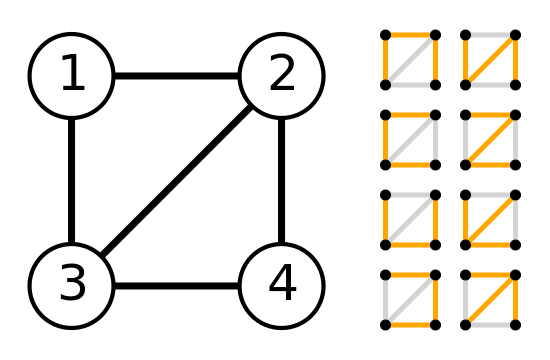
**MST (Minimum Spanning Tree, 최소 신장 트리)**

**1. 신장트리 (Spanning Tree)란**

n개의 정점으로 이루어진 무향그래프에서 n개의 정점과 n-1개의 간선으로 이루어진 트리

예를들어 하단 좌측과 같은 원 그래프의 스패닝 트리는 하단 우측과 같이 총 8개의 스패닝 트리들을 가질 수 있다.



**2. 최소 신장 트리란**

신장 트리를 구성하는 간선들의 가중치의 합이 최소인 신장 트리

MST를 만드는 크루스칼, Prim 알고리즘 모두 결정을 할 때 그 순간에 가장 좋다고 생각되는 것을 선택하는 greedy에 기반되어 있다.

**3. Kruskal’s algorithm**

1. 최초, 모든 간선을 가중치에 따라 오름차순으로 정렬
2. 가중치가 가장 낮은 간선부터 선택하면서 트리를 증가시킴. 사이클이 존재하면 다음으로 가중치가 낮은 간선 선택
3. n-1개의 간선이 선택될때까지 반복

여기서 Cycle이 존재하는지 여부를 확인할 때 DFS방식, Disjoint Set방식이 있음

DFS – 그래프 탐색 기법으로 그래프의 Cycle을 형성하는지 체크

**Disjoint Set (Union-find)**

Make : 원소를 각각 개별적인 집합으로 만듬 (parents 배열)

Find : 두 정점을 합칠 때 대표 값을 확인하는 방법

Union : 만약 두 집합의 대표값이 다르면 두 집합을 합치는 작업

import java.util.Arrays;

import java.util.Scanner;

public class DisJointSetTest {

static int[] parents; // 루트노드의 번호를 기억

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int N = sc.nextInt(); // 원소의 수

int count = sc.nextInt();

parents = new int[N];

// 1. make set : 원소를 각각 개별적인 집합으로 만듦 (즉, 자신이 루트노드가 되도록)

Arrays.fill(parents, -1);

for (int i = 0; i < count; i++) {

int a = sc.nextInt();

int b = sc.nextInt();

System.out.println(union(a, b));

}

System.out.println(find(0) + "//" + find(1));

}

// union : x와 y를 포함하는 두 집합을 통합하는 오퍼레이터

private static boolean union(int a, int b) {

// 2. find set : 자신이 속한 집합 찾기

int aRoot = find(a);

int bRoot = find(b);

if (aRoot != bRoot) { // 3. union set : 두 노드의 집합이 다르면 합치기

parents[bRoot] = aRoot;

return true;

}

return false;

}

// find set : x를 포함하는 집합을 찾음

private static int find(int a) {

if(parents[a] < 0) return a; // root노드면 자신의 번호 리턴

// root노드가 아니면 부모노드 따라가서 루트노드 알아온다.

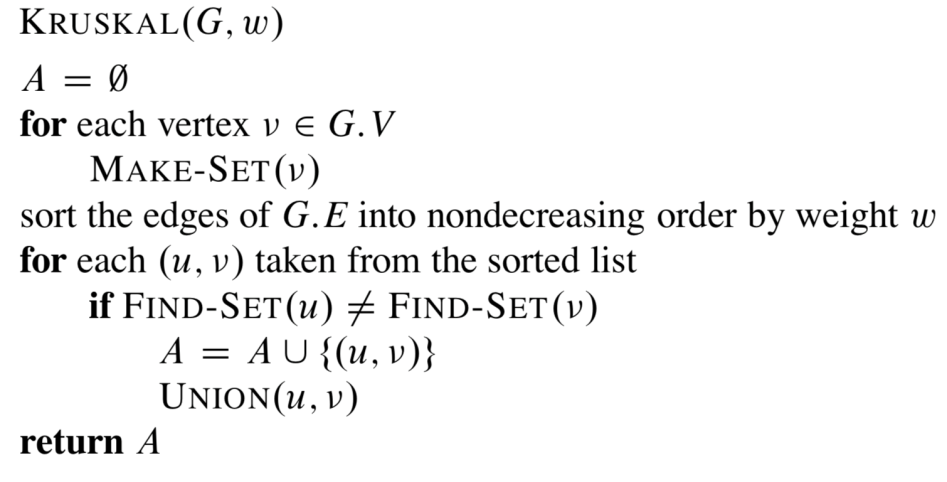
// path compression : 리턴받은 루트노드의 값으로 자신의 부모로 갱신

return parents[a] = find(parents[a]);

}

}

**크루스칼 sudo코드**



**Kruskal의 시간복잡도**

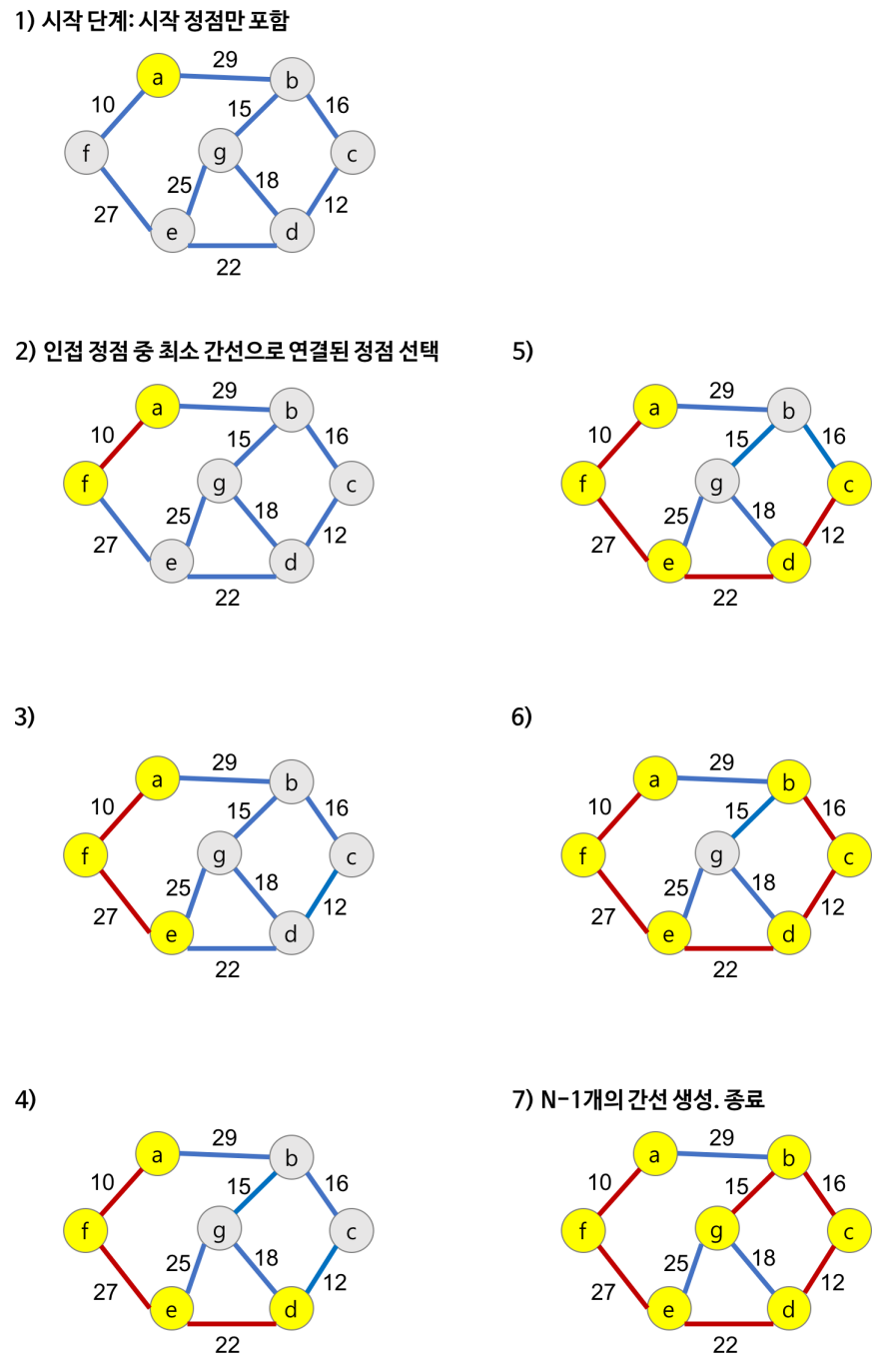
- union-find 알고리즘을 이용하면 Kruskal 알고리즘의 시간 복잡도는 간선들을 정렬하는 시간에 좌우된다.

- 즉, 간선 N개를 퀵 정렬과 같은 효율적인 알고리즘으로 정렬한다면 Kruskal 알고리즘의 시간 복잡도는 O(ElogE) 이 된다.

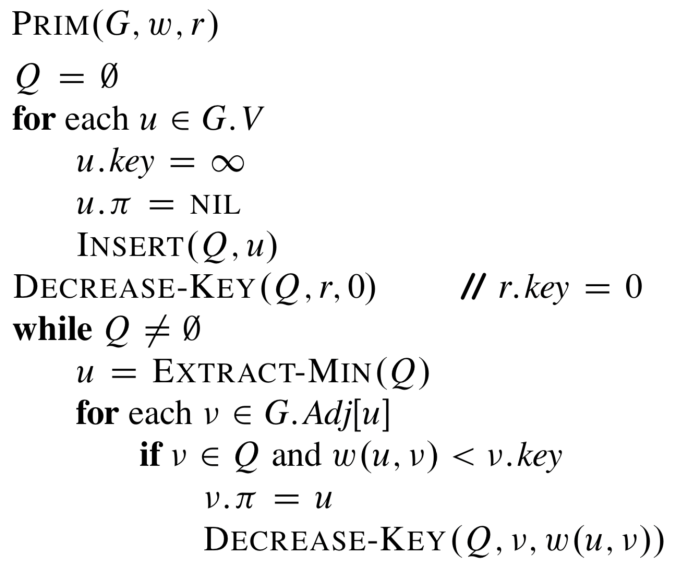
**4. Prim’s algorithm**

하나의 정점에서 연결된 간선들 중에 하나씩 선택하면서 MST를 만들어감

1. 임의 정점을 하나 선택해서 시작
2. 선택한 정점과 인접하는 정점들 중의 최소 비용의 간선이 존재하는 정점을 선택
3. 모든 정점이 선택될 때까지 반복



**prim의 수도코드**



**prim의 시간 복잡도**

변의 개수를 E, 꼭지점의 개수를 N이라고 하면

배열을 사용하면 O(N^2), 이진 힙을 사용하면 O(ElogN), 피보나치 힙을 사용하면 O(E+NlogN)

- 배열을 사용할 경우에 최소비용간선을 가진 정점을 계속 찾아야 하기 때문에 O(N^2)

- 모든 정점을 우선 순위 큐로 확인하니 logN, 그 정점들에 대해 간선을 확인하니 E. 따라서 O(ElogN)

public class PrimTest2 {

public static void main(String[] args) throws NumberFormatException, IOException {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int N = Integer.parseInt(br.readLine());

int[][] adjMatrix = new int[N][N]; // 인접행렬

boolean[] visited = new boolean[N]; // 정점방문여부

StringTokenizer st;

// i 노드에서 j 노드까지 비용 저장

for (int i = 0; i < N; i++) {

st = new StringTokenizer(br.readLine(), " ");

for (int j = 0; j < N; j++) {

adjMatrix[i][j] = Integer.parseInt(st.nextToken());

}

}

int index = 0, min = 0, count = 0, result;

PriorityQueue<Vertex> que = new PriorityQueue<Vertex>();

// 임의의 정점(0)을 첫 정점으로 선택

que.offer(new Vertex(0, 0));

result = 0;

while (!que.isEmpty()) {

Vertex current = que.poll();

if (visited[current.vertex]) continue; // 현정점이 이미 처리된 정점이면 다음 poll

result += current.weight;

visited[current.vertex] = true;

if(++count == N) break;

for (int i = 0; i < N; i++) {

if(!visited[i] && adjMatrix[current.vertex][i] != 0) {

que.offer(new Vertex(i, adjMatrix[current.vertex][i]));

}

}

}

System.out.println(result);

}

static class Vertex implements Comparable<Vertex>{

int vertex;

int weight;

public Vertex(int vertex, int weight) {

super();

this.vertex = vertex;

this.weight = weight;

}

@Override

public int compareTo(Vertex o) {

return Integer.compare(this.weight, o.weight);

}

}

}

**5. Kruskal과 Prim의 비교**

Kruskal 시간 복잡도는 O(ElogE), Prim 시간복잡도는 O(ElogN)

Kruskal은 간선 선택 기반이므로 간선이 많이 없는 그래프(희소그래프)일 경우에 적합하고,

Prim은 정점 선택 기반이므로 간선이 많은 그래프(밀집 그래프)에 적합하다.

**최단 경로 알고리즘**

최단 경로 : 간선의 가중치가 있는 그래프에서 두 정점 사이의 경로들 중에 간선의 가중치의 합이 최소인 경로

**1. Dijkstra 알고리즘 (한점 – 다수의 점)**

- 시작 정점에서 거리가 최소인 정점을 선택해 나가면서 최단 경로를 구함

- Greedy 기법으로 MST Prim 알고리즘과 비슷하다.

1. 현재 선택한 정점(처음엔 시작점)에 곧장 연결되고, 아직 방문하지 않은 정점들을 모두 본다.
2. 선택한 정점과 보고있는 정점 사이의 거리와 시작 정점과 선택한 정점까지의 최단거리의 합이 현재까지 구한 시작 정점과 보고 있는 정점 사이의 거리보다 짧을 경우, 이를 갱신해준다.
3. 1, 2번 수행이 모두 끝난 이후, 아직 방문하지 않은 정점들 중 시작점과의 거리가 가장 짧은 정점을 선택한다.
4. 방문하지 않은 정점이 존재하여 정점을 선택했다면, 현재 구한 시작점으로부터의 현재 선택한 정점 사이의 거리는 최단거리이다.
5. 방문하지 않은 정점이 존재하지 않는다면, 수행을 종료한다. 그렇지 않으면 1번으로 돌아가 수행을 반복한다.

**다익스트라와 프림의 차이**

다익스트라는 시작 노드로 부터 그래프 상에 존재하는 모든 사이 즉, 두 노드 사이의 최단거리를 구하는 것.

반면 프림알고리즘은 Tree상에 존재하는 모든 노드들을 최소비용으로 연결시키는 것이다.

**2. Floyd-Warshall 알고리즘 - DP 접근 방법 (모든점 – 모든점)**

- 각 점을 시작점으로 정해 다익스트라(Dijkstra)의 최단 경로 알고리즘을 수행

- O(n3)

- k가 1에서 n이 될 때 까지 k를 거쳐서 가는 최단 경로를 계산해서, 모든 점을 경유 가능한 점들로 고려된 모든 쌍 i, j의 최단 경로의 거리를 찾는 방식

- 원소끼리 갈 수 없는 경우에는 인접행렬을 무한 행렬으로 갱신

- k = 1일 때 부터를 시작으로 k를 껴서 이동할 경우와 직접 갈 경우 중 적은 수로 갱신

ex) D[4,2] = min{D[4,2], D[4,1] + D[1,2]} 계산

public class Floyd {

static final int INF = 9999999;

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int N = sc.nextInt();

int[][] adjMatrix = new int[N][N];

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

adjMatrix[i][j] = sc.nextInt();

if(adjMatrix[i][j] == 0) adjMatrix[i][j] = INF;

}

}

// 출발지 --> 경유지 --> 목적지로 3증 루프 돌리면 오답

// 경유지 --> 출발지 --> 목적지로 3중 루프 돌려야 정답

for (int k = 0; k < N; ++k) {

for (int i = 0; i < N; i++) {

if (i == k) continue; // 출발지와 경유지가 같다면 다음 출발지

for (int j = 0; i < N; j++) {

if(i==j || k==j) continue; // 경유지와 목적지가 같거나 출발지가 곧 목적지라면 패스

if(adjMatrix[i][k] + adjMatrix[k][j] < adjMatrix[i][j]) {

adjMatrix[i][j] = adjMatrix[i][k] + adjMatrix[k][j];

}

}

}

}

}

}