대회 / 코테 알고리즘 정리

https://github.com/rlatjwls7882

Contents

1 대회 / 코테 알고리즘 정리

이 PDF는 와이파이를 사용할 수 없는 상황에서 참고하기 위해, 깃허브의 마크다운 파일을 이어 붙여 변환한 것입니다.
파일이 일부 깨져 있거나, 원문에 비해 누락된 내용이 있을 수 있으니 가능하면 깃허브에서 직접 내용을 확인하세요.

1.1 목표

- 증명보다는 알고리즘 구현 위주로 설명 추가
- 알고리즘 복습 및 추가 (LCA, BCC, EEA, ...)

1.2 목차

- 기본 알고리즘
  - Binary Search (이분 탐색)
  - Prefix Sum (누적 합)
  - Two Pointer

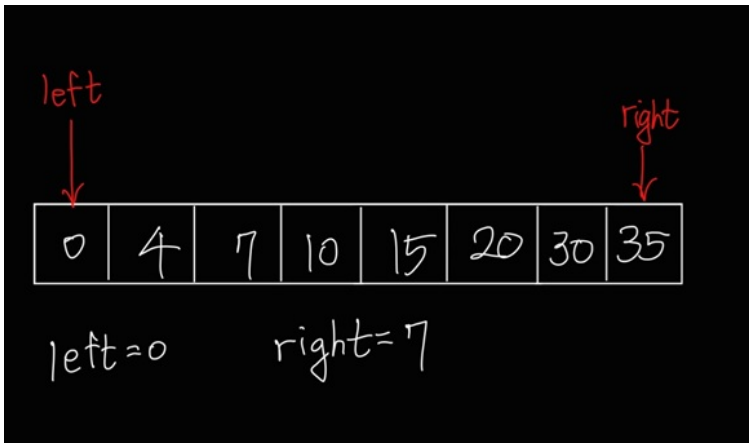
  - DSU (Disjoint Set Union, 분리 집합)
  - Backtracking
- 그래프
  - 경로 탐색
    - 기본 탐색
      - DFS (Depth First Search, 깊이 우선 탐색)
      - BFS (Breadth First Search, 너비 우선 탐색)
    - 최단 경로
      - Dijkstra's Algorithm
      - Bellman-Ford Algorithm
      - Floyd-Warshall Algorithm
      - SPFA (Shortest Path Faster Algorithm)
  - DAG(Directed Acyclic Graph)
    - Kahn's Algorithm (Topological Sort, 위상 정렬)
  - 최소 스패닝 트리
    - Kruskal's Algorithm
  - 유량
    - Kuhn's Algorithm (Maximum Bipartite Matching, 이분 매칭)
    - Edmonds-Karp Algorithm
    - Dinic's Algorithm
    - MCMF (Min Cost Max Flow) Algorithm
  - 컴포넌트 분해
    - Tarjan's Algorithm (SCC)
    - 2-SAT (2-Satisfiability)
  - 트리
    - Segment Tree
    - Lazy Propagation
    - Merge Sort Tree
    - HLD (Heavy Light Decomposition)
- 문자열
  - KMP (Knuth-Morris-Pratt) Algorithm
  - Trie
  - Aho-Corasick
- 기하
  - CCW (Counter ClockWise) Algorithm
  - Line Intersection
  - Graham's Scan (Convex Hull, 볼록 껍질)
  - Point in Convex Hull Algorithm (볼록 껍질 내부의 점 판정 알고리즘)
  - Rotating Calipers (회전하는 캘리퍼스)
- 스위핑
  - Sweeping Algorithm
  - Imos Method (いもす法)
- DP
  - TSP (Traveling Salesman Problem, 외판원 순회 문제)
  - Deque Trick
- 쿼리 처리

- Offline Query
- Sqrt Decomposition (Square Root Decomposition, 평방 분할)
- Mo's Algorithm
- Parallel Binary Search (병렬 이분 탐색)

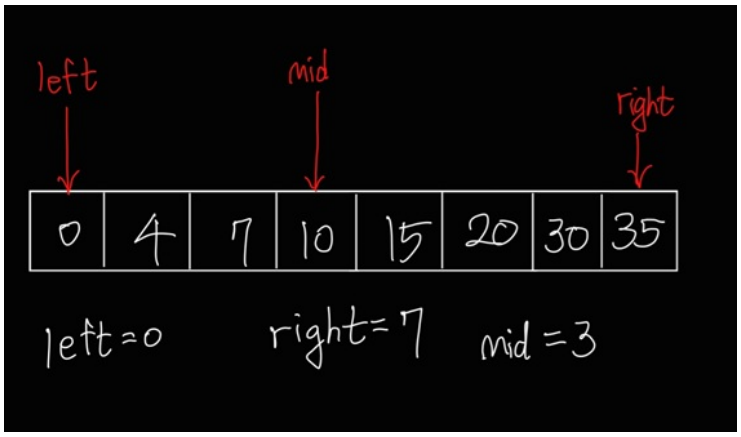
1.3 Binary Search (이분 탐색)

정렬된 데이터에서 원하는 값을 찾기 위해 탐색 범위를 절반씩 줄여가는 알고리즘

시간복잡도 : O(MlogN) (N : 데이터 개수, M : 탐색 횟수)

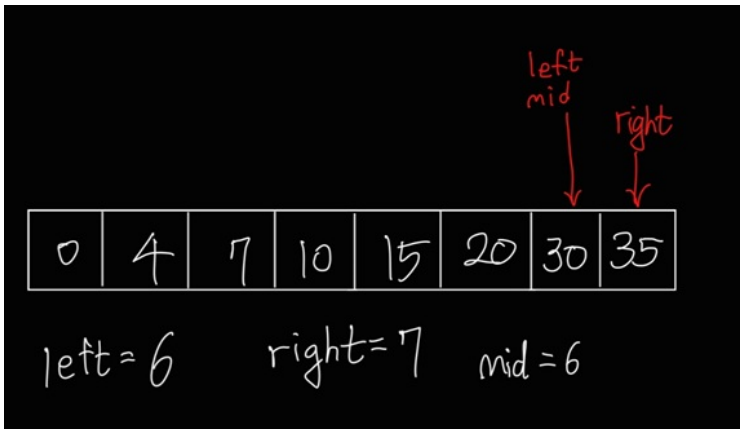


정렬된 배열 A에서 21이라는 값을 찾는다면 초기의 left와 right 값은 다음과 같고, 전체 범위를 나타냅니다.

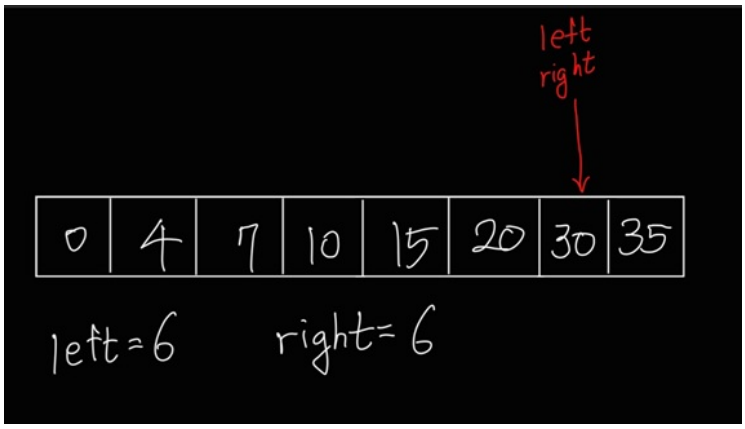


첫번 째 반복에서는, mid=3으로 찾는 값 21보다 A[mid]인 10이 작아 자동으로 0~mid까지 모든 값이 21보다 작은 것을 알 수 있다. (left = mid+1)

두번 째 반복에서는, mid=5로 찾는 값 21보다 A[mid]인 20이 작아 자동으로 0~mid까지 모든 값이 21보다 작은 것을 알 수 있다. (left = mid+1)



세번 째 반복에서는, mid=6로 찾는 값 21보다 A[mid]인 30이 크거나 같아 자동으로 mid+1~right까지 모든 값이 21보다 크거나 같은 것을 알 수 있다. (right = mid)



마지막으로 A[left]나 A[right]이 찾는 값인 21과 같다면 이 배열에 21이 존재한다.

연습 문제 (백준 1920번)

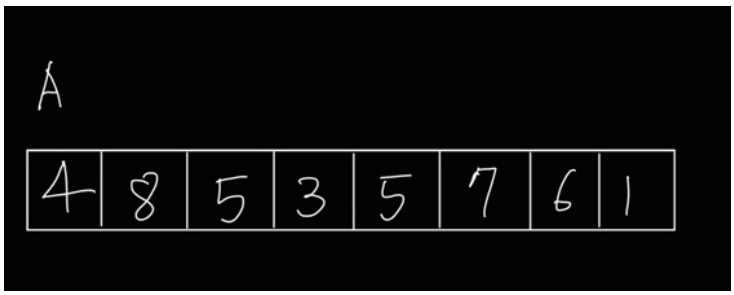
```
/** https://www.acmicpc.net/problem/1920 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

int n;
int A[100'001];

/**
 * Binary Search
 * 처음 범위 : 0 ~ n-1
 *
 * right가 left보다 큰 동안 :
 * - mid를 (left+right)으로 설정
 * - mid 위치의 값이 현재 탐색중인 값보다 작다면 left = mid+1 (left ~ mid의 값은 전부 작다는 것이 확정
 * - 그렇지 않다면 right = mid (mid+1 ~ right의 값은 전부 크다는 것이 확정되었으니 커팅)
 */
bool binary_search(int val) {
    int left=0, right=n-1;
    while(left<right) {
        int mid = (left+right)/2;
        if(A[mid]<val) left=mid+1;
        else right=mid;
    }
    return A[left] == val;
}

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    cin >> n;
```

```
    for(int i=0;i<n;i++) cin >> A[i];
    sort(A, A+n); // 오름차순으로 정렬

    int m; cin >> m;
    while(m--) {
        int val; cin >> val; // 현재 탐색할 값
        cout << binary_search(val) << '₩n';
    }
}
```

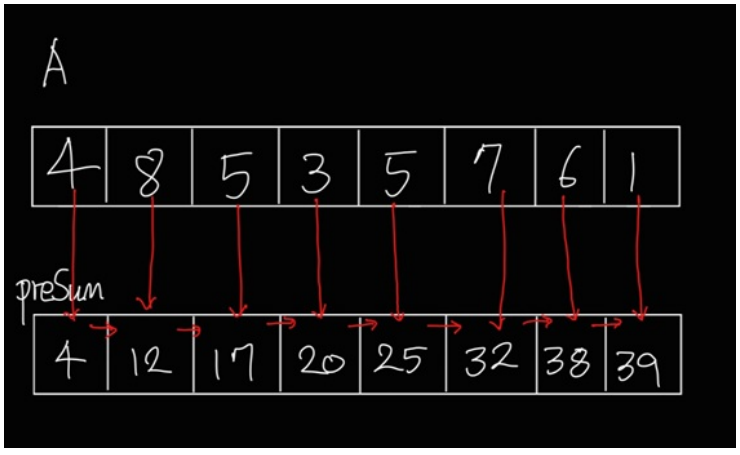
1.4 Prefix Sum (누적 합)

배열의 각 인덱스까지의 합을 미리 계산해 두어, 임의 구간의 합을 O(1)에 구하는 알고리즘

시간복잡도 : 전처리 O(N), 쿼리 O(1) (N : 데이터 개수)



입력된 배열 A가 다음과 같다고 하자.



그러면 배열 A의 누적 합 preSum은 다음과 같다고 할 수 있다. 여기서 preSum[i]는 A[0] ~ A[i] 까지의 합이다.

이제 A[i] ~ A[j] 까지의 합은 A[0] ~ A[j]의 합 - (A[0] ~ A[i-1]의 합)으로 나타낼

수 있고, 이는 preSum[j] - preSum[i-1]과 같다.

연습 문제 (백준 11659번)

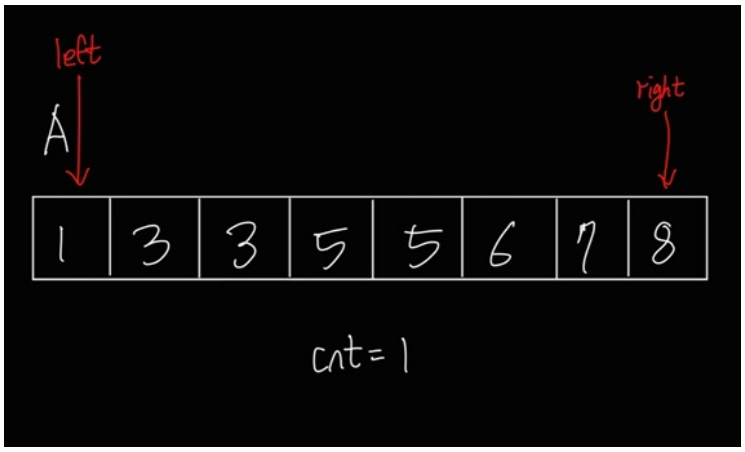
```cpp
/** https://www.acmicpc.net/problem/11659 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

// MAX : 수의 최대 개수
const int MAX = 100'001;

// preSum[i] : 1번 수부터 i번 수까지의 누적 합
int preSum[MAX];

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    for(int i=1;i<=n;i++) {
        cin >> preSum[i]; // i번째 수 입력
        preSum[i] += preSum[i-1]; // i번째 수에 0 ~ i-1 수의 합 더하기 -> 0 ~ i 수의 합 완성
    }

    while(m--) {
        int i, j; cin >> i >> j;
        cout << preSum[j] - preSum[i-1] << '₩n'; // i ~ j 수의 합 = 0 ~ j 수의 합 - 0 ~ i-1 수의 합
    }
}
```

1.5 Two Pointer

두 개의 포인터를 움직이며 배열이나 리스트에서 원하는 조건을 만족하는 구간을 효율적으로 찾는 알고리즘

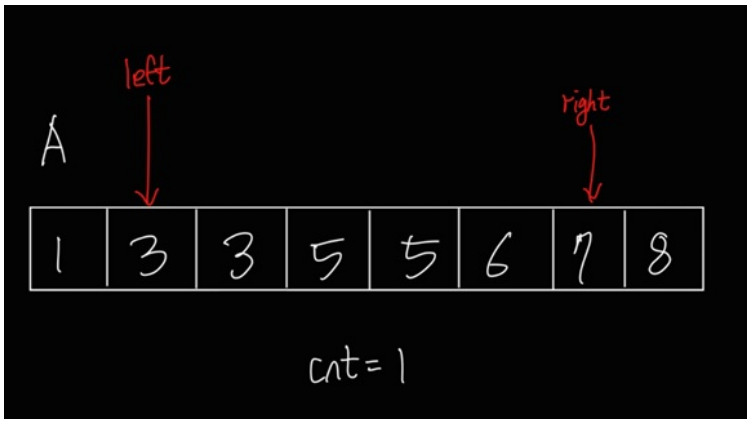
시간복잡도 : O(NlogN) (N : 데이터 개수, 정렬 O(NlogN) + 스캔 O(N))

배열 A가 위와 같이 주어졌고, 두 수의 합이 9가 되는 쌍을 구한다고 하자.

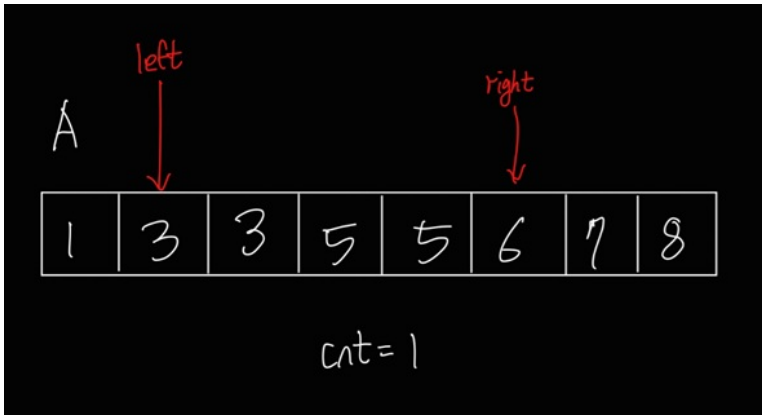


left를 왼쪽 끝, right를 오른쪽 끝으로 두고 비교를 시작한다.

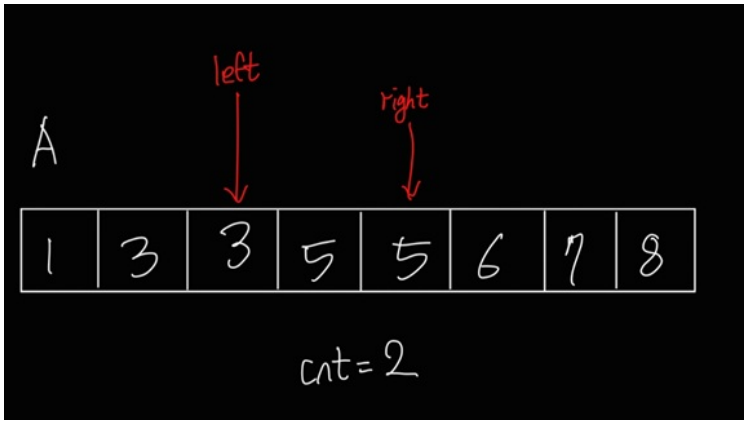
A[left]+A[right]가 9와 같기에 cnt를 1 증가, left를 1 증가, right를 1 감소시킨다.



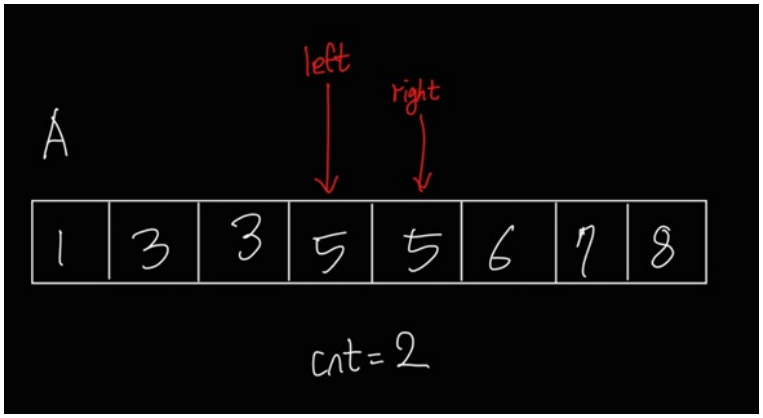
A[left]+A[right]가 9보다 크기에 right를 1 감소시킨다.



A[left]+A[right]가 9와 같기에 cnt를 1 증가, left를 1 증가, right를 1 감소시킨다.



A[left]+A[right]가 9보다 작기에 left를 1 증가시킨다.



A[left]+A[right]가 9보다 크기에 right를 1 감소시킨다.

연습 문제 (백준 3273번)

```
/** https://www.acmicpc.net/problem/3273 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

int a[100'000];

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n; cin >> n;
    for(int i=0;i<n;i++) cin >> a[i];
    int val; cin >> val;
    sort(a, a+n); // 정렬
```

```
    int cnt=0, left=0, right=n-1; // left와 right는 각각 왼쪽 끝에 위치한 피벗, 오른쪽 끝에 위치한 피벗
    while(left<right) {
        if(a[left]+a[right]<val) { // 두 원소의 합이 val보다 작으면
            left++; // 왼쪽 피벗을 오른쪽으로 이동
        } else if(a[left]+a[right]>val) { // 두 원소의 합이 val보다 크면
            right--; // 오른쪽 피벗을 왼쪽으로 이동
        } else { // 두 원소의 합이 val과 동일하면
            cnt++;
            left++; // 왼쪽 피벗을 오른쪽으로, 오른쪽 피벗을 왼쪽으로 이동 (사용한 원소 제외)
            right--;
        }
    }
    cout << cnt;
}
```

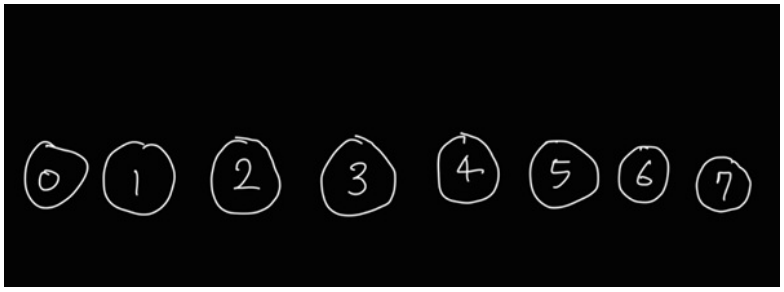
1.6 DSU (Disjoint Set Union, 분리 집합)

서로 겹치지 않는 집합을 관리하고 합치거나 찾는 연산을 효율적으로 처리하는 자료구조

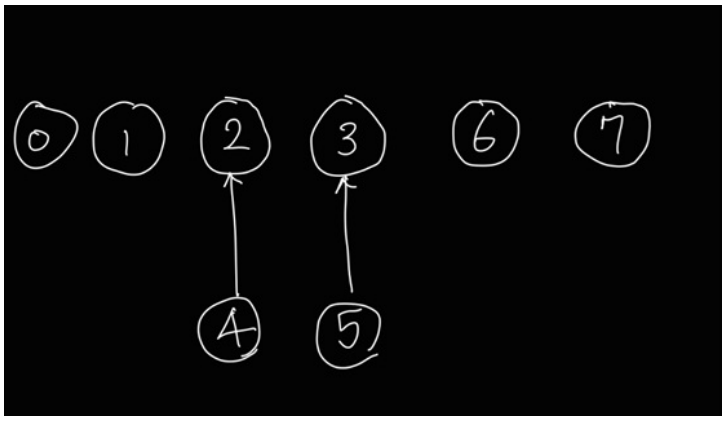
시간복잡도 : O(α(N)) (α : 역아커만 함수 ≒ 상수 시간, N : 데이터 개수)

DSU에는 union 연산과, find 연산이 존재한다.

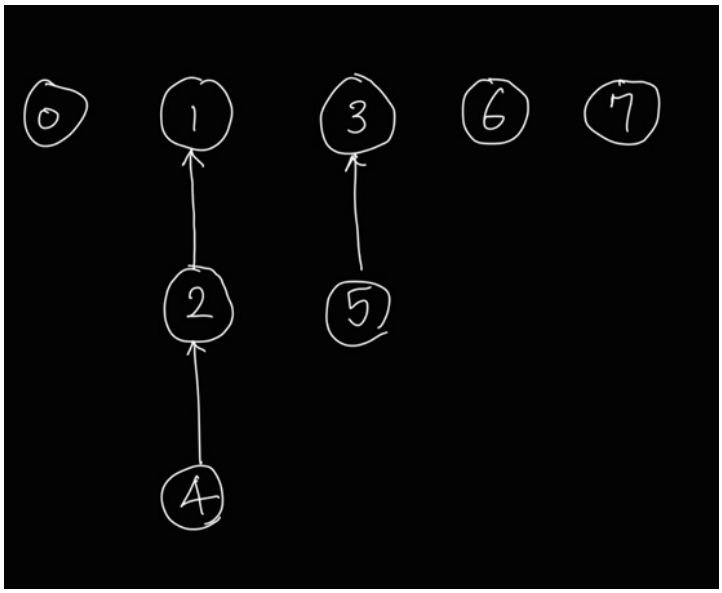
find 연산은 해당 그룹의 대표 원소를 반환하는 연산이고 (경로 압축까지 포함), union 연산은 두 그룹의 대표 원소를 연결하여 합치는 연산이다. (대표 원소 찾을때 각각 find 연산 사용)



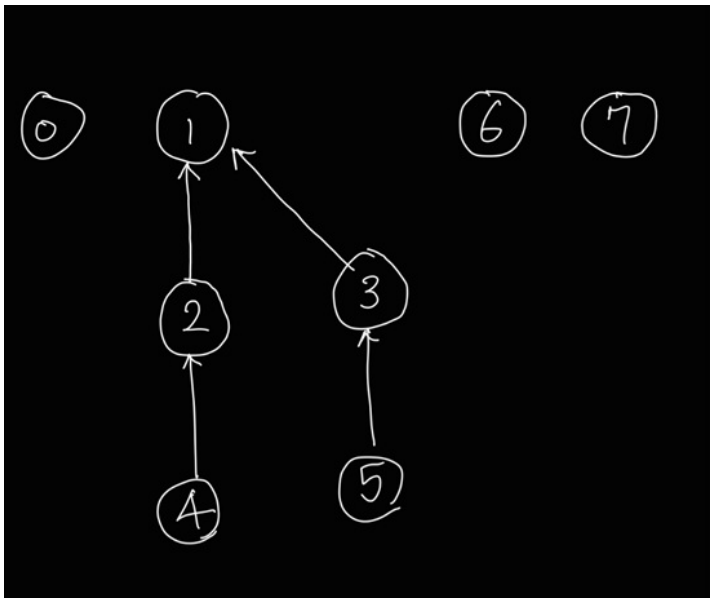
다음과 같이 7개의 점이 있다고 생각하자.



2와 4를 union하고 3과 5를 union하면 다음과 같다. (해당 집합의 가장 작은 원소가 대표 원소)

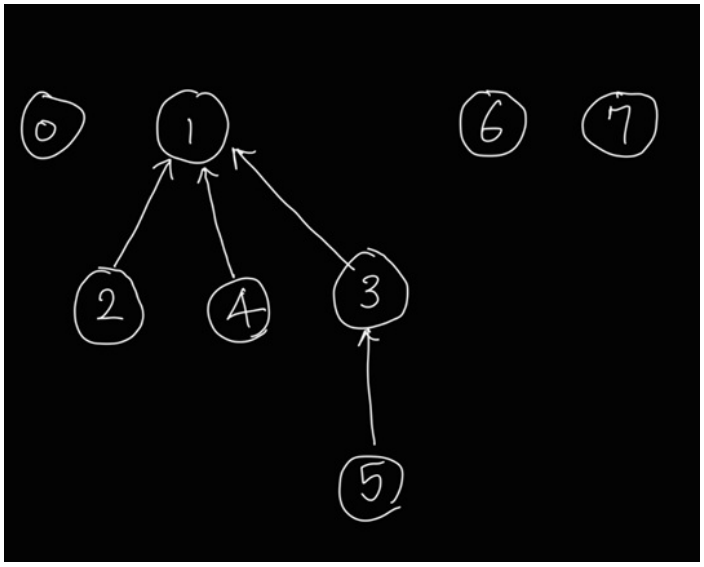


이후 1과 4를 union하면 다음과 같다.



여기서 4와 5를 union하면 이렇게 되야 할까?

$O(N^2)$ DSU에서는 그렇지만 $O(\alpha(N))$ DSU에서는 경로 압축 해주어 자신보다 위에 있는 원소들이 전부 대표 원소를 가리키게 해야 한다.



따라서 이 사진과 같게 되어야 한다.

연습 문제 (백준 1717번)

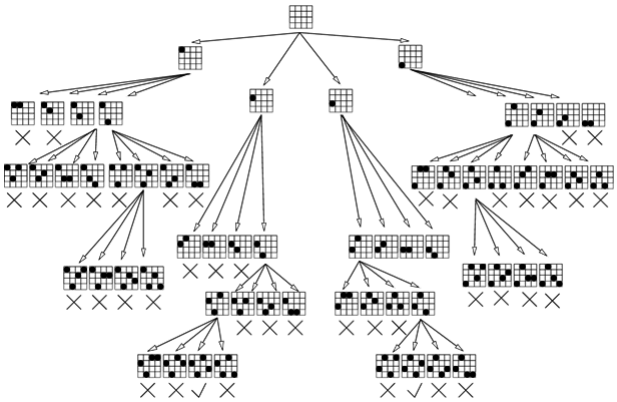
```
/** https://www.acmicpc.net/problem/1717 제출 코드 */
```

```cpp
#include<bits/stdc++.h>
using namespace std;

const int MAX = 1'000'000;

int parent[MAX]; // parent[i] : 같은 집합에 속한 부모 원소 (최상위 원소이면 자기 자신)

/**
 * x와 같은 집합에 속한 가장 작은 원소를 parent로 설정(경로 압축)하고 추출
 */
int _find(int x) {
    if(parent[x]==x) return x;
    return parent[x] = _find(parent[x]);
}

/**
 * x가 속한 집합과 y가 속한 집합을 합치는 연산
 * - x가 속한 집합의 가장 작은 원소 추출
 * - y가 속한 집합의 가장 작은 원소 추출
 * - x와 y중 작은 원소를 부모로 설정
 */
void _union(int x, int y) {
    x = _find(x);
    y = _find(y);
    if(x<y) parent[y]=x;
    else parent[x]=y;
}

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    for(int i=0;i<=n;i++) parent[i]=i; // i의 부모를 i로 설정 -> 각각이 고유의 집합이 됨

    while(m--) {
        int op, a, b; cin >> op >> a >> b;
        if(op==0) _union(a, b); // a가 속한 집합과 b가 속한 집합 합치기
        else cout << (_find(a) == _find(b) ? "YES₩n" : "NO₩n"); // a가 속한 집합의 가장 작은 원소와 b가 속
    }
}
```

1.7 Backtracking

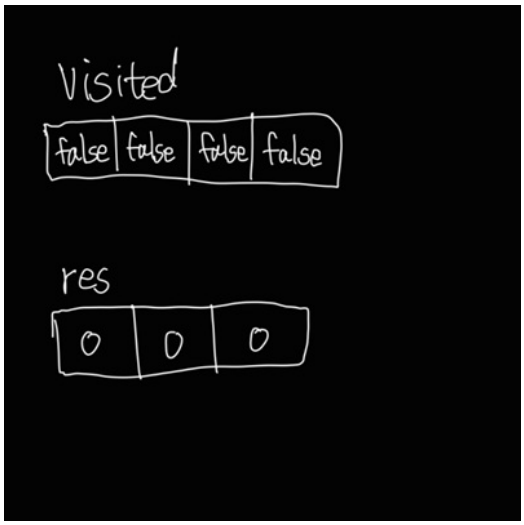
모든 경우를 탐색하되, 해답이 될 수 없는 경로는 중간에 가지치기하여 탐색을 중단하는 완전 탐색 알고리즘

시간복잡도 : 최악 $O(K^n)$ (K : 선택지 개수, n : 깊이)



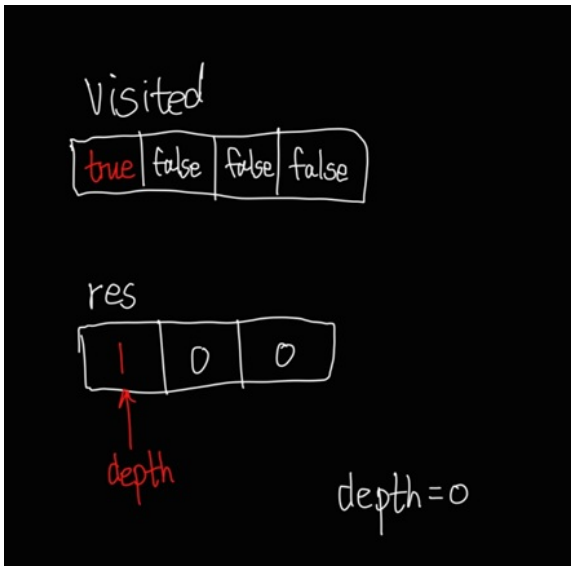
연습 문제 (백준 15649번)

위 문제는 nPm을 전부 출력하는 문제로, 깊이마다 선택지를 하나씩 채워 넣고, 사용된 숫자는 제외하는 방식으로 구현된다.



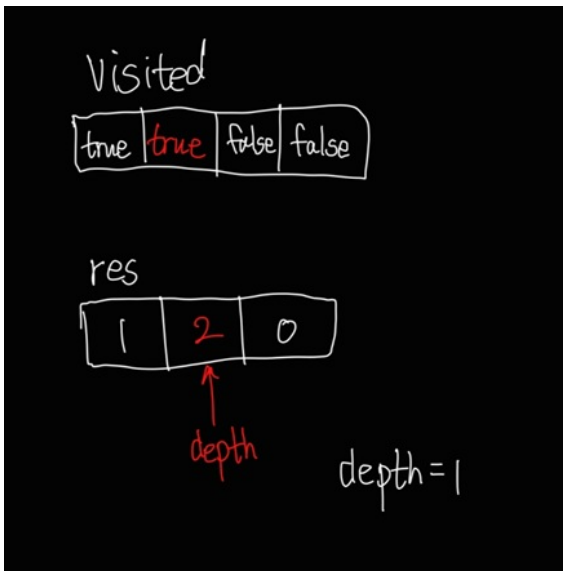
n=4, m=3인 경우를 예로 살펴보겠다.

초기에는 위와 같이 할당되어 있다.

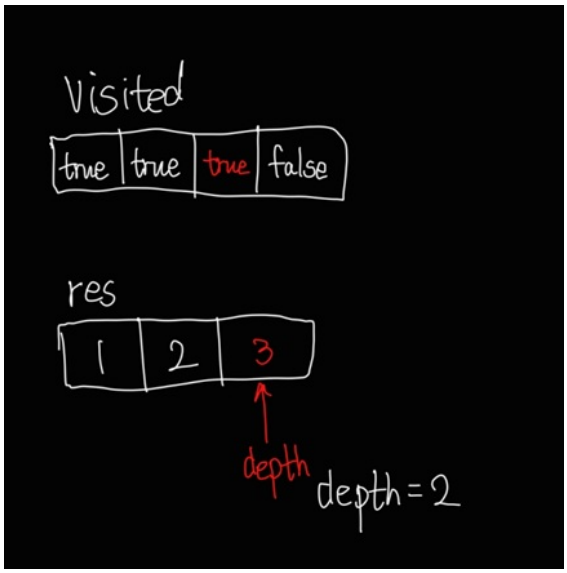


출력의 깊이 0 (첫 번째 칸)에서 1~4까지 반복문을 돌며 사용하지 않은 것을 사용한다.

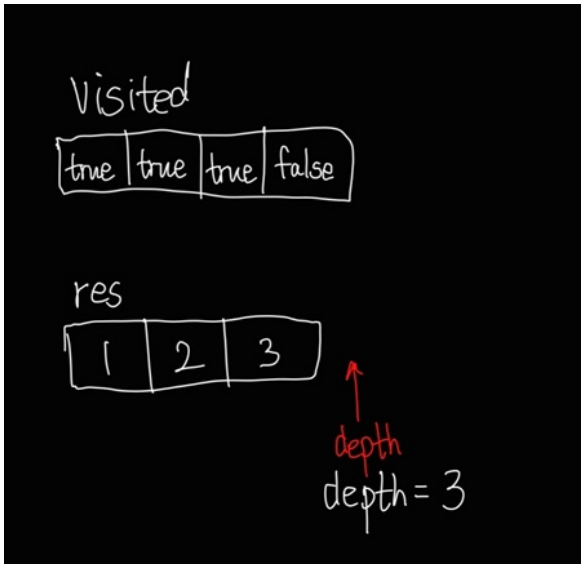
그 결과 첫 번째 칸에는 1이 할당되었다.



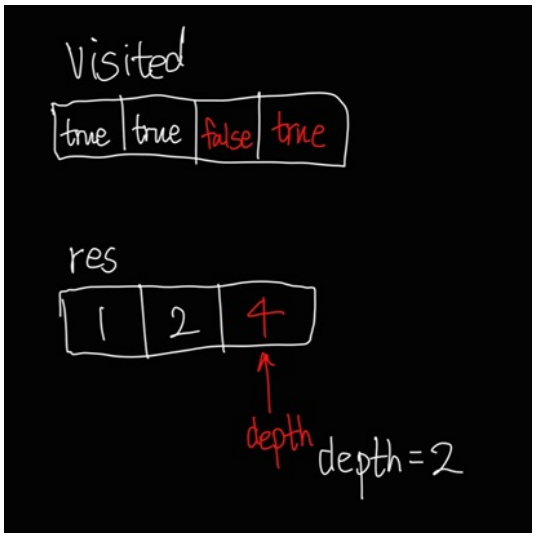
깊이 1에서는 2가 할당되어 다음과 같아진다.



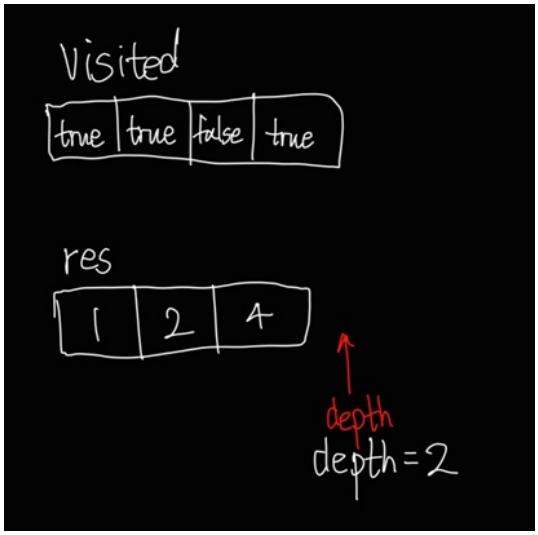
깊이 2에서는 3이 할당되어 다음과 같아진다.



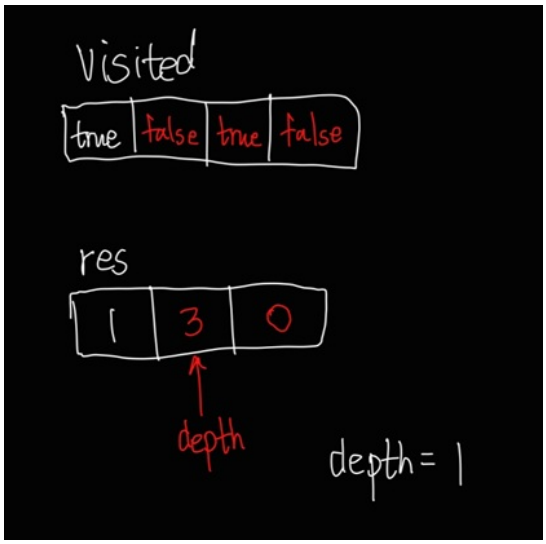
깊이 3에서는 범위를 벗어나 지금까지 기록한 '1 2 3'을 출력된다.



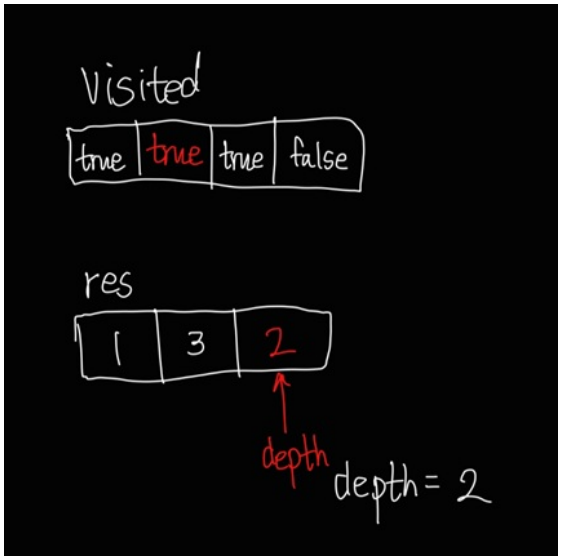
다시 깊이 2로 돌아가보니 이번에는 4가 할당된다.



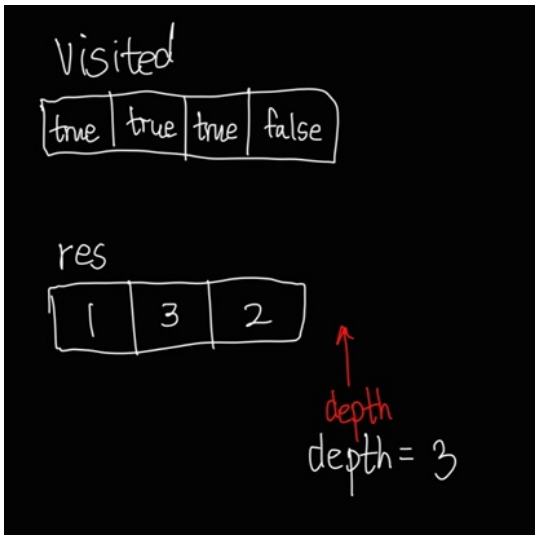
깊이 3에서는 이번에는 기록한 '1 2 4'를 출력된다.



그 다음에는 깊이 1까지 돌아가 깊이 1에 3이 할당된다.



깊이 2에서는 2가 할당된다.



깊이 3에서는 기록한 '1 3 2'가 출력된다.

이러한 과정을 통해

123
124
132
134
213
214
...
이 출력된다.

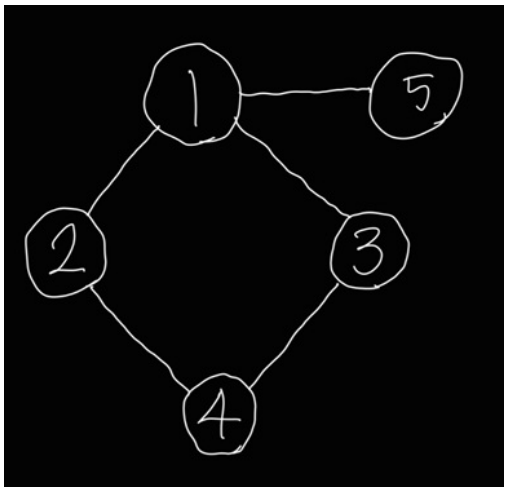
```
/** https://www.acmicpc.net/problem/15649 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

/**
 * n : 선택지의 개수
 * m : 최대 깊이
 * arr[i] : 깊이 i일때의 저장된 값
 * visited[i] : i번째 선택지가 사용되었는지 여부
 */
int n, m, arr[8];
bool visited[8];

/**
 * Backtracking
 * - 최대 깊이에 도달한 경우
 * - - 문제에서 요구하는 것을 수행하고 return으로 빠져나오기
```

```
 * - 최대 깊이에 도달하지 않은 경우
 * - - n개의 선택지 중 사용 안된 것을 사용하고 다음 깊이로 이동
 */
void back(int depth=0) {
    if(depth==m) {
        // 문제 요구사항 : 출력
        for(int i=0;i<m;i++) cout << arr[i] << ' ';
        cout << '₩n';
        return;
    }

    for(int i=0;i<n;i++) {
        if(!visited[i]) { // i번째 선택지를 사용 안한 경우
            visited[i]=true; // 사용 마킹
            arr[depth]=i+1;
            back(depth+1); // 다음 깊이로 이동
            visited[i]=false; // 사용 마킹 해제
        }
    }
}

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    cin >> n >> m;
    back();
}
```

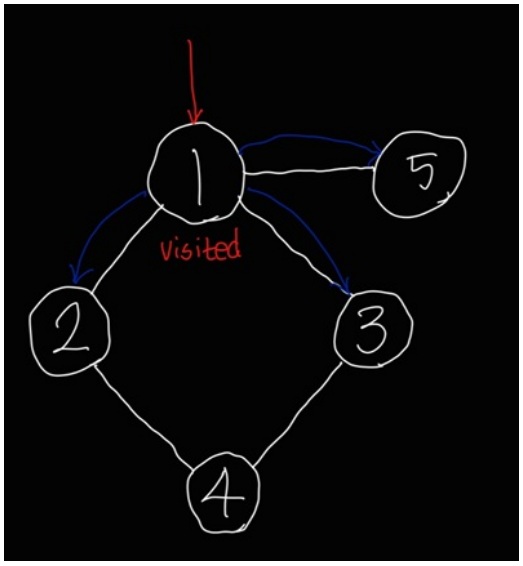
1.8 DFS (Depth First Search, 깊이 우선 탐색)

그래프에서 한 정점에서 한 경로를 끝까지 따라간 뒤 막히면 이전 분기점으로 되돌아가 다른 분기를 탐색하는 알고리즘

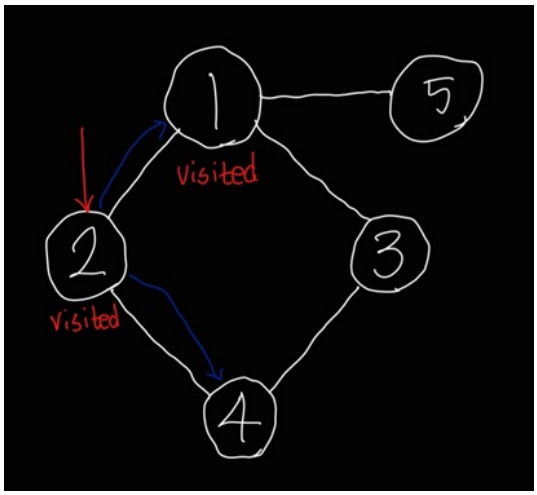
시간복잡도 : O(V+E) (V : 정점 수, E : 간선 수)



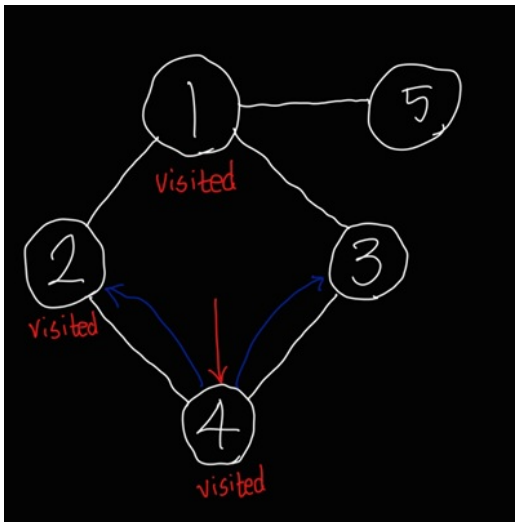
이렇게 연결된 그래프가 있고, 1번 정점에서 탐색을 시작하며, 동시에 여러 개의 정점에 도달할 수 있다면 정점 번호 오름차순으로 이동한다고 가정하자.



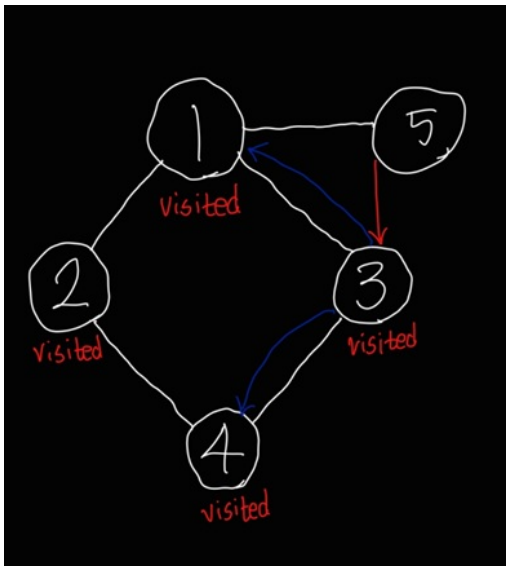
맨 처음에는 1번 정점에 이동할 것이다. 이 정점에서 이동할 수 있는 정점은 2번, 3번, 5번 정점이다.



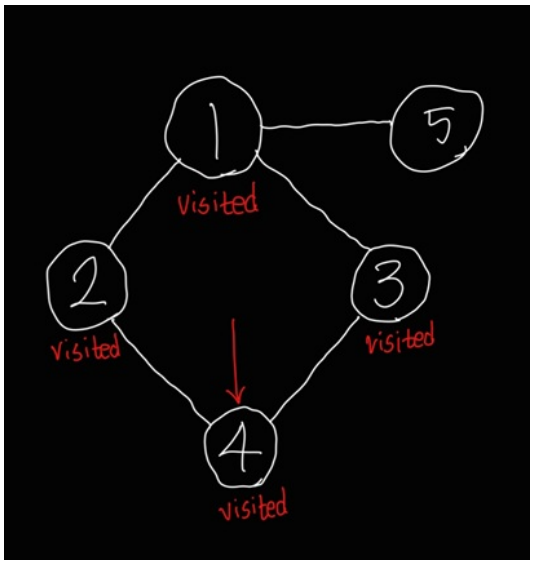
2번, 3번, 5번 정점 중 번호가 작은 2번으로 이동한다. 2번 정점에서 이동할 수 있는 정점은 1번, 4번 정점이다.



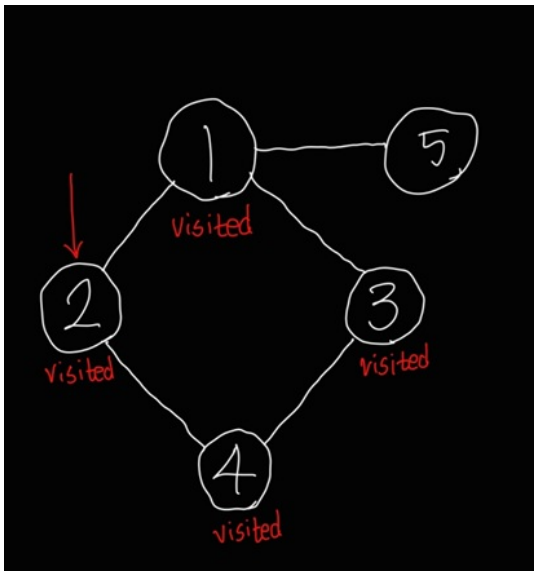
방문한 정점인 1번을 제외하고 가장 작은 4번에 이동한다. 4번 정점에서 이동할 수 있는 정점은 2번, 3번 정점이다.



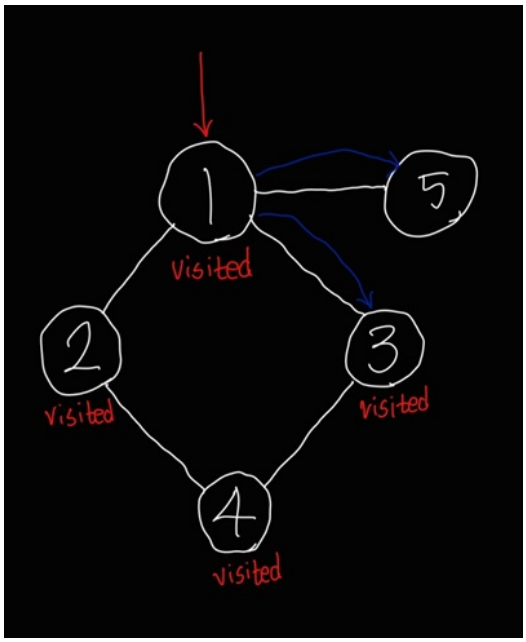
방문한 정점인 2번을 제외하고 가장 작은 3번에 이동한다. 3번 정점에서 이동할 수 있는 정점은 1번, 4번 정점이다.



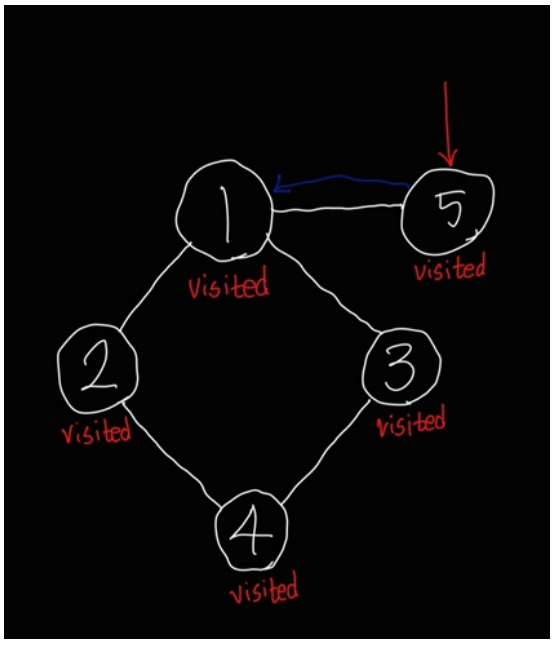
더 이상 3번 정점에서 이동할 수 있는 정점에 없어서 이전에 방문한 4번 정점으로 되돌아온다.



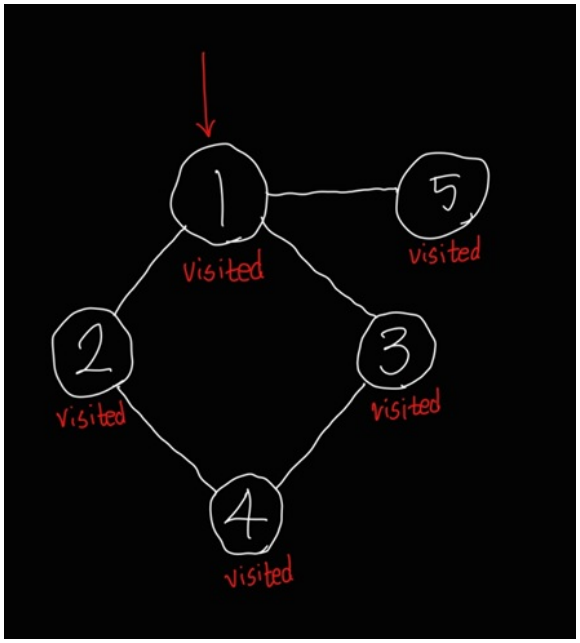
더 이상 4번 정점에서 이동할 수 있는 정점에 없어서 이전에 방문한 2번 정점으로 되돌아온다.



더 이상 2번 정점에서 이동할 수 있는 정점에 없어서 이전에 방문한 1번 정점으로 되돌아온다.



3번 정점은 이미 방문한 상태여서 이동하지 않고, 5번 정점으로 이동한다. 5번 정점에서 이동 가능한 정점은 1번 정점이다.



5번 정점이서 이동할 수 있는 정점이 없어, 1번 정점으로 돌아오고, 1번 정점에서도 이동할 수 있는 정점이 없어 종료한다.

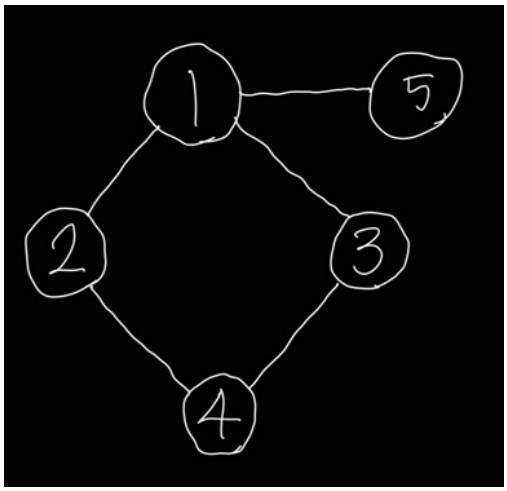
연습 문제 (백준 24479번)

```cpp
/** https://www.acmicpc.net/problem/24479 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

const int MAX = 200'001;

int cnt, visited[MAX];
vector<vector<int>> conn(MAX);

/**
 * DFS
 * 방문할 수 있는 노드가 보이면 이동.
 * 방문할 수 있는 노드가 보이지 않으면 다시 뒤로 되돌아감.
 * 미로 찾기에서 왼손의 법칙을 쓰는 것과 같음.
 */
void dfs(int cur) {
    visited[cur] = ++cnt; // 방문 순서 마킹
    for(int next:conn[cur]) {
        if(!visited[next]) { // 방문하지 않은 정점이면 방문
            dfs(next);
        }
    }
}

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n, m, r; cin >> n >> m >> r;
    while(m--) {
        int u, v; cin >> u >> v;
        conn[u].push_back(v); // u번 노드에서 v번 노드로 이동 가능
        conn[v].push_back(u); // v번 노드에서 u번 노드로 이동 가능
    }
    for(int i=1;i<=n;i++) sort(conn[i].begin(), conn[i].end()); // 문제의 조건: 방문할 수 있는 노드가 여러 개일

    dfs(r); // DFS 시작
    for(int i=1;i<=n;i++) cout << visited[i] << '₩n';
}
```

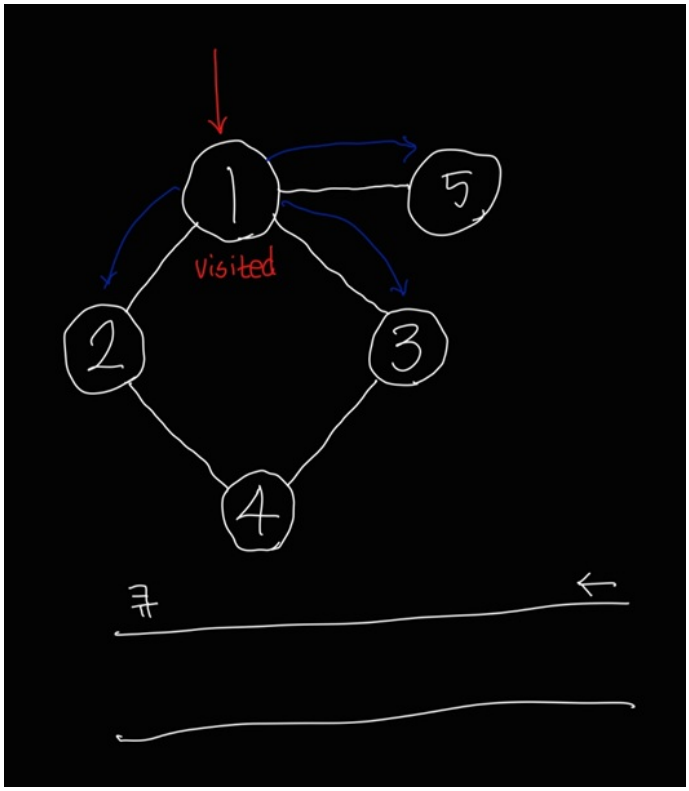
1.9 BFS (Breadth First Search, 너비 우선 탐색)

그래프에서 시작 정점에서 가까운 정점부터 차례대로 탐색하는 알고리즘

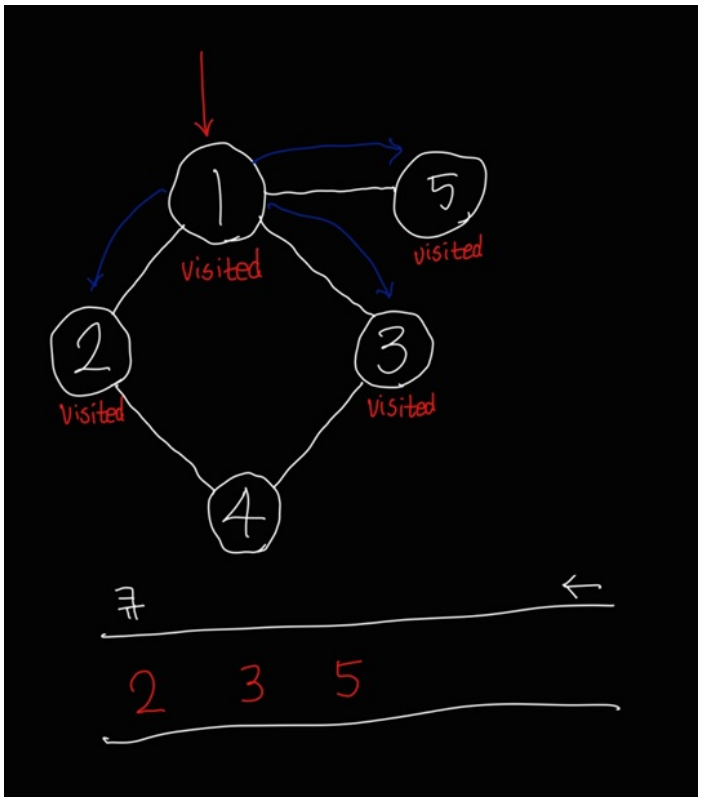
시간복잡도 : O(V+E) (V : 정점 수, E : 간선 수)



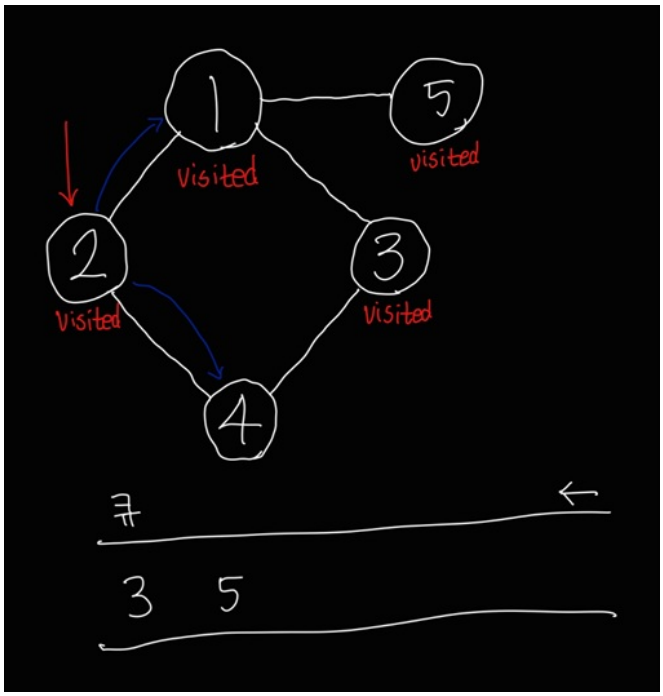
이렇게 연결된 그래프가 있고, 1번 정점에서 탐색을 시작하며, 동시에 여러 개의 정점에 도달할 수 있다면 정점 번호 오름차순으로 이동한다고 가정하자.



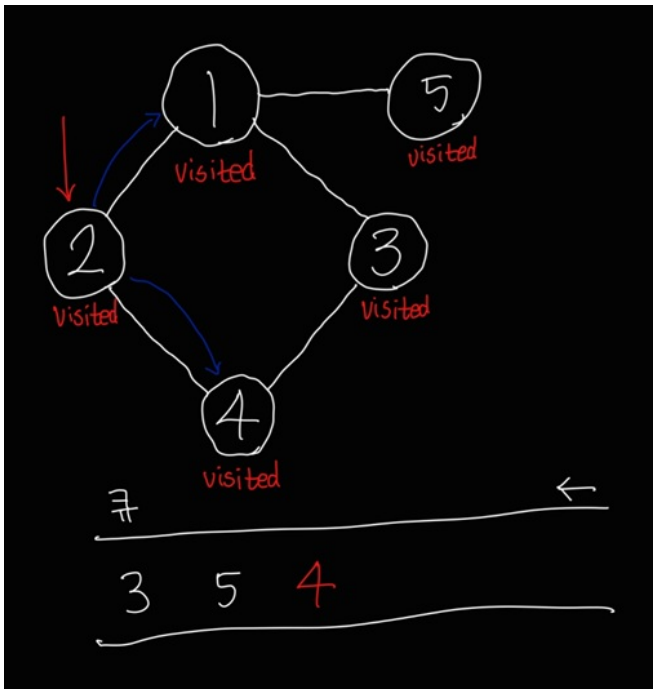
맨 처음에는 1번 정점에 이동할 것이다. 이 정점에서 이동할 수 있는 정점은 2번, 3번, 5번 정점이다.



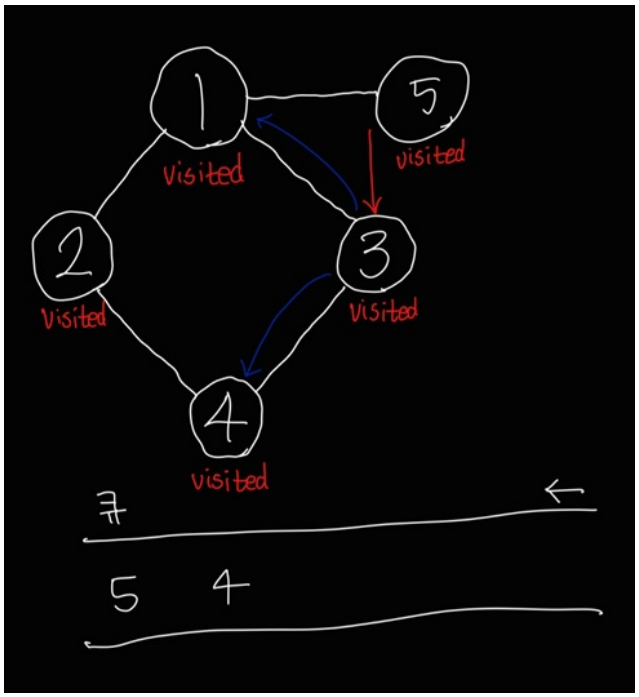
아직 방문하지 않은 2번, 3번, 5번 정점을 큐에 넣는다.



큐에서 맨 앞에 있는 2번 정점으로 이동한다. 이 정점에서 이동할 수 있는 정점은 1번, 4번 정점이다.

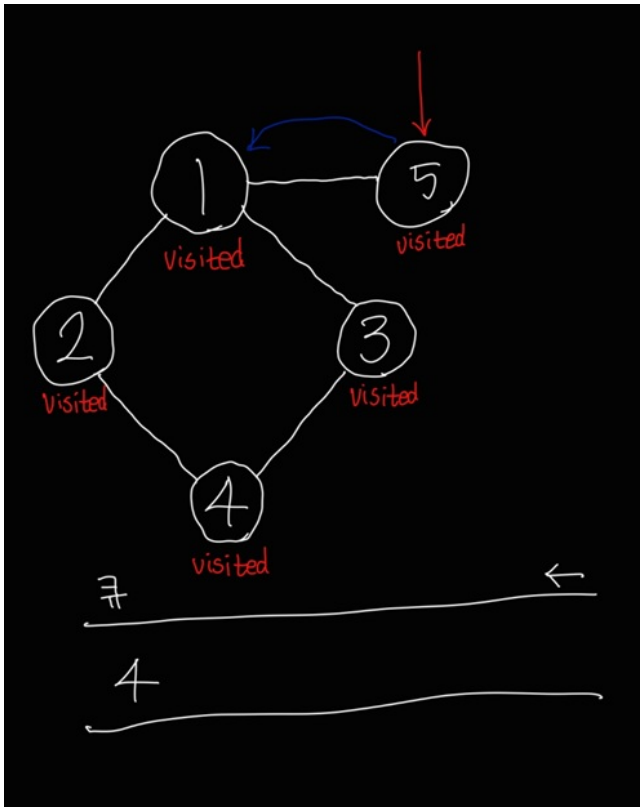


아직 방문하지 않은 4번 정점을 큐에 넣는다.



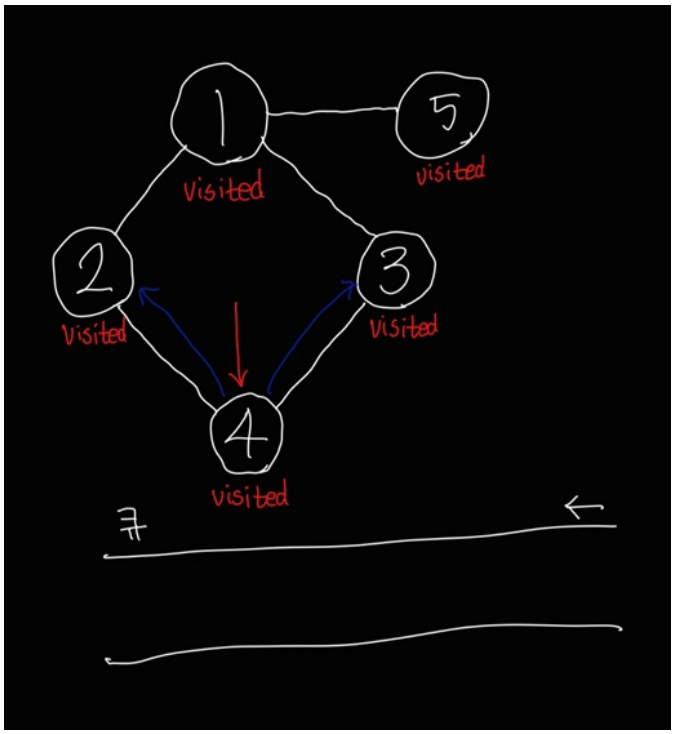
큐에서 맨 앞에 있는 3번 정점으로 이동한다. 이 정점에서 이동할 수 있는 정점은 1번, 4번 정점이다.

큐에 넣을 수 있는 정점은 없다.



큐에서 맨 앞에 있는 5번 정점으로 이동한다. 이 정점에서 이동할 수 있는 정점은 1번 정점이다.

큐에 넣을 수 있는 정점은 없다.



큐에서 맨 앞에 있는 4번 정점으로 이동한다. 이 정점에서 이동할 수 있는 정점은 2번, 3번 정점이다.

큐에 넣을 수 있는 정점은 없다. 큐가 비어 종료한다.

연습 문제 (백준 24444번)

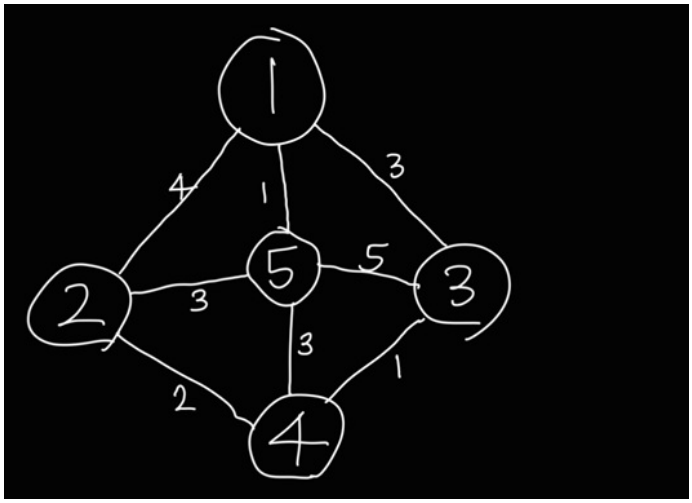
```
/** https://www.acmicpc.net/problem/24444 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

const int MAX = 200'001;

int cnt, visited[MAX];
vector<vector<int>> conn(MAX);

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n, m, r; cin >> n >> m >> r;
    while(m--) {
        int u, v; cin >> u >> v;
        conn[u].push_back(v); // u번 노드에서 v번 노드로 이동 가능
        conn[v].push_back(u); // v번 노드에서 u번 노드로 이동 가능
    }
```

```
    for(int i=1;i<=n;i++) sort(conn[i].begin(), conn[i].end()); // 문제의 조건: 방문할 수 있는 노드가 여러 개일

    /**
     * BFS
     * 가까운 위치의 정점부터 탐색하는 알고리즘
     * 방문할 수 있는 노드가 보이면 큐에 넣는다.
     */
    queue<int> q; q.push(r);
    visited[r] = ++cnt;
    while(!q.empty()) {
        int cur = q.front(); q.pop();
        for(int next: conn[cur]) {
            if(!visited[next]) { // 방문하지 않은 정점이면 방문
                visited[next] = ++cnt;
                q.push(next);
            }
        }
    }
    for(int i=1;i<=n;i++) cout << visited[i] << '₩n';
}
```

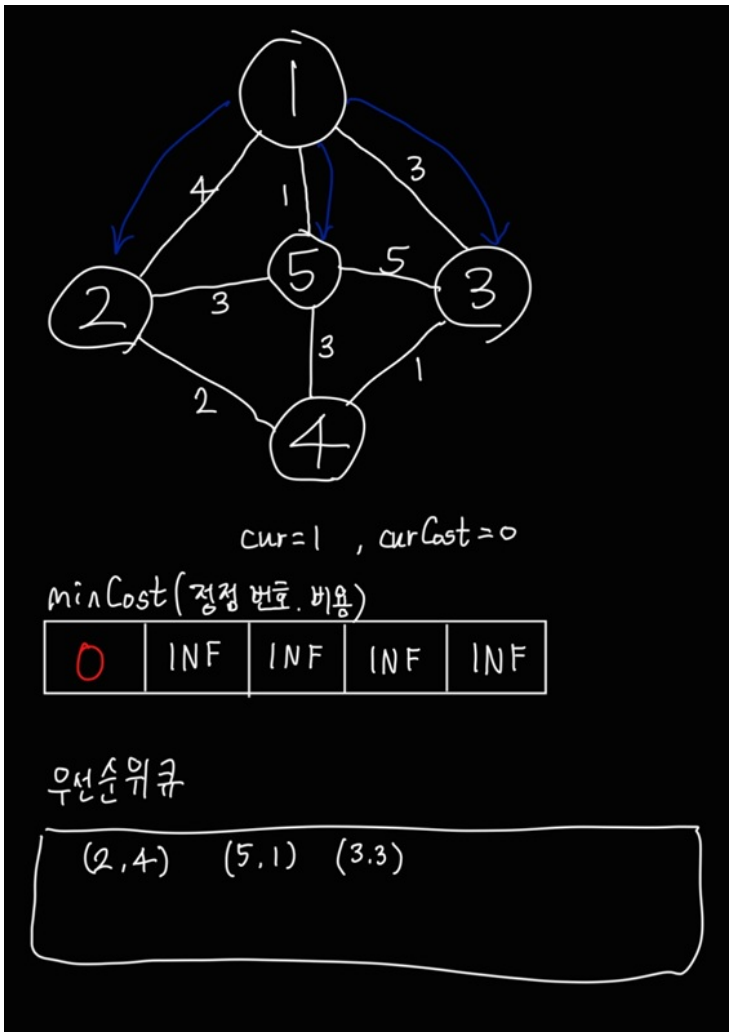
1.10 Dijkstra's Algorithm

가중치가 음수가 없는 그래프에서, 시작 정점으로부터 누적 거리가 가장 짧은 정점부터 차례대로 탐색하여 최단 거리를 구하는 알고리즘

시간복잡도 : O(ElogV)

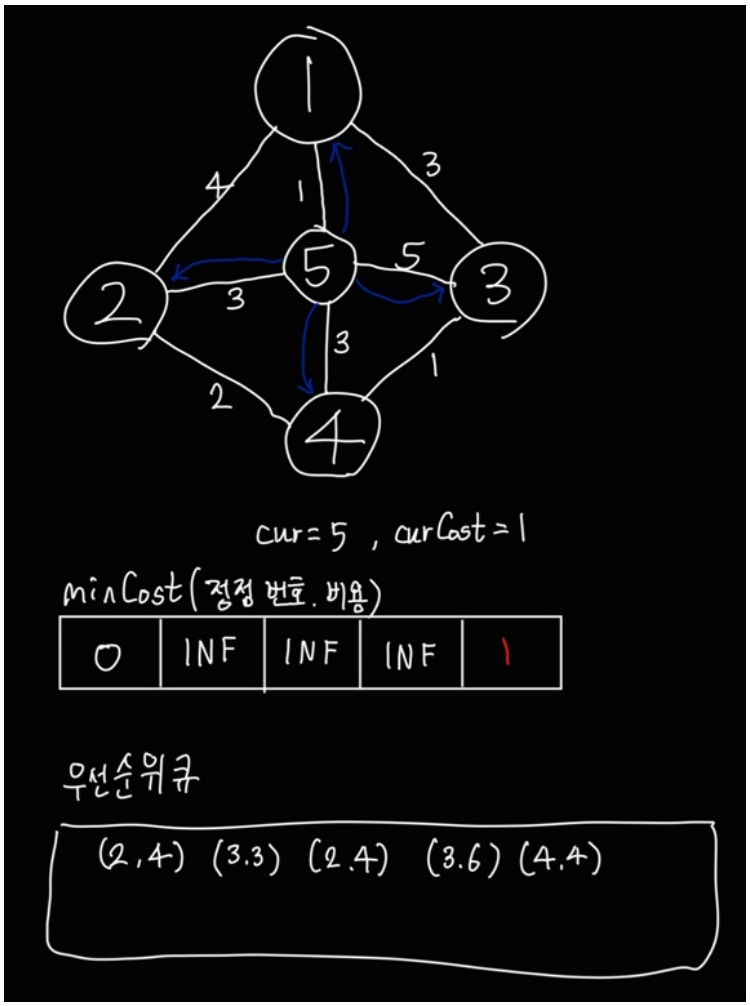


이렇게 연결된 그래프가 있고 1번 정점에서 탐색을 시작한다고 가정하자.



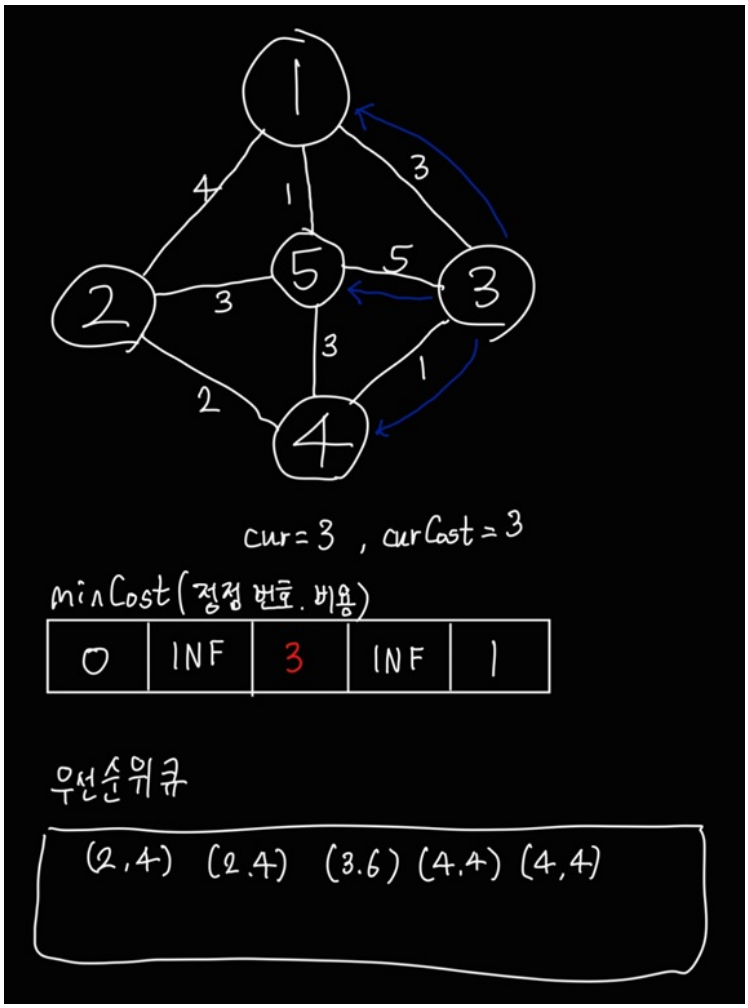
1번 정점을 탐색할 때 접근 가능한 정점은 2번, 3번, 5번 정점이다.

1번 정점에서 만들 수 있는 (정점, 비용) 쌍은 (2, 4), (5, 1), (3, 3)이고 이를 우선순위 큐에 넣는다.



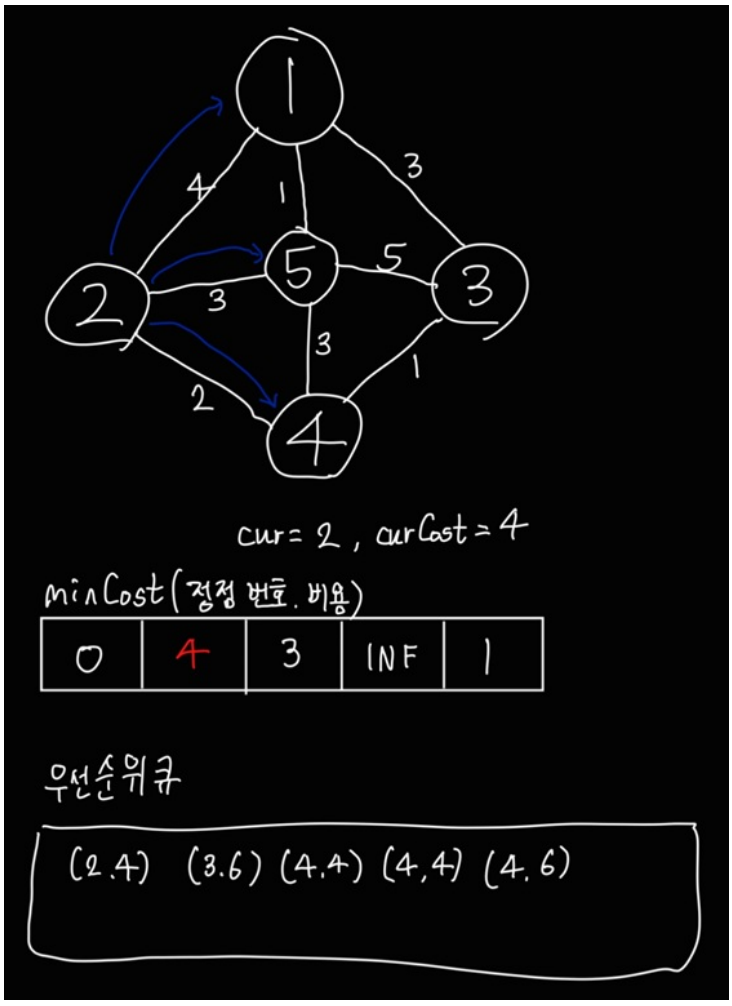
우선순위 큐에서 비용이 가장 적은 쌍인 (5, 1)를 꺼내 비용을 기록한다.

5번 정점에서 만들 수 있는 쌍은 (1, 2), (2, 4), (3, 6), (4, 4)이고, 이 중 minCost보다 비용이 더 적은 (2, 4), (3, 6), (4, 4)만 우선순위 큐에 넣는다.



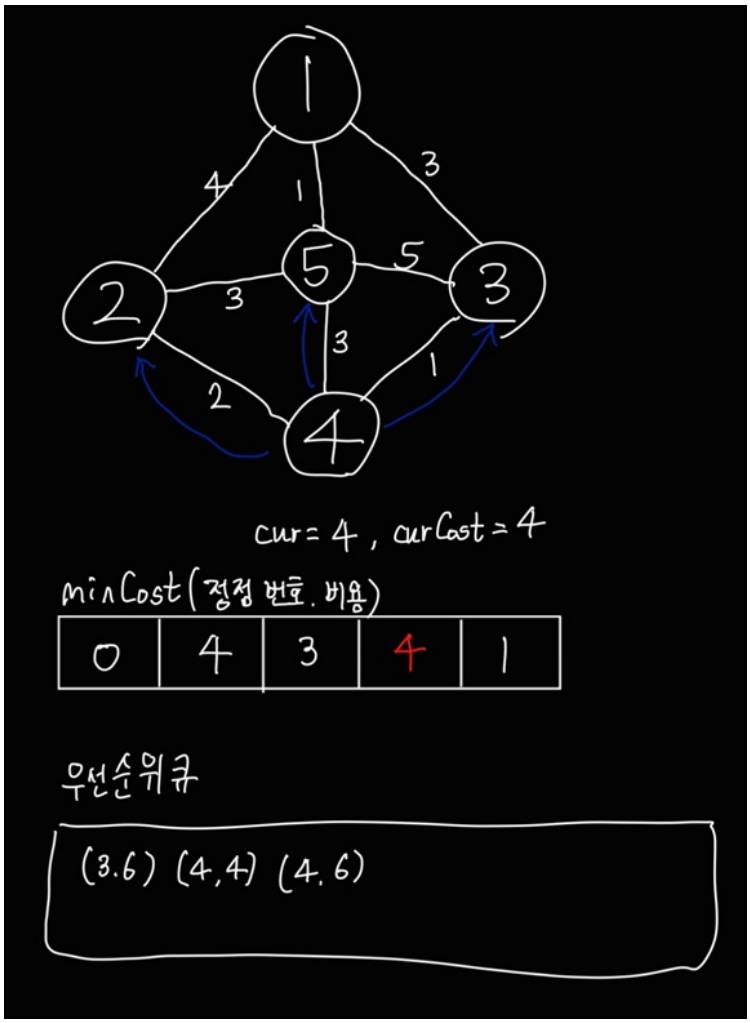
우선순위 큐에서 비용이 가장 적은 쌍인 (3, 3)를 꺼내 비용을 기록한다.

3번 정점에서 만들 수 있는 쌍은 (1, 6), (4, 4), (5, 8)이고, 이 중 minCost보다 비용이 더 적은 (4, 4)만 우선순위 큐에 넣는다.



우선순위 큐에서 비용이 가장 적은 쌍인 (2, 4)를 꺼내 비용을 기록한다.

3번 정점에서 만들 수 있는 쌍은 (1, 8), (4, 6), (5, 7)이고, 이 중 minCost보다 비용이 더 적은 (4, 6)만 우선순위 큐에 넣는다.



우선순위 큐에서 비용이 가장 적은 쌍인 (2, 4)를 꺼내고 비용이 커서 패스, (4, 4)를 꺼내 비용을 기록한다.

4번 정점에서 만들 수 있는 쌍은 (2, 6), (3, 5), (5, 7)이고, 이 중 minCost보다 비용이 더 적은 쌍이 없어 우선순위 큐에 넣지 않는다.

이후 minCost보다 비용보다 더 적은 비용 쌍이 없어 이대로 종료된다.

연습 문제 (백준 1753번)

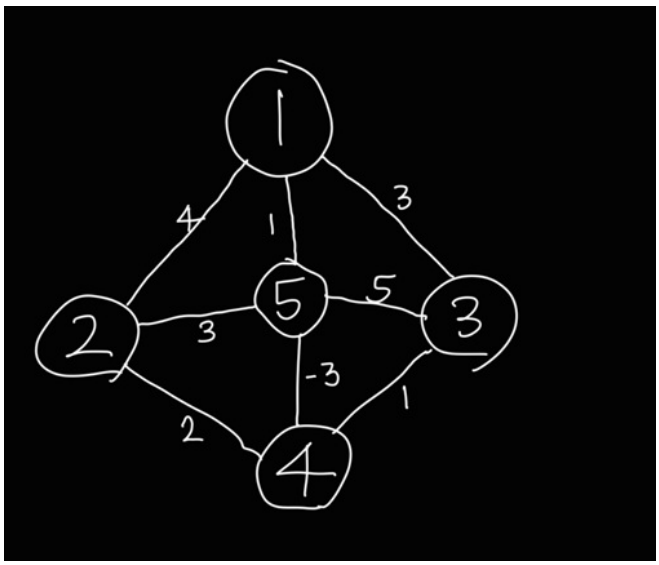
```
/** https://www.acmicpc.net/problem/1753 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

const int INF = 0x3f3f3f3f;
```

```cpp
const int MAX = 20001;

struct element {
    int pos, cost;
    bool operator<(const element e) const {
        return cost > e.cost; // 우선순위 큐라 일반 정렬과 부호 반대
    }
};

int minCost[MAX]; // 최단 거리를 저장하는 배열
vector<vector<element>> conn(MAX);

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int v, e, k; cin >> v >> e >> k;
    while(e--) {
        int u, v, w; cin >> u >> v >> w;
        conn[u].push_back({v, w}); // 단방향 간선
    }

    /**
     * Dijkstra Algorithm
     * 우선순위 큐에서 비용이 작은 경로부터 꺼내 탐색함
     */
    fill(minCost, minCost+MAX, INF); // 기본값 초기화
    priority_queue<element> pq; pq.push({k, 0}); // 시작 정점 넣기
    while(!pq.empty()) {
        auto cur = pq.top(); pq.pop();
        if(minCost[cur.pos]<=cur.cost) continue; // 현재 경로가 비용이 더 크다면 제외
        minCost[cur.pos] = cur.cost;

        for(auto next:conn[cur.pos]) {
            int nextCost = cur.cost + next.cost;
            if(nextCost<minCost[next.pos]) { // 현재 경로로 다음 정점에 도달하는 비용이 더 적다면
                pq.push({next.pos, nextCost});
            }
        }
    }

    for(int i=1;i<=v;i++) {
        if(minCost[i]==INF) cout << "INF₩n";
        else cout << minCost[i] << '₩n';
    }
}
```

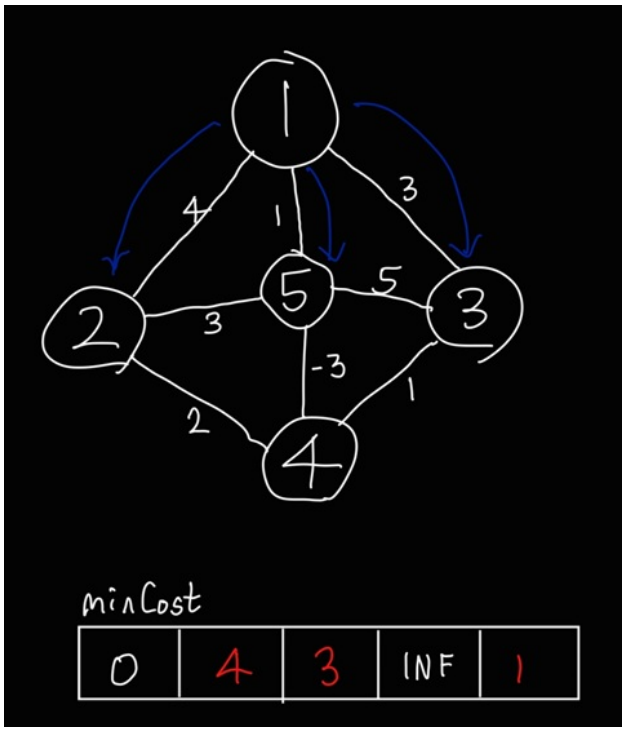
1.11 Bellman-Ford Algorithm

가중치가 음수인 그래프에서도, 모든 간선을 최대 V-1번 완화해 최단 거리를 구하고 음수 사이클 여부를 판별하는 알고리즘

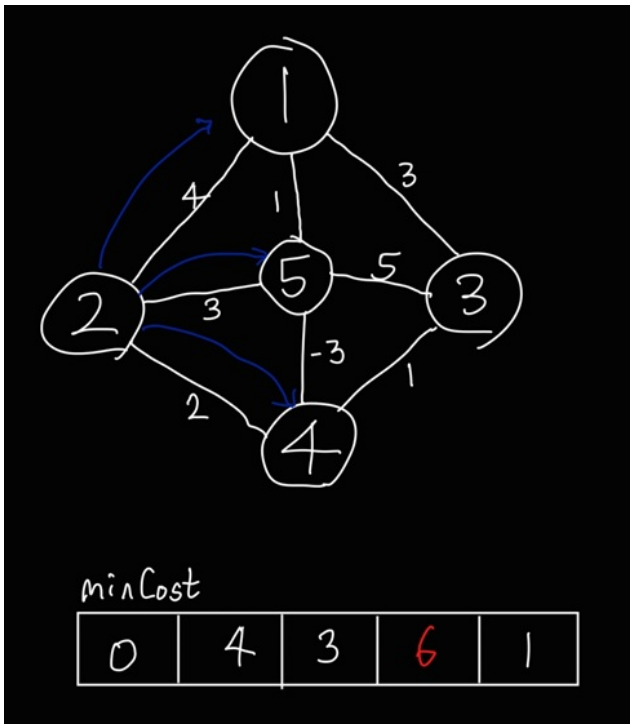
시간복잡도 : O(VE)



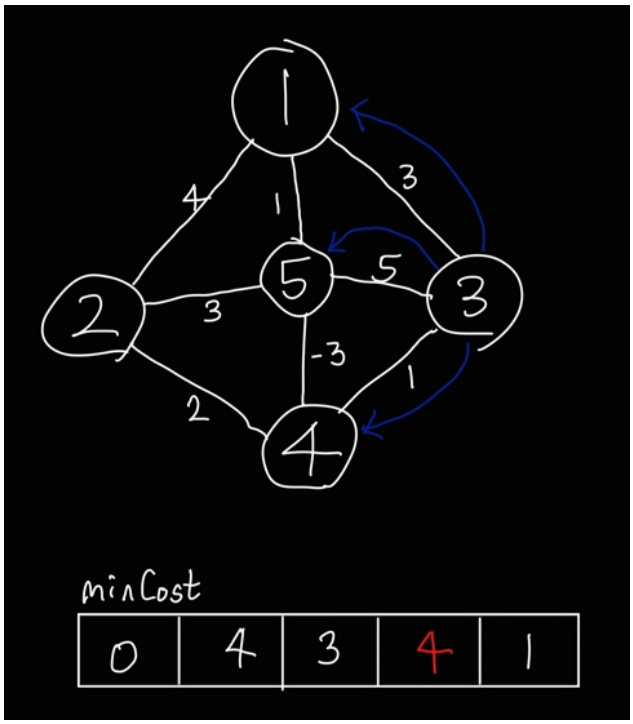
이렇게 연결된 그래프가 있고 1번 정점에서 탐색을 시작한다고 가정하자.



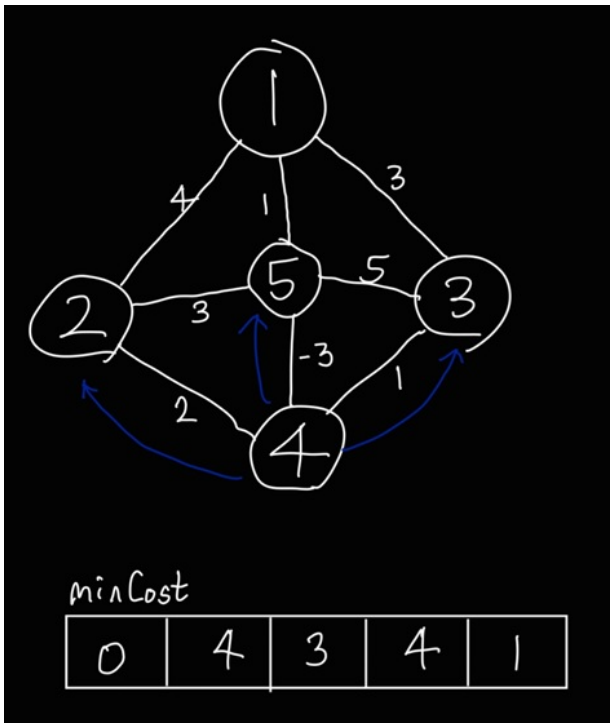
1번 정점에서 연결된 다른 모든 정점에 대해 최소비용 업데이트를 한다.



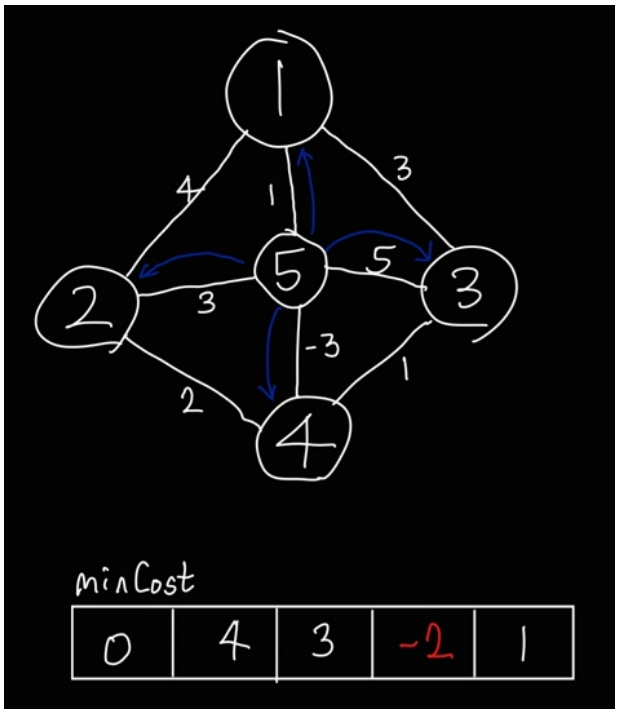
2번 정점에서 연결된 다른 모든 정점에 대해 최소비용 업데이트를 한다.



3번 정점에서 연결된 다른 모든 정점에 대해 최소비용 업데이트를 한다.

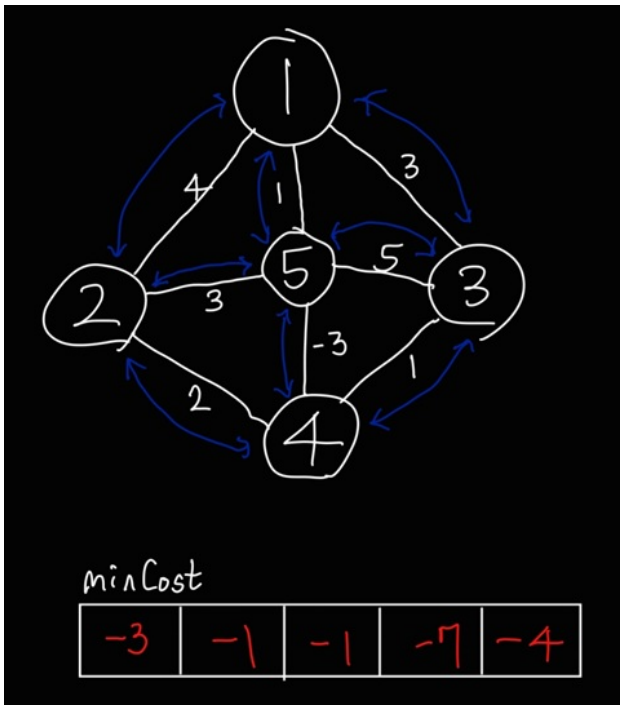


4번 정점에서 연결된 다른 모든 정점에 대해 최소비용 업데이트를 한다.

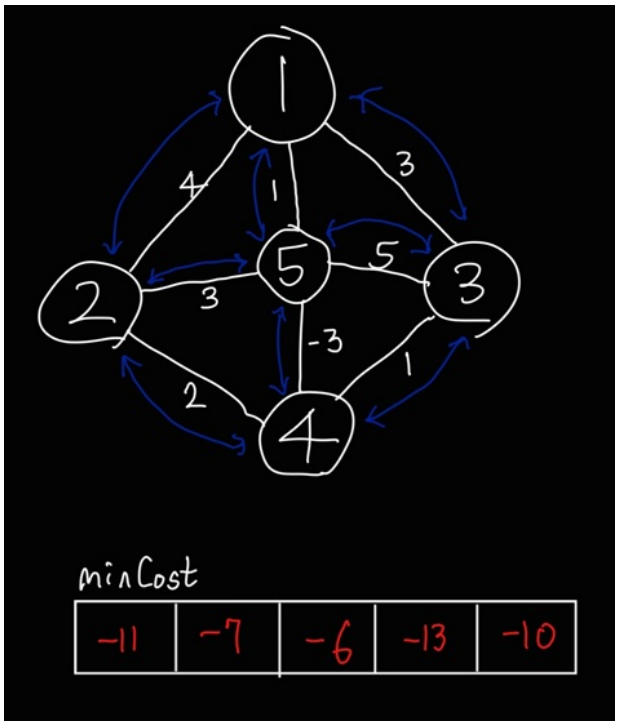


5번 정점에서 연결된 다른 모든 정점에 대해 최소비용 업데이트를 한다.

첫번째 반복이 끝났다. 이번 반복에서 최단거리 업데이트가 이루어졌으니 다음 반복을 시작한다. (최단거리 업데이트가 한번이라도 발생하지 않으면 종료)



두번째 반복의 결과는 다음과 같다. 이번 반복에서 최단거리 업데이트가 이루어졌으니 다음 반복을 시작한다.



세번째 반복의 결과는 다음과 같다. 이번 반복에서 최단거리 업데이트가 이루어졌으니 다음 반복을 시작한다.

...

이렇게 쭉 반복하다 n(정점 수)번째 반복에서도 반복이 이루어지면 이 그래프에는 음수 사이클이 존재한다고 판단하고 종료한다.

그 이유는 한번의 반복에서 아주 운이 없어서 업데이트가 적게 이루어진다 하더라도 최소 1개의 정점은 업데이트 된다. (0개면 종료)

처음 거리가 0인 기준점을 제외하면 업데이트 시킬 정점이 n-1개이니, n번째 반복에는 정상적인 경우라면 업데이트가 이루어질 수 없다.

연습 문제 (백준 11657번)

```
/** https://www.acmicpc.net/problem/11657 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

/**
 * INF : 대략 10억
 * MAX : 최대 정점 수
 */
const int INF = 0x3f3f3f3f;
```

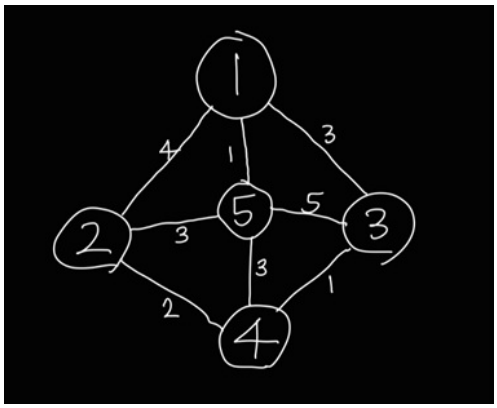
```cpp
const int MAX = 501;

struct element {
    int pos, cost;
};

int n, m;
long long minCost[MAX];
vector<vector<element>> conn(MAX);

/**
 * Bellman-Ford Algorithm
 * - 음수 사이클이 존재하는 경우 return false
 */
bool bellman_ford() {
    fill(minCost, minCost+MAX, INF);
    minCost[1]=0;
    for(int i=0;i<n;i++) {
        bool change=false;
        for(int cur=1;cur<=n;cur++) { // 모든 정점에 대해 모든 간선을 살펴보기
            if(minCost[cur]==INF) continue; // 도달 못한 정점은 패스
            for(auto next:conn[cur]) {
                long long nextCost = minCost[cur] + next.cost;
                if(nextCost<minCost[next.pos]) {
                    minCost[next.pos] = nextCost;
                    change=true;
                }
            }
        }
        if(!change) return true;
    }
    return false; // n-1번의 업데이트를 진행하고도 변화가 있으면 사이클 존재
}

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    cin >> n >> m;
    while(m--) {
        int a, b, c; cin >> a >> b >> c;
        conn[a].push_back({b, c});
    }

    if(!bellman_ford()) { // 사이클이 존재하는 경우
        cout << -1;
        return 0;
    }
```

```
    for(int i=2;i<=n;i++) {
        if(minCost[i]==INF) cout << "-1₩n";
        else cout << minCost[i] << '₩n';
    }
}
```

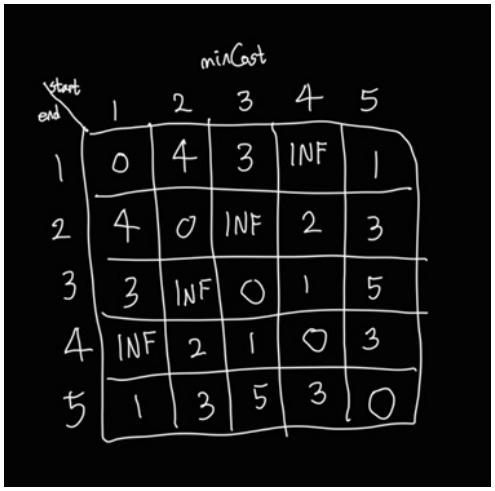
1.12 Floyd-Warshall Algorithm

모든 정점 쌍 사이의 최단 거리를 DP로 구하는 알고리즘

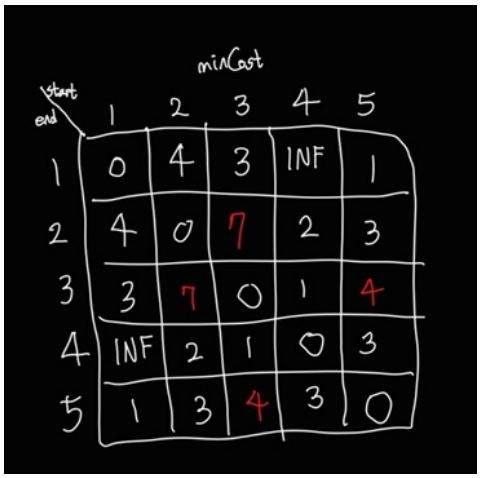
시간복잡도 : $O(V^3)$, 공간복잡도 : $O(V^2)$



위와 같은 그래프를 사용해보겠다.

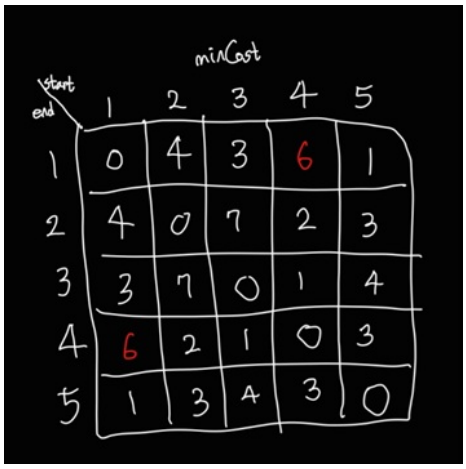


그래프의 간선을 입력해 인접 행렬을 만드면 다음과 같다. i번 행, j번 열의 원소는 i번 정점에서 j번 정점에 도달하기 위한 최소 거리이다.



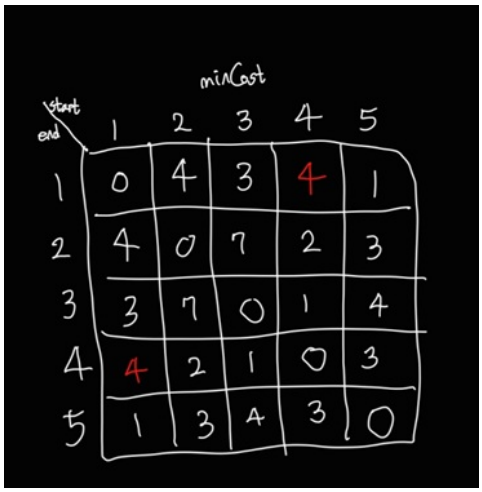
이후 mid(경유지)를 1로 하는 모든 start와 end 쌍에 대해 최단 거리를 구한다.

2 -> 1 -> 3, 3 -> 1 -> 2, 3 -> 1 -> 5, 5 -> 1 -> 3 경로에 대해 최단 거리가 단축된다.



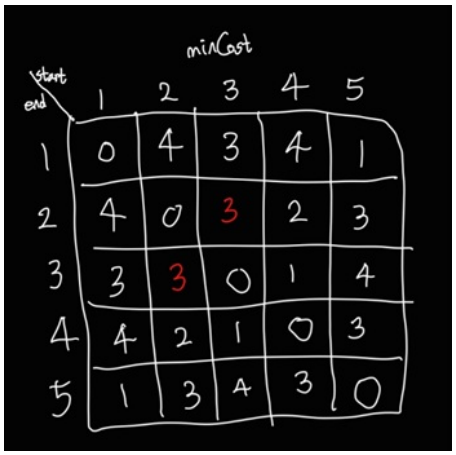
mid를 2로 하는 모든 start와 end 쌍에 대해 최단 거리를 구한다.

1 -> 2 -> 4, 4 -> 2 -> 1 경로에 대해 최단 거리가 단축된다.



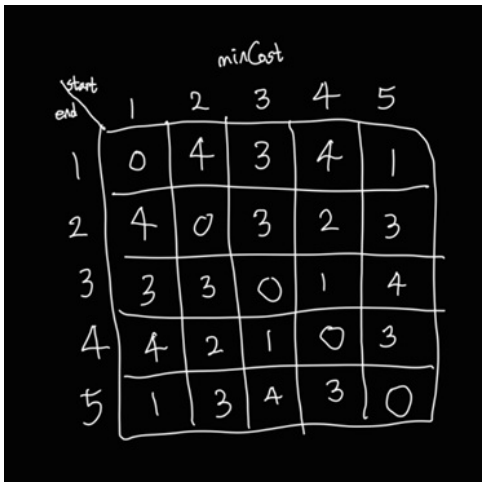
mid를 3으로 하는 모든 start와 end 쌍에 대해 최단 거리를 구한다.

1 -> 3 -> 4, 4 -> 3 -> 1 경로에 대해 최단 거리가 단축된다.



mid를 4로 하는 모든 start와 end 쌍에 대해 최단 거리를 구한다.

2 -> 4 -> 3, 3 -> 4 -> 2 경로에 대해 최단 거리가 단축된다.



mid를 5로 하는 모든 start와 end 쌍에 대해 최단 거리를 구한다.

mid=5에 대해서는 단축될 최단 거리가 없다.

연습 문제 (백준 11404번)

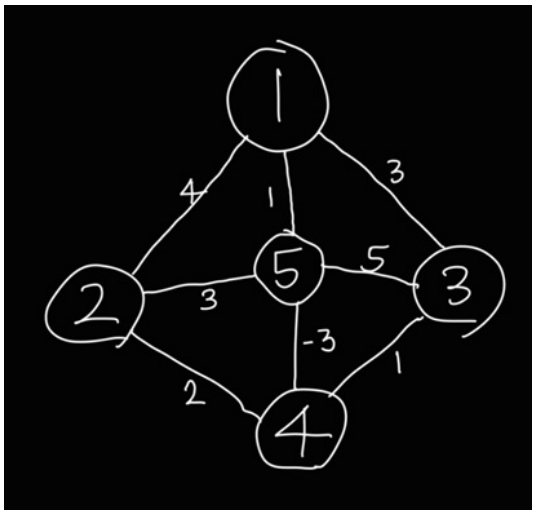
```
/** https://www.acmicpc.net/problem/11404 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

/**
 * INF : 대략 10억
 * MAX : 최대 정점 수
 * minCost[i][j] : i에서 j로 이동할 때의 최소 비용
 */
const int INF = 0x3f3f3f3f;
const int MAX = 101;

int minCost[MAX][MAX];

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;

    fill(&minCost[0][0], &minCost[MAX-1][MAX], INF);
    for(int i=1;i<=n;i++) minCost[i][i]=0;
    while(m--) {
        int a, b, c; cin >> a >> b >> c;
        minCost[a][b] = min(minCost[a][b], c);
    }
```

```
    /**
     * Floyd-Warshall Algorithm
     * 모든 정점 쌍 최단거리 계산
     * - mid : 경유지
     * - start : 출발지
     * - end : 도착지
     *
     * mid를 하나씩 늘려가며 start → end 경로와 start → mid → end 경로 중 더 짧은 경로를 선택
     */
    for(int mid=1;mid<=n;mid++) {
        for(int start=1;start<=n;start++) {
            for(int end=1;end<=n;end++) {
                minCost[start][end] = min(minCost[start][end], minCost[start][mid]+minCost[mid][end]);
            }
        }
    }

    for(int i=1;i<=n;i++) {
        for(int j=1;j<=n;j++) {
            if(minCost[i][j]==INF) cout << "0 ";
            else cout << minCost[i][j] << ' ';
        }
        cout << '₩n';
    }
}
```

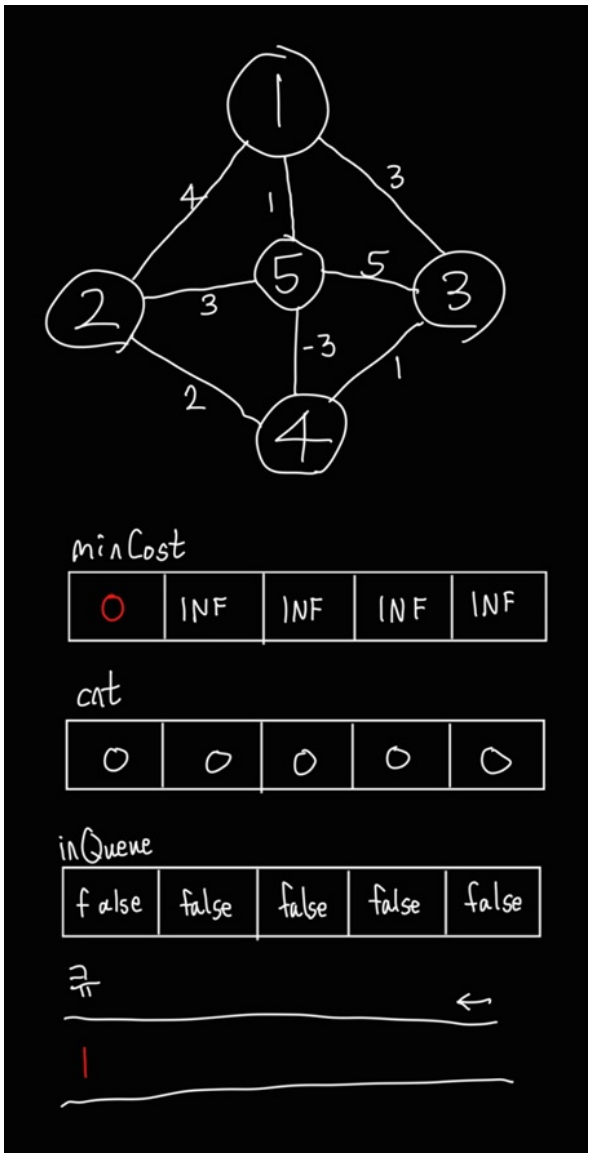
1.13 SPFA (Shortest Path Faster Algorithm)

큐를 사용해 Bellman-Ford의 간선 완화를 휴리스틱으로 가속하여 최단 거리를 구하는 알고리즘

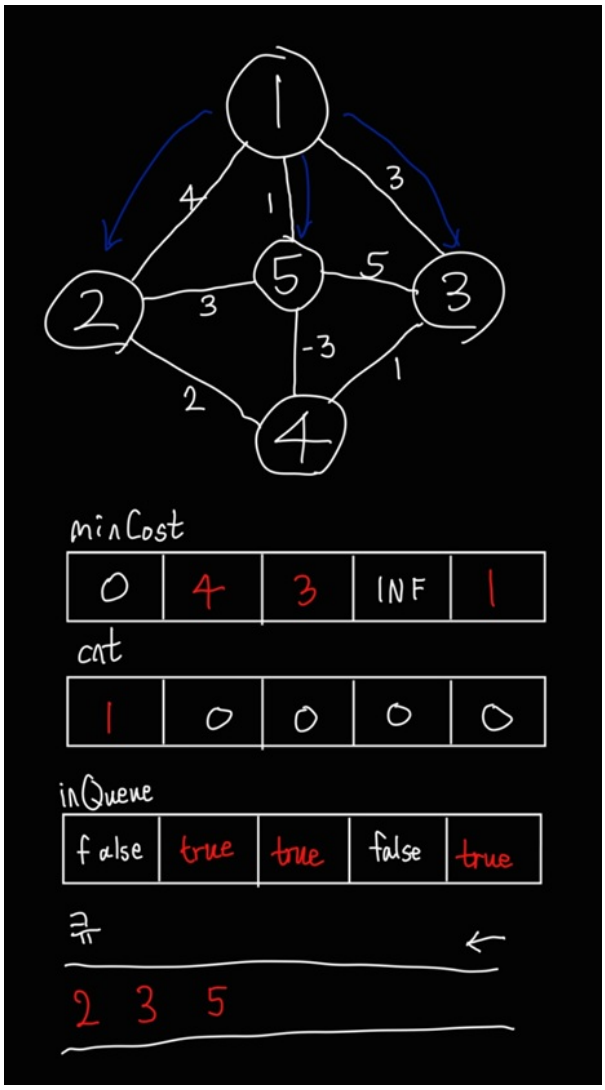
시간복잡도 : 경험적 평균 O(V+E), 최악 O(VE) (V : 정점 수, E : 간선 수)



벨만-포드와 같은 그래프를 사용하고 1번 정점에서 탐색을 시작한다고 가정하자.

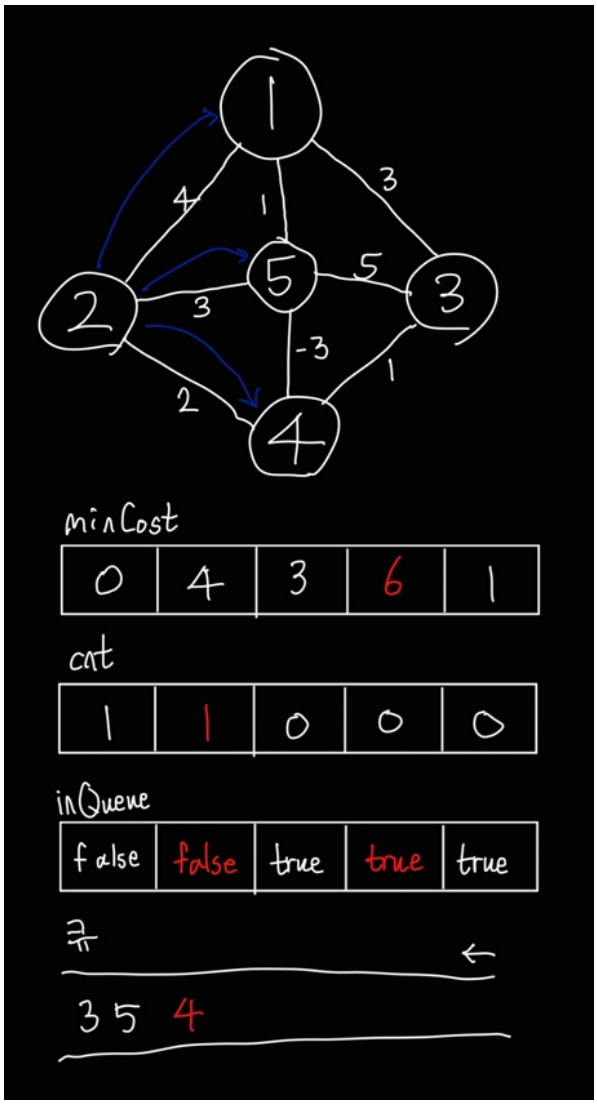


반복을 시작하기 전에 큐에 1이 들어간다.



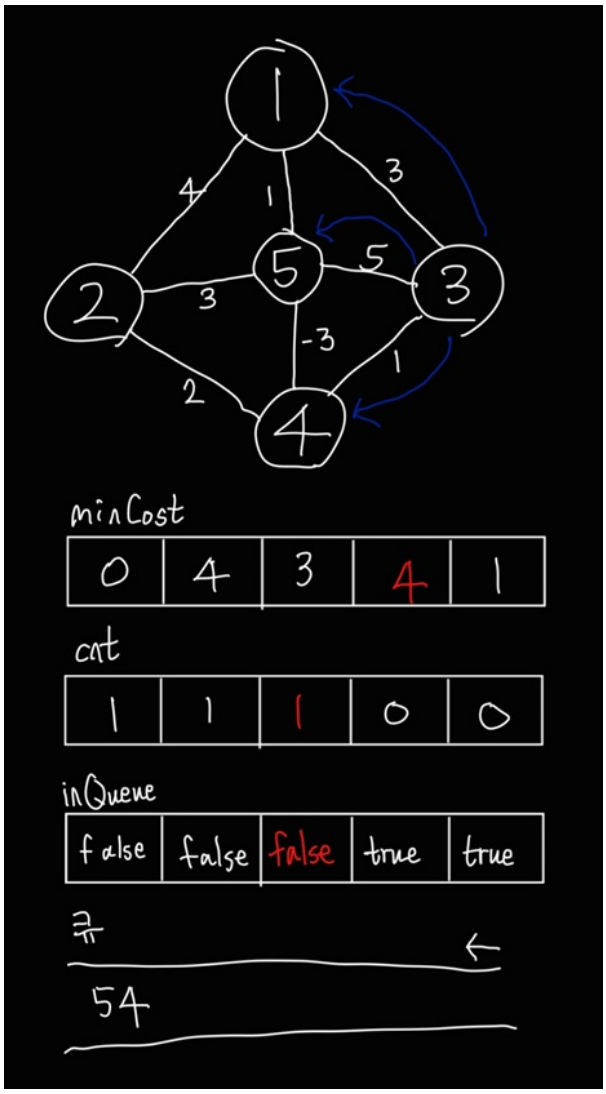
큐의 제일 앞 원소인 1을 꺼낸다. 정점 1번에서 접근 가능한 정점은 2번, 3번, 5번 정점이다.

2번 정점을 비용 4로, 3번 정점을 비용 3으로, 5번 정점을 비용 1로 업데이트 하고, 큐에 들어있지 않으면(inQueue[next]==false) 큐에 넣는다.



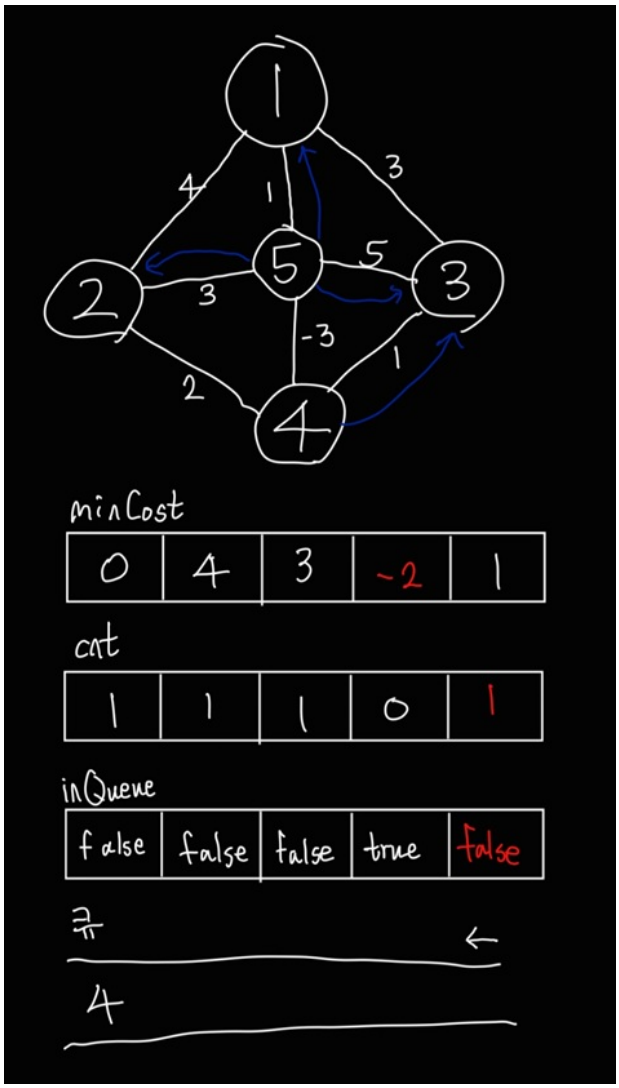
큐의 제일 앞 원소인 2를 꺼낸다. 정점 2번에서 접근 가능한 정점은 1번, 4번, 5번 정점이다.

이 중 거리가 단축되는 4번 정점을 비용 6으로 업데이트 하고, 큐에 들어있지 않으면 큐에 넣는다.



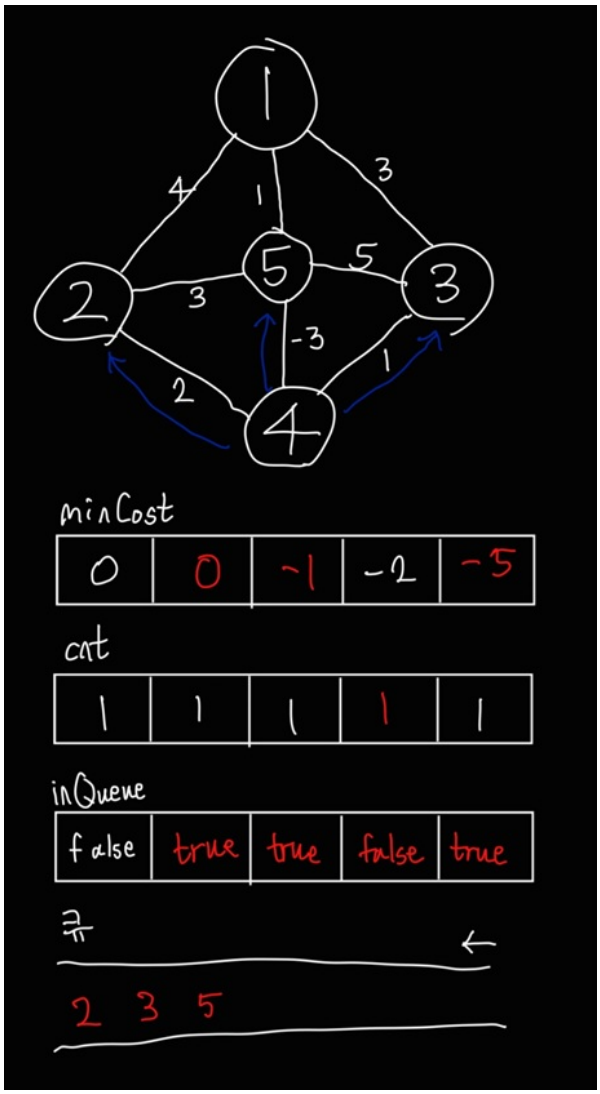
큐의 제일 앞 원소인 3를 꺼낸다. 정점 3번에서 접근 가능한 정점은 1번, 4번, 5번 정점이다.

이 중 거리가 단축되는 4번 정점을 비용 4으로 업데이트 하고, 큐에 들어있지 않으면 큐에 넣는다.



큐의 제일 앞 원소인 5를 꺼낸다. 정점 5번에서 접근 가능한 정점은 1번, 2번, 3번, 4번 정점이다.

이 중 거리가 단축되는 4번 정점을 비용 -3으로 업데이트 하고, 큐에 들어있지 않으면 큐에 넣는다.



큐의 제일 앞 원소인 4를 꺼낸다. 정점 4번에서 접근 가능한 정점은 2번, 3번, 5번 정점이다.

이 중 거리가 단축되는 2번 정점을 비용 0으로, 3번 정점을 비용 -1로, 5번 정점을 비용 -5로 업데이트 하고, 큐에 들어있지 않으면 큐에 넣는다.

이렇게 쭉 진행하다 큐가 비어 있으면 정상적인 종료로, 벨만-포드와 같은 이유로 한 정점을 n번 업데이트할 때는(cnt==n) 음수 사이클이 있다 판단하고 종료한다.

연습 문제 (백준 11657번)

/** https://www.acmicpc.net/problem/11657 제출 코드 */

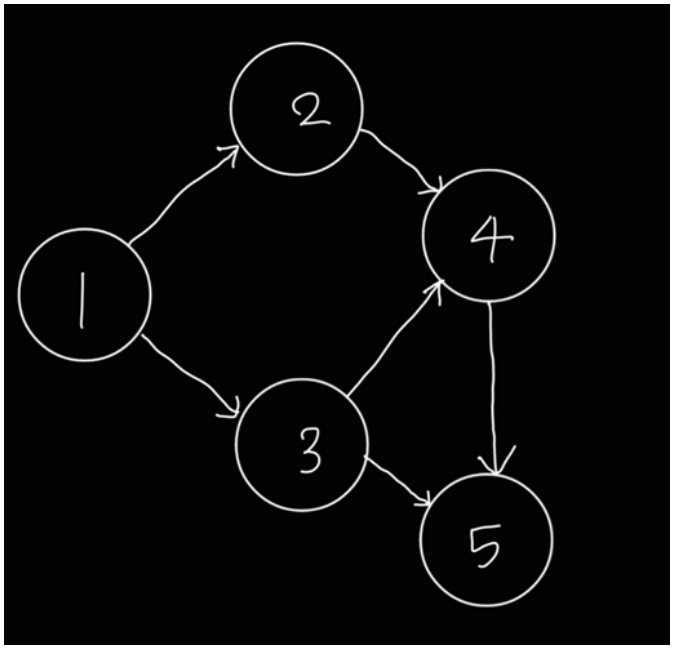
```cpp
#include<bits/stdc++.h>
using namespace std;

/**
 * INF : 대략 10억
 * MAX : 최대 정점 수
 */
const int INF = 0x3f3f3f3f;
const int MAX = 20001;

struct element {
    int pos, cost;
};

/**
 * minCost[i] : 시작 정점에서 i번 정점까지의 최단 거리
 * inQueue[i] : i번 정점이 큐에 들어있는지 확인하는 배열
 * cnt[i] : 음수 사이클을 찾기 위해 i번 정점으로의 최단거리가 몇 번 업데이트 됐는지 기록하는 배열
 */
long long minCost[MAX];
int inQueue[MAX], cnt[MAX];
vector<vector<element>> conn(MAX);

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    while(m--) {
        int a, b, c; cin >> a >> b >> c;
        conn[a].push_back({b, c}); // 단방향 간선
    }

    /**
     * SPFA (Shortest Path Faster Algorithm)
     * - 우선순위 큐 대신 inQueue 배열과 큐를 사용
     */
    fill(minCost, minCost+MAX, INF); // 기본값 초기화
    queue<int> q; q.push(1); minCost[1]=0; // 시작 정점 넣기
    while(!q.empty()) {
        int cur = q.front(); q.pop();
        inQueue[cur]=false;
        if(++cnt[cur]==n) { // 정점이 n번 업데이트 되는 경우 음수 사이클 존재 (Bellman-Ford와 동일)
            cout << -1;
            return 0;
        }
        for(auto next:conn[cur]) {
            long long nextCost = minCost[cur] + next.cost;
```

```
            if(nextCost<minCost[next.pos]) { // 현재 경로로 다음 정점에 도달하는 비용이 더 적다면
                minCost[next.pos] = nextCost; // 최단거리 업데이트
                if(!inQueue[next.pos]) { // 큐에 들어있지 않은 경우만 큐에 넣기
                    q.push(next.pos);
                    inQueue[next.pos]=true;
                }
            }
        }
    }

    for(int i=2;i<=n;i++) {
        if(minCost[i]==INF) cout << "-1₩n";
        else cout << minCost[i] << '₩n';
    }
}
```

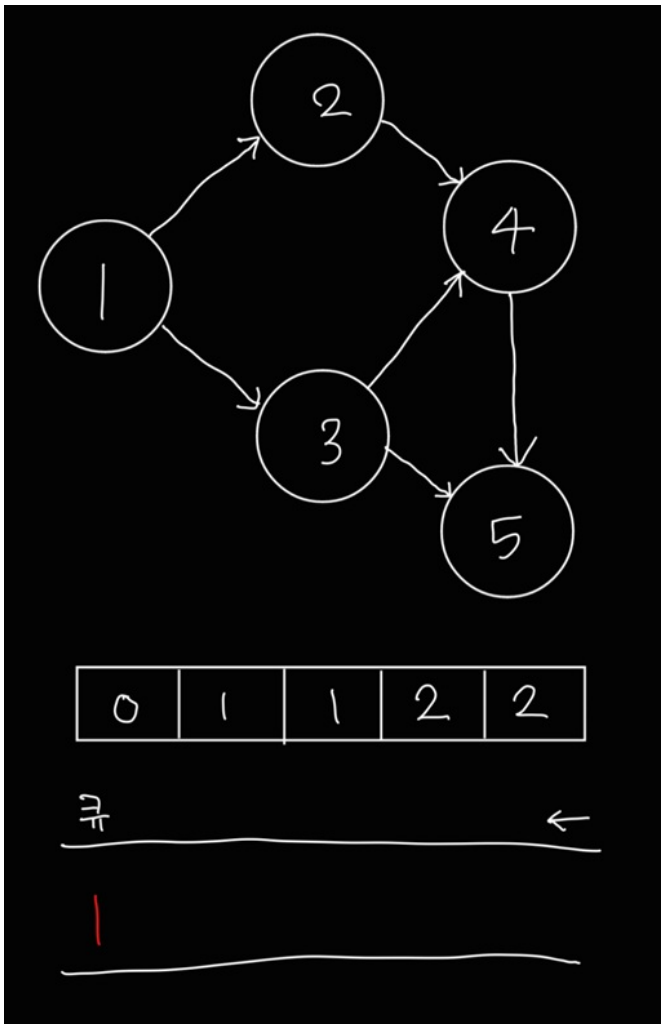
1.14 Kahn's Algorithm (Topological Sort, 위상 정렬)

방향성이 있고 사이클이 없는 그래프(DAG)에서, 모든 정점을 선행 관계를 만족하도록 나열하는 알고리즘

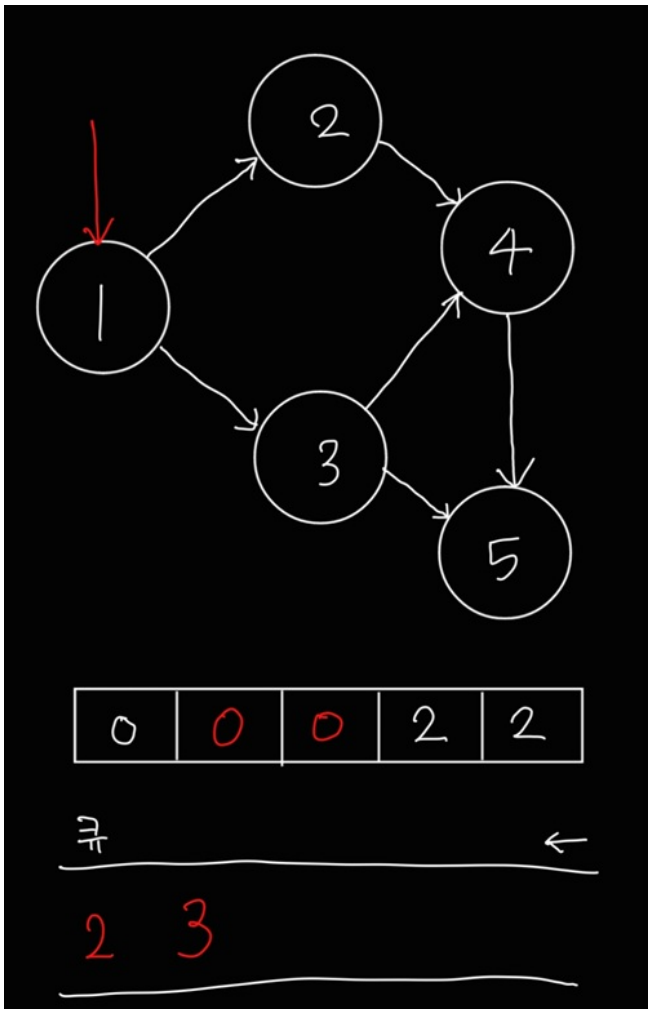
시간복잡도 : O(V+E) (V : 정점 수, E : 간선 수)



이렇게 생긴 그래프가 있다고 하자.

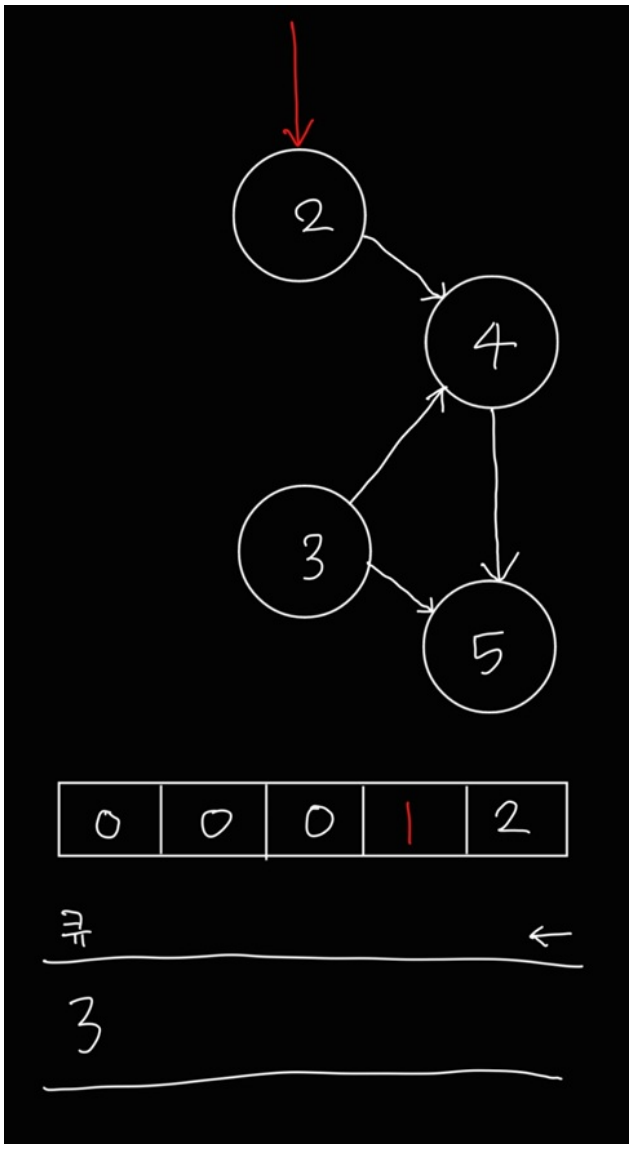


먼저 각 정점의 진입 차수를 구한다. 그 후 진입 차수가 0인 모든 정점을 큐에 넣는다.

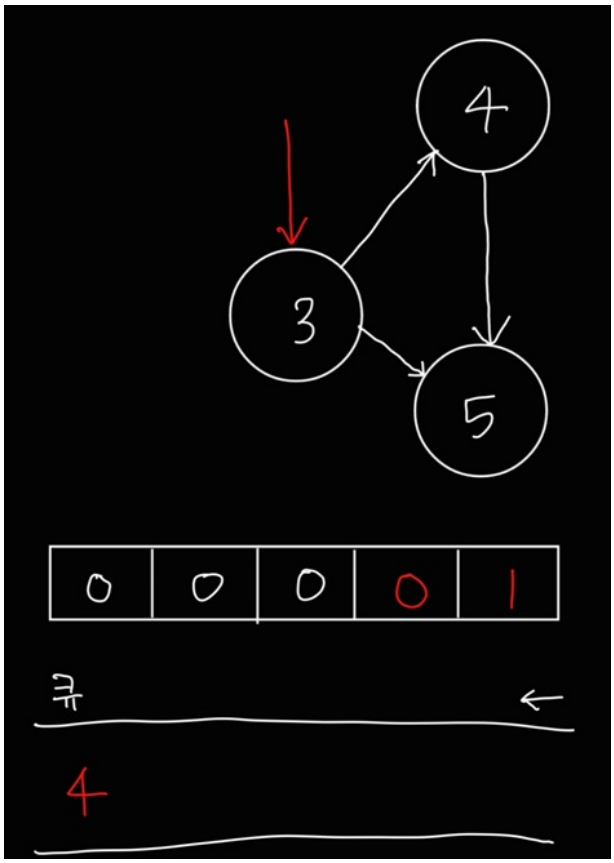


큐에 가장 앞에 있는 정점인 1번 정점을 꺼낸다. 1번 정점에서 접근 가능한 2번, 3번 정점의 진입차수를 지운다.

그 결과 2번, 3번 정점이 진입차수가 0이 되어 큐에 들어간다.

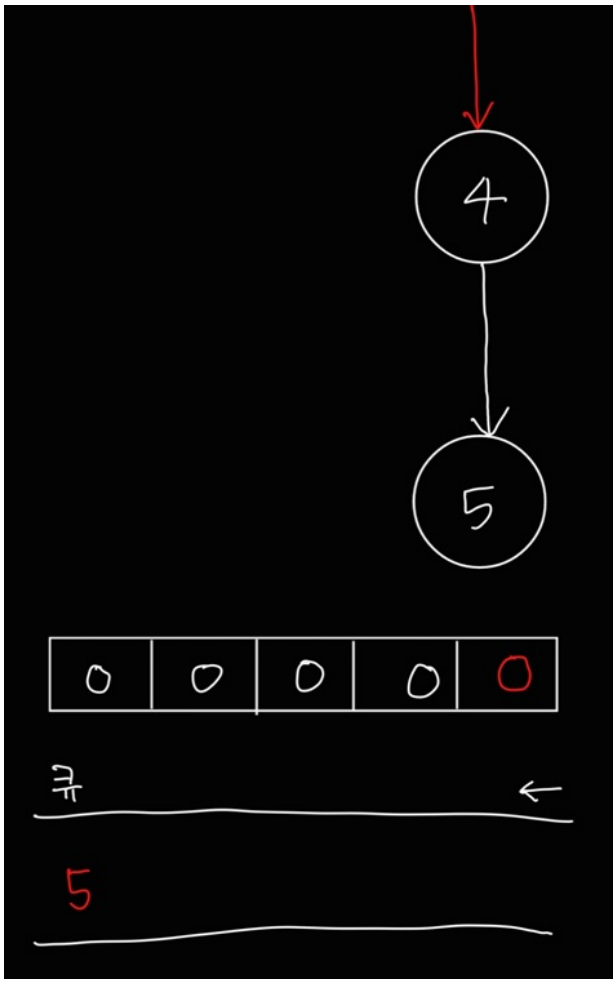


큐에 가장 앞에 있는 정점인 2번 정점을 꺼낸다. 2번 정점에서 접근 가능한 4번 정점의 진입차수를 지운다.



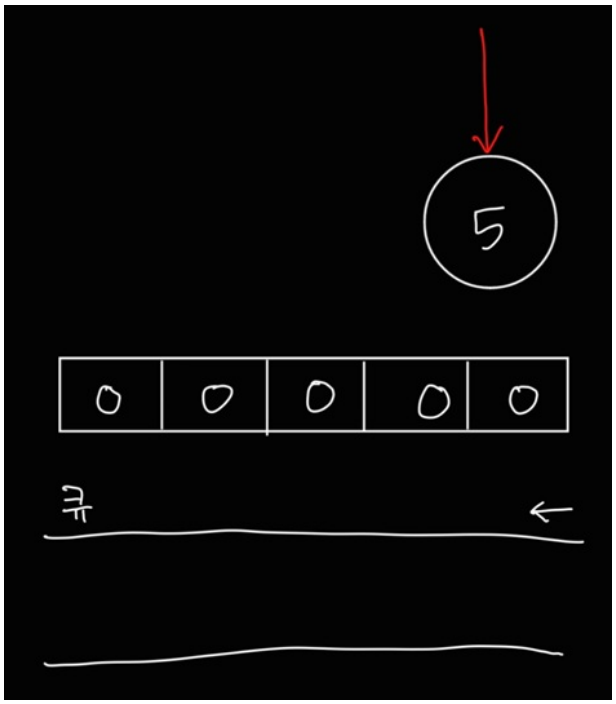
큐에 가장 앞에 있는 정점인 3번 정점을 꺼낸다. 3번 정점에서 접근 가능한 4번, 5번 정점의 진입차수를 지운다.

그 결과 4번 정점이 진입차수가 0이 되어 큐에 들어간다.



큐에 가장 앞에 있는 정점인 4번 정점을 꺼낸다. 3번 정점에서 접근 가능한 5번 정점의 진입차수를 지운다.

그 결과 5번 정점이 진입차수가 0이 되어 큐에 들어간다.



큐에 가장 앞에 있는 정점인 5번 정점을 꺼낸다. 5번 정점에서는 접근 가능한 정점이 없다.

다음 반복에서 큐가 비어 종료한다.

연습 문제 (백준 2252번)

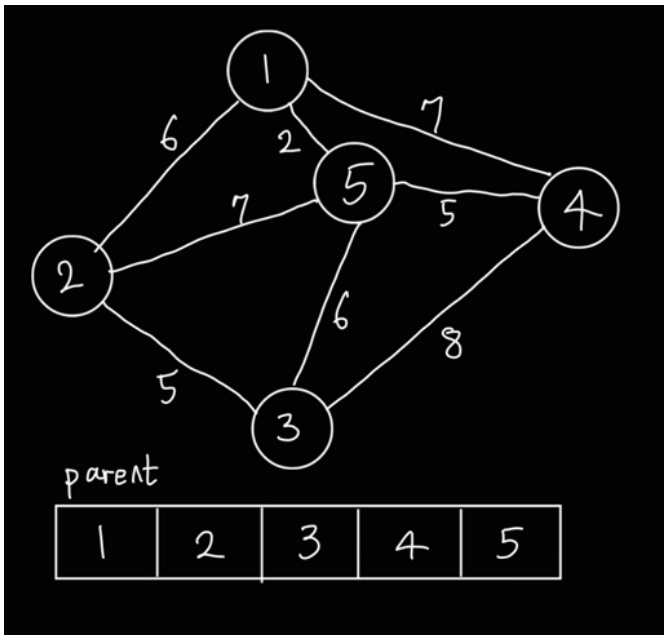
```
/** https://www.acmicpc.net/problem/2252 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

/**
 * MAX : 최대 정점 수
 * inDegree[i] : 정점 i으로 들어오는 간선의 개수
 * conn[i] : 정점 i에서 다른 정점으로 향하는 간선들
 */
const int MAX = 32'001;

int inDegree[MAX];
vector<vector<int>> conn(MAX);

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n, m; cin >> n >> m;
    while(m--) {
```

```
        int a, b; cin >> a >> b;
        inDegree[b]++; // 정점 b로 들어오는 간선의 개수 +1
        conn[a].push_back(b); // 정점 a에서 정점 b로의 간선 연결
    }

    /**
     * Kahn's Algorithm (Topological Sort)
     * 정점 i에 들어오는 간선이 없는 경우 큐에 추가
     * - i에 들어오는 간선이 없다는 뜻은 i보다 선행되야만 하는 정점이 없다는 뜻
     */
    queue<int> q;
    for(int i=1;i<=n;i++) {
        if(!inDegree[i]) {
            q.push(i);
        }
    }

    /**
     * 정점 cur에서 도착 가능한 바로 다음 정점의 inDegree 값을 1씩 감소
     * 다음 정점의 inDegree 값이 0이 되면 큐에 추가
     * - inDegree[next]==0 -> next보다 선행되야 하는 정점이 없다.
     * 이 문제에서는 사이클이 없지만 만약 사이클이 있다면 큐에 들어가는 총 정점 수가 n보다 작음
     */
    while(!q.empty()) {
        int cur = q.front(); q.pop();
        cout << cur << ' ';
        for(int next:conn[cur]) {
            if(--inDegree[next]==0) q.push(next);
        }
    }
}
```

1.15 Kruskal's Algorithm

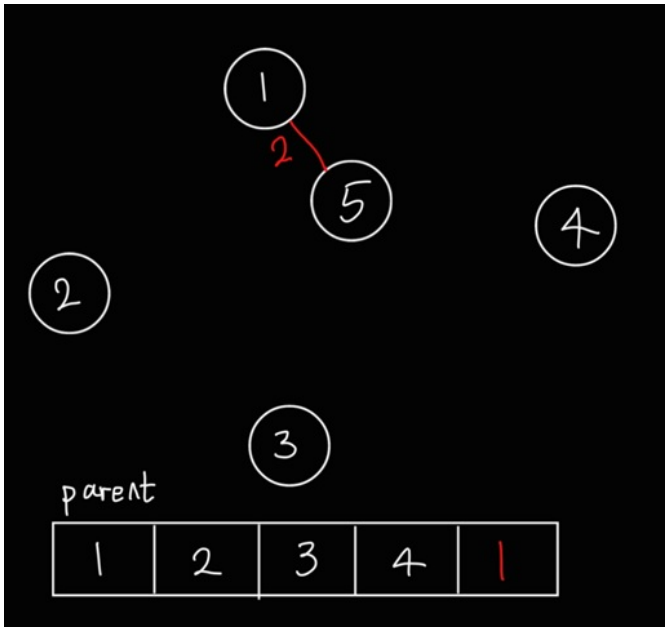
간선을 가중치 순으로 정렬해 사이클을 피하며 최소 스패닝 트리를 만드는 알고리즘

시간복잡도 : O(ElogE) (E : 간선 수)

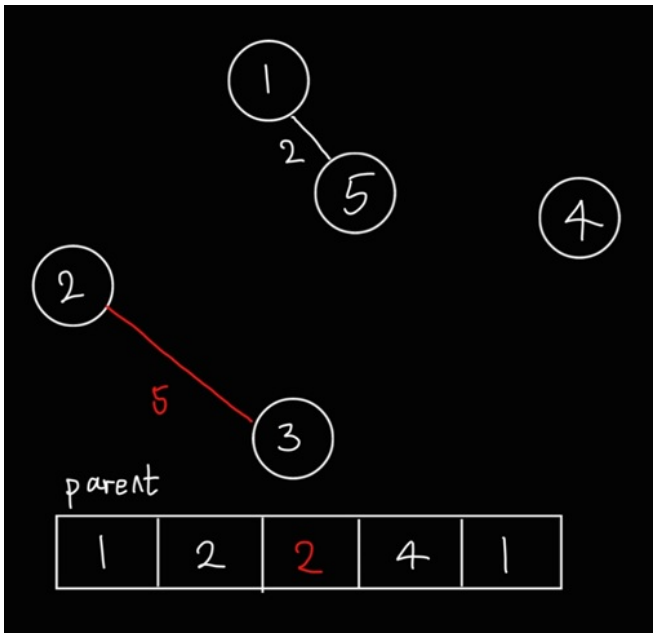


이런 그래프가 주어졌다고 가정하자. 초기에는 집합이 총 5개가 있다.

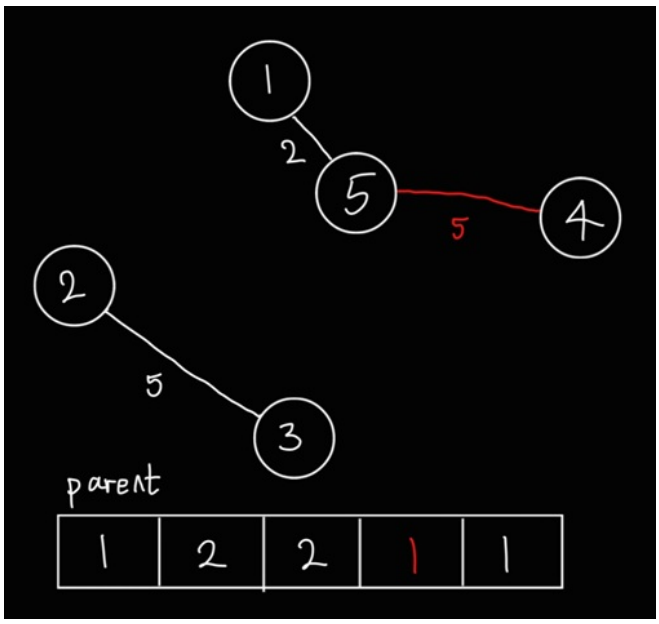
먼저 모든 간선을 끊고, 비용 기준으로 오름차순으로 정렬한다.



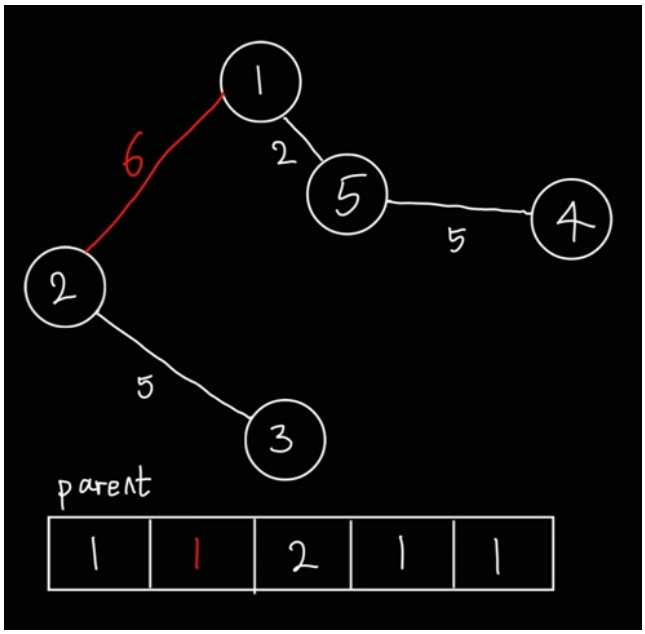
가장 먼저 나오는 간선은 1번 정점과 5번 정점을 2라는 비용으로 연결하는 간선이다. 둘이 다른 집합이니 합친다. (DSU의 union)



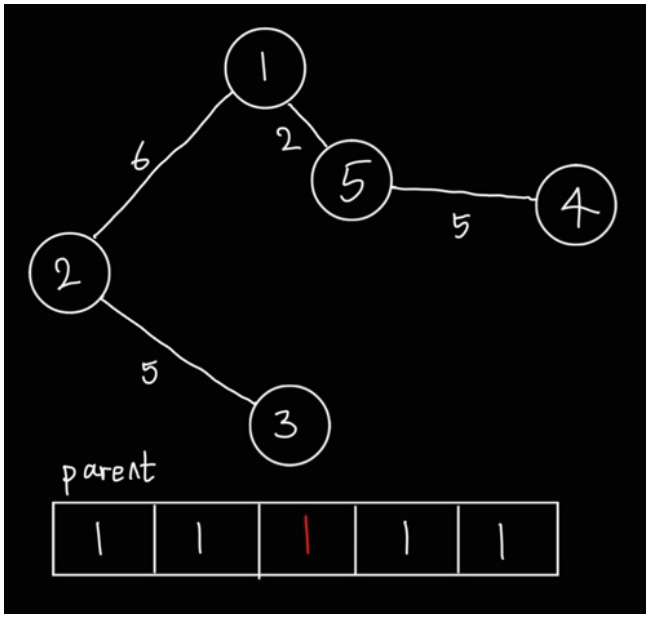
다음으로 나오는 간선은 2번 정점과 3번 정점을 5라는 비용으로 연결하는 간선이다. 둘이 다른 집합이니 합친다.



다음으로 나오는 간선은 4번 정점과 5번 정점을 5라는 비용으로 연결하는 간선이다. 둘이 다른 집합이니 합친다.



다음으로 나오는 간선은 1번 정점과 2번 정점을 6이라는 비용으로 연결하는 간선이다. 둘이 다른 집합이니 합친다.



다음으로 나오는 간선은 3번 정점과 5번 정점을 6이라는 비용으로 연결하는 간선이다. 둘이 같은 집합이니 합치지 않는다.

하지만 두 정점이 같은 집합인지 확인하면서 find연산을 사용하고, 그 때 경로가 압축된다.

나머지 3개의 간선 또한 같은 집합이어서 이대로 끝나게 된다.

연습 문제 (백준 1197번)

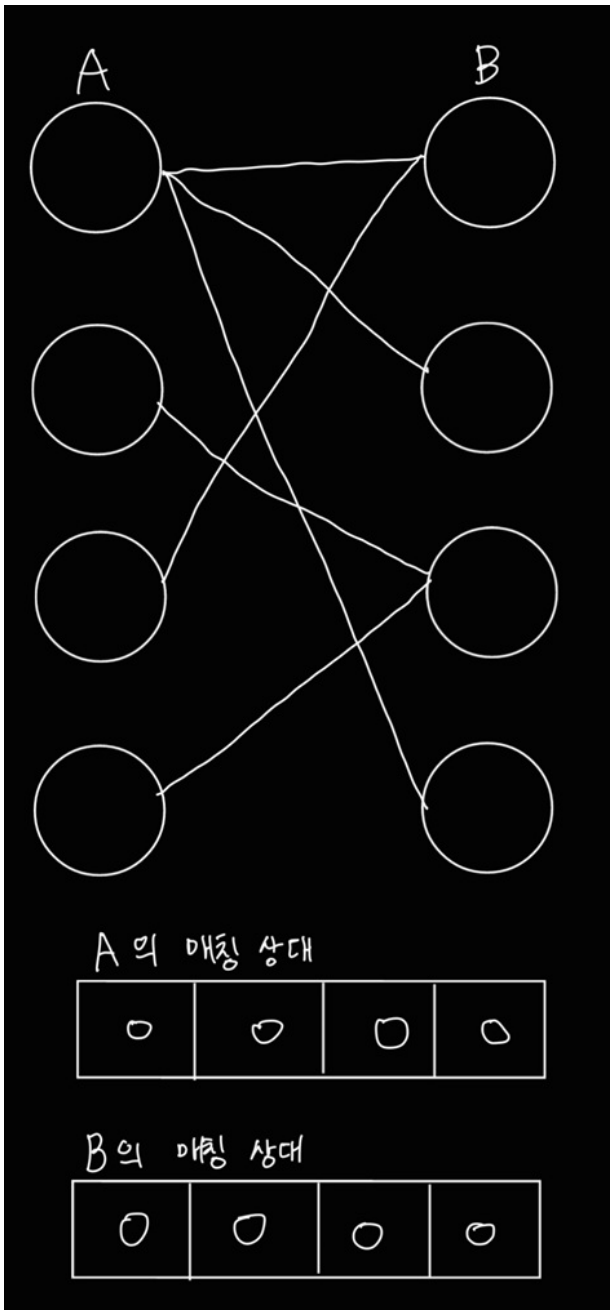
```
/** https://www.acmicpc.net/problem/1197 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

const int MAX = 20'000;

int parent[MAX];

int _find(int x) {
    if(parent[x]==x) return x;
    return parent[x] = _find(parent[x]);
}

bool _union(int x, int y) {
    x = _find(x);
    y = _find(y);

    if(x==y) return false;
    if(x<y) parent[y]=x;
    else parent[x]=y;
    return true;
}

struct edge {
    int a, b, c;
    bool operator<(const edge e) const {
        return c < e.c;
    }
};

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int v, e; cin >> v >> e;
    vector<edge> edges;
    while(e--) {
        int a, b, c; cin >> a >> b >> c;
        edges.push_back({a-1, b-1, c});
    }
    for(int i=0;i<v;i++) parent[i]=i;
```

```
    /**
     * 크루스칼 알고리즘
     * 모든 정점을 최소 비용으로 연결하는 알고리즘
     * - 간선을 비용 기준으로 오름차순으로 정렬
     * - 순서대로 두 정점이 다른 집합에 있을 때만 DSU로 합치면서 비용 누적
     * - - v개의 정점은 v-1번의 합치기 연산 가능
     * - - 나머지 간선은 무시
     */
    int total=0;
    sort(edges.begin(), edges.end());
    for(auto e : edges) {
        if(_union(e.a, e.b)) {
            total += e.c;
        }
    }
    cout << total;
}
```

1.16 Kuhn's Algorithm (Maximum Bipartite Matching, 이분 매칭)

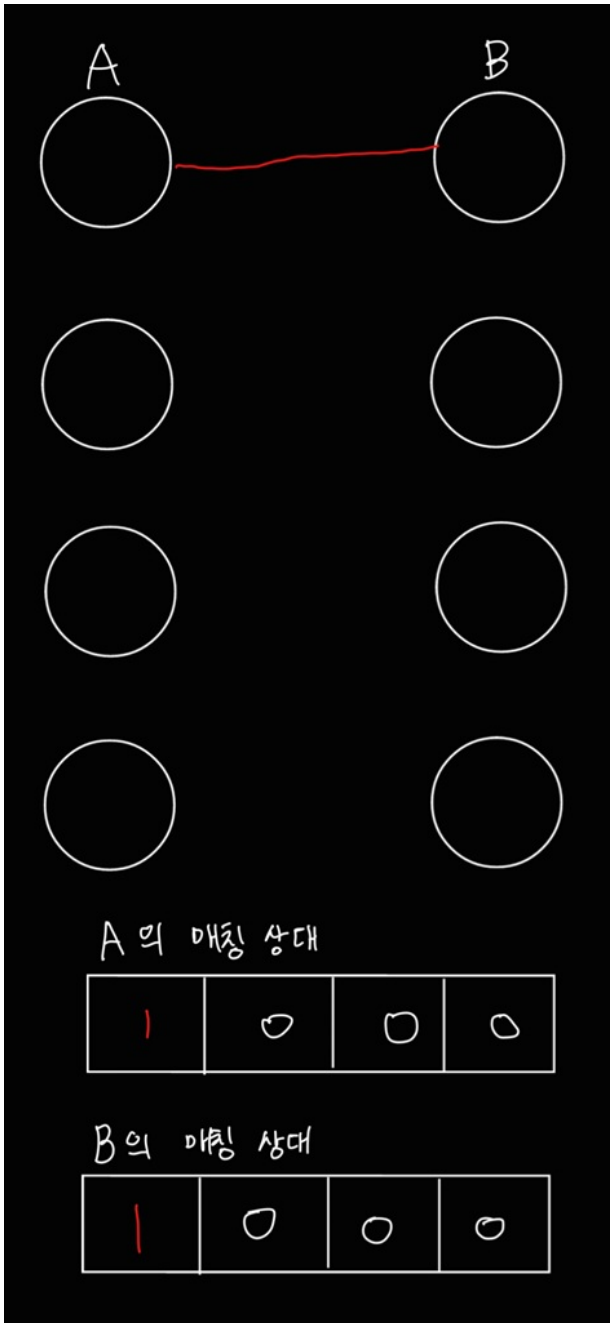
그래프를 이분 그래프로 나타내었을 때 최대 매칭 수(왼쪽과 오른쪽의 쌍의 수)를 찾는 알고리즘

시간복잡도 : O(VE) (V : 왼쪽 그룹의 정점 수)



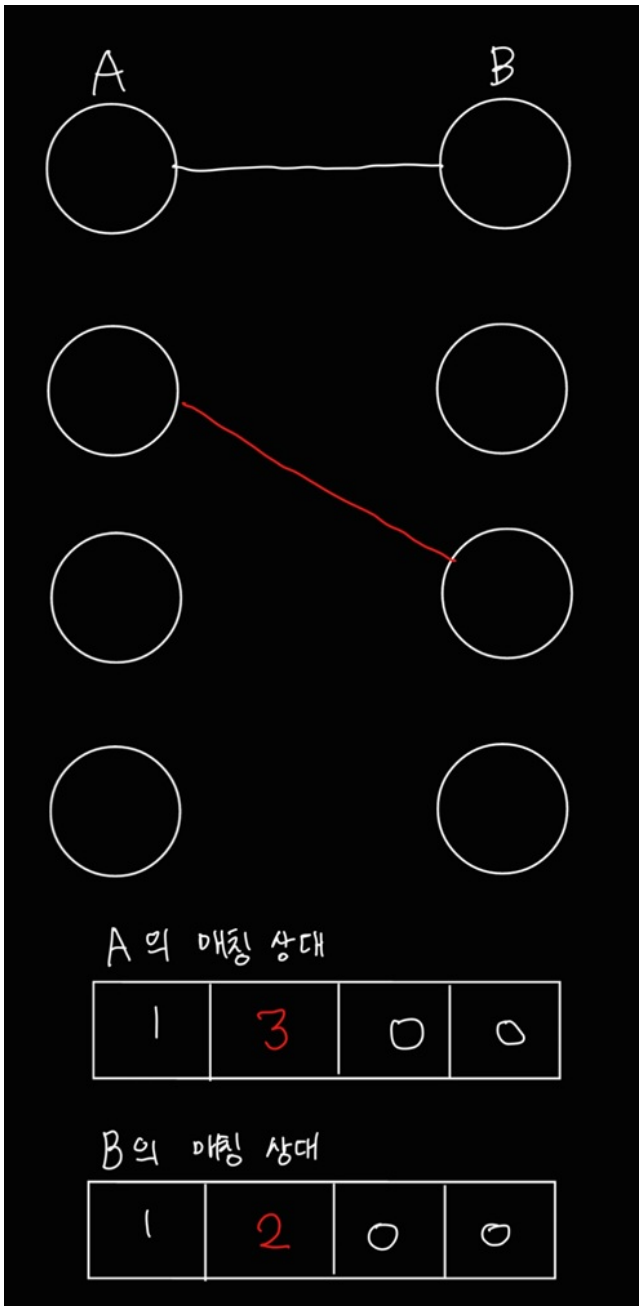
이렇게 생긴 이분 그래프와 초기 상태가 다음과 같은 배열 2개가 있다고 가정하자. 이 그래프에서 왼쪽 정점 - 오른쪽 정점의 어떤 점도 중복되지 않은 최대 쌍의 수를 구할 것이다.

이분 그래프란 정점들을 그림처럼 왼쪽 정점 그룹과 오른쪽 정점 그룹으로 나누었을 때, 서로 다른 그룹들 사이에만 간선이 있는 그래프이다.

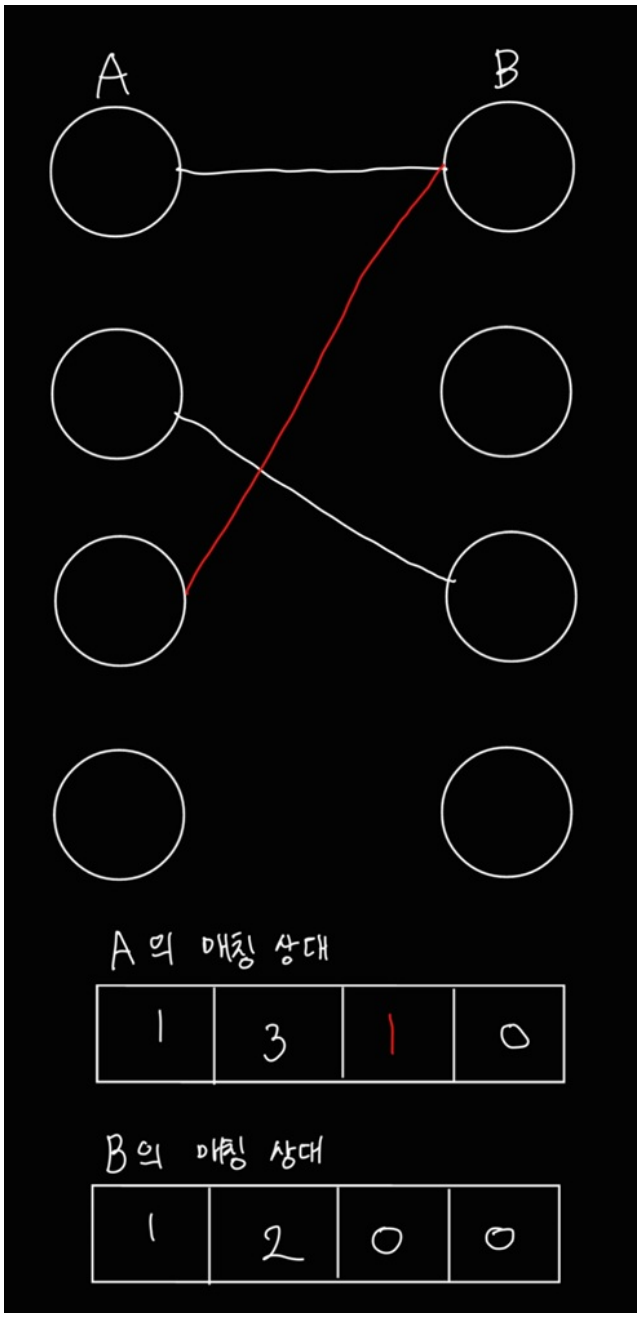


먼저 A1를 연결 가능한 첫번째 매칭 상대인 B1과 연결한다. 중복이 없으니

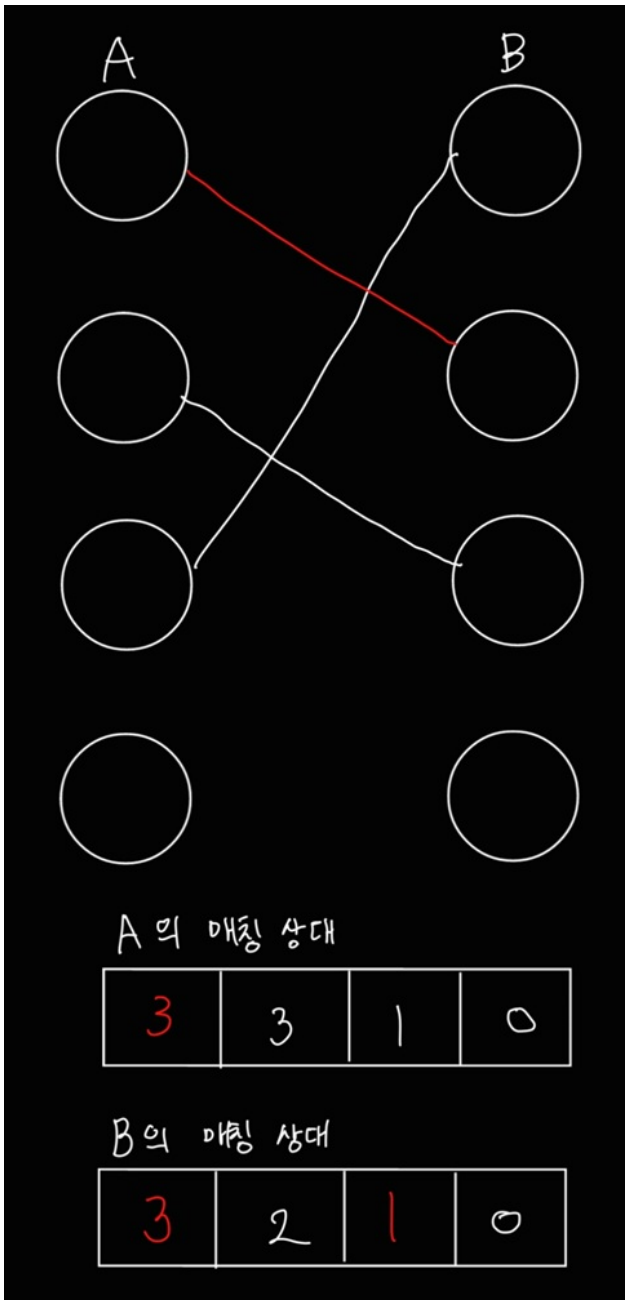
넘어간다.



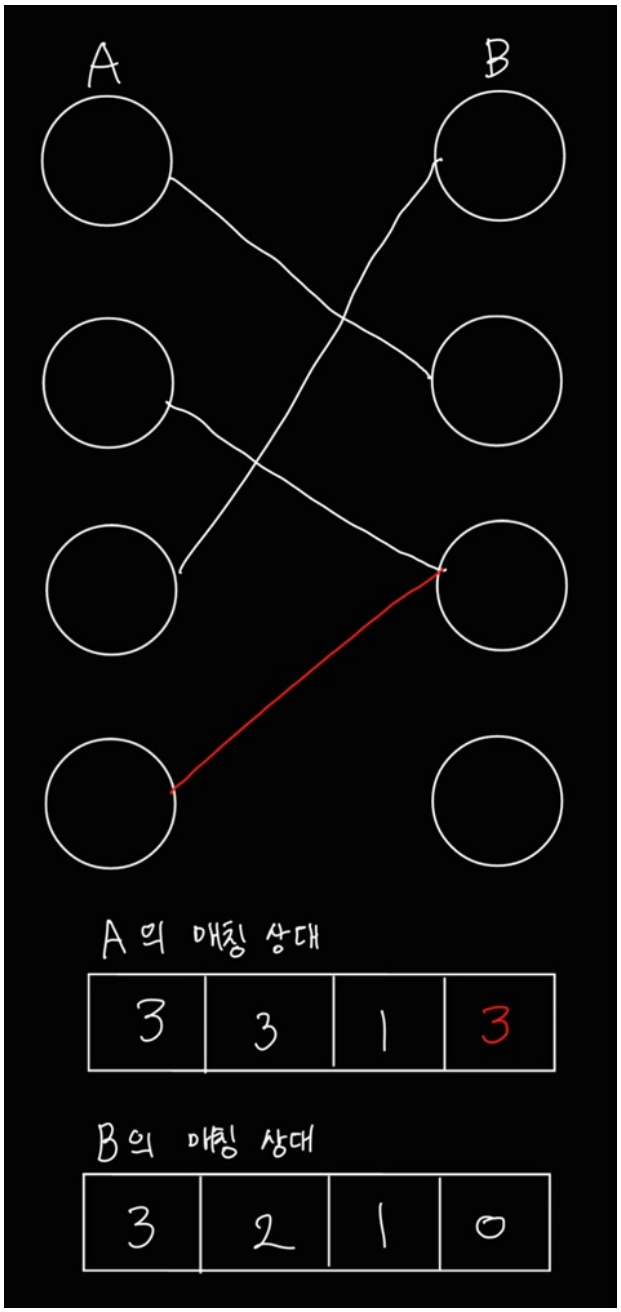
A2를 연결 가능한 첫번째 매칭 상대인 B3와 연결한다. 중복이 없으니 넘어간다.



A3를 연결 가능한 첫번째 매칭 상대인 B1과 연결한다. 중복이 발생했다.



B1과 원래 연결되어있던 A1을 연결 가능한 다음 정점으로 연결하게 한다. 결국 A1은 B2와 연결되었다.



A4를 연결 가능한 첫번째 매칭 상대인 B3과 연결한다. 중복이 발생했다.

하지만 원래 B3과 연결되어있던 A2는 다른 정점과 연결할 수 없고 이대로 종료된다. (B4는 매칭 실패)

연습 문제 (백준 11375번)

직원들을 각각 왼쪽 정점으로, 일들을 각각 오른쪽 정점으로 설정해두고 직원-일 간의 최대 쌍의 수를 구해주면 된다.

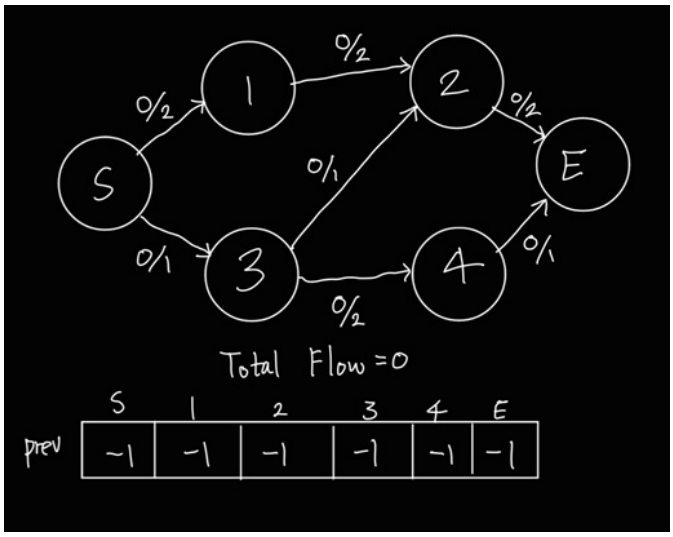
```
/** https://www.acmicpc.net/problem/11375 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

const int MAX = 1001; // 최대 정점 개수 (문제마다 다름)
int A[MAX], B[MAX]; // A : 왼쪽 정점이 선택한 오른쪽 정점 번호, B : 오른쪽 정점이 선택한 왼쪽 정점 번호,
bool visited[MAX]; // 이미 이번 확인에서 방문한 정점인지 확인하기 위한 배열, 매 방문마다 false로 초기화
vector<vector<int>> conn(MAX); // 왼쪽 정점 -> 오른쪽 정점으로의 간선 목록

/**
 * cur번째 왼쪽 정점이 아직 선택이 안된 오른쪽 정점과 새로 매칭이 되는지 확인.
 *
 * 만약 왼쪽 정점과 오른쪽 정점이 다음과 같이 간선이 있다면,
 * 1 -> 1
 * 1 -> 2
 * 2 -> 2
 *
 * main의 for문에서 dfs(1) 호출
 * 1 -> 1 살펴봄 : 첫 매칭 발생. (A[1]==1, B[1]==1)
 *
 * main의 for문에서 dfs(2) 호출
 * 2 -> 1 을 살펴보는데 B[1]에 이미 1이란 값이 들어있음. : 왼쪽 1번 정점을 다른 오른쪽 정점과 매칭할 수
 * 1 -> 1 살펴봄 : 오른쪽 1번 정점은 방금 방문해서 더 살펴볼 필요가 없음(visited[1]==true)
 * 1 -> 2 살펴봄 : 매칭 변경 가능 (1 -> 2로 변경, A[1]==2, B[2]==1)
 * 2 -> 1 다시 살펴봄 : 매칭 가능 (A[2]==1, B[1]==2)
 */
bool dfs(int cur) {
    visited[cur]=true;
    for(int next:conn[cur]) {
        if(B[next]==-1 || !visited[B[next]] && dfs(B[next])) {
            A[cur] = next;
            B[next] = cur;
            return true;
        }
    }
    return false;
}

int main(void) {
    ios::sync_with_stdio(0); cin.tie(0);
    int N, M; cin >> N >> M;
```

```
    /**
     * i번째 왼쪽 정점(직원 정점)이 j번째 오른쪽 정점(일 정점)과 간선이 있음.
     */
    for(int i=1;i<=N;i++) {
        int cnt; cin >> cnt;
        while(cnt--) {
            int j; cin >> j;
            conn[i].push_back(j);
        }
    }

    /**
     * 왼쪽 정점과 오른쪽 정점이 서로 어디에 연결되었는지 확인하기 위한 배열
     * A[i] : i번 왼쪽 정점이 A[i]번 오른쪽 정점과 연결됨.
     * B[i] : i번 오른쪽 정점이 B[i]번 왼쪽 정점과 연결됨.
     * 처음에는 연결이 안되어있어서 -1로 초기화 (꼭 -1일 필요는 없고 절대 등장하지 않은 편한 숫자로 초
     */
    memset(A, -1, sizeof A);
    memset(B, -1, sizeof B);

    /**
     * cnt : 최대 매칭 수
     * i번 왼쪽 정점이 아직 선택되지 않았다면(A[i]==-1) 매칭할 수 있는지 확인
     * dfs(i)==true면, 새로 매칭 가능
     */
    int cnt=0;
    for(int i=1;i<=N;i++) {
        if(A[i]==-1) {
            memset(visited, false, sizeof visited);
            if(dfs(i)) cnt++;
        }
    }
    cout << cnt;
}
```

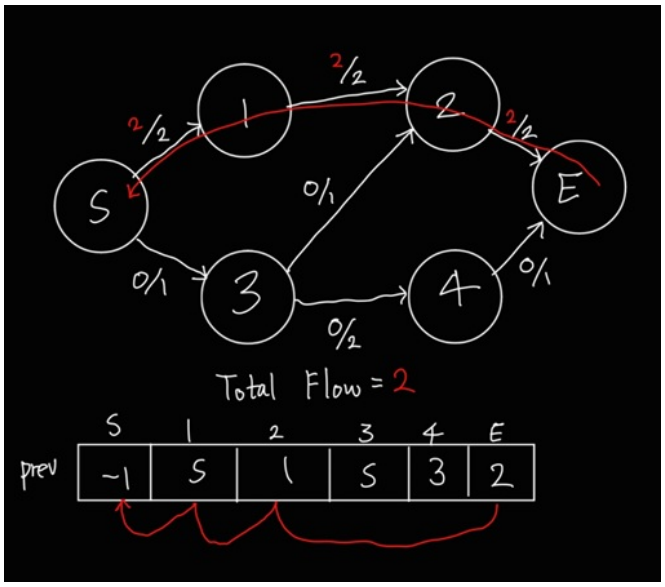
1.17 Edmonds-Karp Algorithm

그래프에서 시작 지점(source)에서 유량을 흘려서, 도착 지점(sink)까지 유량이 얼마나 도착하는지 찾는 알고리즘

시간복잡도 : $O(VE^2)$

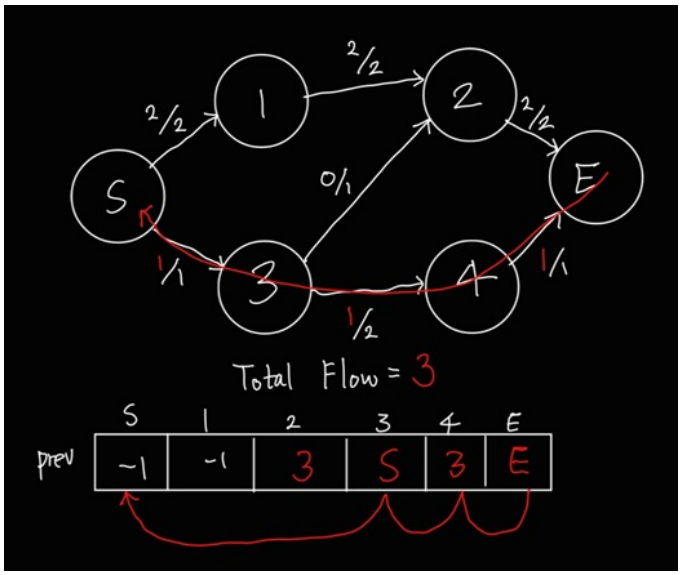


다음과 같은 그래프가 주어졌다고 하자. 간선의 f/c의 f는 현재 흐르는 유량이고(flow), c는 총 용량(capacity)이다.
prev는 이 정점 이전에 방문한 정점을 나타내는 배열이다.
S에서 탐색을 시작하고 인접한 아직 유량을 흘릴 수 있는(c-f>0) 1, 3번 정점을 큐에 추가합니다.
: (bfs라 같아서 패스)
큐가 비어 종료한다.

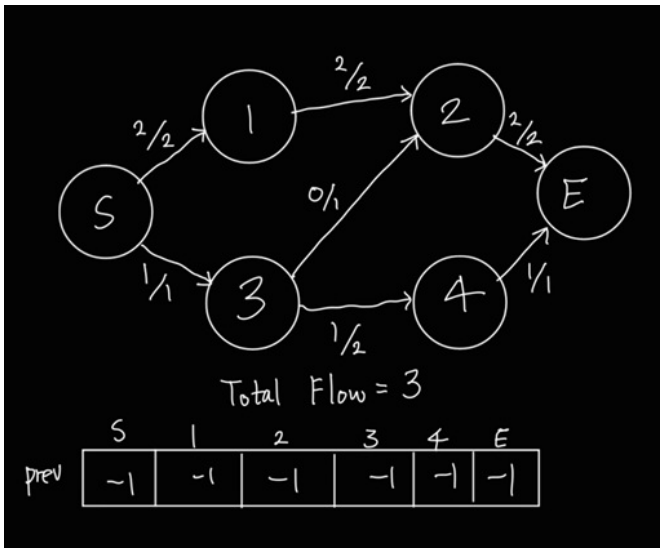


그렇게 만들어진 방문 순서를 통해 E에서 다시 돌아가며 경로를 찾는다.
이 때 E -> 2 -> 1 -> S 경로가 발견되었고, 최대로 흘릴 수 있는 최대 유량이

2이고, 순방향 간선에는 +2를, 역방향 간선에는 -2를 흘려준다. (역방향에도 유량을 흘려주는 이유는 아래에서 설명)

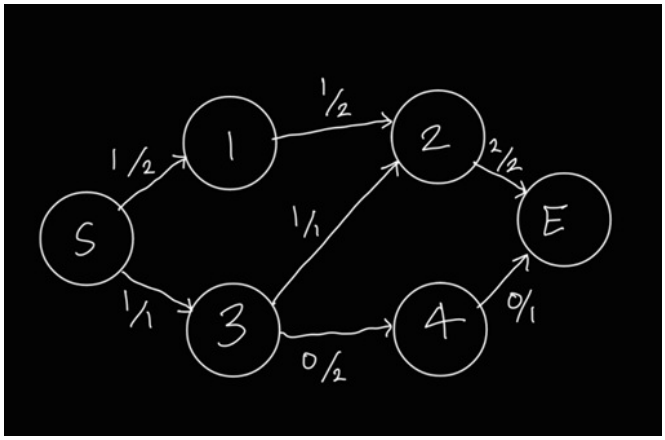


prev를 초기화시키고, 다시 S부터 유량을 흘릴 수 있는 정점들로(c-f>0) bfs를 돌려 경로를 확인한다.
이 때 E -> 4 -> 3 -> S 경로가 발견되었고, 이 경로로 보낼 수 있는 최대 유량인 1이고, 순방향 간선에는 +1을, 역방향 간선에는 -1을 흘려준다.

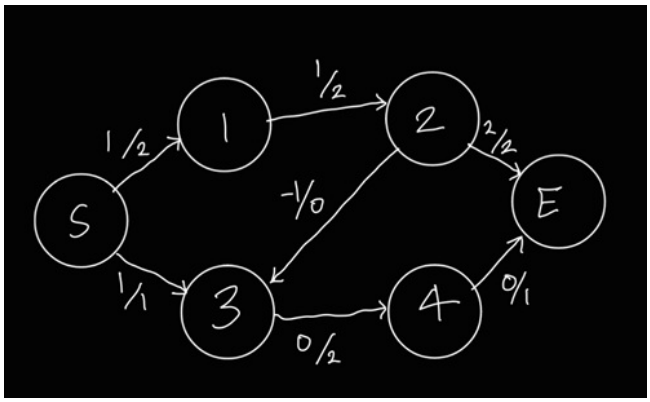


다음 반복에서는 S에서 E로 유량을 흘려보낼 수 없어서 반복을 종료한다.
최종 결과로 총 유량이 3이 나왔다.

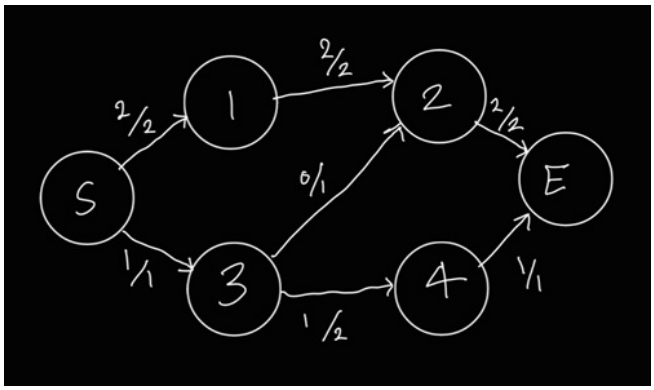
이번 설명에는 순방향 간선만 사용했지만 역방향 간선을 사용해서 더 나은 경로를 찾는 경우도 있다.
역방향 간선으로 더 나은 간선을 찾는 것은 다음과 같은 경우이다.



S -> 3 -> 2 -> E 방향으로 1만큼의 유량이 흐른 후 S -> 1 -> 2 -> E 방향으로 1만큼의 유량이 흘렀다고 가정하면 다음 그림과 같아진다.



이 상황에서 순방향 간선만 보면 더이상 흐를 수 있는 방향이 보이지 않지만, 역방향 간선을 보면 다르다.
3 -> 2의 역간선인 2 -> 3을 나타내면 다음과 같다.



이 역간선까지 포함시키면, S -> 1 -> 2 -> 3 -> 4 -> E 방향으로 1만큼의 유량이 흐를 수 있다.
모든 유량이 흐른 후 최종 상태만 놓고 보면 S -> 3 -> 4 -> E 방향으로 1만큼의 유량이, S -> 1 -> 2 -> E 방향으로 2만큼의 유량이 흐른 것과 동일하다.

연습 문제 (백준 17412번)

```cpp
/** https://www.acmicpc.net/problem/17412 제출 코드 */
#include <bits/stdc++.h>
using namespace std;

/**
 * INF : 대략 10억
 * MAX : 해당 문제의 최대 정점의 개수
 * S : source (시작점)
 * E : sink (도착점)
 */
const int INF = 0x3f3f3f3f;
const int MAX = 400;
const int S = 0;
const int E = 1;

/**
 * c[u][v] : u에서 v로의 최대 용량(capacity)
 * f[u][v] : u에서 v로 흐르는 유량(flow)
 * prv[i] : i에 도달하기 위해 방문한 이전 정점
 * conn[u][v] : 순방향 간선
 * conn[v][u] : 역방향 간선
 */
int c[MAX][MAX], f[MAX][MAX], prv[MAX];
vector<vector<int>> conn(MAX);

int main() {
    ios::sync_with_stdio(0); cin.tie(0);
```

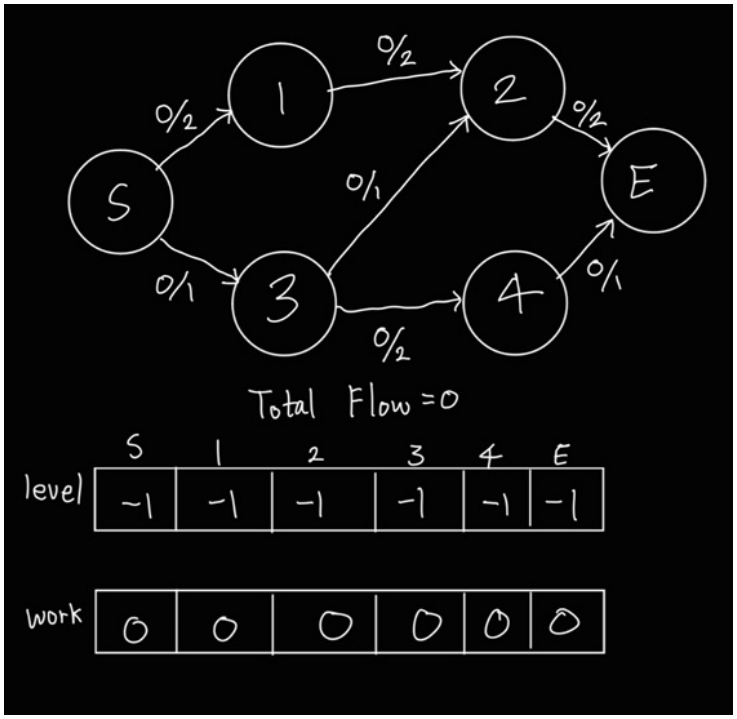
```
int N, P; cin >> N >> P;

/**
 * 정점에 간선 추가
 * 최적의 경로를 찾아 flow를 더 흘리기 위해 역방향 간선도 추가해줘야함.
 */
while(P--) {
    int u, v; cin >> u >> v;
    c[u-1][v-1]++;
    conn[u-1].push_back(v-1);
    conn[v-1].push_back(u-1);
}

/**
 * S -> E 로의 bfs 탐색
 * prv[next] = cur : 역추적을 위해 next -> cur 경로 기록
 */
int totalFlow=0;
while(true) {
    memset(prv, -1, sizeof prv);
    queue<int> q; q.push(S);

    while(!q.empty() && prv[E]==-1) {
        int cur = q.front(); q.pop();
        for(int next:conn[cur]) {
            if(c[cur][next]-f[cur][next]>0 && prv[next]==-1) {
                prv[next] = cur;
                if(next==E) break;
                q.push(next);
            }
        }
    }
    /** 더이상 sink에 도달 못하면 종료 */
    if(prv[E]==-1) break;

    /**
     * sink부터 역추적하면서 최대로 흘릴 수 있는 flow를 찾고, 흘리기
     * f[prv[i]][i] : 순방향 간선이어서 flow만큼 채워짐
     * f[i][prv[i]] : 역방향 간선이어서 flow만큼 빠짐 (더 나은 간선이 있는지 찾기 위해 쓰임)
     *
     * c[1][2] =
     * S -> 1 -> 2 -> E
     */
    int flow=INF;
    for(int i=E;i!=S;i=prv[i]) {
        flow = min(flow, c[prv[i]][i]-f[prv[i]][i]);
```

```
        }
        for(int i=E;i!=S;i=prv[i]) {
            f[prv[i]][i] += flow;
            f[i][prv[i]] -= flow;
        }
        totalFlow += flow;
    }
    cout << totalFlow;
}
```

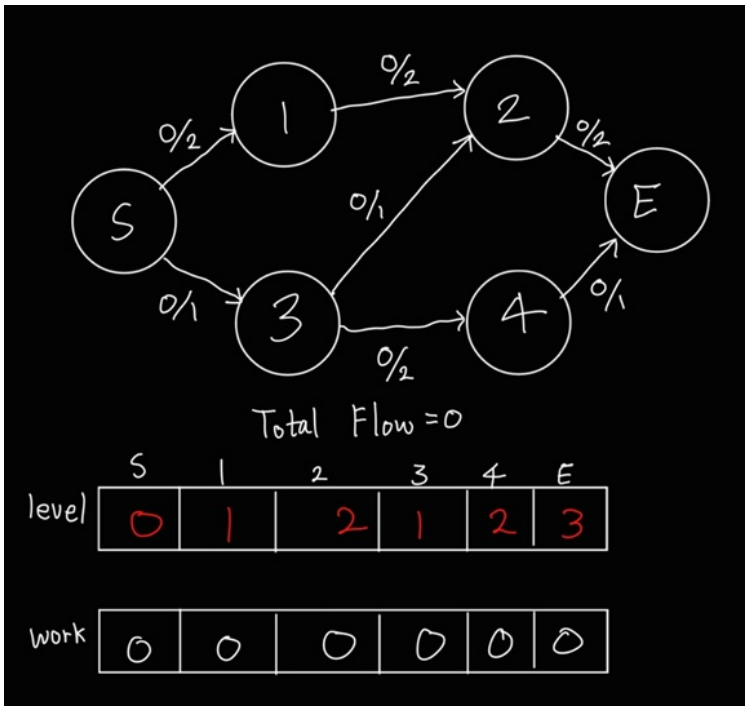
1.18 Dinic's Algorithm

Edmonds-Karp 알고리즘을 최적화한 알고리즘으로, 레벨 그래프를 구성해 블로킹 플로우를 흘려 불필요한 탐색을 줄여 최대 유량을 구하는 알고리즘

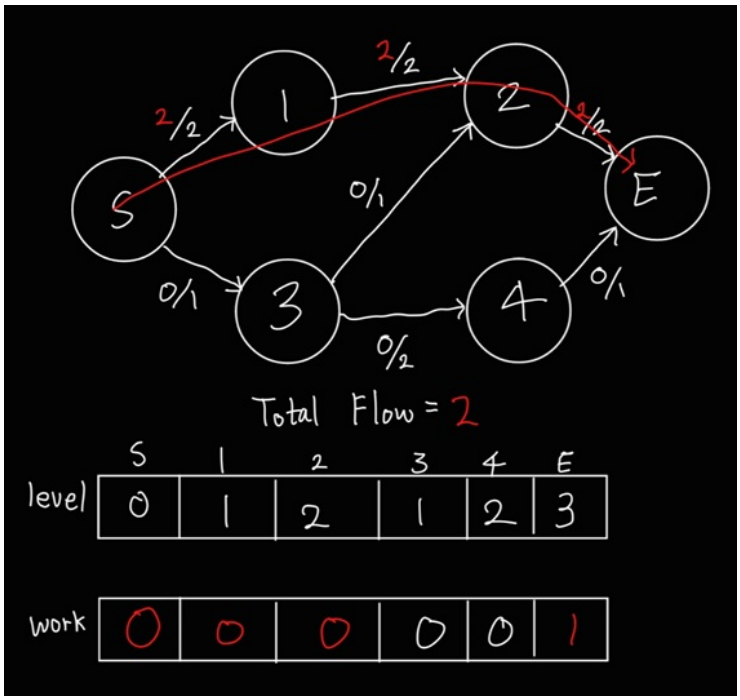
시간복잡도 : $O(V^2E)$



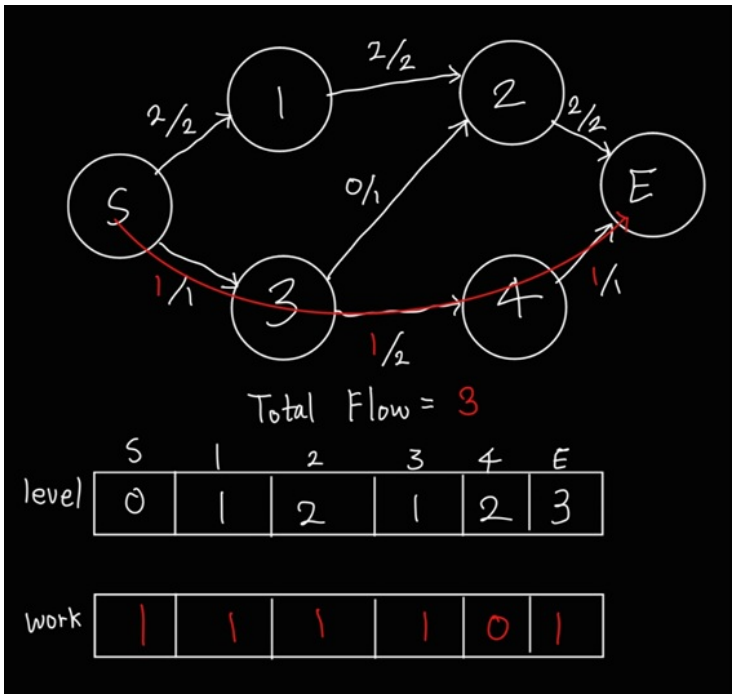
애드몬드-카프 알고리즘에서 썼던 그래프를 다시 사용하겠다. 디닉에서는 prev 배열 대신에 level 배열과 work 배열을 사용한다.



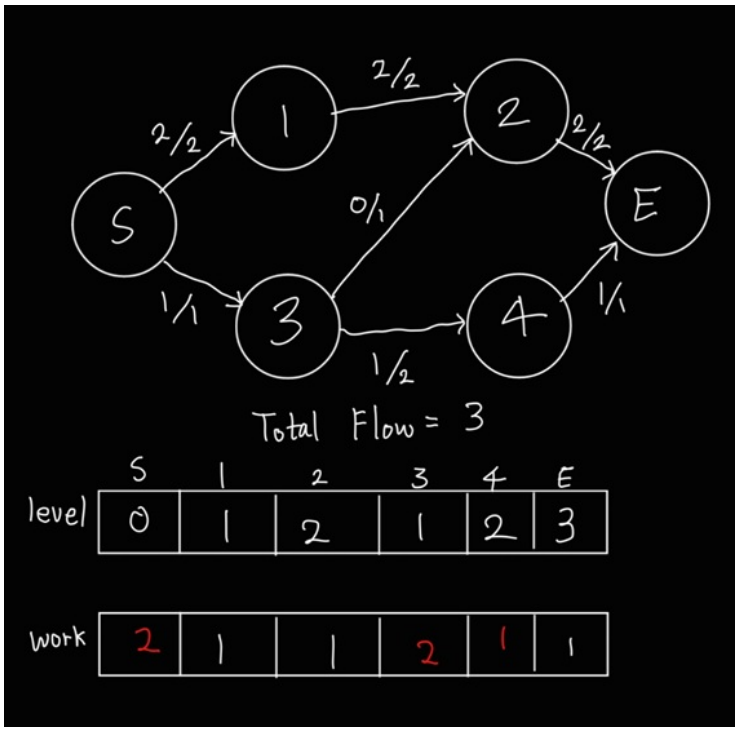
먼저 S에서 유량을 흘려보낼 수 있는(c-f>0) 정점으로 bfs를 한번 돌려 각 정점들의 level(거리)을 찾는다.



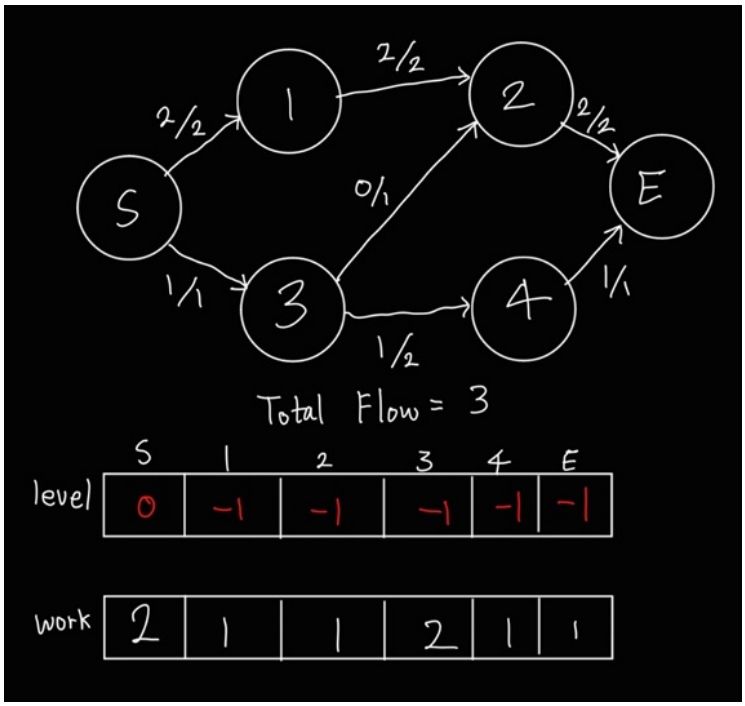
이렇게 만들어진 경로를 통해 dfs로 유량을 흘릴 수 있는 모든 경로에 유량을 보낸다. (조건 : level[next] == level[cur]+1, c-f>0)
첫번 째로 찾은 경로는 S -> 1 -> 2 -> E 이고 이때 경로를 찾으면서 몇 번째 간선을 타고 갔는지 work 배열에 기록한다. (0-Index로 구현, work[i]의 값이 1이면 이번에 1번 간선을 타고 갔고, 1보다 작은 번호의 간선들로는 유량을 흘려도 E에 도착 못한다는 뜻)
E -> 2 -> 1 -> S 로 돌아오며 유량을 2씩 흘려준다. (역간선으로는 -2씩 흘려준다.)



아직 더 유량을 흘려보낼 수 있는지 모르기 때문에 다시 dfs를 시작한다.
이번에 찾은 경로는 S -> 3 -> 4 -> E 이고 이때 경로를 찾으면서 몇 번째 간선을 타고 갔는지 work 배열에 기록한다.
E -> 4 -> 3 -> S 로 돌아오며 유량을 1씩 흘려준다. (역간선으로는 -1씩 흘려준다.)



아직 더 유량을 흘려보낼 수 있는지 모르기 때문에 다시 dfs를 시작한다.
이번에는 E에 도착하지 못해, 찾은 경로가 없다.



이전에 bfs로 찾은 level들로는 더이상 유량을 보낼 수 없기에 다른 level 경로들을 찾는다.
S에서 bfs로 E에 도착하지 못하기 때문에 dfs 탐색을 더 이상 하지 않고 종료한다.
(work배열은 dfs 탐색 바로 전에 초기화되기 때문에 쓰레기 값)

연습 문제 (백준 13161번)

```
/** https://www.acmicpc.net/problem/13161 제출 코드 */
#include<bits/stdc++.h>
using namespace std;

/**
 * MAX : 최대 정점 수
 * INF : 대략 10억
 * S : source (시작 지점)
 * E : sink (도착 지점)
 */
const int MAX = 502;
const int INF = 0x3f3f3f3f;
const int S = MAX-2, E = MAX-1;

/**
 * c[i][j] : i번 정점에서 j번 정점으로의 간선의 최대 용량 (capacity)
 * f[i][j] : i번 정점에서 j번 정점으로의 유량 (flow)
```

```
 * level[depth] : source와 각 정점들 사이의 거리 (현재 흐를 수 있는 간선만 사용해서)
 * work[nodeNum] : 더이상 안봐도 되는 간선들 패스하기 위해 사용
 * conn[i][j] : 순방향 간선
 * conn[j][i] : 역방향 간선
 */
int c[MAX][MAX], f[MAX][MAX], level[MAX], work[MAX];
vector<vector<int>> conn(MAX);

/**
 * bfs를 통해 더 흐를 수 있는 간선만 타고 내려가면서 source와 각 정점들 사이의 거리를 구한다.
 */
bool bfs() {
    queue<int> q; q.push(S);
    memset(level, -1, sizeof level);
    level[S]=0;
    while(!q.empty()) {
        int cur = q.front(); q.pop();
        for(int next:conn[cur]) {
            if(level[next]==-1 && c[cur][next]-f[cur][next]>0) {
                level[next] = level[cur]+1;
                q.push(next);
            }
        }
    }
    return level[E]!=-1;
}

/**
 * bfs로 구해진 정점들의 거리를 통해, 거리가 1씩 멀어지도록 유량을 흘려보내준다.
 * int &i = work[cur]; 로 work[cur]과 i 연결(더이상 보지 않아도 되는 간선은 패스하여 최적화)
 */
int dfs(int cur, int curFlow) {
    if(cur==E) return curFlow;
    for(int &i=work[cur];i<conn[cur].size();i++) {
        int next = conn[cur][i];
        if(level[next]==level[cur]+1 && c[cur][next]-f[cur][next]>0) {
            int flow = dfs(next, min(curFlow, c[cur][next]-f[cur][next]));
            if(flow>0) {
                f[cur][next] += flow;
                f[next][cur] -= flow;
                return flow;
            }
        }
    }
    return 0; // 유량을 더이상 흘리지 못하면 0 return
}
```

```
int main() {
    ios::sync_with_stdio(0); cin.tie(0);
    int n; cin >> n;
    for(int i=0;i<n;i++) {
        int type; cin >> type;
        if(type==1) {
            conn[S].push_back(i);
            conn[i].push_back(S);
            c[S][i]=INF;
        } else if(type==2) {
            conn[E].push_back(i);
            conn[i].push_back(E);
            c[i][E]=INF;
        }
    }

    for(int i=0;i<n;i++) {
        for(int j=0;j<n;j++) {
            cin >> c[i][j];
            if(i!=j) conn[i].push_back(j);
        }
    }

    /**
     * main 함수의 이 전까지는 애드몬드-카프 알고리즘 쓸 때와 같다.
     *
     * 디닉 알고리즘은 bfs로 source과 각 정점들의 거리를 확인 후
     * dfs로 flow를 흘려준다.
     * 더이상 못 흘려보내주면 다시 bfs해서 거리 재설정.
     * source에서 아무 정점에도 가지 못하면 디닉 알고리즘 종료
     */
    int totalFlow=0;
    while(bfs()) {
        memset(work, 0, sizeof work);
        while(true) {
            int flow = dfs(S, INF);
            if(flow==0) break;
            totalFlow += flow;
        }
    }
    /** 디닉 알고리즘 끝 */

    bfs();
    vector<int> A, B;
    for(int i=0;i<n;i++) {
```

```
        if(level[i]==-1) B.push_back(i+1);
        else A.push_back(i+1);
    }

    cout << totalFlow << '₩n';
    for(int e:A) cout << e << ' '; cout << '₩n';
    for(int e:B) cout << e << ' '; cout << '₩n';
}
```