EK

```cpp
// O(n * m ^ 2)

struct Edge {
    int from, to, cap, flow;
    Edge(int from, int to, int cap, int flow) : from(from), to(to), cap(cap), flow(flow) {}
};

struct EK {
    int n, m;
    vector<Edge> edges;
    vector<int> G[N];
    int a[N], p[N];

    void init(int n) {
        for (int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }

    void AddEdge(int from, int to, int cap) {
        edges.push_back({from, to, cap, 0});
        edges.push_back({to, from, 0, 0});
        m = edges.size();
        G[from].push_back(m - 2);
        G[to].push_back(m - 1);
    }

    bool bfs(int s, int t) {
        memset(a, 0, sizeof(a));
        queue<int> Q;
        Q.push(s);
        a[s] = INF;
        while (!Q.empty()) {
            int u = Q.front();
            Q.pop();
            for (auto v : G[u]) {
                Edge& e = edges[v];
                if (!a[e.to] && e.cap > e.flow) {
                    p[e.to] = v;
                    a[e.to] = min(a[u], e.cap - e.flow);
                    Q.push(e.to);
                }
            }
            if (a[t]) break;
        }
```

```
        return a[t];
    }

    int Maxflow(int s, int t) {
        int maxflow = 0;
        while (bfs(s, t)) {
            maxflow += a[t];
            for (int i = t; i != s; i = edges[p[i]].from) {
                edges[p[i]].flow += a[t];
                edges[p[i] ^ 1].flow -= a[t];
            }
        }
        return maxflow;
    }
};
```

BSGS

```
int qpow(ll x, ll y, ll p = mod) {
    x %= p;
    ll res = 1;
    while (y > 0) {
        if (y & 1) res = res * x % p;
        x = x * x % p;
        y >>= 1;
    }
    return res;
}

int bsgs(int a, int b, int p = mod) {
    if (b == 1) return 0;
    unordered_map<int, int> H;
    H.reserve(sqrt(p));
    b %= p;
    int block = sqrt(p) + 1;
    for (int i = 0, x = b; i < block; i++) {
        H[x % p] = i;
        x = 1ll * x * a % p;
    }
    int A = qpow(a, block, p);
    if (!A) return b == 0 ? 1 : -1;
    for (int i = 1, x = A; i <= block; i++) {
        if (H.count(x)) return 1ll * i * block - H[x];
        x = 1ll * x * A % p;
    }
    return -1;
```

```
}
```

Miller rabin

```
ll qpow(ll a, ll n, ll p) {
    ll ans = 1;
    while (n) {
        if (n & 1)
            ans = (__int128)ans * a % p;
        a = (__int128)a * a % p;
        n >>= 1;
    }
    return ans;
}

bool is_prime(ll x) {
    if (x < 3) return x == 2;
    if (x % 2 == 0) return false;
    ll A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022}, d = x - 1, r = 0;
    while (d % 2 == 0) d /= 2, ++r;
    for (int k = 0; k < 7; k++) {
        ll a = A[k];
        ll v = qpow(a, d, x);
        if (v <= 1 || v == x - 1) continue;
        for (int i = 0; i < r; ++i) {
            v = (__int128)v * v % x;
            if (v == x - 1 && i != r - 1) {
                v = 1;
                break;
            }
            if (v == 1) return false;
        }
        if (v != 1) return false;
    }
    return true;
}
```

Mincostflow

```
namespace atcoder {

template <class Cap, class Cost> struct mcf_graph {
  public:
```

```
    mcf_graph() {}
    mcf_graph(int n) : _n(n), g(n) {}

    int add_edge(int from, int to, Cap cap, Cost cost) {
        assert(0 <= from && from < _n);
        assert(0 <= to && to < _n);
        int m = int(pos.size());
        pos.push_back({from, int(g[from].size())});
        int from_id = int(g[from].size());
        int to_id = int(g[to].size());
        if (from == to) to_id++;
        g[from].push_back(_edge{to, to_id, cap, cost});
        g[to].push_back(_edge{from, from_id, 0, -cost});
        return m;
    }

    struct edge {
        int from, to;
        Cap cap, flow;
        Cost cost;
    };

    edge get_edge(int i) {
        int m = int(pos.size());
        assert(0 <= i && i < m);
        auto _e = g[pos[i].first][pos[i].second];
        auto _re = g[_e.to][_e.rev];
        return edge{
            pos[i].first, _e.to, _e.cap + _re.cap, _re.cap, _e.cost,
        };
    }
    std::vector<edge> edges() {
        int m = int(pos.size());
        std::vector<edge> result(m);
        for (int i = 0; i < m; i++) {
            result[i] = get_edge(i);
        }
        return result;
    }

    std::pair<Cap, Cost> flow(int s, int t) {
        return flow(s, t, std::numeric_limits<Cap>::max());
    }
    std::pair<Cap, Cost> flow(int s, int t, Cap flow_limit) {
        return slope(s, t, flow_limit).back();
    }
    std::vector<std::pair<Cap, Cost>> slope(int s, int t) {
        return slope(s, t, std::numeric_limits<Cap>::max());
    }
```

```
    std::vector<std::pair<Cap, Cost>> slope(int s, int t, Cap flow_limit) {
        assert(0 <= s && s < _n);
        assert(0 <= t && t < _n);
        assert(s != t);
        // variants (C = maxcost):
        // -(n-1)C <= dual[s] <= dual[i] <= dual[t] = 0
        // reduced cost (= e.cost + dual[e.from] - dual[e.to]) >= 0 for all edge
        std::vector<Cost> dual(_n, 0), dist(_n);
        std::vector<int> pv(_n), pe(_n);
        std::vector<bool> vis(_n);
        auto dual_ref = [&]() {
            std::fill(dist.begin(), dist.end(),
                      std::numeric_limits<Cost>::max());
            std::fill(pv.begin(), pv.end(), -1);
            std::fill(pe.begin(), pe.end(), -1);
            std::fill(vis.begin(), vis.end(), false);
            struct Q {
                Cost key;
                int to;
                bool operator<(Q r) const { return key > r.key; }
            };
            std::priority_queue<Q> que;
            dist[s] = 0;
            que.push(Q{0, s});
            while (!que.empty()) {
                int v = que.top().to;
                que.pop();
                if (vis[v]) continue;
                vis[v] = true;
                if (v == t) break;
                // dist[v] = shortest(s, v) + dual[s] - dual[v]
                // dist[v] >= 0 (all reduced cost are positive)
                // dist[v] <= (n-1)C
                for (int i = 0; i < int(g[v].size()); i++) {
                    auto e = g[v][i];
                    if (vis[e.to] || !e.cap) continue;
                    // |-dual[e.to] + dual[v]| <= (n-1)C
                    // cost <= C - -(n-1)C + 0 = nC
                    Cost cost = e.cost - dual[e.to] + dual[v];
                    if (dist[e.to] - dist[v] > cost) {
                        dist[e.to] = dist[v] + cost;
                        pv[e.to] = v;
                        pe[e.to] = i;
                        que.push(Q{dist[e.to], e.to});
                    }
                }
            }
            if (!vis[t]) {
                return false;
```

```
            }

            for (int v = 0; v < _n; v++) {
                if (!vis[v]) continue;
                // dual[v] = dual[v] - dist[t] + dist[v]
                //         = dual[v] - (shortest(s, t) + dual[s] - dual[t]) +
(shortest(s, v) + dual[s] - dual[v])
                //         = - shortest(s, t) + dual[t] + shortest(s, v)
                //         = shortest(s, v) - shortest(s, t) >= 0 - (n-1)C
                dual[v] -= dist[t] - dist[v];
            }
            return true;
        };
        Cap flow = 0;
        Cost cost = 0, prev_cost_per_flow = -1;
        std::vector<std::pair<Cap, Cost>> result;
        result.push_back({flow, cost});
        while (flow < flow_limit) {
            if (!dual_ref()) break;
            Cap c = flow_limit - flow;
            for (int v = t; v != s; v = pv[v]) {
                c = std::min(c, g[pv[v]][pe[v]].cap);
            }
            for (int v = t; v != s; v = pv[v]) {
                auto& e = g[pv[v]][pe[v]];
                e.cap -= c;
                g[v][e.rev].cap += c;
            }
            Cost d = -dual[s];
            flow += c;
            cost += c * d;
            if (prev_cost_per_flow == d) {
                result.pop_back();
            }
            result.push_back({flow, cost});
            prev_cost_per_flow = d;
        }
        return result;
    }

  private:
    int _n;

    struct _edge {
        int to, rev;
        Cap cap;
        Cost cost;
    };

```

```
    std::vector<std::pair<int, int>> pos;
    std::vector<std::vector<_edge>> g;
};

}  // namespace atcoder
```