 6

(2, 4) 7

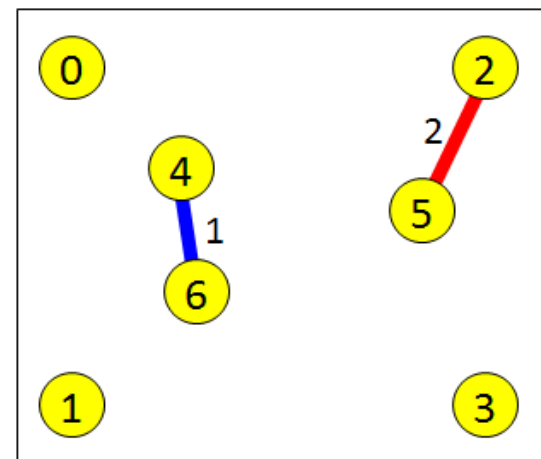
(3, 6) 8

(0, 1) 9

(2, 3) 9

(0, 2) 10

(1, 3) 10



(1, 6) 3

(3, 5) 4

(1, 4) 5

(5, 6) 6

(2, 4) 7

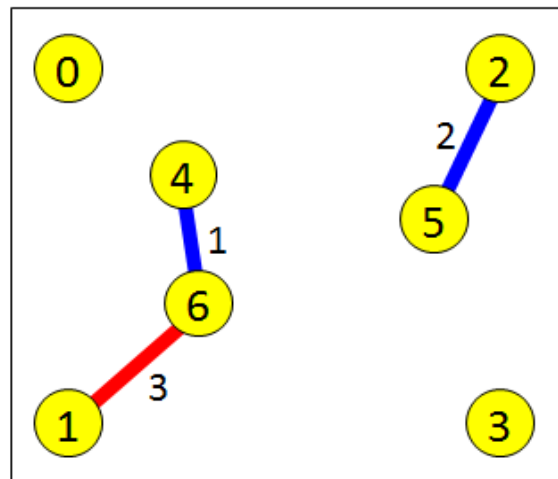
(3, 6) 8

(0, 1) 9

(2, 3) 9

(0, 2) 10

(1, 3) 10



(3, 5) 4

(1, 4) 5

(5, 6) 6

(2, 4) 7

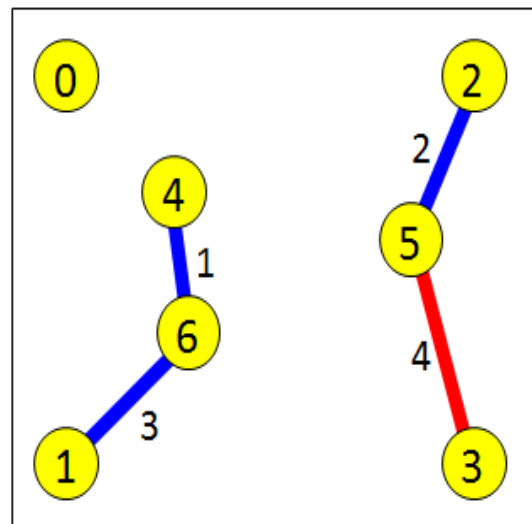
(3, 6) 8

(0, 1) 9

(2, 3) 9

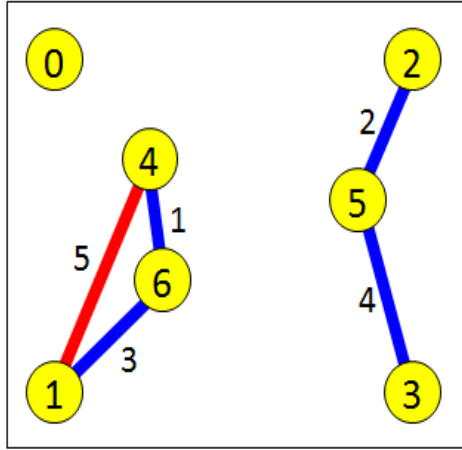
(0, 2) 10

(1, 3) 10

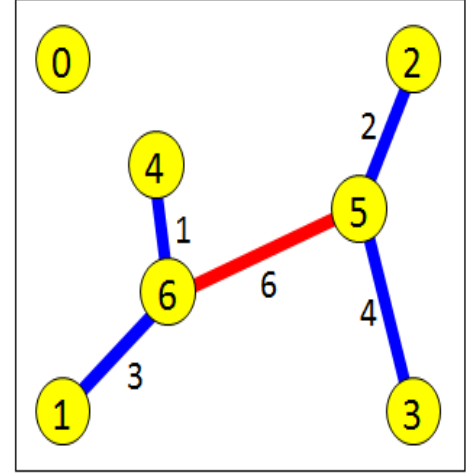




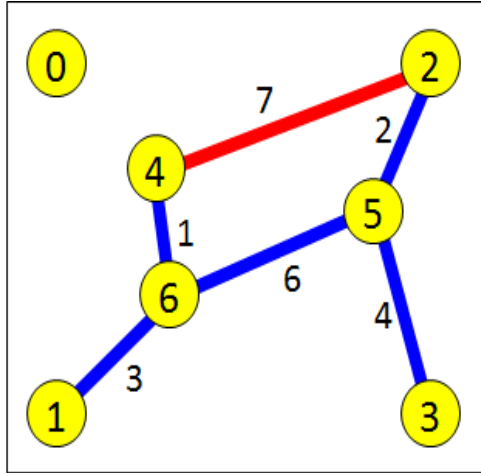
(1, 4) 5  
(5, 6) 6  
(2, 4) 7  
(3, 6) 8  
(0, 1) 9  
(2, 3) 9  
(0, 2) 10  
(1, 3) 10



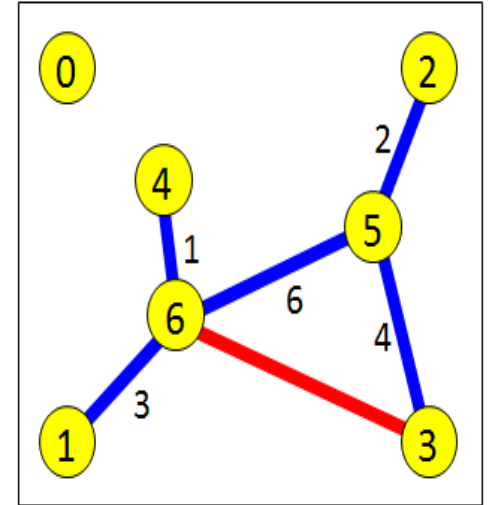
(5, 6) 6  
(2, 4) 7  
(3, 6) 8  
(0, 1) 9  
(2, 3) 9  
(0, 2) 10  
(1, 3) 10



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



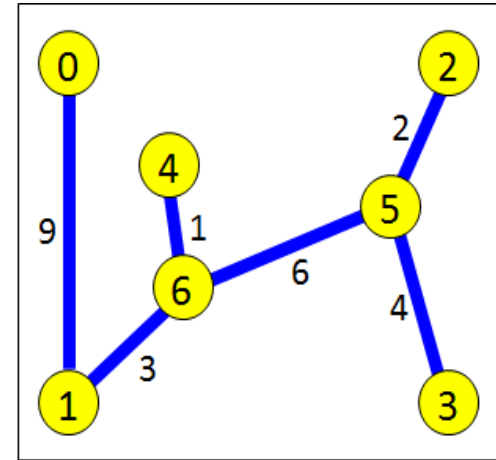
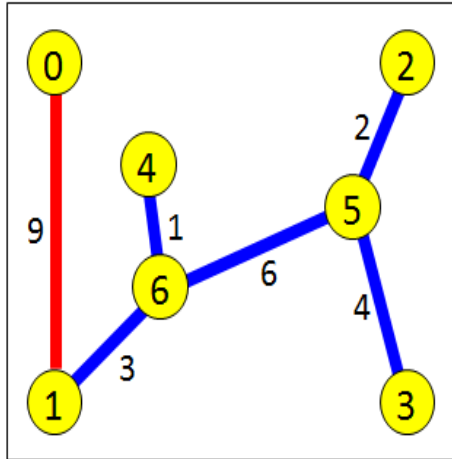
(3, 6) 8  
(0, 1) 9  
(2, 3) 9  
(0, 2) 10  
(1, 3) 10







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



최소신장트리

최소신장트리의 간선의 가중치의 합 =  $1 + 2 + 3 + 4 + 6 + 9 = 25$

입력 그래프  
(간선의 두 정점, 가중치)

```
01 weights = [(0, 1, 9), (0, 2, 10), (1, 3, 10), (1, 4, 5),  
02             (1, 6, 3), (2, 3, 9), (2, 4, 7), (2, 5, 2),  
03             (3, 5, 4), (3, 6, 8), (4, 6, 1), (5, 6, 6)]
```

```
04 weights.sort(key = lambda t: t[2])
```

가중치로 간선 정렬

```
05 mst = []
```

```
06 N = 7
```

상호 배타적 집합

```
07 p = [0] * N
```

```
08 for i in range(N):
```

```
09     p.append(i)
```

각 정점 자신이 집합의 대표(루트)

```
10
```

```
11 def find(u):
```

find 연산

```
12     if u != p[u]:
```

```
13         p[u] = find(p[u])
```

경로압축

```
14     return p[u]
```

```
15
```

```
16 def union(u, v):
```

union 연산

```
17     root1 = find(u)
```

```
18     root2 = find(v)
```

```
19     p[root2] = root1
```

임의로 root2가  
root1의 부모가 됨

```
20
```

```
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)
```

다음 최소 가중치를  
가진 간선 가져오기

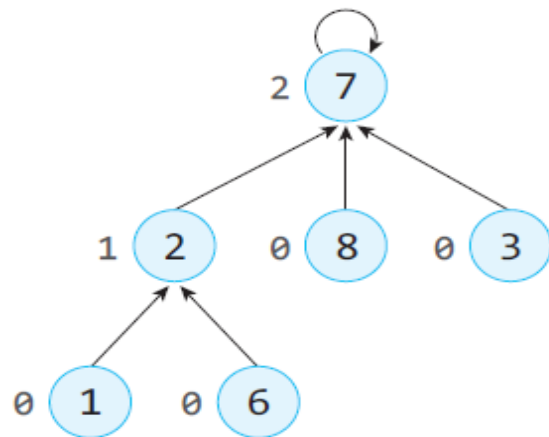
u와 v가 서로 다른  
집합에 속해 있으면

트리에 (u, v) 추가

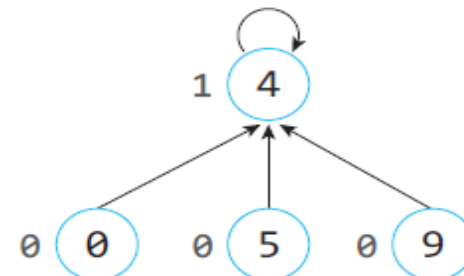
[프로그램 8-5] kruskal.py

- Kruskal 알고리즘에서 추가하려는 간선이 싸이클을 만드는지의 여부는 집합과 관련된 연산인 union(합집합) 연산과 주어진 원소에 대해 어느 집합에 속해 있는지를 찾는 find 연산을 사용
- 특히 어느 두 집합도 중복된 원소를 갖지 않는 경우, 이러한 집합들을 상호 배타적 집합(Disjoint Set)이라고 한다.
- 상호 배타적 집합들은 1차원 리스트로 표현 가능하며, 모든 집합들의 원소를 0, 1, 2, ..., N-1로 놓으면 이를 리스트의 인덱스로 활용할 수 있기 때문이다.

- [그림 8-22]는 2개의 상호 배타적 집합을 일반적인 트리 형태로 표현하여 리스트에 저장된 상태를 나타낸다.
- 여기서 각 집합은 루트가 대표하고, 루트의 리스트 원소에는 루트 자신이 저장되며, 루트가 아닌 노드의 원소에는 부모노드가 저장된다.



집합 {7, 2, 8, 3, 1, 6}



집합 {4, 0, 5, 9}

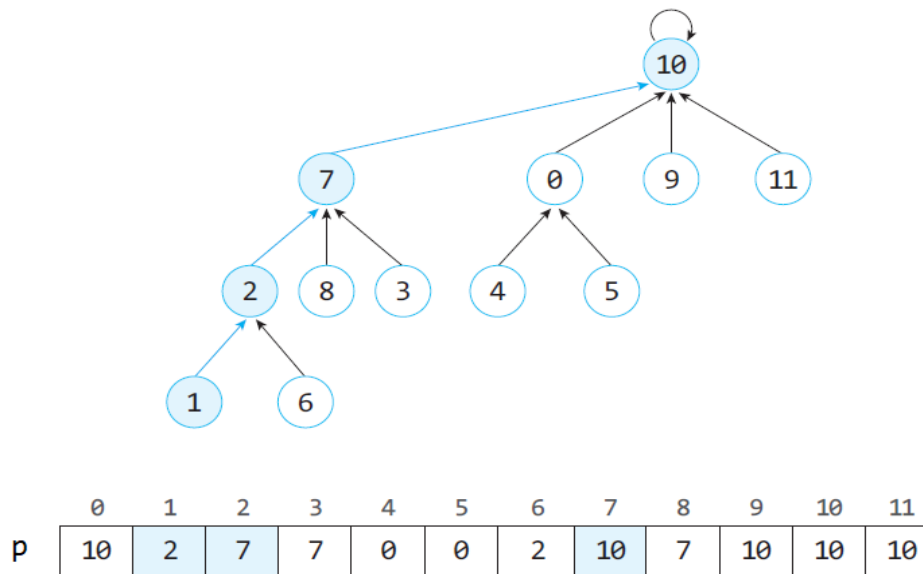
[그림 8-22]

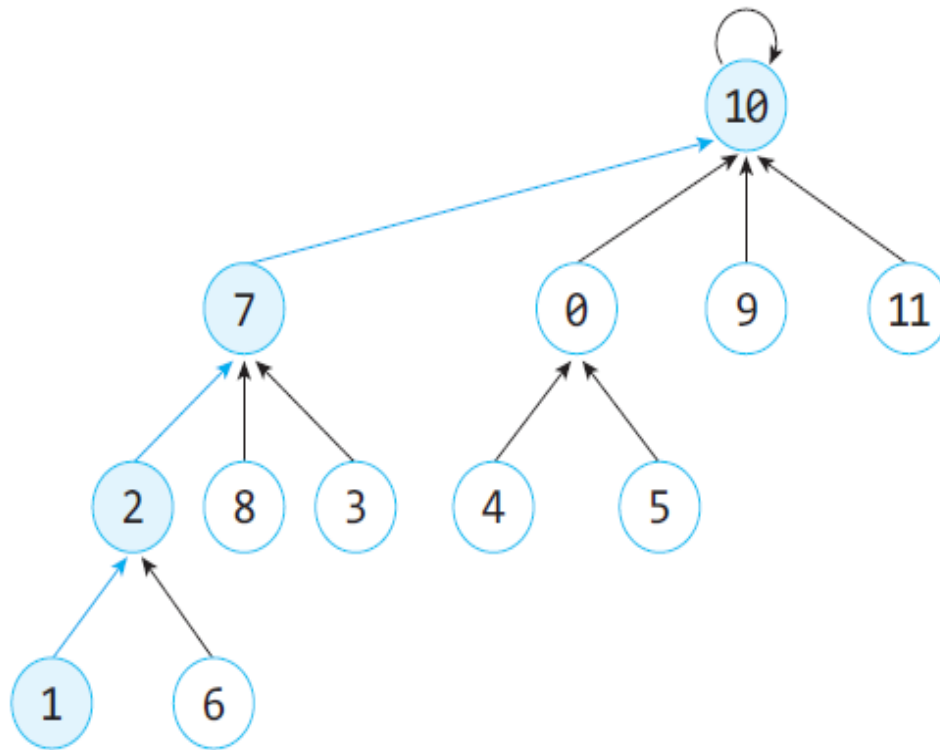
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | 4 | 2 | 7 | 7 | 4 | 4 | 2 | 7 | 7 | 4 |

- 상호 배타적 집합에 대해 수행할 연산은 union과 find 연산 중에서 union 연산은 2개의 집합을 하나의 집합으로 만드는 연산이다.
- **find 연산**은 인자로 주어지는  $x$ 가 속한 집합의 대표 노드, 즉, 루트를 찾는 연산이다.
- $\text{find}(6)$ 은  $p[6] = 2$ 를 통해 6의 부모인 2를 찾고,  $p[2] = 7$ 로 2의 부모를 찾으며, 마지막으로  $p[7] = 7$ 이기 때문에 7을 리턴
- 즉, "6은 7이 대표 노드인 집합에 속해 있다"는 것을 리턴한
- $\text{find}(3)$ 도 7을 리턴하므로, 6과 3은 동일한 집합에 속함. 하지만  $\text{find}(9) = 4$ 이므로, 6과 9는 서로 다른 집합에 속함

- find 연산을 효율적으로 수행하기 위해 **경로압축**을 수행하는데 이는 나중에 수행되는 find 연산을 더 빠르게 수행하기 위해서이다.

[핵심 아이디어] find 연산을 수행하면서 루트까지 올라가는 경로 상의 각 노드의 부모를 루트로 갱신함 이를 **경로압축(Path Compression)**이라고 한다.



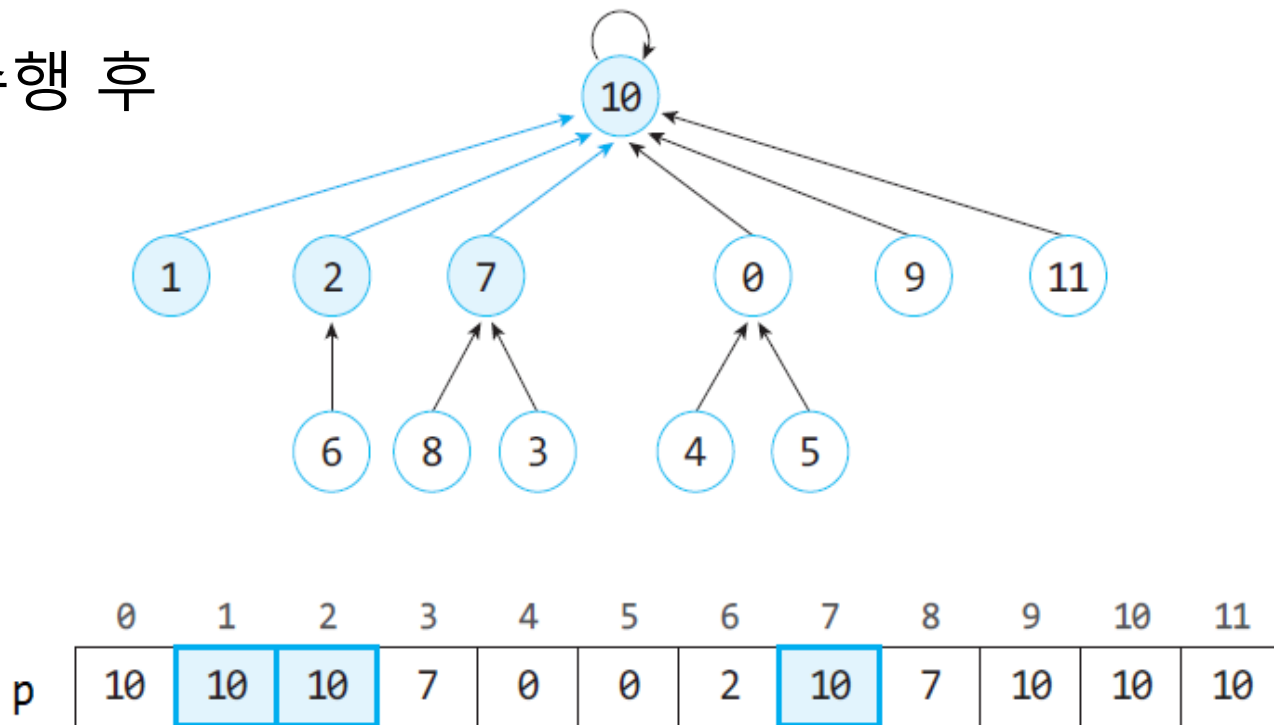


|   | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8 | 9  | 10 | 11 |
|---|----|---|---|---|---|---|---|----|---|----|----|----|
| p | 10 | 2 | 7 | 7 | 0 | 0 | 2 | 10 | 7 | 10 | 10 | 10 |

find(1) 수행 전




find(1) 수행 후



- find(1)을 수행하면 루트인 10까지 올라가는 경로 상의 모든 노드 1, 2, 7의 부모노드를 10으로 갱신하여 (b)와 같은 트리를 만든다.
- 이러한 경로 압축은 당장 find(1)의 수행시간이 줄어들지는 않으나, 추후의 find(1)과 find(2)의 수행시간을 단축

- [프로그램 8-5]의 line 26에서는 남아있는 간선들 중에서 가장 작은 가중치를 가진 간선  $(u, v)$ 를 가져와서  $\text{find}(u)$ 와  $\text{find}(v)$ 를 수행
- $u$ 와  $v$ 가 다른 집합에 속하면 간선  $(u, v)$ 가 싸이클을 만들지 않으므로 line 29에서  $(u, v)$ 를 트리에 추가하고
- 같은 집합에 속하면 간선  $(u, v)$ 를 버리고 line 23의 while-루프가 수행

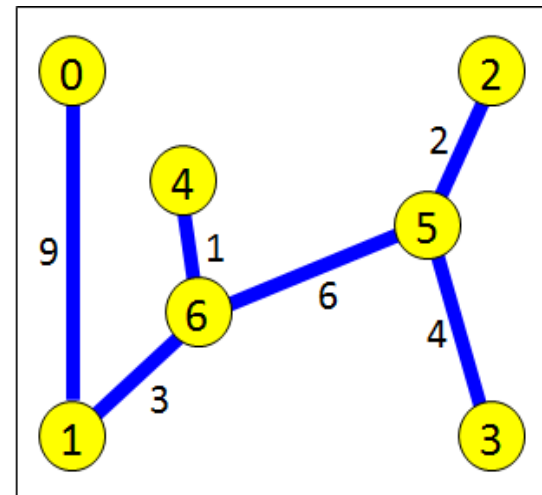
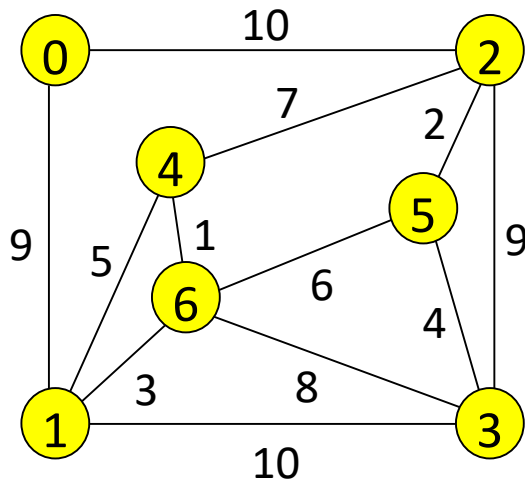
# 프로그램 수행 결과

Console  PyUnit

<terminated> kruskal.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\

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

최소신장트리 가중치: 25



# 수행시간

- 간선을 정렬(또는 우선순위큐의 삽입과 삭제)하는데 소요되는 시간인  $O(M \log M) = O(M \log N)$ 과 트리에 간선을 추가하려 할 때 find와 union을 수행하는 시간인  $O((M+N) \log^* N)$ 의 합이다.
- 즉,  $O(M \log N) + O((M+N) \log^* M) = O(M \log N)$ 이다.
- union 연산은 단순히 하나의 루트가 다른 루트의 자식이 되는 것이므로  $O(1)$  시간 소요
- 실제로 조금 복잡한 분석 방법을 통해서 union과 find 연산들의 수행 시간인  $O((M+N) \log^* N)$ 이 계산된다. 상세한 분석은 생략함

## 8.4.2 Prim 알고리즘

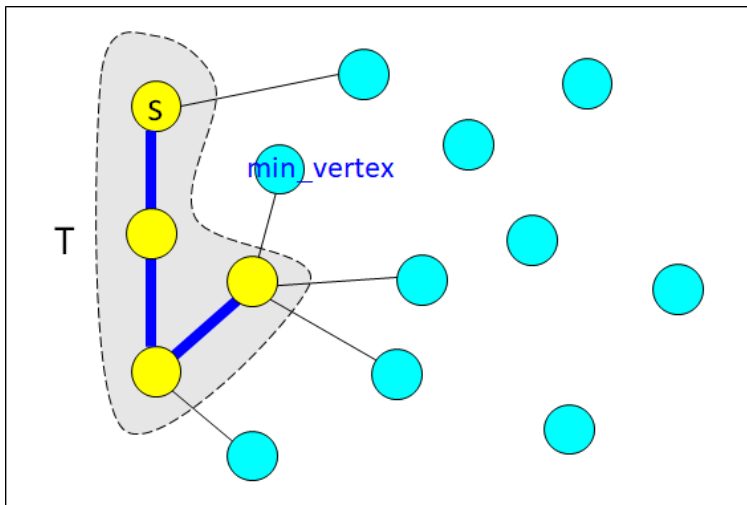
- Prim 알고리즘은 임의의 시작 정점에서 가장 가까운 정점을 추가하여 간선이 하나의 트리를 만들고, 만들어진 트리에 인접한 가장 가까운 정점을 하나씩 추가하여 최소신장트리를 만든다.
- Prim의 알고리즘에서는 초기에 트리  $T$ 는 임의의 정점  $s$ 만을 가지며, 트리에 속하지 않은 각 정점과  $T$ 의 정점(들)에 인접한 간선들 중에서 가장 작은 가중치를 가진 간선의 끝점을 찾기 위해 리스트  $D$ 를 사용

## Prim 알고리즘

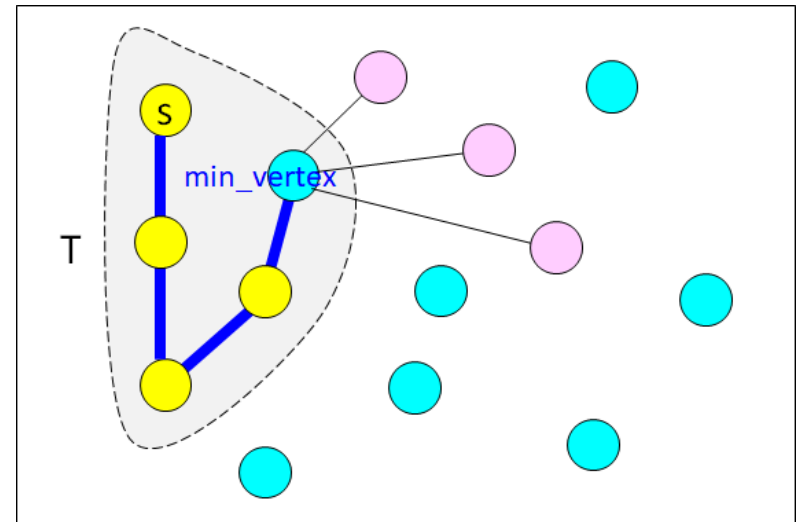
- [1] D를  $\infty$ 로 초기화한다. 시작 정점  $s$ 의  $D[s] = 0$
- [2] **while** T의 정점 수  $< N$ :
- [3]     T에 속하지 않은 각 정점  $i$ 에 대해  $D[i]$ 가 최소인 정점  $\text{min\_vertex}$ 를 찾아 T에 추가
- [4]     **for** T에 속하지 않은 각 정점  $w$ 에 대해서:
- [5]         **if** 간선  $(\text{min\_vertex}, w)$ 의 가중치  $< D[w]$ :
- [6]              $D[w] =$  간선  $(\text{min\_vertex}, w)$ 의 가중치

## Prim 알고리즘의 step [3]~[6]

- (a) 트리에 가장 가까운 정점  $\text{min\_vertex}$ 를 찾아(트리 밖에 있는 정점들의 D의 원소들 중에서 최솟값을 찾아)
- (b) 트리에 추가한 후, 정점  $\text{min\_vertex}$ 에 인접하면서 트리에 속하지 않은 각 정점의 D 원소가 이전 값보다 작으면 갱신

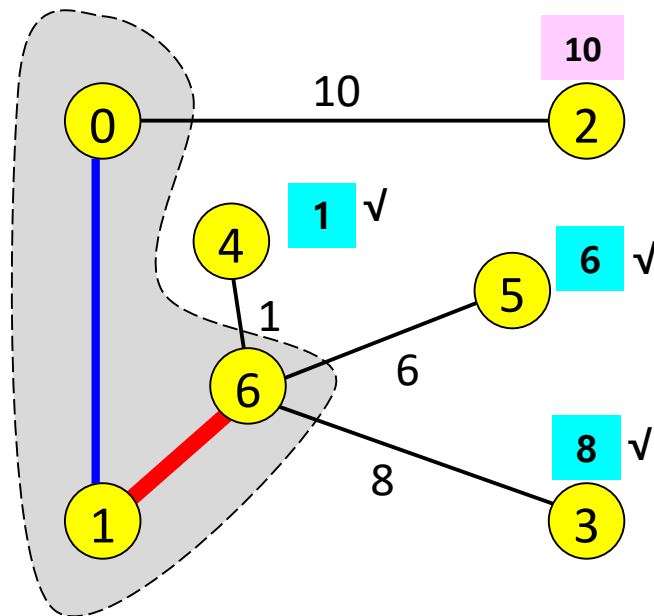
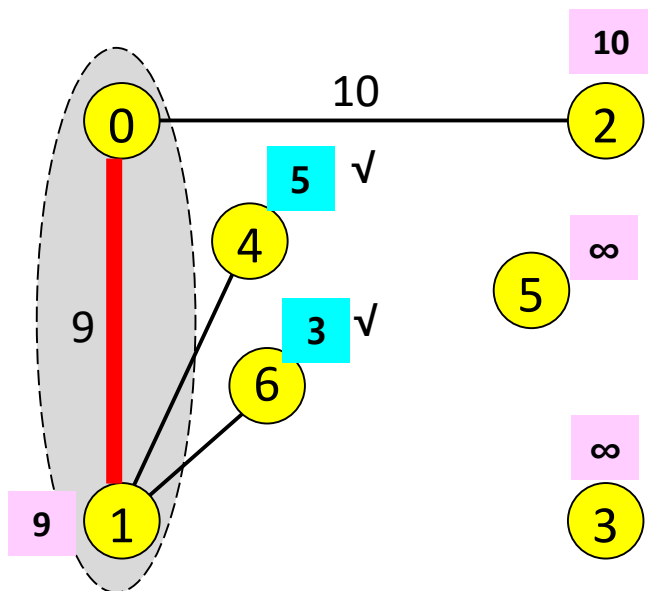
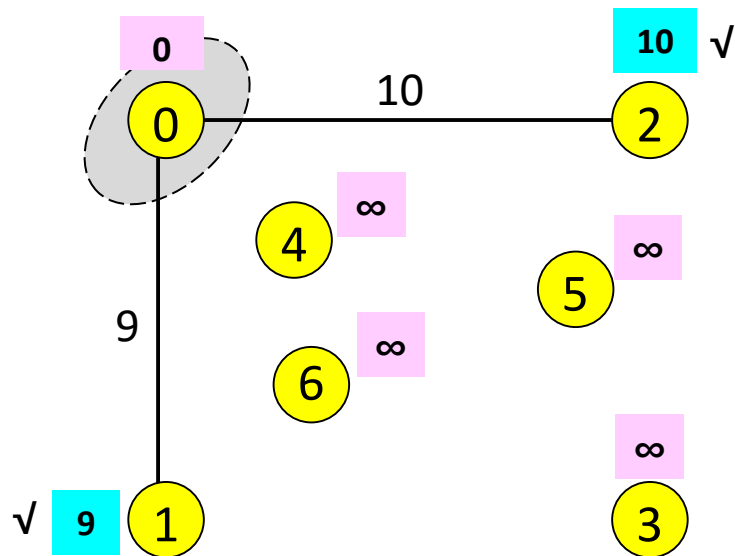
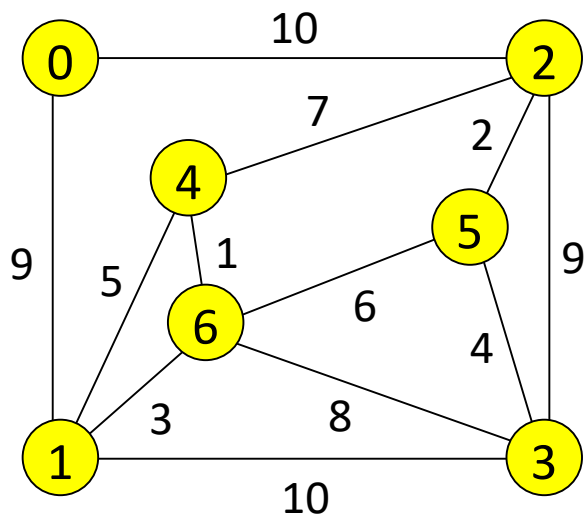


(a)

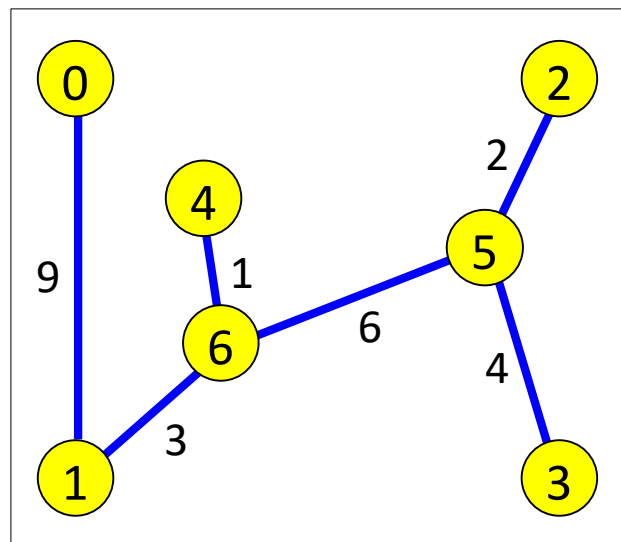
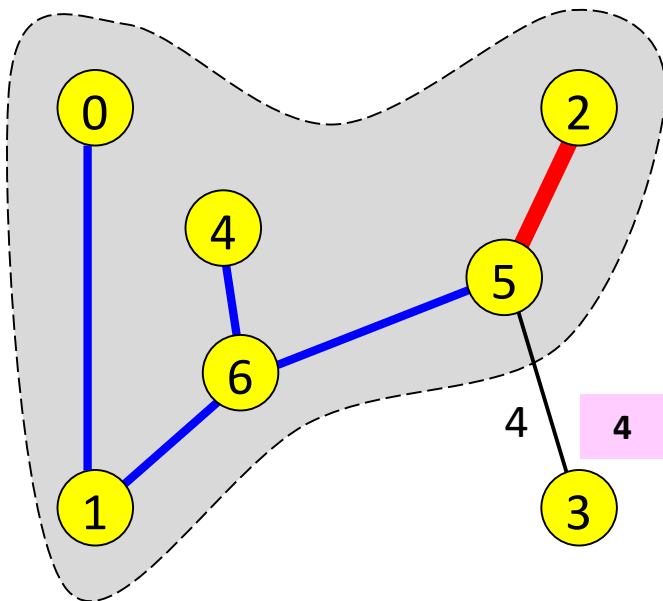
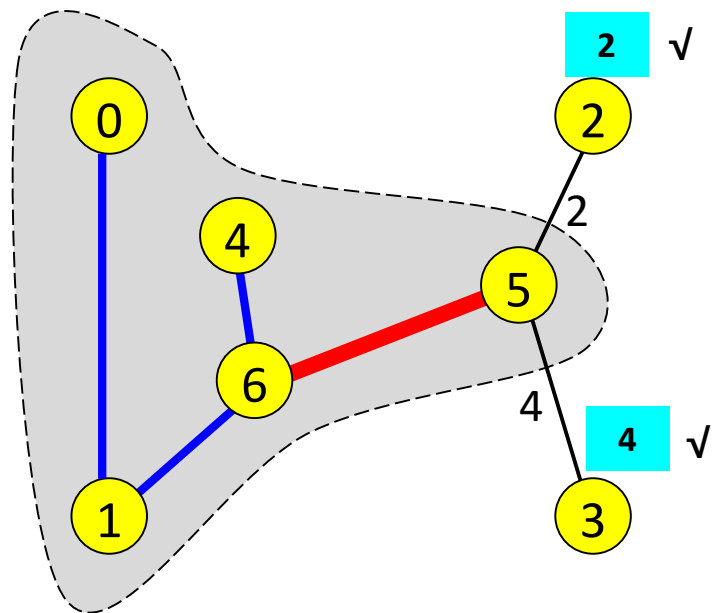
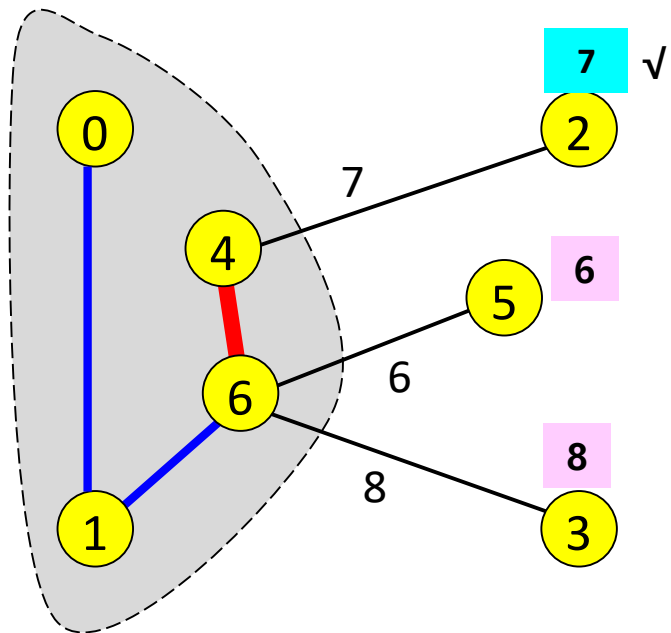


(b)

# [예제]







```

01 import sys
02 N = 7
03 s = 0
04 g = [None] * N
05 g[0] = [(1, 9), (2, 10)]
06 g[1] = [(0, 9), (3, 10), (4, 5), (6, 3)]
07 g[2] = [(0, 10), (3, 9), (4, 7), (5, 2)]
08 g[3] = [(1, 10), (2, 9), (5, 4), (6, 8)]
09 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 visited = [False] * N
14 D = [sys.maxsize] * N
15 D[s] = 0
16 previous = [None] * N
17 previous[s] = s
18

```

sys.maxsize(최댓값) 사용 위해

입력 그래프의  
인접리스트

각 원소를 최댓값으로

초기화

트리 간선 추출을 위해

```

19 for k in range(N):
20     m = -1
21     min_value = sys.maxsize
22     for j in range(N):
23         if not visited[j] and D[j] < min_value:
24             min_value = D[j]
25             m = j
26     visited[m] = True
27
28     for w, wt in list(g[m]):
29         if not visited[w]:
30             if wt < D[w]:
31                 D[w] = wt
32                 previous[w] = m
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 += D[i]
39 print('\n최소신장트리 가중치: ', mst_cost)

```

m = min\_vertex

방문 안된 정점들의 D  
원소들 중에서 최솟값을  
가진 정점 m 찾기


정점 m에 인접한 정점 w와  
간선 (m, w)의 가중치 wt에 대해

D[w] 갱신

D[w]가 정점 m 때문에  
갱신되었음을 기록

- line 16에서 리스트 previous를 선언하여 최소신장트리의 간선을 저장한다.
- 즉,  $\text{previous}[i] = j$ 라면 간선  $(i, j)$ 가 트리의 간선이다.
- Line 13~17까지는 리스트 선언 및 초기화
- Line 19의 for-루프는 N개의 정점을 트리에 추가한 뒤 종료
- Line 20~25에서는 트리에서 가장 가까운 정점 m, 즉,  $\text{min\_vertex}$ 를 찾고, line 28~31에서는  $\text{min\_vertex}$ 에 인접하면서 트리에 속하지 않은 정점의 D 원소를 갱신한다.
- Line 32에서는 m을 previous의 해당 원소에 저장

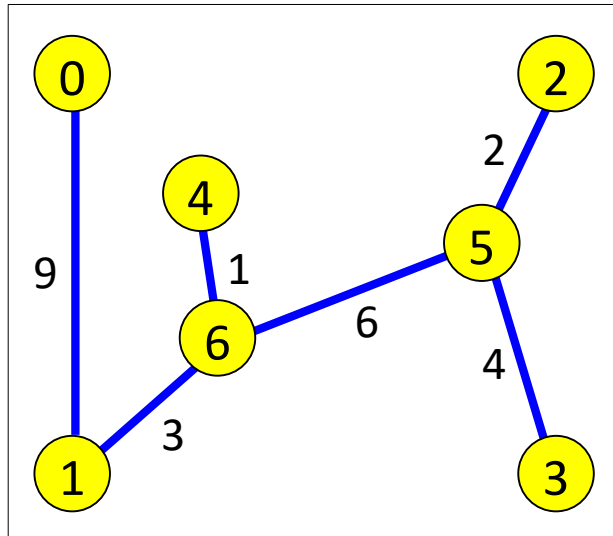
## 프로그램 수행 결과

Console  PyUnit

<terminated> prim.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\

최소신장트리: (1, 0)(2, 5)(3, 5)(4, 6)(5, 6)(6, 1)

최소신장트리 가중치: 25



# 수행 시간(1)

- Prim 알고리즘은 N번의 반복을 통해 min\_vertex를 찾고 min\_vertex에 인접하면서 트리에 속하지 않은 정점에 해당하는 D의 원소 값을 갱신
- min\_vertex를 배열 D에서 탐색하는 과정에서  $O(N)$  시간이 소요되고, min\_vertex에 인접한 정점들을 검사하여 D의 해당 원소를 갱신하므로  $O(N)$  시간이 소요된다.
- 따라서 총 수행 시간은  $N \times (O(N) + O(N)) = O(N^2)$

## 수행 시간(2)

- min\_vertex 찾기 위해 이진힙을 사용하면 각 간선에 대한 D의 원소를 갱신하며 힙 연산을 수행해야 하므로 총  $O(M \log N)$  시간이 필요, M은 그래프 간선의 수
- 이진힙은 각 정점에 대응되는 D원소를 저장하므로 힙의 최대 크기는 N
- 또한 가중치가 갱신되어 감소되었을 때의 힙 연산에는  $O(\log N)$  시간이 소요
- 입력 그래프가 희소 그래프라면, 예를 들어,  $M = O(N)$ 이라면, 수행시간이  $O(M \log N) = O(N \log N)$ 이 되어 이진힙을 사용하는 것이 매우 효율적

- min\_vertex 찾기에 피보나치 힙(Fibonacci Heap) 자료구조를 사용하면  $O(N \log N + M)$  시간에 Prim 알고리즘 수행
- 피보나치 힙은 복잡하고 구현도 쉽지 않아서 이론적인 자료구조임



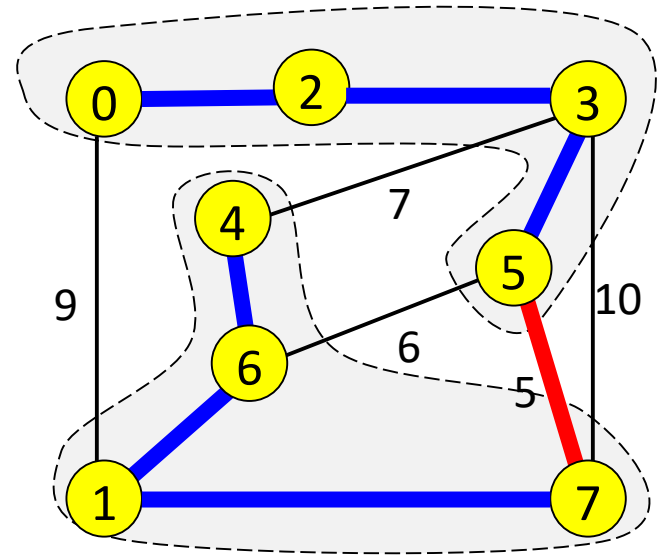
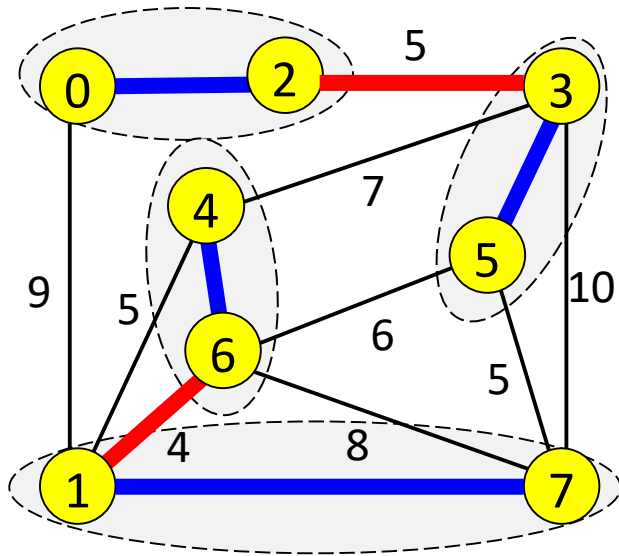
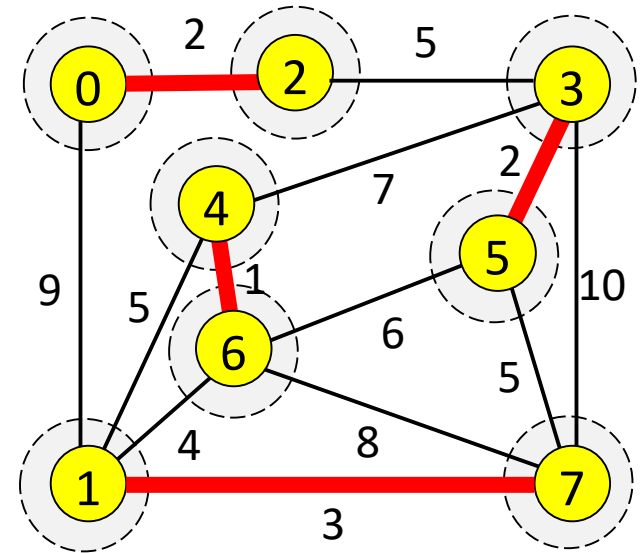
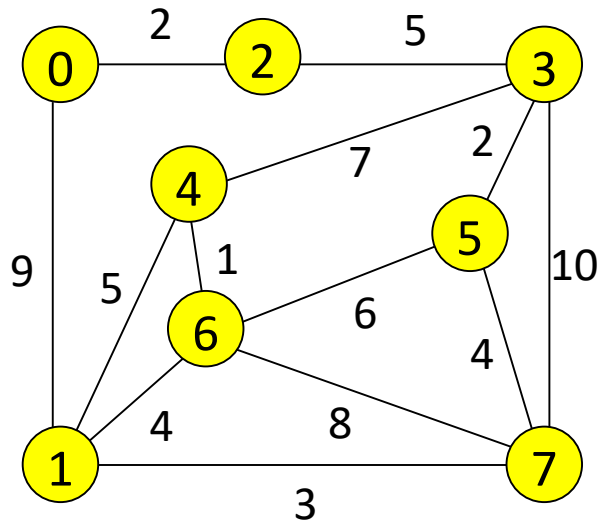
### 8.4.3 Sollin 알고리즘

- Sollin 알고리즘은 각 정점을 독립적인 트리로 간주하고, 각 트리에 연결된 간선들 중에서 가장 작은 가중치를 가진 간선을 선택한다. 이때 선택된 간선은 2 개의 트리를 1개의 트리로 만든다.
- 같은 방법으로 한 개의 트리가 남을 때까지 각 트리에서 최소 가중치 간선을 선택하여 연결
- Sollin 알고리즘은 병렬알고리즘(Parallel Algorithm)으로 구현이 쉽다는 장점을 가짐

## Sollin 알고리즘

- [1] 각 정점은 독립적인 트리이다.
- [2] repeat
- [3]    각 트리에 닿아 있는 간선들 중에서 가중치가 가장 작은 간선을 선택하여 트리를 합친다.
- [4] until (1개의 트리만 남을 때까지)

# [예제]



# 수행 시간

- Sollin 알고리즘에서 repeat-루프가 예제와 같이 각 쌍의 트리가 서로 연결된 간선을 선택하는 경우 최대  $\log N$  번 수행
- 루프 내에서는 각 트리가 자신에 닿아 있는 모든 간선들을 검사하여 최소 가중치를 가진 간선을 선택하므로  $O(M)$  시간이 소요
- 따라서 알고리즘의 수행 시간은  $O(M \log N)$

## 8.5 최단경로 알고리즘

- Dijkstra 알고리즘
- Floyd-Warshall 알고리즘

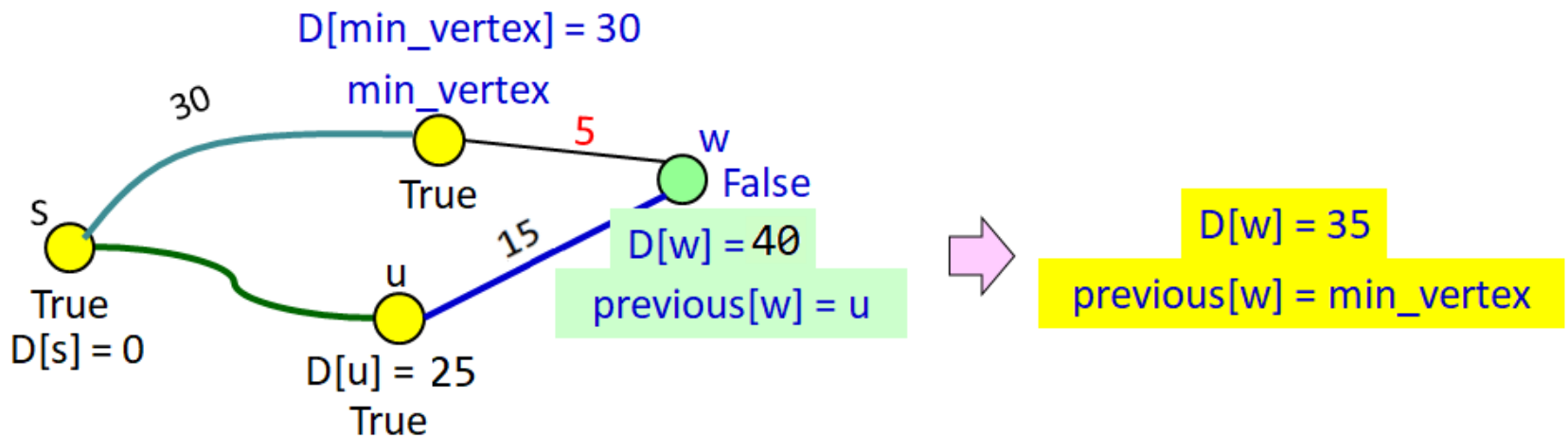
## 8.5.1 Dijkstra 알고리즘

- **최단 경로(Shortest Path)** 찾기는 주어진 가중치그래프에서 출발점으로부터 도착점까지의 최단경로를 찾는 문제
- **Dijkstra 알고리즘**: 출발점으로부터 각 정점까지의 최단거리 및 경로를 계산
- Dijkstra 알고리즘은 Prim의 MST 알고리즘과 매우 유사
- 차이점
  1. Dijkstra 알고리즘은 출발점이 주어지지지만 Prim 알고리즘에서는 출발점이 주어지지 않는다는 것
  2. Prim 알고리즘에서는 D의 원소에 간선의 가중치가 저장되지만, Dijkstra 알고리즘에서는 D의 원소에 출발점으로부터 각 정점까지의 경로의 길이가 저장됨

## Dijkstra 알고리즘

- [1] D를  $\infty$ 로 초기화한다. 단,  $D[s]=0$ 으로 초기화한다.
- [2] **for** k **in** range(N):
- [3] 방문 안된 각 정점 i에 대해  $D[i]$ 가 최소인 정점 min\_vertex를 찾고 방문한다.
- [4] **for** min\_vertex에 인접한 각 정점 w에 대해서:
- [5] **if** w 가 방문 안된 정점이면:  
wt = 간선 (min\_vertex, w)의 가중치
- [6] **if**  $D[\text{min\_vertex}] + \text{wt} < D[w]$ :
- [7]  $D[w] = D[\text{min\_vertex}] + \text{wt}$
- [8]  $\text{previous}[w] = \text{min\_vertex}$

- Step [7]의 **간선 완화(Edge Relaxation)**는 min\_vertex가 step [3]에서 선택된 후에 s로부터 min\_vertex를 경유하여 정점 w까지의 경로의 길이가 현재의  $D[w]$ 보다 더 짧아지면 짧은 길이로  $D[w]$ 를 갱신하는 것을 의미
- 그림은  $D[w]$ 가 min\_vertex 덕분에 40에서 35로 완화된 것을 나타냄



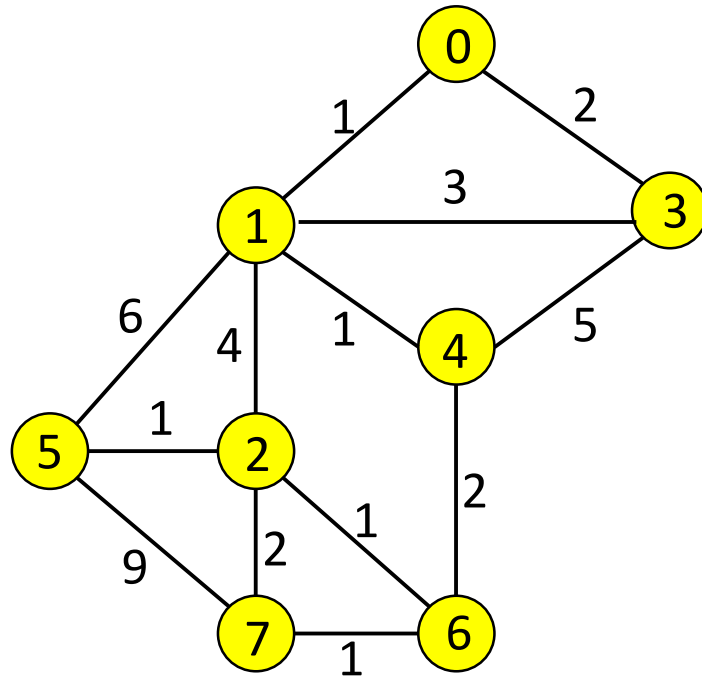


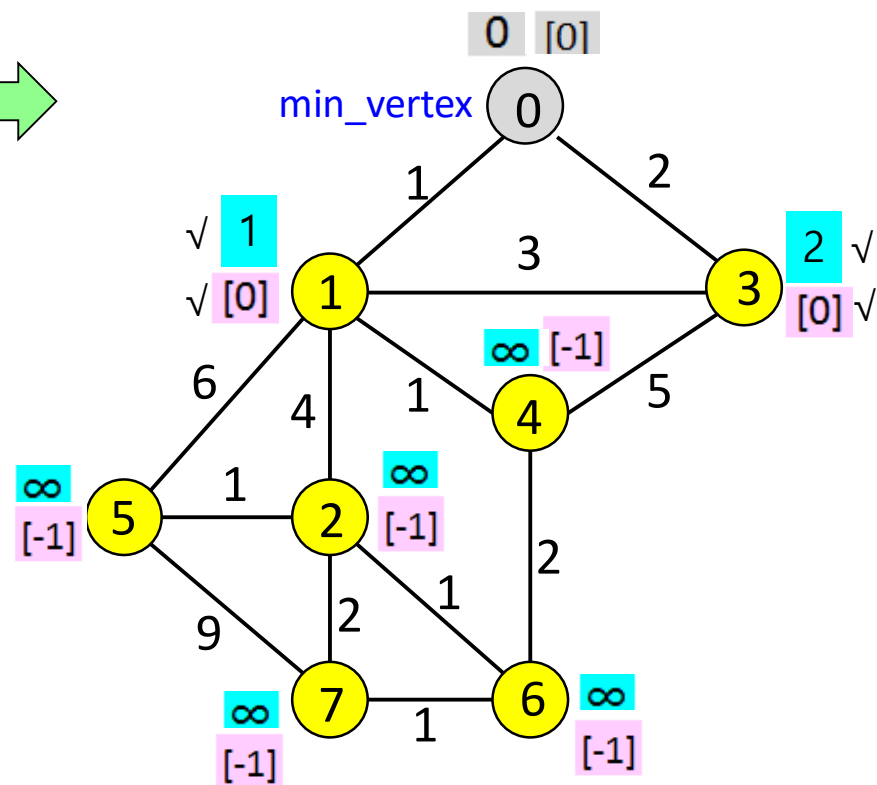
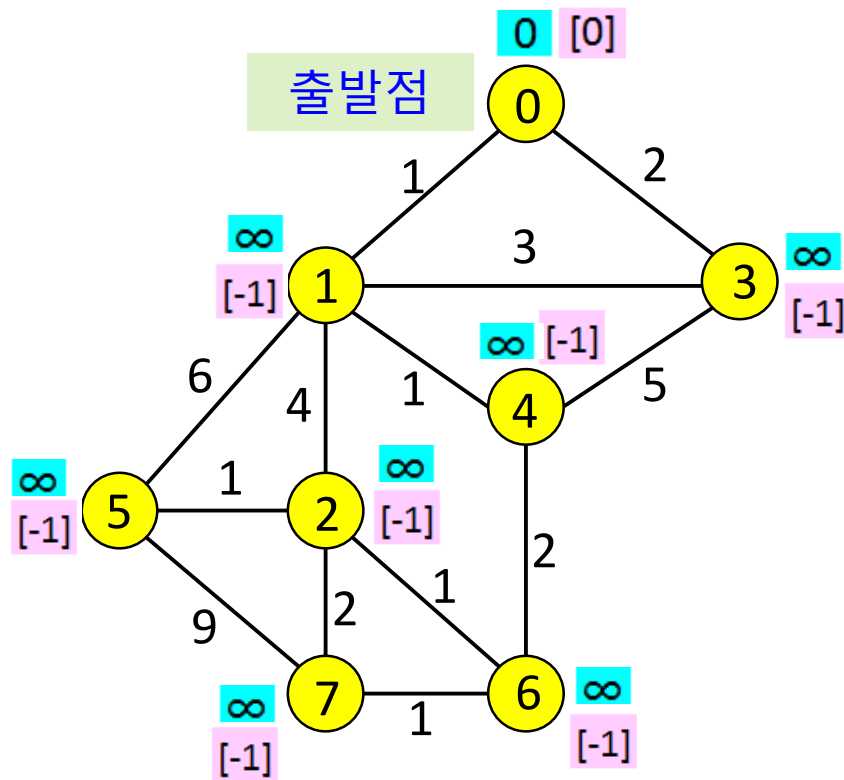
## [핵심 아이디어]

그리디하게 정점을 선택하여 방문하고, 선택한 정점의 방문 안된 인접한 정점들에 대한 간선 완화를 수행한다.

한번 방문된 정점의 D원소 값은 변하지 않는다.

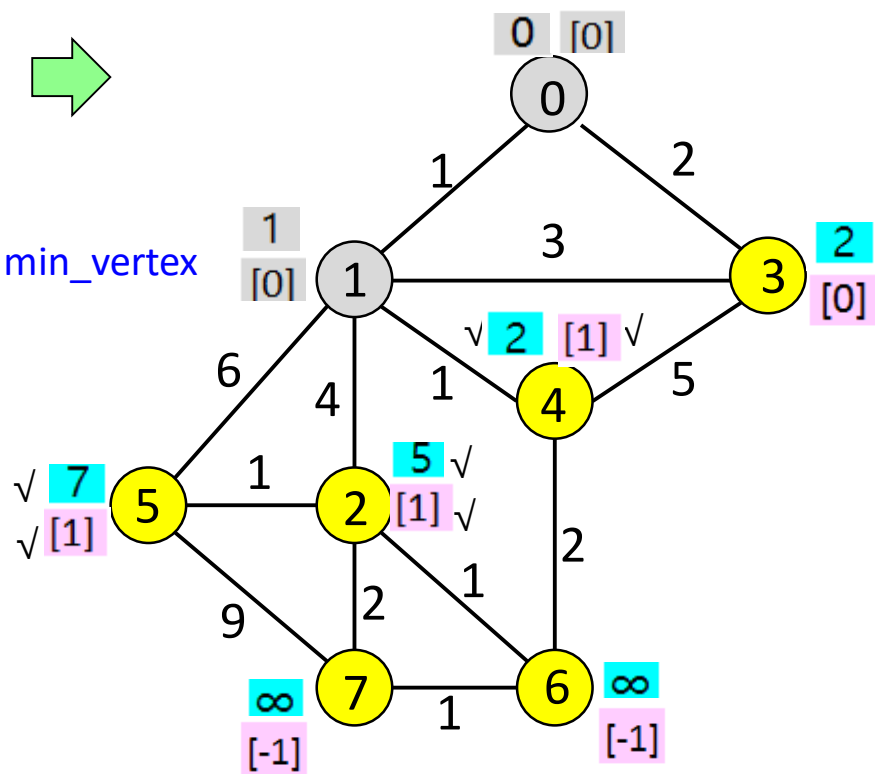
[예제]



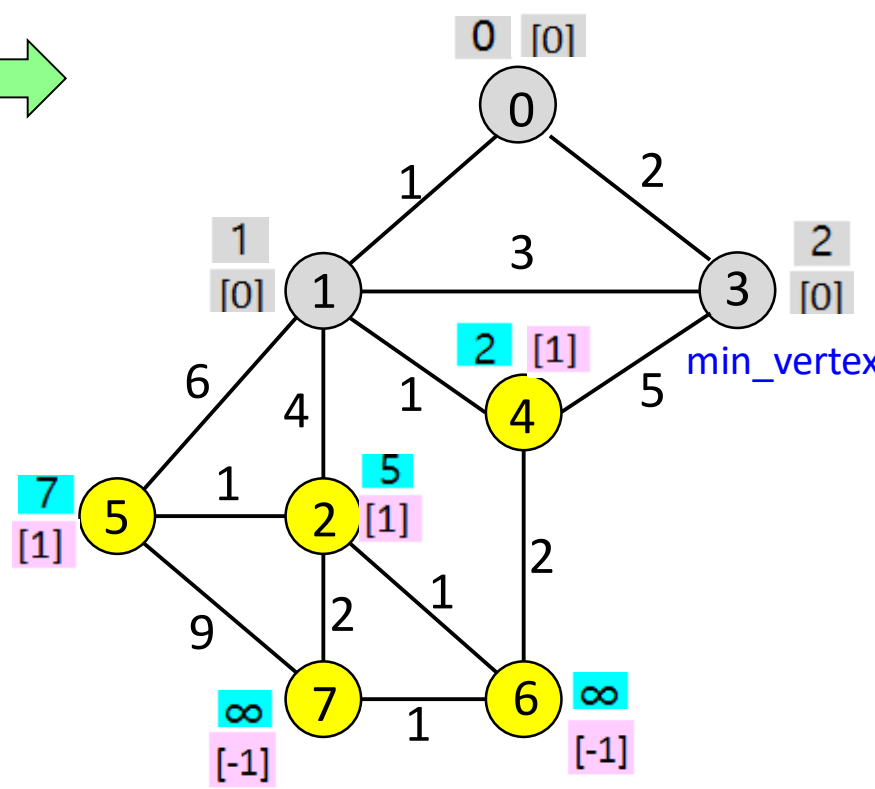


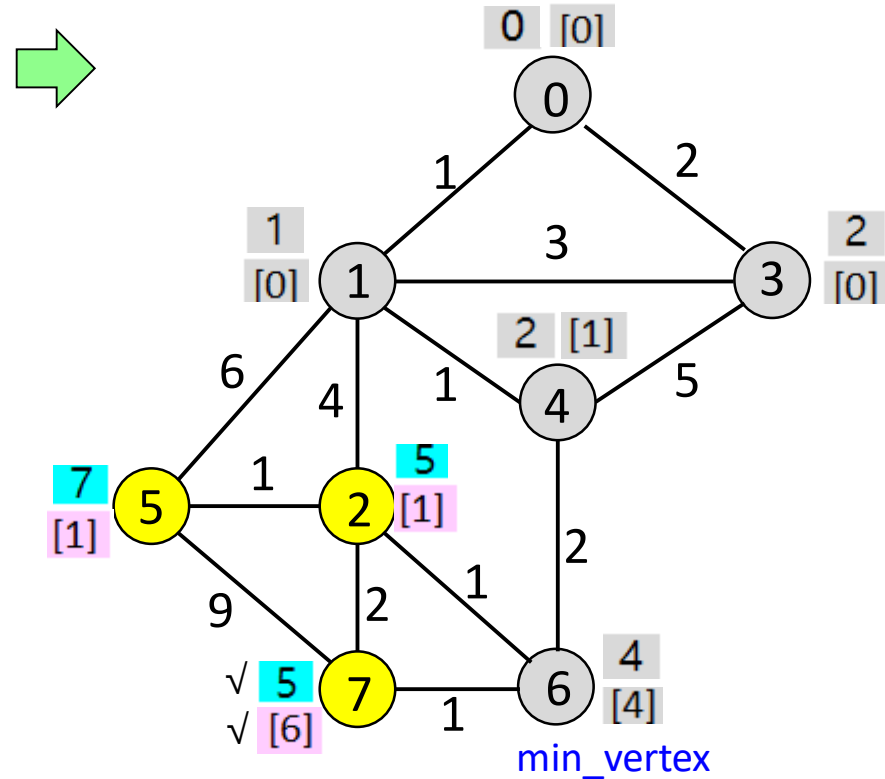
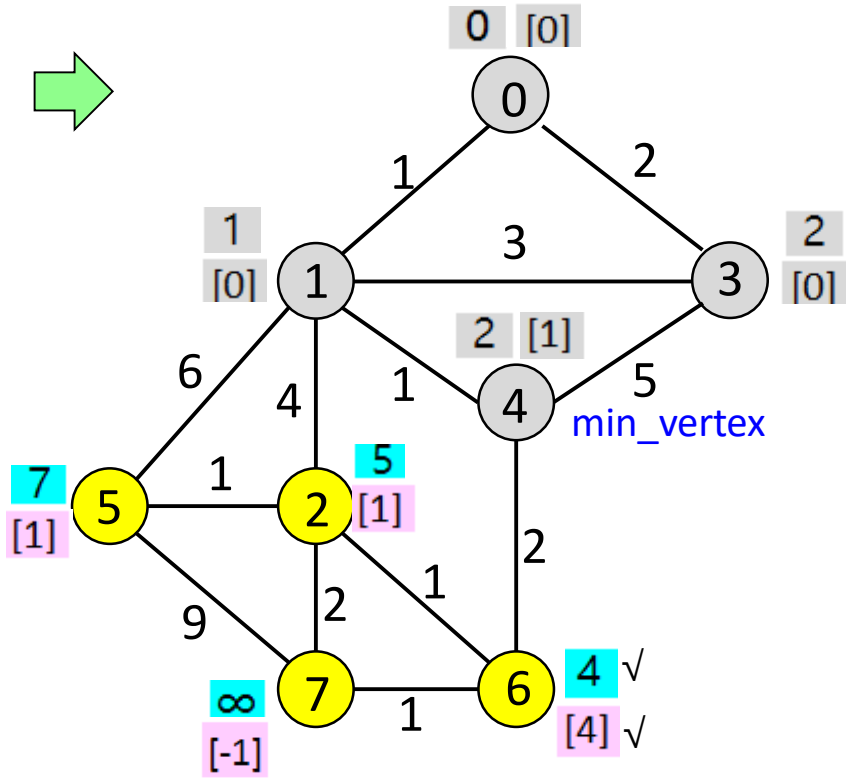


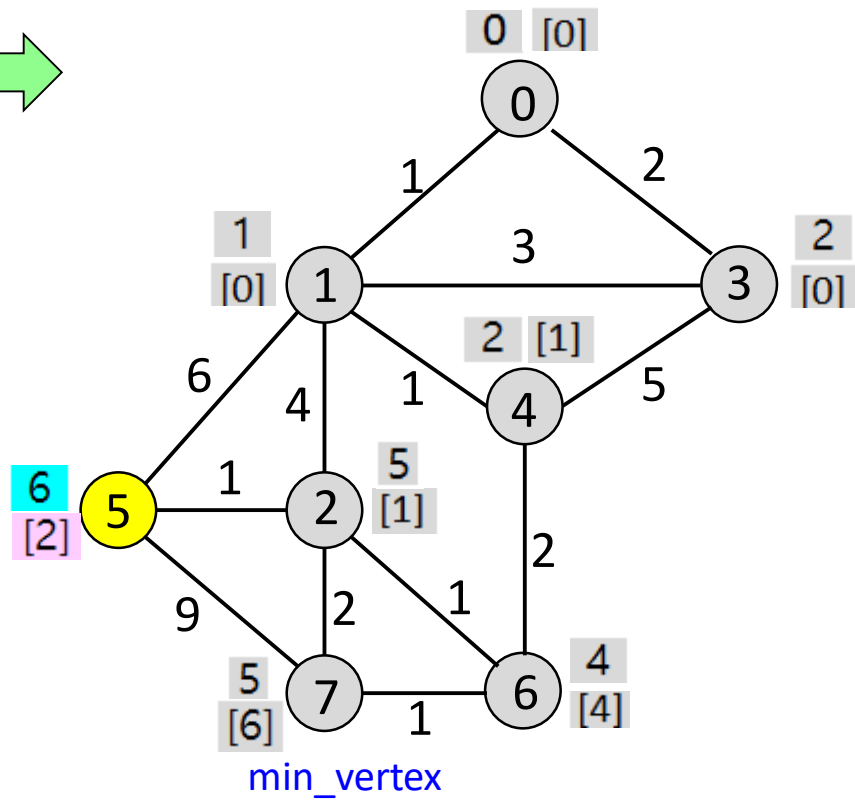
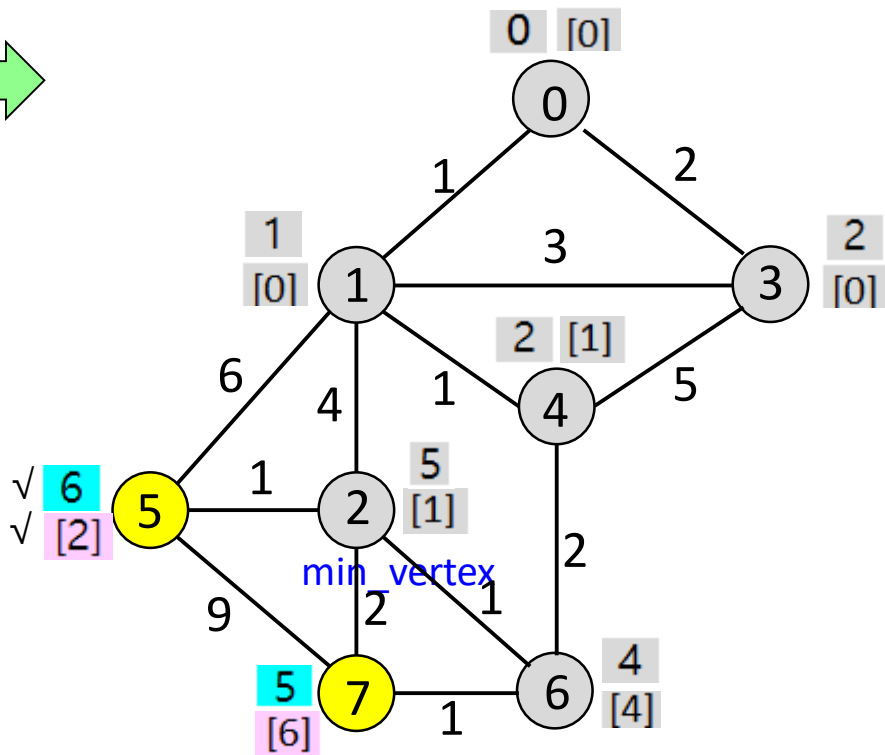
min\_vertex

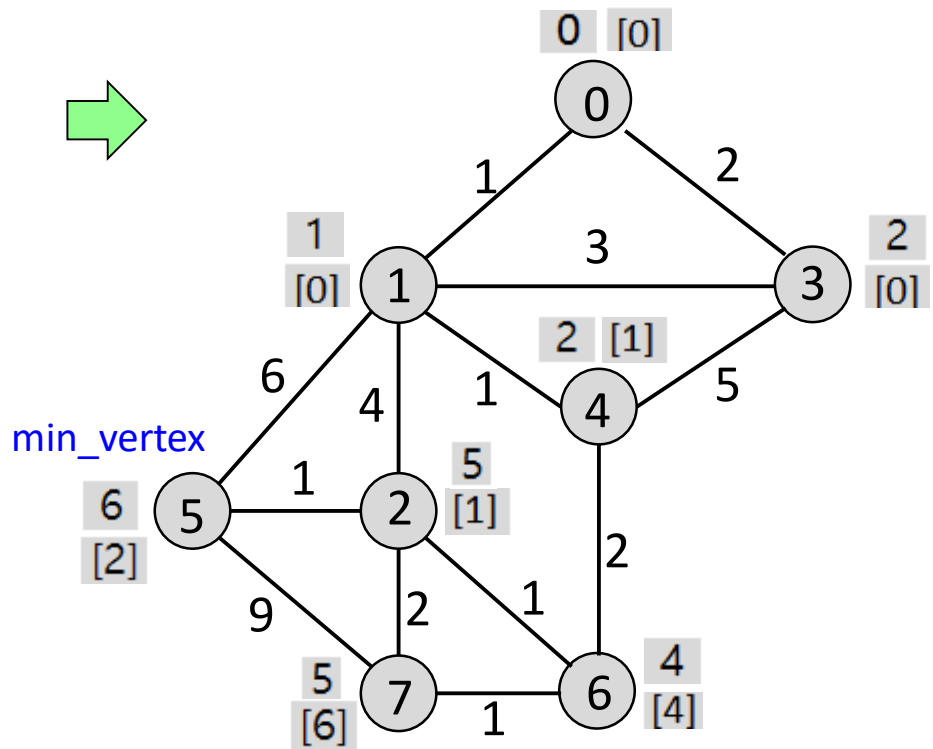


min\_vertex









정점 0으로부터의 최단 거리

$$[0, 1] = 1$$

$$[0, 2] = 5$$

$$[0, 3] = 2$$

$$[0, 4] = 2$$

$$[0, 5] = 6$$

$$[0, 6] = 4$$

$$[0, 7] = 5$$

정점 0으로부터의 최단 경로

$$1 < -0$$

$$2 < -1 < -0$$

$$3 < -0$$

$$4 < -1 < -0$$

$$5 < -2 < -1 < -0$$

$$6 < -4 < -1 < -0$$

$$7 < -6 < -4 < -1 < -0$$

```

01 import sys
02 N = 8
03 s = 0
04 g = [None] * N
05 g[0] = [(1, 1), (3, 2)]
06 g[1] = [(0, 1), (2, 4), (3, 3), (4, 1), (5, 6)]
07 g[2] = [(1, 4), (5, 1), (6, 1), (7, 2)]
08 g[3] = [(0, 2), (1, 3), (4, 5)]
09 g[4] = [(1, 1), (3, 5), (6, 2)]
10 g[5] = [(1, 6), (2, 1), (7, 9)]
11 g[6] = [(2, 1), (4, 2), (7, 1)]
12 g[7] = [(2, 2), (5, 9), (6, 1)]
13
14 visited = [False] * N
15 D = [sys.maxsize] * N
16 D[s] = 0
17 previous = [None] * N
18 previous[s] = s
19

```

sys.maxsize(최댓값) 사용 위해

입력 그래프의  
인접리스트

각 원소를 최댓값으로

초기화

최단경로 추출을 위해



```
20 for k in range(N):
21     m = -1
22     min_value = sys.maxsize
23     for j in range(N):
24         if not visited[j] and D[j] < min_value:
25             min_value = D[j]
26             m = j
27     visited[m] = True
28     for v, wt in list(g[m]):
29         if not visited[v]:
30             if D[m]+wt < D[v]:
31                 D[v] = D[m] + wt
32                 previous[v] = m
33
```

`m = min_vertex`

방문 안된 정점들의 D  
원소들 중에서 최솟값을  
가진 정점 m 찾기

정점 m에 인접한 v와 (m, v)의 가중치 wt에 대해

D[v] 갱신: 간선완화

D[v]가 정점 m 때문에  
갱신되었음을 기록

```


34 print('정점 ', s, '(으)로부터 최단거리:')
35 for i in range(N):
36     if D[i] == sys.maxsize:
37         print(s, '와(과) ', i, ' 사이에 경로 없음.')
38     else:
39         print('[%d, %d]' % (s, i), '=', D[i])
40
41 print('\n정점 ', s, '(으)로부터의 최단 경로')
42 for i in range(N):
43     back = i
44     print(back, end='')
45     while back != s:
46         print(' <-', previous[back], end='')
47         back = previous[back]
48     print()

```

[프로그램 8-7] dijkstra.py

- Line 20의 for-루프는 N개의 정점을 방문한 후 종료
- Line 21~26에서는 아직 방문 안된 정점들 중에서 출발점에서 가장 가까운 정점  $m$  (즉,  $\text{min\_vertex}$ )을 찾고
- Line 31에서는 정점  $m$ 에 인접하면서 방문 안된 정점에 대해 간선완화를 수행한다.
- 특히 line 32에서는  $D$ 의 원소가 갱신될 때 정점  $m$ 을  $\text{previous}[w]$ 에 저장해둔 뒤, 나중에 최단경로 추출에 사용

## 프로그램 수행 결과

Console  PyUnit

<terminated> dijkstra.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\

정점 0 (으)로부터 최단거리:

[0, 0] = 0

[0, 1] = 1

[0, 2] = 5

[0, 3] = 2

[0, 4] = 2

[0, 5] = 6

[0, 6] = 4

[0, 7] = 5

정점 0 (으)로부터의 최단 경로

0

1 <- 0

2 <- 1 <- 0

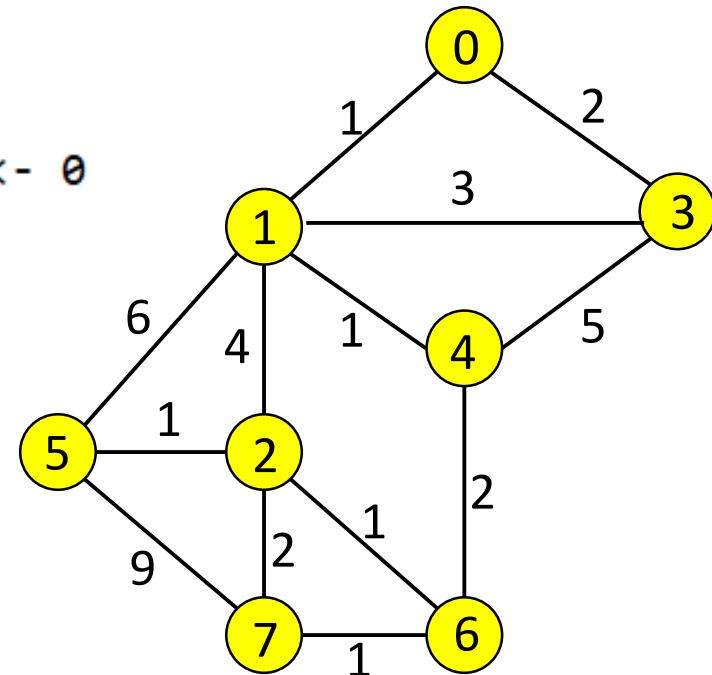
3 <- 0

4 <- 1 <- 0

5 <- 2 <- 1 <- 0

6 <- 4 <- 1 <- 0

7 <- 6 <- 4 <- 1 <- 0



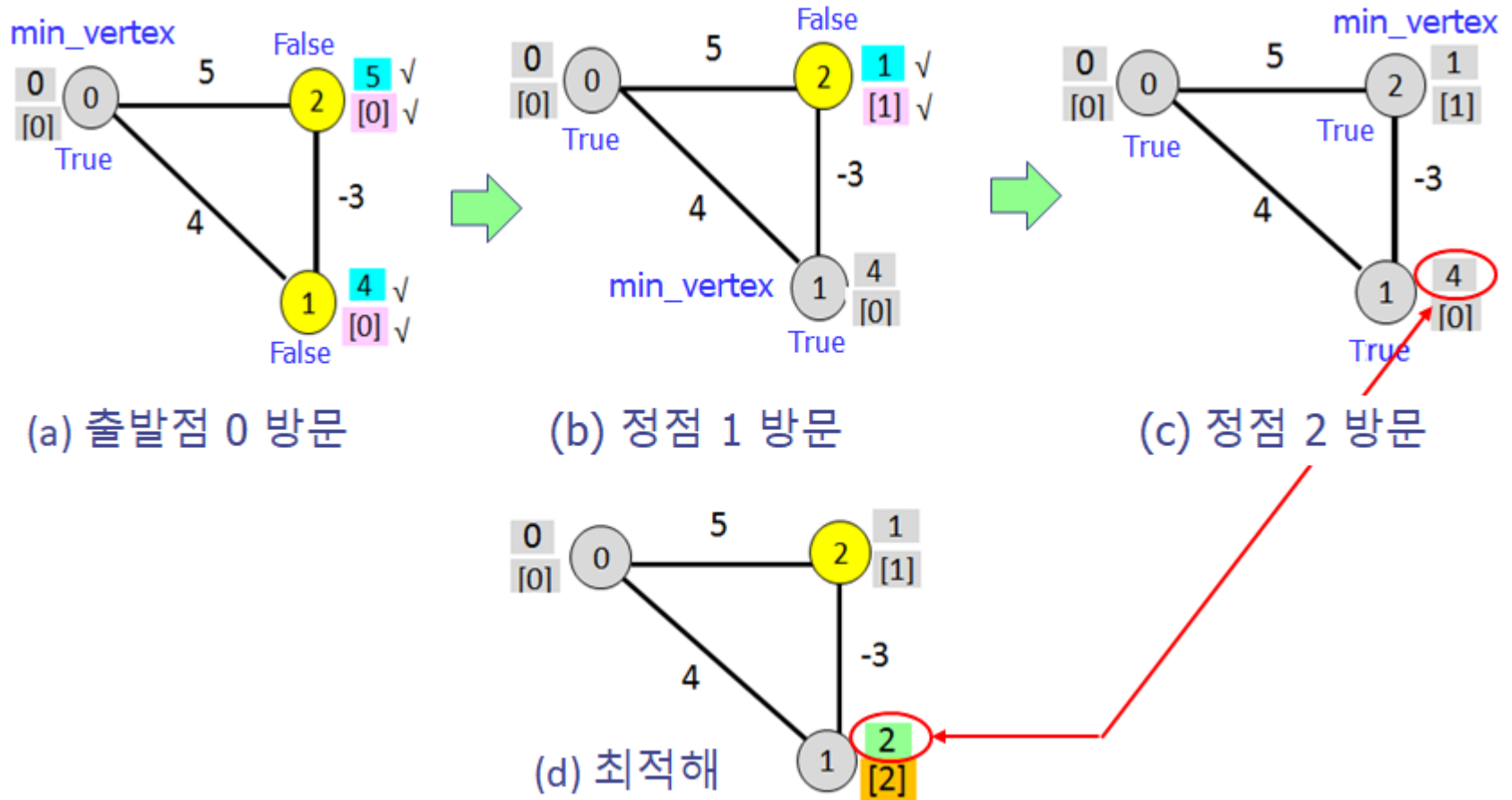
# 수행 시간(1)

- Dijkstra 알고리즘은 N번의 반복을 거쳐 min\_vertex를 찾고 min\_vertex에 인접하면서 방문되지 않은 정점들에 대한 간선완화를 시도
- 이후 D에서 min\_vertex를 탐색하는데  $O(N)$  시간이 소요되고, min\_vertex에 인접한 정점들을 검사하여 D의 원소들을 갱신하므로 추가로  $O(N)$  시간이 소요
- 따라서 총 수행 시간은  $N \times (O(N) + O(N)) = O(N^2)$

## 수행 시간(2)

- Dijkstra 최단경로 알고리즘은 Prim MST 알고리즘과 전체적으로 동일하므로 수행시간도 동일하다.
- 따라서 이진힙과 피보나치힙을 사용하는 경우의 수행시간도 각각 동일한 수행 시간을 갖는다.

- Dijkstra 알고리즘은 입력그래프에 음수 가중치가 있으면 최단 경로 찾기에 실패하는 경우가 발생
- Dijkstra 알고리즘이 최적해를 찾지 못하는 반례



- (a) 출발점이 방문되어  $visited[0] = true$
- 이후  $D[1] = 4$ ,  $previous[1] = 0$  그리고  $D[2] = 5$ ,  $previous[1] = 0$ 으로 각각 갱신
- (b)  $D[1]$ 이 최솟값이므로 정점 1이 방문되고,  $D[2] = 1$ ,  $previous[1] = 1$ 로 갱신
- (c) 마지막으로 방문 안된 정점 2가 방문되고 알고리즘 종료
- 그러나 (d)를 보면 출발점 0에서 정점 1까지 최단 경로는 **[0-2-1]**이고, 거리는 2
- [이러한 문제점이 발생한 이유] Dijkstra 알고리즘이  $D$ 의 원소 값의 증가 순으로  $min\_vertex$ 를 선택하고, 한번 방문된 정점의  $D$  원소를 다시 갱신하지 않기 때문



## 8.5.2 Floyd-Warshall 알고리즘

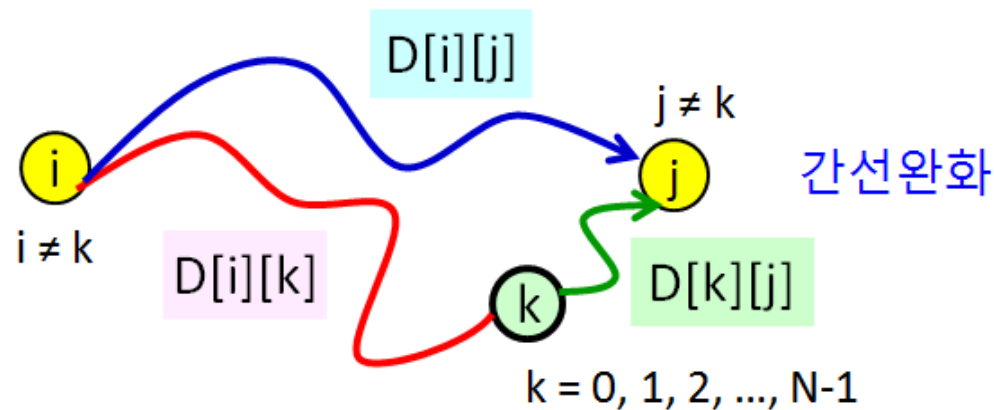
- Floyd-Warshall 알고리즘은 모든 정점 쌍 사이의 최단 경로 계산
- 모든 쌍 최단경로(All Pairs Shortest Paths) 알고리즘으로도 불린다.
- 지도에서 도시간 거리를 계산한 표를 볼 수 있는데, Floyd-Warshall 알고리즘을 사용하면 얻을 수 있음

|    | 서울<br>Seoul | 인천<br>Incheon | 수원<br>Suwon | 대전<br>Daejeon | 전주<br>Jeonju | 광주<br>Gwangju | 대구<br>Daegu | 울산<br>Ulsan | 부산<br>Busan |
|----|-------------|---------------|-------------|---------------|--------------|---------------|-------------|-------------|-------------|
| 서울 |             | 40            | 40          | 155           | 230          | 320           | 300         | 410         | 430         |
| 인천 |             |               | 55          | 175           | 250          | 350           | 320         | 450         | 450         |
| 수원 |             |               |             | 130           | 190          | 300           | 270         | 355         | 390         |
| 대전 |             |               |             |               | 95           | 185           | 150         | 260         | 280         |
| 전주 |             |               |             |               |              | 105           | 220         | 330         | 320         |
| 광주 |             |               |             |               |              |               | 220         | 330         | 270         |
| 대구 |             |               |             |               |              |               |             | 110         | 135         |
| 울산 |             |               |             |               |              |               |             |             | 50          |
| 부산 |             |               |             |               |              |               |             |             |             |

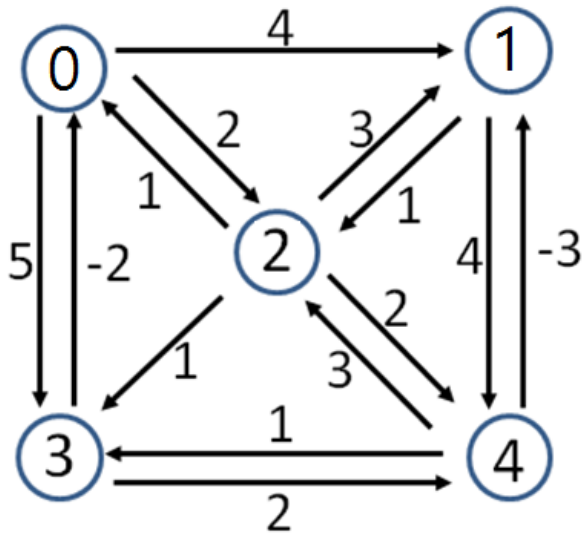
- 모든 쌍 최단 경로 찾기는 출발점을 0에서 N-1까지 바꿔가며 Dijkstra 알고리즘을 각각 수행하는 것으로 모든 쌍에 대한 최단 경로를 찾을 수 있음
- 이때 인접 행렬을 사용하면 수행 시간은  $O(N^3)$
- Floyd-Warshall 알고리즘의 수행 시간도  $O(N^3)$ 이지만 Floyd-Warshall 알고리즘은 Dijkstra 알고리즘에 비해 훨씬 알고리즘이 간단하고, 음수 가중치 그래프에서도 최단 경로를 찾을 수 있다는 장점을 가짐

[핵심 아이디어] 입력그래프의 정점들에  $0, 1, 2, \dots, N-1$ 로 ID를 부여하고, 정점 ID를 증가시키며 간선 완화를 수행

정점 0을 경유하는 경로에 존재하는 정점들에 대해 간선 완화를 수행하고, 갱신된 결과를 바탕으로 정점 1을 경유하는 경로에 존재하는 정점들에 대해 간선 완화를 수행, ..., 정점  $N-1$ 을 경유하는 경로에 존재하는 정점들에 대해 간선 완화 수행

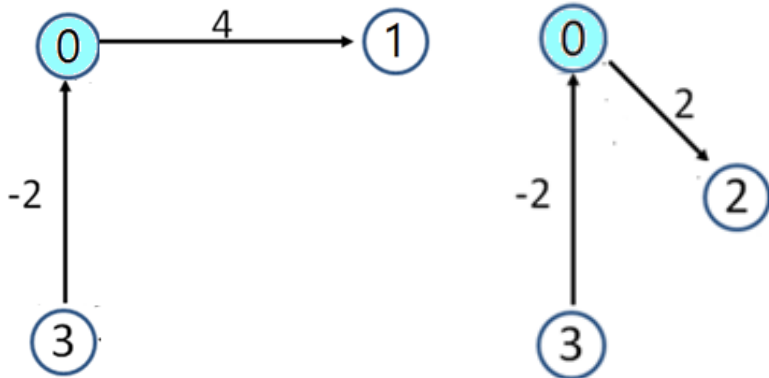


[예제]

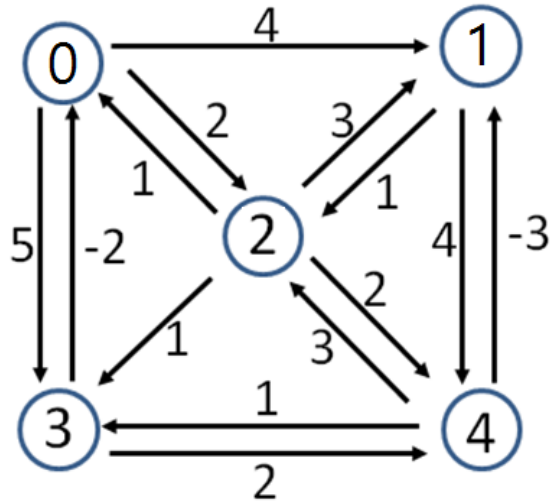


| D | 0        | 1        | 2        | 3        | 4        |
|---|----------|----------|----------|----------|----------|
| 0 | 0        | 4        | 2        | 5        | $\infty$ |
| 1 | $\infty$ | 0        | 1        | $\infty$ | 4        |
| 2 | 1        | 3        | 0        | 1        | 2        |
| 3 | -2       | $\infty$ | $\infty$ | 0        | 2        |
| 4 | $\infty$ | -3       | 3        | 1        | 0        |

k=0



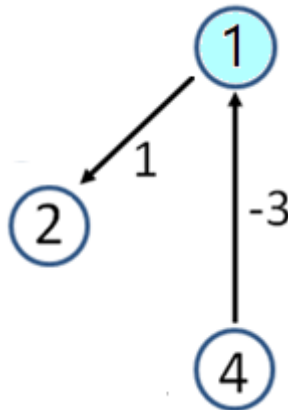
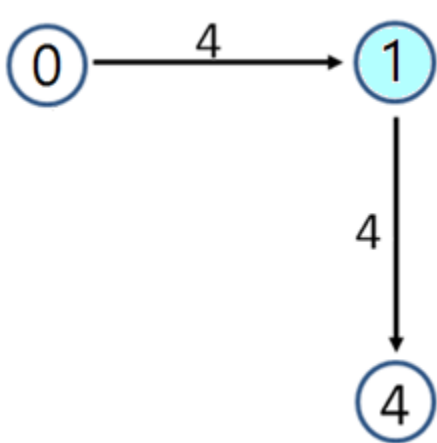
| D | 0        | 1  | 2 | 3        | 4        |
|---|----------|----|---|----------|----------|
| 0 | 0        | 4  | 2 | 5        | $\infty$ |
| 1 | $\infty$ | 0  | 1 | $\infty$ | 4        |
| 2 | 1        | 3  | 0 | 1        | 2        |
| 3 | -2       | 2  | 0 | 0        | 2        |
| 4 | $\infty$ | -3 | 3 | 1        | 0        |



•  $k=1$

–  $D[0,4]=8$ , due to  $0 \rightarrow 1 \rightarrow 4$

–  $D[4,2]=-2$ , due to  $4 \rightarrow 1 \rightarrow 2$



| D | 0        | 1  | 2  | 3        | 4 |
|---|----------|----|----|----------|---|
| 0 | 0        | 4  | 2  | 5        | 8 |
| 1 | $\infty$ | 0  | 1  | $\infty$ | 4 |
| 2 | 1        | 3  | 0  | 1        | 2 |
| 3 | -2       | 2  | 0  | 0        | 2 |
| 4 | $\infty$ | -3 | -2 | 1        | 0 |

- $k=2$

| D | 0         | 1  | 2  | 3         | 4        |
|---|-----------|----|----|-----------|----------|
| 0 | 0         | 4  | 2  | <b>3</b>  | <b>4</b> |
| 1 | <b>2</b>  | 0  | 1  | <b>2</b>  | <b>3</b> |
| 2 | 1         | 3  | 0  | 1         | 2        |
| 3 | -2        | 2  | 0  | 0         | 2        |
| 4 | <b>-1</b> | -3 | -2 | <b>-1</b> | 0        |

- $k=3$

| D | 0         | 1  | 2  | 3  | 4 |
|---|-----------|----|----|----|---|
| 0 | 0         | 4  | 2  | 3  | 4 |
| 1 | <b>0</b>  | 0  | 1  | 2  | 3 |
| 2 | <b>-1</b> | 3  | 0  | 1  | 2 |
| 3 | -2        | 2  | 0  | 0  | 2 |
| 4 | <b>-3</b> | -3 | -2 | -1 | 0 |

- $k=4$

| D | 0  | 1         | 2  | 3  | 4 |
|---|----|-----------|----|----|---|
| 0 | 0  | <b>1</b>  | 2  | 3  | 4 |
| 1 | 0  | 0         | 1  | 2  | 3 |
| 2 | -1 | <b>-1</b> | 0  | 1  | 2 |
| 3 | -2 | <b>-1</b> | 0  | 0  | 2 |
| 4 | -3 | -3        | -2 | -1 | 0 |

## Floyd-Warshall 알고리즘

[1]  $D$  = 입력 그래프의 인접행렬

[2] **for**  $k$  **in**  $\text{range}(N)$ :

[3]     **for**  $i$  **in**  $\text{range}(N)$ :

[4]         **for**  $j$  **in**  $\text{range}(N)$ :

[5]             **if**  $D[i][j] > D[i][k] + D[k][j]$ :

[6]                  $D[i][j] = D[i][k] + D[k][j]$    # 간선완화

- 알고리즘의 step [1]에서 입력 그래프의 인접행렬을 모든 쌍 최단거리를 저장할 리스트 D에 복사한다.
- Step [2]의 for-루프는 경유하는 정점 ID를 0부터 N-1까지 변화시킨다.
- 모든 쌍 i와 j에 대하여 step [3]~[4]의 이중 for-루프가 i와 j를 각각 0부터 N-1까지 증가시키며, step [5]~[6]에서 간선완화를 수행한다.



```
01 import sys
```

sys.maxsize(최댓값) 사용 위해

```
02 N = 5
```

```
03 INF = sys.maxsize
```

```
04 D = [[0, 4, 2, 5, INF], [INF, 0, 1, INF, 4],  
05      [1, 3, 0, 1, 2], [-2, INF, INF, 0, 2],  
06      [INF, -3, 3, 1, 0]]
```

입력 그래프의  
인접행렬

```
07
```

```
08 for k in range(N):
```

```
09     for i in range(N):
```

```
10         for j in range(N):
```

```
11             D[i][j] = min(D[i][j], D[i][k]+D[k][j])
```

간선완화

```
12
```

```
13 for i in range(N):
```

최단거리 출력

```
14     for j in range(N):
```

```
15         print('%3d' % D[i][j], end='')
```

```
16     print()
```

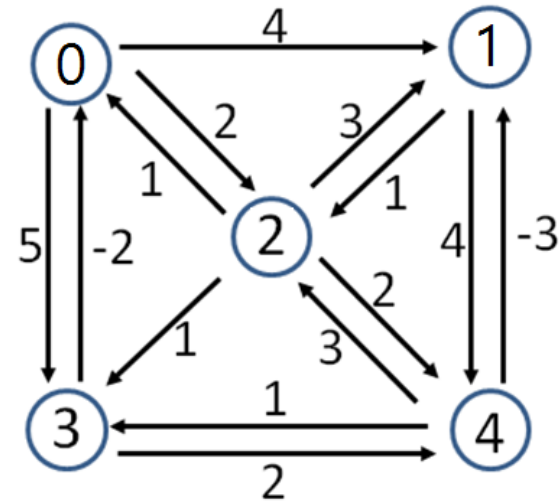
[프로그램 8-8] floyd\_warshall.py

## 프로그램 수행 결과

Console  PyUnit

<terminated> floyd\_warshall.py

|    |    |    |    |   |
|----|----|----|----|---|
| 0  | 1  | 2  | 3  | 4 |
| 0  | 0  | 1  | 2  | 3 |
| -1 | -1 | 0  | 1  | 2 |
| -2 | -1 | 0  | 0  | 2 |
| -3 | -3 | -2 | -1 | 0 |



| D | 0  | 1  | 2  | 3  | 4 |
|---|----|----|----|----|---|
| 0 | 0  | 1  | 2  | 3  | 4 |
| 1 | 0  | 0  | 1  | 2  | 3 |
| 2 | -1 | -1 | 0  | 1  | 2 |
| 3 | -2 | -1 | 0  | 0  | 2 |
| 4 | -3 | -3 | -2 | -1 | 0 |

# 수행 시간

- Floyd-Warshall 알고리즘의 수행 시간은 [1]의 리스트 복사에는  $O(N^2)$ 이 소요되고, 이후 for-루프가 3개가 중첩되므로  $O(N^3)$ 이 소요
- 따라서  $O(N^2) + O(N^3) = O(N^3)$



## 요약

- 그래프를 자료구조로서 저장하기 위해 **인접 행렬**과 **인접 리스트**가 주로 사용된다.
- 그래프는 **깊이 우선 탐색** (DFS)과 **너비 우선 탐색** (BFS)으로 그래프의 모든 정점들을 방문하며, DFS는 스택을 사용하고, BFS는 큐 자료구조를 사용한다.
- 무방향 그래프에서 **연결성분 찾기**는 DFS 또는 BFS를 기반하여 수행된다.
- **위상 정렬** 알고리즘은 DFS를 수행하며 각 정점  $v$ 의 인접한 모든 정점들의 방문이 끝나자마자  $v$ 를 리스트에 추가한다. 리스트의 역순이 위상정렬이다.

- **Kruskal 알고리즘**은 간선들을 가중치로 정렬한 후에, 가장 가중치가 작은 간선이 트리에 사이클을 만들지 않으면 트리 간선으로 선택하고, 만들면 버리는 일을 반복하여  $N-1$ 개의 간선을 선택
- **Prim 알고리즘**은 트리에 인접한 가장 가까운 정점을 하나씩 추가하여 최소신장트리를 만든다.
- **Sollin 알고리즘**은 각 트리에서 트리에 연결된 간선들 중에서 가장 작은 가중치를 가진 간선을 선택한다. 이때 선택된 간선은 두 개의 트리를 하나의 트리로 합친다. 이와 같은 방식으로 하나의 트리가 남을 때까지 각 트리에서 최소 가중치 간선을 선택하여 연결

- Dijkstra 알고리즘은 출발점으로부터 방문 안된 정점들 중에서 가장 가까운 거리의 정점을 방문하고 방문한 정점을 기준으로 간선 완화를 수행하여 최단 경로를 계산
- Floyd-Warshall 알고리즘은 모든 정점 쌍 사이의 최단 경로를 찾는 알고리즘이다. 입력 그래프의 정점들을 0, 1, 2, ..., N-1로 ID를 부여하고, 정점 ID를 증가시키며 간선 완화를 수행