# Team notebook

ACLimitExceeded (Animesh Sinha, Gaurang Tandon, Arpan Dasgupta)

December 25, 2019

## Contents



## 1 DisjointSets

```cpp
#include <template.hpp>

struct DisjointSetTree {
    ll comp_count;
    vector<ll> parent, comp_size;
    set<ll> roots;

    DisjointSetTree(int n) {
        comp_count = n;
        parent.resize(n);
        comp_size.resize(n, 1);
        iota(parent.begin(), parent.end(), 0);
        for (int i = 0; i < n; i++) {
            roots.insert(i);
        }
    }

    int find(int u) {
```

```
        if (parent[u] == u)
            return parent[u];
        return parent[u] = find(parent[u]);
    }

    bool merge(int u, int v) {
        u = find(u), v = find(v);
        if (u == v)
            return false;
        parent[u] = v;
        comp_size[v] += comp_size[u];
        comp_size[u] = 0;
        roots.erase(u);
        comp_count--;
        return true;
    }
};

class DynamicConnectivity {
    void __dfs(int v, int l, int r, vector<long long>& res) {
        long long last_ans = answer;
        int state = save_ptr;
        for (auto query : tree[v])
            merge(query);
        if (l == r - 1)
            res[l] = answer;
        else {
            int m = (l + r) / 2;
            __dfs(v * 2 + 1, l, m, res);
            __dfs(v * 2 + 2, m, r, res);
        }
        while (save_ptr != state)
            rollback();
        answer = last_ans;
    };

public:
    int size_nodes;
    int size_query;

    struct Node {
        long long parent, comp_size = 1;
    };
    long long answer = 0;
    vector<Node> data;
    vector<long long*> saved_object;
    vector<long long> saved_value;
    int save_ptr = 0;

    struct Query {
        int u, v;
        Query(pair<int, int> p = {0, 0}) {
            u = p.first, v = p.second;
        }
    };
    vector<vector<Query>> tree;

    DynamicConnectivity(int n = 600000, int q = 300000) {
        size_nodes = n;
        size_query = q;
        int tree_size = 1;
        while (tree_size < q)
            tree_size <<= 1;
        data = vector<Node>(n);
        tree = vector<vector<Query>>(2 * tree_size);
        saved_object = vector<long long*>(4 * q);
        saved_value = vector<long long>(4 * q);
        for (int i = 0; i < n; i++) {
            data[i].parent = i;
        }
        // Storing the initial answer
        answer = n;
```

```
}

void change(long long& x, long long y) {
    saved_object[save_ptr] = &x;
    saved_value[save_ptr] = x;
    x = y;
    save_ptr++;
}

void rollback() {
    save_ptr--;
    (*saved_object[save_ptr]) = saved_value[save_ptr];
}

int find(int x) {
    if (data[x].parent == x)
        return x;
    return find(data[x].parent);
}

void merge(const Query& q) {
    int x = find(q.u);
    int y = find(q.v);
    if (x == y)
        return;
    if (data[x].comp_size < data[y].comp_size)
        swap(x, y);
    change(data[y].parent, x);
    change(data[x].comp_size, data[x].comp_size +
        data[y].comp_size);
    // Changing the Answer on query
    change(answer, answer - 1);
}

void add(int l, int r, Query edge, int node = 0, int x =
    0, int y = -1) {
        if (y == -1)
            y = size_query;
        if (l >= r)
            return;
        if (l == x && r == y)
            tree[node].push_back(edge);
        else {
            int m = (x + y) / 2;
            add(l, min(r, m), edge, node * 2 + 1, x, m);
            add(max(m, l), r, edge, node * 2 + 2, m, y);
        }
    }

    vector<long long> solve(int v = 0, int l = 0, int r =
        -1) {
        if (r == -1)
            r = size_query;
        vector<long long> vec(size_query);
        if (size_query > 0)
            __dfs(v, l, r, vec);
        return vec;
    }

    DynamicConnectivity(int n, vector<Query> queries)
        : DynamicConnectivity(n, queries.size()) {
        map<pair<int, int>, int> last;
        for (int i = 0; i < size_query; i++) {
            pair<int, int> p(queries[i].u, queries[i].v);
            if (last.count(p)) {
                last[p] = i;
                add(last[p], i, queries[i]);
                last.erase(p);
            } else {
                last[p] = i;
            }
        }
        for (auto x : last)
```

```
        add(x.second, size_query, x.first);
    }
};
```

# 2 DynamicProgramming

```
#include <template.hpp>

class LineContainer {
   private:
    struct Line {
        mutable long long slope, constt, p;
        bool operator<(const Line &o) const {
            return slope < o.slope;
        }
        bool operator<(long long x) const {
            return p < x;
        }
    };
    multiset<Line, less<>> lines;
    // (for doubles, use inf = 1/.0, div(a,b) = a/b)
    bool __is_max_query = false;
    const long long inf = LLONG_MAX;
    long long __div(long long a, long long b) { // floored
        division
        return a / b - ((a ^ b) < 0 && a % b);
    }
    bool __intersect(multiset<Line>::iterator x,
        multiset<Line>::iterator y) {
        if (y == lines.end()) {
            x->p = inf;
            return false;
        }
        if (x->slope == y->slope)
            x->p = x->constt > y->constt ? inf : -inf;
        else
            x->p = __div(y->constt - x->constt, x->slope -
                y->slope);
        return x->p >= y->p;
    }

   public:
    LineContainer(bool is_max = false) {
        this->__is_max_query = is_max;
    }
    void add(long long slope, long long constt) {
        if (!__is_max_query) {
            slope = -slope;
            constt = -constt;
        }
        auto z = lines.insert({slope, constt, 0}), y = z++,
            x = y;
        while (__intersect(y, z))
            z = lines.erase(z);
        if (x != lines.begin() && __intersect(--x, y))
            __intersect(x, y = lines.erase(y));
        while ((y = x) != lines.begin() && (--x)->p >= y->p)
            __intersect(x, lines.erase(y));
    }
    long long query(long long x) {
        assert(!lines.empty());
        auto l = *lines.lower_bound(x);
        return (l.slope * x + l.constt) * (__is_max_query ?
            1 : -1);
    }
};

void dp_sos(vll &arr) {
    const int bitsize = 20;
    for (int i = 0; i < bitsize; ++i)
```

```
        for (int mask = 0; mask < (1 << bitsize); ++mask)
            if (mask & (1 << i))
                arr[mask] += arr[mask ^ (1 << i)];
}
```

# 3 FastFourier

```
#include "template.hpp"

class Polynomial {
#define NTT
    static ll __mod_pow(ll a, ll n) {
        int res = 1;
        for (a %= MOD; n > 0; n >>= 1) {
            if (n & 1)
                res = (res * 1ll * a) % MOD;
            a = (a * 1ll * a) % MOD;
        }
        return res;
    }

  public:
    int order;

    explicit Polynomial(vll coefficients) {
        order = coefficients.size() - 1;
        this->resize(order);
        for (int i = 0; i <= order; i++)
            coeff[i] = coefficients[i];
    }

    void resize(int order) {
        int size;
        for (size = 1; size < order + 1; size *= 2)
            ;
        coeff.resize(size);
    }
#ifdef NTT
    vll coeff;
    void fft(bool invert = false) {
        static const int root = 973800541;
        static const int root_1 = 595374802;
        static const int root_pw = 1 << 20;
        static const ll MOD = 998244353;
        int n = coeff.size();
        for (int i = 1, j = 0; i < n; i++) {
            int bit = n >> 1;
            for (; j & bit; bit >>= 1)
                j ^= bit;
            j ^= bit;
            if (i < j)
                swap(coeff[i], coeff[j]);
        }
        for (int len = 2; len <= n; len <<= 1) {
            int wlen = invert ? root_1 : root;
            for (int i = len; i < root_pw; i <<= 1)
                wlen = (int)(1LL * wlen * wlen % MOD);
            for (int i = 0; i < n; i += len) {
                int w = 1;
                for (int j = 0; j < len / 2; j++) {
                    int u = coeff[i + j],
                        v = (ll)((coeff[i + j + len / 2] * 1ll
                            * w) % MOD);
                    coeff[i + j] = u + v < MOD ? u + v : u + v
                        - MOD;
                    coeff[i + j + len / 2] = u - v >= 0 ? u -
                        v : u - v + MOD;
                    w = (int)((w * 1ll * wlen) % MOD);
                }
            }
```

```
        }
        if (invert) {
            int n_1 = __mod_pow(n, MOD - 2);
            for (ll &x : coeff)
                x = (ll)((x * 1ll * n_1) % MOD);
        }
    }
#endif
#ifdef FFT
    vcd coeff;
    int reverse(int num, int lg_n) {
        int res = 0;
        for (int i = 0; i < lg_n; i++) {
            if (num & (1 << i))
                res |= 1 << (lg_n - 1 - i);
        }
        return res;
    }
    void fft(vector<cd> &a, bool invert) {
        const ld PI = acos(-1);
        int n = a.size();
        int lg_n = 0;
        while ((1 << lg_n) < n)
            lg_n++;
        for (int i = 0; i < n; i++)
            if (i < reverse(i, lg_n))
                swap(a[i], a[reverse(i, lg_n)]);
        for (int len = 2; len <= n; len <<= 1) {
            double ang = 2 * PI / len * (invert ? -1 : 1);
            cd wlen(cos(ang), sin(ang));
            for (int i = 0; i < n; i += len) {
                cd w(1);
                for (int j = 0; j < len / 2; j++) {
                    cd u = a[i + j], v = a[i + j + len / 2] *
                        w;
                    a[i + j] = u + v;
                    a[i + j + len / 2] = u - v;
                    w *= wlen;
                }
            }
        }
        if (invert)
            for (cd &x : a)
                x /= n;
    }
#endif
    friend Polynomial operator*(const Polynomial &a, const
        Polynomial &b) {
        Polynomial x(a), y(b);
        int order = a.order + b.order;
        x.resize(order), y.resize(order);
        x.fft(), y.fft();
        int size = x.coeff.size();
        vll poly(size);
        for (int i = 0; i < size; i++)
            poly[i] = (x.coeff[i] * y.coeff[i]) % MOD;
        Polynomial res(poly);
        res.fft(true), res.order = order;
        return res;
    }

    friend Polynomial operator^(const Polynomial &a, ll pow)
        {
        Polynomial x(a);
        int order = a.order * pow;
        x.resize(order);
        x.fft();
        int size = x.coeff.size();
        vll poly(size);
        for (int i = 0; i < size; i++)
            poly[i] = __mod_pow(x.coeff[i], pow);
        Polynomial res(poly);
```

```
        res.fft(true), res.order = order;
        return res;
    }
};
```

# 4 FlowAlgorithms

```
#include "template.hpp"

class Dinics {
  public:
    typedef int FT;         // can use float/doublestatic
    const FT INF = 1e9;     // maximum capacity
    static const FT EPS = 0; // minimum capacity/flow change
    int nodes, src, dest;
    vector<int> dist, q, work;
    struct Edge {
        int to, rev;
        FT f, cap;
    };
    vector<vector<Edge> > g;
    bool dinic_bfs() {
        fill(dist.begin(), dist.end(), -1);
        dist[src] = 0;
        int qt = 0;
        q[qt++] = src;
        for (int qh = 0; qh < qt; qh++) {
            int u = q[qh];
            for (int j = 0; j < (int)g[u].size(); j++) {
                Edge &e = g[u][j];
                int v = e.to;
                if (dist[v] < 0 && e.f < e.cap)
                    dist[v] = dist[u] + 1;
                q[qt++] = v;
            }
        }
        return dist[dest] >= 0;
    }
    int dinic_dfs(int u, int f) {
        if (u == dest)
            return f;
        for (int &i = work[u]; i < (int)g[u].size(); i++) {
            Edge &e = g[u][i];
            if (e.cap <= e.f)
                continue;
            int v = e.to;
            if (dist[v] == dist[u] + 1) {
                FT df = dinic_dfs(v, min(f, e.cap - e.f));
                if (df > 0) {
                    e.f += df, g[v][e.rev].f -= df;
                    return df;
                }
            }
        }
        return 0;
    }
    Dinics(int n) : dist(n, 0), q(n, 0), work(n, 0), g(n),
        nodes(n) {
    } // *** s->t (cap); t->s (rcap)
    void addEdge(int s, int t, FT cap, FT rcap = 0) {
        g[s].push_back({t, (int)g[t].size(), 0, cap});
        g[t].push_back({s, (int)g[s].size() - 1, 0, rcap});
    } // ***
    FT maxFlow(int _src, int _dest) {
        src = _src, dest = _dest;
        FT result = 0, delta;
        while (dinic_bfs()) {
            fill(work.begin(), work.end(), 0);
            while ((delta = dinic_dfs(src, INF)) > EPS)
                result += delta;
```

```
        }
        return result;
    }
};

class HopcroftKarp {
   public:
    static const int INF = 1e9;
    int U, V, nil;
    vector<int> pairU, pairV, dist;
    vector<vector<int> > adj;
    bool bfs() {
        queue<int> q;
        for (int u = 0; u < U; u++)
            if (pairU[u] == nil)
                dist[u] = 0, q.push(u);
            else
                dist[u] = INF;
        dist[nil] = INF;
        while (not q.empty()) {
            int u = q.front();
            q.pop();
            if (dist[u] >= dist[nil])
                continue;
            for (int v : adj[u])
                if (dist[pairV[v]] == INF)
                    dist[pairV[v]] = dist[u] + 1,
                        q.push(pairV[v]);
        }
        return dist[nil] != INF;
    }
    bool dfs(int u) {
        if (u == nil)
            return true;
        for (int v : adj[u])
            if (dist[pairV[v]] == dist[u] + 1)
                if (dfs(pairV[v])) {
                    pairV[v] = u, pairU[u] = v;
                    return true;
                }
        dist[u] = INF;
        return false;
    }

   public:
    HopcroftKarp(int U_, int V_) {
        nil = U = V = max(U_, V_);
        adj.resize(U + 1);
        dist.resize(U + 1);
        pairU.resize(U + 1);
        pairV.resize(V);
    }
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }
    int maxMatch() {
        fill(pairU.begin(), pairU.end(), nil);
        fill(pairV.begin(), pairV.end(), nil);
        int res = 0;
        while (bfs())
            for (int u = 0; u < U; u++)
                if (pairU[u] == nil && dfs(u))
                    res++;
        return res;
    }
};
```

# 5 Geometry

```
#include "template.hpp"
```

```
class Point {
  public:
   typedef long long coord_t; coord_t x, y;
   Point(coord_t coord_x = 0, coord_t coord_y = 0) {
      this->x = coord_x; this->y = coord_y; }
   Point(pair<coord_t, coord_t> coord) { this->x =
      coord.first; this->y = coord.second; }
   friend bool operator<(const Point &a, const Point &b) {
      return (a.x != b.x) ? a.x < b.x : a.y < b.y; }
   friend bool operator==(const Point &a, const Point &b) {
      return (a.x == b.x) && (a.y == b.y); }
   friend istream &operator>>(istream &in, Point &p) { in
      >> p.x >> p.y; return in; }
   friend ostream &operator<<(ostream &out, Point &p) { out
      << p.x << " " << p.y; return out; }
   static coord_t area(const Point &a, const Point &b,
      const Point &c) { return a.x * (b.y - c.y) + b.x *
      (c.y - a.y) + c.x * (a.y - b.y); }; // Area
      function: area < 0 = clockwise, area > 0
      counterclockwise
   static coord_t sq_dist(const Point &a, const Point &b) {
      return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) *
      (a.y - b.y); }
   static coord_t cross(const Point &O, const Point &A,
      const Point &B) { return (A.x - O.x) * (B.y - O.y) -
      (A.y - O.y) * (B.x - O.x); }
   static coord_t dot(const Point &O, const Point &A, const
      Point &B) { return (A.x - O.x) * (B.x - O.x) + (A.y
      - O.y) * (B.y - O.y); }
   static coord_t area(const vector<Point> &polygon) {
      int n = polygon.size();
      coord_t ans = 0;
      for (int i = 0; i < n; i++)
         ans += polygon[i].x * polygon[(i + 1) % n].y -
            polygon[i].y * polygon[(i + 1) % n].x;
   }
   static vector<Point> convex_hull(vector<Point> &a) {
      if (a.size() <= 3) return a;
      int n = a.size(), k = 0; sort(a.begin(), a.end());
      vector<Point> result(2 * n);
      for (int i = 0; i < n; ++i) {
         while (k >= 2 && cross(result[k - 2], result[k -
            1], a[i]) <= 0) k--;
         result[k++] = a[i];
      }
      for (int i = n - 1, t = k + 1; i > 0; --i) {
         while (k >= t && cross(result[k - 2], result[k -
            1], a[i - 1]) <= 0) k--;
         result[k++] = a[i - 1];
      }
      result.resize(k - 1); return result;
   }
};
```

# 6 GraphAlgorithms

```
#include "template.hpp"

class Graph {
  public:
   enum NodeColor { VISITED, VISITING, UNVISITED };
   struct Node { int index; vpl adjacent; NodeColor color =
      UNVISITED; };
   vector<Node> list; int n;
   Graph(int n) {
      list.resize(n);
      for (int i = 0; i < n; i++) list[i].index = i;
      this->n = n;
   }
```

```
void add_edge(int u, int v, long long w = 1) {
    list[u].adjacent.emplace_back(v, w);
    list[v].adjacent.emplace_back(u, w);
}

pair<vll, vll> dijkstra(vll from) {
    vll dist(n, INT64_MAX), parent(n, INT32_MAX);
    priority_queue<pll, vpl, greater<>> q;
    for (auto index : from) dist[index] = 0,
        q.emplace(0, index);
    while (!q.empty()) {
        pll top = q.top(); q.pop();
        if (top.first > dist[top.second]) continue;
        for (auto edge : list[top.second].adjacent)
            if (top.first + edge.second <
                dist[edge.first])
                dist[edge.first] = top.first + edge.second,
                parent[edge.first] = top.second,
                q.emplace(top.first + edge.second,
                    edge.first);
    } return {dist, parent};
}
vector<int> topological_sort() {
    vector<int> in_degree(list.size(), 0), result;
    result.reserve(list.size());
    for (auto node : list) for (auto route :
        node.adjacent)
            in_degree[route.first - 1]++;
    queue<int> process;
    for (int i = 0; i < list.size(); i++)
        if (in_degree[i] == 0) { process.push(i);
            result.push_back(i); }
    while (!process.empty()) {
        int processing = process.front(); process.pop();
        for (auto route : list[processing].adjacent) {
            in_degree[route.first - 1]--;
            if (in_degree[route.first - 1] == 0) {
                process.push(route.first - 1);
                result.push_back(route.first - 1);
            }
        }
    }
    return result;
}
mll components() {
    vbl visited(n); mll result(0);
    for (int i = 0; i < n; i++) {
        if (visited[i]) continue;
        vll component; stack<ll> process;
        process.push(list[i].index);
            component.push_back(i); visited[i] = true;
        while (!process.empty()) {
            ll processing = process.top(); process.pop();
            for (pll neighbor : list[processing].adjacent)
                if (!visited[neighbor.first])
                    process.push(neighbor.first),
                        component.push_back(neighbor.first),
                        visited[neighbor.first] = true;
        } result.push_back(component);
    } return result;
}
pair<vll, vll> bellman_ford(vll from) {
    vll distances(n, INT64_MAX);
    vll parent(n, INT32_MAX);
    for (ll &i : from)
        distances[i] = 0;
    for (int i = 0; i < n - 1; i++)
        for (int source = 0; source < n - 1; source++) {
            if (distances[source] == INT64_MAX)
                continue;
            for (const auto &edge :
                list[source].adjacent) {
```

```
                ll sink = edge.first;
                if (distances[source] + edge.second <
                    distances[sink])
                    distances[sink] = distances[source] +
                        edge.second, parent[sink] = source;
            }
        }
    for (ll source = 0; source < n - 1; source++) // -ve
        cycle check
        for (const auto &edge : list[source].adjacent) {
            ll sink = edge.first;
            if (distances[source] + edge.second <
                distances[sink]) {
                for (ll i : from) distances[i] = -1;
                return {distances, parent};
            }
        }
    return {distances, parent};
}
mll floyd_warshall() {
    mll distances(n, vll(n, INT64_MAX));
    for (int i = 0; i < n; i++) distances[i][i] = 0;
    for (int i = 0; i < n; i++) for (auto route :
        list[i].adjacent)
        distances[i][route.first] = route.second;
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (distances[i][k] == INT64_MAX ||
                    distances[k][j] == INT64_MAX) continue;
                distances[i][j] = min(distances[i][j],
                    distances[i][k] + distances[k][j]);
            }
        }
    } return distances;
}
```

```
    pair<ll, vll> prims_mst() {
        priority_queue<pll, vpl, greater<>> routes;
        vll costs(n);
        vbl visited(n, false);
        for (int i = 0; i < n; i++) {
            if (!visited[i]) routes.emplace(INT32_MAX, i);
            while (!routes.empty()) {
                pll best = routes.top(); routes.pop();
                if (!visited[best.second]) costs[best.second]
                    = best.first;
                visited[best.second] = false;
                for (const auto &path :
                    list[best.second].adjacent)
                    if (!visited[path.second])
                        routes.push(path);
            }
        }
        return {accumulate(costs.begin(), costs.end(), 0),
            costs};
    }
};
```

# 7 MergeSortTree

```
#include "template.hpp"

template <typename Type>
struct MergeSortTree {
    int size;
    vector<Type> data;
    vector<vector<int>> tree_idx;
    vector<vector<Type>> tree_val;
    long long inversions;
```

```
template <typename DataType>
vector<DataType> merge(const vector<DataType> &arr1,
    const vector<DataType> &arr2) {
    int n = arr1.size(), m = arr2.size();
    vector<DataType> result; result.reserve(n + m);
    for (int x = 0, y = 0; x < n || y < m;) {
        if (x < n && (y >= m || arr1[x] <= arr2[y]))
            result.push_back(arr1[x++]);
        else result.push_back(arr2[y++]), inversions += n
            - x;
    } return move(result);
}
int order_fn(const Type &value, const vector<Type> &arr)
    {
    return lower_bound(arr.begin(), arr.end(), value) -
        arr.begin();
}
explicit MergeSortTree(const vector<Type> &list) {
    for (size = 1; size < list.size(); size *= 2) ;
    // Make a tree based on the values
    tree_val.resize(2 * size); data = vector<Type>(list);
    for (int i = 0; i < list.size(); i++)
        tree_val[i + size].push_back(i);
    for (int i = size - 1; i > 0; --i)
        tree_val[i] = merge<Type>(tree_val[i << 1],
            tree_val[i << 1 | 1]);
    // Make a tree based on the indices
    tree_idx.resize(2 * size);
    vector<pair<Type, int>> convert(list.size());
    for (int i = 0; i < list.size(); i++)
        convert[i].first = list[i], convert[i].second = i;
    sort(convert.begin(), convert.end());
    for (int i = 0; i < list.size(); i++)
        tree_idx[i + size].push_back(convert[i].second);
    for (int i = size - 1; i > 0; --i)
            tree_idx[i] = merge<int>(tree_idx[i << 1],
                tree_idx[i << 1 | 1]);
    }
    int order_of_key(int l, int r, Type value) {
        int result = 0;
        for (l = l + size, r = r + size; l < r; l >>= 1, r
            >>= 1) {
            if (l & 1) result += order_fn(value,
                tree_val[l++]);
            if (r & 1) result += order_fn(value,
                tree_val[--r]);
        } return result;
    }
    int key_of_order(int l, int r, int order, int node = 0,
        int x = 0, int y = -1) {
        if (y == -1) y = size;
        if (x + 1 == y) return tree_idx[node][0];
        int m = (upper_bound(tree_idx[2 * node].begin(),
            tree_idx[2 * node].end(), r - 1) - tree_idx[2 *
            node].begin())
              - (lower_bound(tree_idx[2 * node].begin(),
                tree_idx[2 * node].end(), l) - tree_idx[2 *
                node].begin());
        if (m >= order) return key_of_order(l, r, order,
            node << 1, x, (x + y) / 2);
        else return key_of_order(l, r, order - m, node << 1
            | 1, (x + y) / 2, y);
    }
};
```

# 8 Miscelleneous

```
#include "template.hpp"
```

```
ll binary_search(ll TOP, ll BOT, function<bool(ll)> check) {
    ll result = BOT - 1;
    for (ll top = TOP, bot = BOT, mid = bot + (top - bot) /
        2; bot <= top; mid = bot + (top - bot) / 2) {
        if (check(mid) && !check(mid - 1)) { result = mid;
            break; }
        (check(mid)) ? (top = mid - 1) : (bot = mid + 1);
    }
}

ll gcd(ll a, ll b, ll &x, ll &y) {
    int g = a; x = 1, y = 0;
    if (b != 0) g = gcd(b, a % b, y, x), y -= a / b * x;
    return g;
}
ll mod_inverse(ll a, ll mod) {
    ll x, y; gcd(a, mod, x, y);
    return (x + mod) % mod;
}

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;
```

# 9 MobiusSieve

```
#include "template.hpp"

class Multiplicative {
// This is the definition for PHI
#define fn_prime_values(prime) (prime - 1)
#define fn_non_coprime(num, prime) (fn[num] * prime)
public:
 ll size;
 vector<ll> fn;
 vector<ll> primes;
 vector<ll> lowest_prime_factor;
 Multiplicative(ll size) {
     size = size;
     lowest_prime_factor = vector<ll>(size, 0);
     fn = vector<ll>(size, 0);
     // https://stackoverflow.com/questions/34260399
     // linear sieve
     for (ll i = 2; i < size; i++)
         lowest_prime_factor[i] = i;
     // put any specific initialization code here like
     // multiplicativeFn[1] = 1;
     for (ll i = 2; i < size; i++) {
         if (lowest_prime_factor[i] == i) {
             fn[i] = fn_prime_values(i);
             primes.push_back(i);
         }
         for (auto p : primes) {
             ll ith_multiple = i * p;
             if (ith_multiple >= size)
                 break;
             lowest_prime_factor[ith_multiple] =
                 min(lowest_prime_factor[i], p);

             if (i % p) {
                 fn[ith_multiple] = fn[i] * fn[p];
             } else {
                 fn[ith_multiple] = fn_non_coprime(i, p);
                 break;
             }
         }
     }
 }
```

```
};
```

# 10 PalindromicTree

```
#include "template.hpp"

class PalindromicTree {
    const static long long MAXN = 100000;

   public:
    struct Node {
        int start, end;
        int length;
        int insert_edge[26];
        int suffix_edge;
    };

    Node root1, root2;
    Node tree[MAXN];
    int curr_node, ptr, size;
    string s;

    void insert(int idx) {
        int tmp = curr_node;
        while (true) {
            int curLength = tree[tmp].length;
            if (idx - curLength >= 1 and s[idx] == s[idx -
                curLength - 1])
                break;
            tmp = tree[tmp].suffix_edge;
        }
        if (tree[tmp].insert_edge[s[idx] - 'a'] != 0) {
            curr_node = tree[tmp].insert_edge[s[idx] - 'a'];
            return;
        }
        ptr++;
        tree[tmp].insert_edge[s[idx] - 'a'] = ptr;
        tree[ptr].length = tree[tmp].length + 2;
        tree[ptr].end = idx;
        tree[ptr].start = idx - tree[ptr].length + 1;
        tmp = tree[tmp].suffix_edge;
        curr_node = ptr;
        if (tree[curr_node].length == 1) {
            tree[curr_node].suffix_edge = 2;
            return;
        }
        while (true) {
            int cur_length = tree[tmp].length;
            if (idx - cur_length >= 1 and s[idx] == s[idx -
                cur_length - 1])
                break;
            tmp = tree[tmp].suffix_edge;
        }
        tree[curr_node].suffix_edge =
            tree[tmp].insert_edge[s[idx] - 'a'];
    }

    PalindromicTree(string st) {
        root1.length = -1, root1.suffix_edge = 1,
            root2.length = 0,
        root2.suffix_edge = 1, tree[1] = root1, tree[2] =
            root2, ptr = 2;
        curr_node = 1, s = st, size = st.size();
        for (int i = 0; i < size; i++)
            insert(i);
    }

    vpl get_palindromes() {
        vpl res(ptr - 2);
        for (int i = 3; i <= ptr; i++)
```

```
            res[i - 2] = {tree[i].start, tree[i].end};
        return res;
    }
};
```

## 11 SegmentTree

```
#include <template.hpp>

template <typename Type>
class LazySegtree {
    int size;
    vector<Type> tree, lazy;
    Type _default;
    function<Type(Type, Type)> _operation;
    function<Type(Type, Type)> _setter;

    void split(int node) {
        lazy[2 * node] = _setter(lazy[2 * node], lazy[node]);
        tree[2 * node] = _setter(tree[2 * node], lazy[node]);
        lazy[2 * node + 1] = _setter(lazy[2 * node + 1],
            lazy[node]);
        tree[2 * node + 1] = _setter(tree[2 * node + 1],
            lazy[node]);
        lazy[node] = _default;
    }
    void merge(int node) {
        tree[node] = _operation(tree[2 * node], tree[2 *
            node + 1]);
    }

  public:
    LazySegtree(int n, const function<Type(Type, Type)> &op,
            const function<Type(Type, Type)> &set, const
                Type identity) {
        for (size = 1; size < n; size <<= 1)
            ;
        _setter = set, _operation = op, _default = identity;
        tree.assign(2 * size, _default);
        lazy.assign(2 * size, _default);
    }

    void modify(int l, int r, Type delta, int node = 1, int
        x = 0, int y = -1) {
        if (y == -1)
            y = size;
        if (r <= x || l >= y)
            return;
        if (l <= x && y <= r) {
            lazy[node] = _setter(lazy[node], delta);
            tree[node] = _setter(tree[node], delta);
            return;
        }
        split(node);
        modify(l, r, delta, 2 * node, x, (x + y) / 2);
        modify(l, r, delta, 2 * node + 1, (x + y) / 2, y);
        merge(node);
    }
    Type query(int l, int r, int node = 1, int x = 0, int y
        = -1) {
        if (y == -1)
            y = size;
        if (r <= x || l >= y)
            return _default;
        if (l <= x && y <= r) {
            return tree[node];
        }
        split(node);
        Type lres = query(l, r, 2 * node, x, (x + y) / 2);
```

```
        Type rres = query(l, r, 2 * node + 1, (x + y) / 2,
            y);
        merge(node);
        return _operation(lres, rres);
    }
};

template <typename Type>
class ImplicitSegupdate {
    struct Node {
        Type data = 0;
        Node *l_ptr = nullptr, *r_ptr = nullptr;
        Node *l_child() {
            if (l_ptr == nullptr)
                l_ptr = new Node, r_ptr = new Node;
            return l_ptr;
        }
        Node *r_child() {
            if (r_ptr == nullptr)
                l_ptr = new Node, r_ptr = new Node;
            return r_ptr;
        }
    };
    int size;
    Node *root;
    function<Type(Type, Type)> _setter;

   public:
    ImplicitSegupdate(int n, const function<Type(Type,
        Type)> &set) {
        for (size = 1; size < n; size <<= 1)
            ;
        _setter = set;
        root = new Node;
    }
    void modify(int l, int r, Type delta, Node *node =
        nullptr, int x = 0,
                int y = -1) {
        if (node == nullptr)
            node = root, y = size;
        if (r <= x || l >= y)
            return;
        if (l <= x && y <= r) {
            node->data = _setter(node->data, delta);
            return;
        }
        modify(l, r, delta, node->l_child(), x, (x + y) / 2);
        modify(l, r, delta, node->r_child(), (x + y) / 2, y);
    }
    Type query(int p, Node *node = nullptr, int x = 0, int y
        = -1) {
        if (node == nullptr)
            node = root, y = size;
        if (x == p && y == p + 1) {
            return node->data;
        }
        if (x <= p && p < (x + y) / 2)
            return _setter(node->data,
                           query(p, node->l_child(), x, (x +
                               y) / 2));
        else
            return _setter(node->data,
                           query(p, node->r_child(), (x + y) /
                               2, y));
    }
};

class PersistentSegtree {
    struct Node {
        int l, r, val;
        Node() {
```

```
        l = r = val = 0;
    }
};
int node_size, query_size;
int curr;
vector<int> root;
vector<Node> seg;

PersistentSegtree(int n, int q) {
    node_size = n, query_size = q;
    seg.resize(2 * (n + q * (log2(n) + 1)));
    root = vector<int>(query_size + 10);
    curr = 1, seg[curr].l = seg[curr].r = seg[curr].val
        = 0;
}
int _new_node(int val, int l, int r) {
    seg[curr].val = val, seg[curr].l = l, seg[curr].r =
        r;
    return curr++;
}
int insert(int cur, int idx, int val, int lo, int hi) {
    if (idx < lo || idx > hi)
        return cur;
    else if (lo == hi)
        return _new_node(val, 0, 0);
    int mid = (lo + hi) >> 1;
    int pos = _new_node(-1, insert(seg[cur].l, idx, val,
        lo, mid),
                    insert(seg[cur].r, idx, val, mid +
                        1, hi));
    seg[pos].val = max(seg[seg[pos].l].val,
        seg[seg[pos].r].val);
    return pos;
}
};
```

# 12 StonglyConnected

```
#include "template.hpp"

struct DirectedGraph {
    int size, curr;
    vector<vector<int>> adjacent_f, adjacent_r, comp_nodes;
    vector<int> order, comp;
    vector<bool> visited;

    DirectedGraph(int n) {
        size = n;
        order.resize(size);
        adjacent_f.resize(size);
    }
    void add_edge(int v1, int v2) {
        adjacent_f[v1].push_back(v2);
        adjacent_r[v2].push_back(v1);
    }

    void _scc_dfs1(int u) {
        visited[u] = 1;
        for (auto w : adjacent_f[u])
            if (!visited[w])
                _scc_dfs1(w);
        order.push_back(u);
    }
    void _scc_dfs2(int u) {
        visited[u] = 1;
        comp[u] = curr;
        comp_nodes[curr].push_back(u);
        for (auto w : adjacent_r[u])
            if (!visited[w])
                _scc_dfs2(w);
    }
```

```
    void stongly_connected_components() {
        fill(visited.begin(), visited.end(), false);
        order.clear();
        for (int i = 0; i < size; i++)
            if (!visited[i])
                _scc_dfs1(i);
        fill(visited.begin(), visited.end(), false);
        reverse(order.begin(), order.end());
        curr = 0;
        // components are generated in topological order
            (Kosaraju)
        for (auto u : order)
            if (!visited[u])
                comp_nodes[++curr].clear(), _scc_dfs2(u);
    }
};

struct Satisfiability : DirectedGraph {
    vector<bool> val;
    Satisfiability(int size) : DirectedGraph(2 * size) {
        val = vector<bool>(size, false);
    }

    bool solvable(int vars) {
        stongly_connected_components();
        for (int i = 0; i < vars; i++)
            if (comp[var(i)] == comp[NOT(var(i))])
                return false;
        return true;
    }
    vector<bool> solve() {
        fill(val.begin(), val.end(), 0);
        for (int i = 1; i <= curr; i++)
            for (auto it : comp_nodes[i]) {
                int u = it >> 1;
                if (val[u])
                    continue;
                val[u] = (it & 1 ? +1 : -1);
            }
        return val;
    }
    int var(int x) {
        return x << 1;
    }
    int NOT(int x) {
        return x ^ 1;
    }
    void add_imp(int v1, int v2) {
        add_edge(v1, v2);
        add_edge(1 ^ v2, 1 ^ v1);
    }
    void add_equiv(int v1, int v2) {
        add_imp(v1, v2);
        add_imp(v2, v1);
    }
    void add_or(int v1, int v2) {
        add_edge(1 ^ v1, v2);
        add_edge(1 ^ v2, v1);
    }
    void add_xor(int v1, int v2) {
        add_or(v1, v2);
        add_or(1 ^ v1, 1 ^ v2);
    }
    void add_true(int v1) {
        add_edge(1 ^ v1, v1);
    }
    void add_and(int v1, int v2) {
        add_true(v1);
        add_true(v2);
    }
};
```

# 13 StringAlgorithms

```cpp
#include "template.hpp"

class KMPstring {
    string pattern;
    vll lps;

  public:
    explicit KMPstring(const string &pattern) {
        this->pattern = pattern;
        ll m = pattern.size();
        lps = vll(m + 1, 0);
        ll i = 0, j = -1;
        lps[0] = -1;
        while (i < m) {
            while (j >= 0 && pattern[i] != pattern[j])
                j = lps[j];
            i++, j++;
            lps[i] = j;
        }
    }
    vll match(const string &text) {
        ll n = text.size(), m = pattern.size();
        vll matches, m_length(n);
        ll i = 0, j = 0;
        while (i < n) {
            while (j >= 0 && text[i] != pattern[j])
                j = lps[j];
            i++, j++;
            m_length[i - 1] = j;
            if (j == m) {
                matches.push_back(i - m);
                j = lps[j];
            }
        }
        return move(matches); // or m_length
    }
};

struct SuffixArray {
    string s;
    int n, __log_n;
    vector<int> sa;          // Suffix Array
    vector<vector<int>> ra; // Rank Array
    vector<vector<int>> _lcp; // Longest Common Prefix
    vector<int> __msb, __dollar;

    SuffixArray(string st) {
        n = st.size();
        __log_n = log2(n) + 1;
        ra = vector<vector<int>>(__log_n, vector<int>(n));
        sa = vector<int>(n);

        __msb = vector<int>(n);
        int mx = -1;
        for (int i = 0; i < n; i++) {
            if (i >= (1 << (mx + 1)))
                mx++;
            __msb[i] = mx;
        }
        this->s = st;
        build_SA();
    }

    void __counting_sort(int l, int k) {
        int maxi = max(300, n);
        vector<int> count(maxi, 0), temp_sa(n, 0);
        for (int i = 0; i < n; i++) {
            int idx = (i + k < n ? ra[l][i + k] : 0);
            count[idx]++;
        }
```

```
    for (int i = 0, sum = 0; i < maxi; i++) {
        int t = count[i];
        count[i] = sum;
        sum += t;
    }
    for (int i = 0; i < n; i++) {
        int idx = sa[i] + k < n ? ra[l][sa[i] + k] : 0;
        temp_sa[count[idx]++] = sa[i];
    }
    sa = temp_sa;
}

void build_SA() {
    for (int i = 0; i < n; i++)
        ra[0][i] = s[i];
    for (int i = 0; i < n; i++)
        sa[i] = i;
    for (int i = 0; i < __log_n - 1; i++) {
        int k = (1 << i);
        if (k >= n)
            break;
        __counting_sort(i, k);
        __counting_sort(i, 0);
        int rank = 0;
        ra[i + 1][sa[0]] = rank;
        for (int j = 1; j < n; j++)
            if (ra[i][sa[j]] == ra[i][sa[j - 1]] &&
                ra[i][sa[j] + k] == ra[i][sa[j - 1] + k])
                ra[i + 1][sa[j]] = rank;
            else
                ra[i + 1][sa[j]] = ++rank;
    }
}
void build_LCP() {
    _lcp = vector<vector<int>>(__log_n, vector<int>(n));
    for (int i = 0; i < n - 1; i++) { // Build the LCP
        array in O(NlogN)
        int x = sa[i], y = sa[i + 1], k, ret = 0;
        for (k = __log_n - 1; k >= 0 && x < n && y < n;
            k--) {
            if ((1 << k) >= n)
                continue;
            if (ra[k][x] == ra[k][y])
                x += 1 << k, y += 1 << k, ret += 1 << k;
        }
        if (ret >= __dollar[sa[i]] - sa[i])
            ret = __dollar[sa[i]] - sa[i];
        _lcp[0][i] = ret; // LCP[i] shouldnt exceed
            __dollar[sa[i]]
    } // __dollar[i] : index of __dollar to the right of
        i.
    _lcp[0][n - 1] = 10 * n;
    for (int i = 1; i < __log_n; i++) { // O(1) RMQ
        structure in O(NlogN)
        int add = (1 << (i - 1));
        if (add >= n)
            break; // small optimization
        for (int j = 0; j < n; j++)
            if (j + add < n)
                _lcp[i][j] = min(_lcp[i - 1][j], _lcp[i -
                    1][j + add]);
            else
                _lcp[i][j] = _lcp[i - 1][j];
    }
}

int lcp(int x, int y) {
    // O(1) LCP. x & y are indexes of the suffix in sa!
    if (x == y)
        return __dollar[sa[x]] - sa[x];
    if (x > y)
```

```
            swap(x, y);
        y--;
        int idx = __msb[y - x + 1], sub = (1 << idx);
        return min(_lcp[idx][x], _lcp[idx][y - sub + 1]);
    }

    bool equal(int i, int j, int p, int q) {
        if (j - i != q - p)
            return false;
        int idx = __msb[j - i + 1], sub = (1 << idx);
        return ra[idx][i] == ra[idx][p] &&
               ra[idx][j - sub + 1] == ra[idx][q - sub + 1];
    } // Note : Do not forget to add a terminating $
};

// To check substring/LCS, run the string on the automaton.
    Each path in the
// automaton is a substring(if it ends in a terminal node,
    it is a suffix). To
// find occurences of a string, run it on the automaton, and
    the number of its
// occurences would be number of ways to reach a terminal
    node. Or, we can keep
// reverse edges of suffix links(all prefixes for that
    substring), and number of
// ways to reach a root, would be the answer(can be used to
    print all answers)
struct AhoCorasick {
    vector<int> sufflink, out;
    vector<map<char, int>> trie; // call findnextstate
    AhoCorasick() {
        out.resize(1);
        trie.resize(1);
    }
    inline void insert(string &s) {
        int curr = 0; // clear to reinit
        for (int i = 0; i < s.size(); i++) {
            if (!trie[curr].count(s[i])) {
                trie[curr][s[i]] = trie.size();
                trie.push_back(map<char, int>());
                out.push_back(0);
            }
            curr = trie[curr][s[i]];
        }
        ++out[curr];
    }
    inline void build_automation() {
        sufflink.resize(trie.size());
        queue<int> q;
        for (auto x : trie[0]) {
            sufflink[x.second] = 0;
            q.push(x.second);
        }
        while (!q.empty()) {
            int curr = q.front();
            q.pop();
            for (auto x : trie[curr]) {
                q.push(x.second);
                int tmp = sufflink[curr];
                while (!trie[tmp].count(x.first) && tmp)
                    tmp = sufflink[tmp];
                if (trie[tmp].count(x.first))
                    sufflink[x.second] = trie[tmp][x.first];
                else
                    sufflink[x.second] = 0;
                out[x.second] += out[sufflink[x.second]];
            }
        }
    }
    int find_next_state(int curr, char ch) {
        while (curr && !trie[curr].count(ch))
            curr = sufflink[curr];
```

```
        return (!trie[curr].count(ch)) ? 0 : trie[curr][ch];
    }
    int query(string &s) {
        int ans = 0;
        int curr = 0;
        for (int i = 0; i < s.size(); i++) {
            curr = find_next_state(curr, s[i]);
            ans += out[curr];
        }
        return ans;
    }
    void clear() {
        trie.clear();
        sufflink.clear();
        out.clear();
        out.resize(1);
        trie.resize(1);
    }
};

// To check substring/LCS, run the string on the automaton.
    Each path in the
// automaton is a substring(if it ends in a terminal node,
    it is a suffix) To
// find occurences of a string, run it on the automaton, and
    the number of its
// occurences would be number of ways to reach a terminal
    node. Or, we can keep
// reverse edges of suffix links(all prefixes for that
    substring), and number of
// ways to reach a root, would be the answer(can be used to
    print all answers)
struct SuffixAutomaton {
    vector<map<char, int>> edges;
    vector<int> link, length; // length[i]: longest string
        in i-th class
    int last;               // index of equivalence class of
        whole string
    SuffixAutomaton(string s) {
        edges.push_back(map<char, int>());
        link.push_back(-1);
        length.push_back(0);
        last = 0;
        for (int i = 0; i < s.size(); i++) {
            edges.push_back(map<char, int>());
            length.push_back(i + 1);
            link.push_back(0);
            int r = edges.size() - 1;
            int p = last;
            while (p >= 0 && edges[p].find(s[i]) ==
                edges[p].end())
                edges[p][s[i]] = r, p = link[p];
            if (p != -1) {
                int q = edges[p][s[i]];
                if (length[p] + 1 == length[q])
                    link[r] = q;
                else {
                    edges.push_back(edges[q]);
                    length.push_back(length[p] + 1);
                    link.push_back(link[q]);
                    int qq = edges.size() - 1;
                    link[q] = qq;
                    link[r] = qq;
                    while (p >= 0 && edges[p][s[i]] == q)
                        edges[p][s[i]] = qq, p = link[p];
                }
            }
            last = r;
        }
        vector<int> terminals;
        int p = last;
        while (p > 0)
```

```
            terminals.push_back(p), p = link[p];
    }
};
```

## 14 TreesCentroids

```
#include "template.hpp"

class Tree {
  public:
    struct Node {
        vector<Node *> adjacent; Node *parent = nullptr;
        int start_time = 0, end_time = 0, subtree_size = 1;
        int depth = 0, height = 0, index = INT32_MAX;
    };
    vector<Node> list; Node *root = nullptr;
    vector<vector<Node *>> __anc;
    Tree(int n = 1e5) {
        list.resize(n); this->root = &list[0];
        for (int i = 0; i < n; i++) list[i].index = i;
    }
    void add_edge(int x, int y) {
        list[x].adjacent.push_back(&list[y]);
        list[y].adjacent.push_back(&list[x]);
    }
    Node *lca(Node *a, Node *b) {
        if (b->depth > a->depth) swap(a, b);
        for (int ptr = __anc[0].size() - 1; a->depth >
            b->depth && ptr >= 0; ptr--)
            if (__anc[a->index][ptr] != nullptr &&
                __anc[a->index][ptr]->depth >= b->depth)
                a = __anc[a->index][ptr];
        if (a == b) return a;
        for (long step = __anc[0].size() - 1; step >= 0;
            step--)
            if (__anc[a->index][step] !=
                __anc[b->index][step])
                a = __anc[a->index][step], b =
                    __anc[b->index][step];
        return a->parent;
    }
    Node *ancestor(Node *a, int degree) {
        ll target_depth = a->depth - degree;
        for (int ptr = __anc[0].size() - 1; a->depth >
            target_depth && ptr >= 0; ptr--)
            if (__anc[a->index][ptr] != nullptr &&
                __anc[a->index][ptr]->depth >= target_depth)
                a = __anc[a->index][ptr];
        return a;
    }
    int __build(Node *root = nullptr, int time = 0) {
        if (root == nullptr) root = this->root;
        root->start_time = time;
        for (auto child : root->adjacent) {
            if (child == root->parent) continue;
            child->parent = root;
            child->depth = root->depth + 1;
            time = __build(child, time + 1);
            root->height = max(root->height, child->height +
                1);
            root->subtree_size += child->subtree_size;
        }
        root->end_time = time;
        return time;
    }
    void __build_lca_matrix() {
        int n = list.size();
        __anc = *new vector<vector<Node *>>(n, vector<Node
            *>(log2(n) + 1, nullptr));
```

```
        for (int i = 0; i < list.size(); i++) __anc[i][0] =
            list[i].parent;
        for (int level = 1; level < __anc[0].size(); level++)
            for (int i = 0; i < list.size(); i++) {
                if (__anc[i][level - 1] == nullptr) continue;
                __anc[i][level] = __anc[__anc[i][level -
                    1]->index][level - 1];
            }
    }
};

struct CentroidTree : Tree {
    vector<bool> __visited; vector<int> __dir_parents,
        __subtree_size; Tree base;
    void __dfs_centroid(int node) {
        __subtree_size[node] = 1;
        for (Node *next : base.list[node].adjacent)
            if (!__visited[next->index] && next->index !=
                __dir_parents[node]) {
                __dir_parents[next->index] = node;
                __dfs_centroid(next->index);
                __subtree_size[node] +=
                    __subtree_size[next->index];
            }
    }
    int __get_centroid(int x) {
        __dir_parents[x] = 0; __dfs_centroid(x); int sz =
            __subtree_size[x];
        while (true) {
            pair<int, int> mx = {0, 0};
            for (Node *next : base.list[x].adjacent)
                if (!__visited[next->index] && next->index !=
                    __dir_parents[x])
                    mx = max(mx, {__subtree_size[next->index],
                        next->index});
            if (mx.first * 2 > sz) x = mx.second; else return
                x;
        }
    }
    void __build_centroid(int node, Node *parent) {
        node = __get_centroid(node); list[node].parent =
            parent; __visited[node] = true;
        for (Node *next : base.list[node].adjacent)
            if (!__visited[next->index])
                __build_centroid(next->index, &list[node]);
    }
    CentroidTree(Tree &tree) : Tree((int)tree.list.size()) {
        __visited = vector<bool>(tree.list.size());
        __subtree_size = vector<int>(tree.list.size());
        __dir_parents = vector<int>(tree.list.size());
        base = tree;
        __build_centroid(0, nullptr);
        for (auto el : list) {
            if (el.parent == nullptr) root = &list[el.index];
            else add_edge(el.index, el.parent->index);
        } __build(root);
    }
};

ll diameter(Tree tree) {
    ll n = tree.list.size() + 1; vbl visited(n + 1, false);
        vll distances(n + 1, -1);
    queue<pll> q; q.push({tree.root->index, 0});
    ll node_max = tree.root->index, distance_max = 0;
    while (!q.empty()) {
        auto node = q.front(); q.pop();
        if (node.second < distance_max) distance_max =
            node.second, node_max = node.first;
        for (auto neighbor : tree.list[node.first].adjacent)
            if (!visited[neighbor->index]) {
                auto d = node.second + 1;
```

```
                q.push({neighbor->index, d});
                visited[neighbor->index] = 1;
            }
    }
    visited = vbl(n + 1, false);
    q.push({node_max, 0}); distance_max = 0;
    while (!q.empty()) {
        auto node = q.front(); q.pop();
        maximize(distance_max, node.second);
        for (auto neighbor : tree.list[node.first].adjacent)
            if (!visited[neighbor->index]) {
                auto d = node.second + 1;
                q.push({neighbor->index, d});
                visited[neighbor->index] = 1;
            }
    }
}

struct HeavyLightDecomp : Tree {
    int chain_count = 1, narr;
    vector<int> subtree_size, chain, chain_head, chain_next;
    function<ll(int, int, ll)> answer;
    vector<int> pos;

    HeavyLightDecomp(int n, function<ll(int, int, ll)> &ans)
        : Tree(n) {
        subtree_size.resize(n); pos.resize(n);
            chain.resize(n);
        chain_head.resize(n); chain_next.resize(n); answer =
            ans;
    }

    void decompose(int node = 0, int parent = -1) {
        pos[node] = ++narr, chain[node] = chain_count; int
            big = 0;
        for (Node *adj : list[node].adjacent) {
            int u = adj->index;
            if (u == parent) continue;
            else if (!big) big = u;
            else if (subtree_size[u] > subtree_size[big]) big
                = u;
        }
        if (big) decompose(big, node);
        for (Node *adj : list[node].adjacent) {
            int u = adj->index;
            if (u == parent || u == big) continue;
            ++chain_count, chain_head[chain_count] = u,
                        chain_next[chain_count] = node;
            decompose(u, node);
        }
    }
    // Build Segment Tree using indices of pos array
    // Update ans using Range queries on said segment tree
    int query_up(int r, int q) {
        int ans = 0, t;
        while (chain[q] != chain[r]) {
            t = chain[q];
            ans = answer(pos[chain_head[t]], pos[q], ans);
            q = chain_next[t];
        }
        ans = answer(pos[r], pos[q], ans);
        return ans;
    }
};
```