Graph

Created	@2025년 5월 17일 오후 6:36
Tags	BM DSU by size LCA OfDC SCC bellman dijk floyd kruskal prim

Floyd-Warshall's Algorithm

```
//Usage: auto [negative cycle, distance]=floyd(V, adj);
//O(V^3)
pair<bool,vvl> floyd(ll n, vector<vector<pll>>& adj){
    bool cycle = 0;
    const ll INF = 1e18;
    vvl dis(n,vector<ll>(n, INF));
    for (ll i=0;i<n;++i) dis[i][i] = 0;

    for (ll u=0;u<n;++u)
        for (auto& [v,w]:adj[u])
        dis[u][v]=min(dis[u][v],w); //multi-edges?
    for (ll k=0;k<n;++k)
        for (ll i=0;i<n;++i)
            for (ll j=0;j<n;++j)
                dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j]);
    for (ll k=0;k<n;++k) // Check negative cycle
        for (ll i=0;i<n;++i)
            for (ll j=0;j<n;++j)
                if (dis[i][j]>dis[i][k]+dis[k][j]) cycle=1;
    return {!cycle, dis};
}
```

Dijkstra's Algorithm

```
//Usage: vl distance = dijk(V, start, adj);
//O(ElogV)
vl dijk(ll n, ll s, vector<vector<pll>>& adj){
    const ll INF = 1e18;
    vl dis(n,INF);
    vector<bool> visit(n, false);
    priority_queue<pll, vector<pll>, greater<pll> > q; // pair(dist, v)
    dis[s] = 0;
    q.push({dis[s], s});
    while (!q.empty()){
        while (!q.empty() && visit[q.top().second]) q.pop();
        if (q.empty()) break;
        ll next=q.top().second; q.pop();
        visit[next]=1;
        for (ll i=0;i<adj[next].size();++i)
            if (dis[adj[next][i].first] > dis[next] + adj[next][i].second){
                dis[adj[next][i].first] = dis[next] + adj[next][i].second;
                q.push({dis[adj[next][i].first], adj[next][i].first});}}
    for(ll i=0;i<n;i++) if(dis[i]==INF) dis[i]=-1;
    return dis;
}
```

Bellman-Ford Algorithm

```
//Usage: auto [negative cycle, distance] = bellman(V, start, adj);
//O(VE)
pair<bool,vl> bellman(ll n, ll s, vector<vector<pll>>& adj){
    bool cycle=0;
    const ll INF=1e18;
    vector<ll>dis(n,INF);
    dis[s]=0;
    for (ll i=0;i<n;++i)
        for (ll j=0;j<n;++j)
            for (ll k=0;k<adj[j].size();++k){
                ll next=adj[j][k].first;
                ll cost=adj[j][k].second;
                if (dis[j]!=INF && dis[next]>dis[j]+cost) {
                    dis[next]=dis[j]+cost;
                    if (i==n-1) cycle=1;
                }
            }
    return {!cycle, dis};
}
```

Prim Algorithm

```
//Usage: ll mst = prim(V, adj);
// O(ElogV)
ll prim(ll n,vector<vector<pll>>& adj) {
    vector<bool> visit(n, false);
    priority_queue<pll, vector<pll>, greater<pll> > q;
    ll count=0; ll ret=0;
    q.push(make_pair(0, 0)); // (cost, vertex)
    while (!q.empty()){
        ll x=q.top().second; // also able to get edges
        visit[x]=1; ret+=q.top().first; q.pop(); count++;
        for (ll i=0;i<adj[x].size();++i)
            q.push({adj[x][i].second, adj[x][i].first});
        while (!q.empty() && visit[q.top().second]) q.pop();
    }
    if (count!=n) return -1;
    else return ret;
}
```

Kruskal's Algorithm

```
//Usage: ll mst = kruskal(V, adj);
// O(ElogE)
ll kruskal(ll n,vector<vector<pll>>& adj){
    DSU dsu(n);
    ll ret = 0;
    vector<pair<ll, pll>> e;
    for(ll i= 0; i < n; i++)
        for(ll j=0; j < adj[i].size(); j++)
            e.push_back({adj[i][j].second, {i, adj[i][j].first}});
    sort(e.begin(), e.end());
    for(ll i=0; i < e.size(); i++){
        ll x = e[i].second.first,y = e[i].second.second;
        if(dsu.find(x) != dsu.find(y)){
            dsu._union(x, y);
            ret += e[i].first;
        }
    }
    ll p=dsu.find(0);
    for(ll i=1;i<n;i++){
        if(dsu.find(i)!=p) return -1;
    }
    return ret;
}
```

DSU by size

```
//Usage: DSU dsu(V); ll root = dsu._find(node); dsu._union(node,node);
// O(alpha(V))
struct DSU {
    vl par, sz;

    DSU(ll n) {
        par.resize(n+1);
        sz.assign(n+1,1);
        iota(par.begin(),par.end(),0);
    }
    ll _find(ll x) {
        if (par[x]==x) return x;
        return par[x]=_find(par[x]);
        //for RollBack
        //return _find(par[x]);
    }
    pll _union(ll x,ll y){
        x=_find(x);
        y=_find(y);
        if (x==y) return {-1,-1};
        if (sz[x]<sz[y]) swap(x,y);
        par[y]=x;
        sz[x]+=sz[y];
        return {x,y};
    }
    void _delete(ll x, ll y){
        sz[x]-=sz[y];
        par[y]=y;
    }
};
```

LCA

```
// Usage: LCA lca(V,tree); ll anc = lca.solve(u,v);
// O(logV)
// memo : 0-indexed

struct LCA {
    ll MAXLN;
    vl depth; vvl anc;

    LCA(ll n, vvl& tree){
        ll root = 0;
        depth.assign(n,0);
        MAXLN=1;
        while ((1<<MAXLN)<=n) ++MAXLN;
        anc.assign(MAXLN,vl(n));

        function<void(ll,ll)> dfs4lca = [&](ll node,ll parent) {
            for (ll next: tree[node]) {
                if (next==parent) continue;
                depth[next]=depth[node]+1;
                anc[0][next]=node;
                dfs4lca(next, node);
            }
        };

        dfs4lca(root,-1);
        anc[0][root]=root;
        for (ll i=1;i<MAXLN;++i)
            for (ll j=0;j<n;++j)
                anc[i][j]=anc[i-1][anc[i-1][j]];
    }

    ll solve(ll u, ll v){
        if (depth[u]<depth[v]) swap(u, v);
        if (depth[u]>depth[v]) {
            for (ll i=MAXLN-1;i>=0;--i)
                if (depth[u]-(1<<i) >= depth[v])
                    u=anc[i][u];
        }
        if (u==v) return u;
        for (ll i=MAXLN-1;i>=0;--i) {
            if (anc[i][u]!=anc[i][v]) {
                u=anc[i][u];
                v=anc[i][v];
            }
        }
        return anc[0][u];
    }
};
```

Bipartite Matching

```
// Usage :
// Constructor1 : BipartiteGraph bg(Lsize, Rsize); bg.add_edge(l_node, r_no
de);
// Constructor2 : BipartiteGraph bㄹg(vvl Adj);
// Maximum matching : bg.maximum_matching();
// O(E*sqrt(V))
struct BipartiteGraph {
   ll n; vl color; vvl adj;
   bool is_bipartite = true;
   ll leftSz, rightSz;
   vl leftNodes, rightNodes;   // partition idx → original idx
   vl leftId, rightId;         // original idx → partition idx
   vvl adjL;                   // left index → [right indices...]
   // maximum matching
   ll matching, distNil, INF = 1e10;
   vl pairL, pairR, dist;

   // 엣지를 바로 입력
   BipartiteGraph(ll L, ll R)
      : leftSz(L), rightSz(R), adjL(L) {}

   void add_edge(ll l, ll r) {
      adjL[l].emplace_back(r);
   }

   void erase_edge(ll ln, ll rn) {  // adjL 정렬 후 사용
      ll l = leftId[ln], r = rightId[rn];
      adjL[l].erase(lower_bound(adjL[l].begin(), adjL[l].end(), r));
   }

   // 일반 그래프 → 이분 그래프 변환
   BipartiteGraph(const vvl& G)
    : n(G.size()), adj(G), color(n, 0)
   {
      for (ll i=0; i<n; ++i) {    // 0-based index
```

```
        if (color[i] == 0 && !bfs_color(i)) {
            is_bipartite = false;
            return;
        }
    }
    build_bipartite();     // 좌우 파티션 노드 분리, 매핑, 인접 리스트 생성
}

bool bfs_color(ll s) {
    queue<ll> q;
    color[s] = 1;
    q.push(s);
    while (!q.empty()) {
        ll node = q.front(); q.pop();
        for (auto& nxt : adj[node]) {
            if (color[nxt] == 0) {
                color[nxt] = -color[node];
                q.push(nxt);
            } else if (color[nxt] == color[node])
                return false;
        }
    }
    return true;
}

void build_bipartite() {
    leftId.assign(n, -1);
    rightId.assign(n, -1);
    for (ll i=0; i<n; ++i) {
        if (color[i] == 1) {
            leftId[i] = leftNodes.size();
            leftNodes.push_back(i);
        } else {
            rightId[i] = rightNodes.size();
            rightNodes.push_back(i);
        }
    }
    leftSz = leftNodes.size();
    rightSz = rightNodes.size();
    adjL.assign(leftSz, {});
    for (ll l=0; l<leftSz; ++l)
        for (auto& r : adj[leftNodes[l]])
            adjL[l].emplace_back(rightId[r]);
}

// maximum matching: Hopcroft-Karp
ll max_matching() {
    matching = 0;
    pairL.assign(leftSz, -1);
    pairR.assign(rightSz, -1);
    dist.assign(leftSz, 0);
    while (bfs_HK())    // augmenting path 존재하는 동안 반복
        for (ll l=0; l<leftSz; ++l)
            if (pairL[l]==-1 && dfs_HK(l))
                matching++;
    return matching;
}

bool bfs_HK() {    // 가장 짧은 augmenting path 찾음
    queue<ll> q;
    for (ll l=0; l<leftSz; l++) {
        if (pairL[l] == -1) {
            dist[l] = 0;
            q.emplace(l);
        } else dist[l] = INF;
    }
    distNil = INF;
    while (!q.empty()) {
        ll l = q.front(); q.pop();
        if (dist[l] < distNil) {
            for (auto& r : adjL[l]) {
                ll pl = pairR[r];
                if (pl != -1) {
                    if (dist[pl] == INF) {
                        dist[pl] = dist[l] + 1;
                        q.emplace(pl);
                }
            } else distNil = dist[l] + 1;
        }
    }
   }
   return distNil != INF;
  }

  bool dfs_HK(ll l) {
   for (ll r : adjL[l]) {
    ll pl = pairR[r];
    if (pl==-1 || (dist[pl]==dist[l]+1 && dfs_HK(pl))) {
     pairL[l] = r;
     pairR[r] = l;
     return true;
    }
   }
   dist[l] = INF;
   return false;
  }
};
```

```
// Usage: BipartiteMatching bm(leftV,rightV,graph);
//        ll ans = bm.max_matching; vl LtoR=bm.match; vl RtoL=bm.matched;
// O(E*sqrt(V))
// memo: vertex cover: (reached[0][left_node] == 0) || (reached[1][right_no
de] == 1)
struct BM {
  ll n, m, max_matching;
  vvl graph;
  vl matched, match, edgeview, level;
  vl reached[2];
  BM(ll n, ll m, vvl& graph) : n(n), m(m), graph(graph), matched(m,-1), matc
h(n,-1) {
    ll max_matching = 0;
    while (assignLevel()) {
      edgeview.assign(n, 0);
      for (ll i = 0; i < n; i++)
```

```
            if (match[i]==-1)
                max_matching += findpath(i);
        }
    }
    bool assignLevel(){
        bool reachable = false;
        level.assign(n,-1);
        reached[0].assign(n, 0);
        reached[1].assign(m, 0);
        queue<ll> q;
        for (ll i = 0; i < n; i++) {
            if (match[i] ==-1) {
                level[i] = 0;
                reached[0][i] = 1;
                q.push(i);
            }
        }
        while (!q.empty()) {
            auto cur = q.front(); q.pop();
            for (auto adj : graph[cur]) {
                reached[1][adj] = 1;
                auto next = matched[adj];
                if (next ==-1) {
                    reachable = true;
                }
                else if (level[next] ==-1) {
                    level[next] = level[cur] + 1;
                    reached[0][next] = 1;
                    q.push(next);
                }
            }
        }
        return reachable;
    }
    ll findpath(ll node){
        for (ll &i = edgeview[node]; i < graph[node].size(); i++) {
            ll adj = graph[node][i];
            ll next = matched[adj];
            if (next >= 0 && level[next] != level[node] + 1) continue;
            if (next ==-1 || findpath(next)) {
                match[node] = adj;
                matched[adj] = node;
                return 1;
            }
        }
        return 0;
    };
};
```

SCC

```
//Usage: SCC scc(V, graph); vl component = scc.scc_idx;
// O(V+E)
// memo: the order of scc_idx constitutes a reverse topological sort
struct SCC {
    ll n,vtime,scc_cnt;
    vvl graph;
    vl up, visit,scc_idx,stk;

    SCC(ll n, vvl& graph):
    n(n),graph(graph),up(n),visit(n,0),scc_idx(n,0),vtime(0),scc_cnt(0) {
        for (ll i=0;i<n;++i)
            if (visit[i]==0) dfs(i);
    }

    void dfs(ll node){
        up[node] = visit[node] = ++vtime;
        stk.push_back(node);
        for (ll next : graph[node]){
            if (visit[next] == 0) {
                dfs(next);
                up[node] = min(up[node], up[next]);
            }
            else if (scc_idx[next] == 0)
                up[node] = min(up[node], visit[next]);
        }
        if (up[node]==visit[node]){
            ++scc_cnt;
            ll t;
            do{
                t = stk.back();
                stk.pop_back();
                scc_idx[t] = scc_cnt;
            } while (!stk.empty() && t != node);
        }
    }
};
```

OFDC

```
//Usage: vector<tlll> query; OFDC ofdc(V, #query, query);
//O(QlogQ * alpha(V))
struct OFDC{
    vector<tlll>query;
    vector<vector<pll>> tree;
    map<pll,ll>connected_time;
    ll n, q; vl ans;
    DSU dsu;

    OFDC(ll n, ll q,vector<tlll>&query): n(n), q(q), query(query), tree(4*(q+
1)), dsu(n+1) {
        for(ll i=0;i<q;i++){
            auto&[type,u,v]=query[i];
            if(u>v)swap(u,v);
            if(type==1) connected_time[{u,v}]=i; //union
            else if(type==2){ //delete
                update(1,0,q,connected_time[{u,v}],i,{u,v});
                connected_time.erase({u,v});
            }
        }
        for(auto&[edge,time]:connected_time){
            auto&[u,v]=edge;
            update(1,0,q,time,q,{u,v});
        }
        dfs(1,0,q);
    }

    void update(ll node, ll s, ll e, ll l, ll r, pll edge){
        if(r<s||e<l) return;
        if(l<=s&&e<=r){
            tree[node].pb(edge);
            return;
        }
        ll mid=(s+e)>>1;
        update(node<<1,s,mid,l,r,edge);
        update(node<<1|1,mid+1,e,l,r,edge);
    }

    void dfs(ll node, ll s, ll e){
        vector<pll>real_connected;
        for(auto&[u,v]:tree[node]){
            auto [x,y]=dsu._union(u,v);
            if(x!=-1) real_connected.push_back({x,y});
        }
        if(s==e){
            if(get<0>(query[s])==3){ //connect?
                ans.pb((dsu._find(get<1>(query[s]))==dsu._find(get<2>(query
[s]))));
            }
        }
        else{
            ll mid = (s+e)>>1;
            dfs(node<<1, s, mid);
            dfs(node<<1|1, mid+1, e);
        }
        reverse(all(real_connected));
        for(auto&[x,y]:real_connected) dsu._delete(x,y);
    }
};
```

HLD

```
// Usage: auto [sz, dep, par, in, out, top] = get_hld(adj);
// Time Complexity: O(V)
// Memo: 1-indexed
tuple<vl, vl, vl, vl, vl, vl> get_hld(vvl adj) {
    ll n = adj.size() - 1;
    vl sz(n+1, 1), dep(n+1), par(n+1);
    vl in(n+1), out(n+1), top(n+1);
    ll ord = 0;

    auto dfs1 = [&](auto& self, ll cur, ll prv){
        if (prv) adj[cur].erase(ranges::find(adj[cur], prv));
        for (ll &nxt : adj[cur]) {
            dep[nxt] = dep[cur] + 1;
            par[nxt] = cur;
            self(self, nxt, cur);
            sz[cur] += sz[nxt];
            if (sz[adj[cur][0]] < sz[nxt]) swap(adj[cur][0], nxt);
        }
    };

    auto dfs2 = [&](auto& self, ll cur){
        in[cur] = ++ord;
        for (ll nxt : adj[cur]) {
            top[nxt] = (nxt == adj[cur][0] ? top[cur] : nxt);
            self(self, nxt);
        }
        out[cur] = ord;
    };

    dfs1(dfs1, 1, 0);
    dfs2(dfs2, top[1] = 1);
    return {sz, dep, par, in, out, top};
}
```

Tree Isomorphism

```
// Usage: TreeIsomorphism ti(T1, T2) or ti(T1, r1, T2, r2)
//        bool isIso = ti.isIsomorphic
//        ti.isTreeIsomorphicMap ⇒ ti.mapping[i]=j := T1의 i노드는 T2의 j노드와 
대응
// O(NlogN)
struct TreeIsomorphism {
    ll id = 1;
    vl root1, root2, mapping;
    vvl tree1, tree2;
    map<vl,ll> isoClass;

    TreeIsomorphism(const vvl& T1, const vvl& T2)
        : tree1(T1), tree2(T2) {
        root1 = findCenter(T1);
        root2 = findCenter(T2);
    }
    TreeIsomorphism(const vvl& T1, ll r1, const vvl& T2, ll r2)
        : tree1(T1), tree2(T2), root1({r1}), root2({r2}) {}

    vector<ll> findCenter(const vvl &tree) {
        ll n = tree.size();
        vl degree(n), leaves;
        for (ll i=0; i<n; i++) {
            degree[i] = tree[i].size();
            if (degree[i] <= 1)
                leaves.emplace_back(i);
        }
        ll removed = leaves.size();
        while (removed < n) {
            vl newLeaves;
            for (ll u : leaves)
                for (ll v : tree[u])
                    if (--degree[v] == 1)
                        newLeaves.push_back(v);
            removed += newLeaves.size();
            leaves = move(newLeaves);
        }
        return leaves;
    }

    ll getID(const vvl& tree, ll node, ll pa) {
        vl childID;
        for (auto& ch : tree[node])
            if (ch != pa)
                childID.emplace_back(getID(tree, ch, node));
        sort(childID.begin(), childID.end());
        if (!isoClass.contains(childID))
            isoClass[childID] = id++;
        return isoClass[childID];
    }

    bool isTreeIsomorphic() {
        if (tree1.size() <= 1 || tree2.size() <= 1)
            return tree1.size() == tree2.size();
        ll id1 = getID(tree1, root1[0], -1);
        for (auto& r : root2)
            if (id1 == getID(tree2, r, -1))
                return true;
        return false;
    }

    void mapSubtree(ll node1, ll pa1, ll node2, ll pa2) {
        mapping[node1] = node2;
        vector<pll> ch1, ch2;
        for (auto& ch : tree1[node1])
            if (ch != pa1)
                ch1.emplace_back(getID(tree1, ch, node1), ch);
        for (auto& ch : tree2[node2])
            if (ch != pa2)
                ch2.emplace_back(getID(tree2, ch, node2), ch);
        sort(ch1.begin(), ch1.end());
        sort(ch2.begin(), ch2.end());
        for (ll i=0; i<ch1.size(); i++)
            mapSubtree(ch1[i].second, node1, ch2[i].second, node2);
    }
```

```
    bool isTreeIsomorphicMap() {
        if (tree1.size() <= 1 || tree2.size() <= 1) {
            if (tree1.size() != tree2.size()) return false;
            mapping.assign(tree1.size(), 0);
            return true;
        }
        ll id1 = getID(tree1, root1[0], -1);
        for (auto& r : root2) {
            if (id1 == getID(tree2, r, -1)) {
                mapping.assign(tree1.size(), -1);
                mapSubtree(root1[0], -1, r, -1);
                return true;
            }
        }
        return false;
    }
};
```

Maximum Flow (Dinic)

```
// Usage : DINIC flow(#node)
//        flow.add_edge(start, end, capacity)
//        ans = flow.solve(source, sink)
// O(V^2*E)
struct DINIC {
    struct Edge { ll nxt, rev, res; };
    ll n; vl level, start;
    vector<vector<Edge>> graph;
    DINIC(ll _n): n(_n), graph(n), level(n), start(n) {}
    void add_edge(ll s, ll e, ll cap, ll rev_cap = 0) {
        graph[s].push_back({e, (ll)graph[e].size(), cap});
        graph[e].push_back({s, (ll)graph[s].size() - 1, rev_cap});
    }
    bool assign_level(ll src, ll sink) {
        fill(level.begin(), level.end(), -1);
        queue<ll> q;
        level[src] = 0; q.emplace(src);
        while (!q.empty()) {
            ll cur = q.front(); q.pop();
            for (auto& [nxt,rev,res] : graph[cur]) {
                if (level[nxt]==-1 && res>0) {
                    level[nxt] = level[cur] + 1;
                    q.emplace(nxt);
                }
            }
        }
        return level[sink] != -1;
    }
    ll block_flow(ll cur, ll sink, ll flow) {
        if (cur==sink) return flow;
        for (ll& i=start[cur]; i<graph[cur].size(); i++) {
            auto& [nxt,rev,res] = graph[cur][i];
            if (res>0 && level[nxt]==level[cur]+1) {
                ll pushed = block_flow(nxt, sink, min(flow, res));
                if (pushed > 0) {
                    res -= pushed;
                    graph[nxt][rev].res += pushed;
                    return pushed;
                }
            }
        }
        return 0;
    }
    ll solve(ll src, ll sink) {
        ll total=0;
        while (assign_level(src, sink)) {
            fill(start.begin(), start.end(), 0);
            while (ll pushed = block_flow(src, sink, LLONG_MAX)) {
                total += pushed;
            }
        }
        return total;
    }
};
```

Min-Cost Maximum Flow (SSAP)

```
// Usage : MCMF flow(#node)
//        flow.add_edge(start, end, capacity, cost)
//        [maxFlow, minCost] = flow.solve(source, sink, [한 번에 흘릴 수 있는 최
대 유량])
// O(VE + F·ElogV)  **F:=증강 횟수
struct MCMF {
    struct Edge { ll nxt, rev, res, cost; };
    ll n;
    vector<vector<Edge>> g;
    MCMF(ll n): n(n), g(n) {}

    void add_edge(ll s, ll e, ll cap, ll cost, ll rev_cap = 0){
        g[s].emplace_back(e, (ll)g[e].size(), cap, cost);
        g[e].emplace_back(s, (ll)g[s].size()-1, rev_cap, -cost);
    }

    // s→t로 최대 maxf만큼 보냄(기본: 무한). (flow, cost) 반환
    pll solve(ll src, ll sink, ll maxf = LLONG_MAX){
        const ll INF = LLONG_MAX;
        ll flow = 0, minCost = 0;

        vl pi(n, 0), dist(n), avail(n);
        vl pv(n), pe(n);    // prev vertex, prev edge idx

        // 초기 포텐셜: 음수비용 간선이 있을 수 있으면 SPFA/BF로 한 번 계산
        auto spfa_init = [&](){
            deque<ll> dq; vector<bool> inq(n,false);
            fill(pi.begin(), pi.end(), INF);
            pi[src] = 0; dq.push_back(src); inq[src] = true;
            while (!dq.empty()){
                ll cur = dq.front(); dq.pop_front(); inq[cur] = false;
                for (auto& [nxt,rev,res,cost] : g[cur])
                    if (res>0 && pi[nxt]>pi[cur]+cost) {
                        pi[nxt] = pi[cur] + cost;
                        if (!inq[nxt]) {
                            inq[nxt] = true;
                        if (!dq.empty() && pi[nxt]<pi[dq.front()]) dq.push_front(nx
t);
                        else dq.push_back(nxt);
                    }
                }
            }
            for (ll i=0; i<n; i++)
                if (pi[i] == INF) pi[i] = 0; // 도달불가는 0으로
        };
        spfa_init();

        while (flow < maxf){
            fill(dist.begin(), dist.end(), INF);
            fill(avail.begin(), avail.end(), 0);
            dist[src] = 0; avail[src] = INF;
            priority_queue<pll,vector<pll>,greater<pll>> pq;
            pq.emplace(0, src);
            while (!pq.empty()){
                auto [d,cur] = pq.top(); pq.pop();
                if (d != dist[cur]) continue;
                for (ll i=0; i<g[cur].size(); i++) {
                    auto& [nxt,rev,res,cost] = g[cur][i];
                    if (res <= 0) continue;
                    ll w = cost + pi[cur] - pi[nxt]; // 감소비용
                    if (dist[nxt] > dist[cur]+w){
                        dist[nxt] = dist[cur] + w;
                        pv[nxt] = cur; pe[nxt] = i;
                        avail[nxt] = min(avail[cur], res);
                        pq.push({dist[nxt], nxt});
                    }
                }
            }
            if(dist[sink] == INF) break; // 더 이상 증가경로 없음
            for(ll i=0; i<n; i++)
                if(dist[i] < INF) pi[i] += dist[i]; // 포텐셜 업데이트
            ll add = min(avail[sink], maxf-flow);
            flow += add;
            for (ll i=sink; i!=src; i=pv[i]) {
                auto& [nxt,rev,res,cost] = g[pv[i]][pe[i]];
                res -= add;
                g[i][rev].res += add;
                minCost += add*cost;
            }
        }
        return {flow, minCost};
    }
};
```