Team notebook

December 12, 2024

Contents

1 AhoCorasick

```
const int ALP = 26;

struct Node{
	Node *link;
	Node *next[ALP];
	int cnt = 0;
};

struct AhoCorasick{
	Node* root;
	AhoCorasick() {
		root = new Node();
		root->link = root;
	}
	void addWord(string s) {
		Node* cur = root;
		for(char c : s) {
			int id = c - 'a';
			if(cur->next[id] == nullptr) {
				cur->next[id] = new Node();
			}
			cur = cur->next[id];
		}
		cur->cnt ++;
```

```
	}
	void buildSuffix() {
		queue<Node*> qu;
		for(int i = 0; i < ALP; i ++) {
			if(root->next[i] == nullptr) {
				root->next[i] = root;
				continue;
			}
			root->next[i]->link = root;
			qu.push(root->next[i]);
		}
		while(!qu.empty()){
			Node* u = qu.front();
			qu.pop();
			for(int i = 0; i < ALP; i ++) {
				Node* v = u->next[i];
				if(v != nullptr) {
					Node* p = u->link;
					while(p != root && p->next[i] == nullptr) p = p->link;
					v->link = p->next[i];
					v->cnt += v->link->cnt;
					qu.push(v);
				} else {
					Node* p = u->link;
					while(p != root && p->next[i] == nullptr) p = p->link;
					u->next[i] = p->next[i];
				}
			}
		}
	}
};

string t, s[N];
int n, m, pref[N], suf[N];

void Lalisa(){
	cin >> t >> m;
	n = t.size();
	t = " " + t + " ";
	AhoCorasick* AC = new AhoCorasick();
	for(int i = 1; i <= m; i ++) cin >> s[i], AC->addWord(s[i]);
	AC->buildSuffix();
	Node* cur = AC->root;
	for(int i = 1; i <= n; i ++) {
		cur = cur->next[ t[i] - 'a' ];
		pref[i] = cur->cnt;
	}
	delete AC;
	AC = new AhoCorasick();
	for(int i = 1; i <= m; i ++) {
		reverse(s[i].begin(), s[i].end());
		AC->addWord(s[i]);
	}
	AC->buildSuffix();
	cur = AC->root;
	for(int i = n; i >= 1; i --) {
		cur = cur->next[t[i] - 'a'];
		suf[i] = cur->cnt;
	}
	ll res = 0;
	for(int i = 1; i < n; i ++) res += pref[i] * 1ll * suf[i + 1];
	cout << res << "\n";
}
```

2 BasicSuffixArray

```
#define all(a) (a).begin(), (a).end()
#define uni(a) (a).erase(unique(all(a)), (a).end())

struct suffix{
	int id, sL, sR;
	suffix(int _id = 0, int _sL = 0, int _sR = 0) : id(_id), sL(_sL), sR(_sR) {}
	bool operator < (const suffix& other) const{
		if(sL != other.sL) return sL < other.sL;
		return sR < other.sR;
	}
	bool operator == (const suffix& other) const{
		return sL == other.sL && sR == other.sR;
	}
};

int P[logN][N], cnt[N], Rank[N];

void radixSort(vector<suffix> &a, int n){
	vector<suffix> tmp(n);
	for(int i = 0; i < n; i ++) cnt[a[i].sR] ++;
	for(int i = 1; i < n; i ++) cnt[i] += cnt[i - 1];
	for(int i = n - 1; i >= 0; i --) tmp[-- cnt[a[i].sR]] = a[i];

	for(int i = 0; i < n; i ++) cnt[i] = 0;
	a = tmp;

	for(int i = 0; i < n; i ++) cnt[a[i].sL] ++;
	for(int i = 1; i < n; i ++) cnt[i] += cnt[i - 1];
	for(int i = n - 1; i >= 0; i --) tmp[-- cnt[a[i].sL]] = a[i];

	for(int i = 0; i < n; i ++) cnt[i] = 0;
	a = tmp;
}

vector<int> suffixArray(const string &s, int n){
	vector<char> ord(s.begin(), s.end());
	sort(all(ord));
```

```
    uni(ord);
    for(int i = 0; i < n; i ++){
        P[0][i] = lower_bound(all(ord), s[i]) - ord.begin();
    }
    vector<int> sufA(n);
    for(int i = 1; i < logN; i ++){
        vector<suffix> a;
        for(int j = 0; j < n; j ++) a.push_back( suffix(j, P[i - 1][j], P[i - 1][(j
            + (1 << i - 1)) % n]) );
        radixSort(a, n);
        int classes = 0;
        for(int j = 0; j < n; j ++){
            if(j > 0 && a[j - 1] < a[j]) classes ++;
            P[i][a[j].id] = classes;
        }
        for(int j = 0; j < n; j ++) sufA[j] = a[j].id;
    }
    return sufA;
}

vector<int> buildLCP(const vector<int>& sufA, const string& s, int n){
    vector<int> LCP(n - 1);
    for(int i = 0; i < n; i ++) Rank[sufA[i]] = i;
    int k = 0;
    for(int i = 0; i < n - 1; i ++){
        int j = sufA[Rank[i] - 1];
        while(i + k < n && j + k < n && s[i + k] == s[j + k]) k ++;
        LCP[Rank[i] - 1] = k;
        if(k > 0) k --;
    }
    return LCP;
}

void Lalisa(){
    string s, t;
    cin >> s >> t;
    string a = s + '#' + t + '$';
    int n = a.size(), S = s.size(), T = t.size();
    vector<int> suf = suffixArray(a, n);
    vector<int> lcp = buildLCP(suf, a, n);
}
```

3 BiconnectedComponent

```
// Input graph: vector< vector<int> > a, int n
// Note: 0-indexed
// Usage: BiconnectedComponent bc; (bc.components is the list of components)
//
// This is biconnected components by edges (1 vertex can belong to
// multiple components). For vertices biconnected component, remove
// bridges and find components
int n;
vector<vector<int>> g;
struct BiconnectedComponent {
    vector<int> low, num, s;
    vector< vector<int> > components;
    int counter;

    BiconnectedComponent() : low(n, -1), num(n, -1), counter(0) {
        for (int i = 0; i < n; i++)
            if (num[i] < 0)
                dfs(i, 1);
    }

    void dfs(int x, int isRoot) {
        low[x] = num[x] = ++counter;
        if (g[x].empty()) {
            components.push_back(vector<int>(1, x));
            return;
        }
        s.push_back(x);

        for (int i = 0; i < (int) g[x].size(); i++) {
            int y = g[x][i];
            if (num[y] > -1) low[x] = min(low[x], num[y]);
            else {
                dfs(y, 0);
                low[x] = min(low[x], low[y]);

                if (isRoot || low[y] >= num[x]) {
                    components.push_back(vector<int>(1, x));
                    while (1) {
                        int u = s.back();
                        s.pop_back();
                        components.back().push_back(u);
                        if (u == y) break;
                    }
                }
            }
        }
    }
};
```

4 BridgeArticulation

```
// UndirectedDFS, for finding bridges & articulation points {{{
// Assume already have undirected graph vector< vector<int> > G with V vertices
// Vertex index from 0
// Usage:
// UndirectedDfs tree;
```

```
// Then you can use tree.bridges and tree.articulation_points
//
// Tested:
// - https://judge.yosupo.jp/problem/two_edge_connected_components
struct UndirectedDfs {
    vector<vector<int>> g;
    int n;
    vector<int> low, num, parent;
    vector<bool> is_articulation;
    int counter, root, children;

    vector< pair<int,int> > bridges;
    vector<int> articulation_points;
    map<pair<int,int>, int> cnt_edges;

    UndirectedDfs(const vector<vector<int>>& _g) : g(_g), n(g.size()),
           low(n, 0), num(n, -1), parent(n, 0), is_articulation(n, false),
           counter(0), children(0) {
        for (int u = 0; u < n; u++) {
            for (int v : g[u]) {
                cnt_edges[{u, v}] += 1;
            }
        }
        for(int i = 0; i < n; ++i) if (num[i] == -1) {
            root = i; children = 0;
            dfs(i);
            is_articulation[root] = (children > 1);
        }
        for(int i = 0; i < n; ++i)
            if (is_articulation[i]) articulation_points.push_back(i);
    }
private:
    void dfs(int u) {
        low[u] = num[u] = counter++;
        for (int v : g[u]) {
            if (num[v] == -1) {
                parent[v] = u;
                if (u == root) children++;
                dfs(v);
                if (low[v] >= num[u])
                    is_articulation[u] = true;
                if (low[v] > num[u]) {
                    if (cnt_edges[{u, v}] == 1) {
                        bridges.push_back(make_pair(u, v));
                    }
                }
                low[u] = min(low[u], low[v]);
            } else if (v != parent[u])
                low[u] = min(low[u], num[v]);
        }
    }
};
```

5 ConvexHullIT

```
struct Line {
    long long a, b; // y = ax + b
    Line(long long a = 0, long long b = -INF): a(a), b(b) {}
    long long eval(long long x) {
        return a * x + b;
    }
};
struct Node {
    Line line;
    int l, r;
    Node *left, *right;
    Node(int l, int r): l(l), r(r), left(NULL), right(NULL), line() {}
    void update(int i, int j, Line newLine) {
        if (r < i || j < l) return;
        if (i <= l && r <= j) {
            Line AB = line, CD = newLine;
            if (AB.eval(valueX[l]) < CD.eval(valueX[l])) swap(AB, CD);
            if (AB.eval(valueX[r]) >= CD.eval(valueX[r])) {
                line = AB;
                return;
            }
            int mid = valueX[l + r >> 1];
            if (AB.eval(mid) < CD.eval(mid))
                line = CD, left->update(i, j, AB);
            else
                line = AB, right->update(i, j, CD);
            return;
        }
        left->update(i, j, newLine);
        right->update(i, j, newLine);
    }
    long long getMax(int i) {
        if (l == r) return line.eval(valueX[i]);
        if (i <= (l + r >> 1)) return max(line.eval(valueX[i]), left->getMax(
        i));
        return max(line.eval(valueX[i]), right->getMax(i));
    }
};
Node* build(int l, int r) {
    Node *x = new Node(l, r);
    if (l == r) return x;
    x->left = build(l, l + r >> 1);
    x->right = build((l + r >> 1) + 1, r);
    return x;
}
```

6 EulerPathDirected

```cpp
struct EulerDirected {
    EulerDirected(int _n) : n(_n), adj(n), in_deg(n, 0), out_deg(n, 0) {}

    void add_edge(int u, int v) { // directed edge
        assert(0 <= u && u < n);
        assert(0 <= v && v < n);
        adj[u].push_front(v);
        in_deg[v]++;
        out_deg[u]++;
    }

    std::pair<bool, std::vector<int>> solve() {
        int start = -1, last = -1;
        for (int i = 0; i < n; i++) {
            if (std::abs(in_deg[i] - out_deg[i]) > 1) return {false, {}};

            if (out_deg[i] > in_deg[i]) {
                if (start >= 0) return {false, {}};
                start = i;
            }

            if (in_deg[i] > out_deg[i]) {
                if (last >= 0) return {false, {}};
                last = i;
            }
        }

        if (start < 0) {
            for (int i = 0; i < n; i++) {
                if (in_deg[i]) {
                    start = i;
                    break;
                }
            }
            if (start < 0) return {true, {}};
        }

        std::vector<int> path;
        find_path(start, path);
        std::reverse(path.begin(), path.end());

        // check that we visited all vertices with degree > 0
        std::vector<bool> visited(n, false);
        for (int u : path) visited[u] = true;

        for (int u = 0; u < n; u++) {
            if (in_deg[u] && !visited[u]) {
                return {false, {}};
            }
        }

        return {true, path};
    }

private:
    int n;
    std::vector<std::list<int>> adj;
    std::vector<int> in_deg, out_deg;

    void find_path(int v, std::vector<int>& path) {
        while (adj[v].size() > 0) {
            int next = adj[v].front();
            adj[v].pop_front();
            find_path(next, path);
        }
        path.push_back(v);
    }
};
```

7 Gaussian

```cpp
// Gauss-Jordan elimination.
// Returns: number of solution (0, 1 or INF)
//   When the system has at least one solution, ans will contains
//   one possible solution
// Possible improvement when having precision errors:
//   - Divide i-th row by a(i, i)
//   - Choosing pivoting row with min absolute value (sometimes this is better that
    maximum, as implemented here)
int gauss (vector < vector<double> > a, vector<double> & ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }
```

```
    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

8 HungarianLMH

```
// Index from 1
// Min cost matching
// Usage: init(); for[i,j,cost] addEdge(i, j, cost)

#define arg __arg
long long c[MN][MN];
long long fx[MN], fy[MN];
int mx[MN], my[MN];
int trace[MN], qu[MN], arg[MN];
long long d[MN];
int front, rear, start, finish;

void init() {
    FOR(i,1,N) {
        fy[i] = mx[i] = my[i] = 0;
        FOR(j,1,N) c[i][j] = inf;
    }
}

void addEdge(int i, int j, long long cost) {
    c[i][j] = min(c[i][j], cost);
}

inline long long getC(int i, int j) {
    return c[i][j] - fx[i] - fy[j];
}

void initBFS() {
    front = rear = 1;
    qu[1] = start;
    memset(trace, 0, sizeof trace);
    FOR(j,1,N) {
        d[j] = getC(start, j);
        arg[j] = start;
    }
    finish = 0;
}

void findAugPath() {
    while (front <= rear) {
        int i = qu[front++];
        FOR(j,1,N) if (!trace[j]) {
            long long w = getC(i, j);
            if (!w) {
                trace[j] = i;
                if (!my[j]) {
                    finish = j;
                    return ;
                }
                qu[++rear] = my[j];
            }
            if (d[j] > w) {
                d[j] = w;
                arg[j] = i;
            }
        }
    }
}

void subx_addy() {
    long long delta = inf;
    FOR(j,1,N)
        if (trace[j] == 0 && d[j] < delta) delta = d[j];

    // xoay
    fx[start] += delta;
    FOR(j,1,N)
        if (trace[j]) {
            int i = my[j];
            fy[j] -= delta;
            fx[i] += delta;
        }
        else d[j] -= delta;

    FOR(j,1,N)
        if (!trace[j] && !d[j]) {
            trace[j] = arg[j];
            if (!my[j]) { finish = j; return ; }
            qu[++rear] = my[j];
        }
}
```

```
void enlarge() {
    do {
        int i = trace[finish];
        int next = mx[i];
        mx[i] = finish;
        my[finish] = i;
        finish = next;
    } while (finish);
}

long long mincost() {
    FOR(i,1,N) fx[i] = *min_element(c[i]+1, c[i]+N+1);
    FOR(j,1,N) {
        fy[j] = c[1][j] - fx[1];
        FOR(i,1,N) {
            fy[j] = min(fy[j], c[i][j] - fx[i]);
        }
    }
    FOR(i,1,N) {
        start = i;
        initBFS();
        while (finish == 0) {
            findAugPath();
            if (!finish) subx_addy();
        }
        enlarge();
    }
    long long res = 0;
    FOR(i,1,N) res += c[i][mx[i]];
    return res;
}
```

9 Interpolation

```
const int N = 1e6 + 5, logN = 20;
const int MOD = 1e9 + 7;
inline ll sqr(int x) {return x * 1ll * x;}
inline int fpow(ll n, ll k, int p = MOD) {ll r = 1; for (; k; k >>= 1) {if (k & 1) r =
    r * n % p; n = n * n % p;} return r;}
inline void addmod(int& a, int val, int p = MOD) {if ((a = (a + val)) >= p) a -= p;}
inline void submod(int& a, int val, int p = MOD) {if ((a = (a - val)) < 0) a += p;}
inline int mult(int a, int b, int p = MOD) {return (ll) a * b % p;}

inline int inverse(int x){
      return fpow(x, MOD - 2);
}

int f[N], n, k, ft[N];

int F(int x){
      if(x <= k + 1) return f[x];
      int fact = 1;
      for(int i = 0; i <= k + 1; i ++) fact = mult(fact, x - i);
      int ans = 0;
      for(int i = 0; i <= k + 1; i ++){
            int num = mult( f[i], mult( fact, inverse(x - i) ) ); // num
            int dem = mult(ft[i], ft[k + 1 - i]);
            if((k - i + 1) % 2 == 1) dem = (MOD - dem) % MOD; // mult with pow(-1, k -
                i + 1);
            addmod( ans, mult(num, inverse(dem)) );
      }
      return ans;
}
void Lalisa(){
      cin >> n >> k;
      ft[0] = 1;
      for(int i = 1; i <= k + 1; i ++){
          f[i] = (f[i - 1] + fpow(i, k)) % MOD;
          ft[i] = mult(ft[i - 1], i);
      }
      cout << F(n) << "\n";
}
```

10 Judge

```
#include <bits/stdc++.h>
using namespace std;
const string NAME = "Codeforces";

const int NTEST = 100;

mt19937_64 rd(chrono::steady_clock::now().time_since_epoch().count());
#define rand rd

long long Rand(long long l, long long h) {
    assert(l <= h);
    return l + rd() % (h - l + 1);
}

int main()
{
    srand(time(NULL));
    for (int iTest = 1; iTest <= NTEST; iTest++)
    {
        ofstream inp((NAME + ".inp").c_str());


        inp.close();
```

```
        system((NAME + ".exe").c_str());
        system((NAME + "_trau.exe").c_str());

        if (system(("fc " + NAME + ".out " + NAME + ".ans").c_str()) != 0)
        {
            cout << "Test " << iTest << ": WRONG!\n";
            return 0;
        }
        cout << "Test " << iTest << ": CORRECT!\n";
    }
    return 0;
}
```

11 MaxFlowDinic

```
// Source: e-maxx.ru
// Tested with: VOJ - NKFLOW, VOJ - MCQUERY (Gomory Hu)

// Usage:
// MaxFlow flow(n)
// For each edge: flow.addEdge(u, v, c)
// Index from 0
const int INF = 1000000000;

struct Edge {
    int a, b, cap, flow;
};

struct MaxFlow {
    int n, s, t;
    vector<int> d, ptr, q;
    vector< Edge > e;
    vector< vector<int> > g;

    MaxFlow(int _n) : n(_n), d(_n), ptr(_n), q(_n), g(_n) {
        e.clear();
        for (int i = 0; i < n; i++) {
            g[i].clear();
            ptr[i] = 0;
        }
    }

    void addEdge(int a, int b, int cap) {
        Edge e1 = { a, b, cap, 0 };
        Edge e2 = { b, a, 0, 0 };
        g[a].push_back( (int) e.size() );
        e.push_back(e1);
        g[b].push_back( (int) e.size() );
        e.push_back(e2);
    }

    int getMaxFlow(int _s, int _t) {
        s = _s; t = _t;
        int flow = 0;
        for (;;) {
            if (!bfs()) break;
            std::fill(ptr.begin(), ptr.end(), 0);
            while (int pushed = dfs(s, INF))
                flow += pushed;
        }
        return flow;
    }

private:
    bool bfs() {
        int qh = 0, qt = 0;
        q[qt++] = s;
        std::fill(d.begin(), d.end(), -1);
        d[s] = 0;

        while (qh < qt && d[t] == -1) {
            int v = q[qh++];
            for (int i = 0; i < (int) g[v].size(); i++) {
                int id = g[v][i], to = e[id].b;
                if (d[to] == -1 && e[id].flow < e[id].cap) {
                    q[qt++] = to;
                    d[to] = d[v] + 1;
                }
            }
        }
        return d[t] != -1;
    }

    int dfs (int v, int flow) {
        if (!flow) return 0;
        if (v == t) return flow;
        for (; ptr[v] < (int)g[v].size(); ++ptr[v]) {
            int id = g[v][ptr[v]],
                to = e[id].b;
            if (d[to] != d[v] + 1) continue;
            int pushed = dfs(to, min(flow, e[id].cap - e[id].flow));
            if (pushed) {
                e[id].flow += pushed;
                e[id^1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }
};
```

12 MinCostMaxFlowPR

```cpp
// Source:
    https://github.com/dacin21/dacin21_codebook/blob/master/flow/mincost_PRonly.cpp
//
// Notes:
// - Index from 0
// - Costs multiplied by N --> overflow when big cost?
// - Does not work with floating point..
// - DO NOT USE Edge.f DIRECTLY. Call getFlow(e)
template<typename flow_t = int, typename cost_t = int>
struct MinCostFlow {
    struct Edge {
        cost_t c;
        flow_t f; // DO NOT USE THIS DIRECTLY. SEE getFlow(Edge const& e)
        int to, rev;
        Edge(int _to, cost_t _c, flow_t _f, int _rev) : c(_c), f(_f), to(_to), rev(_rev)
            {}
    };

    int N, S, T;
    vector<vector<Edge> > G;
    MinCostFlow(int _N, int _S, int _T) : N(_N), S(_S), T(_T), G(_N), eps(0) {}

    void addEdge(int a, int b, flow_t cap, cost_t cost) {
        assert(cap >= 0);
        assert(a >= 0 && a < N && b >= 0 && b < N);
        if (a == b) { assert(cost >= 0); return; }
        cost *= N;
        eps = max(eps, abs(cost));
        G[a].emplace_back(b, cost, cap, G[b].size());
        G[b].emplace_back(a, -cost, 0, G[a].size() - 1);
    }

    flow_t getFlow(Edge const &e) {
        return G[e.to][e.rev].f;
    }

    pair<flow_t, cost_t> minCostMaxFlow() {
        cost_t retCost = 0;
        for (int i = 0; i<N; ++i) {
            for (Edge &e : G[i]) {
                retCost += e.c*(e.f);
            }
        }
        //find max-flow
        flow_t retFlow = max_flow();
        h.assign(N, 0); ex.assign(N, 0);
        isq.assign(N, 0); cur.assign(N, 0);
        queue<int> q;
        for (; eps; eps >>= scale) {
            //refine
            fill(cur.begin(), cur.end(), 0);
            for (int i = 0; i < N; ++i) {
                for (auto &e : G[i]) {
                    if (h[i] + e.c - h[e.to] < 0 && e.f) push(e, e.f);
                }
            }
            for (int i = 0; i < N; ++i) {
                if (ex[i] > 0){
                    q.push(i);
                    isq[i] = 1;
                }
            }
            // make flow feasible
            while (!q.empty()) {
                int u = q.front(); q.pop();
                isq[u]=0;
                while (ex[u] > 0) {
                    if (cur[u] == G[u].size()) {
                        relabel(u);
                    }
                    for (unsigned int &i=cur[u], max_i = G[u].size(); i < max_i; ++i) {
                        Edge &e = G[u][i];
                        if (h[u] + e.c - h[e.to] < 0) {
                            push(e, ex[u]);
                            if (ex[e.to] > 0 && isq[e.to] == 0) {
                                q.push(e.to);
                                isq[e.to] = 1;
                            }
                            if (ex[u] == 0) break;
                        }
                    }
                }
            }
            if (eps > 1 && eps>>scale == 0) {
                eps = 1<<scale;
            }
        }
        for (int i = 0; i < N; ++i) {
            for (Edge &e : G[i]) {
                retCost -= e.c*(e.f);
            }
        }
        return make_pair(retFlow, retCost / 2 / N);
    }

private:
    static constexpr cost_t INFCOST = numeric_limits<cost_t>::max()/2;
    static constexpr int scale = 2;

    cost_t eps;
    vector<unsigned int> isq, cur;
    vector<flow_t> ex;
    vector<cost_t> h;
```

```
vector<vector<int> > hs;
vector<int> co;

void add_flow(Edge& e, flow_t f) {
    Edge &back = G[e.to][e.rev];
    if (!ex[e.to] && f) {
        hs[h[e.to]].push_back(e.to);
    }
    e.f -= f; ex[e.to] += f;
    back.f += f; ex[back.to] -= f;
}

void push(Edge &e, flow_t amt) {
    if (e.f < amt) amt = e.f;
    e.f -= amt; ex[e.to] += amt;
    G[e.to][e.rev].f += amt; ex[G[e.to][e.rev].to] -= amt;
}

void relabel(int vertex){
    cost_t newHeight = -INFCOST;
    for (unsigned int i = 0; i < G[vertex].size(); ++i){
        Edge const&e = G[vertex][i];
        if(e.f && newHeight < h[e.to] - e.c){
            newHeight = h[e.to] - e.c;
            cur[vertex] = i;
        }
    }
    h[vertex] = newHeight - eps;
}

flow_t max_flow() {
    ex.assign(N, 0);
    h.assign(N, 0); hs.resize(2*N);
    co.assign(2*N, 0); cur.assign(N, 0);
    h[S] = N;
    ex[T] = 1;
    co[0] = N-1;
    for (auto &e : G[S]) {
        add_flow(e, e.f);
    }
    if (hs[0].size()) {
        for (int hi = 0; hi>=0;) {
            int u = hs[hi].back();
            hs[hi].pop_back();
            while (ex[u] > 0) { // discharge u
                if (cur[u] == G[u].size()) {
                    h[u] = 1e9;
                    for(unsigned int i = 0; i < G[u].size(); ++i) {
                        auto &e = G[u][i];
                        if (e.f && h[u] > h[e.to]+1) {
                            h[u] = h[e.to]+1, cur[u] = i;
                        }
                    }
                    if (++co[h[u]], !--co[hi] && hi < N) {
                        for (int i = 0; i < N; ++i) {
                            if (hi < h[i] && h[i] < N) {
                                --co[h[i]];
                                h[i] = N + 1;
                            }
                        }
                    }
                    hi = h[u];
                } else if (G[u][cur[u]].f && h[u] == h[G[u][cur[u]].to]+1) {
                    add_flow(G[u][cur[u]], min(ex[u], G[u][cur[u]].f));
                } else {
                    ++cur[u];
                }
            }
            while (hi>=0 && hs[hi].empty()) {
                --hi;
            }
        }
    }
    return -ex[S];
}
};
```

13 OrderSetMap

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
/*
change null_type to int if we want to use map instead
find_by_order(k) returns an iterator to the k-th element (0-indexed)
order_of_key(k) returns numbers of item being strictly smaller than k
*/
template<typename T = int>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;
//==============
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().
        time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```

```
unordered_map<long long, int, custom_hash> safe_map;
gp_hash_table<long long, int, custom_hash> safe_hash_table;
```

14 PalindromeTree

```
const int MAXN = 105000;

struct node {
    int next[26];
    int len;
    int sufflink;
    int num;
};

int len;
char s[MAXN];
node tree[MAXN];
int num;          // node 1 - root with len -1, node 2 - root with len 0
int suff;         // max suffix palindrome
long long ans;

bool addLetter(int pos) {
    int cur = suff, curlen = 0;
    int let = s[pos] - 'a';

    while (true) {
        curlen = tree[cur].len;
        if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos])
            break;
        cur = tree[cur].sufflink;
    }
    if (tree[cur].next[let]) {
        suff = tree[cur].next[let];
        return false;
    }

    num++;
    suff = num;
    tree[num].len = tree[cur].len + 2;
    tree[cur].next[let] = num;

    if (tree[num].len == 1) {
        tree[num].sufflink = 2;
        tree[num].num = 1;
        return true;
    }

    while (true) {
        cur = tree[cur].sufflink;
        curlen = tree[cur].len;
        if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
            tree[num].sufflink = tree[cur].next[let];
            break;
        }
    }

    tree[num].num = 1 + tree[tree[num].sufflink].num;

    return true;
}

void initTree() {
    num = 2; suff = 2;
    tree[1].len = -1; tree[1].sufflink = 1;
    tree[2].len = 0; tree[2].sufflink = 1;
}

int main() {
    gets(s);
    len = strlen(s);

    initTree();

    for (int i = 0; i < len; i++) {
        addLetter(i);
        ans += tree[suff].num;
    }

    cout << ans << endl;

    return 0;
}
```

15 RabinMiller

```
// From https://github.com/SnapDragon64/ContestLibrary/blob/master/math.h
// which also has specialized versions for 32-bit and 42-bit

inline uint64_t mod_mult64(uint64_t a, uint64_t b, uint64_t m) {
    return __int128_t(a) * b % m;
}
uint64_t mod_pow64(uint64_t a, uint64_t b, uint64_t m) {
    uint64_t ret = (m > 1);
    for (;;) {
        if (b & 1) ret = mod_mult64(ret, a, m);
        if (!(b >>= 1)) return ret;
        a = mod_mult64(a, a, m);
    }
}
```

```cpp
// Works for all primes p < 2^64
bool is_prime(uint64_t n) {
    if (n <= 3) return (n >= 2);
    static const uint64_t small[] = {
        2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
        71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139,
        149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199,
    };
    for (size_t i = 0; i < sizeof(small) / sizeof(uint64_t); ++i) {
        if (n % small[i] == 0) return n == small[i];
    }

    // Makes use of the known bounds for Miller-Rabin pseudoprimes.
    static const uint64_t millerrabin[] = {
        2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
    };
    static const uint64_t A014233[] = { // From OEIS.
        2047LL, 1373653LL, 25326001LL, 3215031751LL, 2152302898747LL,
        3474749660383LL, 341550071728321LL, 341550071728321LL,
        3825123056546413051LL, 3825123056546413051LL, 382512305654641305 1LL, 0,
    };
    uint64_t s = n-1, r = 0;
    while (s % 2 == 0) {
        s /= 2;
        r++;
    }
    for (size_t i = 0, j; i < sizeof(millerrabin) / sizeof(uint64_t); i++) {
        uint64_t md = mod_pow64(millerrabin[i], s, n);
        if (md != 1) {
            for (j = 1; j < r; j++) {
                if (md == n-1) break;
                md = mod_mult64(md, md, n);
            }
            if (md != n-1) return false;
        }
        if (n < A014233[i]) return true;
    }
    return true;
}
```

16 SegmentTreeBeats

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long
#define FOR(i, a, b) for (int i = (a), _##i = (b); i <= _##i; ++i)
#define REP(i, a) for (int i = 0, _##i = (a); i < _##i; ++i)

struct Node {
    int max1; // max value
    int max2; // 2nd max value (must be different from max1)
    int cnt_max; // how many indices have value == max1
    int sum;
    int lazy;

    Node() {}
    Node(int val) { // initialize with a single number.
        max1 = val;
        max2 = -1; // Note that values are in [0, 2^31), so -1 works here.
        cnt_max = 1;
        sum = val;
        lazy = -1; // Note that values are in [0, 2^31), so -1 works here.
    }

    void setMin(int val) { // for each i, set a[i] = min(a[i], val)
        assert(val > max2);

        if (max1 <= val) return;

        // Sample: 1 3 5 8 8 --> 1 3 5 6 6
        sum -= (max1 - val) * cnt_max;
        lazy = val;
        max1 = val;
    }
} it[8000111];

Node operator + (const Node& a, const Node& b) {
    Node res;
    res.max1 = max(a.max1, b.max1);

    res.max2 = max(a.max2, b.max2);
    if (a.max1 != res.max1) res.max2 = max(res.max2, a.max1);
    if (b.max1 != res.max1) res.max2 = max(res.max2, b.max1);

    res.cnt_max = 0;
    if (a.max1 == res.max1) res.cnt_max += a.cnt_max;
    if (b.max1 == res.max1) res.cnt_max += b.cnt_max;

    res.sum = a.sum + b.sum;
    res.lazy = -1;
    return res;
}

void down(int i) {
    if (it[i].lazy < 0) return;

    it[i*2].setMin(it[i].lazy);
    it[i*2+1].setMin(it[i].lazy);

    it[i].lazy = -1;
}
```

```
int a[1000111];
void build(int i, int l, int r) {
    if (l == r) {
        it[i] = Node(a[l]);
        return;
    }
    int mid = (l + r) / 2;
    build(i*2, l, mid);
    build(i*2 + 1, mid + 1, r);

    it[i] = it[i*2] + it[i*2 + 1];
}

void setMin(int i, int l, int r, int u, int v, int x) {
    if (v < l || r < u) return;
    if (it[i].max1 <= x) return;
    // now max1 > x

    if (u <= l && r <= v && it[i].max2 < x) {
        it[i].setMin(x);
        return;
    }

    down(i);
    int mid = (l + r) / 2;
    setMin(i*2, l, mid, u, v, x);
    setMin(i*2 + 1, mid+1, r, u, v, x);
    it[i] = it[i*2] + it[i*2 + 1];
}

int getMax(int i, int l, int r, int u, int v) {
    if (v < l || r < u) return -1;
    if (u <= l && r <= v) return it[i].max1;

    down(i);
    int mid = (l + r) / 2;
    return max(getMax(i*2, l, mid, u, v),
           getMax(i*2+1, mid+1, r, u, v));
}

int getSum(int i, int l, int r, int u, int v) {
    if (v < l || r < u) return 0;
    if (u <= l && r <= v) return it[i].sum;

    down(i);
    int mid = (l + r) / 2;
    return getSum(i*2, l, mid, u, v) + getSum(i*2+1, mid+1, r, u, v);
}

int32_t main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    // read initial array
    int n; cin >> n;
    FOR(i,1,n) cin >> a[i];

    // initialize segment tree beats
    build(1, 1, n);

    // queries
    int q; cin >> q;
    while (q--) {
        int typ; cin >> typ;
        if (typ == 1) { // for each i in [l, r] set a[i] = min(a[i], x)
            int l, r, x; cin >> l >> r >> x;
            setMin(1, 1, n, l, r, x);
        } else if (typ == 2) { // find max(a[i]) for i in [l, r]
            int l, r; cin >> l >> r;
            cout << getMax(1, 1, n, l, r) << '\n';
        } else { // find sum(a[i]) for i in [l, r]
            int l, r; cin >> l >> r;
            cout << getSum(1, 1, n, l, r) << '\n';
        }
    }
    return 0;
}
```

17 StronglyConnected

```
// Index from 0
// Usage:
// DirectedDfs tree;
// Now you can use tree.scc
//
// Note: reverse(tree.scc) is topo sorted
struct DirectedDfs {
    vector<vector<int>> g;
    int n;
    vector<int> num, low, current, S;
    int counter;
    vector<int> comp_ids;
    vector< vector<int> > scc;

    DirectedDfs(const vector<vector<int>>& _g) : g(_g), n(g.size()),
            num(n, -1), low(n, 0), current(n, 0), counter(0), comp_ids(n, -1) {
        for (int i = 0; i < n; i++) {
            if (num[i] == -1) dfs(i);
        }
    }

    void dfs(int u) {
        low[u] = num[u] = counter++;
        S.push_back(u);
```

```
        current[u] = 1;
        for (auto v : g[u]) {
            if (num[v] == -1) dfs(v);
            if (current[v]) low[u] = min(low[u], low[v]);
        }
        if (low[u] == num[u]) {
            scc.push_back(vector<int>());
            while (1) {
                int v = S.back(); S.pop_back(); current[v] = 0;
                scc.back().push_back(v);
                comp_ids[v] = ((int) scc.size()) - 1;
                if (u == v) break;
            }
        }
    }

    // build DAG of strongly connected components
    // Returns: adjacency list of DAG
    std::vector<std::vector<int>> build_scc_dag() {
        std::vector<std::vector<int>> dag(scc.size());
        for (int u = 0; u < n; u++) {
            int x = comp_ids[u];
            for (int v : g[u]) {
                int y = comp_ids[v];
                if (x != y) {
                    dag[x].push_back(y);
                }
            }
        }
        return dag;
    }
};
```

18 SuffixArray

```
// Efficient O(N + alphabet_size) time and space suffix array
// For ICPC notebook, it's better to copy a shorter code such as
//
    https://github.com/kth-competitive-programming/kactl/blob/main/content/strings/SuffixArray.h

// Usage:
// - sa = suffix_array(s, 'a', 'z')
// - lcp = LCP(s, sa)
//   lcp[i] = LCP(sa[i], sa[i+1])
void induced_sort(const std::vector<int>& vec, int val_range,
                  std::vector<int>& SA, const std::vector<bool>& sl,
                  const std::vector<int>& lms_idx) {
    std::vector<int> l(val_range, 0), r(val_range, 0);
    for (int c : vec) {
        if (c + 1 < val_range) ++l[c + 1];
        ++r[c];
    }
    std::partial_sum(l.begin(), l.end(), l.begin());
    std::partial_sum(r.begin(), r.end(), r.begin());
    std::fill(SA.begin(), SA.end(), -1);
    for (int i = (int)lms_idx.size() - 1; i >= 0; --i)
        SA[--r[vec[lms_idx[i]]]] = lms_idx[i];
    for (int i : SA)
        if (i >= 1 && sl[i - 1]) SA[l[vec[i - 1]]++] = i - 1;
    std::fill(r.begin(), r.end(), 0);
    for (int c : vec) ++r[c];
    std::partial_sum(r.begin(), r.end(), r.begin());
    for (int k = (int)SA.size() - 1, i = SA[k]; k >= 1; --k, i = SA[k])
        if (i >= 1 && !sl[i - 1]) {
            SA[--r[vec[i - 1]]] = i - 1;
        }
}

std::vector<int> SA_IS(const std::vector<int>& vec, int val_range) {
    const int n = vec.size();
    std::vector<int> SA(n), lms_idx;
    std::vector<bool> sl(n);
    sl[n - 1] = false;
    for (int i = n - 2; i >= 0; --i) {
        sl[i] = (vec[i] > vec[i + 1] || (vec[i] == vec[i + 1] && sl[i + 1]));
        if (sl[i] && !sl[i + 1]) lms_idx.push_back(i + 1);
    }
    std::reverse(lms_idx.begin(), lms_idx.end());
    induced_sort(vec, val_range, SA, sl, lms_idx);
    std::vector<int> new_lms_idx(lms_idx.size()), lms_vec(lms_idx.size());
    for (int i = 0, k = 0; i < n; ++i)
        if (!sl[SA[i]] && SA[i] >= 1 && sl[SA[i] - 1]) {
            new_lms_idx[k++] = SA[i];
        }
    int cur = 0;
    SA[n - 1] = cur;
    for (size_t k = 1; k < new_lms_idx.size(); ++k) {
        int i = new_lms_idx[k - 1], j = new_lms_idx[k];
        if (vec[i] != vec[j]) {
            SA[j] = ++cur;
            continue;
        }
        bool flag = false;
        for (int a = i + 1, b = j + 1;; ++a, ++b) {
            if (vec[a] != vec[b]) {
                flag = true;
                break;
            }
            if ((!sl[a] && sl[a - 1]) || (!sl[b] && sl[b - 1])) {
                flag = !((!sl[a] && sl[a - 1]) && (!sl[b] && sl[b - 1]));
                break;
            }
        }
```

```
        SA[j] = (flag ? ++cur : cur);
    }
    for (size_t i = 0; i < lms_idx.size(); ++i) lms_vec[i] = SA[lms_idx[i]];
    if (cur + 1 < (int)lms_idx.size()) {
        auto lms_SA = SA_IS(lms_vec, cur + 1);
        for (size_t i = 0; i < lms_idx.size(); ++i) {
            new_lms_idx[i] = lms_idx[lms_SA[i]];
        }
    }
    induced_sort(vec, val_range, SA, sl, new_lms_idx);
    return SA;
}
// }}}

template<typename ContainerT = std::string, typename ElemT = unsigned char>
std::vector<int> suffix_array(const ContainerT& s, const ElemT first = 'a',
                        const ElemT last = 'z') {
    std::vector<int> vec(s.size() + 1);
    std::copy(std::begin(s), std::end(s), std::begin(vec));
    for (auto& x : vec) x -= (int)first - 1;
    vec.back() = 0;
    auto ret = SA_IS(vec, (int)last - (int)first + 2);
    ret.erase(ret.begin());
    return ret;
}
// Uses kasai's algorithm linear in time and space
std::vector<int> LCP(const std::string& s, const std::vector<int>& sa) {
    int n = s.size(), k = 0;
    std::vector<int> lcp(n), rank(n);
    for (int i = 0; i < n; i++) rank[sa[i]] = i;
    for (int i = 0; i < n; i++, k ? k-- : 0) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = sa[rank[i] + 1];
        while (i + k < n && j + k < n && s[i + k] == s[j + k]) k++;
        lcp[rank[i]] = k;
    }
    lcp[n - 1] = 0;
    return lcp;
}
// }}}
// Number of distinct substrings {{{
int64_t cnt_distinct_substrings(const std::string& s) {
    auto lcp = LCP(s, suffix_array(s, 0, 255));
    return s.size() * (int64_t) (s.size() + 1) / 2
        - std::accumulate(lcp.begin(), lcp.end(), 0LL);
}
// }}}
// K-th distinct substring {{{
// Consider all distinct substring of string `s` in lexicographically increasing
// order. Find k-th substring.
//
// Preprocessing: O(N)
// Each query: O(log(N))
//
// Returns {start index, length}. If not found -> {-1, -1}
std::vector<std::pair<int,int>> kth_distinct_substring(
        const std::string& s,
        const std::vector<int64_t>& ks) {
    if (s.empty()) {
        return {};
    }
    auto sa = suffix_array(s, 0, 255);
    auto lcp = LCP(s, sa);
    int n = s.size();

    std::vector<int64_t> n_new_substrs(n);
    for (int i = 0; i < n; ++i) {
        int substr_len = n - sa[i];
        int new_substr_start = (i > 0 ? lcp[i-1] : 0);
        n_new_substrs[i] = substr_len - new_substr_start;
    }
    std::partial_sum(n_new_substrs.begin(), n_new_substrs.end(), n_new_substrs.begin());

    std::vector<std::pair<int,int>> res;
    for (int64_t k : ks) {
        if (k > *n_new_substrs.rbegin()) {
            res.emplace_back(-1, -1);
        } else {
            int i = std::lower_bound(n_new_substrs.begin(), n_new_substrs.end(), k) -
                n_new_substrs.begin();
            int new_substr_start = (i > 0 ? lcp[i-1] : 0);
            if (i > 0) k -= n_new_substrs[i-1];
            res.emplace_back(sa[i], new_substr_start + k);
        }
    }
    return res;
}
// }}}
// Count substring occurrences {{{
// given string S and Q queries pat_i, for each query, count how many
// times pat_i appears in S
// O(min(|S|, |pat|) * log(|S|)) per query
int cnt_occurrences(const string& s, const vector<int>& sa, const string& pat) {
    int n = s.size(), m = pat.size();
    assert(n == (int) sa.size());
    if (n < m) return 0;

    auto f = [&] (int start) { // compare S[start..] and pat[0..]
        for (int i = 0; start + i < n && i < m; ++i) {
            if (s[start + i] < pat[i]) return true;
            if (s[start + i] > pat[i]) return false;
        }
        return n - start < m;
```

```
    };
    auto g = [&] (int start) {
        for (int i = 0; start + i < n && i < m; ++i) {
            if (s[start + i] > pat[i]) return false;
        }
        return true;
    };
    auto l = std::partition_point(sa.begin(), sa.end(), f);
    auto r = std::partition_point(l, sa.end(), g);

    return std::distance(l, r);
}
// Count substring occurrences using hash {{{
// If hash array can be pre-computed, can answer each query in
// O(log(|S|) * log(|S| + |pat|)
#include "./hash.h"
int cnt_occurrences_hash(
        const vector<int>& sa,     // suffix array
        const HashGenerator& gen,
        const string& s,
        const vector<Hash>& hash_s, // hash of `s`, generated with `gen`
        const string_view& pat,
        const vector<Hash>& hash_pat // hash of `pat`, generated with `gen`
        ) {
    int n = s.size(), len = pat.size();
    assert(len == (int) hash_pat.size());
    assert(n == (int) sa.size());
    if (n < len) return 0;

    // f(start) = compare string S[start..] and pat[0..len-1]
    auto f = [&] (int start) {
        return gen.cmp(
                s, hash_s, start, n-1,
                pat, hash_pat, 0, len-1) < 0;
    };
    // g(start) = true if S[start..] == pat[0..]
    auto g = [&] (int start) {
        int max_len = std::min(n - start, len);
        return gen.cmp(
                s, hash_s, start, start + max_len - 1,
                pat, hash_pat, 0, max_len-1) == 0;
    };
    auto l = std::partition_point(sa.begin(), sa.end(), f);
    auto r = std::partition_point(l, sa.end(), g);
    return std::distance(l, r);
}
// Returns length of LCS of strings s & t {{{
// O(N)
int longestCommonSubstring(const string& s, const string& t) {
    char c = 127;
    string combined = s + c + t;
    auto sa = suffix_array(combined, 0, 127);
    auto lcp = LCP(combined, sa);
    // s -> 0 .. |s|-1
    // 255 -> |s|
    // t -> |s|+1 ..
    int ls = s.size(), lcombined = combined.size();
    auto is_s = [&] (int id) { return sa[id] < ls; };
    auto is_t = [&] (int id) { return sa[id] > ls; };

    assert(sa[lcombined - 1] == ls);

    int res = 0;
    for (int i = 0; i < lcombined - 2; ++i) {
        if ((is_s(i) && is_t(i+1)) || (is_s(i+1) && is_t(i))) {
            res = max(res, lcp[i]);
        }
    }
    return res;
}

// Returns length of LCS of n strings {{{
#include "../DataStructure/RMQ.h"
int longestCommonSubstring(const std::vector<std::string> strs) {
    char c = 127;
    string combined = "";
    vector<int> ids;
    for (size_t i = 0; i < strs.size(); ++i) {
        const auto& s = strs[i];
        combined += s;
        while (ids.size() < combined.size()) ids.push_back(i);

        combined += c;
        ids.push_back(-1);

        --c;
    }
    auto sa = suffix_array(combined, 0, 127);
    auto lcp = LCP(combined, sa);
    RMQ<int, _min> rmq(lcp);

    // count frequency of i-th string in current window
    std::vector<int> cnt(strs.size(), 0);
    int strs_in_window = 0;
    auto add = [&] (int i) {
        if (i < 0) return;
        ++cnt[i];
        if (cnt[i] == 1) ++strs_in_window;
    };
    auto rem = [&] (int i) {
        if (i < 0) return;
        --cnt[i];
        if (cnt[i] == 0) --strs_in_window;
    };
```

```
    int i = 0, j = -1;
    int lcombined = combined.size();
    int n = strs.size();
    int res = 0;
    while (i < lcombined - 1) {
        while (j + 1 < lcombined - 1 && strs_in_window < n) {
            ++j;
            add(ids[sa[j]]);
        }
        if (strs_in_window == n) {
            res = max(res, rmq.get(i, j));
        }

        rem(ids[sa[i]]); ++i;
    }
    return res;
}
```

19 TwoSAT

```
inline int pos(int u) { return u << 1; }
inline int neg(int u) { return u << 1 | 1; }
// ZERO-indexed
// color[i] = 1 means we choose i
struct TwoSAT {
    int n;
    int numComp;
    vector<int> adj[V];
    int low[V], num[V], root[V], cntTarjan;
    vector<int> stTarjan;
    int color[V];
    TwoSAT(int n) : n(n * 2) {
        memset(root, -1, sizeof root);
        memset(low, -1, sizeof low);
        memset(num, -1, sizeof num);
        memset(color, -1, sizeof color);
        cntTarjan = 0;
        stTarjan.clear();
    }
    // u | v
    void addEdge(int u, int v) {
        adj[u   1].push_back(v);
        adj[v   1].push_back(u);
    }
    void tarjan(int u) {
        stTarjan.push_back(u);
        num[u] = low[u] = cntTarjan++;
        for (int v : adj[u]) {
            if (root[v] != -1) continue;
            if (low[v] == -1) tarjan(v);
            low[u] = min(low[u], low[v]);
        }
        if (low[u] == num[u]) {
            while (1) {
                int v = stTarjan.back();
                stTarjan.pop_back();
                root[v] = numComp;
                if (u == v) break;
            }
            numComp++;
        }
    }
    bool solve() {
        for (int i = 0; i < n; i++) if (root[i] == -1) tarjan(i);
        for (int i = 0; i < n; i += 2) {
            if (root[i] == root[i  1]) return 0;
            color[i >> 1] = (root[i] < root[i  1]);
        }
        return 1;
    }
};
```

20 bigint

```
const int BASE_DIGITS = 9;
const int BASE = 1000000000;

struct BigInt {
    int sign;
    vector<int> a;

    BigInt() : sign(1) {}
    BigInt(long long v) {*this = v;}

    BigInt& operator = (long long v) {
        sign = 1;
        if (v < 0) {
            sign = -1;
            v = -v;
        }
        a.clear();
        for (; v > 0; v = v / BASE)
            a.push_back(v % BASE);
        return *this;
    }

    // Initialize from string.
    BigInt(const string& s) {
        read(s);
    }
```

```
// -------------------- Input / Output --------------------
void read(const string& s) {
    sign = 1;
    a.clear();
    int pos = 0;
    while (pos < (int) s.size() && (s[pos] == '-' || s[pos] == '+')) {
        if (s[pos] == '-')
            sign = -sign;
        ++pos;
    }
    for (int i = s.size() - 1; i >= pos; i -= BASE_DIGITS) {
        int x = 0;
        for (int j = max(pos, i - BASE_DIGITS + 1); j <= i; j++)
            x = x * 10 + s[j] - '0';
        a.push_back(x);
    }
    trim();
}
friend istream& operator>>(istream &stream, BigInt &v) {
    string s;
    stream >> s;
    v.read(s);
    return stream;
}

friend ostream& operator<<(ostream &stream, const BigInt &v) {
    if (v.sign == -1 && !v.isZero())
        stream << '-';
    stream << (v.a.empty() ? 0 : v.a.back());
    for (int i = (int) v.a.size() - 2; i >= 0; --i)
        stream << setw(BASE_DIGITS) << setfill('0') << v.a[i];
    return stream;
}

// -------------------- Comparison --------------------
bool operator<(const BigInt &v) const {
    if (sign != v.sign)
        return sign < v.sign;
    if (a.size() != v.a.size())
        return a.size() * sign < v.a.size() * v.sign;
    for (int i = ((int) a.size()) - 1; i >= 0; i--)
        if (a[i] != v.a[i])
            return a[i] * sign < v.a[i] * sign;
    return false;
}

bool operator>(const BigInt &v) const {
    return v < *this;
}
bool operator<=(const BigInt &v) const {
    return !(v < *this);
}
bool operator>=(const BigInt &v) const {
    return !(*this < v);
}
bool operator==(const BigInt &v) const {
    return !(*this < v) && !(v < *this);
}
bool operator!=(const BigInt &v) const {
    return *this < v || v < *this;
}

// Returns:
// 0 if |x| == |y|
// -1 if |x| < |y|
// 1 if |x| > |y|
friend int __compare_abs(const BigInt& x, const BigInt& y) {
    if (x.a.size() != y.a.size()) {
        return x.a.size() < y.a.size() ? -1 : 1;
    }

    for (int i = ((int) x.a.size()) - 1; i >= 0; --i) {
        if (x.a[i] != y.a[i]) {
            return x.a[i] < y.a[i] ? -1 : 1;
        }
    }
    return 0;
}

// -------------------- Unary operator - and operators +- --------------------
BigInt operator-() const {
    BigInt res = *this;
    if (isZero()) return res;

    res.sign = -sign;
    return res;
}

// Note: sign ignored.
void __internal_add(const BigInt& v) {
    if (a.size() < v.a.size()) {
        a.resize(v.a.size(), 0);
    }
    for (int i = 0, carry = 0; i < (int) max(a.size(), v.a.size()) || carry; ++i) {
        if (i == (int) a.size()) a.push_back(0);

        a[i] += carry + (i < (int) v.a.size() ? v.a[i] : 0);
        carry = a[i] >= BASE;
        if (carry) a[i] -= BASE;
    }
}

// Note: sign ignored.
void __internal_sub(const BigInt& v) {
    for (int i = 0, carry = 0; i < (int) v.a.size() || carry; ++i) {
```

```
        a[i] -= carry + (i < (int) v.a.size() ? v.a[i] : 0);
        carry = a[i] < 0;
        if (carry) a[i] += BASE;
    }
    this->trim();
}

BigInt operator += (const BigInt& v) {
    if (sign == v.sign) {
        __internal_add(v);
    } else {
        if (__compare_abs(*this, v) >= 0) {
            __internal_sub(v);
        } else {
            BigInt vv = v;
            swap(*this, vv);
            __internal_sub(vv);
        }
    }
    return *this;
}

BigInt operator -= (const BigInt& v) {
    if (sign == v.sign) {
        if (__compare_abs(*this, v) >= 0) {
            __internal_sub(v);
        } else {
            BigInt vv = v;
            swap(*this, vv);
            __internal_sub(vv);
            this->sign = -this->sign;
        }
    } else {
        __internal_add(v);
    }
    return *this;
}

// Optimize operators + and - according to
//
     https://stackoverflow.com/questions/13166079/move-semantics-and-pass-by-rvalue-reference-in-overloaded-arithmetic
template< typename L, typename R >
    typename std::enable_if<
        std::is_convertible<L, BigInt>::value &&
        std::is_convertible<R, BigInt>::value &&
        std::is_lvalue_reference<R&&>::value,
        BigInt>::type friend operator + (L&& l, R&& r) {
    BigInt result(std::forward<L>(l));
    result += r;
    return result;
}
template< typename L, typename R >
    typename std::enable_if<
        std::is_convertible<L, BigInt>::value &&
        std::is_convertible<R, BigInt>::value &&
        std::is_rvalue_reference<R&&>::value,
        BigInt>::type friend operator + (L&& l, R&& r) {
    BigInt result(std::move(r));
    result += l;
    return result;
}

template< typename L, typename R >
    typename std::enable_if<
        std::is_convertible<L, BigInt>::value &&
        std::is_convertible<R, BigInt>::value,
        BigInt>::type friend operator - (L&& l, R&& r) {
    BigInt result(std::forward<L>(l));
    result -= r;
    return result;
}

// -------------------- Operators * / % --------------------
friend pair<BigInt, BigInt> divmod(const BigInt& a1, const BigInt& b1) {
    assert(b1 > 0); // divmod not well-defined for b < 0.

    long long norm = BASE / (b1.a.back() + 1);
    BigInt a = a1.abs() * norm;
    BigInt b = b1.abs() * norm;
    BigInt q = 0, r = 0;
    q.a.resize(a.a.size());

    for (int i = a.a.size() - 1; i >= 0; i--) {
        r *= BASE;
        r += a.a[i];
        long long s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
        long long s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];
        long long d = ((long long) BASE * s1 + s2) / b.a.back();
        r -= b * d;
        while (r < 0) {
            r += b, --d;
        }
        q.a[i] = d;
    }

    q.sign = a1.sign * b1.sign;
    r.sign = a1.sign;
    q.trim();
    r.trim();
    auto res = make_pair(q, r / norm);
    if (res.second < 0) res.second += b1;
    return res;
}
BigInt operator/(const BigInt &v) const {
    if (v < 0) return divmod(-*this, -v).first;
    return divmod(*this, v).first;
```

```
}

BigInt operator%(const BigInt &v) const {
    return divmod(*this, v).second;
}

void operator/=(int v) {
    assert(v > 0); // operator / not well-defined for v <= 0.
    if (llabs(v) >= BASE) {
        *this /= BigInt(v);
        return ;
    }
    if (v < 0)
        sign = -sign, v = -v;
    for (int i = (int) a.size() - 1, rem = 0; i >= 0; --i) {
        long long cur = a[i] + rem * (long long) BASE;
        a[i] = (int) (cur / v);
        rem = (int) (cur % v);
    }
    trim();
}

BigInt operator/(int v) const {
    assert(v > 0); // operator / not well-defined for v <= 0.

    if (llabs(v) >= BASE) {
        return *this / BigInt(v);
    }
    BigInt res = *this;
    res /= v;
    return res;
}
void operator/=(const BigInt &v) {
    *this = *this / v;
}

long long operator%(long long v) const {
    assert(v > 0); // operator / not well-defined for v <= 0.
    assert(v < BASE);
    int m = 0;
    for (int i = a.size() - 1; i >= 0; --i)
        m = (a[i] + m * (long long) BASE) % v;
    return m * sign;
}

void operator*=(int v) {
    if (llabs(v) >= BASE) {
        *this *= BigInt(v);
        return ;
    }
    if (v < 0)
        sign = -sign, v = -v;
    for (int i = 0, carry = 0; i < (int) a.size() || carry; ++i) {
        if (i == (int) a.size())
            a.push_back(0);
        long long cur = a[i] * (long long) v + carry;
        carry = (int) (cur / BASE);
        a[i] = (int) (cur % BASE);
    }
    trim();
}

BigInt operator*(int v) const {
    if (llabs(v) >= BASE) {
        return *this * BigInt(v);
    }
    BigInt res = *this;
    res *= v;
    return res;
}

// Convert BASE 10^old --> 10^new.
static vector<int> convert_base(const vector<int> &a, int old_digits, int
     new_digits) {
    vector<long long> p(max(old_digits, new_digits) + 1);
    p[0] = 1;
    for (int i = 1; i < (int) p.size(); i++)
        p[i] = p[i - 1] * 10;
    vector<int> res;
    long long cur = 0;
    int cur_digits = 0;
    for (int i = 0; i < (int) a.size(); i++) {
        cur += a[i] * p[cur_digits];
        cur_digits += old_digits;
        while (cur_digits >= new_digits) {
            res.push_back((long long)(cur % p[new_digits]));
            cur /= p[new_digits];
            cur_digits -= new_digits;
        }
    }
    res.push_back((int) cur);
    while (!res.empty() && !res.back())
        res.pop_back();
    return res;
}

void fft(vector<complex<double> > &x, bool invert) const {
    int n = (int) x.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(x[i], x[j]);
```

```
    }

    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * 3.14159265358979323846 / len * (invert ? -1 : 1);
        complex<double> wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            complex<double> w(1);
            for (int j = 0; j < len / 2; ++j) {
                complex<double> u = x[i + j];
                complex<double> v = x[i + j + len / 2] * w;
                x[i + j] = u + v;
                x[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }
    if (invert)
        for (int i = 0; i < n; ++i)
            x[i] /= n;
}

void multiply_fft(const vector<int> &x, const vector<int> &y, vector<int> &res)
     const {
    vector<complex<double> > fa(x.begin(), x.end());
    vector<complex<double> > fb(y.begin(), y.end());
    int n = 1;
    while (n < (int) max(x.size(), y.size()))
        n <<= 1;
    n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; ++i)
        fa[i] *= fb[i];
    fft(fa, true);

    res.resize(n);
    long long carry = 0;
    for (int i = 0; i < n; ++i) {
        long long t = (long long) (fa[i].real() + 0.5) + carry;
        carry = t / 1000;
        res[i] = t % 1000;
    }
}

BigInt mul_simple(const BigInt &v) const {
    BigInt res;
    res.sign = sign * v.sign;
    res.a.resize(a.size() + v.a.size());
    for (int i = 0; i < (int) a.size(); ++i)
        if (a[i])
            for (int j = 0, carry = 0; j < (int) v.a.size() || carry; ++j) {
                long long cur = res.a[i + j] + (long long) a[i] * (j < (int)
                     v.a.size() ? v.a[j] : 0) + carry;
                carry = (int) (cur / BASE);
                res.a[i + j] = (int) (cur % BASE);
            }
    res.trim();
    return res;
}

typedef vector<long long> vll;

static vll karatsubaMultiply(const vll &a, const vll &b) {
    int n = a.size();
    vll res(n + n);
    if (n <= 32) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                res[i + j] += a[i] * b[j];
        return res;
    }

    int k = n >> 1;
    vll a1(a.begin(), a.begin() + k);
    vll a2(a.begin() + k, a.end());
    vll b1(b.begin(), b.begin() + k);
    vll b2(b.begin() + k, b.end());

    vll a1b1 = karatsubaMultiply(a1, b1);
    vll a2b2 = karatsubaMultiply(a2, b2);

    for (int i = 0; i < k; i++)
        a2[i] += a1[i];
    for (int i = 0; i < k; i++)
        b2[i] += b1[i];

    vll r = karatsubaMultiply(a2, b2);
    for (int i = 0; i < (int) a1b1.size(); i++)
        r[i] -= a1b1[i];
    for (int i = 0; i < (int) a2b2.size(); i++)
        r[i] -= a2b2[i];

    for (int i = 0; i < (int) r.size(); i++)
        res[i + k] += r[i];
    for (int i = 0; i < (int) a1b1.size(); i++)
        res[i] += a1b1[i];
    for (int i = 0; i < (int) a2b2.size(); i++)
        res[i + n] += a2b2[i];
    return res;
}

BigInt mul_karatsuba(const BigInt &v) const {
    vector<int> x6 = convert_base(this->a, BASE_DIGITS, 6);
```

```
    vector<int> y6 = convert_base(v.a, BASE_DIGITS, 6);
    vll x(x6.begin(), x6.end());
    vll y(y6.begin(), y6.end());
    while (x.size() < y.size())
        x.push_back(0);
    while (y.size() < x.size())
        y.push_back(0);
    while (x.size() & (x.size() - 1))
        x.push_back(0), y.push_back(0);
    vll c = karatsubaMultiply(x, y);
    BigInt res;
    res.sign = sign * v.sign;
    long long carry = 0;
    for (int i = 0; i < (int) c.size(); i++) {
        long long cur = c[i] + carry;
        res.a.push_back((int) (cur % 1000000));
        carry = cur / 1000000;
    }
    res.a = convert_base(res.a, 6, BASE_DIGITS);
    res.trim();
    return res;
}

void operator*=(const BigInt &v) {
    *this = *this * v;
}
BigInt operator*(const BigInt &v) const {
    if (a.size() * v.a.size() <= 1000111) return mul_simple(v);
    if (a.size() > 500111 || v.a.size() > 500111) return mul_fft(v);
    return mul_karatsuba(v);
}

BigInt mul_fft(const BigInt& v) const {
    BigInt res;
    res.sign = sign * v.sign;
    multiply_fft(convert_base(a, BASE_DIGITS, 3), convert_base(v.a, BASE_DIGITS, 3),
         res.a);
    res.a = convert_base(res.a, 3, BASE_DIGITS);
    res.trim();
    return res;
}

// -------------------- Misc --------------------
BigInt abs() const {
    BigInt res = *this;
    res.sign *= res.sign;
    return res;
}
void trim() {
    while (!a.empty() && !a.back())
        a.pop_back();
    if (a.empty())
        sign = 1;
}

bool isZero() const {
    return a.empty() || (a.size() == 1 && !a[0]);
}

friend BigInt gcd(const BigInt &x, const BigInt &y) {
    return y.isZero() ? x : gcd(y, x % y);
}
friend BigInt lcm(const BigInt &x, const BigInt &y) {
    return x / gcd(x, y) * y;
}

friend BigInt sqrt(const BigInt &a1) {
    BigInt a = a1;
    while (a.a.empty() || a.a.size() % 2 == 1)
        a.a.push_back(0);

    int n = a.a.size();

    int firstDigit = (int) sqrt((double) a.a[n - 1] * BASE + a.a[n - 2]);
    int norm = BASE / (firstDigit + 1);
    a *= norm;
    a *= norm;
    while (a.a.empty() || a.a.size() % 2 == 1)
        a.a.push_back(0);

    BigInt r = (long long) a.a[n - 1] * BASE + a.a[n - 2];
    firstDigit = (int) sqrt((double) a.a[n - 1] * BASE + a.a[n - 2]);
    int q = firstDigit;
    BigInt res;

    for(int j = n / 2 - 1; j >= 0; j--) {
        for(; ; --q) {
            BigInt r1 = (r - (res * 2 * BigInt(BASE) + q) * q) * BigInt(BASE) *
                 BigInt(BASE) + (j > 0 ? (long long) a.a[2 * j - 1] * BASE + a.a[2 *
                 j - 2] : 0);
            if (r1 >= 0) {
                r = r1;
                break;
            }
        }
        res *= BASE;
        res += q;

        if (j > 0) {
            int d1 = res.a.size() + 2 < r.a.size() ? r.a[res.a.size() + 2] : 0;
            int d2 = res.a.size() + 1 < r.a.size() ? r.a[res.a.size() + 1] : 0;
            int d3 = res.a.size() < r.a.size() ? r.a[res.a.size()] : 0;
            q = ((long long) d1 * BASE * BASE + (long long) d2 * BASE + d3) /
                 (firstDigit * 2);
        }
    }
```

```
        res.trim();
        return res / norm;
    }
};
```

21 kmp

```
// prefix function: *length* of longest prefix which is also suffix:
//   pi[i] = max(k: s[0..k-1] == s[i-(k-1)..i]
//
// KMP {{{
template<typename Container>
std::vector<int> prefix_function(const Container& s) {
    int n = s.size();
    std::vector<int> pi(n);
    for (int i = 1; i < n; ++i) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j]) j = pi[j-1];
        if (s[i] == s[j]) ++j;
        pi[i] = j;
    }
    return pi;
}

// Return all positions (0-based) that pattern `pat` appears in `text`
std::vector<int> kmp(const std::string& pat, const std::string& text) {
    auto pi = prefix_function(pat + '\0' + text);
    std::vector<int> res;
    for (size_t i = pi.size() - text.size(); i < pi.size(); ++i) {
        if (pi[i] == (int) pat.size()) {
            res.push_back(i - 2 * pat.size());
        }
    }
    return res;
}

// Returns cnt[i] = # occurrences of prefix of length-i
// NOTE: cnt[0] = n+1 (0-length prefix appears n+1 times)
std::vector<int> prefix_occurrences(const string& s) {
    int n = s.size();
    auto pi = prefix_function(s);
    std::vector<int> res(n + 1);
    for (int i = 0; i < n; ++i) res[pi[i]]++;
    for (int i = n-1; i > 0; --i) res[pi[i-1]] += res[i];
    for (int i = 0; i <= n; ++i) res[i]++;
    return res;
}
// }}}
```

22 zfunc

```
// z[i] = length of longest common prefix of s[0..N] and s[i..N]
vector<int> zfunc(const string& s) {
    int n = (int) s.length();
    vector<int> z(n);
    z[0] = n;
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min(r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```