# 소프트웨어와 문제해결

Dr. Young-Woo Kwon

KNU

# 수업 개요

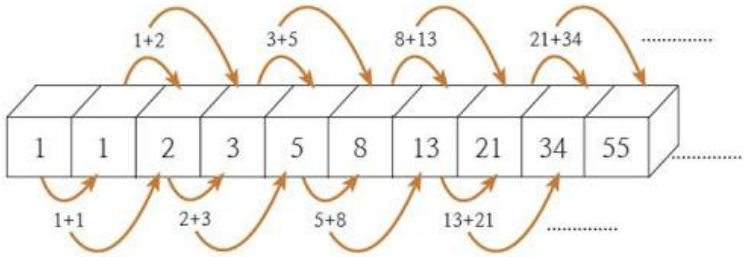
- 재귀호출
- 그래프
  - Directed and Undirected 그래프 표현법
  - Kruskal 알고리즘
  - 최소신장트리(MST - Minimum Spanning Tree)
  - 클러스터링 (Clustering)
- 재귀호출 - 연습
- 실습

# 순환 (재귀), Recursion

- 순환, 재귀, Recursive 함수
  - 함수가 자기 자신을 호출하는 함수
    - 반드시 종료 조건이 필요함
- 재귀 함수의 예
  - F(n) = F(n-1) + F(n-2)
- 재귀 함수를 사용하는 이유
  - 복잡한 알고리즘의 간단 명료한 구현 가능
- 단점 및 주의사항
  - 함수 호출이 중첩됨
  - 함수 호출의 오버헤드(overhead)가 큼
  - 재귀의 정도가 너무 깊으면 프로그램의 실행이 실패할 수도 있음

# 피보나치 수열

- 피보나치 수열이란 제0항을 0, 제1항을 1로 두고, 둘째 번 항부터는 바로 앞의 두 수를 더한 수로 놓는다. 1번째 수를 1로, 2번째 수도 1로 놓고, 3번째 수부터는 바로 앞의 두 수를 더한 수로 정의하는 게 좀 더 흔하게 알려져 있는 피보나치 수열이다.
- 이 수열의 항을 피보나치 수(Fibonacci Number)라 부른다.




- 구하고자 하는 피보나치 수의 개수를 입력 받아 개수만큼의 피보나치 수열을 출력하는 코드를 작성 하시오.

# 피보나치 수열

- 피보나치 수열을 구하는 알고리즘을 1) 재귀함수를 사용하여 작성하고 2) 재귀함수를 사용하지 않고 작성하시오

$$F_n := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

# 피보나치 수열

- 재귀 사용

```
k = int(input("구하고자 하는 피보나치 수의 갯수: "))

def fibo(n):
    return fibo(n-1) + fibo(n-2) if n >= 2 else n

for n in range(1, k+1):
    print(n, fibo(n))
```

# 보나치 수열

- 재귀 없이

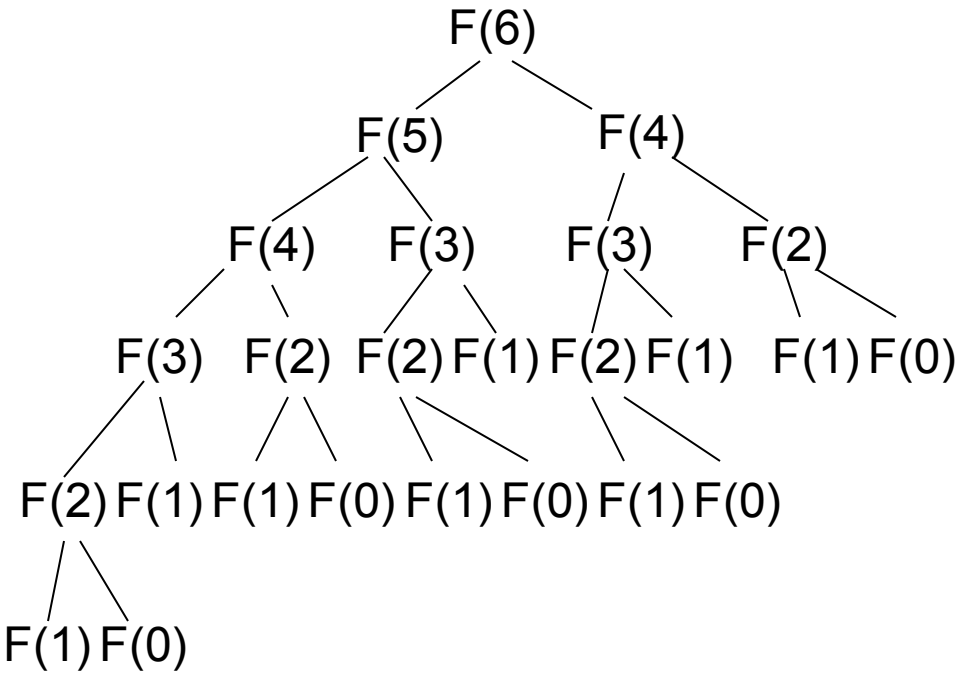
```
k = int(input("구하고자 하는 피보나치 수의 갯수: "))


def fibo(n):
    if n < 2:
        return n

    a, b = 0, 1
    for i in range(n - 1):
        a, b = b, a + b

    return b


for n in range(1, k + 1):
    print(n, fibo(n))
```

KNU

# 재귀함수의 문제점

- $F_n = F_{n-1} + F_{n-2}$; $F_0 = 0$; $F_1 = 1$

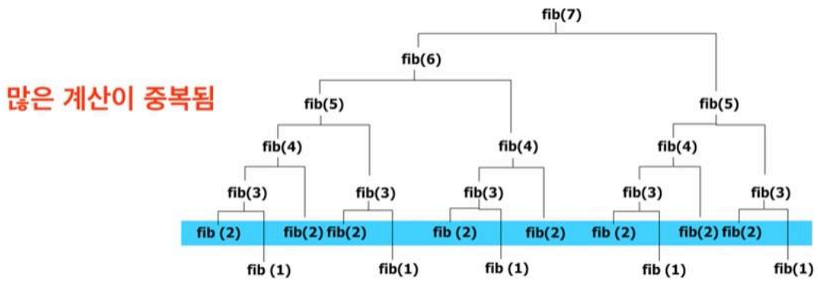
```
// 재귀(recursion)를 사용
// 비효율적인 알고리즘
int fib( unsigned int n ) {
    if (n == 0 || n == 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```



# 해결 방안

- 기억하기

```
def fibo(num):
    if num == 0:
        return 0
    elif num == 1:
        return 1

    return fibo(num - 1) + fibo(num - 2)
```



# 개선된 피보나치 수열 (기억하기 기법 사용)

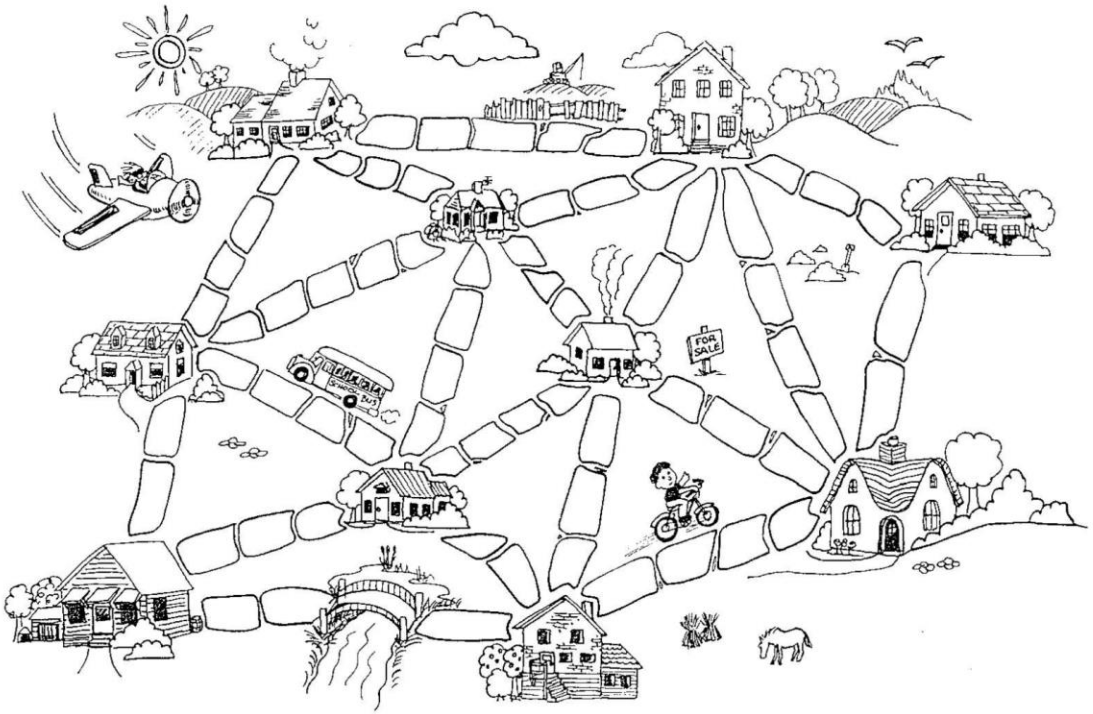
```
def fibo(n, memory = dict()):
    if n == 1 or n == 2:
        return 1

    if n in memory:
        return memory[n]

    memory[n] = fibo(n-1, memory) + fibo(n-2, memory)

    return memory[n]
```

메모리 제어법
→ dynamic programming

- 실행 시간 비교하기

# 자료구조: 그래프

# 진흙 도시




- 모든 집이 포장된 도로를 이용해서 서로 방문할 수 있도록 도로는 충분히 포장되어야 함
- 포장 비용: $1/block

# 진흙 도시

- 지도를 표현하는 방법?

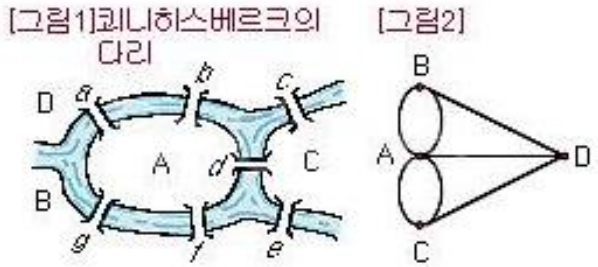
# 그래프

- 그래프: 여러개의 노드로 이루어진 네트워크
- 비용: 두 개의 노드간의 이동 비용
- 최소신장트리: 가장 저비용으로 그래프의 모든 노드를 연결하는 그래프

# 그래프(Graph)의 정의

- 1736년에 오일러(Euler)에 의해 처음 활용

- 그래프 G = ( V, E )
  - V : 정점(Vertices)
  - E : 에지(Edge)

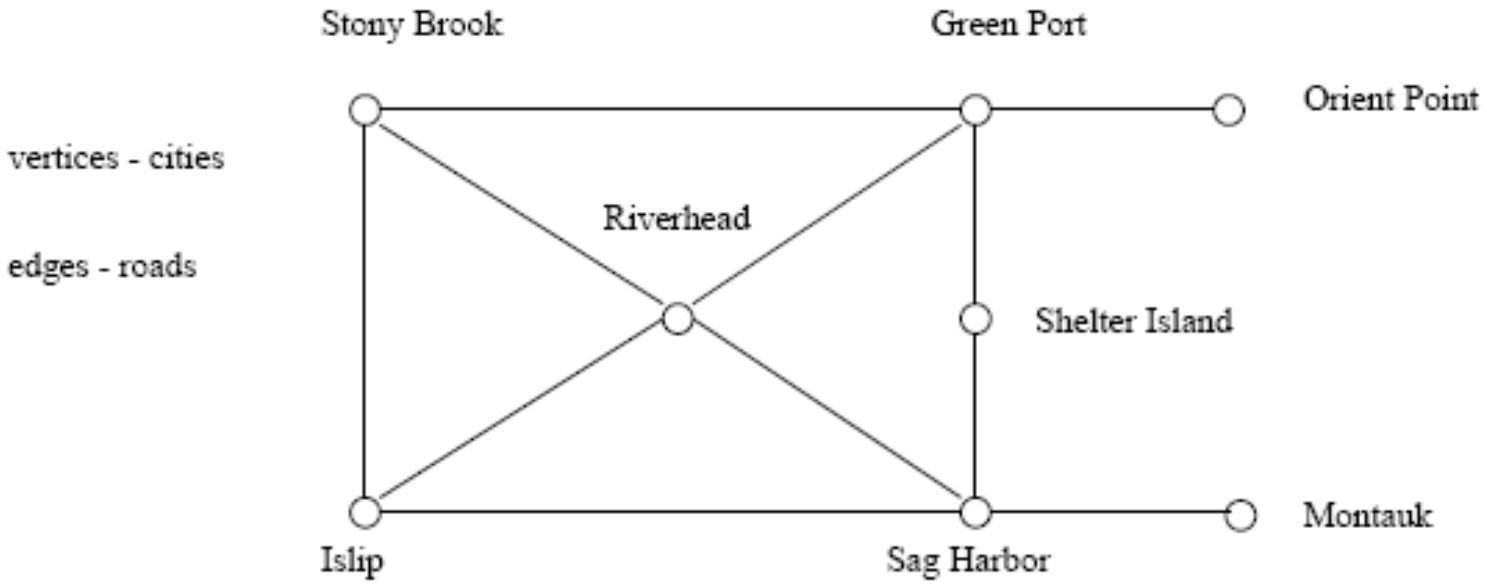



- 정의
  - 정점들의 집합 V, 에지들의 집합 E로 구성
  - 에지는 순서가 있거나 없는 정점들의 쌍으로 구성

그래프 G = (V, E)

# 그래프 응용의 예 : 도로망

- 도로망을 표현할 경우
  - 정점 : 도시 또는 교차로



# 그래프의 종류

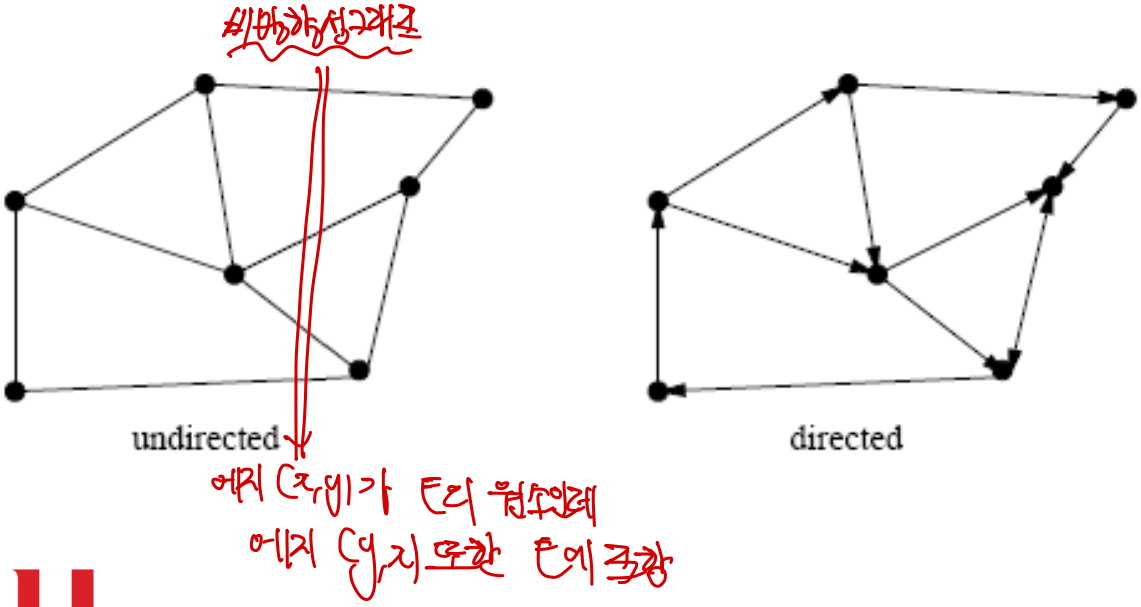
- 방향성(directed) vs. 비방향성(undirected) 그래프
- 가중(weighted) vs. 비가중(unweighted) 그래프
- 순환(cyclic) vs. 비순환(acyclic) 그래프

# 방향성 vs. 비방향성 그래프

(directed) (undirected)

- 그래프 G = (V, E) 에서 에지(x, y)가 E의 원소일 때, 에지(y, x) 또한 E에 포함되면 '**비방향성 그래프**'이다.

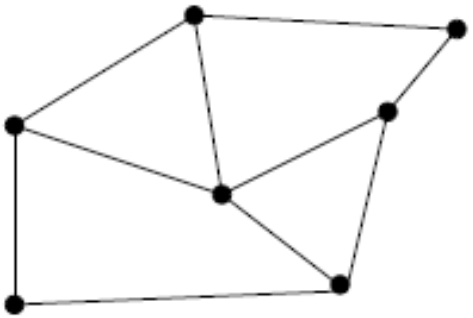
- 도로망에서의 경우
  - 비방향 : 도시와 도시 사이의 도로
  - 방향성 : 일방통행 or 시내도로 대부분의 경우



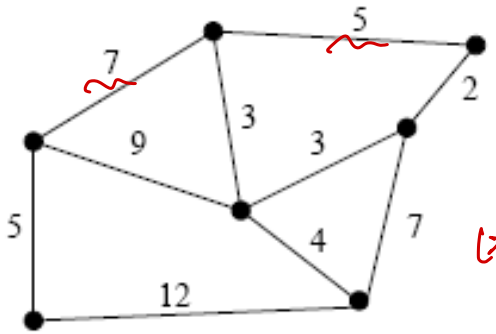
# 가중 vs. 비가중 그래프

- 각 에지나 정점이 어떤 값을 가지는 그래프를 '**가중 그래프**'라고 한다.
  - '비가중 그래프'의 경우 에지나 정점에 구별되는 값이 없다.

- 도로망의 경우
  - 거리, 주행시간, 제한속도 등의 값을 가질 수 있다.



unweighted




weighted

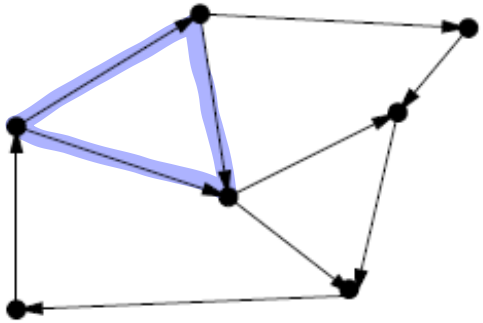
(가중 그래프)

↓

Kruskal 알고리즘으로 계산 가능.

# 순환 vs. 비순환 그래프

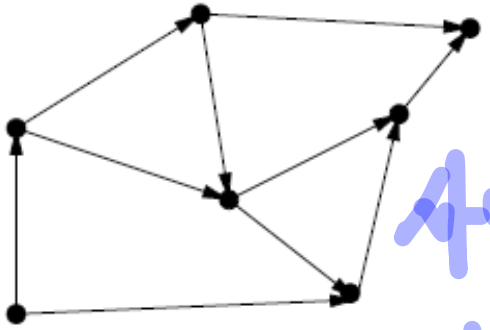
- 순환(cyclic) 그래프
  - 사이클이 존재하는 그래프
  - '비방향성'일 경우, 순환 그래프가 많음



cyclic

- 비 순환(acyclic) 그래프
  - 사이클을 포함하지 않는 그래프
  - '방향성 있는 비순환 그래프(DAGs, Directed Acyclic graph)
  - 스케줄링 문제에서 자주 발생

Directed Acyclic graph
= DAGs.



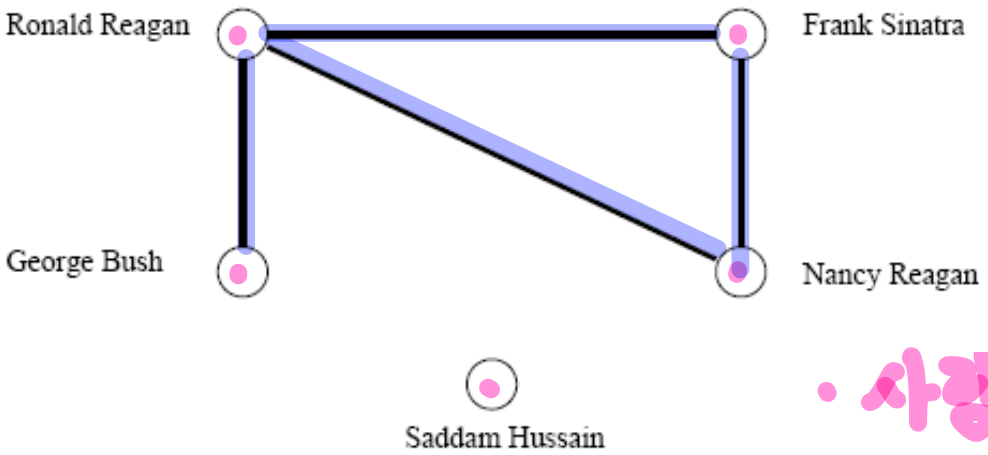
acyclic

사이클 X

# 그래프의 예 : 우정 그래프- The Friendship Graph

- 구성
  - 정점 : 사람
  - 에지 : 두 사람간의 관계(친분)

- 어떤 사람들의 집단이 있을 경우, 이 그래프를 통해 쉽게 정의할 수 있다.

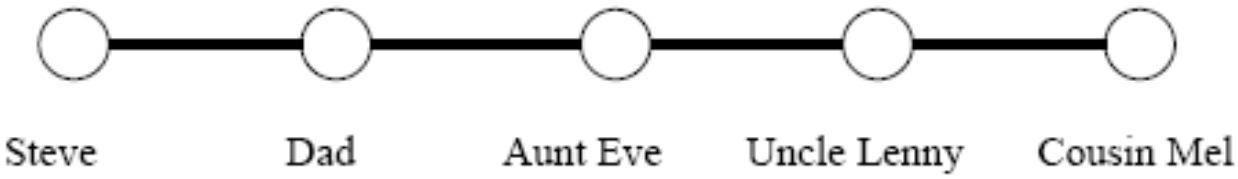


# 내가 당신을 안다면, 당신도 나를 알고 있는가?

- '비방향성 그래프' 로 표현된다면, 서로 알고 있다고 할 수 있다.
  - ex) 한 학교의 같은 반
- '방향성 그래프' 로 표현된다면, 알 수 없다.
  - ex) 유명한 사람일 경우, 연예인 등..

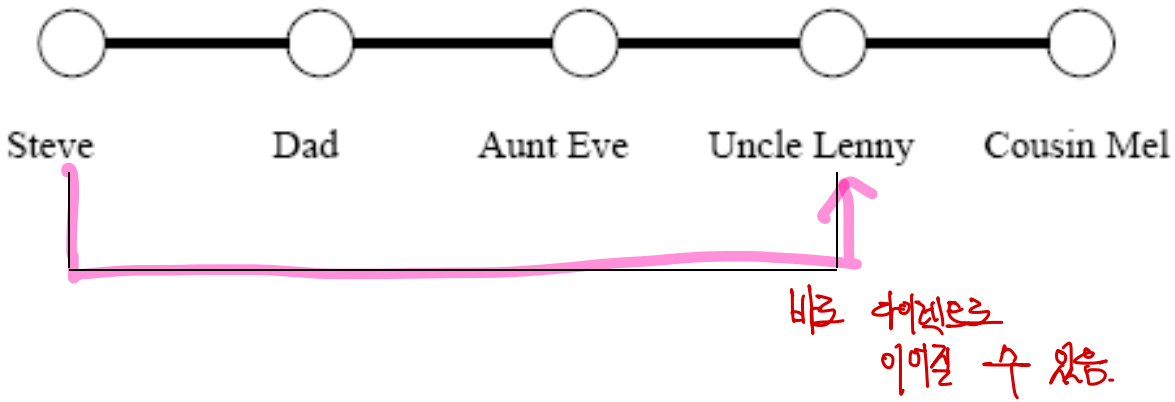
# 나는 대통령과 친분이 있는가?

- Mel Brooks라는 대통령이 있다.
- 그가 내 아버지의 여동생의 남편의 사촌일 경우



# 나는 대통령과 얼마나 가까운 사이인가?

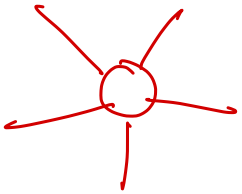
- Lenny 삼촌과 친분 관계 edge가 있다면, Mel Brooks와 나는 더욱 가까운 사이라고 말할 수 있다.




- 이와 같이 최단거리(shortest path)를 찾는 일은 중요한 문제로 인식된다.

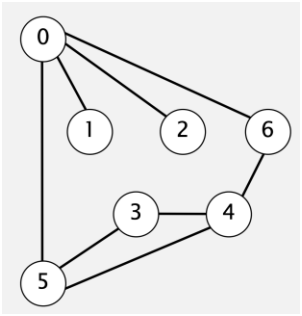
# 발이 가장 넓은 사람은?

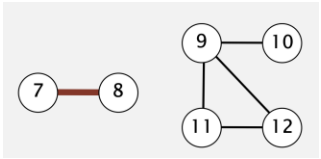
- 차수(Degree)가 가장 높은 정점이 발이 가장 넓은 사람이다.
  - 차수 : 정점에 연결된 에지의 수



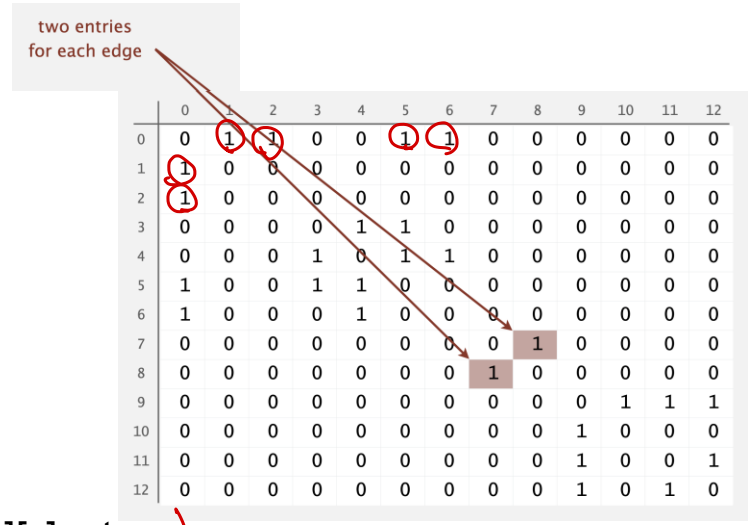
# 그래프 표현법

- 그래프 표현
  - Adjacency Matrix




인접 행렬




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

정방 행렬 (대칭)

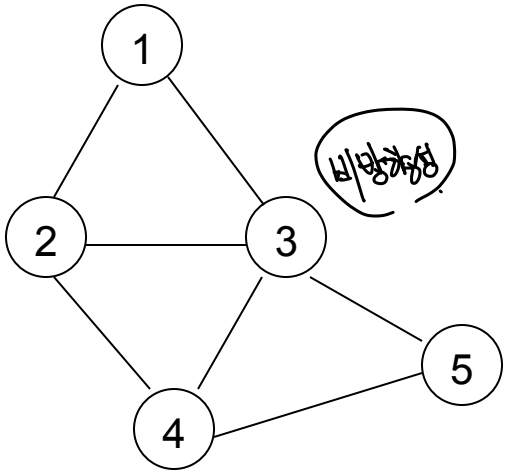
  - For each edge v-w: adj[v][w] = adj[w][v] = true

# 그래프 표현 : 인접 행렬 - Adjacency Matrix

- 그래프 G = (V, E)에서
  - 그래프 G는 n개의 정점과 m개의 에지를 포함
  - n x n 행렬 M을 이용하여 G를 표현
  - M[i, j]가 1이면, E(i, j)가 존재
  - 정점의 수는 많지만, 에지가 적다면 많은 공간이 낭비됨

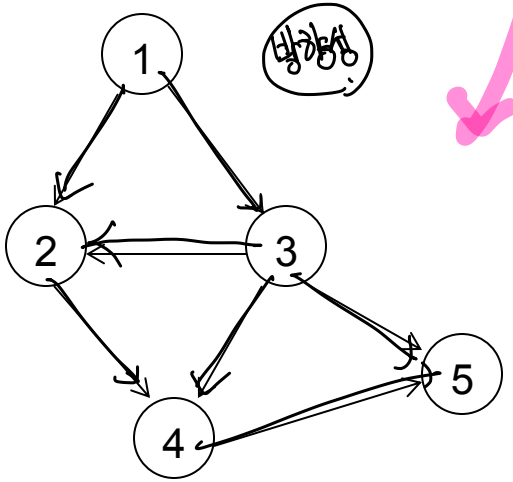
- 다음과 같은 경우 공간을 줄일 수 있는가?
  - 비방향성 그래프(undirected graph)
  - 희소 그래프(sparse graph)

# 인접행렬 예




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 |
| 3 | 1 | 1 | 0 | 1 | 1 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 0 | 0 | 1 | 1 | 0 |

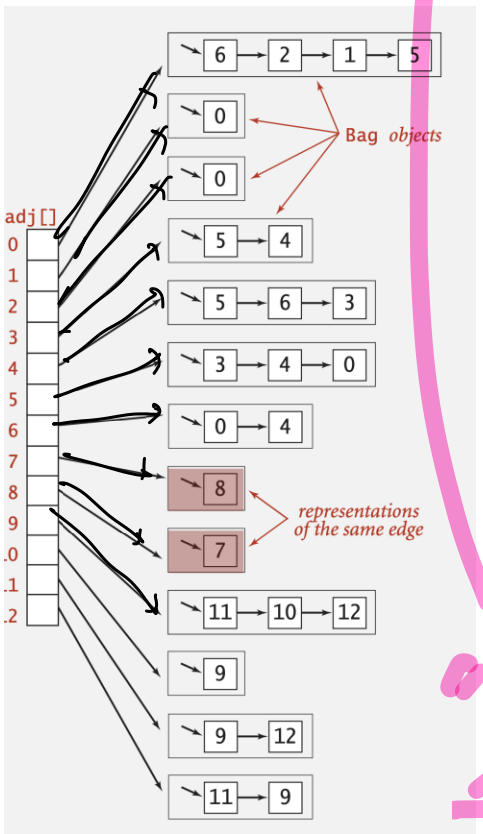
비 방향성 그래프!!

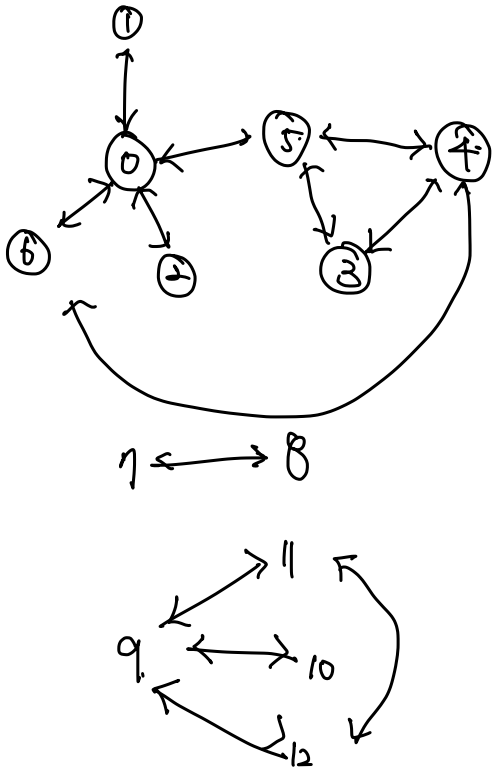



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 |

# 그래프 표현법

- 그래프 표현
  - Adjacency List
    인접 리스트

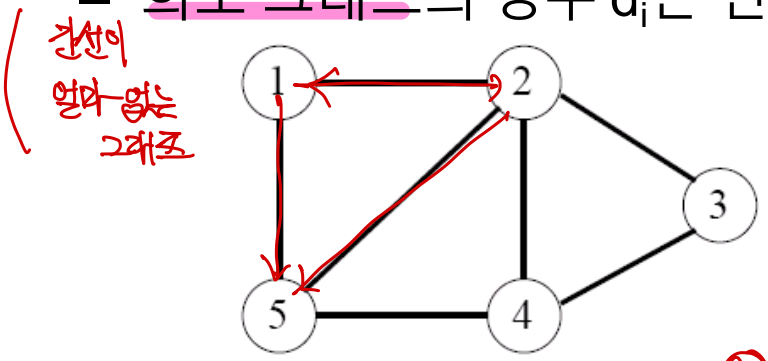


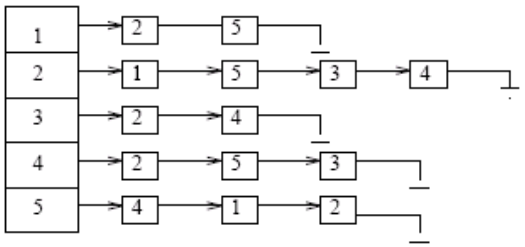




그래프로 구현 가능.

# 인접 리스트(Adjacency Lists)

- 인접 리스트
  - n개의 연결된 리스트 포인터로 이루어짐
  - i번째 연결된 리스트 포인터는 i번째 정점을 의미
  - i번째 연결된 포인터가 가리키는 리스트에 정점 j가 존재한다면, E(i, j)가 존재함을 의미
  - E(i, j)의 존재 여부를 찾기 위해서는 $O(d_i)$의 시간이 걸림 ($d_i$는 정점 i의 차수)
  - 희소 그래프의 경우 $d_i$는 전체 정점 수 n의 값보다 작을 수 있음

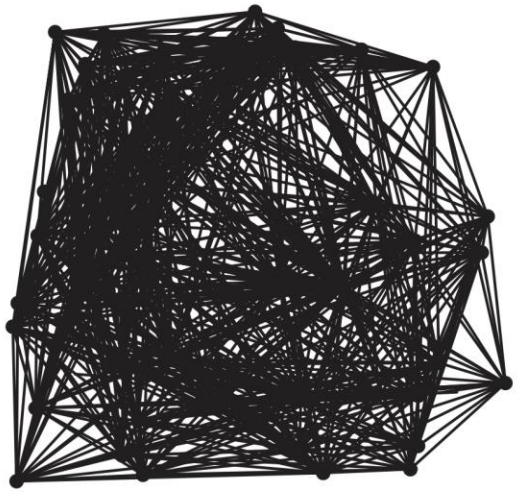


# 그래프 표현법

sparse (E = 200)

dense (E = 1000)



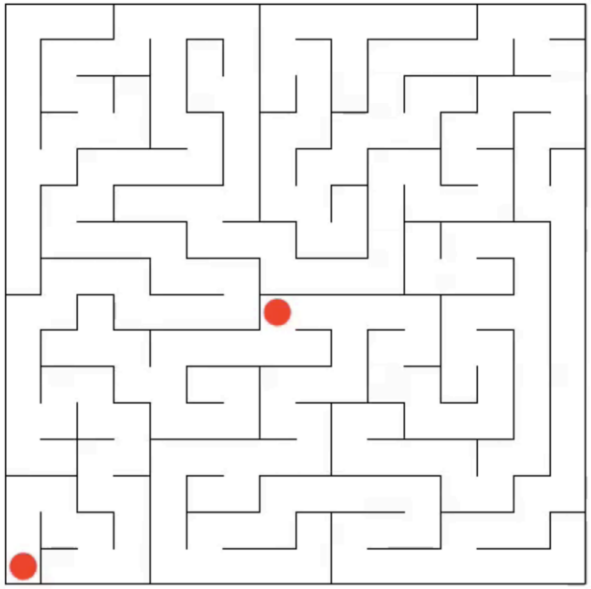
Two graphs (V = 50)

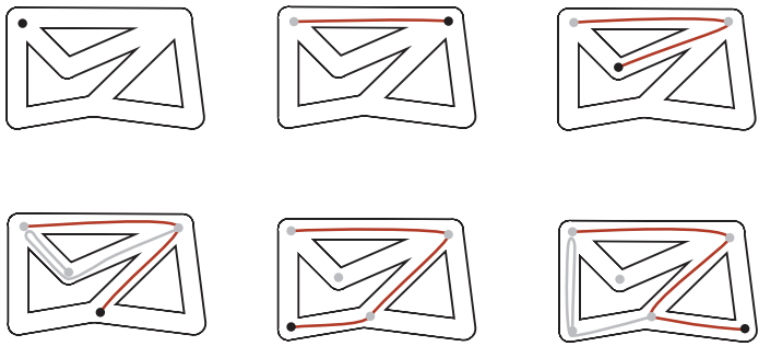
# 인접 리스트와 인접 행렬의 비교

| 비교 | 승자 |
|---|---|
| edge(x, y)를 찾는 시간 | 인접행렬 |
| 정점의 차수를 알아내는 시간 | 인접리스트 |
| 작은 그래프에서 더 작은 메모리를 사용하는 것 | 인접리스트 $(m + n)$ vs. $(n^2)$ |
| 큰 그래프에서 더 작은 메모리를 사용하는 것 | 인접행렬 (약간) |
| 에지를 삽입/삭제할 경우 | 인접행렬( O(1) ) |
| 그래프를 더 빨리 순회하는 것 | 인접리스트 $(m + n)$ vs. $(n^2)$ |
| 대부분의 문제에서 좋은 것 | 인접리스트 |

# 그래프 탐색

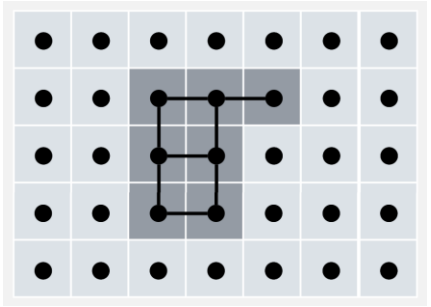
- 미로 탐색



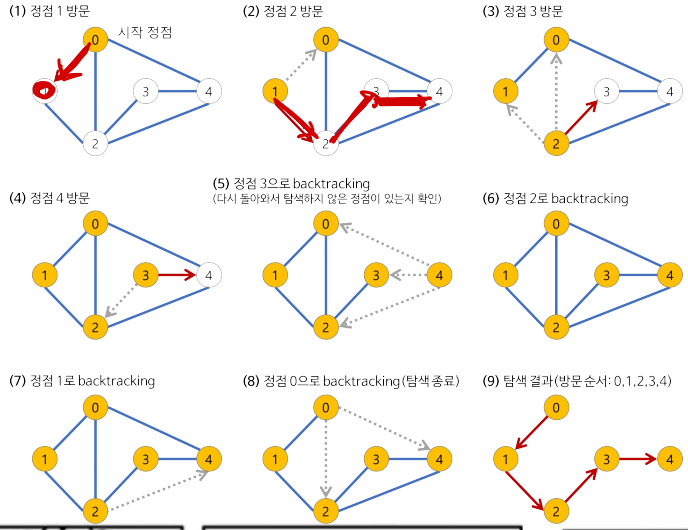


# 그래프 탐색

- Depth First Search
  - 방문한 vertex를 표시함
  - 방문하지 않은 vertex를 재귀호출로 방문함 (adjacency list 혹은 matrix 이용)
- DFS 응용
  - 포토샵에서 색 바꾸기
    - 그래프로 표현 (같은 색을 가진 경우 edge 생성)



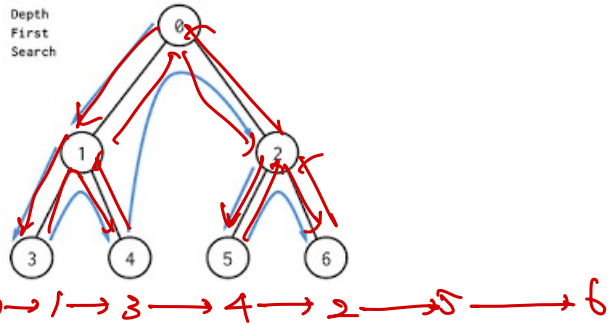
# DFS 응용





예시 참고하여

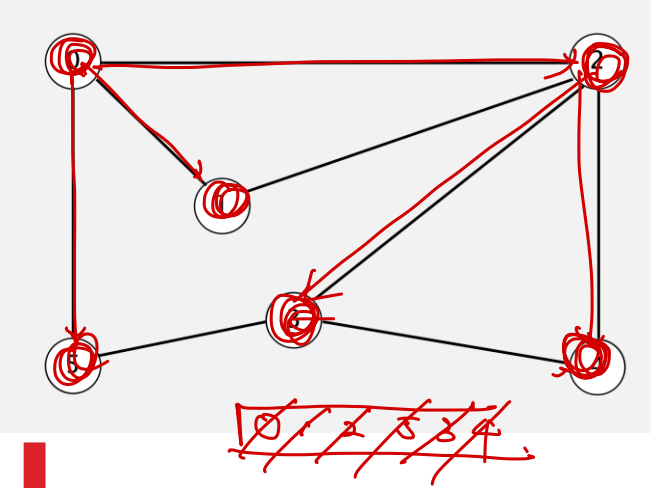
DFS (Depth)

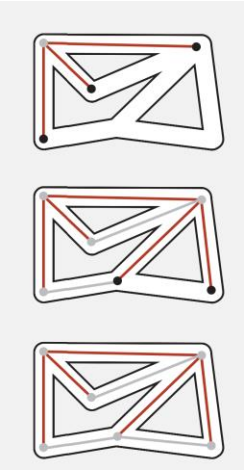



KNU

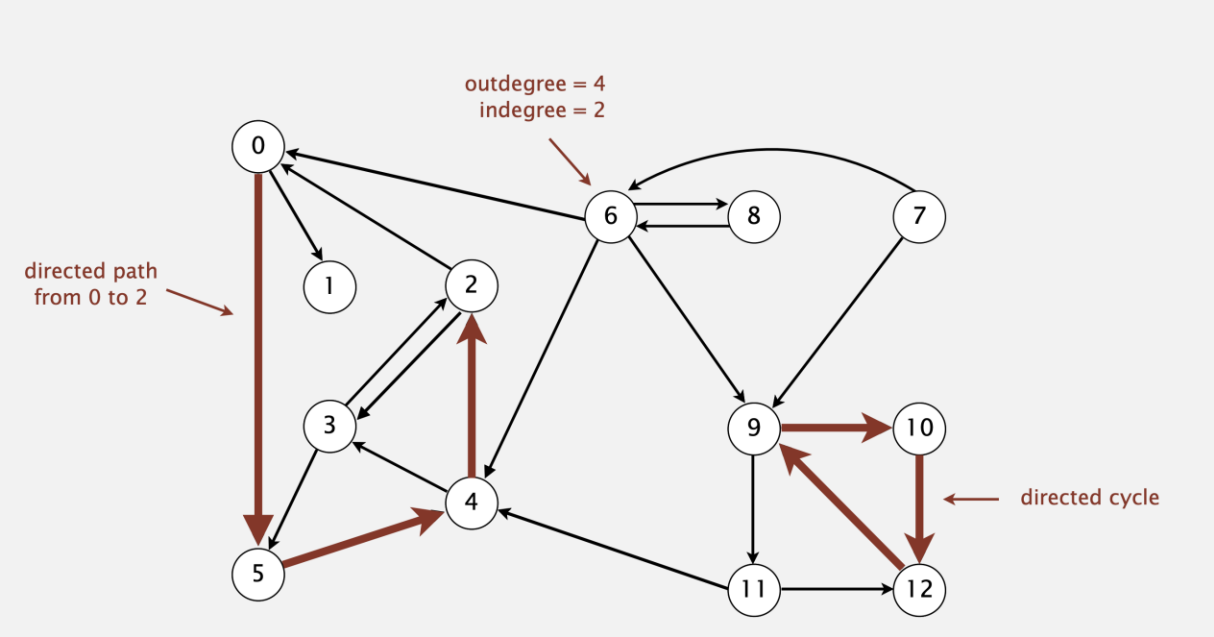
# 그래프 탐색 BFS

- Breadth First Search
  - 방문하지 않은 모든 인접 vertex를 큐에 넣고 다음 vertex를 방문
  - 방문한 vertex의 모든 인접 vertex를 다시 큐에 넣고 큐가 빌 때 까지 반복



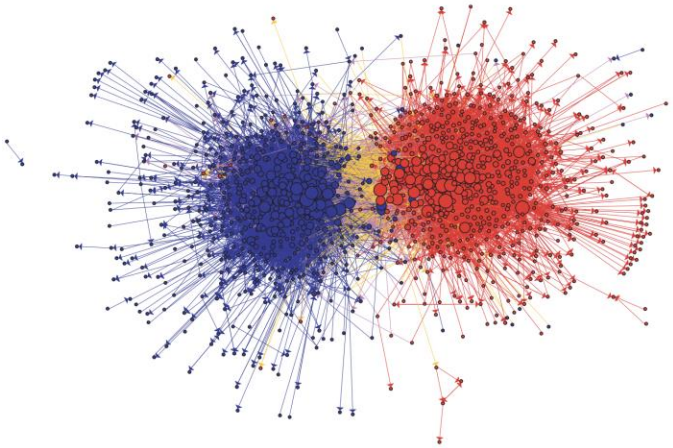


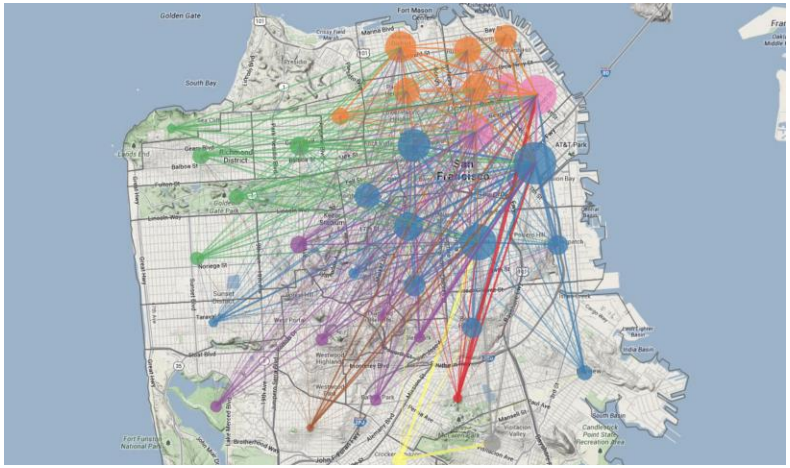
# Directed 그래프



# Directed 그래프

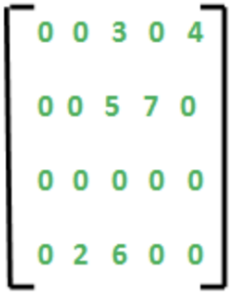
- Directed 그래프 예





# Directed 그래프 표현법

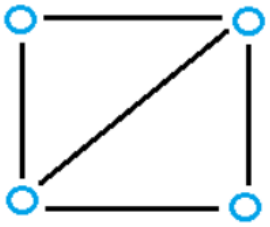
- Sparse Matrix 를 사용
  - [[0, 0, 3, 0, 4], [0, 0, 5, 7, 0],
  - [0, 0, 0, 0, 0], [0, 2, 6, 0, 0]]



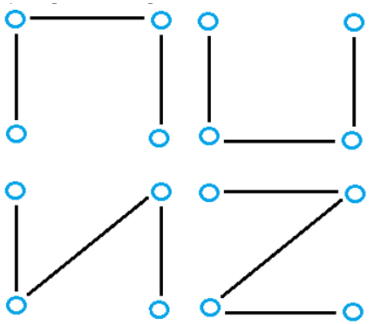

- 0을 채우는 것이 메모리 낭비, 다음과 같이 표현 가능
  - [('0', '2', 3), ('0', '4', 4), … ]

| Row | 0 | 0 | 1 | 1 | 3 | 3 |
|---|---|---|---|---|---|---|
| Column | 2 | 4 | 2 | 3 | 1 | 2 |
| Value | 3 | 4 | 5 | 7 | 2 | 6 |

# 최소신장트리 (MST)



- 신장트리 (Spanning Tree)란?
  - 원 그래프의 모든 노드를 포함
  - 모든 노드가 서로 연결
  - 트리 형태

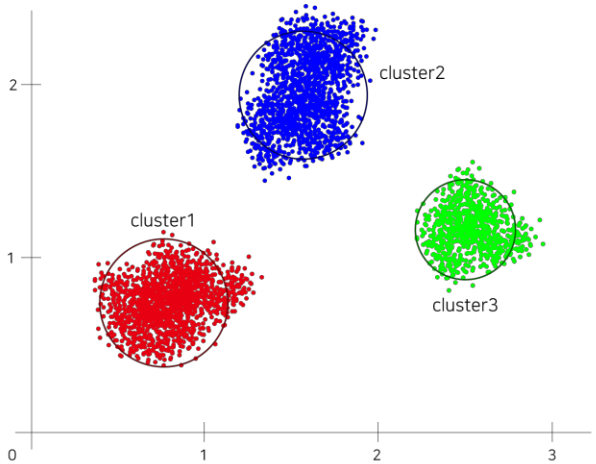


- 최소신장트리(Minimum Spanning Tree)란?
  - 가능한 신장트리 가운데 엣지 가중치의 합이 최소인 신장트리

# 최소신장트리의 활용

- 디더링
- 클러스터링
- 병목구간 찾기
- 얼굴 인식
- 에러 찾기
- 네트워크 자동설정
- 최단경로 찾기

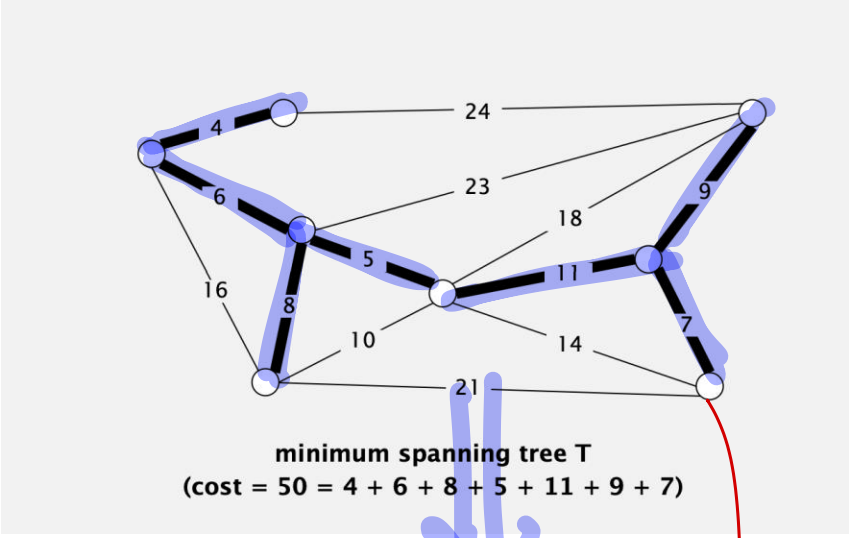
<디더링>




<클러스터링>

# MST

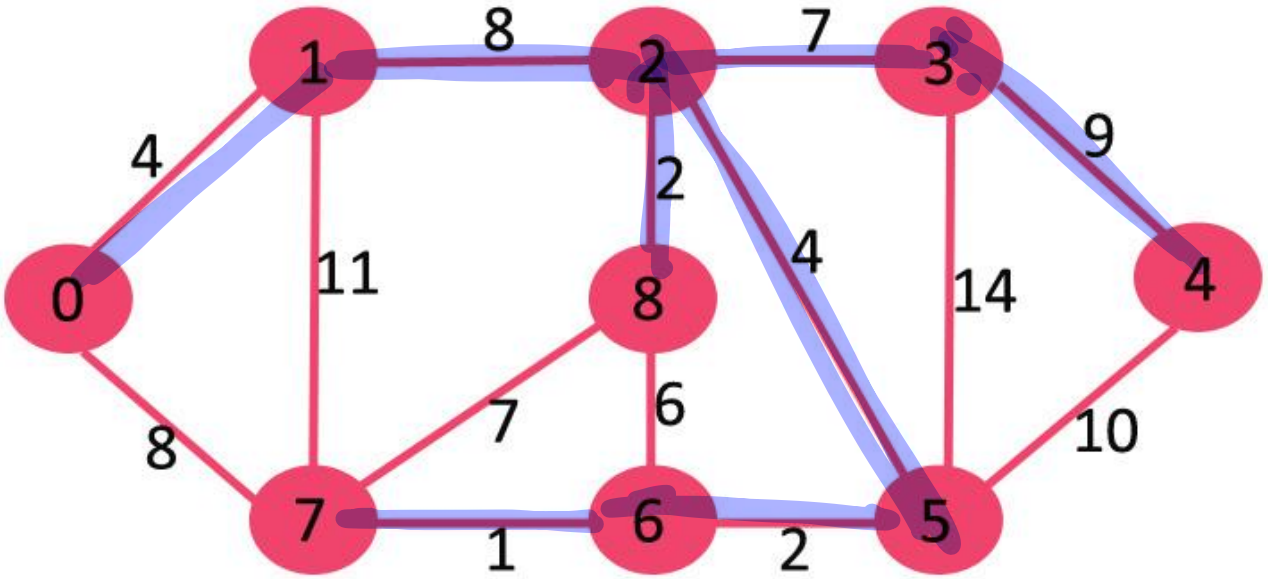
- MST를 찾는 방법




10+13+18+9
=23+27
=50

Kruskal Algorithm.

# Kruskal 알고리즘

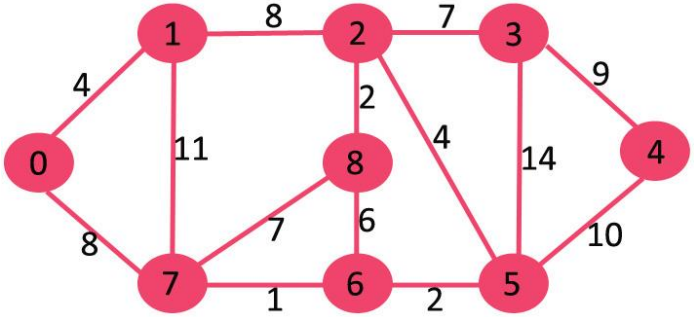



12+13+12

= 24+13

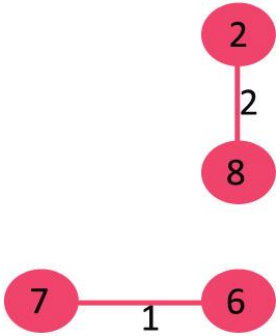
=37

# Kruskal 알고리즘

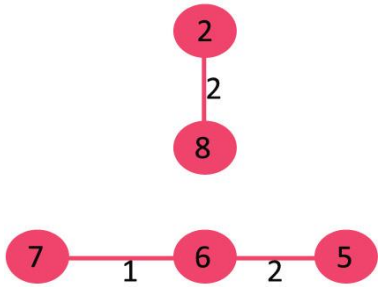



- 1

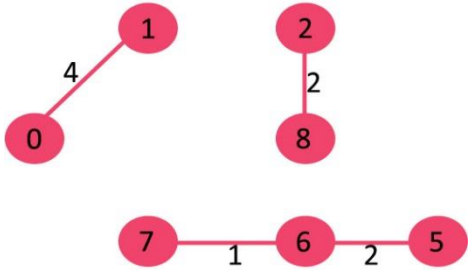

- 2



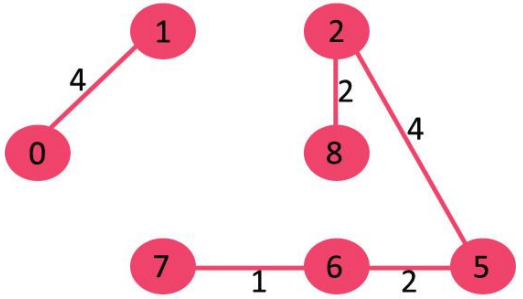

- 3



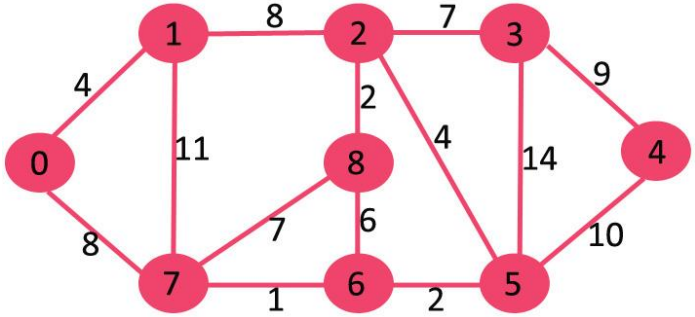

- 4



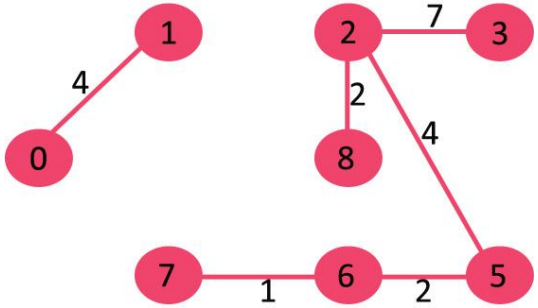

- 5



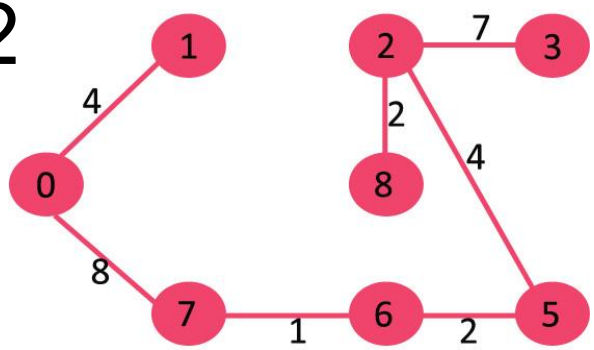
# Kruskal 알고리즘



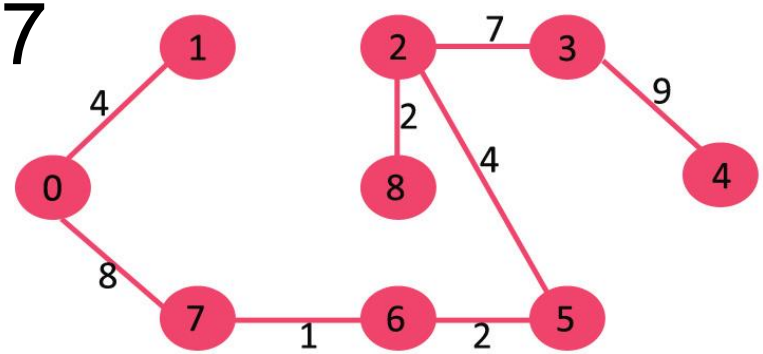

- 6




- 2



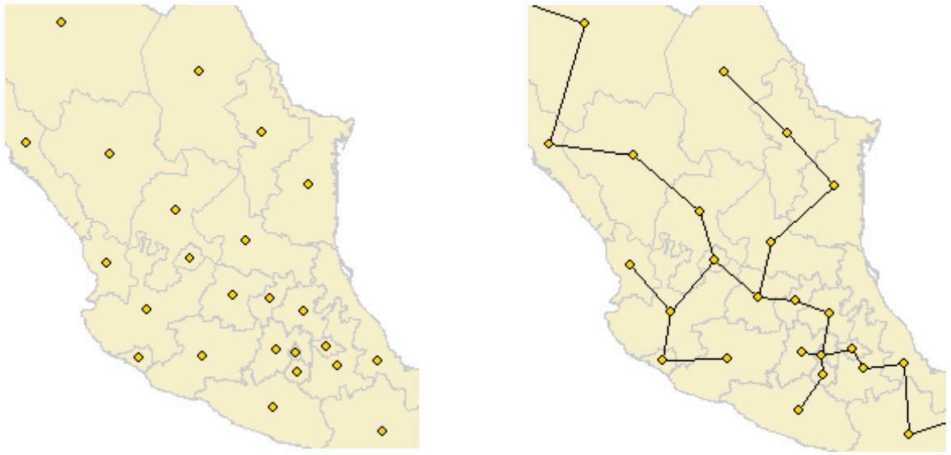

- 7




12+25
=37

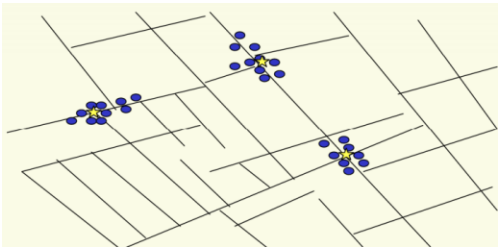
# Euclidean MST

- 주어진 N개의 점에서 MST를 구하고 각 점들의 거리를 Euclidean Distance라고 한다.



# 클러스터링

- K-Clustering
  - Divide a set of objects classify into k coherent groups.
  - Distance function: Euclidean dist.



- Applications
  - Routing in mobile ad hoc networks.
  - Document categorization for web search.
  - Similarity searching in medical image databases.
  - Skycat: cluster 109 sky objects into stars, quasars, galaxies.

# Questions?