

Team Note of Junho0219

Kim Junho

Compiled on February 3, 2026

Contents

| | | |
|----------|--------------------------|----------|
| 1 | Data Structure | 1 |
| 1.1 | DSU Potential | 1 |
| 1.2 | Fenwick | 1 |
| 1.3 | Mergesort Tree | 1 |
| 1.4 | Segtree | 2 |
| 2 | Graph | 2 |
| 2.1 | Bellman-Ford | 2 |
| 2.2 | SPFA | 2 |
| 2.3 | SCC | 2 |
| 2.4 | Bimatch | 2 |
| 2.5 | Hopcroft Karp | 3 |
| 2.6 | MCMF | 3 |
| 2.7 | Dinic | 4 |
| 2.8 | Cycle Counting | 4 |
| 2.9 | Euler Circuit | 4 |
| 2.10 | PEO | 5 |
| 2.11 | theory | 5 |
| 3 | Math | 5 |
| 3.1 | XOR Basis | 5 |
| 3.2 | exgcd | 6 |
| 3.3 | CRT | 6 |
| 3.4 | Miller-Rabin | 6 |
| 3.5 | Pollard's rho | 6 |
| 4 | String | 6 |
| 4.1 | Z | 6 |
| 5 | Misc | 6 |
| 5.1 | RMQ | 6 |
| 5.2 | Random | 7 |
| 5.3 | Time | 7 |
| 5.4 | Debug | 7 |

1 Data Structure**1.1 DSU Potential**

```
template <typename T>
struct DSU {
    vector<int> parent;
    vector<T> weight;
    DSU(int n) : parent(n + 1), weight(n + 1) {
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int a) {
        if (parent[a] == a) return a;
        find(parent[a]); // calculate weight[parent[a]]
        weight[a] += weight[parent[a]];
        return parent[a] = find(parent[a]);
    }
    void merge(int a, int b, T w) { // b = a + w
        int ra = find(a);
        int rb = find(b);
        if (ra > rb) {
            swap(ra, rb);
            swap(a, b);
            w *= -1;
        }
        weight[rb] = weight[a] + w - weight[b];
        parent[rb] = ra;
    }
    bool isConnected(int a, int b) {
        return find(a) == find(b);
    }
    T getDiff(int a, int b) {
        assert(isConnected(a, b)); // 아니면 비교 불가
        return weight[b] - weight[a];
    }
};
```

1.2 Fenwick

```
template <typename T, int D>
struct FenwickTree {
    vector<FenwickTree<T, D - 1>> tree;

    template <typename... Args>
    FenwickTree(int n, Args... args) : tree(n + 1, FenwickTree<T, D - 1>(args...)) {}

    template <typename... Args>
    void update(int pos, Args... args) {
        for (; pos < tree.size(); pos += (pos & -pos))
            tree[pos].update(args...);
    }

    template <typename... Args>
    T sum(int r, Args... args) const {
        T res = 0;
        for (; r; r -= (r & -r)) res += tree[r].query(args...);
        return res;
    }

    template <typename... Args>
    T query(int l, int r, Args... args) const {
        return sum(r, args...) - sum(l - 1, args...);
    }

    template <typename T>
    struct FenwickTree<T, 0> {
        T val = 0;
        void update(T v) { val += v; }
        T query() const { return val; }
    };

    // ex) 3D fenwick
    // FenwickTree<int, 3> fw(n, m, k);
    // fw.update(x, y, z, add); // O(logN logM logK)
    // fw.query(x1, x2, y1, y2, z1, z2); // O(logN logM logK)
};

1.3 Mergesort Tree
template <typename T>
class MergesortTree {
private:
    int n;
    vector<vector<T>> tree;

    int count(const vector<T> &v, int k) {
        // ex)
        // return upper_bound(v.begin(), v.end(), k) -
        v.begin();
    }

    void init(const vector<T> &v, int node, int s, int e) {
        if (s == e) {
            tree[node].push_back(v[s]);
            return;
        }

        int m = s + e >> 1;
        init(v, node * 2, s, m);
        init(v, node * 2 + 1, m + 1, e);
        tree[node].resize(tree[node * 2].size() + tree[node * 2 + 1].size());
        merge(tree[node * 2].begin(), tree[node * 2].end(),
              tree[node * 2 + 1].begin(), tree[node * 2 + 1].end(),
              tree[node].begin());
    }

    int query(int node, int s, int e, int l, int r, int k) {
        if (l <= s && e <= r) return count(tree[node], k);
    }
```

```

    if (r < s || e < l) return 0;

    int m = s + e >> 1;
    return query(node * 2, s, m, l, r, k) + query(node * 2
+ 1, m + 1, e, l, r, k);
}

public:
MergesortTree(const vector<T> &v) : n(v.size()), tree(4 *
v.size()) {
    init(v, 1, 0, n - 1);
}

int query(int l, int r, int k) { // 1-based
    return query(1, 0, n - 1, l - 1, r - 1, k);
}
};

// time, space O(N~logN)
1.4 Segtree
template<typename T, T (*op)(const T&, const T&), T (*e)()>
class SegmentTree {
private:
    int n, sz, log;
    vector<T> tree;

    static int pow2GE(int n) { assert(n); return 1 << 32 -
__builtin_clz(n) - !(n & ~n); }

public:
SegmentTree() = default;
SegmentTree(int n) : SegmentTree(vector<T>(n, e())) {}
SegmentTree(const vector<T>& v) : n(v.size()) { // vector
v는 0-based로 넘겨주지만 이후 update, query 등은 1-based로
하게 됨에 주의
    sz = pow2GE(n);
    tree = vector<T>(sz << 1, e());
    for (int i = 0; i < n; i++) tree[sz + i] = v[i];
    for (int i = sz - 1; i >= 1; i--) tree[i] = op(tree[i << 1],
tree[i << 1 | 1]);
}

void updateChange(int i, T val) { // 1-based
    assert(1 <= i && i <= n);
    tree[--i |= sz] = val;
    while (i >= 1) tree[i] = op(tree[i << 1], tree[i << 1 |
1]);
}

void updateAdd(int i, T val) { // 1-based
    assert(1 <= i && i <= n);
    --i |= sz;
    tree[i] = op(tree[i], val);
    while (i >= 1) tree[i] = op(tree[i << 1], tree[i << 1 |
1]);
}

T get(int i) const { // 1-based
    assert(1 <= i && i <= n);
    return tree[--i | sz];
}

T query(int l, int r) const { // 1-based // [l:r]
    assert(1 <= l && l <= r && r <= n);
    T resL = e(), resR = e();
    for (--l |= sz, --r |= sz; l <= r; l >>= 1, r >>= 1) {
        if (l & 1) resL = op(resL, tree[l++]);
        if (~r & 1) resR = op(tree[r--], resR);
    }
    return op(resL, resR);
}

int findRightMost(int l, const function<bool(T)> &f) const
{ // f([l:r]) = true인 최대의 r을 찾음, 1과 r 둘 다
1-based // 만족하는 r이 없다면 l-1리턴
    assert(1 <= l && l <= n);
    int node = (l - 1) | sz;
    T acc = e();

    node >>= __builtin_ctz(node); // node[1, r]에서 l은
그대로 두고 r만 최대한 오른쪽으로 이동
    while (f(op(acc, tree[node]))) {

```

```

        acc = op(acc, tree[node++]); // f(node[1, r])
        만족하므로 node+1[r+1, ...]로 이동
        if (!(node & (node - 1))) return n; // node가
        2^k꼴 -> 이전의 node-1=2^k-1[l,r=n]꼴
        node >>= __builtin_ctz(node);
    }

    while (node < sz) {
        node <<= 1; // node[1, r] -> node*2[1, m]
        if (f(op(acc, tree[node]))) acc = op(acc,
tree[node++]); // [m + 1, r]
    }
    return node ^ sz; // f(node)=false인 상황 -> node-1
    리턴해야 해야 되서 (node - 1 - sz) -> 1-based로 (node
- sz)
}

int findKthSmallest(int l, T k) const { // query([l, r])
>= k인 최소의 r 찾음, 1과 r 둘 다 1-based // 즉, 1에서부터
오른쪽으로 k번째 지점 찾는거
    assert(k >= 0);
    return query(l, n) < k ? -1 : findRightMost(l, [&](T
sum) { return sum < k; }) + 1;
}

int findLeftMost(int r, const function<bool(T)> &f) const
{ // f([1:r]) = true인 최소의 l을 찾음, 1과 r 둘 다
1-based // 만족하는 1이 없다면 r+1리턴
    assert(1 <= r && r <= n);
    int node = (r - 1) | sz;
    T acc = e();

    node = max(1, node >> __builtin_ctz(~node)); //
node[1, r]에서 r은 그대로 두고 1만 최대한 왼쪽으로
이동, 최대로 이동한 게 루트여야 되므로 0이 되면 1로
바꿔줌
    while (f(op(tree[node], acc))) {
        acc = op(tree[node--], acc); // f(node[1, r])
        만족하므로 node-1[..., 1-1]로 이동
        if (!(node + 1) & node) return n; // node가
        2^k-1꼴 -> 이전의 node+1=2^k[l=1,r]꼴이었던 거
        node = max(1, node >> __builtin_ctz(~node));
    }

    while (node < sz) {
        node = node << 1 | 1; // node[1, r] ->
        node*2+1[m+1,r]
        if (f(op(tree[node], acc))) acc = op(tree[node--],
acc); // [1, m]
    }
    return node + 2 - sz; // f(node)=false인 상황 ->
    node+1 리턴해야 해야 되서 (node + 1 - sz) -> 1-based로
    (node + 2 - sz)
}

int findKthLargest(int r, T k) const { // query([l, r])
>=
k인 최대의 l 찾음, 1과 r 둘 다 1-based // 즉, r에서부터
왼쪽으로 k번째 지점 찾는거
    assert(k >= 0);
    return query(1, r) < k ? -1 : findLeftMost(r, [&](T
sum) { return sum < k; }) - 1;
}

/*
findLeftMost(l, f)은 만족하는 r 없는 경우 l-1 리턴
findRightMost(r, f)은 만족하는 l 없는 경우 r+1 리턴
findKthSmallest, findKthLargest은 만족하는 값이 없는 경우 -1
리턴
*/

// ex) 합 세그
int op(const int &a, const int &b) { return a + b; }
int e() { return 0; }
SegmentTree<int, sum, zero> seg(v);
*/
2 Graph
2.1 Bellman-Ford
auto bellmanFord = [&](int s) { // O(VE)
    vector<dist_t> dist(n + 1, INF);
    dist[s] = 0;
    for (int _ = n - 1; _--;) {

```

```

    for (auto [u, v, w] : edges) if (dist[u] != INF)
        dist[v] = min(dist[v], dist[u] + w);
    }
    bool hasNegCycle = false;
    for (auto [u, v, w] : edges) hasNegCycle |= (dist[u] != INF && dist[v] > dist[u] + w);
    return {hasNegCycle, dist};
};

2.2 SPFA
auto spfa = [&](int s) { // O(VE) // average O(V+E)
    vector<dist_t> dist(n + 1, INF);
    dist[s] = 0;
    queue<int> q;
    q.push(s);
    vector<int> cnt(n + 1);
    ++cnt[s];
    vector<bool> inq(n + 1);
    while (!q.empty()) {
        auto cur = q.front();
        q.pop();
        inq[cur] = false;
        for (auto [next, cost] : adj[cur]) if (dist[next] > dist[cur] + cost) {
            dist[next] = dist[cur] + cost;
            if (inq[next]) continue;
            if (++cnt[next] >= n) return {true, dist}; // hasNegCycle
            q.push(next);
            inq[next] = true;
        }
    }
    return {false, dist}; // {hasNegCycle, dist}
};

2.3 SCC
pair<int, vector<int> > getSCC(int n, const vector<vector<int> > &adj) { // adj[1-based] // O(V+E)
    vector<int> dfsn(n + 1, sccn(n + 1, -1));
    vector<int> s(n);
    int top = 0, dfsi = 0, scci = 0;
    function<int(int)> dfs = [&](int cur) {
        int low = dfsn[cur] = ++dfsi;
        s[top++] = cur;
        for (auto next : adj[cur]) if (!~sccn[next]) low = min(low, dfsn[next] ? dfsn[next] : dfs(next));
        if (low == dfsn[cur]) {
            do { sccn[s[--top]] = scci; } while (s[top] != cur);
            ++scci;
        }
        return low;
    };
    for (int i = 1; i <= n; i++) if (!dfsn[i]) dfs(i);
    for (int i = 1; i <= n; i++) sccn[i] = scci - 1 - sccn[i];
    return {scci, sccn}; // 0-based // scci = SCC 개수
};

// graphToDAG(n, adj, sccn)
//     vector<vector<int> > sccs(scc);
//     for (int i = 1; i <= n; i++)
sccs[sccn[i]].push_back(i);
//     vector<int> visited(sccts.size(), -1);
//     vector<vector<int> > dag(sccts.size());
//     for (auto &scct : sccts) for (auto u : scct) {
//         for (auto v : adj[u]) if (sccn[v] != sccn[u] && visited[sccn[v]] != sccn[u]) {
//             visited[sccn[v]] = sccn[u];
//             dag[sccn[u]].push_back(sccn[v]);
//         }
//     }

2.4 Bimatch
// addEdge(l, r) : adj[l].push_back(r); // 0<=l<n1, 0<=r<n2
tuple<int, vector<int>, vector<int> > bimatch(int n1, int n2,
const vector<vector<int> > &adj) { // O(VE) // 0-based
    vector<int> matchL(n1, -1), matchR(n2, -1), checkedR(n2, 0);
    auto dfs = [&](auto &&dfs, int l, int trueValue) -> bool {
        for (auto r : adj[l]) if (checkedR[r] != trueValue) {
            checkedR[r] = trueValue;
            if (!~matchR[r] || dfs(dfs, matchR[r], trueValue))
            {
                matchL[l] = r;
                matchR[r] = l;
            }
        }
    }
    return true;
}
    }
    return false;
};

int res = 0;
for (int i = 0; i < adj.size(); i++) res += dfs(dfs, i, i + 1);
return {res, matchL, matchR};
};

2.5 Hopcroft Karp
tuple<int, vector<int>, vector<int> > bimatch(int n1, int n2,
const vector<vector<int> > &adj) { // O(E sqrt(V)) // 0-based
    vector<int> matchL(n1, -1), matchR(n2, -1), lvl(n1),
ptr(n1);
    auto bfs = [&]() -> bool {
        queue<int> q;
        for (int l = 0; l < n1; l++) {
            if (!~matchL[l]) {
                lvl[l] = 0;
                q.push(l);
            }
            else lvl[l] = -1;
        }
        bool flag = false;
        while (!q.empty()) {
            int l = q.front();
            q.pop();
            for (auto r : adj[l]) {
                if (!~matchR[r]) flag = true;
                else if (!~lvl[matchR[r]]) {
                    lvl[matchR[r]] = lvl[l] + 1;
                    q.push(matchR[r]);
                }
            }
        }
        return flag;
    };
    auto dfs = [&](auto &&dfs, int l) -> bool {
        for (int &i = ptr[l]; i < adj[l].size(); i++) {
            int r = adj[l][i];
            if (!~matchR[r] || lvl[matchR[r]] == lvl[l] + 1 && dfs(dfs, matchR[r])) {
                matchL[l] = r;
                matchR[r] = l;
                return true;
            }
        }
        return false;
    };
    int res = 0;
    while (bfs()) {
        fill(ptr.begin(), ptr.end(), 0);
        for (int l = 0; l < n1; l++) res += (~matchL[l] && dfs(dfs, l));
    }
    return {res, matchL, matchR};
};

2.6 MCMF
template <typename C, typename F>
struct Graph {

```

```

    return true;
}
}
return false;
};

int res = 0;
for (int i = 0; i < adj.size(); i++) res += dfs(dfs, i, i + 1);
return {res, matchL, matchR};
};

pair<vector<int>, vector<int> > getMVC(int n1, int n2, const vector<vector<int> > &adj, const vector<int> &matchL, const vector<int> &matchR) { // O(V+E) // 0-based
    vector<bool> visitedL(n1), visitedR(n2);
    auto dfs = [&](auto &&dfs, int l) -> void {
        visitedL[l] = true;
        for (auto r : adj[l]) if (~matchR[r] && !visitedR[r] && !visitedL[matchR[r]]) {
            visitedR[r] = true;
            dfs(dfs, matchR[r]);
        }
    };
    for (int l = 0; l < n1; l++) if (~matchL[l]) dfs(dfs, l);
    vector<int> mvcL, mvcR;
    for (int l = 0; l < n1; l++) if (!visitedL[l]) mvcL.push_back(l);
    for (int r = 0; r < n2; r++) if (visitedR[r]) mvcR.push_back(r);
    return {mvcL, mvcR};
};

2.6 MCMF
template <typename C, typename F>
struct Graph {

```

```

struct Edge {
    int to, rev;
    C cost;
    F c, oc;
};

vector<vector<Edge>> adj;
C DIST_INF = numeric_limits<C>::max();
Graph<int> n : adj(n) {} // 0-based
void addEdge(int a, int b, C cost, F c, F rcap = 0) { // 양방향이면 rcap=c로 호출
    adj[a].push_back({b, adj[b].size(), cost, c, c});
    adj[b].push_back({a, adj[a].size() - 1, -cost, rcap, rcap});
}

pair<C, F> mcmf(int s, int t, F targetFlow =
numeric_limits<F>::max()) { // 0(VEf) // average 0(Ef)
    C minCost = 0;
    F maxFlow = 0;
    vector<Edge *> pedge(adj.size());
    vector<C> dist(adj.size());
    vector<bool> inq(adj.size());
    while (maxFlow < targetFlow) {
        fill(dist.begin(), dist.end(), DIST_INF);
        dist[s] = 0;
        queue<int> q;
        q.push(s);
        inq[s] = true;
        while (!q.empty()) {
            int cur = q.front();
            q.pop();
            inq[cur] = false;
            for (int i = 0; i < adj[cur].size(); i++) {
                auto &e = adj[cur][i];
                if (e.c && dist[e.to] > dist[cur] + e.cost) {
                    dist[e.to] = dist[cur] + e.cost;
                    pedge[e.to] = &e;
                    if (!inq[e.to]) {
                        q.push(e.to);
                        inq[e.to] = true;
                    }
                }
            }
        }
        if (dist[t] == DIST_INF) break;
        F f = targetFlow - maxFlow;
        for (int cur = t; cur != s;) {
            auto &e = *pedge[cur];
            f = min(f, e.c);
            cur = adj[e.to][e.rev].to;
        }
        for (int cur = t; cur != s;) {
            auto &e = *pedge[cur];
            e.c -= f;
            adj[e.to][e.rev].c += f;
            cur = adj[e.to][e.rev].to;
        }
        minCost += f * dist[t];
        maxFlow += f;
    }
    return {minCost, maxFlow};
}

```

2.7 Dinic

```

template <typename F>
struct Dinic {
    struct Edge {
        int to, rev;
        F c, oc;
        F flow() { return max(oc - c, F(0)); }
    };
    vector<int> lvl, ptr, q;
    vector<vector<Edge>> adj;
    Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {} // 0-based
    void addEdge(int a, int b, F c, F rcap = 0) { // 양방향이면 rcap=c로 호출
        if (a == b) return; // self-loop는 최대유량에 영향 X
        adj[a].push_back({b, adj[b].size(), c, c});
        adj[b].push_back({a, adj[a].size() - 1, rcap, rcap});
    }
    F dfs(int cur, int t, F mn) {
        if (cur == t || !mn) return mn;

```

```

        for (int &i = ptr[cur]; i < adj[cur].size(); i++) {
            auto &e = adj[cur][i];
            if (lvl[e.to] != lvl[cur] + 1) continue;
            if (F f = dfs(e.to, t, min(mn, e.c))) {
                e.c -= f, adj[e.to][e.rev].c += f;
                return f;
            }
        }
        return 0;
    }
    template <bool scaling=false> F maxFlow(int s, int t) { // O(V^2 E) // max(cap)이 큰 경우 scaling=true가 빠름
        F res = 0; q[0] = s;
        for (int i = scaling ? 0 : 30; i < 31; i++) do {
            lvl = ptr = vector<int>(q.size());
            int qs = 0, qe = lvl[s] = 1;
            while (qs < qe && !lvl[t]) {
                int cur = q[qs++];
                for (Edge &e : adj[cur]) if (!lvl[e.to] && e.c > (30 - i)) {
                    q[qe++] = e.to, lvl[e.to] = lvl[cur] + 1;
                }
            }
            while (F f = dfs(s, t, numeric_limits<F>::max()))
                res += f;
        } while (lvl[t]);
        return res;
    }
    bool leftOfMinCut(int a) { return lvl[a] != 0; } // min-cut에서 source 집합에 속하는지
};

// vector<pair<int, int>> ref;
// ref.emplace_back(u, graph.adj[u].size()); graph.addEdge(u, v);
// auto [u, idx] = ref[i]; cout << graph.adj[u][idx].flow() << "\n";
// O(min(Ef, V^2 E)), all cap = 1: O(min(V^{2/3}, E^{1/2})E)
// dinic with scaling O(VE log(max_cap))
// 근데 보통 그냥 dinic이 더 빠름, 오히려 max_cap이 클 때
// dinic with scaling이 좋음
// 무한간선 많은 경우 proxysource -> source로 K 용량의 간선을
// 두고 계산하면 애초에 최대유량이 K만큼으로 제한되므로
// 오버플로우 방지 가능
// 정점분할 시 양방향이라면 addEdge(out(a), in(b), c),
// addEdge(out(b), int(a), c)
// 정점분할 시 maxFlow(out(s), in(t))

2.8 Cycle Counting
void getC3(const vector<vector<int>> &adj) { // sum_E{min(deg(u), deg(v))} = 0(m sqrt(m))
    vector<vector<int>> dag(adj.size());
    for (int u = 0; u < adj.size(); u++) {
        for (auto v : adj[u]) {
            if (adj[u].size() < adj[v].size() || (adj[u].size() == adj[v].size() && u < v))
                dag[u].push_back(v);
        }
    }
    // dag에서 차수는 전부 0(sqrt(m)) 이하
    vector<int> mark(adj.size());
    for (int i = 0; i < adj.size(); i++) {
        for (auto j : adj[i]) mark[j] = i;
        for (auto j : dag[i]) {
            for (auto k : dag[j]) {
                if (mark[k] == i) cout << i << " " << j << " "
                << k << "\n";
            }
        }
    }
}
int getC4(const vector<vector<int>> &adj, int MOD) {
    auto cmp = [&](int u, int v) { // u < v
        return adj[u].size() < adj[v].size() || (adj[u].size() == adj[v].size() && u < v);
    };
    vector<vector<int>> dag(adj.size());
    for (int u = 0; u < adj.size(); u++) {
        for (auto v : adj[u]) if (cmp(u, v))
            dag[u].push_back(v);
    }
    int res = 0;
    vector<int> cnt(adj.size());
    for (int i = 0; i < adj.size(); i++) {

```

```

    for (auto j : adj[i]) {
        for (auto k : dag[j]) if (cmp(i, k)) {
            res += cnt[k];
            if (res >= MOD) res -= MOD;
            ++cnt[k];
        }
    }
    for (auto j : adj[i]) {
        for (auto k : dag[j]) if (cmp(i, k)) cnt[k] = 0;
    }
}
return res;
}

// C3에선 rank(i)<rank(j)<rank(k)이므로 dag->dag순 탐색
// C4에선 rank(j)<rank(j')<rank(k) && rank(i)<rank(k) 일뿐 i랑
j간의 대소관계는 모두 고려해야하므로 adj->dag순 탐색

```

2.9 Euler Circuit

```

// 양방향 그래프
struct Edge { int to, rev, cnt; };
pair<bool, vector<int> > getEulerCircuit(vector<vector<Edge> > adj,
int start) { // adj비워져도 괜찮으면 참조 사용 // O(V+E)
    int E = 0;
    for (auto &r : adj) for (auto edge : r) E += edge.cnt;
    E >>= 1;
    vector<int> res;
    auto dfs = [&](auto &&dfs, int cur) -> void {
        while (!adj[cur].empty()) {
            auto &edge = adj[cur].back();
            if (edge.cnt == 0) adj[cur].pop_back();
            else {
                --edge.cnt;
                --adj[edge.to][edge.rev].cnt;
                dfs(dfs, edge.to);
            }
        }
        res.push_back(cur);
    };
    dfs(dfs, start);
    bool possible = (res.size() == E + 1 && res[0] ==
    res.back());
    return {possible, res};
}

```

// 단방향 그래프

```

pair<bool, vector<int> > getEulerCircuit(vector<vector<pair<int,
int> > > adj, int start) { // adj비워져도 괜찮으면 참조 사용
// O(V+E)
    int E = 0;
    for (auto &r : adj) for (auto [next, cnt] : r) E += cnt;
    vector<int> res;
    auto dfs = [&](auto &&dfs, int cur) -> void {
        while (!adj[cur].empty()) {
            auto &[next, cnt] = adj[cur].back();
            int tmp = next; // dangling 방지
            if (--cnt == 0) adj[cur].pop_back();
            dfs(dfs, tmp);
        }
        res.push_back(cur);
    };
    dfs(dfs, start);
    reverse(res.begin(), res.end());
    bool possible = (res.size() == E + 1 && res[0] ==
    res.back());
    return {possible, res};
}

// 오일러 회로는 모든 간선을 지나는 게 목표지, 모든 정점을
지나는 게 목표가 아님
// 따라서 간선들끼리만 연결되어있으면 가능하고, start는 간선과
연결되어 있는 정점 아무거나 가능

// 오일러 경로는 허수 차수 정점이 2개 or 0개여야 가능.
// 만약 2개라면 해당정점들을 cnt=1인 간선으로 잇고 회로를 찾은
뒤 해당간선만 제거하면 됨

// 경로 구하고 싶으면 indg < outdeg인 지점을 start로 호출하면
됨, 이 경우 res[0] != res.back()이 됨

```

2.10 PEO

```

pair<bool, vector<int> > getPEO(int n, const
vector<vector<int>>& adj) { // O(V+E) // 1-based
    vector<int> label(n + 1), order;

```

```

    vector<bool> visited(n + 1, false);
    vector<vector<int> > buckets(n + 1);
    for (int i = 1; i <= n; i++) buckets[0].push_back(i);
    int mx = 0;
    while (order.size() < n) { // mcs
        while (buckets[mx].empty()) --mx;
        int cur = buckets[mx].back();
        buckets[mx].pop_back();
        if (visited[cur]) continue;
        visited[cur] = true;
        order.push_back(cur);
        for (auto next : adj[cur]) if (!visited[next])
            buckets[++label[next]].push_back(next);
        mx = max(mx, label[next]);
    }
}
reverse(order.begin(), order.end());
// check peo
vector<int> pos(n + 1), parent(n + 1, -1);
for (int i = 0; i < n; i++) pos[order[i]] = i;
vector<vector<int> > chd(n + 1);
for (int u = 1; u <= n; u++) {
    int mn = n + 1, w = -1;
    for (auto v : adj[u]) if (pos[v] > pos[u] && mn >
    pos[v]) mn = pos[v], w = v;
    if (<w) {
        parent[u] = w;
        chd[w].push_back(u);
    }
}
vector<int> tag(n + 1);
for (auto u : order) {
    tag[u] = u;
    for (auto v : adj[u]) tag[v] = u;
    for (auto v : chd[u]) {
        for (auto w : adj[v]) {
            if (pos[w] > pos[v] && w != u) {
                if (tag[w] != u) return {false, order};
            }
        }
    }
}
return {true, order};
} // {isChordal, peo} // chordal: 길이 4이상의 모든 simple
cycle이 chord를 포함

```

2.11 theory

트리의 중심

모든 트리는 1개 or 2개의 중심을 가지며 2개라면 그 두 정점은 반드시 인접함
트리의 지름은 트리의 중심을 지나며 모든 지름은 반드시 트리의 중심을 공유함

트리의 지름이 짹수라면: 중심은 1개, 지름의 중간지점이 중심
홀수라면: 중심은 2개, 지름의 가운데 간선으로 연결된 두 정점이 중심
즉, 중심이 1개라면 모든 지름은 그 중심을 지나고 2개라면 모든 지름은 그 두
중심을 잇는 간선을 지님
트리의 모든 지름이 공유하는 정점들의 집합은 항상 하나의 연결된 경로를
이룸

평면그래프

연결된 평면 그래프에서 $v - e + f = 2$ (v : 정점, e : 간선, f : 면)
 $v \geq 3$ 인 중복 간선없는 단순 평면 그래프는 $e \leq 3v - 6$ 을 만족
 $2e =$ 각 면에서 변의 개수의 합 $\geq 3f$ 이므로 $v - e + f = 2$ 와 연립하면 $e \leq 3v - 6$
이분그래프라면 사이클의 최소 길이가 4이므로 $2e \geq 4f$, $e \leq 2v - 4$

매칭

König's theorem: 이분그래프에서 최대매칭 = $|mvc|$
Maximum Independent Set = $V - mvc$: 서로 간선으로 연결되지 않은 정
점들의 모임 중 가장 정점수가 많은 것. 일반그래프에선 NP-hard임
 $mvcL = L - mvcR$, $mvcR = R - mvcL$

Dilworth's theorem: 부분 순서 집합에서 최대 반사슬의 크기는 사슬 분할의
최소 개수와 같음

antichain은 $v_in[i]$ $mvcL[i]$ 이고 $&& v_out[i]$ $mvcR$ 인 v 들을 모으면 됨
clique는 complement graph에서의 Independent set과 같음
hall's marriage theorem: 이분그래프 $G = (L + R, E)$ 에서 L 의 부분집합 S
에 대해 이와 연결된 R 의 부분집합을 $N(S)$ 라 할 때, 이 이분그래프에 perfect
matching이 존재할 필요충분조건은 모든 S 에 대하여 $|S| \leq |N(S)|$

3 Math

3.1 XOR Basis

```

template <typename T>
vector<T> getXorBasis(const vector<T> &v) {
    vector<T> basis;
    for (auto e : v) {

```

```

        for (auto b : basis) e = min(e, e ^ b);
        if (e) basis.push_back(e);
    }
    sort(basis.rbegin(), basis.rend());
    return basis;
}

// T mxSum = 0;
// for (auto b : getXorBasis(v)) mxSum = max(mxSum, mxSum ^
// b);

3.2 exgcd
tuple<ll, ll, ll> extendedGCD(ll a, ll b) { // ax + by =
gcd(a, b)
    if (b == 0) return {1, 0, a};
    auto [x, y, g] = extendedGCD(b, a % b);
    return {y, x - (a / b) * y, g};
}
ll modInverse(ll a, ll b) {
    auto [x, y, g] = extendedGCD(a, b); //ax + by = g
    if (g == 1) return (x + b) % b;
    return -1;
} // modInverse(n, MOD)

3.3 CRT
pll merge(const pll &a, const pll &b) {
    auto [p1, m1] = a;
    auto [p2, m2] = b;
    ll g = __gcd(p1, p2);
    if ((m2 - m1) % g) return {-1, -1};
    ll x = (m2 - m1) / g * modInverse(p1 / g, p2 / g) % (p2 /
g);
    return {p1 / g * p2, p1 * x + m1};
}
pll crt(const vector<pll> &rems) { // rems 원소는 {p, n % p}꼴
// O(N logP) (P:= p1*p2*...*pn)
    pll res(1, 0);
    for (auto &p : rems) {
        res = merge(res, p);
        if (res.first == -1) return {-1, -1};
    }
    if (res.second < 0) res.second += res.first;
    return res;
} // crt().first = -10|면 연립합동식에 해 존재

```

3.4 Miller-Rabin

```

inline ll multiply(ll a, ll b, ll mod) { return __int128(a) *
b % mod; }
ll power(ll a, ll n, ll mod) { //a ^ n % mod
    ll res = 1;
    while (n) {
        if (n & 1) res = multiply(res, a, mod);
        a = multiply(a, a, mod);
        n >>= 1;
    }
    return res;
}
bool isPrime(ll n) { // 밀러-라빈, O(7log^3N)
    if (n <= 1) return false;
    ll d = n - 1, r = 0;
    while (~d & 1) d >>= 1, ++r;
    auto check = [&](ll a) {
        ll remain = power(a, d, n);
        if (remain == 1 || remain == n - 1) return true;
        for (int i = 1; i < r; i++) {
            remain = multiply(remain, remain, n);
            if (remain == n - 1) return true;
        }
        return false;
    };
    vector<ll> nums = {2, 325, 9375, 28178, 450775, 9780504,
1795265022}; // ull
// int 범위는 {2, 7, 61}
    for (ll a : nums) if (a % n && !check(a)) return false;
    return true;
}

```

3.5 Pollard's rho

```

namespace PollardRho {
    ll getPrime(ll n) {
        if (~n & 1) return 2;
        if (isPrime(n)) return n;
        ll a, b, c, g;
        auto f = [&c, &n](ll x) { return (multiply(x, x, n) + c) %
n; };
    }
}
```

```

do {
    a = b = rand() % (n - 2) + 2;
    c = rand() % 10 + 1;
    do {
        a = f(a);
        b = f(f(b));
        g = __gcd(abs(a - b), n);
    } while (g == 1);
} while (g == n);
return getPrime(g);
}

vector<pair<ll, ll> > factorize(ll n) { // O(\sqrt{4}N)
    assert(n > 1);
    vector<pair<ll, ll> > primes;
    while (n > 1) {
        ll prime = getPrime(n), cnt = 0;
        while (n % prime == 0) n /= prime, ++cnt;
        primes.push_back({prime, cnt});
    }
    sort(primes.begin(), primes.end());
    return primes;
}

vector<ll> getAllDivisors(ll n) {
    auto factors = factorize(n);
    int cnt = 1;
    for (auto [p, e] : factors) cnt *= e + 1;
    vector<ll> res = {1};
    res.reserve(cnt);
    for (auto [p, e] : factorize(n)) {
        int sz = res.size();
        ll curP = p;
        for (int _ = e; _--;) {
            for (int i = 0; i < sz; i++) res.push_back(res[i] *
curP);
            curP *= p;
        }
    }
    assert(false, "정렬 필요한지 확인");
    // sort(res.begin(), res.end());
    return res;
}

4 String
4.1 Z
template <typename Container> // Container = string or
vector<>
vector<int> getZ(const Container &st) { // O(N)
    vector<int> z(st.size());
    for (int i = 1, p = 0; i < st.size(); ++i) { // p는
0=<j<i인 j들 중 j + z[j] - 1가 가장 큰 j
        if (i <= p + z[p] - 1) z[i] = min(p + z[p] - 1 - i +
1, z[i - p]);
        while (i + z[i] < st.size() && st[z[i]] == st[i +
z[i]]) ++z[i];
        if (p + z[p] - 1 < i + z[i] - 1) p = i;
    }
    z[0] = st.size();
    return z;
} // z[i] = lcp(st, st[i:])
5 Misc
5.1 RMQ
template <typename T, const T& (*op)(const T&, const T&)>
struct RMQ {
    vector<vector<T> > ac; // ac[i][j] = op(v[j:j+2^i])
    RMQ(const vector<T> &v) : ac(1, v) { // O(N logN)
        for (int i = 1, len = 1; len * 2 <= v.size(); ++i, len *=
2) {
            ac.emplace_back(v.size() - len * 2 + 1);
            for (int j = 0; j < ac[i].size(); j++) ac[i][j] =
op(ac[i - 1][j], ac[i - 1][j + len]);
        }
    }
    T query(int a, int b) const { // 0-based // op[a:b]의 op()
        assert(0 <= a && a <= b && b < ac[0].size());
        int i = __lg(b - a + 1);
        return op(ac[i][a], ac[i][b - (1 << i) + 1]);
    }
}; // const RMQ<int, min> rmq(v);

```

5.2 Random

```
mt19937 rng;
shuffle(v.begin(), v.end(), rng); // O(N)
uniform_int_distribution<int> dist(a, b); // a<=x<=b
uniform_real_distribution<double> dist(a, b); // a<=x<b
x = dist(rng);
```

5.3 Time

```
clock_t sttime = clock();
while(double(clock() - sttime) / CLOCKS_PER_SEC < 0.95) {}
```

5.4 Debug

```
g++ -std=c++17 -O0 -g -fsanitize=undefined
-fno-omit-frame-pointer main.cpp -o main
# -fsanitize=address
```