6주차 알고리즘 세미나

PoolC 2021 1학기

지난시간 도전문제 – 동전 0

문제 요약 :

- $1 \leq N \leq 10$, $1 \leq K \leq 100{,}000{,}000$
- 동전의 종류는 총 N가지, 동전을 적절히 사용해서 합을 K로 맞춘다.
- $1 \leq A_i \leq 1{,}000{,}000$, $A_1 = 1$, $i \geq 2$인 경우에 A_i는 A_{i-1}의 배수
- K원을 맞추는데 필요한 동전의 개수 최소값?

고려해야 할 사항 :

- 완전탐색이나 DP?

지난시간 도전문제 – 동전 0

- 완전탐색이나 DP?
  - 완전탐색

    K가 크고 A_i가 작을 때, 시간이 많이 걸린다

  - DP

    cache[i] : i원을 만드는데 필요한 동전의 최소 개수

    i가 1억까지 갈 수 있으므로 DP 사용 불가

지난시간 도전문제 – 동전 0

- 그리디
  - **$i \geq 2$인 경우에 A_i는 A_{i-1}의 배수**
  - 120원을 만드는 경우
    - 10원 7개 + 50원 1개 vs 10원 2개 + 50원 2개

직관 : **A_{i-1}** 로 **A_i**를 무조건 만들 수 있으므로,

가치가 더 큰 **A_i**를 먼저 사용한다!

지난시간 도전문제 – 동전 0

- 증명

$A_1 = 1, A_2 = 5, A_3 = 10 , A_4 = 50 , A_5 = 100$ 인 경우

K = 72

A_1	A_2	A_3	A_4	A_5
2	**0**	**7**	**0**	**0**

- 탐욕적 선택 속성 : 보통 탐욕적 선택을 포함하지 않는 최적해의 존재를 가정하고,

  이 최적해를 적절히 조작하여 탐욕적 선택을 포함하는 최적해를 만들어서 증명

가정 : i-1번째 동전을 **(A_i / A_{i-1})** 보다 많이 사용하는 최적해가 존재한다!

동전의 개수는 무제한이어서 단순히 i번째 동전의 개수가 최대가 되게끔 바꾸면 되므로

항상 가치가 큰 동전부터 사용하는 최적해가 존재한다

지난시간 도전문제 – 동전 0

- 실제 풀이

```
for(int i=N-1; i>=0; i--)
    if(k == 0)
        break
    if(k / coin[i] == 0)
        continue
    ans += k / coin[i]
    k -= (k / coin[i]) * coin[i]
```

지난시간 도전문제 – 최소비용 구하기

문제 요약 :

- $1 \leq N \leq 1{,}000$, $1 \leq M \leq 100{,}000$
- $0 \leq$ 버스비용 $\leq 100{,}000$
- 도시가 정점, 버스가 간선

고려해야 할 사항 :

- 전형적인 1:N 최단거리 구하는 문제, 다익스트라 사용
- $O(E\log V)$

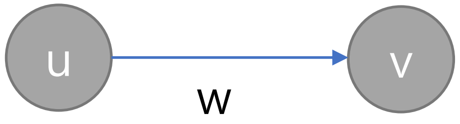
지난시간 도전문제 – 최소비용 구하기

- 방향 그래프의 표현 방법

vector<pair<int, int> > linked[N]

간선 비용　　　연결된 노드 번호

(u, v, w)　　->



->　　linked[u] : {w,v}

```
void dijkstra()
{
    pq.push(make_pair(0,start));
    dist[start] = 0;
    while (!pq.empty())
    {
        int cost = -pq.top().first;
        int here = pq.top().second;
        pq.pop();
        if (dist[here] < cost)
            continue;
        for (int i = 0; i < linked[here].size(); i++)
        {
            int there = linked[here][i].first;
            int nextDist = cost + linked[here][i].second;
            if (dist[there] > nextDist)
            {
                dist[there] = nextDist;
                pq.push(make_pair(-nextDist, there));
            }
        }
    }
}
```

지난시간 도전문제 – 숨바꼭질 3

문제 요약 :

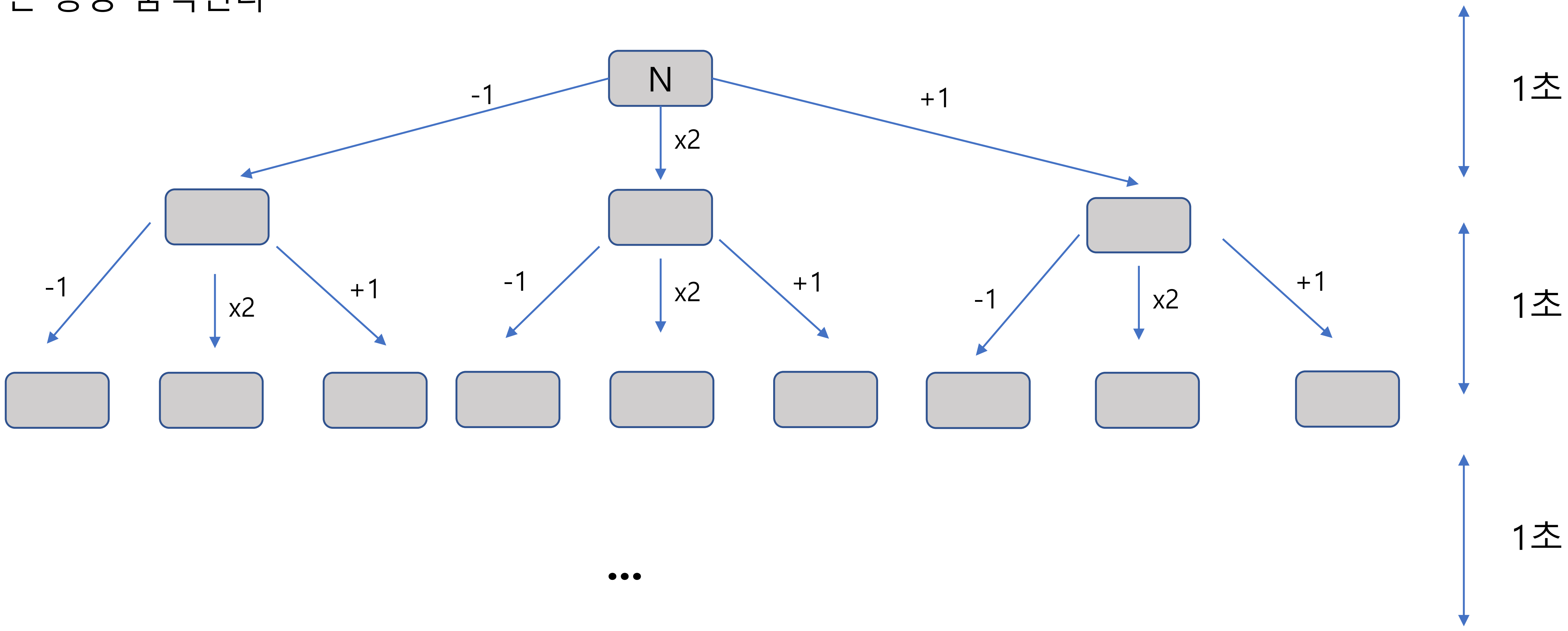
- $0 \le N \le 100{,}000$, $0 \le K \le 100{,}000$
- 걷는데는 1초, 순간이동은 0초가 걸린다
- 현재 위치가 X일때, 걷는다면 X+1, X-1로 이동, 순간이동은 2*X로 이동
- 동생을 찾을 수 있는 최단 시간은?

고려해야 할 사항 :

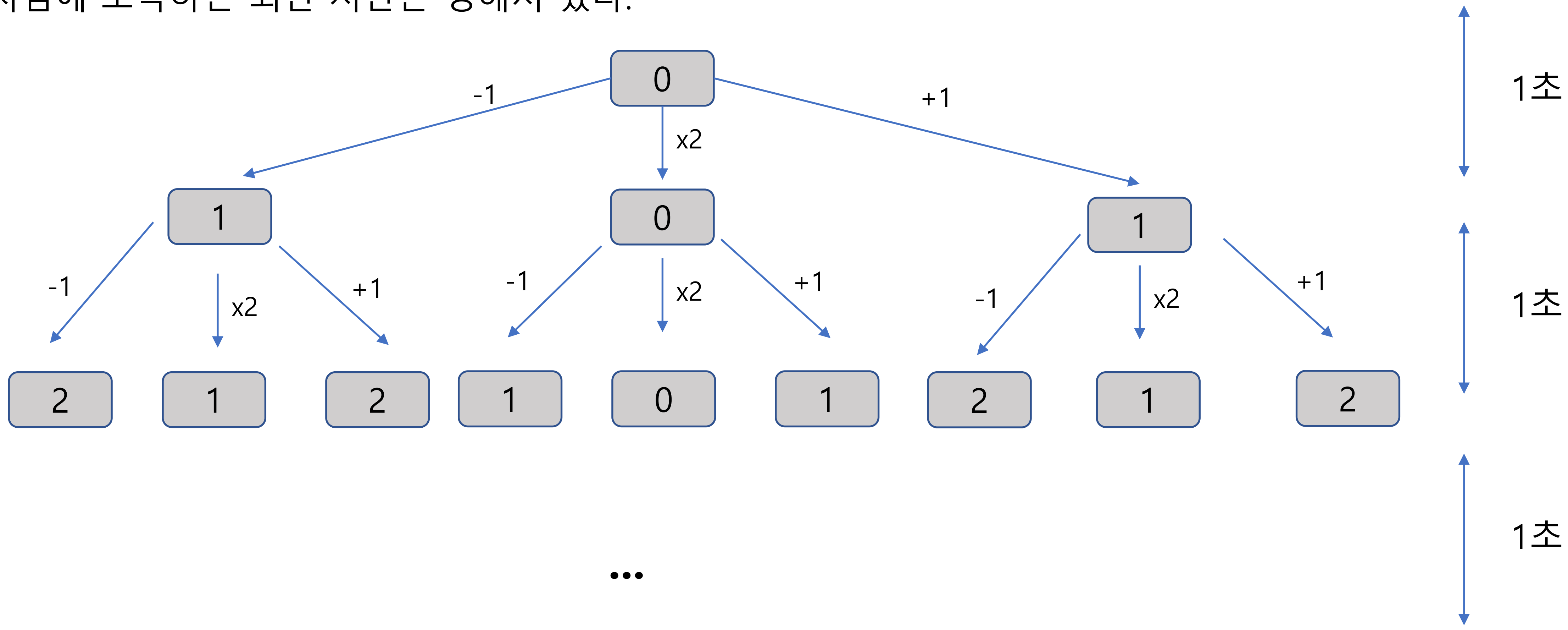
- 1697번 숨바꼭질과 다른점은?

지난시간 도전문제 – 숨바꼭질 3

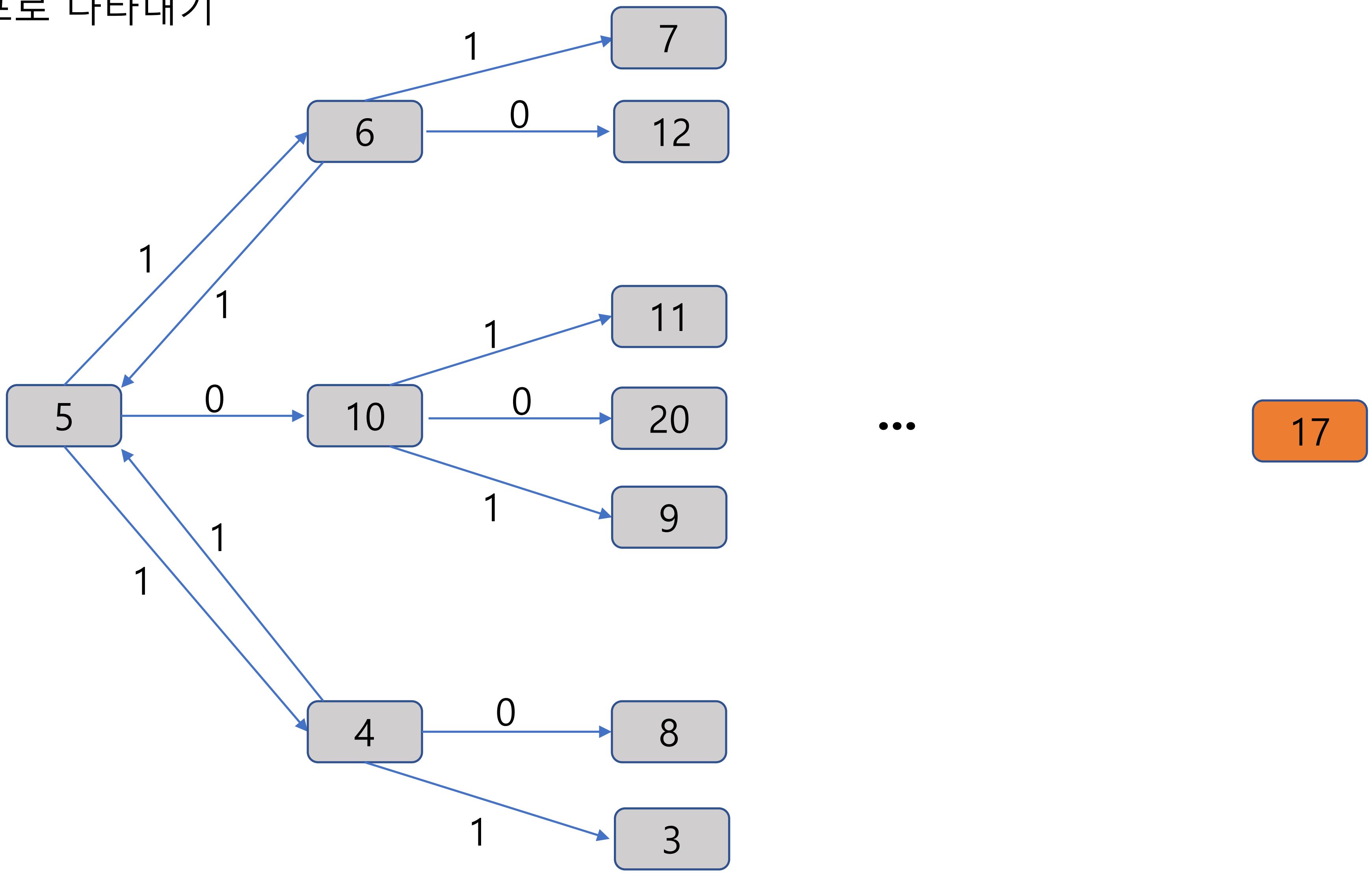
- 수빈이는 항상 움직인다



- 어떤 지점에 도착하는 최단 시간은 정해져 있다.



- 그래프로 나타내기



지난시간 도전문제 – 숨바꼭질 3

- 실제 구현

```
void dijk(int start)
    pq.Enqueue(0, start)
    dist[start] = 0
    while(!pq.empty())
        int cost = -pq.top().front()
        int here = pq.top().second()
        if(dist[here] < cost) continue
        if(inRange(here+1) && dist[here+1] > cost + 1)
            dist[here+1] = cost + 1
            pq.Enqueue(-cost+1,here+1)
        if(inRange(here-1) && dist[here-1] > cost + 1)
            dist[here-1] = dist[here] + 1
            pq.Enqueue(-cost+1,here-1)
        if(inRange(here*2) && dist[here*2] > cost)
            dist[here*2] = cost
            pq.Enqueue(-cost,here*2)
```

```
dijk(N)

cout<<dist[K]
```

지난시간 도전문제 – 미로 만들기

문제 요약 :

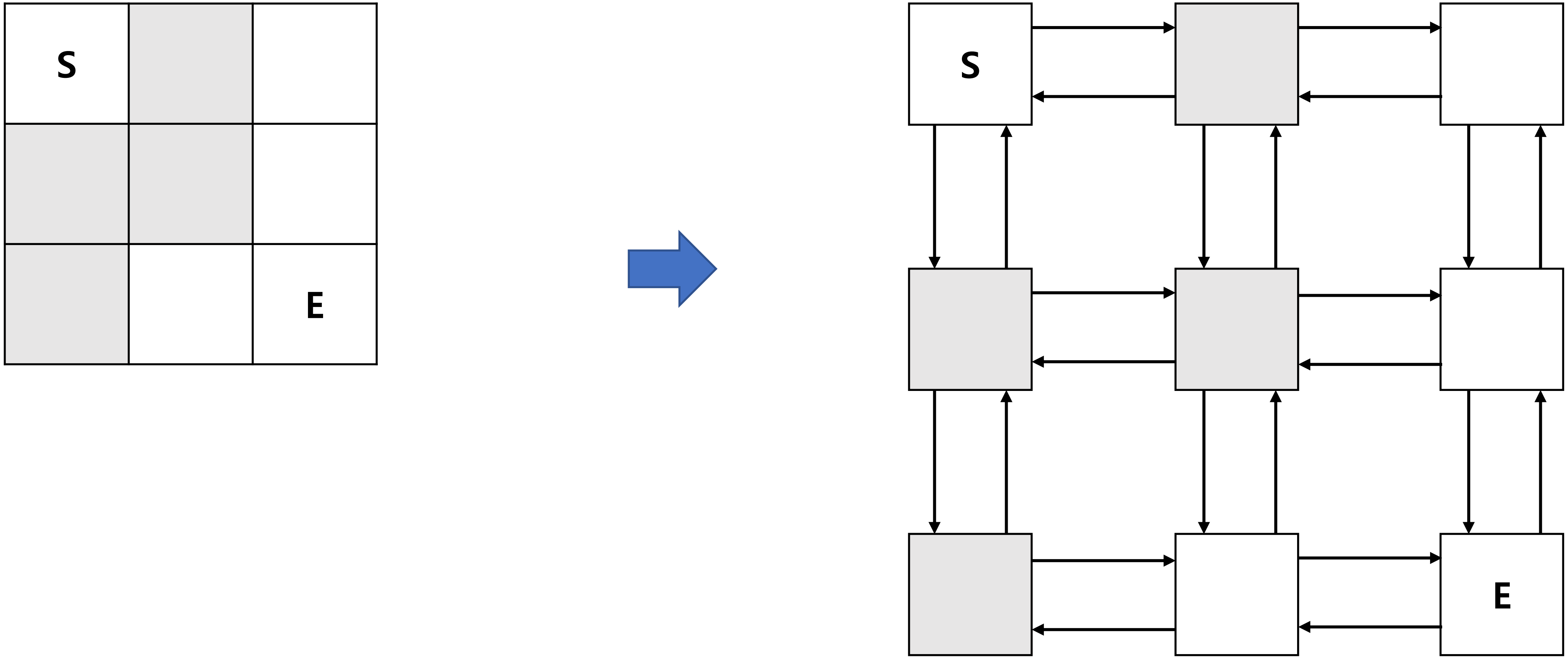
- $1 \le N \le 50$
- 0은 검은 방, 1은 흰 방
- 검은 방은 일종의 벽으로 존재하고, 흰 방으로 임의로 바꿀 수 있다.
- (0,0)에서 (N-1,N-1)까지 갈 때, 흰 방으로 바꾸어야 할 검은 방의 최소 개수는?

고려해야 할 사항 :

- 문제 상황을 그래프로 나타낼 수 있지 않을까?

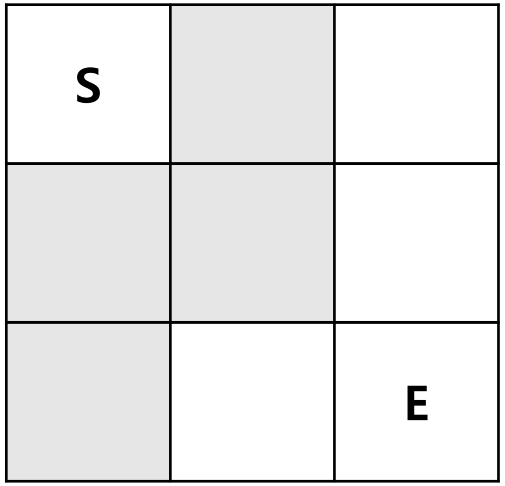
지난시간 도전문제 – 미로 만들기

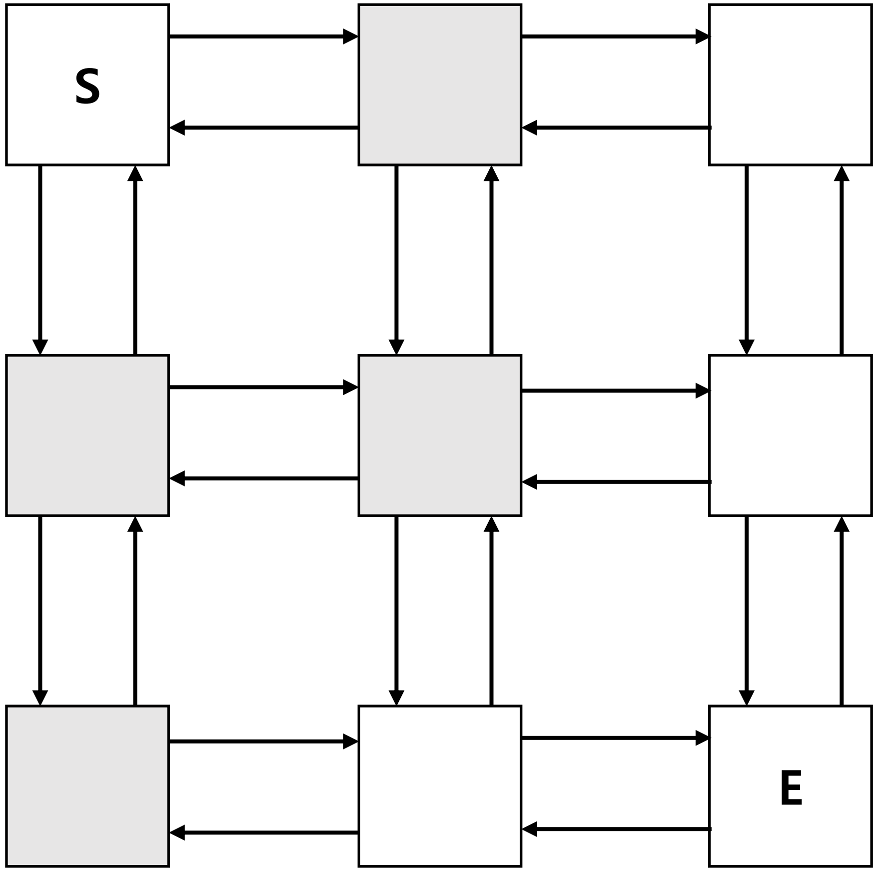
- 문제 상황을 그래프로 나타낼 수 있지 않을까?



지난시간 도전문제 – 미로 만들기

- 간선의 비용 설정?





1. 검은 방으로 가려면 흰 방으로 바꿔줘야 한다.

2. (흰 방으로 바꿔야 할 검은 방의 최소 개수)

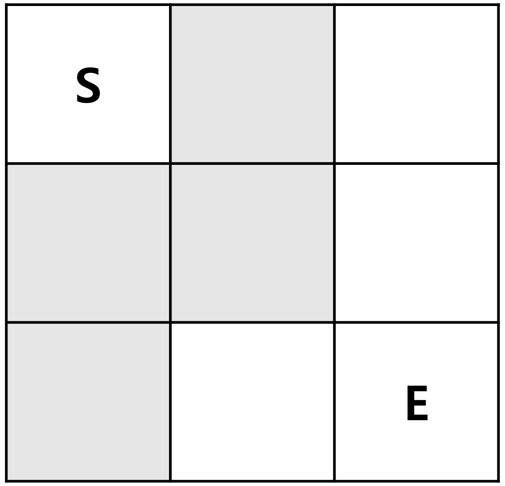
=

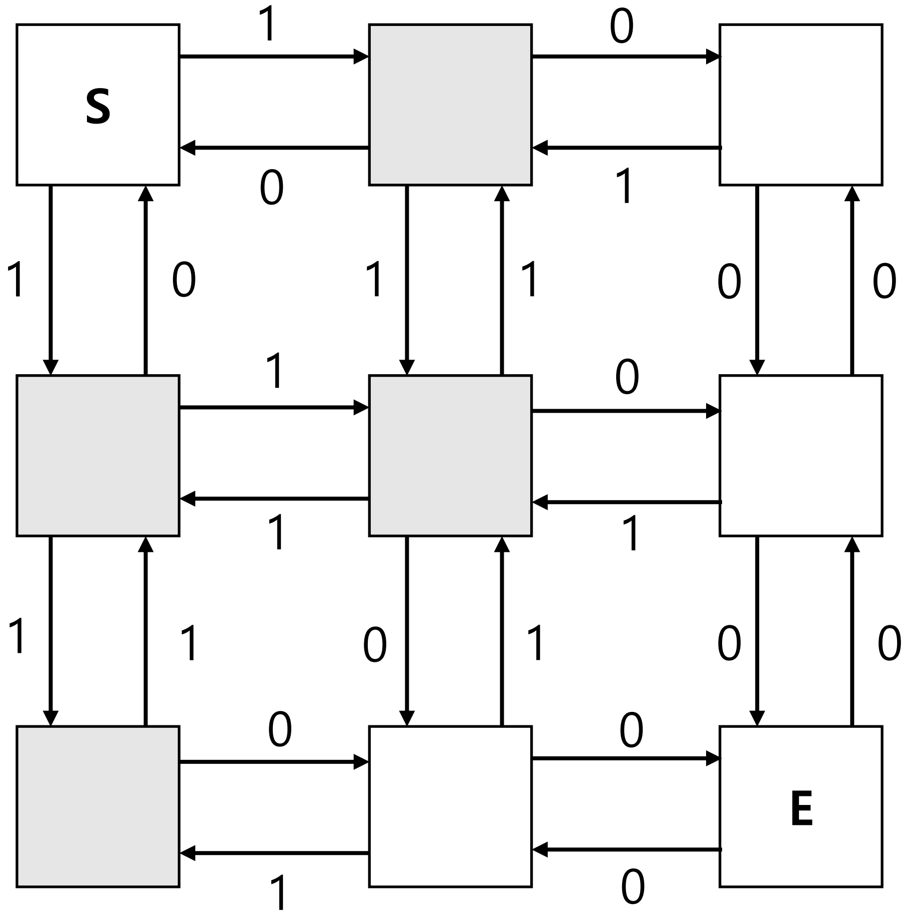
(S에서 E까지 최단 거리)

단, 물리적인 최단거리가 아닌 비용이 최소가 되는

지난시간 도전문제 – 미로 만들기

- 간선의 비용 설정!





지난시간 도전문제 – 미로 만들기

- 실제 구현

4방향 탐색 도중 0을 만나면

- 간선의 비용을 1로 잡아준다

4방향 탐색 도중 1을 만나면

- 간선의 비용을 0으로 잡아준다

```
void dijk()
{
    pq.push(make_pair(0, make_pair(0, 0)));
    dist[0][0] = 0;
    while (!pq.empty())
    {
        int cost = pq.top().first;
        int y = pq.top().second.first;
        int x = pq.top().second.second;
        pq.pop();
        if (dist[y][x] < cost)
            continue;
        for (int i = 0; i < 4; i++)
        {
            if (inRange(y + dy[i], x + dx[i]))
            {
                if (board[y + dy[i]][x + dx[i]] == '0')
                {
                    if (dist[y + dy[i]][x + dx[i]] > cost + 1)
                    {
                        dist[y + dy[i]][x + dx[i]] = cost + 1;
                        pq.push(make_pair(cost + 1, make_pair(y + dy[i], x + dx[i])));
                    }
                }
                else
                {
                    if (dist[y + dy[i]][x + dx[i]] > cost)
                    {
                        dist[y + dy[i]][x + dx[i]] = cost;
                        pq.push(make_pair(cost, make_pair(y + dy[i], x + dx[i])));
                    }
                }
            }
        }
    }
}
```

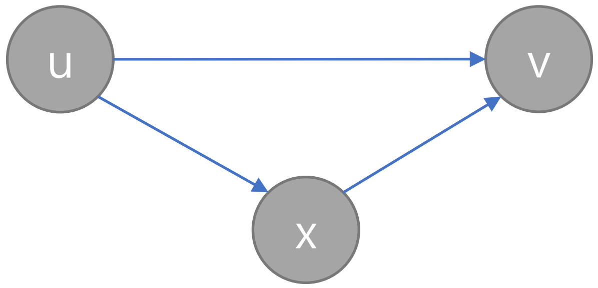
플로이드-워셜

- Floyd-Warshall(플로이드-워셜)
    - BFS, 다익스트라, 벨만포드와 같은 최단거리 알고리즘
    - 1:N이 아닌 **N:N**
    - 음의 가중치가 있어도 사용 가능 / 구현이 쉽다
    - 인접행렬로 나타낸 그래프에서 사용
    - N개의 정점이 있을 때, O(N^3)의 시간복잡도

- 언제 써야 할까?
    - N:N 최단거리 구할 때
    - 각 정점 간의 도달 가능성 여부를 판단해야 할 때

플로이드-워셜

- 아이디어의 출발점
  - 경유점 : u -> v 로 가는 경로중에 u -> x -> v 같이 거쳐가는 x점을 경유점이라 한다.
  - $D_{S_k}(u,v)$: 정점 집합 S_k = {0, 1, 2, … , k}에 포함된 정점만을 경유점으로 사용해 u -> v로 가는 최단경로
  - $D_{S_k}(u,v)$의 정의?
    1. $D_{S_k}(u,v)$는 S_k 에 속하는 어떤 점 k를 경유하지 않는다
       - S_{k-1} 에 포함된 정점들 만을 경유점으로 사용한다.
       - $D_{S_k}(u,v) = D_{S_{k-1}}(u,v)$
    2. $D_{S_k}(u,v)$는 S_k 에 속하는 어떤 점 k를 경유한다.
       - 경로를 u -> k 와 k -> v로 나눌 수 있다.
       - $D_{S_k}(u,v) = D_{S_{k-1}}(u,k) + D_{S_{k-1}}(k,v)$



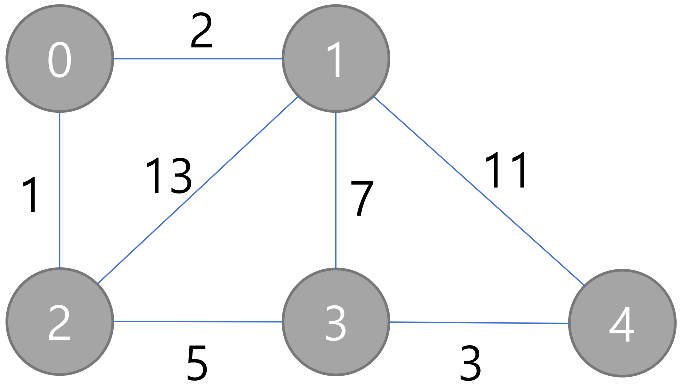
$$D_{S_k}(u,v) = \min(D_{S_{k-1}}(u,v),\ D_{S_{k-1}}(u,k) + D_{S_{k-1}}(k,v))$$

플로이드-워셜

- 실제 코드 구현

```
void floyd()
for(int k=0; k<N; k++)
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

- 동작 과정



dist	0	1	2	3	4
0	0	2	1	INF	INF
1	2	0	13	7	11
2	1	13	0	5	INF
3	INF	7	5	0	3
4	INF	11	INF	3	0

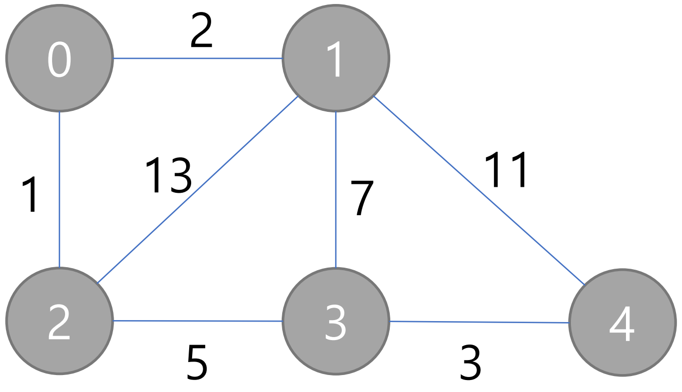
k :

i :

j :

플로이드-워셜

- 동작 과정



dist	0	1	2	3	4
0	0	2	1	INF	INF
1	2	0	13	7	11
2	1	13	0	5	INF
3	INF	7	5	0	3
4	INF	11	INF	3	0

k : 0

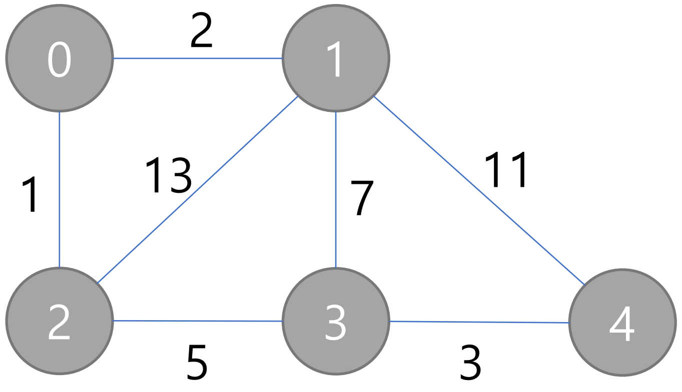
dist[0][0] = min(dist[0][0], dist[0][0] + dist[0][0])

i : 0

j : 0

플로이드-워셜

- 동작 과정



dist	0	1	2	3	4
0	0	2	1	INF	INF
1	2	0	13	7	11
2	1	13	0	5	INF
3	INF	7	5	0	3
4	INF	11	INF	3	0

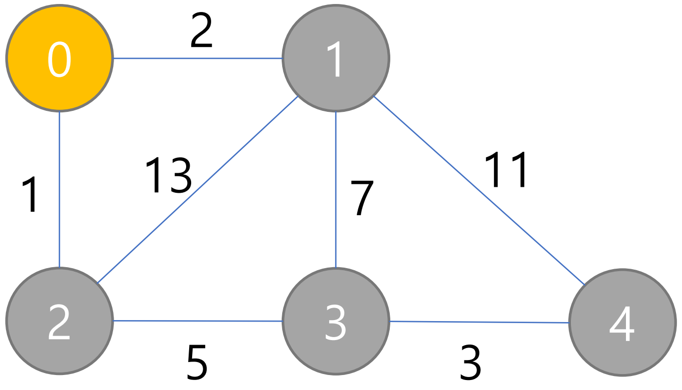
k : 0

i : 0

j : 1

dist[0][1] = min(dist[0][1], dist[0][0] + dist[0][1])

- 동작 과정



dist	0	1	2	3	4
0	0	2	1	INF	INF
1	2	0	3	7	11
2	1	13(3)	0	5	INF
3	INF	7	5	0	3
4	INF	11	INF	3	0

k : 0

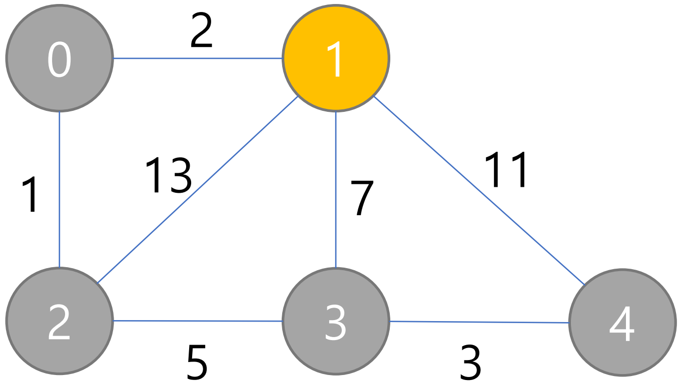
i : 1

j : 2

dist[1][2] = min(dist[1][2], dist[1][0] + dist[0][2])

dist[1][2] = 3으로 갱신

- 동작 과정



dist	0	1	2	3	4
0	0	2	1	9	INF
1	2	0	3	7	11
2	1	3	0	5	INF
3	INF(9)	7	5	0	3
4	INF	11	INF	3	0

k : 1

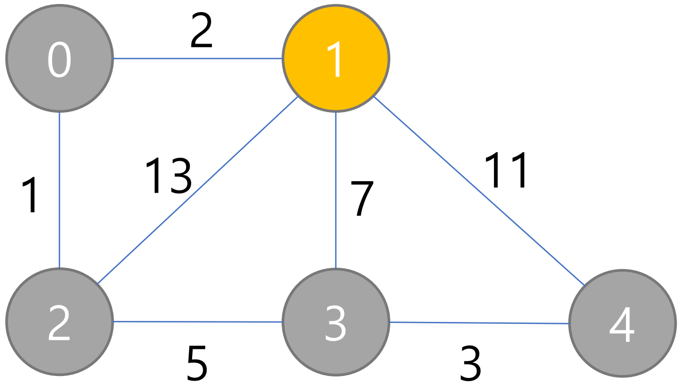
i : 0

j : 3

dist[0][3] = min(dist[0][3], dist[0][1] + dist[1][3])

dist[0][3] = 9로 갱신

- 동작 과정



k : 1

i : 0

j : 4

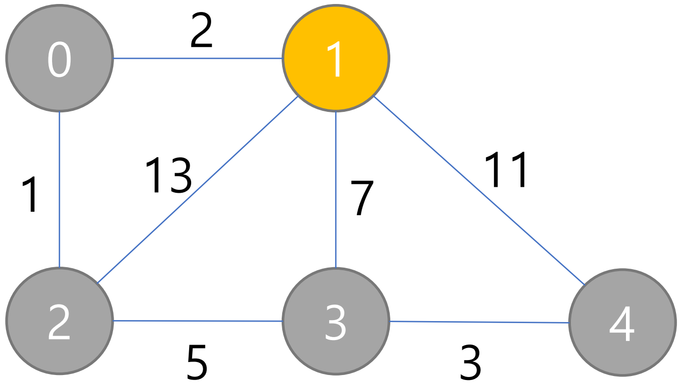
dist[0][4] = min(dist[0][4], dist[0][1] + dist[1][4])

dist[0][4] = 13으로 갱신

dist	0	1	2	3	4
0	0	2	1	9	13
1	2	0	3	7	11
2	1	3	0	5	INF
3	9	7	5	0	3
4	INF(13)	11	INF	3	0

플로이드-워셜

- 동작 과정



dist	0	1	2	3	4
0	0	2	1	9	13
1	2	0	3	7	11
2	1	3	0	5	14
3	9	7	5	0	3
4	13	11	INF(14)	3	0

k : 1

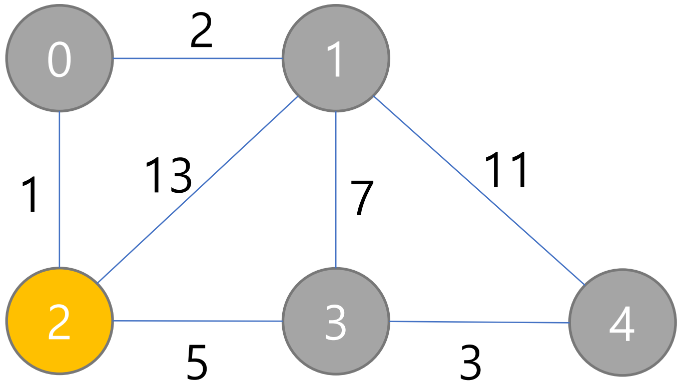
i : 2

j : 4

dist[2][4] = min(dist[2][4], dist[2][1] + dist[1][4])

dist[2][4] = 14로 갱신

- 동작 과정



dist	0	1	2	3	4
0	0	2	1	6	13
1	2	0	3	7	11
2	1	3	0	5	14
3	9(6)	7	5	0	3
4	13	11	14	3	0

k : 2

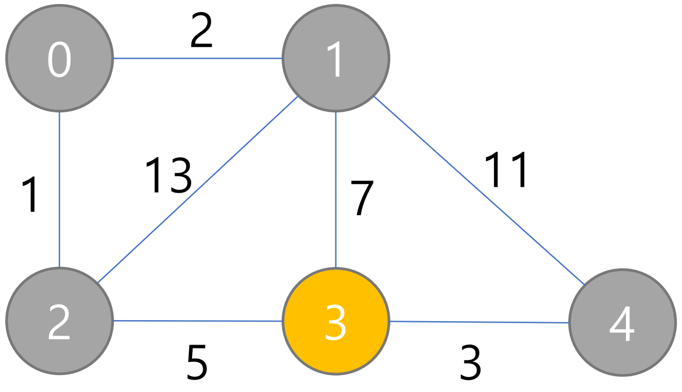
i : 0

j : 3

dist[0][3] = min(dist[0][3], dist[0][2] + dist[2][3])

dist[0][3] = 6으로 갱신

- 동작 과정



k : 3

i : 0

j : 4

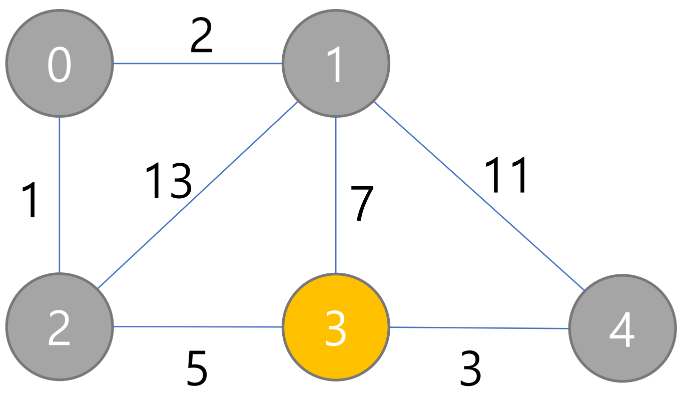
dist[0][4] = min(dist[0][4], dist[0][3] + dist[3][4])

dist[0][4] = 9로 갱신

dist	0	1	2	3	4
0	0	2	1	6	9
1	2	0	3	7	11
2	1	3	0	5	14
3	6	7	5	0	3
4	13(9)	11	14	3	0

플로이드-워셜

- 동작 과정



k : 3

i : 1

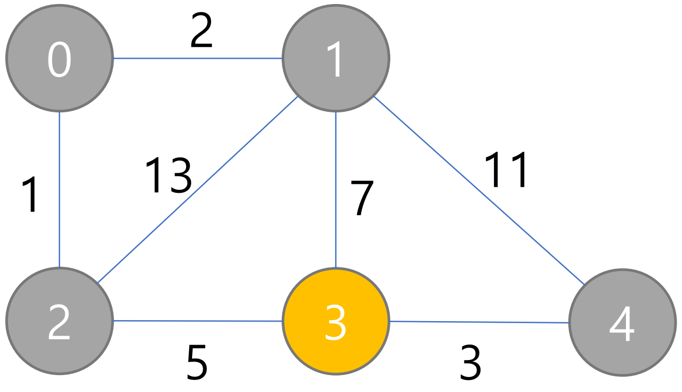
j : 4

dist[1][4] = min(dist[1][4], dist[1][3] + dist[3][4])

dist[1][4] = 10으로 갱신

dist	0	1	2	3	4
0	0	2	1	6	9
1	2	0	3	7	10
2	1	3	0	5	14
3	6	7	5	0	3
4	9	11(10)	14	3	0

- 동작 과정



k : 3

i : 2

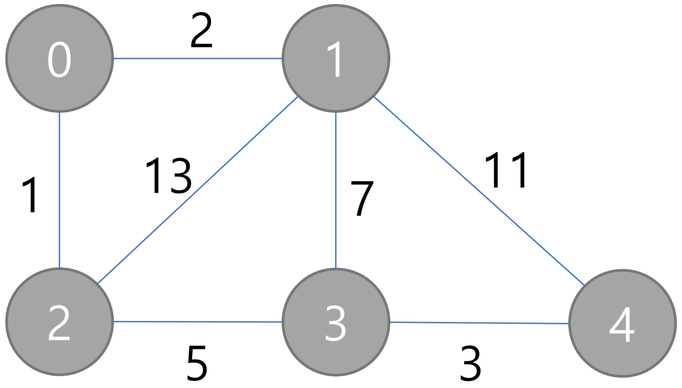
j : 4

dist[2][4] = min(dist[2][4], dist[2][3] + dist[3][4])

dist[2][4] = 8로 갱신

dist	0	1	2	3	4
0	0	2	1	6	9
1	2	0	3	7	10
2	1	3	0	5	8
3	6	7	5	0	3
4	9	10	14(8)	3	0

- 실행 결과



dist	0	1	2	3	4
0	0	2	1	6	9
1	2	0	3	7	10
2	1	3	0	5	8
3	6	7	5	0	3
4	9	10	8	3	0

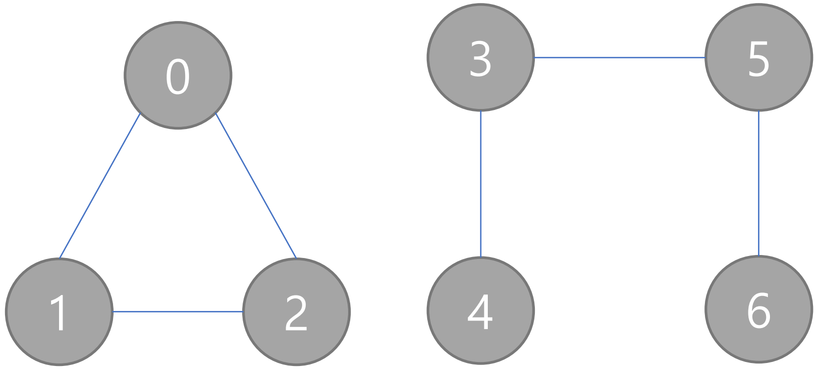
각 정점 간의 도달 가능성 여부를 판단해야 할 때 ?

- 최단거리가 존재하면 도달 가능
- 도전문제

- 최단거리 알고리즘 정리

	BFS	다익스트라	플로이드
언제?	1:N		N:N
가중치?	0 또는 1	0 또는 양수	Free
사용 자료구조	큐	우선순위 큐	x
시간복잡도	O(V+E)	O(ElogV)	O(N^3)
dist 초기화	-1	INF	INF
구현 난이도	어렵다		쉽다

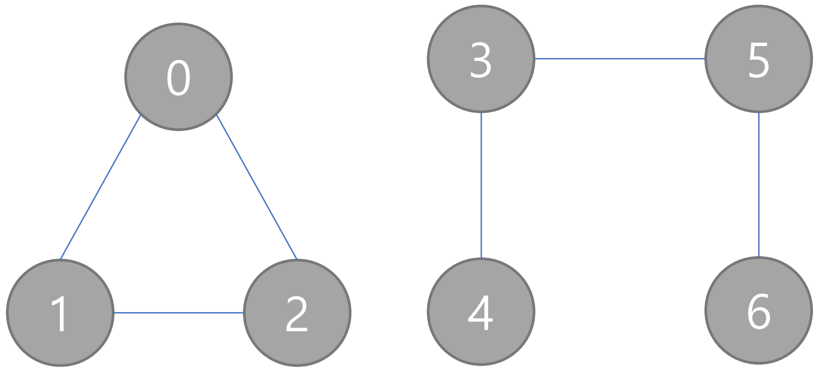
- Union-Find(유니온-파인드)
  - 유니온-파인드, 상호 배타적 집합, Disjoint-set, 분리 집합 모두 같은 말!
  - 그래프상의 어떤 두 노드가 **같은 집합**에 속해 있는지 확인하는 알고리즘



- 0번이랑 4번이 같은 그래프(집합)에 속해 있는가?

유니온-파인드

- Union-Find(유니온-파인드)



0번이랑 4번이 같은 그래프(집합)에 속해 있는가?

- 0번을 기준으로 그래프 탐색 ? DFS? BFS?

시간이 너무 오래걸린다(N에 비례)

Union-Find를 사용하면?

- N의 크기와는 상관없이 거의 상수시간만에 판단 가능
- Find(0)과 Find(4)만 비교해주면 된다

유니온-파인드

- Union-Find(유니온-파인드)

Union : 두 원소 a, b가 주어질 때, 이들이 속한 두 집합을 하나로 합친다

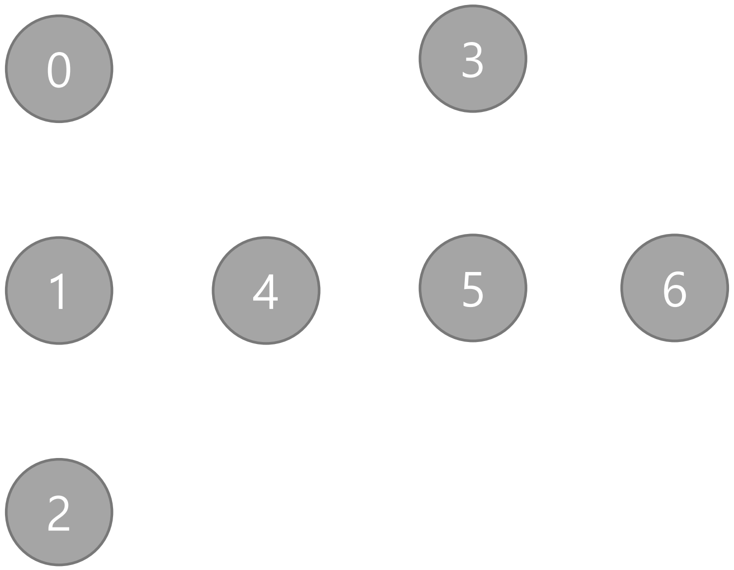
Find : 어떤 원소 a가 주어질 때, 이 원소가 속한 집합을 반환한다

- parent 배열을 통해 알고리즘을 구현한다

parent[x] : x번째 노드의 부모를 저장

parent[i] = i로 미리 초기화 해주어야 함!

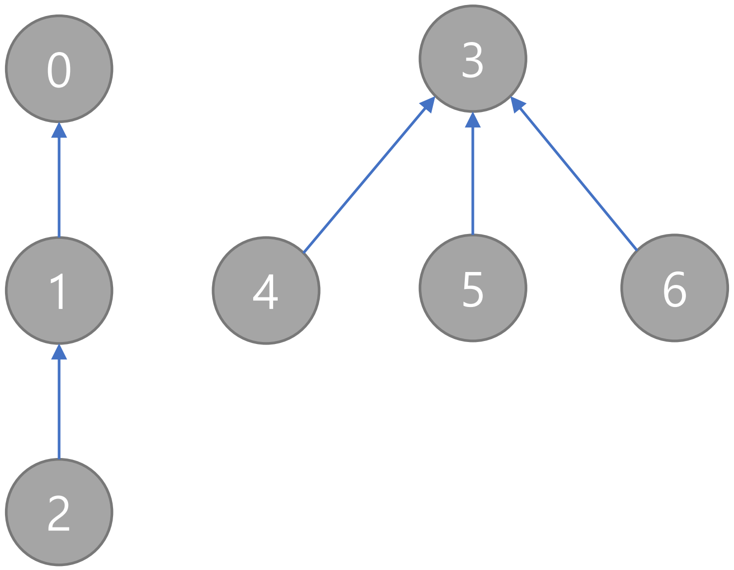
- Union-Find(유니온-파인드)



	0	1	2	3	4	5	6
parent	**0**	**1**	**2**	**3**	**4**	**5**	**6**

여기에 유니온-파인드를 적용 시키면?

- Union-Find(유니온-파인드)



집합 A(0)　　집합 B(3)

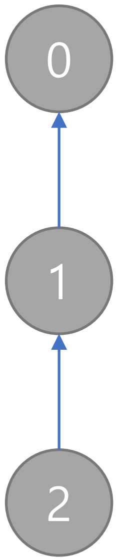
	0	1	2	3	4	5	6
parent	**0**	**0**	**1**	**3**	**3**	**3**	**3**

트리로 나타낸 다음, 바로 위의 노드를 가리킨다.

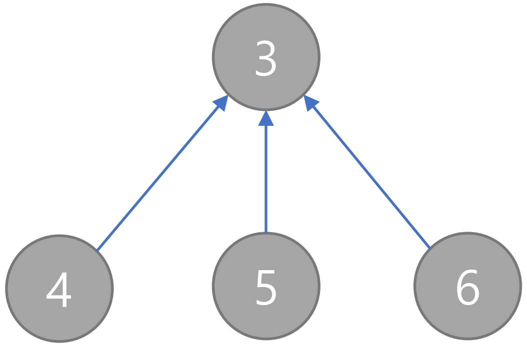
트리의 최상위 노드라면 자기 자신을 가리킨다.

- Union-Find(유니온-파인드)

Find : 어떤 원소 a가 주어질 때, 이 원소가 속한 집합의 최상위 노드를 반환한다



집합 A(0)



집합 B(3)

	0	1	2	3	4	5	6
parent	0	0	1	3	3	3	3
find	0	0	0	3	3	3	3

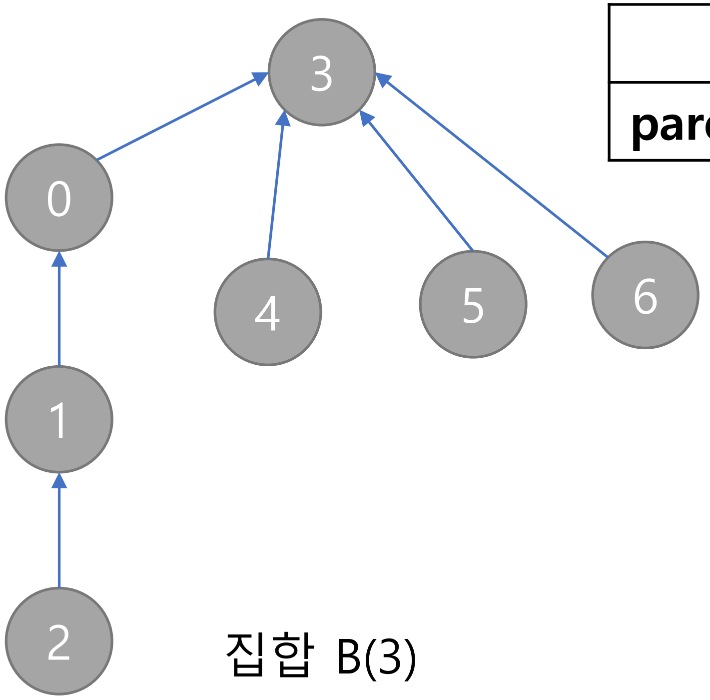
트리로 나타낸 다음, 바로 위의 노드를 가리킨다.

트리의 최상위 노드라면 자기 자신을 가리킨다.

유니온-파인드

- Union-Find(유니온-파인드)

Union : 두 원소 a, b가 주어질 때, a의 최상위 노드가 b의 최상위 노드를 가르키게 한다.



집합 B(3)

	0	1	2	3	4	5	6
parent	3	0	1	3	3	3	3

parent[a] 값도 수정해 준다

유니온-파인드

- Union-Find(유니온-파인드) 실제 구현

```
int find(int u)
    if(u == parent[u])
        return u
    return find(parent[u])
```

u == parent[u] -> 루트노드를 뜻함

루트노드를 만날때 까지 재귀 호출 하다가 만나면 루트노드를 반환

```
void merge(int u, int v)
    u = find(u)
    v = find(v)
    if(u == v)
        return
    parent[u] = v
```

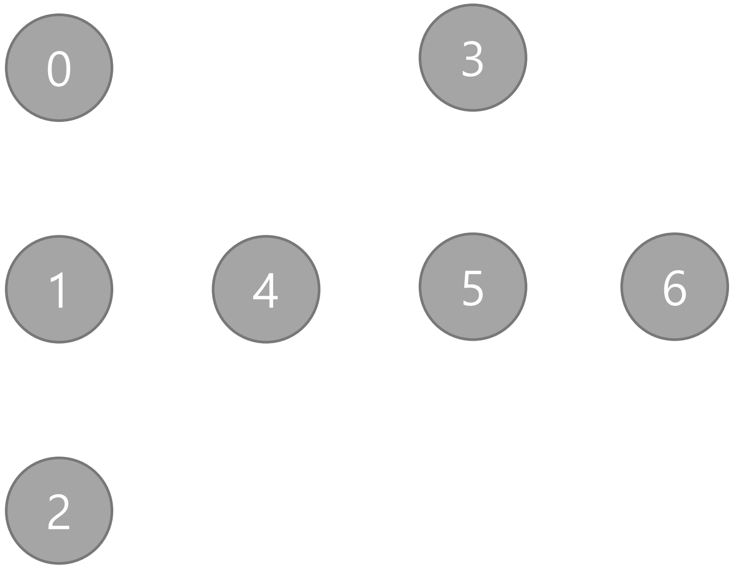
합치고자 하는 u랑 v에 각자 속하는 집합의 **루트 노드**를 대입

만약 그 둘이 같다면 u랑 v는 같은 집합에 속해있다

다르다면 u를 v에 병합시킨다

유니온-파인드

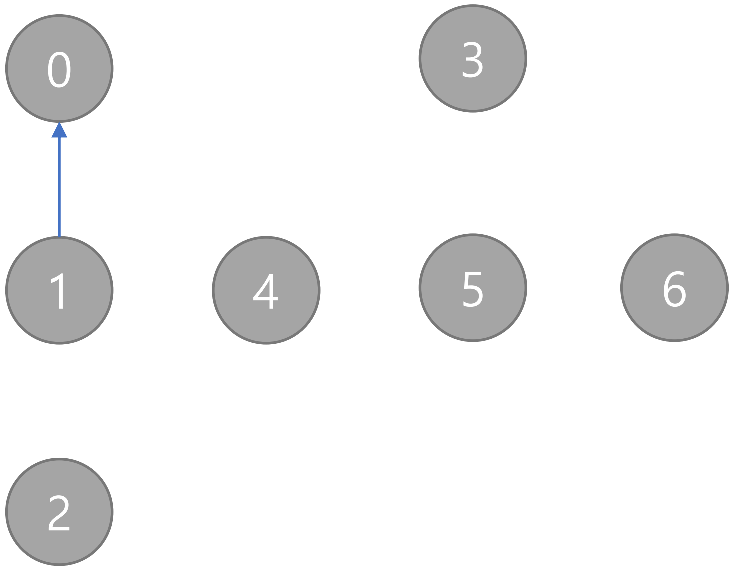
- Union-Find(유니온-파인드) 동작 과정



	0	1	2	3	4	5	6
parent	**0**	**1**	**2**	**3**	**4**	**5**	**6**

- Union-Find(유니온-파인드) 동작 과정

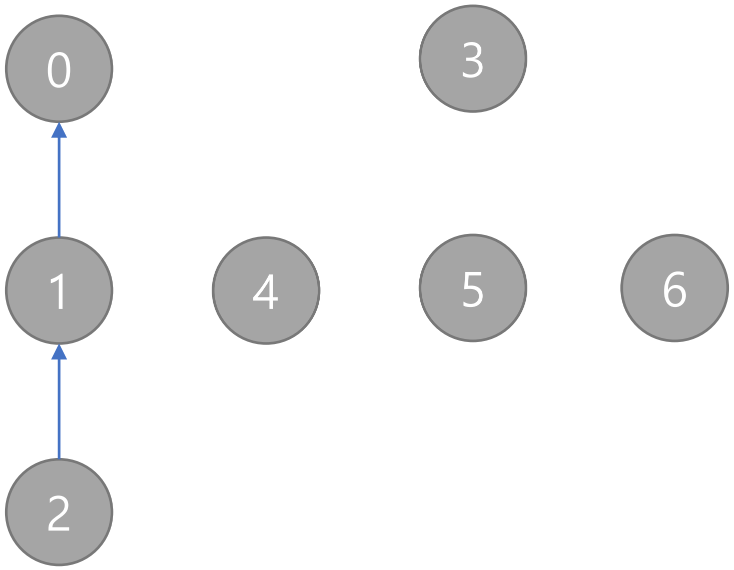
merge(0, 1)



	0	1	2	3	4	5	6
parent	0	0	2	3	4	5	6

- Union-Find(유니온-파인드) 동작 과정

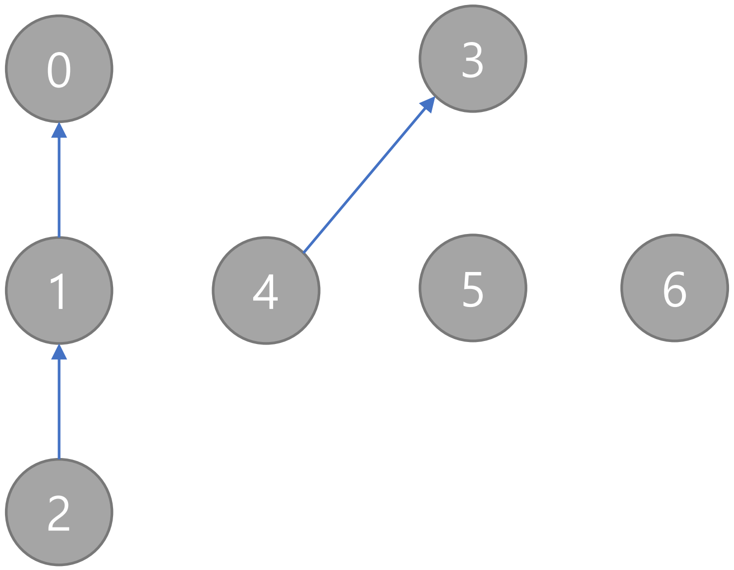
merge(1, 2)



	0	1	2	3	4	5	6
parent	**0**	**0**	**1**	**3**	**4**	**5**	**6**

- Union-Find(유니온-파인드) 동작 과정

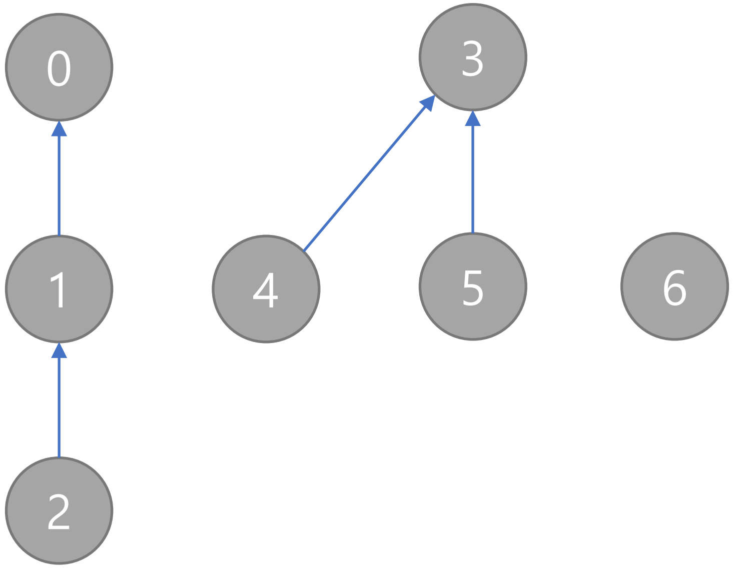
merge(3, 4)



	0	1	2	3	4	5	6
parent	**0**	**0**	**1**	**3**	**3**	**5**	**6**

- Union-Find(유니온-파인드) 동작 과정

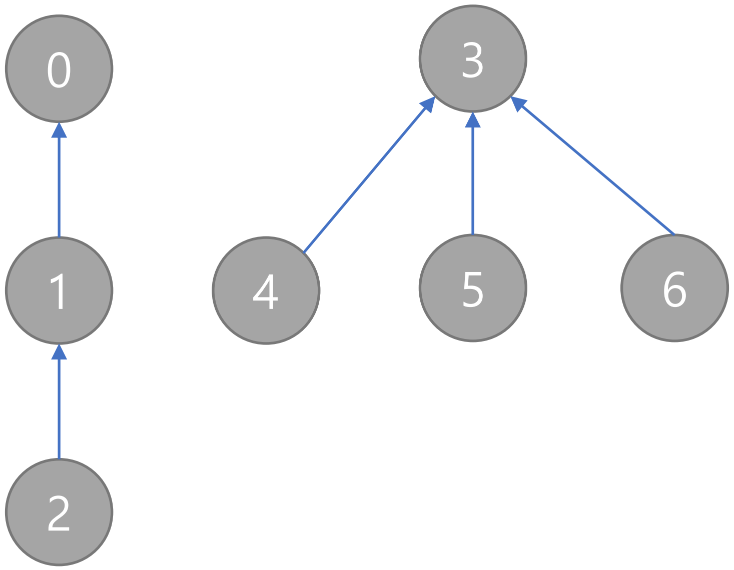
merge(3, 5)



	0	1	2	3	4	5	6
parent	**0**	**0**	**1**	**3**	**3**	**3**	**6**

- Union-Find(유니온-파인드) 동작 과정

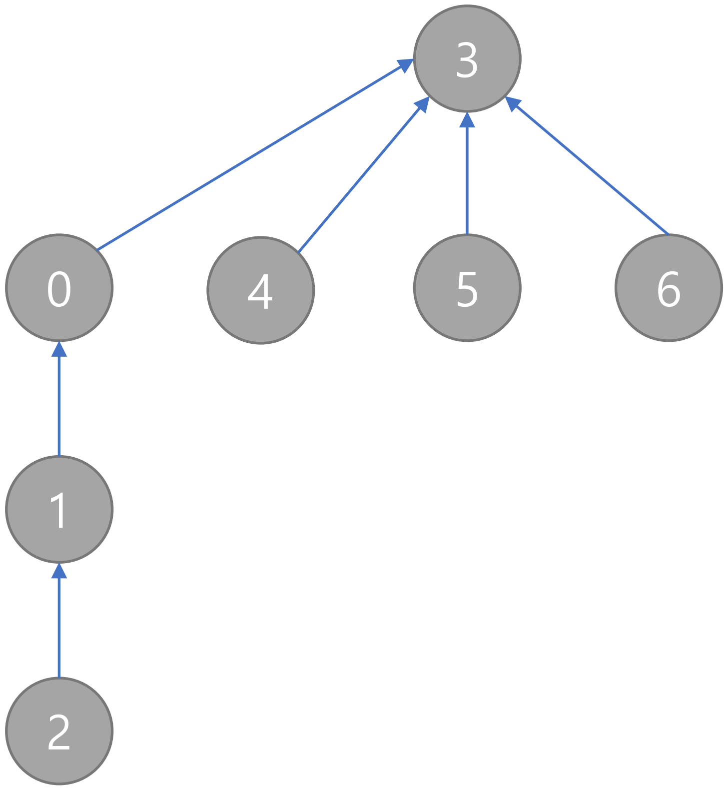
merge(3, 6)



	0	1	2	3	4	5	6
parent	**0**	**0**	**1**	**3**	**3**	**3**	**3**

- Union-Find(유니온-파인드) 동작 과정

merge(2, 4)



	0	1	2	3	4	5	6
parent	**3**	**0**	**1**	**3**	**3**	**3**	**3**

유니온-파인드

- Union-Find(유니온-파인드) 최적화

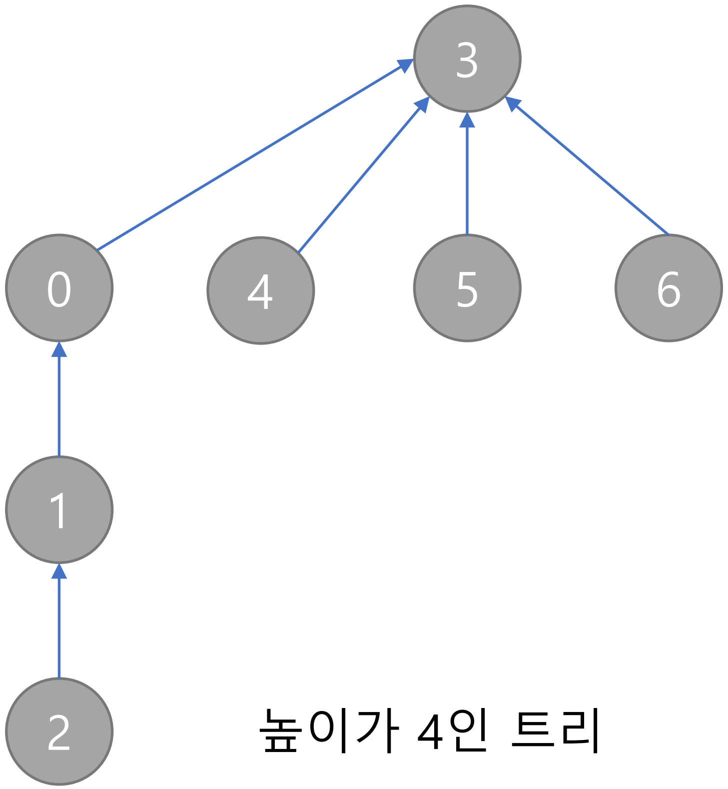
    1. Union-by-rank

    2. Path-compression

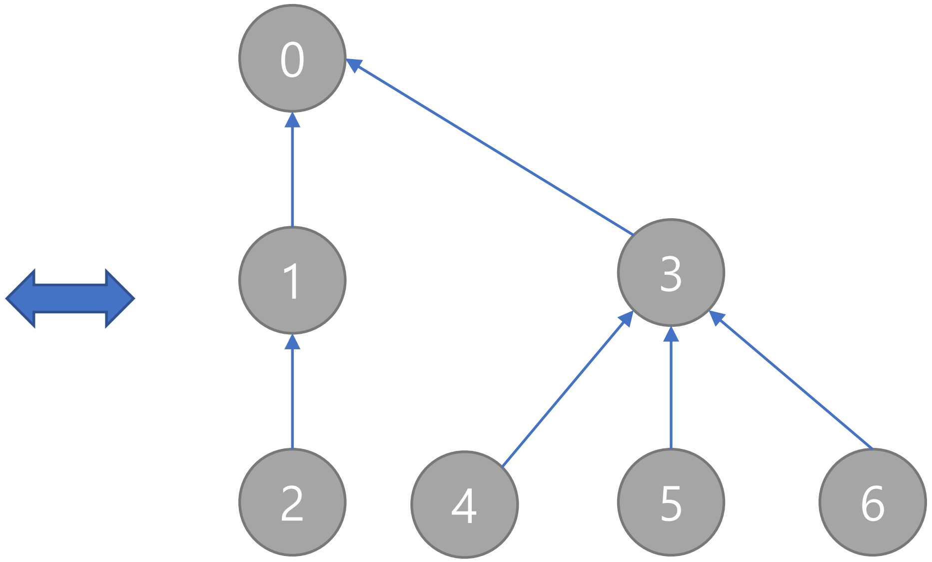
- Union-Find(유니온-파인드) 최적화

1. Union-by-rank

집합을 나타내는 트리끼리 merge 했을 때, 트리가 한쪽으로 기울어질 수 있다.



높이가 4인 트리



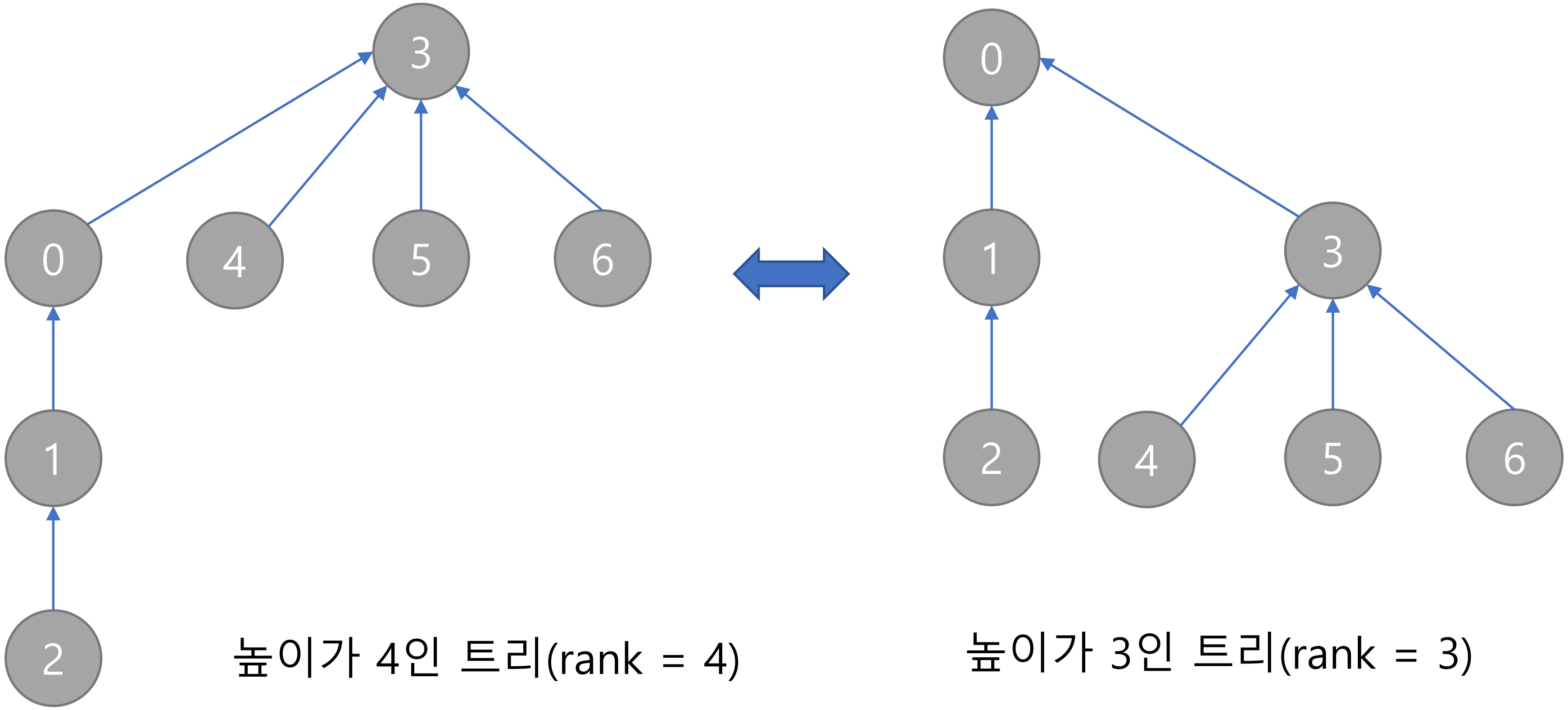
높이가 3인 트리

유니온-파인드

- Union-Find(유니온-파인드) 최적화

1. Union-by-rank

    find는 트리의 높이만큼 재귀할 수 있으므로 높이가 낮을 수록 좋다.



높이가 4인 트리(rank = 4)

높이가 3인 트리(rank = 3)

유니온-파인드

- Union-Find(유니온-파인드) 최적화

  1. Union-by-rank

rank[x] : x가 속한 집합의 트리의 높이를 나타냄, 초기값은 1

merge를 수행할 때 rank가 낮은 쪽이 높은 쪽으로 merge 되게끔 해준다

```
void merge(int u, int v)
    u = find(u)
    v = find(v)
    if(u == v)
        return
    parent[u] = v
```

```
void merge(int u, int v)
    u = find(u)
    v = find(v)
    if(u == v)
        return
    if(rank[u] > rank[v])
        swap(u, v)
    parent[u] = v
    if(rank[u] == rank[v])
        rank[v] ++
```

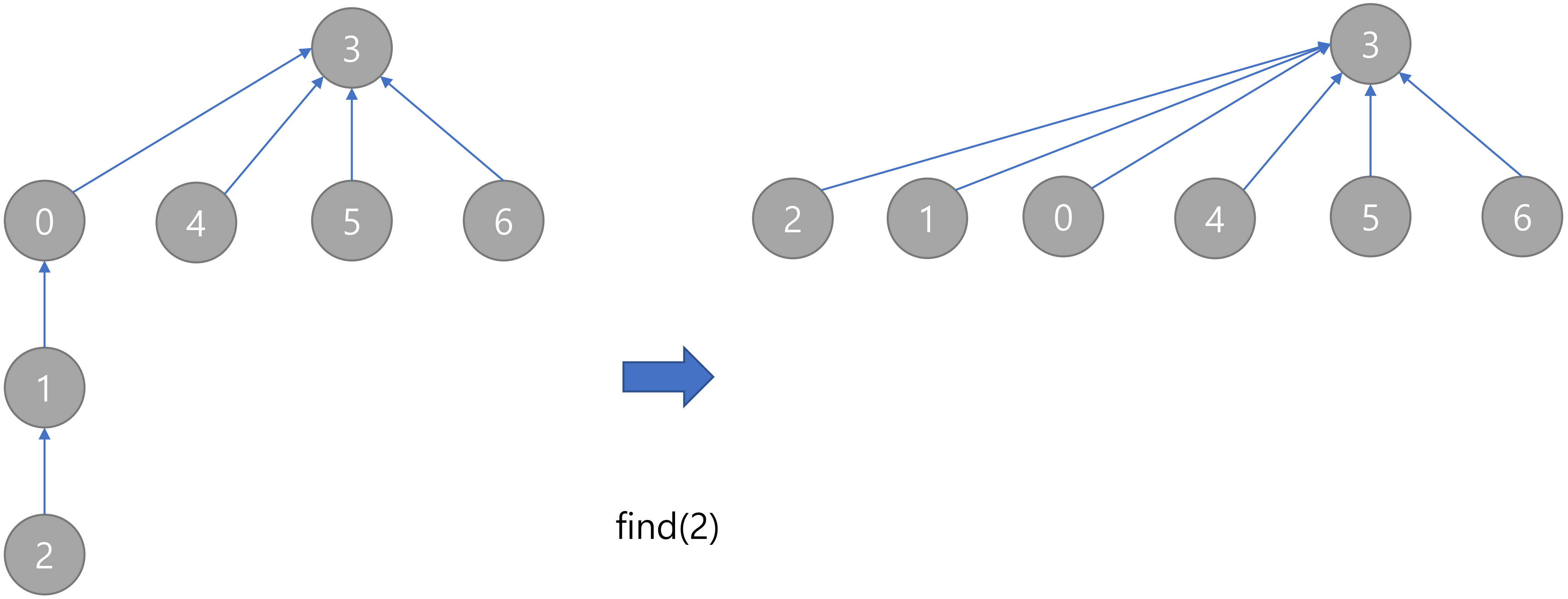
항상 rank[v] >= rank[u]가 된다

만약 둘의 rank가 같다면 rank[v]++

- Union-Find(유니온-파인드) 최적화

  2. Path-compression

     find 연산의 결과를 parent[u]에 바로 대입한다



- Union-Find(유니온-파인드) 최적화

2. Path-compression

find 연산의 결과를 parent[u]에 바로 대입한다

```
int find(int u)
    if(u == parent[u])
        return u
    return find(parent[u])
```

→

```
int find(int u)
    if(u == parent[u])
        return u
    return parent[u] = find(parent[u])
```

- Union-Find(유니온-파인드) 최종 구현

```
int parent[N] = {0,1,2,...,N-1}
int rank[N]   = {1,1,1,...,1}
```

```
int find(int u)
    if(u == parent[u])
        return u
    return parent[u] = find(parent[u])
```

```
void merge(int u, int v)
    u = find(u)
    v = find(v)
    if(u == v)
        return
    if(rank[u] > rank[v])
        swap(u, v)
    parent[u] = v
    if(rank[u] == rank[v])
        rank[v] ++
```

플로이드-워셜을 이용하는 문제 소개

1. 경로 찾기 : https://www.acmicpc.net/problem/11403

2. 백양로 브레이크 : https://www.acmicpc.net/problem/11562

플로이드-워셜을 이용하는 문제 – 경로 찾기

문제 요약 : https://www.acmicpc.net/problem/11403

- $1 \le N \le 100$
- 가중치 없는 방향 그래프 G가 주어진다.
- adj[i][j]가 0이면 i->j로 가는 간선이 없다는 뜻, 1이면 있다. adj[i][i]는 항상 0이다.
- 모든 정점 (i, j)에 대해서 i->j로 가는 경로가 있는지 없는지 구하여라

고려해야 할 사항 :

- 각 정점 간의 도달 가능성 여부를 판단해야 할 때 ?

플로이드-워셜을 이용하는 문제 – 경로 찾기

- 각 정점 간의 도달 가능성 여부를 판단해야 할 때 ?

플로이드-워셜 알고리즘으로 빠르게 판단

O(N^3)으로도 충분히 통과

문제에서 주어진 adj[][]를 최단거리라 생각

플로이드-워셜을 이용하는 문제 – 경로 찾기

- 실제 풀이

```
void Floyd()
for(int k=0; k<N; k++)
    for(int i=0; i<N; i++)
        for(int j=0; j<N; j++)
            dist[i][j] = min(dist[i][j],
            dist[i][k] + dist[k][j])
```

```
for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        cin>>dist[i][j]
        if(dist[i][j] == 0)
            dist[i][j] = INF
Floyd()
for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
        if(dist[i][j] == INF)
        cout<<0
        else
        cout<<1
```

플로이드-워셜을 이용하는 문제 – 백양로 브레이크

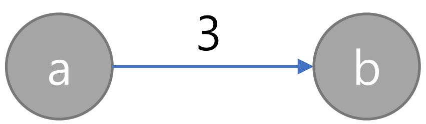
문제 요약 : https://www.acmicpc.net/problem/11562

- $n \le 250$, $m \le n * (n - 1) / 2$, $1 \le k \le 30{,}000$
- 방향 그래프로 나타낼 수 있다.
- s -> e로 갈 때, 최소 몇 개의 길을 양방통행으로 바꾸어야 하는가?
- 서로 도달 불가능한 건물은 없다.

고려해야 할 사항 :

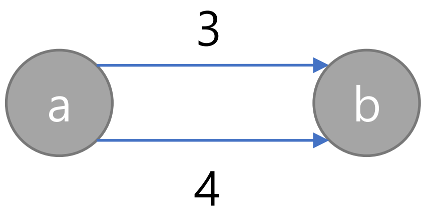
- 건물 사이를 잇는 길은 최대 한개이다.
- 미로 만들기와 비슷한 문제

- 건물 사이를 잇는 길은 최대 한개이다.



(a, b, c) : a와 b는 c의 비용으로 연결되어있다.

dist[a][b] = c

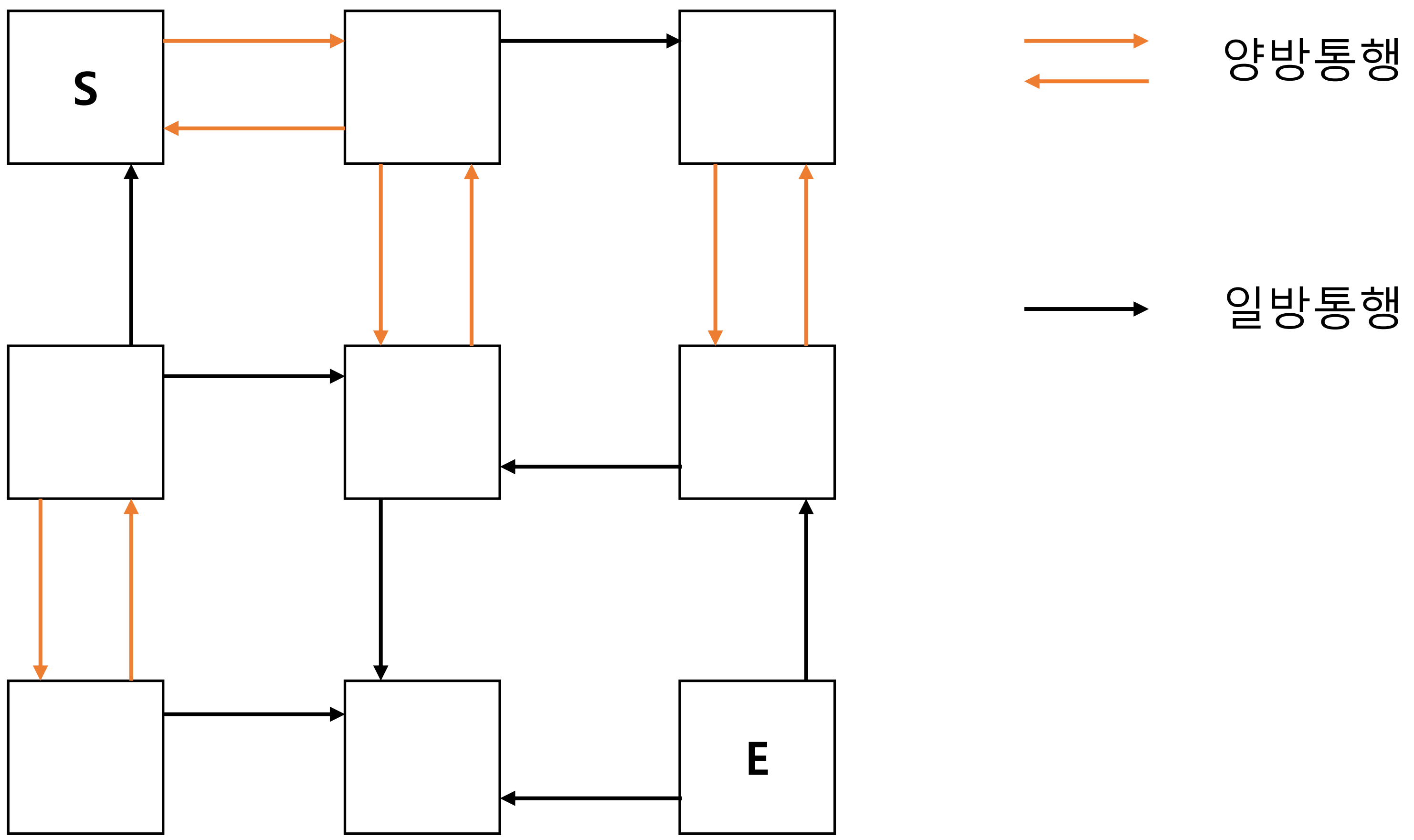


(a, b, c) : a와 b는 c의 비용으로 연결되어있다.

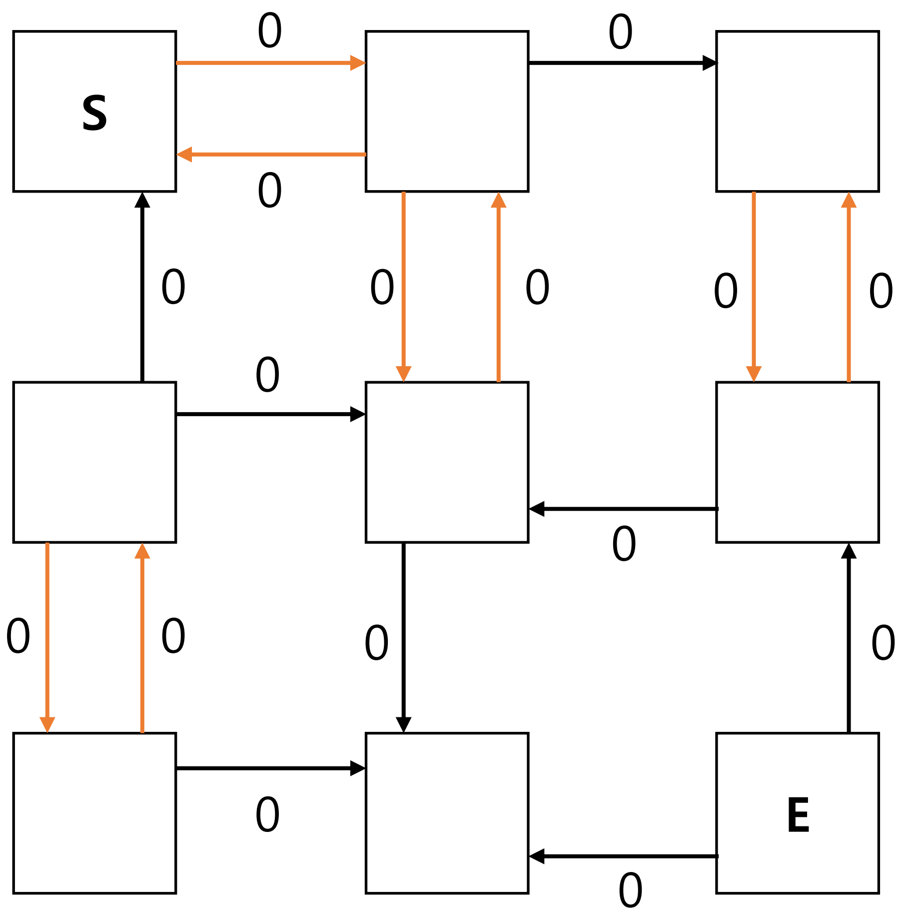
dist[a][b] = min(dist[a][b], c)

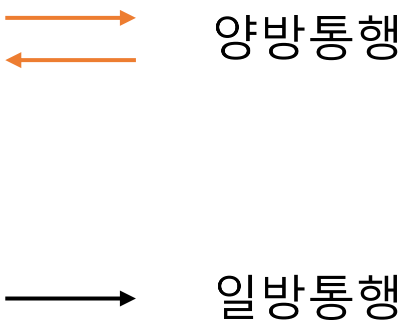
플로이드-워셜을 이용하는 문제 – 백양로 브레이크

- 미로 만들기와 비슷한 문제
  - 간선의 비용 설정?



- 미로 만들기와 비슷한 문제
  - 간선의 비용 설정?



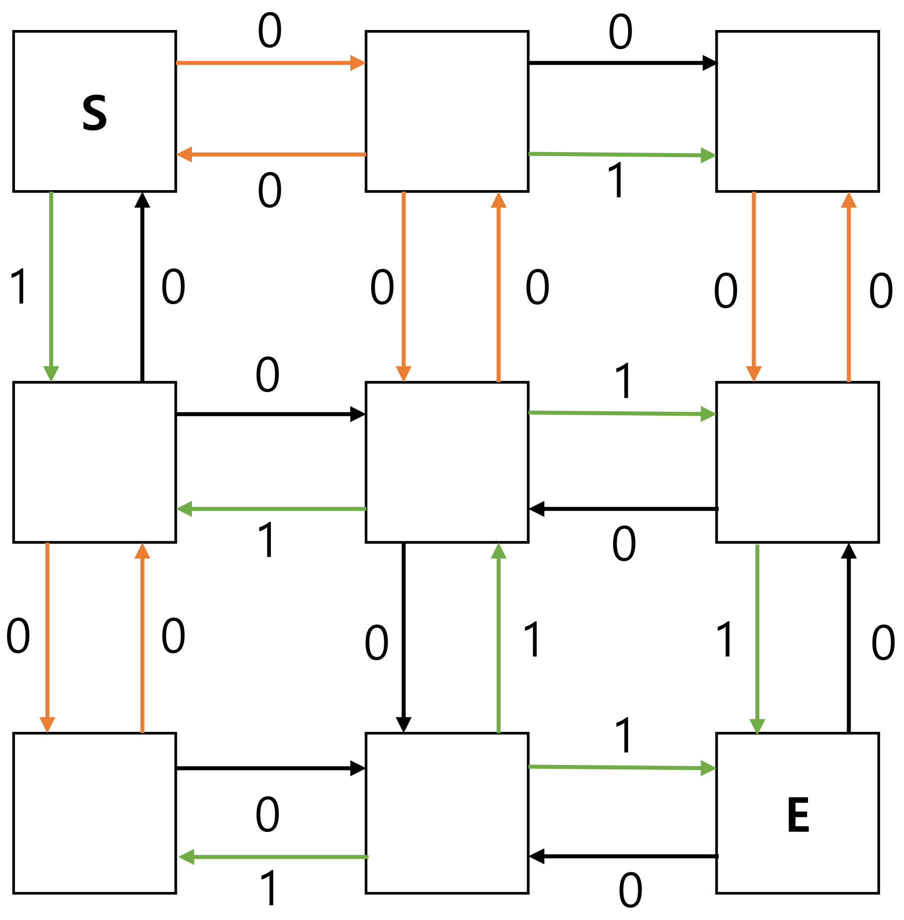


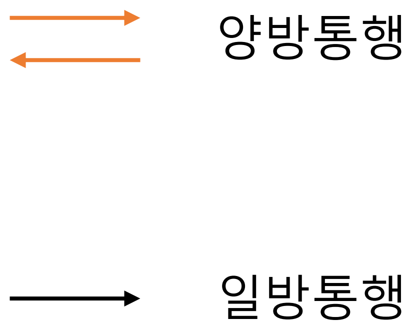
맨 처음 상태들의 간선의 비용을 0으로 설정

=> 일방통행을 양방통행으로 바꾸지 않은 경우

플로이드-워셜을 이용하는 문제 – 백양로 브레이크

- 미로 만들기와 비슷한 문제
  - 간선의 비용 설정?



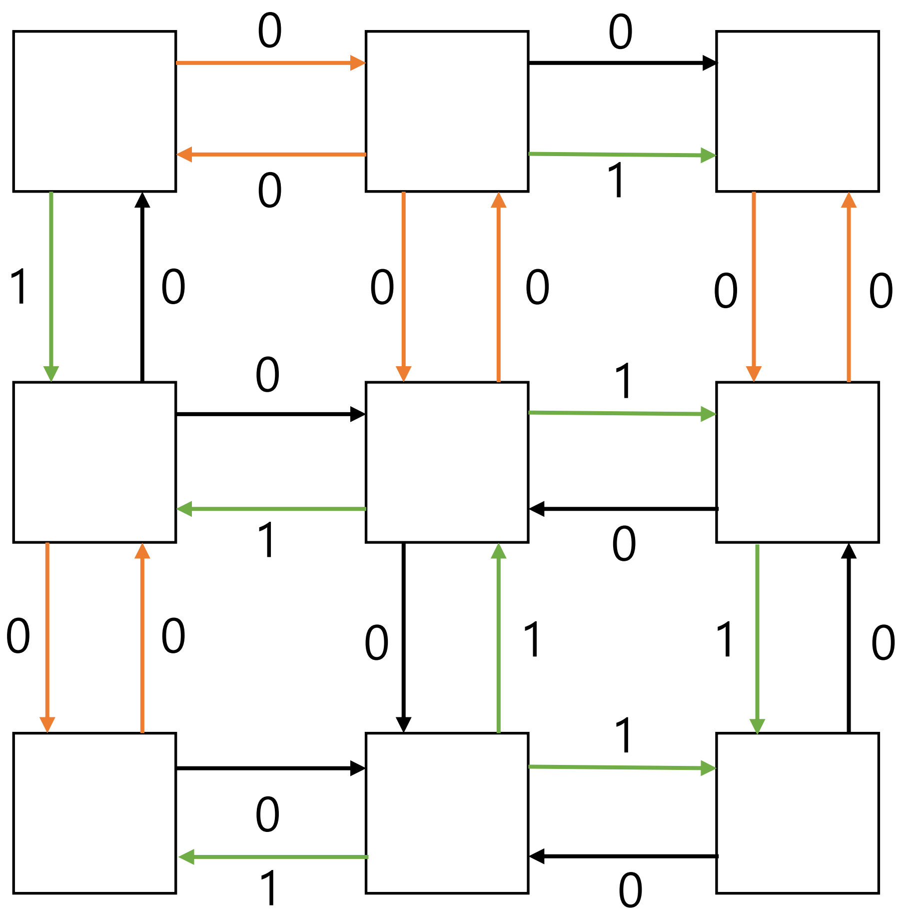


일방통행인 길을 양방통행으로 전부 바꿔준 다음

새로 만든 길의 비용을 1로 설정

=> 비용을 지불하고 양방통행으로 바꾼다

- 미로 만들기와 비슷한 문제



양방통행

일방통행

최단거리 알고리즘으로 경로상의 비용이 최소가 되게끔

플로이드-워셜을 이용하는 문제 – 백양로 브레이크

- 실제 구현

```cpp
ios_base::sync_with_stdio(false);
cin.tie(NULL);
for (int i = 1; i < 251; i++)
{
    for (int j = 1; j < 251; j++)
    {
        dist[i][j] = INF;
        if (i == j)
            dist[i][j] = 0;
    }
}
```

```cpp
cin >> N >> M;
int a, b, c;
for (int i = 0; i < M; i++)
{
    cin >> a >> b >> c;
    if (c == 0)
    {
        dist[a][b] = 0;
        dist[b][a] = 1;
    }
    else if (c == 1)
    {
        dist[a][b] = 0;
        dist[b][a] = 0;
    }
}
```

```cpp
for (int k = 1; k <= N; k++)
{
    for (int i = 1; i <= N; i++)
    {
        for (int j = 1; j <= N; j++)
        {
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
        }
    }
}
```

유니온-파인드를 이용하는 문제 소개

1. 집합의 표현 : https://www.acmicpc.net/problem/1717

2. 여행 가자 : https://www.acmicpc.net/problem/1976

유니온-파인드를 이용하는 문제 – 집합의 표현

문제 요약 :

https://www.acmicpc.net/problem/1717

- $1 \le n \le 1{,}000{,}000$, $1 \le m \le 100{,}000$
- m개의 연산이 주어지고 각 연산은 합치기, 두 원소가 같은 집합에 포함되어있나 확인하기

고려해야 할 사항 :

- 유니온-파인드 알고리즘을 사용하기에 적합
- 같은 집합에 포함되어 있다?

- 같은 집합에 포함되어 있다?

    find(a), find(b)의 값이 서로 같다면 같은 집합에 포함되어있다

```cpp
int N, M;
int parent[1000001];
int height[1000001];

int find(int u)
{
    if (parent[u] == u)
        return u;
    return parent[u] = find(parent[u]);
}

void merge(int u, int v)
{
    u = find(u);
    v = find(v);
    if (u == v)
        return;
    if (height[u] > height[v])
        parent[v] = u;
    else if (height[u] < height[v])
        parent[u] = v;
    else
    {
        parent[u] = v;
        height[v] ++;
    }
}
```

```cpp
int main()
{
    cin.tie(NULL);
    ios_base::sync_with_stdio(false);
    cin >> N >> M;
    for (int i = 1; i <= N; i++)
    {
        parent[i] = i;
        height[i] = 1;
    }
    int a, b, c;
    for (int i = 0; i < M; i++)
    {
        cin >> a >> b >> c;
        if (a == 0)
            merge(b, c);
        else
        {
            if (find(b) == find(c))
                cout << "YES" << "\n";
            else
                cout << "NO" << "\n";
        }
    }
    return 0;
}
```

유니온-파인드를 이용하는 문제 – 여행 가자

문제 요약 :

https://www.acmicpc.net/problem/1976

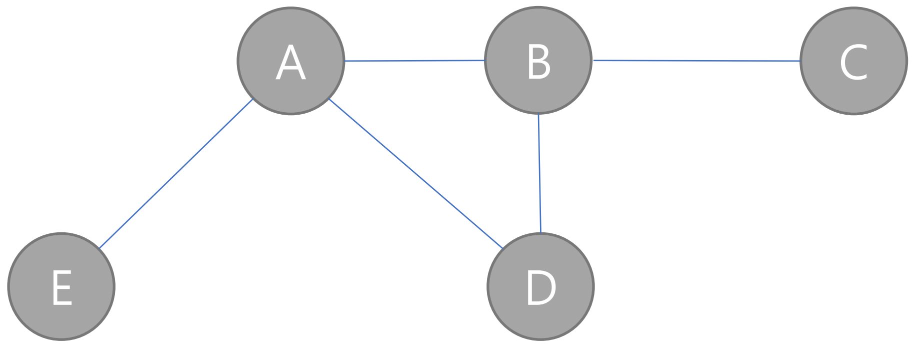
- $1 \le N \le 200$, $1 \le M \le 1{,}000$
- 주어진 여행경로가 가능한가?
- 같은 도시를 여러 번 방문 가능하다
- 다른 도시를 경유해서 갈 수 있다

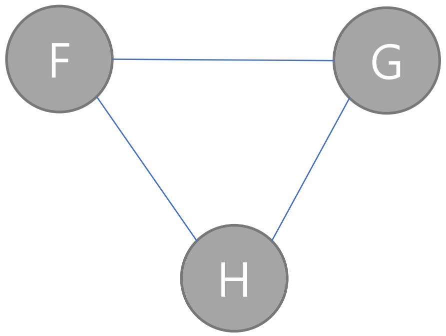
고려해야 할 사항 :

- 경로상의 다음 도시와 연결만 되어 있다면 갈 수 있다
- 연결되어 있는지 어떻게 판단?

유니온-파인드를 이용하는 문제 – 여행 가자

- 경로상의 다음 도시와 연결만 되어 있다면 갈 수 있다





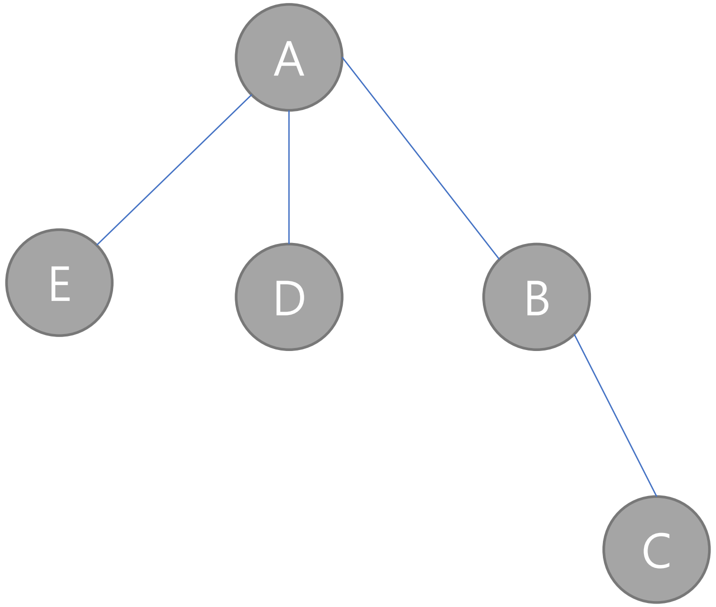
여행 계획 : E C B C D

다 연결되어 있으므로 가능!

같은 집합에 속한다

유니온-파인드를 이용하는 문제 – 여행 가자

- 연결되어 있는지 어떻게 판단?
  - 유니온-파인드 사용



	A	B	C	D	E
parent	**A**	**A**	**B**	**A**	**A**
find	**A**	**A**	**A**	**A**	**A**

유니온-파인드를 이용하는 문제 – 여행 가자

- 실제 구현
  - 유니온-파인드 사용

```cpp
int N, M;
int linked[201][201];
int parent[201];
int height[201];

int find(int u)
{
    if(u == parent[u])
        return u;
    return parent[u] = find(parent[u]);
}

void merge(int u, int v)
{
    u = find(u);
    v = find(v);
    if(u == v)
        return;
    if(height[u] > height[v])
        swap(u, v);
    parent[u] = v;
    if (height[u] == height[v])
        height[v] ++;
}
```

```cpp
int main()
{
    for (int i = 0; i < 201; i++)
    {
        parent[i] = i;
        height[i] = 1;
    }
    cin >> N >> M;
    for (int i = 1; i <= N; i++)
    {
        for (int j = 1; j <= N; j++)
        {
            cin >> linked[i][j];
            if (linked[i][j])
            {
                merge(i, j);
            }
        }
    }
```

```cpp
int x;
int first = -1;
bool canTrip = true;
for (int i = 0; i < M; i++)
{
    cin >> x;
    if(i == 0)
        first = find(x);
    else
    {
        if (first != find(x))
        {
            canTrip = false;
            break;
        }
    }
}
if (canTrip)
    cout << "YES\n";
else
    cout << "NO\n";
```

자유로운 질문 및 토의!