



# 5주차 알고리즘 세미나

PoolC 2021 1학기

### 문제 요약 :

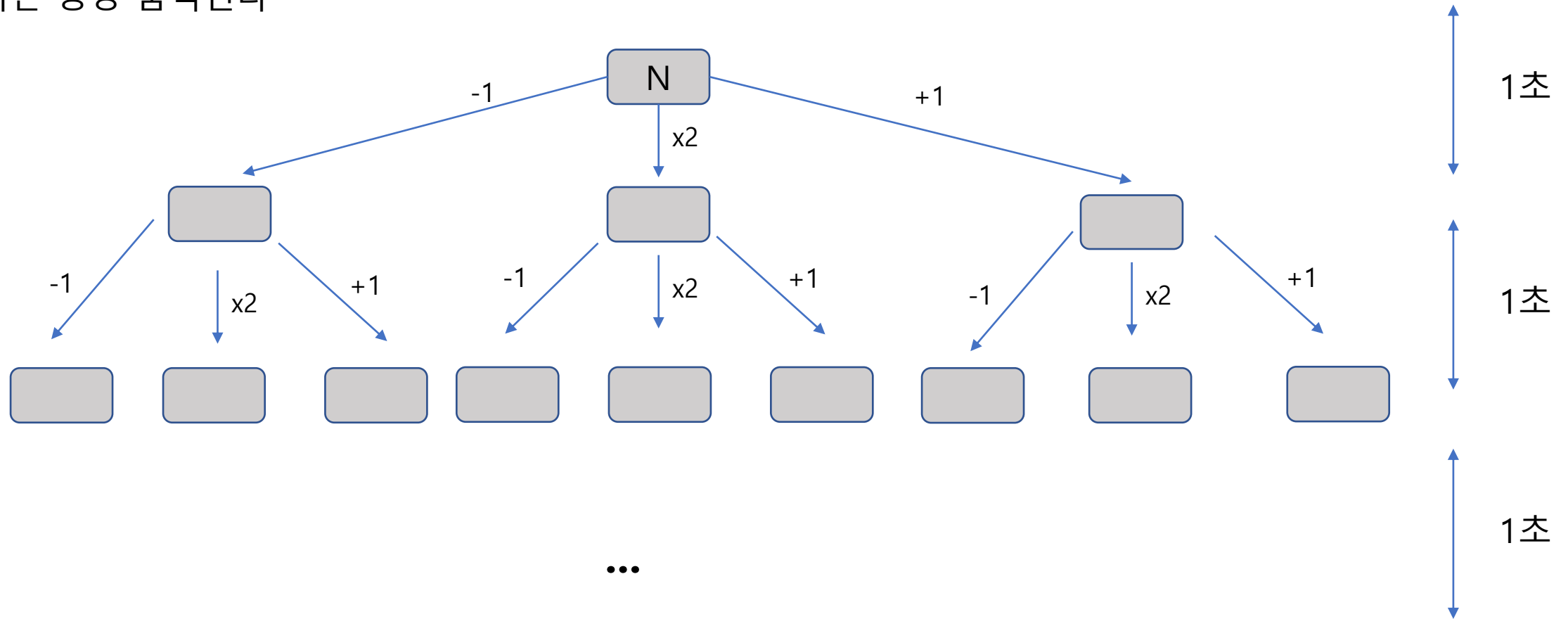
- $0 \leq N \leq 100,000$ ,  $0 \leq K \leq 100,000$
- 동생은 점 K에 고정되어 있고, 수빈이는 점 X에서 걸거나 순간이동을 한다.
- 수빈이가 동생을 찾을 수 있는 가장 빠른 시간이 얼마인가?

### 고려해야 할 사항 :

- 수빈이는 항상 움직인다.
- 어떤 지점에 도착하는 최단 시간은 정해져 있다.

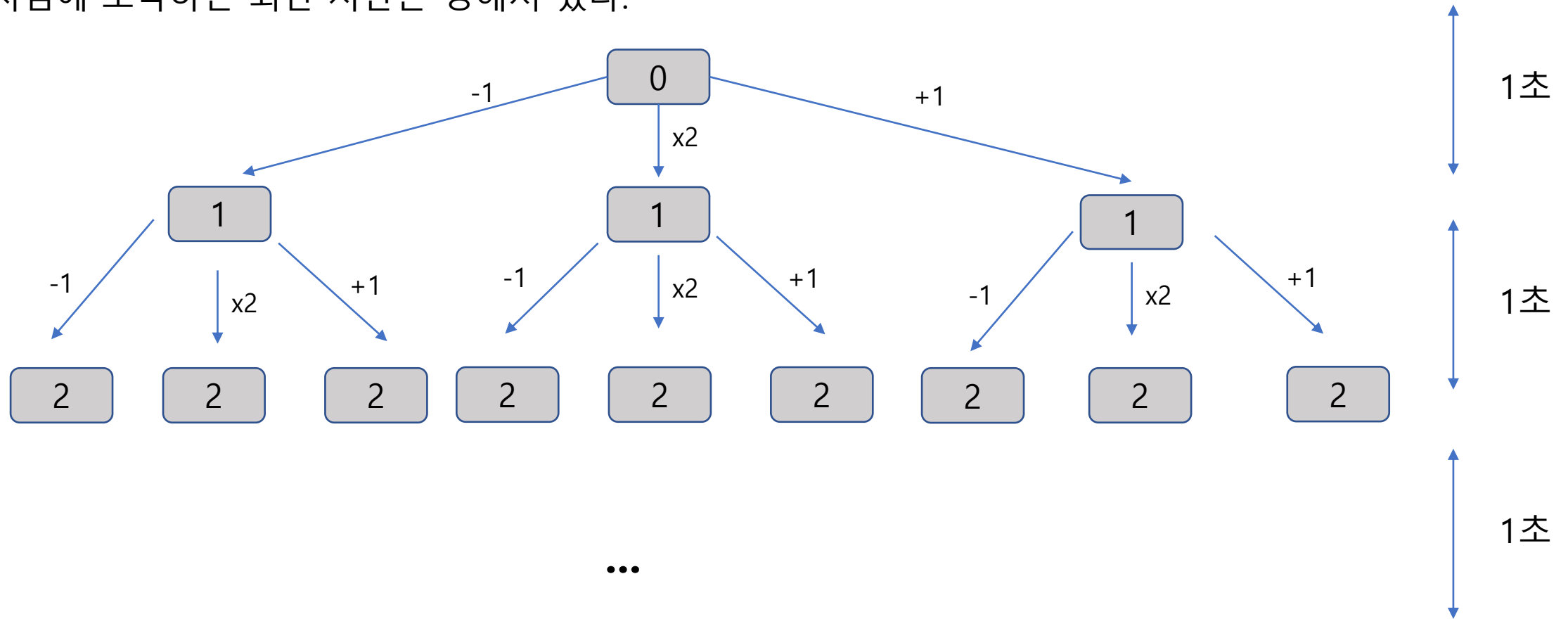
## 지난시간 도전문제 - 숨바꼭질

- 수빈이는 항상 움직인다



## 지난시간 도전문제 - 숨바꼭질

- 어떤 지점에 도착하는 최단 시간은 정해져 있다.



## ● ● 지난시간 도전문제 - 숨바꼭질

- BFS 진행
  - x라는 지점이 있을 때
  - 수빈이가 x에 도착하는 최단 시간 = N에서 x까지 dist의 값 = dist[x]
- 동생은 K에 고정되어 있으므로, dist[K]만 구해주면 된다.

### - 실제 구현

```
void BFS(int start)
    q.Enqueue(start)
    dist[start] = 0
    while(!q.empty())
        int here = q.Dequeue()
        if(inRange(here+1) && dist[here+1] == -1)
            dist[here+1] = dist[here] + 1
            q.Enqueue(here+1)
        if(inRange(here-1) && dist[here-1] == -1)
            dist[here-1] = dist[here] + 1
            q.Enqueue(here-1)
        if(inRange(here*2) && dist[here*2] == -1)
            dist[here*2] = dist[here] + 1
            q.Enqueue(here*2)
```

BFS(N)

cout << dist[K]

### 문제 요약 :

- $3 \leq N, M \leq 8$ , 벽을 3개 세워야 한다.
- 0은 빈 칸, 1은 벽, 2는 바이러스
- 빈 칸 중에서 3개를 선택해 벽을 세운 다음, 안전영역의 최대 크기를 구하여라
- 바이러스는 상하좌우로 퍼져나갈 수 있으며, 벽을 통과하지 못한다

### 고려해야 할 사항 :

- 벽을 세우는 장소 선택
- 바이러스가 퍼지는 구현

### - 문제에 사용되는 변수

```
int N,M, board[8][8], dx[4], dy[4], MAX
```

```
bool visited[8][8]
```

```
vector<pair<int, int> > pos
```

```
vector<pair<int, int> > virus
```

```
vector<int> list
```

```
queue<pair<int, int> > q
```

숫자가 0인 칸의 좌표 정보 저장

숫자가 2인 칸의 좌표 정보 저장

벽을 세우기로 한 pos의 index 저장

BFS를 위한 좌표정보 저장



### - 문제 풀이 방향

1. 숫자가 0인 칸 중에서(pos) 벽을 세울 장소 3곳을 선택한다
2. 선택한 곳에 실제로 벽을 세운다(board값 수정)
3. 바뀐 board에 바이러스를 뿌린다(BFS)
4. 안전지역의 크기를 계산하고 값을 갱신한다
5. 벽을 세웠던 곳의 벽을 다시 허문다(board값 수정)

반복

- 벽을 세우는 장소 선택

2	0	0	0	1	1	0
0	0	1	0	1	2	0
0	1	1	0	1	0	0
0	1	0	0	0	0	0
0	0	0	0	0	1	1
0	1	0	0	0	0	0
0	1	0	0	0	0	0

흰색칸 중에서 벽을 세울 장소 3곳을 고르기

- 벽을 세우는 장소 선택

2	0	0	0	1	1	0
0	0	1	0	1	2	0
0	1	1	0	1	0	0
0	1	0	0	0	0	0
0	0	0	0	0	1	1
0	1	0	0	0	0	0
0	1	0	0	0	0	0

```
vector<pair<int, int> > pos
```

2차원 배열을 돌면서 0을 만나면  
`pos.push_back( make_pair(y, x) )`

pos들 중에서 3개를 고르는 함수 구현

### - 벽을 세우는 장소 선택

```
void selectWall(int index)
```

```
    if(list.size() == 3)
```

```
        createWall(true)
```

```
        bfs()
```

```
        createWall(false)
```

```
        return
```

```
    if(index >= pos.size())
```

```
        return
```

```
    list.push_back(index)
```

```
    selectWall(index+1)
```

```
    list.pop_back()
```

```
    selectWall(index+1)
```

### - 벽을 세우기(허물기)

```
void createWall(bool flag)
```

```
    for(int i=0; i<list.size(); i++)
```

```
        board[pos[list[i]].first][pos[list[i]].second] = flag
```

list[i] : 우리가 선택한 index

pos[list[i]] : index번째 0의 좌표(pair)

pos[list[i]].first : index번째 0의 y좌표

pos[list[i]].second : index번째 0의 x좌표

### - 바이러스가 퍼지는 구현

```
void bfs()
{
    int cnt = 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            visited[i][j] = false;
        }
    }
    for (int i = 0; i < virus.size(); i++)
    {
        q.push(virus[i]);
        visited[virus[i].first][virus[i].second] = true;
    }
    while (!q.empty())
    {
        int y = q.front().first;
        int x = q.front().second;
        q.pop();
        for (int i = 0; i < 4; i++)
        {
            if (inRange(y + dy[i], x + dx[i]) && board[y + dy[i]][x + dx[i]] != 1 && !visited[y + dy[i]][x + dx[i]])
            {
                visited[y + dy[i]][x + dx[i]] = true;
                q.push(make_pair(y + dy[i], x + dx[i]));
                cnt++;
            }
        }
    }
    MAX = max(MAX, (int)pos.size() - 3 - cnt);
}
```

### - 바이러스가 퍼지는 구현

```
void bfs()
{
    int cnt = 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            visited[i][j] = false;
        }
    }
}
```

cnt : 바이러스가 퍼진 영역의 개수

왜 visited를 매 bfs 마다 초기화 해주어야 하는가?

- 매 bfs마다 visited값은 바뀌는데, 초기화를 하지 않게 되면 우리의 의도와는 다르게 bfs가 진행될 수 있다.

- 현재 bfs가 이전 bfs의 영향을 받지 않도록 visited를 초기화

```
for (int i = 0; i < virus.size(); i++)
{
    q.push(virus[i]);
    visited[virus[i].first][virus[i].second] = true;
}
```

bfs의 시작점이 여러개이므로 각각의 좌표를 모두 queue에 넣어준다  
visited도 true로 만들어준다

### - 바이러스가 퍼지는 구현

```
while (!q.empty())
{
    int y = q.front().first;
    int x = q.front().second;
    q.pop();
    for (int i = 0; i < 4; i++)
    {
        if (inRange(y + dy[i], x + dx[i]) && board[y + dy[i]][x + dx[i]] != 1 && !visited[y + dy[i]][x + dx[i]])
        {
            visited[y + dy[i]][x + dx[i]] = true;
            q.push(make_pair(y + dy[i], x + dx[i]));
            cnt++;
        }
    }
}
MAX = max(MAX, (int)pos.size() - 3 - cnt);
}
```

바이러스가 퍼질 때  
cnt++로 퍼지는 영역의  
크기를 세준다.

pos.size() : 맨 처음 0인칸의 개수

3 : 우리가 벽을 세운 개수

cnt : 바이러스가 퍼진 영역의 크기

- 문제 요약
  - DFS와 BFS가 적절히 섞여있는 문제
  - vector, pair, queue 등 자료구조의 활용
  - 비슷한 문제인 연구소2, 연구소3도 풀어보면 좋습니다



### 문제 요약 :

- $1 \leq L \leq 30, 1 \leq R \leq 30, 1 \leq C \leq 30$
- L은 층수, R과 C는 한 층의 행과 열의 개수
- #은 벽, .은 빈 칸, S는 시작점, E는 탈출구

### 고려해야 할 사항 :

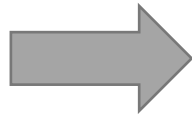
- 한 칸에서 4방향이 아닌 6방향으로 움직일 수 있다.
- 탈출할 수 없다?

## ● ● 지난시간 도전문제 - 상범빌딩

- 4방향이 아닌 6방향으로 움직일 수 있다

int dx[4] : {1,0,-1,0}

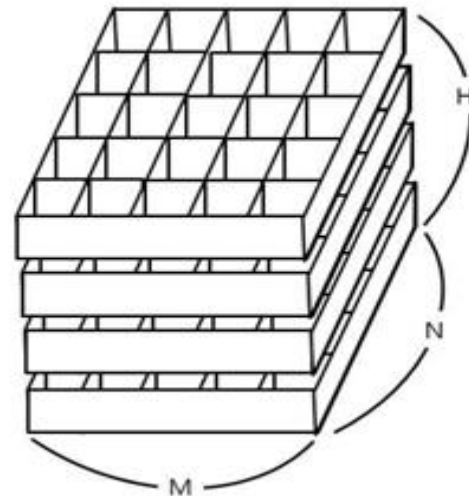
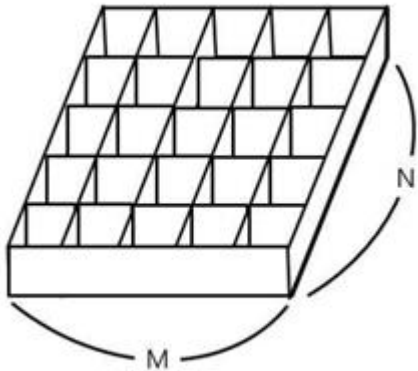
int dy[4] : {0,1,0,-1}



int dx[6] : {1,0,-1,0,0,0}

int dy[6] : {0,1,0,-1,0,0}

int dz[6] : {0,0,0,0,1,-1}



- 실제 구현

```
void BFS(pair<int, pair<int, int> > start)
    q.Enqueue(start)
    dist[start.z][start.y][start.x] = 0
    while(!q.empty())
        int here = q.Dequeue()
        for(int i=0; i<6; i++)
            if(inRange && dist == -1 && board != '#')
                dist[z+dz[i]][y+dy[i]][x+dx[i]] = dist[z][y][x] + 1
                q.Enqueue(z+dz[i], y+dy[i], x+dx[i])
```

- 탈출할 수 없다?

BFS(start)

```
if(dist[end] == -1)
```

```
cout<<Trapped!
```

```
else
```

```
cout<<정답
```

### 문제 요약 :

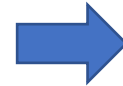
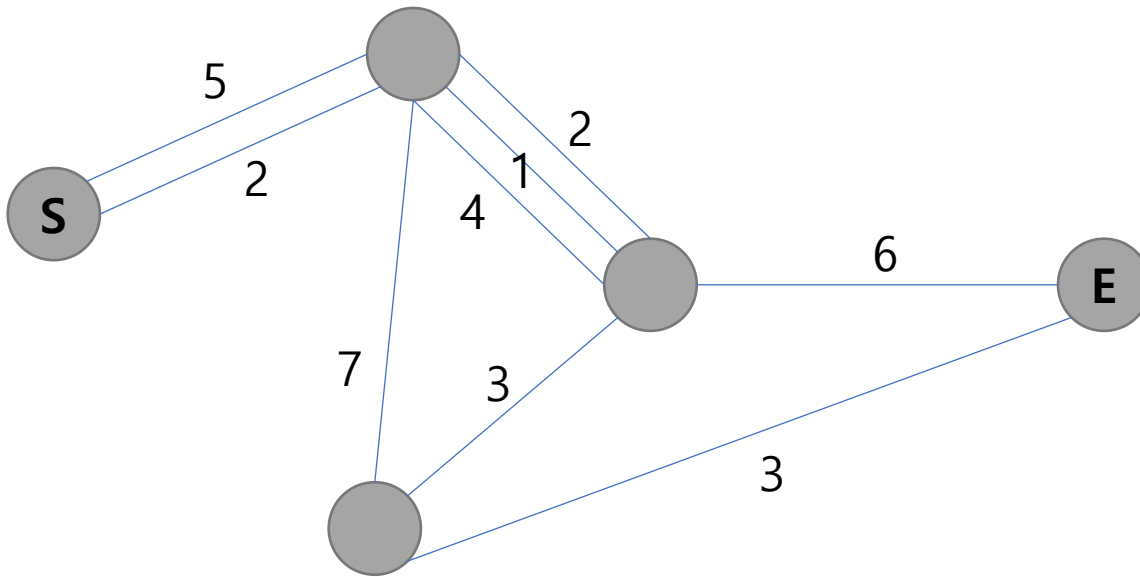
- $2 \leq N \leq 10,000$ ,  $1 \leq M \leq 100,000$ ,  $1 \leq C \leq 1,000,000,000$
- 시작섬에서 끝섬까지 물건을 옮겨야 한다.
- 섬에서 섬으로 갈때 다리를 건너는데, 각각의 다리마다 버틸 수 있는 중량이 다르다.
- 최적의 경로를 선택해서 최대로 옮길 수 있는 중량은?
- 모든 다리는 양방향 / 두 섬 사이에 여러 개의 다리가 존재 할 수 있음
- 시작섬에서 끝섬까지 항상 갈 수 있는 경로만 주어진다

### 고려해야 할 사항 :

- 화물무게의 범위
- 끝섬에  $w$ 의 무게로 도착 가능한지
- 다리의 정보 저장

## 지난시간 도전문제 - 중량제한

- 화물무게의 범위
  - 다리가 버틸 수 있는 무게  $C$ 는  $1 \leq C \leq 1,000,000,000$
  - 따라서 자연스럽게 화물의 무게도  $C$ 의 범위를 따른다

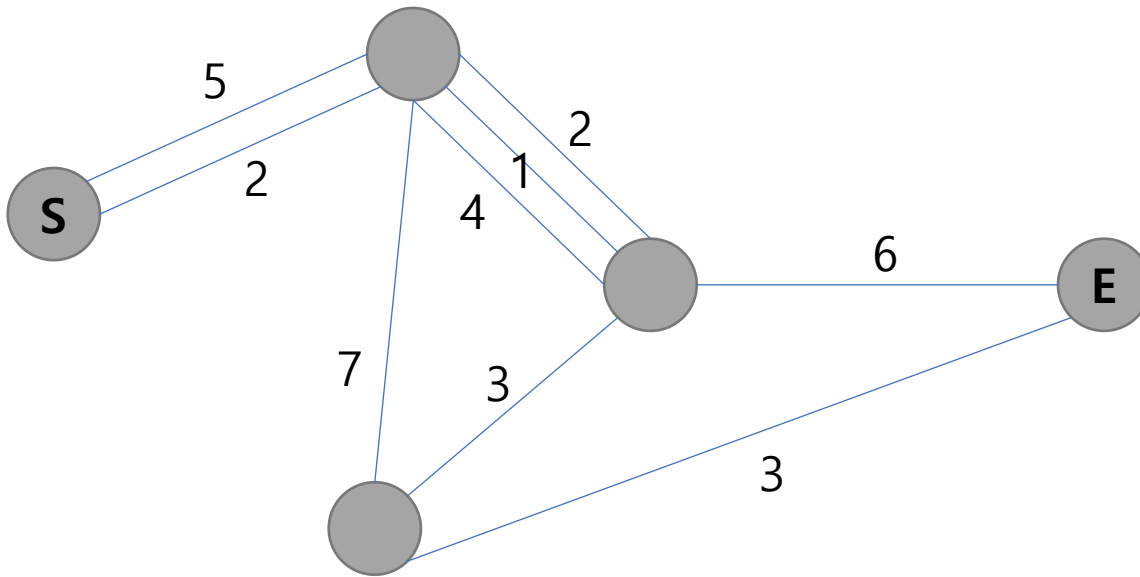


?

## 지난시간 도전문제 - 중량제한

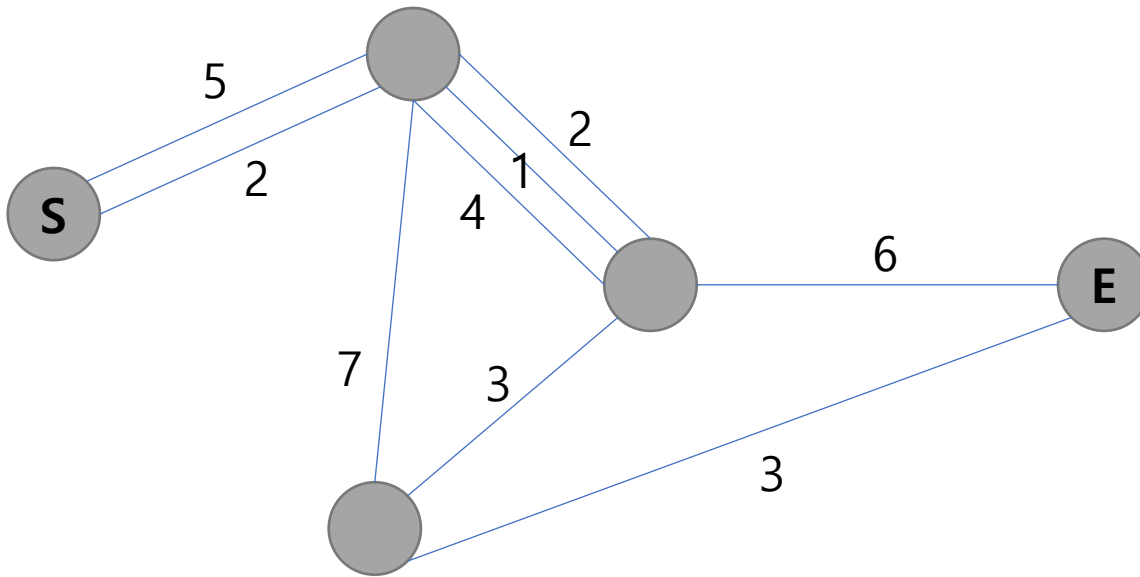
- w의 무게로 도착 가능한지?

- !visited[E]면 도착 불가능



## 지난시간 도전문제 - 중량제한

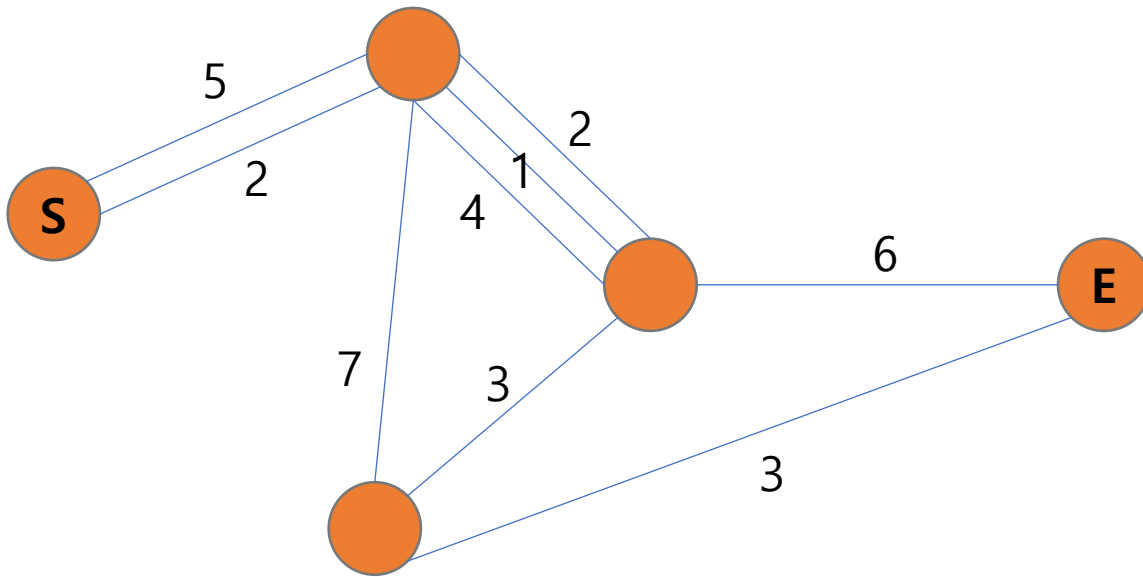
- $w$ 에 대해 이분탐색
- 이분탐색으로 설정된 각각의  $w$ 로 시작점에서 BFS진행



## 지난시간 도전문제 - 중량제한

-  $w$ 에 대해 이분탐색

-  $w = 1$

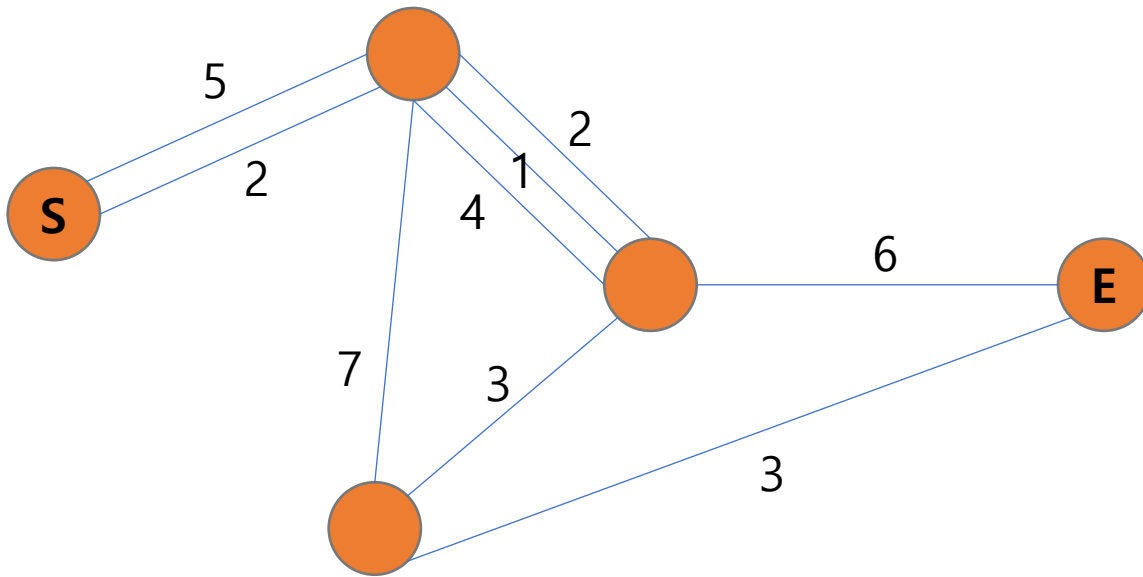




## 지난시간 도전문제 - 중량제한

-  $w$ 에 대해 이분탐색

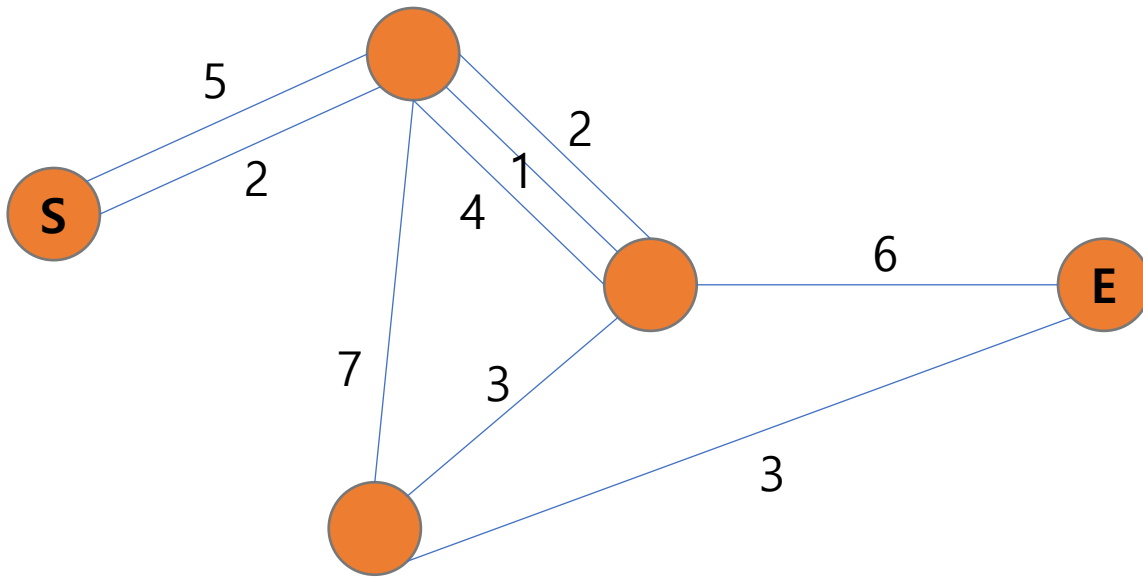
-  $w = 2$



## 지난시간 도전문제 - 중량제한

-  $w$ 에 대해 이분탐색

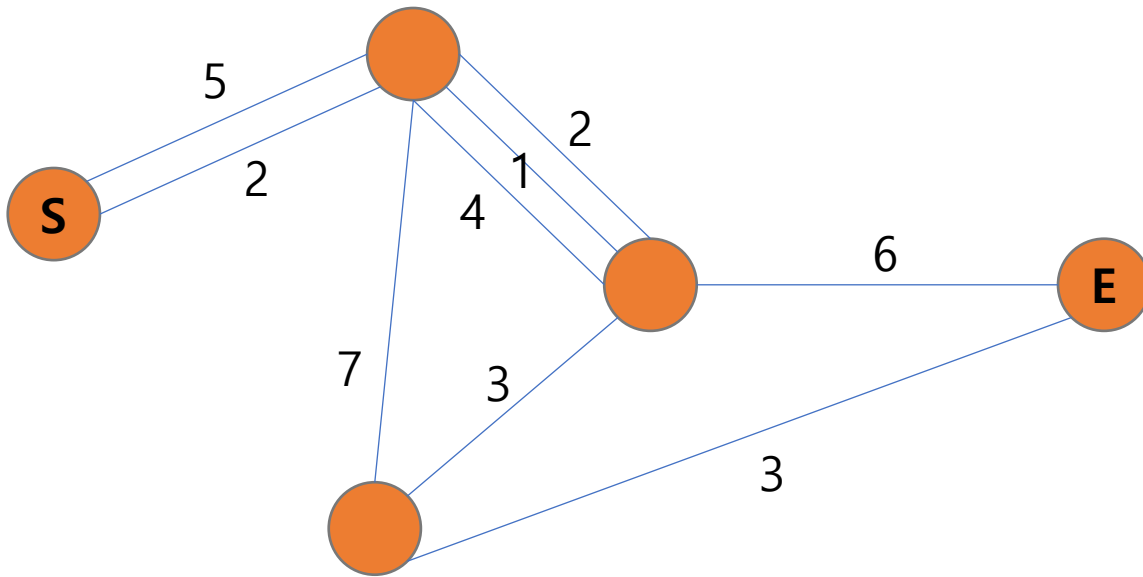
-  $w = 3$



## 지난시간 도전문제 - 중량제한

-  $w$ 에 대해 이분탐색

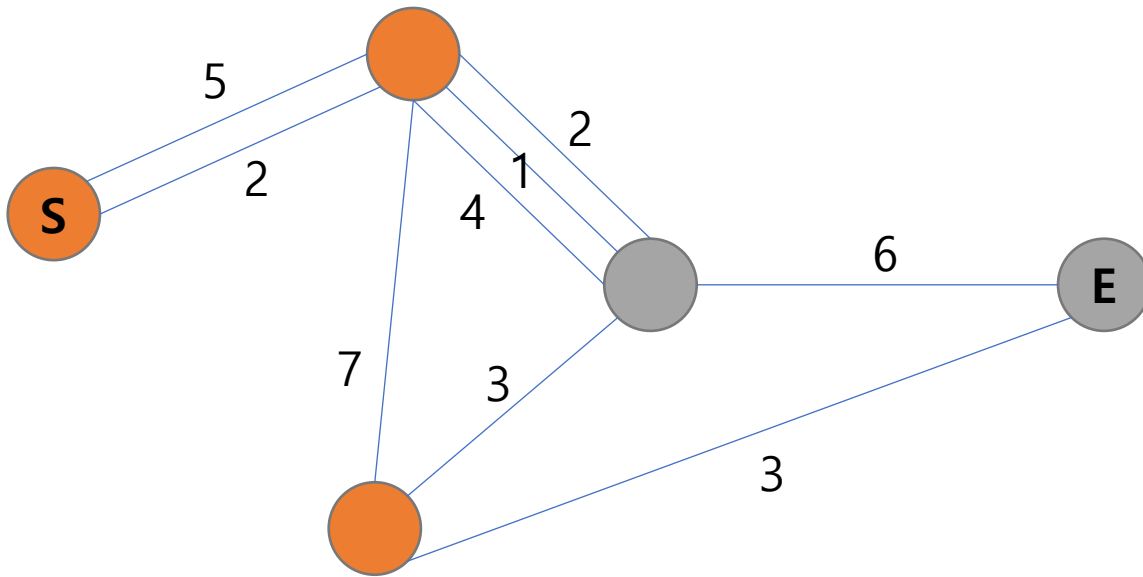
-  $w = 4$



## 지난시간 도전문제 - 중량제한

-  $w$ 에 대해 이분탐색

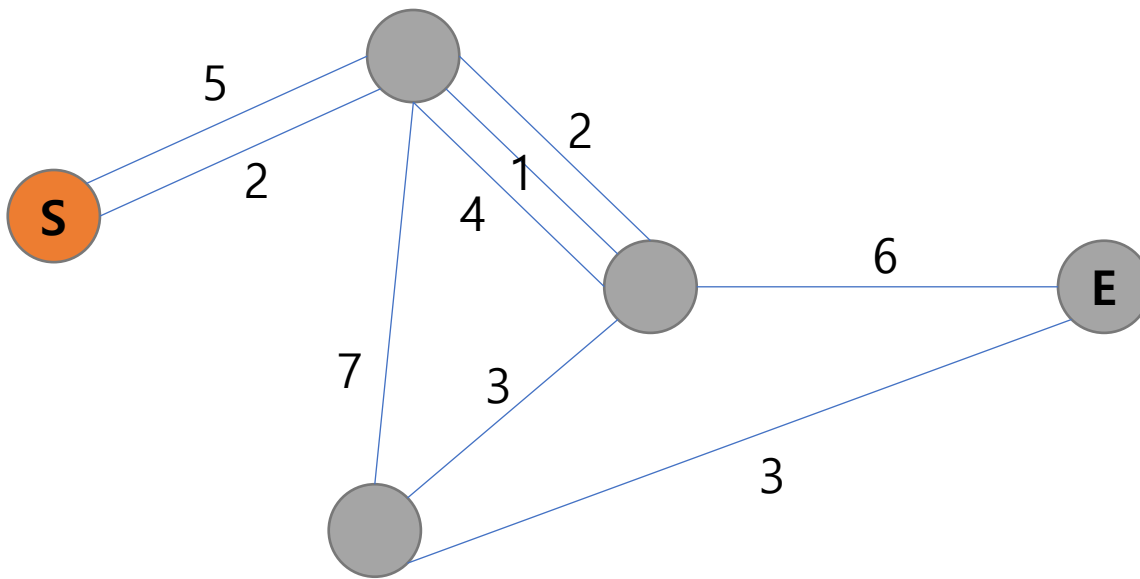
-  $w = 5$



## 지난시간 도전문제 - 중량제한

-  $w$ 에 대해 이분탐색

-  $w = 6$



## 지난시간 도전문제 - 중량제한

### - 다리의 정보 저장

#### - 일반적인 그래프 표현 방법

노드가 가지고 있어야 할 정보 :

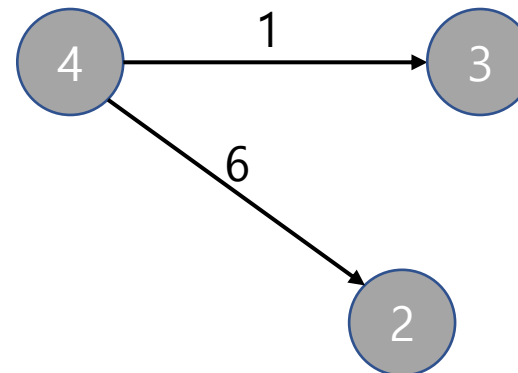
- 연결된 다른 노드
- 연결된 다른 노드로 가는 간선의 비용

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

비용

노드

`linked[4] = {{1,3}, {6,2}}`



4번 노드는 3번 노드와 1의 비용으로, 2번 노드와 6의 비용으로 연결

## 지난시간 도전문제 - 중량제한

### - 실제 구현

vector<pair<int, int> > linked[10001]

queue<int> q

```
bool bfs(int weight)
{
    for (int i = 0; i < 10001; i++)
    {
        visited[i] = false;
    }
    q.push(S);
    visited[S] = true;
    while (!q.empty())
    {
        int here = q.front();
        q.pop();
        for (int i = 0; i < linked[here].size(); i++)
        {
            int cost = linked[here][i].first;
            int next = linked[here][i].second;
            if (!visited[next] && cost >= weight)
            {
                visited[next] = true;
                q.push(next);
            }
        }
    }
    if (visited[E])
        return true;
    return false;
}
```

```
void bs(int l, int r)
{
    int mid = (l + r) / 2;
    if (l > r)
        return;
    if (bfs(mid))
    {
        MAX = max(MAX, mid);
        bs(mid + 1, r);
    }
    else
        bs(l, mid - 1);
}
```

```
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> N >> M;
    int a, b, c;
    for (int i = 0; i < M; i++)
    {
        cin >> a >> b >> c;
        linked[a].push_back(make_pair(c, b));
        linked[b].push_back(make_pair(c, a));
    }
    cin >> S >> E;
    bs(0, 1000000000);
    cout << MAX;
}
```

## 지난시간 도전문제 - 중량제한

### - 실제 구현

```
bool bfs(int weight)
{
    for (int i = 0; i < 10001; i++)
    {
        visited[i] = false;
    }
    q.push(S);
    visited[S] = true;
    while (!q.empty())
    {
        int here = q.front();
        q.pop();
        for (int i = 0; i < linked[here].size(); i++)
        {
            int cost = linked[here][i].first;
            int next = linked[here][i].second;
            if (!visited[next] && cost >= weight)
            {
                visited[next] = true;
                q.push(next);
            }
        }
    }
    if (visited[E])
        return true;
    return false;
}
```

cost : 연결된 다리의 무게 제한

next : 연결된 다리로 갈 수 있는 섬의 번호

끝섬에 weight의 무게로 도달 가능하면 true return



## ● ● 지난시간 도전문제 - 중량제한

### - 실제 구현

```
void bs(int l, int r)
{
    int mid = (l + r) / 2;
    if (l > r)
        return;
    if (bfs(mid))
    {
        MAX = max(MAX, mid);
        bs(mid + 1, r);
    }
    else
        bs(l, mid - 1);
}
```

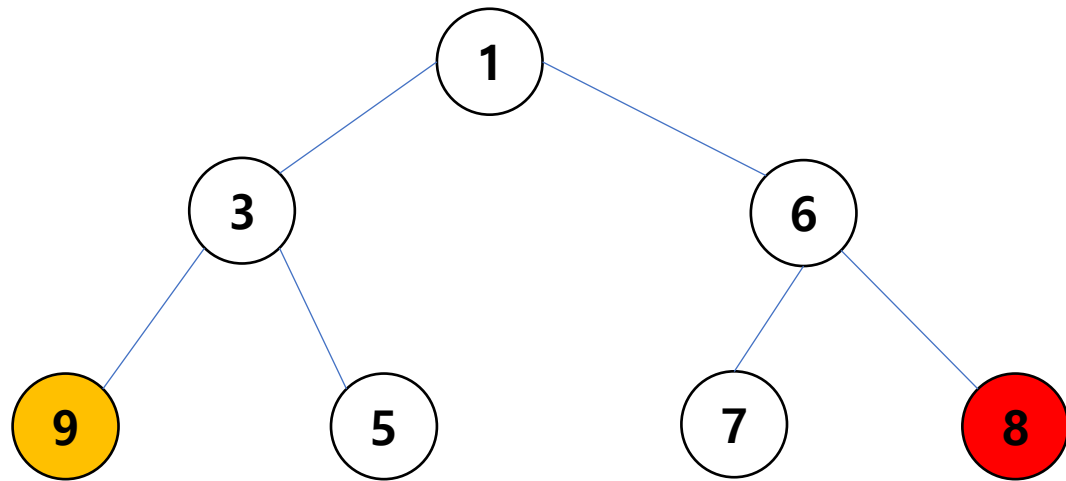
[l,r]에서 이분탐색 진행

```
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cin >> N >> M;
    int a, b, c;
    for (int i = 0; i < M; i++)
    {
        cin >> a >> b >> c;
        linked[a].push_back(make_pair(c, b));
        linked[b].push_back(make_pair(c, a));
    }
    cin >> S >> E;
    bs(0, 1000000000);
    cout << MAX;
}
```

- 그리디(Greedy)

- 모든 선택지를 고려해보지 않고, 각 단계마다 **지금 당장 가장 좋은** 방법만을 선택
- 사람의 직관과 비슷하다
- 부분적인 최적해를 통해서 전체의 최적해를 얻는 기법
- 수행 시간은 보통 탐색 알고리즘보다 더 빠르다
- 증명이 까다로운 경우가 많아 어렵다
  
- **정렬을 이용하면 쉬워지는 경우가 많다!**

- 제일 큰 숫자 찾기



탐색으로 구한 값

그리디로 구한 값

- 두더지 잡기

2		4
1		
	1	



	5	1
2	3	
		7

- 정당성 증명

- 탐욕적 선택 속성 : 보통 탐욕적 선택을 포함하지 않는 최적해의 존재를 가정하고,  
이 최적해를 적절히 조작하여 탐욕적 선택을 포함하는 최적해를 만들어서 증명

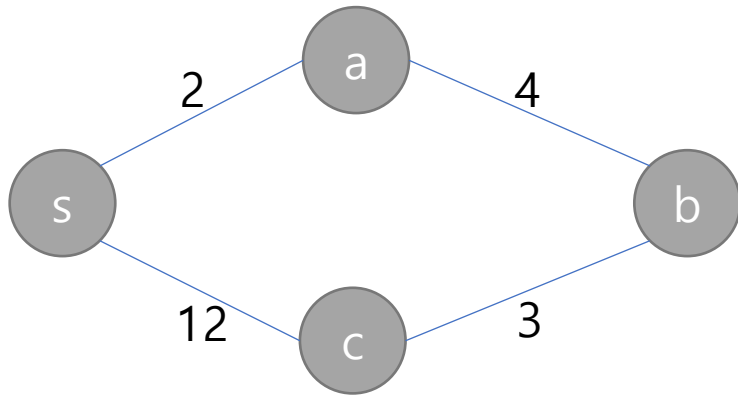
- 그리디가 쓰이는 알고리즘

- 다익스트라
  - 크루스칼(MST)

- 다익스트라(Dijkstra)
  - BFS와 마찬가지로 그래프에서 **1:N** 최단거리를 구하는 알고리즘
  - 간선의 가중치가 일정하지 않아도, 한 점에서 나머지 모든 점까지의 최단거리를 구할 수 있다.
  - 간선의 가중치가 음수이면 사용 불가
    - 벨만 포드 알고리즘으로 대체
  - 시간복잡도  $O(E \log V)$
  - 우선순위큐라는 자료구조를 사용한다

## - 다익스트라(Dijkstra)

- BFS와 유사하게 Queue를 사용하며, 시작점에서 가까운 순서대로 정점을 방문



늦게 발견한 정점이라도 더 먼저 방문(pop)할 수 있어야 한다

=> 우선순위큐 사용

- 다익스트라(Dijkstra)
  - 우선순위큐(Priority Queue)
    - 들어온 순서와 상관없이 무조건 우선순위가 높은(낮은) 순서대로 pop

	큐	우선순위큐
삽입(Enqueue)	$O(1)$	$O(\log n)$
삭제(Dequeue)	$O(1)$	$O(\log n)$
나오는순서	FIFO	우선순위





## - 다익스트라(Dijkstra)

### - 우선순위큐(Priority Queue)

- c++ 기준 priority\_queue의 기본 우선순위는 큰값이 먼저 나오도록

--	--	--	--

3			
---	--	--	--

3	1		
---	---	--	--

6	3	1	
---	---	---	--

3	1		
---	---	--	--

3	2	1	
---	---	---	--

## - 다익스트라(Dijkstra)

### - 우선순위큐(Priority Queue)에 정보 저장

1. 지금까지 온 경로의 길이

2. 현재 노드 번호



pair<int, int>

순서가 중요하다

### - 동작 과정

1. 구하고자 하는 dist들을 전부 INF로 초기화

2. 시작점을 pq에 Enqueue, dist[start] = 0

3. 현재 노드의 정보를 가지고 갈 수 있는 지점들의 dist들을 갱신 후 pq에 Enqueue

## - 실제 구현

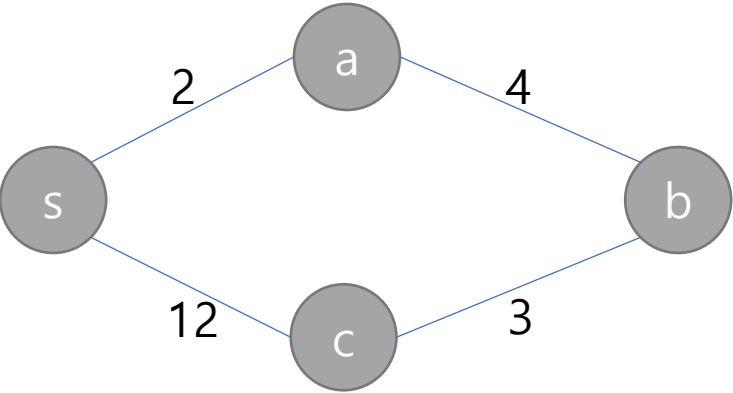
```
void dijkstra()
{
    pq.push(make_pair(0, start));
    dist[start] = 0;
    while (!pq.empty())
    {
        int cost = -pq.top().first;
        int here = pq.top().second;
        pq.pop();
        if (dist[here] < cost)
            continue;
        for (int i = 0; i < linked[here].size(); i++)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

cost를 음수로 취급하는 이유는

c++ 우선순위큐 기본 우선순위는

"큰값이 나오도록"이기 때문

- 동작 과정



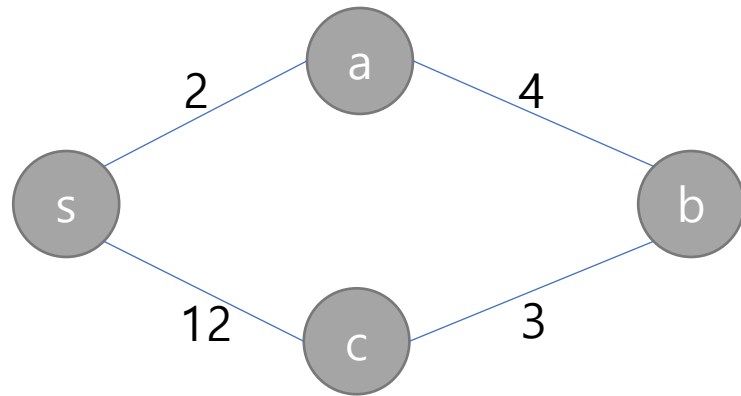
pq

--	--	--	--

dist

s	a	b	c
INF	INF	INF	INF

- 동작 과정



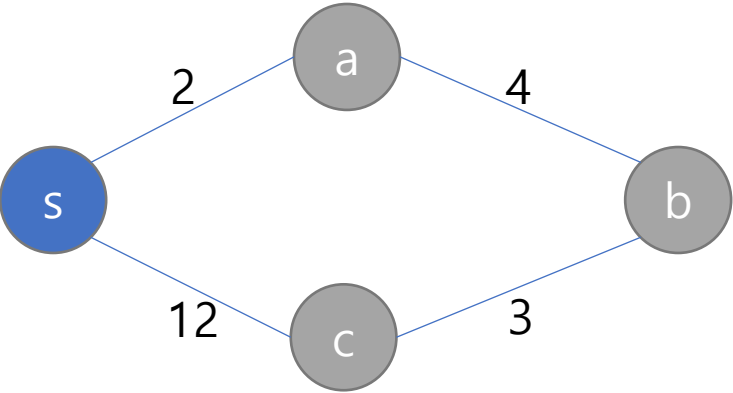
pq

(0,s)			
-------	--	--	--

dist

s	a	b	c
0	INF	INF	INF

- 동작 과정



here : s  
cost : 0

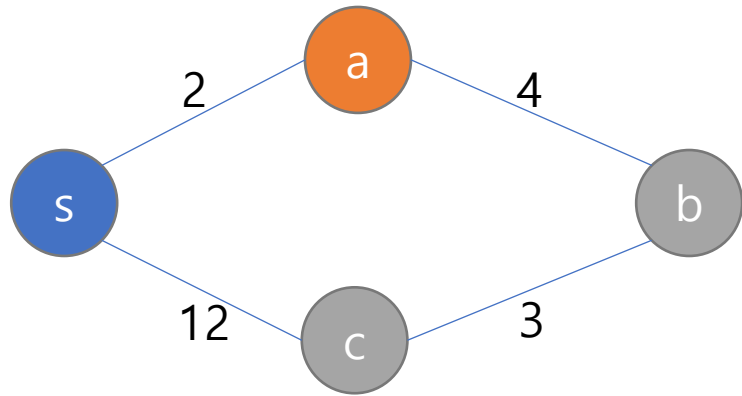
pq

--	--	--	--

dist

s	a	b	c
0	INF	INF	INF

## - 동작 과정



here : s

cost : 0

dist[a]와 cost+2를 비교

dist[a] > cost+2 이므로 dist[a] = cost+2 갱신 후 pq에 Enqueue

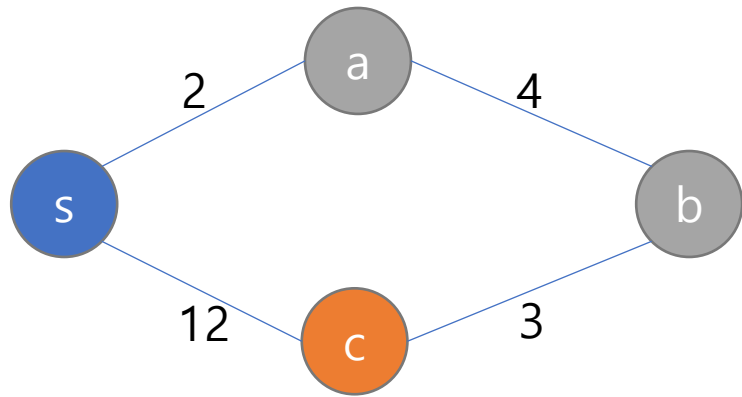
pq

(2,a)			
-------	--	--	--

dist

s	a	b	c
0	2	INF	INF

## - 동작 과정



here : s

cost : 0

dist[c]와 cost+12를 비교

dist[c] > cost+12 이므로 dist[c] = cost+12 갱신 후 pq에 Enqueue

pq

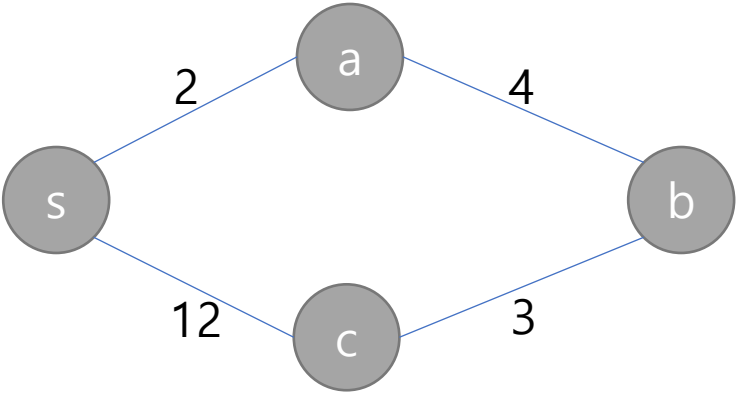
(2,a)	(12,c)		
-------	--------	--	--

dist

s	a	b	c
0	2	INF	12



- 동작 과정



here :  
cost :

s와 연결된 노드를 다 보았으므로 pq에서 pop한다.

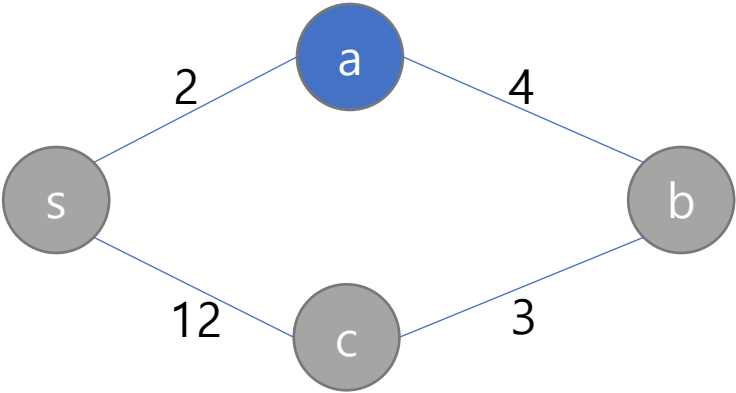
pq

(2,a)	(12,c)		
-------	--------	--	--

dist

s	a	b	c
0	2	INF	12

- 동작 과정



here : a  
cost : 2

s와 연결된 노드를 다 보았으므로 pq에서 pop한다.

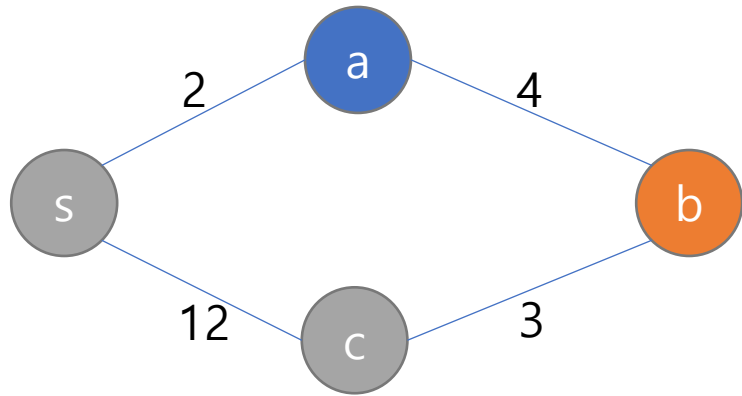
pq

(12,c)			
--------	--	--	--

dist

s	a	b	c
0	2	INF	12

## - 동작 과정



here : a  
cost : 2

dist[b]와 cost+4를 비교

dist[b] > cost+4 이므로 dist[b] = cost+4 갱신 후 pq에 Enqueue

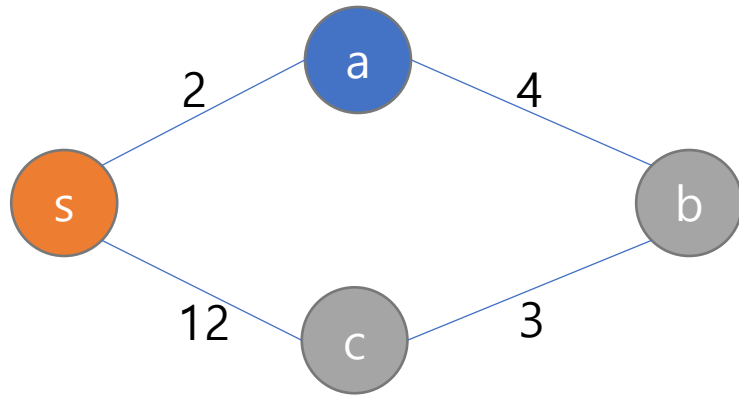
pq

(6,b)	(12,c)		
-------	--------	--	--

dist

s	a	b	c
0	2	6	12

## - 동작 과정



here : a

cost : 2

dist[s]와 cost+2를 비교

dist[s] < cost+2 이므로 무시

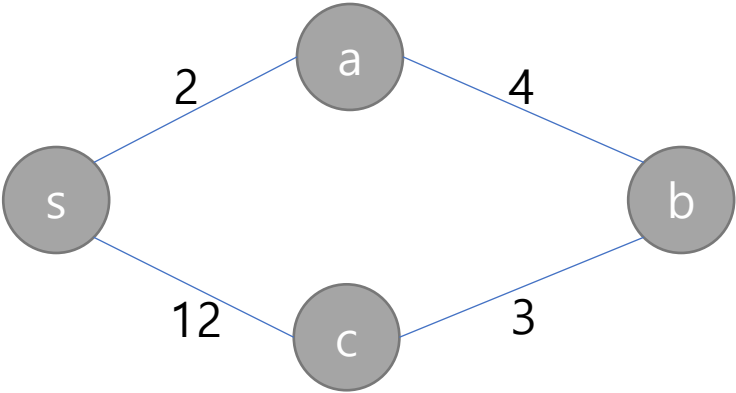
pq

(6,b)	(12,c)		
-------	--------	--	--

dist

s	a	b	c
0	2	6	12

- 동작 과정



here :  
cost :

a와 연결된 노드를 다 보았으므로 pq에서 pop한다

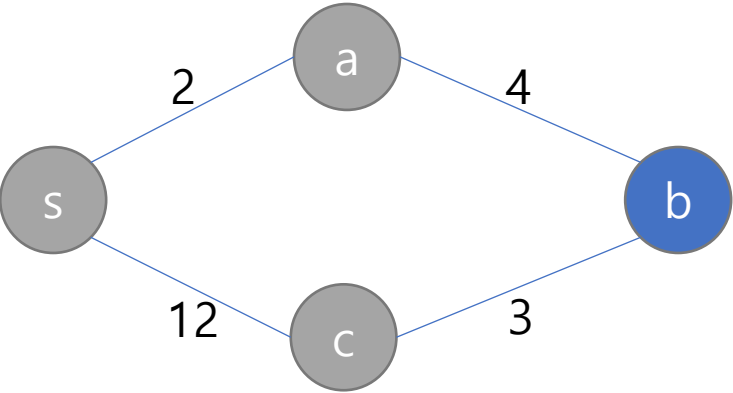
pq

(6,b)	(12,c)		
-------	--------	--	--

dist

s	a	b	c
0	2	6	12

- 동작 과정



here : b  
cost : 6

a와 연결된 노드를 다 보았으므로 pq에서 pop한다

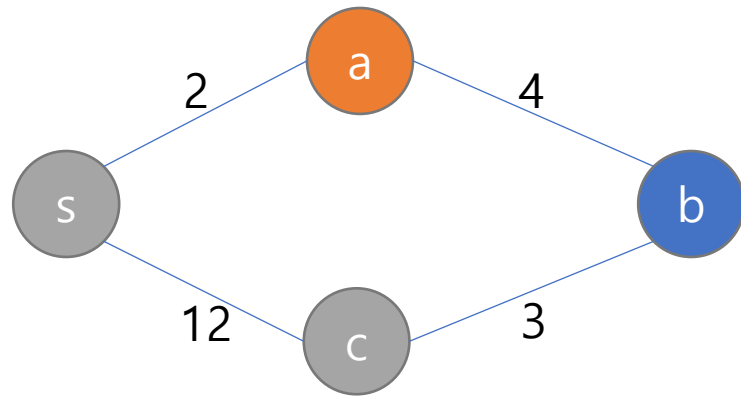
pq

(12,c)			
--------	--	--	--

dist

s	a	b	c
0	2	6	12

## - 동작 과정



here : b

cost : 6

dist[a]와 cost+2를 비교  
 $\text{dist}[a] < \text{cost} + 2$  이므로 무시

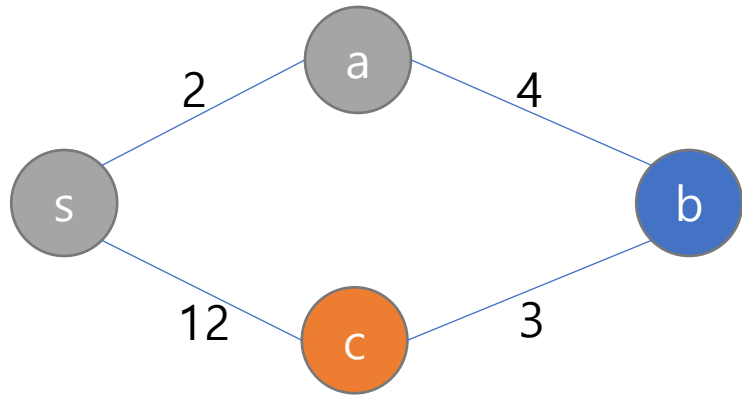
pq

(12,c)			
--------	--	--	--

dist

s	a	b	c
0	2	6	12

## - 동작 과정



here : b

cost : 6

dist[c]와 cost+3을 비교

dist[c] > cost+3 이므로 dist[c] = cost+3 갱신 후 pq에 Enqueue

pq

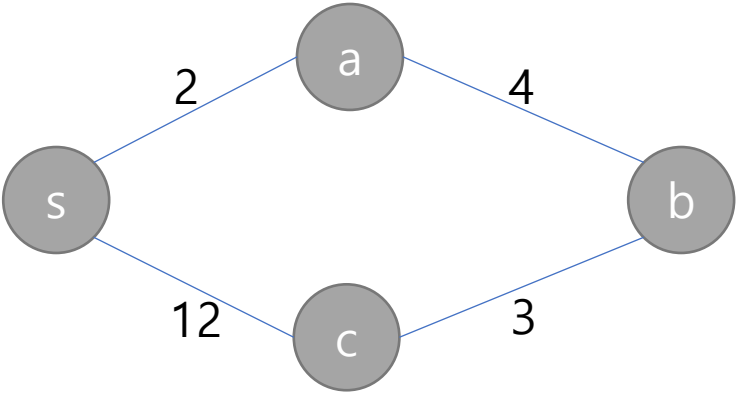
(9,c)	(12,c)		
-------	--------	--	--

dist

s	a	b	c
0	2	6	9



- 동작 과정



here :  
cost :

b와 연결된 노드를 다 보았으므로 pq에서 pop한다

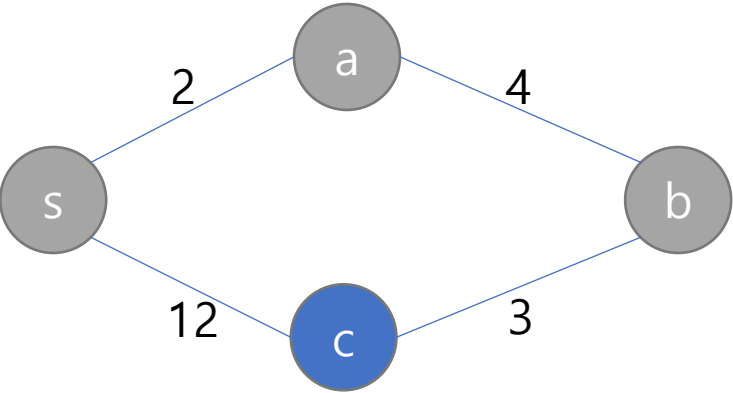
pq

(9,c)	(12,c)		
-------	--------	--	--

dist

s	a	b	c
0	2	6	9

- 동작 과정

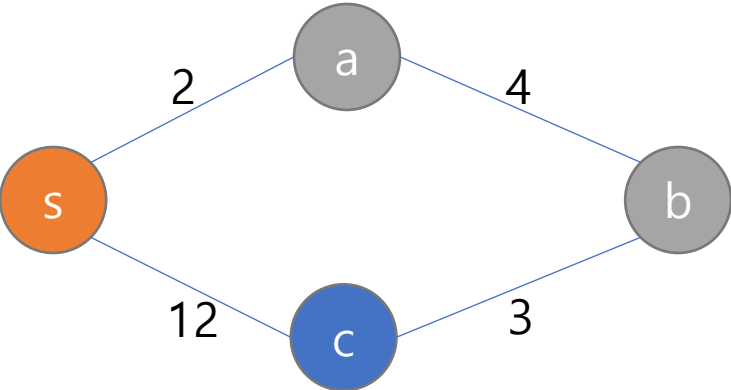


here : c  
cost : 9

pq	(12,c)			
----	--------	--	--	--

dist	s	a	b	c
	0	2	6	9

- 동작 과정



here : c  
cost : 9

dist[s]와 cost+12를 비교  
dist[s] < cost+12 이므로 무시

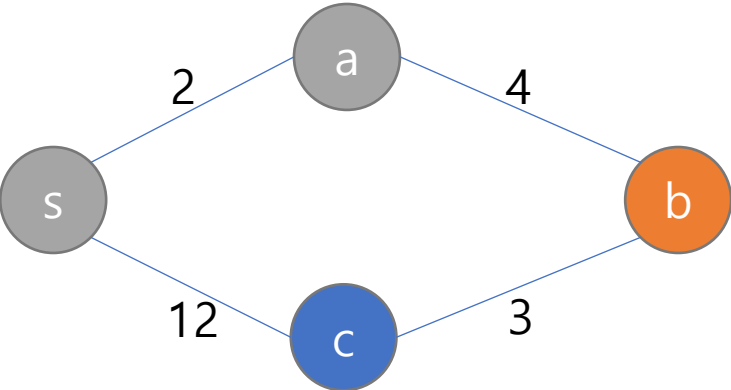
pq

(12,c)			
--------	--	--	--

dist

s	a	b	c
0	2	6	9

- 동작 과정



here : c  
cost : 9

dist[b]와 cost+3을 비교  
dist[b] < cost+3 이므로 무시

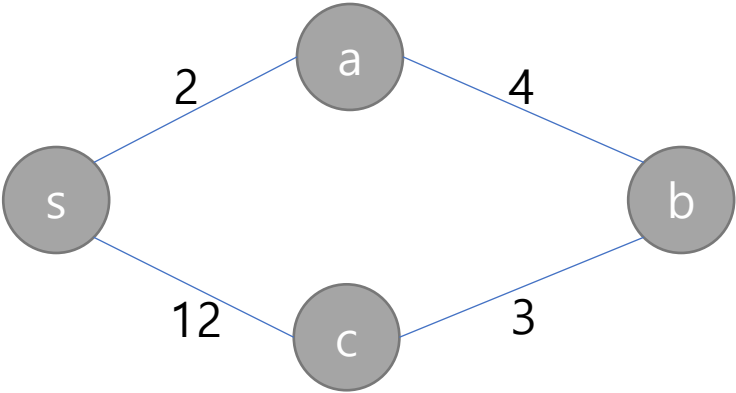
pq

(12,c)			
--------	--	--	--

dist

s	a	b	c
0	2	6	9

- 동작 과정



here :  
cost :

c와 연결된 노드를 다 보았으므로 pq에서 pop한다

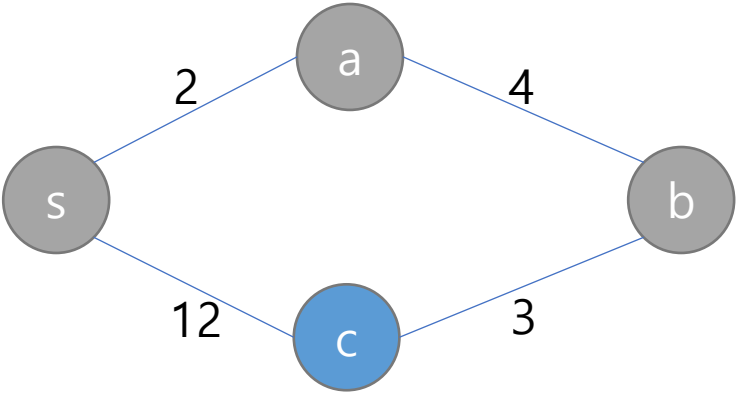
pq

(12,c)			
--------	--	--	--

dist

s	a	b	c
0	2	6	9

- 동작 과정



here : c  
cost : 12

여기서  $\text{dist}[c] = 9 < 12$ 이므로 continue;

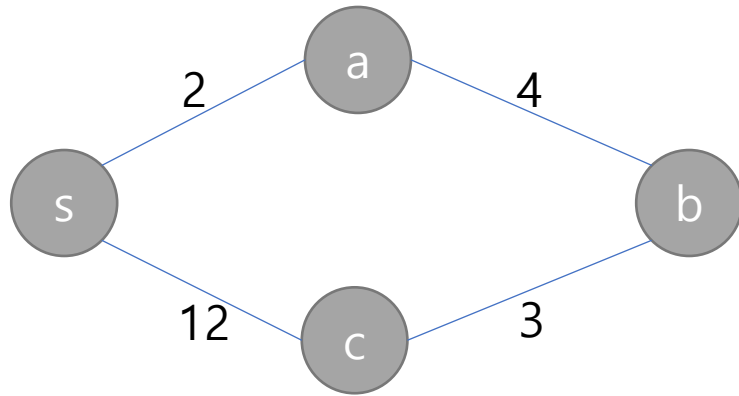
pq

--	--	--	--

dist

s	a	b	c
0	2	6	9

## - 최종 결과



dist

s	a	b	c
0	2	6	9

```

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));
            }
        }
    }
}
  
```

## - 다익스트라 vs BFS

	다익스트라	BFS
쓰이는 분야	1:N 최단거리	
사용 자료구조	우선순위 큐	큐
간선의 가중치	0 또는 양수	0 또는 1
시간복잡도	$O(E \log V)$	$O(V + E)$
dist 초기화	INF	-1



1. ATM : <https://www.acmicpc.net/problem/11399>

2. 회의실 배정 : <https://www.acmicpc.net/problem/1931>

문제 요약 :

<https://www.acmicpc.net/problem/13023>

- $1 \leq N \leq 1,000, 1 \leq P_i \leq 1,000$
- 사람마다 돈을 인출하는데 걸리는 시간은  $P_i$  이다
- 전체 걸리는 시간의 합의 최소는 얼마인가?

고려해야 할 사항 :

- 내가 돈을 인출 하는동안 다른 사람들은 인출할 수 없다.

## ● 그리드를 이용하는 문제 - ATM

- 예제

$P_1 = 3, P_2 = 1, P_3 = 4, P_4 = 3, P_5 = 2$  인 경우

1번 사람	2번 사람	3번 사람	4번 사람	5번 사람
3	3+1	3+1+4	3+1+4+3	3+1+4+3+2

직관 : 앞 순서 사람의  $P_i$ 가 뒤에도 영향을 미치므로,

앞 순서에  $P_i$ 가 적은 사람을 배치

$P_i$ 가 오름차순인 최적해가 존재한다!

## ● 그리디를 이용하는 문제 - ATM

### - 증명

$P_1 = 2, P_2 = 1, P_3 = a, P_4 = b, P_5 = c$  인 경우

1번 사람	2번 사람	3번 사람	4번 사람	5번 사람
2	2+1	2+1+a	2+1+a+b	2+1+a+b+c

- 탐욕적 선택 속성 : 보통 탐욕적 선택을 포함하지 않는 최적해의 존재를 가정하고, 이 최적해를 적절히 조작하여 탐욕적 선택을 포함하는 최적해를 만들어서 증명

가정 :  $P_i$ 가 오름차순이지 않은 최적해가 존재한다!

단순히 1번사람과 2번사람의 순서만 바꾸면 되므로 항상  $P_i$ 가 오름차순인 최적해가 존재한다

따라서 오름차순으로 정렬했을 때, 최적해를 얻는것이 불가능한 경우는 없다.

## ● ● 그리드를 이용하는 문제 - ATM

- 실제 풀이

$P_i$  를 단순히 오름차순으로 정렬해주고  $\sum_{i=1}^n P_i \times i$  를 계산하면 끝

```
vector<int> P
sort(P.begin(), P.end())

for(int i=1; i<=N; i++)
    ret += i*P[i-1]

cout<<ret
```

문제 요약 :

<https://www.acmicpc.net/problem/1931>

- $1 \leq N \leq 100,000$
- 한 개의 회의실, N개의 회의
- 회의는 한번 시작되면 도중에 중단될 수 없다
- 할 수 있는 회의 개수의 최대값은?

고려해야 할 사항 :

- 빠른 입출력
- 회의들중에 어떤 회의들을 선택해야 최적해가 나올것인가

## ● 그리드를 이용하는 문제 - 회의실 배정

- 직관?

1. 길이가 짧은 회의부터 배정



2. 가장 빨리 시작하는 회의부터 배정



3. 가장 빨리 끝나는 회의부터 배정

?

## ● ● 그리디를 이용하는 문제 - 회의실 배정

### - 가장 빨리 끝나는 회의부터 배정

가장 종료 시간이 빠른 회의( $S_{\min}$ )를 포함하는 최적해가 반드시 존재한다

이것을 증명하려면?

가장 종료 시간이 빠른 회의( $S_{\min}$ )를 포함하지 않는 최적해가 존재한다 가정

이때 이 최적해에서 선택한 회의 목록중에 첫 번째로 배정된 회의를 지우고  $S_{\min}$ 을 대신 추가

$S_{\min}$ 은 가장 빨리 끝나는 회의이기 때문에 두 번째 회의와 겹치는 일이 없다.

따라서 새로 만든 회의 리스트도 최적해 중 하나가 된다.



## ● 그리드를 이용하는 문제 - 회의실 배정

- 가장 빨리 끝나는 회의부터 배정

$S_{\min}$ 을 포함하지 않는 최적해




새로운 최적해




## ● ● 그리드를 이용하는 문제 - 회의실 배정

### - 실제 풀이

회의가 시작하마자 동시에 끝날수도 있다.

(4, 5)            인풋이 이렇게 들어오면 상관 없지만  
(5, 5)

(5, 5)            이렇게 되면 (4, 5)를 무시하게 될 수 있다.  
(4, 5)

따라서 빨리 끝나는 회의 순으로 정렬하되, 만약 끝나는 시간이 같다면 빨리 시작하는 회의 순으로

## ● ● 그리드를 이용하는 문제 - 회의실 배정

### - 실제 구현

```
int main()
{
    cin.tie(NULL);
    ios_base::sync_with_stdio(false);
    cin >> N;
    int a, b;
    for (int i = 0; i < N; i++)
    {
        cin >> a >> b;
        arr.push_back(make_pair(a, b));
    }
    sort(arr.begin(), arr.end(), compare);
    int ans = 0;
    int start = 0;
    for (int i = 0; i < N; i++)
    {
        if (arr[i].first < start)
            continue;
        ans++;
        start = arr[i].second;
    }
    cout << ans;
}
```

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int N;
vector<pair<int, int> > arr;

bool compare(pair<int, int> a, pair<int, int> b)
{
    if (a.second == b.second)
        return a.first < b.first;
    return a.second < b.second;
}
```

## 다익스트라를 이용하는 문제 - 특정 거리의 도시 찾기

문제 요약 :

<https://www.acmicpc.net/problem/18352>

- $2 \leq N \leq 300,000$ ,  $1 \leq M \leq 1,000,000$ ,  $1 \leq K \leq 300,000$ ,  $1 \leq X \leq N$
- 방향 그래프이며, 간선들의 가중치가 모두 1이다
- X로부터 최단거리가 K인 도시들의 번호를 오름차순으로 출력

고려해야 할 사항 :

- 빠른 입출력
- 간선들의 가중치가 1이니 BFS로도 풀릴 것 같은데?
- 다익스트라

## 다익스트라를 이용하는 문제 - 특정 거리의 도시 찾기

- 간선들의 가중치가 1이니 BFS로 풀릴 것 같은데?

당연!

```
void BFS(int start)
    q.Enqueue(start)
    dist[start] = 0
    while(!q.empty())
        int here = q.Dequeue()
        for(int i=0; i<linked[here].size(); i++)
            if( dist[linked[here][i]] == -1 )
                dist[linked[here][i]] = dist[here] + 1
                q.Enqueue(linked[here][i])
```

## 다익스트라를 이용하는 문제 - 특정 거리의 도시 찾기

- 간선들의 가중치가 양수이니 다익스트라로 풀릴 것 같은데?

당연!

```
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));
            }
        }
    }
}
```

문제 요약 :

<https://www.acmicpc.net/problem/18352>

- $1 \leq V \leq 20,000, 1 \leq E \leq 300,000$
- 방향 그래프이며, 간선들의 가중치가 모두 10 이하의 자연수이다
- 서로 다른 두 정점 사이에 여러 개의 간선이 존재할 수 있다.

고려해야 할 사항 :

- 빠른 입출력
- 간선들의 가중치가 자연수이다
- $O(E \log V)$

## 다익스트라를 이용하는 문제 - 최단경로

- 그래프의 표현 방법

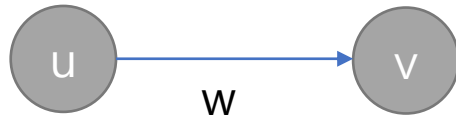
`vector<pair<int, int> > linked[N]`

간선 비용

연결된 노드 번호

$(u, v, w)$

->



->

`linked[u] : {w,v}`



## 다익스트라를 이용하는 문제 - 최단경로

### - 실제 구현

```
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));
            }
        }
    }
}
```

```
int main()
{
    cin.tie(NULL);
    ios_base::sync_with_stdio(false);
    cin >> V >> E;
    cin >> K;
    int a, b, c;
    for (int i = 0; i < E; i++)
    {
        cin >> a >> b >> c;
        linked[a].push_back(make_pair(c, b));
    }
    dijk();
    for (int i = 1; i <= V; i++)
    {
        if (dist[i] != INF)
            cout << dist[i] << "\n";
        else
            cout << "INF" << "\n";
    }
}
```



## 도전 문제!

1. 동전 0 : <https://www.acmicpc.net/problem/11047>
2. 최소비용 구하기 : <https://www.acmicpc.net/problem/1916>
3. 숨바꼭질 3 : <https://www.acmicpc.net/problem/13549>
4. 미로만들기 : <https://www.acmicpc.net/problem/2665>

## ● ● 자유로운 질문 및 토의!