// Ford - Bellman O(n\*m)

#include <iostream>

#include <vector>

using namespace std;

using ll = int64\_t;

ll INF = 1e18;

struct wedge

{

int u, v;

ll w;

};

using wgraph = vector<vector<wedge>>;

int main()

{

int n, m, start;

cin >> n >> m >> start;

--start;

// wgraph g(n);

vector<wedge> edges;

for (int i = 0; i < m; ++i)

{

int u, v;

ll w;

cin >> u >> v >> w;

--u;

--v;

edges.push\_back({u, v, w});

edges.push\_back({v, u, w});

}

bool flag = true;

vector<ll> dp(n, INF);

dp[start] = 0;

for (int i = 0; i < n; ++i)

{

flag = false;

for (wedge elem : edges)

{

int u = elem.u;

int v = elem.v;

ll w = elem.w;

if (dp[v] > dp[u] + w)

{

dp[v] = dp[u] + w;

flag = true;

}

}

if (!flag)

{

break;

}

}

// int e = edges.size() - 1;

// dp[e] = dp[e - 1] + edges[e - 1].w;

if (flag)

{

cout << "Negative cycle\n";

}

else

{

for (ll el : dp)

{

if (el == 1e18)

{

cout << -1 << ' ';

continue;

}

{

cout << el << ' ';

}

}

cout << endl;

}

}

// ------------------------------------------------------------------------------------------------------------------------------------

// djikstra\_slow O(n^2+m)

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

using ll = int64\_t;

ll INF = 1e18;

struct wedge

{

int u, v;

ll w;

};

using wgraph = vector<vector<wedge>>;

void djikstra\_slow(wgraph &g, const int u, vector<ll> &d, vector<int> &prev)

{

int n = g.size();

d[u] = 0;

prev[u] = u;

vector<char> visited(n, false);

for (int i = 0; i < n; ++i)

{

int min\_id = -1;

ll val = INF;

for (int j = 0; j < n; ++j)

{

if (!visited[j] and d[j] < val)

{

min\_id = j;

val = d[j];

}

}

int u = min\_id;

visited[u] = true;

for (wedge el : g[u])

{

int v = el.v;

ll w = el.w;

if (d[v] > d[u] + w)

{

d[v] = d[u] + w;

prev[v] = u;

}

}

}

}

int main()

{

int n, m, start, finish;

cin >> n >> m >> start >> finish;

--start;

--finish;

wgraph g(n);

for (int i = 0; i < m; ++i)

{

int u, v;

ll w;

cin >> u >> v >> w;

--u;

--v;

g[u].push\_back({u, v, w});

}

vector<ll> d(n, INF);

vector<int> prev(n, -1);

djikstra\_slow(g, start, d, prev);

vector<int> path;

int u=finish;

while(prev[u]!=u){

path.push\_back(u);

u = prev[u];

}

path.push\_back(u);

reverse(path.begin(), path.end());

for(int el: path){

cout << el+1 << ' ';

}

// for(int elem: d){cout << elem << ' ';}

cout << endl;

}

// -----------------------------------------------------------------------------------------------------

// djikstra\_fast O(n+mlogm)

#include <iostream>

#include <vector>

#include <algorithm>

#include <set>

using namespace std;

using ll = int64\_t;

ll INF = 1e18;

using item = pair<ll, int>;

struct wedge

{

int u, v;

ll w;

};

using wgraph = vector<vector<wedge>>;

void djikstra\_fast(wgraph &g, const int u, vector<ll> &d, vector<int> &prev)

{

int n = g.size();

d[u] = 0;

prev[u] = u;

set<item> pq;

for (int i = 0; i < n; ++i)

{

pq.insert({d[i], i});

}

vector<char> visited(n, false);

while (!pq.empty())

{

item el = \*pq.begin();

pq.erase(pq.begin());

int u = el.second;

visited[u] = true;

for (wedge elem : g[u])

{

int v = elem.v;

if (visited[v])

{

continue;

}

ll w = elem.w;

if (d[u] + w < d[v])

{ pq.erase({d[v], v});

d[v] = d[u] + w;

prev[v] = u;

pq.insert({d[v], v});

}

}

}

}

int main()

{

int n, m, start, finish;

cin >> n >> m >> start >> finish;

--start;

--finish;

wgraph g(n);

for (int i = 0; i < m; ++i)

{

int u, v;

ll w;

cin >> u >> v >> w;

--u;

--v;

g[u].push\_back({u, v, w});

}

vector<ll> d(n, INF);

vector<int> prev(n, -1);

djikstra\_fast(g, start, d, prev);

vector<int> path;

int u = finish;

while (prev[u] != u)

{

path.push\_back(u);

u = prev[u];

}

path.push\_back(u);

reverse(path.begin(), path.end());

for (int el : path)

{

cout << el + 1 << ' ';

}

// for(int elem: d){cout << elem << ' ';}

cout << endl;

}

// Floyd Warshell O(n^3)

#include <iostream>

#include <vector>

using ll=int64\_t;

using namespace std;

const ll INF = 1e18;

using matrix = vector<vector<ll>>;

int main(){

int n, m;

cin>>n>>m;

matrix g(n, vector<ll>(n, INF));

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

int u,v;

ll w;

cin>>u>>v>>w;

--u;

--v;

g[u][v]=w;

}

matrix d(g);

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

d[i][i]=0;

}

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

for(int u=0;u<n;++u){

for(int v=0;v<n;++v){

d[u][v]= min(d[u][v],d[u][k]+d[k][v]);

}

}

}

for(int u=0;u<n;++u){

for(int v=0;v<n;++v){

cout<<d[u][v]<<" ";

}

cout<<endl;

}

}

// Min spanning tree MST O((n+m)logn) Прима

#include <iostream>

#include <vector>

#include <algorithm>

#include <set>

using namespace std;

using ll = int64\_t;

ll INF = 1e18;

using item = pair<ll, int>;

struct wedge

{

int u, v;

ll w;

};

using wgraph = vector<vector<wedge>>;

void djikstra\_fast(wgraph &g, const int u, vector<ll> &d, vector<int> &prev)

{

int n = g.size();

d[u] = 0;

prev[u] = u;

set<item> pq;

for (int i = 0; i < n; ++i)

{

pq.insert({d[i], i});

}

vector<char> visited(n, false);

while (!pq.empty())

{

item el = \*pq.begin();

pq.erase(pq.begin());

int u = el.second;

visited[u] = true;

for (wedge elem : g[u])

{

int v = elem.v;

if (visited[v])

{

continue;

}

ll w = elem.w;

if (w < d[v])

{

pq.erase({d[v], v});

d[v] = w;

prev[v] = u;

pq.insert({d[v], v});

}

}

}

}

int main()

{

ios::sync\_with\_stdio(false);

cin.tie(0);

int n, m, start;

cin >> n >> m >> start;

--start;

wgraph g(n);

for (int i = 0; i < m; ++i)

{

int u, v;

ll w;

cin >> u >> v >> w;

--u;

--v;

g[u].push\_back({u, v, w});

g[v].push\_back({v, u, w});

}

vector<ll> d(n, INF);

vector<int> prev(n, -1);

djikstra\_fast(g, start, d, prev);

for (ll elem : d)

{

if (elem != INF)

{

cout << elem << ' ';

}

else

{

cout << -1 << ' ';

}

}

cout << endl;

}

// ----------------------------------------------------------------------------------------------------------

// Kruskal, O(mlogm+m)

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

using ll = int64\_t;

struct wedge

{

int u, v;

ll w;

};

bool operator<(wedge a, wedge b)

{

return a.w < b.w;

}

struct dsu

{

vector<int> leader;

vector<int> size;

int n;

dsu(int other\_n)

{

n = other\_n;

leader.resize(n);

for (int i = 0; i < n; ++i)

{

leader[i] = i;

}

size.resize(n, 1);

}

~dsu() = default;

int find(int u)

{

if (leader[u] == u)

{

return u;

}

leader[u] = find(leader[u]);

return leader[u];

}

bool unite(int u, int v)

{

int lu = find(u);

int lv = find(v);

if (lu == lv)

{

return false;

}

if (size[lu] > size[lv])

{

swap(lu, lv);

}

leader[lu] = lv;

size[lv] += size[lu];

return true;

}

};

int main()

{

ios::sync\_with\_stdio(false);

cin.tie(0);

int n, m;

cin >> n >> m;

vector<wedge> edges;

for (int i = 0; i < m; ++i)

{

int u, v;

ll w;

cin >> u >> v >> w;

--u; --v;

edges.push\_back({u, v, w});

}

sort(edges.begin(), edges.end());

dsu d(n);

ll w\_total = 0;

for(wedge elem: edges){

int u = elem.u;

int v = elem.v;

if(!d.unite(u,v)){

continue;

}

ll w = elem.w;

w\_total += w;

}

cout << w\_total << endl;

}

// Наименьший общий предок, дерево

#include <iostream>

#include <vector>

using namespace std;

struct edge

{

int u, v;

};

using graph = vector<vector<edge>>;

void dfs(int u, int p, const graph &g, vector<int> &d)

{

for (edge elem : g[u])

{

int v = elem.v;

if (v == p)

{

continue;

}

d[v] = 1 + d[u];

dfs(v, u, g, d);

}

}

int main()

{

int n;

cin >> n;

graph g(n);

for (int i = 0; i < n - 1; ++i)

{

int u, v;

cin >> u >> v;

--u;

--v;

g[u].push\_back({u, v});

g[v].push\_back({v, u});

}

int u = 0;

vector<int> d1(n);

dfs(u, u, g, d1);

int v = 0;

for (int i = 0; i < n; ++i)

{

if (d1[i] > d1[v])

{

v = i;

}

}

vector<int> d2(n);

dfs(v, v, g, d2);

for (int i = 0; i < n; ++i)

{

if (d2[i] > d2[u])

{

u = i;

}

}

cout << d2[u] << " "

<< "u = " << u + 1 << " "

<< "v = " << v + 1<< endl;

}

// -------------------------------------------------------------------------------------------------------------------

#include <iostream>

#include <vector>

using namespace std;

const int MAX\_D = 20;

struct edge

{

int u, v;

};

using graph = vector<vector<edge>>;

struct lca

{

vector<vector<int>> up;

vector<int> d;

// vector<vector<long long>> data;

int n;

void dfs(int u, int p, const graph &g)

{

for (edge elem : g[u])

{

int v = elem.v;

if (v == p)

{

continue;

}

d[v] = 1 + d[u];

up[v][0] = u;

dfs(v, u, g);

}

}

lca(const graph &g, int root)

{

int n = g.size();

up.resize(n, vector<int>(MAX\_D));

d.resize(n);

// data.resize();

up[root][0] = root;

dfs(root, root, g);

for (int j = 1; j < MAX\_D; ++j)

{

for (int u = 0; u < n; ++u)

{

up[u][j] = up[up[u][j - 1]][j - 1];

}

}

}

int find(int u, int v)

{

if (d[u] > d[v])

{

swap(u, v);

}

int delta = d[v] - d[u];

for (int j = MAX\_D - 1; j >= 0; --j)

{

if (delta & (1 << j))

// w+=data[v][j];

v = up[v][j];

}

if (u == v)

{

return u;

}

for (int j = MAX\_D - 1; j >= 0; --j)

{

if (up[u][j] != up[v][j])

{

// w+=data[u][j];

// w+=data[v][j];

u = up[u][j];

v = up[v][j];

}

}

// return w+data[v][0]+data[u][0];

return up[u][0];

}

int path\_len(int u, int v)

{

if (d[u] > d[v])

{

swap(u, v);

}

int delta = d[v] - d[u];

int res = delta;

for (int j = MAX\_D - 1; j >= 0; --j)

{

if (delta & (1 << j))

v = up[v][j];

}

if (u == v)

{

return res;

}

for (int j = MAX\_D - 1; j >= 0; --j)

{

if (up[u][j] != up[v][j])

{

u = up[u][j];

v = up[v][j];

res += 2 \* (1 << j);

}

}

return res + 2;

}

void leaf(int p)

{

up.push\_back(vector<int>(MAX\_D));

up[n][0] = p;

d.push\_back(d[p] + 1);

for(int j=1;j<MAX\_D; ++j){

up[n][j]= up[up[n][j-1]][j-1];

}

++n;

}

};

int main()

{

int n;

cin >> n;

graph g(n);

for (int i = 0; i < n - 1; ++i)

{

int u, v;

cin >> u >> v;

--u;

--v;

g[u].push\_back({u, v});

g[v].push\_back({v, u});

}

int root = 4;

lca item(g, root - 1);

int q;

cin >> q;

while (q--)

{

int u, v;

cin >> u >> v;

--u;

--v;

cout << item.find(u, v) + 1 << ' ';

}

}

// Потоки

// Kuhn паросочетания в двудольном графе, выводит пары O(n\*m)

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

using graph = vector<vector<int>>;

vector<int> bfs(int u, graph &g)

{

int n = g.size();

vector<int> d(n, -1);

d[u] = 0;

queue<int> q;

q.push(u);

while (!q.empty())

{

u = q.front();

q.pop();

for (int v : g[u])

{

if (d[v] == -1)

{

d[v] = 1 + d[u];

q.push(v);

}

}

}

return d;

}

bool dfs(int u, const graph &g, vector<int> &mt, vector<int> visited)

{

if (visited[u])

{

return false;

}

visited[u] = true;

for (int v : g[u])

{

if (mt[v] == -1)

{

mt[v] = u;

return true;

}

bool tmp = dfs(mt[v], g, mt, visited);

if (tmp)

{

mt[v] = u;

return true;

}

}

return false;

}

int main()

{

ios::sync\_with\_stdio(false);

cin.tie(0);

int n, m;

cin >> n >> m;

graph g(n);

for (int i = 0; i < m; ++i)

{

int u, v;

cin >> u >> v;

--u;

--v;

g[u].push\_back(v);

g[v].push\_back(u);

}

vector<int> d = bfs(0, g);

int ans = 0;

vector<int> mt(n, -1);

for (int i = 0; i < n; ++i)

{

if (d[i] & 1)

{

continue;

}

vector<int> visited(n, false);

ans += dfs(i, g, mt, visited);

}

for (int i = 0; i < n; ++i)

{

if (mt[i] == -1)

{

continue;

}

cout << i + 1 << " " << mt[i] + 1 << endl;

}

}

---------------------------------------------------------------------------

// Dinic O(m\*sqrt(n))

#include <iostream>

#include <vector>

#include <queue>

using ll = int64\_t;

const ll INF = 1e18;

using namespace std;

struct network

{

struct edge

{

int u, v;

ll f, c;

edge(int \_u, int \_v, ll \_c) : u(\_u), v(\_v), f(0), c(\_c) {}

};

int n;

vector<edge> edges;

vector<vector<int>> g;

vector<int> d;

vector<size\_t> ptr;

network(int \_n) : n(\_n), g(n) {}

void add\_edge(int u, int v, ll c)

{

g[u].push\_back(edges.size());

edges.push\_back(edge(u, v, c));

g[v].push\_back(edges.size());

edges.push\_back(edge(v, u, 0));

}

friend ostream &operator<<(ostream &out, const network &g)

{

int m = g.edges.size();

for (int i = 0; i < m; i += 2)

{

edge elem = g.edges[i];

cout << elem.u + 1 << ' ' << elem.v + 1 << ' ' << elem.f << "/" << elem.c << endl;

}

return out;

}

void bfs(int u, ll b)

{

d.assign(n, -1);

queue<int> q;

q.push(u);

d[u] = 0;

while (!q.empty())

{

u = q.front();

q.pop();

for (int ind : g[u])

{

ll c = edges[ind].c;

ll f = edges[ind].f;

if (c - f < b)

{

continue;

}

int v = edges[ind].v;

if (d[v] == -1)

{

d[v] = d[u] + 1;

q.push(v);

}

}

}

}

ll dfs(int u, int finish, ll b, ll fp)

{

if (u == finish)

{

return fp;

}

for (; ptr[u] < g.size(); ++ptr[u])

{

int ind = g[u][ptr[u]];

ll c = edges[ind].c;

ll f = edges[ind].f;

if (c - f < b)

{

continue;

}

int v = edges[ind].v;

if (d[v] != d[u] + 1)

{

continue;

}

ll tmp = dfs(v, finish, b, min(c - f, fp));

if (tmp > 0)

{

edges[ind].f += tmp;

edges[ind ^ 1].f = tmp;

return tmp;

}

}

return 0;

}

ll dinic(int start, int finish, ll b)

{

ll res = 0;

while (1)

{

bfs(start, b);

if (d[finish] == -1)

{

break;

}

ptr.assign(n, 0);

while (1)

{

ll f = dfs(start, finish, b, INF);

if (f > 0)

{

res += f;

}

else

{

break;

}

}

}

return res;

}

ll max\_flow(int start, int finish)

{

d.clear();

d.resize(n, -1);

ptr.clear();

ptr.resize(n, 0);

ll res = 0;

for (int pow = 30; pow >= 0; --pow)

{

res += dinic(start, finish, 1ll << pow);

}

return res;

}

};

int main()

{

ios::sync\_with\_stdio(false);

cin.tie(0);

int n, m;

cin >> n >> m;

network g(n);

for (int i = 0; i < m; ++i)

{

int u, v;

ll c;

cin >> u >> v >> c;

--u, --v;

g.add\_edge(u, v, c);

}

int start = 1;

int finish = 8;

ll ans = g.max\_flow(start - 1, finish - 1);

cout << ans << endl;

cout << g << endl;

}