**Contents**

1. **Data structures**
   1. Segment Tree ……………………………………………………………………………………………………….. 2
   2. Merge Sort Tree…………………………………………………………………………………………………….. 2
   3. Persistent Segment Tree………………………………………………………………………………… …….. 2
   4. Lichao Tree………………………………………………………………….………………………………………….. 3
2. **Graph**

2.1 Bellman-Ford ………………………………………………………………………………………………………….. 4

2.2 Lowest Common Ancestor …………………………………………………………………………………….. 4

2.3 Strongly Connected Component …………………………………………………….…………………….. 4

2.4 2-Satisfiability ……………………………………………………………..………………………………………….. 5

2.5 Heavy Light Decomposition ………………………………………………………………………………….. 5

2.6 Dominator Tree ……………………………..……………………………………………………………………….. 6

1. **Flow**

3.1 Bitpartite Matching ……………………………………………………………………………………………….. 6

3.2 Hopcroft-Karp………………………………………………………………………………………………………….. 7

3.3 MCMF…………………………………………………………………………………………………..………………….. 7

3.4 Dinic…………………………………………………………………………………..…………………………………….. 8

3.5 Circulation……………………………………………………………………………………………………………….. 9

1. **Strings**

4.1 KMP………………………………………………………………………………………………………………………….. 9

4.2 Trie……………………………………………………………………….………………………………………………….. 9

4.3 Rabin-Karp Fingerprint ………………………………………………………………………….……….…….. 10

4.4 Manacher ………………………………………………………………………………………………….………….. 10

4.5 Aho-Corasick ……………………………………………………………………………………………..……….. 10

4.6 Suffix Array and LCP ………………………………………………………………………….……………….. 11

1. **Geometry**

5.1 Line-Segment Intersection ………………………………………….……………………,…….………….. 11

5.2 Convex Hull …………………………………………………………………………………,………….………….. 11

5.3 Smallest Enclosing Circle ………………………………….…………………………………,….………….. 12

5.4 Point In Convex Polygon Check …………………….………………………………………,.………….. 12

5.5 Point In Non-Convex Polygon Check …….…………………………………….,,……,…….……….. 12

5.6 Rotating Calipers …………………………….…………………………………………….….…,.……………... 13

5.7 Half Plane Intersection ……………………………………………………………...………...,…….……….. 13

1. **Math**

6.1 FFT, XOR-FFT …………………..…………………………………………………………………………………….. 14

6.2 Extended Euclidean ……………………………………………...………………………..…………………….. 15

6.3 Z2 Matrix ……………………………………………………………………………………………..……………….. 15

6.4 Miller-Rabin ………………………………..………………………………………………………..……………….. 15

6.5 Euler phi Function ……………………………………………………………………………………………….. 16

6.6 Mobius Function ………………………………………………………………………………………………….. 16

6.7 Modular Integer …………………………………………………………………………..……………………….. 16

1. **Miscellaneous**

7.1 CHT, DNC dp optimization ………………………………………………………………………………….. 17

7.2 Lucas Theorem …………………………………………………………………………………………………….. 17

7.3 Burnside’s Lemma …………………………………………….………………………………………………….. 17

7.4 Hall’s Theorem ………………………………….………………………………………………………………….. 17

1. **Data Structures**

**1.1 Segment Tree**

template<typename T, T (\*merge)(T, T), T e>

struct Seg {

int n;

vector<T> seg;

void init(int \_n) {

n = \_n;

seg.resize(2 \* n);

}

void upt(int i, T p) {

for (seg[i += n] += p; i >>= 1; )

seg[i] = merge(seg[i << 1], seg[i << 1 | 1]);

}

T get(int i) { return seg[i + n]; }

T query(int l, int r) {

T nl = e, nr = e;

for (l += n, r += n+1; l < r; l >>= 1, r >>= 1) {

if (l & 1) nl = merge(nl, seg[l++]);

if (r & 1) nr = merge(seg[--r], nr);

}

return merge(nl, nr);

}

};

**1.2 Merge Sort Tree**

struct MergeSortTree {

int n;

vector<int> arr;

vector<vector<int>> seg;

void Init(vector<int>& vec) {

n = vec.size();

arr.assign(vec.begin(), vec.end());

seg.resize(4 \* n);

SetTree(1, 0, n - 1);

}

void SetTree(int num, int s, int e) {

if (s == e) {

seg[num].push\_back(arr[s]);

return;

}

int mid = s + e >> 1;

SetTree(2 \* num, s, mid);

SetTree(2 \* num + 1, mid + 1, e);

vector<int>& now = seg[num];

vector<int>& l = seg[2 \* num], r = seg[2 \* num + 1];

int pl = 0, pr = 0;

for (int i = 0; i < l.size() + r.size(); i++) {

if (pl < l.size() && pr < r.size()) {

if (l[pl] < r[pr]) now.push\_back(l[pl]), pl++;

else now.push\_back(r[pr]), pr++;

}

else if (pl < l.size())

now.push\_back(l[pl]), pl++;

else if (pr < r.size())

now.push\_back(r[pr]), pr++;

}

}

int query(int num, int s, int e, int l, int r, int k) {

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

if (l <= s && e <= r) {

int idx = upper\_bound(seg[num].begin(), seg[num].end(), k) - seg[num].begin();

return seg[num].size() - idx;

}

int mid = s + e >> 1;

return query(2 \* num, s, mid, l, r, k) + query(2 \* num + 1, mid + 1, e, l, r, k);

}

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

}tree;

**1.3 Persistent Segment Tree**

typedef long long ll;

const int MAX = 300001;

//[l..r] find k-th minimum number in O(logN)

struct PST {

struct Node {

ll cnt;

Node\* l, \* r;

};

int n;

Node\* root[MAX];

void func(int \_n) {

n = \_n;

root[0] = new Node();

init(root[0], 0, n - 1);

}

void init(Node\* node, int s, int e) {

if (s == e) return;

int mid = s + e >> 1;

node->l = new Node();

node->r = new Node();

init(node->l, s, mid); init(node->r, mid + 1, e);

}

void update(Node\* prv, Node\* now, int s, int e, int idx, ll a) {

if (s == e) {

now->cnt += a;

return;

}

int mid = s + e >> 1;

if (idx <= mid) {

now->l = new Node(); now->r = prv->r;

now->l->cnt += prv->l->cnt;

update(prv->l, now->l, s, mid, idx, a);

}

else {

now->l = prv->l; now->r = new Node();

now->r->cnt += prv->r->cnt;

update(prv->r, now->r, mid + 1, e, idx, a);

}

now->cnt = now->l->cnt + now->r->cnt;

}

void update(int num, int idx, ll a) {

root[num] = new Node();

update(root[num - 1], root[num], 0, n - 1, idx, a);

}

ll query(Node\* nl, Node\* nr, int s, int e, int k) {

if (s == e) return s;

int mid = s + e >> 1;

int cnt = nr->l->cnt - nl->l->cnt;

if (k <= cnt)

return query(nl->l, nr->l, s, mid, k);

else

return query(nl->r, nr->r, mid + 1, e, k - cnt);

}

ll query(int il, int ir, int k) { return query(root[il - 1], root[ir], 0, n - 1, k); }

};

**1.4 Lichao Tree**

//Minimum

typedef long long ll;

const ll inf = 1e18;

struct Line {

ll a, b;

ll f(ll x) { return a \* x + b; }

};

struct Lichao {

struct Node {

ll l, r;

Line line;

};

ll n, psum, ns, ne;

vector<Node> seg;

vector<Line> lines;

void init(int s, int e) {

ns = s, ne = e;

seg.push\_back({ -1, -1, {0, inf} });

}

int size() { return lines.size(); }

void insert(int num, int s, int e, Line l) {

Line lo = seg[num].line, hi = l;

if (lo.f(s) > hi.f(s)) swap(lo, hi);

if (lo.f(e) <= hi.f(e)) {

seg[num].line = lo;

return;

}

int mid = s + e >> 1;

if (lo.f(mid) < hi.f(mid)) {

seg[num].line = lo;

if (seg[num].r == -1) {

seg[num].r = seg.size();

seg.push\_back({ -1, -1, {0, inf} });

}

insert(seg[num].r, mid + 1, e, hi);

}

else {

seg[num].line = hi;

if (seg[num].l == -1) {

seg[num].l = seg.size();

seg.push\_back({ -1, -1, {0, inf} });

}

insert(seg[num].l, s, mid, lo);

}

}

void insert(Line l) {

l.b -= psum;

lines.push\_back(l);

insert(0, ns, ne, l);

}

void apply() {

for (auto& l : lines) l.b += psum;

for (auto& l : seg) l.line.b += psum;

psum = 0;

}

ll query(int num, int s, int e, ll x) {

if (num == -1) return inf;

int mid = s + e >> 1;

ll d = seg[num].line.f(x) + psum;

if (x <= mid) return min(d, query(seg[num].l, s, mid, x));

else return min(d, query(seg[num].r, mid + 1, e, x));

}

ll query(ll x) { return query(0, ns, ne, x); }

};

1. **Graph**

**2.1 Bellman-Ford**

const ll inf = 1e18;

struct Line { ll u, v, c; };

vector<ll> bellman(int n, vector<Line>& e) {

vector<ll> dst(n, inf);

dst[1] = 0;

for (int i = 0; i < n; i++) for (auto& l : e) {

if (dst[l.u] != inf && dst[l.v] > dst[l.u] + l.c) {

dst[l.v] = dst[l.u] + l.c;

if (i == n - 1) return vector<ll>(n, -1);

}

}

return dst;

}

**2.2 Lowest Common Ancestor**

const int MAX = 30001;

const int LV = 17;

int dep[MAX], dp[LV + 1][MAX];

vector<int> V[MAX];

void dfs(int pos, int d = 0, int p = 0) {

dep[pos] = d;

dp[0][pos] = p;

for (int i = 1; i <= LV; i++)

dp[i][pos] = dp[i - 1][dp[i - 1][pos]];

for (int w : V[pos]) {

if (w == p) continue;

dfs(w, d + 1, pos);

}

}

int lca(int a, int b) {

if (dep[a] < dep[b]) swap(a, b);

int d = dep[a] - dep[b];

for (int i = 0; d; i++, d >>= 1)

if (d & 1) a = dp[i][a];

if (a == b) return a;

for (int i = LV; ~i; i--)

if (dp[i][a] != dp[i][b]) a = dp[i][a], b = dp[i][b];

return dp[0][a];

}

**2.3 Strongly Connected Component**

struct SCC {

vector<int> visited, scc\_id;

int scc\_cnt, n;

vector<vector<int>> adj;

vector<vector<int>> scc;

stack<int> st;

void init(int \_n) {

n = \_n;

scc\_cnt = 0;

adj.clear(); adj.resize(n);

visited.clear(); visited.resize(n, -1);

scc\_id.clear(); scc\_id.resize(n, -1);

scc.clear();

}

int dfs(int cur) {

int ret = visited[cur] = scc\_cnt++;

st.push(cur);

for (auto nxt : adj[cur]) {

if (visited[nxt] == -1) ret = min(ret, dfs(nxt));

else if (scc\_id[nxt] == -1) ret = min(ret, visited[nxt]);

}

if (ret == visited[cur]) {

vector<int> v;

while (true) {

int t = st.top(); st.pop();

scc\_id[t] = scc.size() + 1;

v.push\_back(t);

if (t == cur) break;

}

scc.push\_back(v);

scc\_cnt++;

}

return ret;

}

void get\_scc() {

for (int i = 1; i <= n; i++) {

if (visited[i] == -1) dfs(i);

}

}

};

**2.4 2-Satisfiability**

struct TwoSat{

int n;

SCC scc;

void init(int \_n) {

n = \_n;

scc.init(2\*n);

}

int inv(int i) { return i + n; }

void add\_clause(int a, int b, bool arev = false, bool brev = false) {

int u1 = (arev) ? inv(a) : a, v1 = (brev) ? b : inv(b);

int u2 = (brev) ? inv(b) : b, v2 = (arev) ? a : inv(a);

scc.adj[u1].push\_back(v1);

scc.adj[u2].push\_back(v2);

}

bool correct() {

for (int i = 0; i < n; i++)

if (scc.scc\_id[i] == scc.scc\_id[inv(i)]) return false;

return true;

}

}

**2.5 Heavy Light Decomposition**

const int MAX = 300001;

vector<int> V[MAX], g[MAX];

int sz[MAX], dep[MAX], top[MAX], par[MAX], in[MAX], out[MAX];

bitset<MAX> vit;

void dfs(int pos) {

vit[pos] = true;

for (int& w : V[pos]) {

if (vit[w]) continue;

g[pos].push\_back(w);

dfs(w);

}

}

void dfs1(int pos) {

sz[pos] = 1;

for (int& w : g[pos]) {

dep[w] = dep[pos] + 1, par[w] = pos;

dfs1(w);

sz[pos] += sz[w];

if (sz[w] > sz[g[pos][0]]) swap(w, g[pos][0]);

}

}

int pv;

void dfs2(int pos) {

in[pos] = ++pv;

for (int& w : g[pos]) {

top[w] = w == g[pos][0] ? top[pos] : w;

dfs2(w);

}

out[pos] = pv;

}

void update(int a, int b, int diff) {

int ans = 0;

while (top[a] != top[b]) {

if (dep[top[a]] < dep[top[b]]) swap(a, b);

int x = top[a];

tree.update(in[x], in[a], diff);

a = par[x];

}

if (dep[a] > dep[b]) swap(a, b);

tree.update(in[a], in[b], diff);

}

int query(int a, int b) {

int ans = 0;

while (top[a] != top[b]) {

if (dep[top[a]] < dep[top[b]]) swap(a, b);

int x = top[a];

ans += tree.query(in[x], in[a]);

a = par[x];

}

if (dep[a] > dep[b]) swap(a, b);

ans += tree.query(in[a], in[b]);

return ans;

}

**2.6 Dominator Tree**

namespace dtree { // by cki86201

const int MAXN = 300001;

vector<int> E[MAXN], RE[MAXN], rdom[MAXN];

int S[MAXN], RS[MAXN], cs;

int par[MAXN], val[MAXN], sdom[MAXN], rp[MAXN], dom[MAXN];

void clear(int n) {

cs = 0;

for (int i = 0; i <= n; i++) {

par[i] = val[i] = sdom[i] = rp[i] = dom[i] = S[i] = RS[i] = 0;

E[i].clear(); RE[i].clear(); rdom[i].clear();

}

}

void add\_edge(int x, int y) { E[x].push\_back(y); }

void Union(int x, int y) { par[x] = y; }

int Find(int x, int c = 0) {

if (par[x] == x) return c ? -1 : x;

int p = Find(par[x], 1);

if (p == -1) return c ? par[x] : val[x];

if (sdom[val[x]] > sdom[val[par[x]]]) val[x] = val[par[x]];

par[x] = p;

return c ? p : val[x];

}

void dfs(int x) {

RS[S[x] = ++cs] = x;

par[cs] = sdom[cs] = val[cs] = cs;

for (int e : E[x]) {

if (S[e] == 0) dfs(e), rp[S[e]] = S[x];

RE[S[e]].push\_back(S[x]);

}

}

int solve(int s, int\* up) { // Calculate idoms

dfs(s);

for (int i = cs; i; i--) {

for (int e : RE[i]) sdom[i] = min(sdom[i], sdom[Find(e)]);

if (i > 1) rdom[sdom[i]].push\_back(i);

for (int e : rdom[i]) {

int p = Find(e);

if (sdom[p] == i) dom[e] = i;

else dom[e] = p;

}

if (i > 1) Union(i, rp[i]);

}

for (int i = 2; i <= cs; i++) if (sdom[i] != dom[i]) dom[i] = dom[dom[i]];

for (int i = 2; i <= cs; i++) up[RS[i]] = RS[dom[i]];

return cs;

}

}

**3. Flow**

**3.1 Bitpartite Matching**

const int MAX = 501;

vector<int> V[MAX];

int ma[MAX], mb[MAX];

bool vit[MAX];

bool dfs(int pos) {

vit[pos] = true;

for (int w : V[pos]) {

if (mb[w] == -1 || !vit[mb[w]] && dfs(mb[w])) {

ma[pos] = w;

mb[w] = pos;

return true;

}

}

return false;

}

int match(int n) {

memset(ma, -1, sizeof(ma));

memset(mb, -1, sizeof(mb));

int ans = 0;

for (int i = 0; i < n; i++) {

if (ma[i] == -1) {

memset(vit, false, sizeof(vit));

ans += dfs(i);

}

}

return ans;

}

**3.2 Hopcroft-Karp**

struct HopcroftKarp {

int n;

vector<vector<int>> V;

vector<int> ma, mb, lv;

vector<bool> vit;

void init(int \_n) {

n = \_n;

ma.resize(n); mb.resize(n);

lv.resize(n); V.resize(n);

vit.resize(n);

}

void add\_edge(int u, int v) { V[u].push\_back(v); }

bool bfs() {

queue<int> q;

fill(lv.begin(), lv.end(), 0);

for (int i = 0; i < n; i++)

if (ma[i] == -1 && !lv[i])

q.push(i), lv[i] = 1;

bool ok = false;

while (q.size()) {

int top = q.front(); q.pop();

for (int w : V[top]) {

if (mb[w] == -1) ok = true;

else if (!lv[mb[w]]) {

lv[mb[w]] = lv[top] + 1;

q.push(mb[w]);

}

}

}

return ok;

}

bool dfs(int a) {

if (vit[a]) return false;

vit[a] = true;

for (int w : V[a]) {

if (mb[w] == -1 || (!vit[mb[w]] && lv[mb[w]] == lv[a] + 1 && dfs(mb[w]))) {

ma[a] = w;

mb[w] = a;

return true;

}

}

return false;

}

int match() {

fill(ma.begin(), ma.end(), -1);

fill(mb.begin(), mb.end(), -1);

int ans = 0;

while (bfs()) {

fill(vit.begin(), vit.end(), false);

for (int i = 0; i < n; i++)

if (ma[i] == -1 && dfs(i)) ans++;

}

return ans;

}

};

**3.3 MCMF**

template<typename T>

struct MCMF {

struct Edge {

int to;

T cap, f, cost;

int dual;

T spare() { return cap - f; }

};

int n;

T ans, cot;

vector<vector<Edge>> E;

void init(int \_n) {

n = \_n;

E.clear(); E.resize(n);

}

void add\_edge(int u, int v, T cap, T cost) {

E[u].push\_back({ v,cap, 0, cost });

E[v].push\_back({ u, 0, 0, -cost });

E[u].back().dual = E[v].size() - 1;

E[v].back().dual = E[u].size() - 1;

}

bool spfa(int s, int t, bool apply = true) {

vector<T> dst(n, 1e9);

vector<int> prv(n, -1);

vector<Edge\*> sel(n);

vector<bool> chk(n);

dst[s] = 0;

queue<int> q;

q.push(s); chk[s] = true;

while (q.size()) {

int top = q.front(); q.pop();

chk[top] = false;

for (auto& l : E[top]) {

if (l.spare() > 0 && dst[top] + l.cost < dst[l.to]) {

dst[l.to] = dst[top] + l.cost;

prv[l.to] = top;

sel[l.to] = &l;

if (!chk[l.to]) {

q.push(l.to);

chk[l.to] = true;

}

}

}

}

if (prv[t] == -1) return false;

if (apply) {

T flow = 1e9;

for (int i = t; i != s; i = prv[i]) flow = min(flow, sel[i]->spare());

for (int i = t; i != s; i = prv[i]) {

sel[i]->f += flow;

E[sel[i]->to][sel[i]->dual].f -= flow;

cot += flow \* sel[i]->cost;

}

ans += flow;

}

return true;

}

pair<T, T> flow(int s, int t) {

ans = 0; cot = 0;

while (spfa(s, t));

return { ans, cot };

}

};

**3.4 Dinic**

template<typename T>

struct Dinic {

struct Edge {

int to;

T cap, f;

int dual;

T spare() { return cap - f; }

};

int n;

T ans;

vector<vector<Edge>> E;

vector<int> lv, work;

void init(int \_n) {

n = \_n;

E.clear(); E.resize(n);

lv.resize(n); work.resize(n);

}

void add\_edge(int u, int v, T cap) {

E[u].push\_back({ v,cap, 0 });

E[v].push\_back({ u, 0, 0 });

E[u].back().dual = E[v].size() - 1;

E[v].back().dual = E[u].size() - 1;;

}

bool bfs(int s, int t) {

fill(lv.begin(), lv.end(), -1);

lv[s] = 0;

queue<int> q; q.push(s);

while (q.size()) {

int top = q.front(); q.pop();

for (auto& l : E[top]) {

if (lv[l.to] == -1 && l.spare()) {

lv[l.to] = lv[top] + 1;

q.push(l.to);

}

}

}

return lv[t] != -1;

}

T dfs(int pos, int t, T flow) {

if (pos == t) return flow;

for (int& i = work[pos]; i < E[pos].size(); i++) {

auto& l = E[pos][i];

if (lv[l.to] == lv[pos] + 1 && l.spare()) {

T df = dfs(l.to, t, min(flow, l.spare()));

if (df) {

l.f += df;

E[l.to][l.dual].f -= df;

return df;

}

}

}

return 0;

}

T flow(int s, int t) {

ans = 0;

while (bfs(s, t)) {

fill(work.begin(), work.end(), 0);

while (1) {

T flow = dfs(s, t, 1e9);

if (!flow) break;

ans += flow;

}

}

return ans;

}

};

**3.5 Circulation**

template<typename T>

struct LRFlow { //by sgc109

Dinic<T> dinic;

int n, src, sink, fsrc, fsink;

vector<T> inSum, outSum;

void init(int \_n, int \_src, int \_sink) {

n = \_n, src = \_src, sink = \_sink;

fsrc = n, fsink = n + 1;

inSum = vector<T>(n);

outSum = vector<T>(n);

}

void add\_edge(int u, int v, int l, int r) {

dinic.add\_edge(u, v, r);

inSum[v] += l;

outSum[u] += l;

}

int flow() {

for (int i = 0; i < n; i++)

if (inSum[i]) dinic.add\_edge(fsrc, i, inSum[i]);

for (int i = 0; i < n; i++)

if (outSum[i]) dinic.add\_edge(i, fsink, outSum[i]);

dinic.add\_edge(sink, src, 1e9);

return dinic.flow();

}

};

**4. Strings**

**4.1 KMP**

vector<int> KMP(string from, string to) {

int n = from.size(), m = to.size();

vector<int> fail(m + 1);

for (int i = 1, j = 0; i < m; i++) {

while (j && to[i] != to[j]) j = fail[j];

if (to[i] == to[j]) j++;

fail[i + 1] = j;

}

vector<int> ans;

for (int i = 0, j = 0; i < n; i++) {

while (j && from[i] != to[j]) j = fail[j];

if (from[i] == to[j]) j++;

if (j == m) ans.push\_back(i - m + 1), j = fail[j];

}

return ans;

}

**4.2 Trie**

struct Trie {

map<char, Trie\*> to;

Trie\* fail;

bool end;

void insert(int idx, string& vec) {

if (idx == vec.size()) {

end = true;

return;

}

if (to.find(vec[idx]) == to.end())

to[vec[idx]] = new Trie();

to[vec[idx]]->insert(idx + 1, vec);

}

};

**4.3 Rabin-Karp Fingerprint**

typedef long long ll;

template<ll key = 29, ll mod = 1'000'000'007>

struct RabinKarp {

int n;

vector<ll> p;

void init(int n) {

this->n = n;

p.resize(n);

p[0] = 1;

for (int i = 1; i < n; i++) p[i] = (p[i - 1] \* key) % mod;

}

vector<Q> hashing(string& arr, int gap) {

assert(arr.size() <= n);

vector<Q> ans;

ll now = 0, idx = 0;

for (int i = 0; i < arr.size(); i++) {

if (i >= gap) {

ans.push\_back({ now, idx++ });

now = (now - p[gap - 1] \* arr[i - gap] % mod + mod) % mod;

}

now = (now \* key) % mod;

now = (now + arr[i]) % mod;

}

ans.push\_back({ now, idx });

return ans;

}

};

**4.4 Manacher**

int manacher(string str) {

string arr;

for (char c : str) {

arr.push\_back('#');

arr.push\_back(c);

}

arr.push\_back('#');

swap(str, arr);

int n = str.size();

vector<int> vec(n);

int r = 0, p = 0;

for (int i = 0; i < n; i++) {

if (i > r) vec[i] = 0;

else vec[i] = min(r - i, vec[2 \* p - i]);

while (i - vec[i] - 1 >= 0 && i + vec[i] + 1 < n && str[i - vec[i] - 1] == str[vec[i] + i + 1])

vec[i]++;

if (r < i + vec[i]) {

r = i + vec[i];

p = i;

}

}

return \*max\_element(vec.begin(), vec.end());

}

**4.5 Aho-Corasick**

Trie\* CreateTrie(vector<string> str) {

Trie\* trie = new Trie();

for (auto& s : str) trie->insert(0, s);

queue<Trie\*> q;

q.push(trie->fail = trie);

while (q.size()) {

Trie\* top = q.front(); q.pop();

for (auto& p : top->to) {

char c = p.first; Trie\* nxt = p.second;

if (top == trie) nxt->fail = trie;

else {

Trie\* f = top->fail;

while (f != trie && f->to.find(c) == f->to.end())

f = f->fail;

if (f->to.find(c) != f->to.end())

f = f->to[c];

nxt->fail = f;

}

if (nxt->fail->end) top->end = true;

q.push(nxt);

}

}

return trie;

}

bool AhoCorasick(string str, Trie\* root) {

bool ans = false;

Trie\* curr = root;

for (char c : str) {

while (curr != root && curr->to.find(c) == curr->to.end())

curr = curr->fail;

if (curr->to[c])

curr = curr->to[c];

if (curr->end) {

ans = true;

break;

}

}

return ans;

}

**4.6 Suffix Array and LCP**

vector<int> buildSA(string& str) {

int n = str.size();

vector<int> sa(n), r(n + 1), nr(n + 1);

for (int i = 0; i < n; i++) sa[i] = i, r[i] = str[i];

for (int d = 1; d < n; d <<= 1) {

auto cmp = [&](int i, int j) -> bool {

return r[i] < r[j] || (r[i] == r[j] && r[i + d] < r[j + d]);

};

sort(sa.begin(), sa.end(), cmp);

nr[sa[0]] = 1;

for (int i = 1; i < n; i++)

nr[sa[i]] = nr[sa[i - 1]] + cmp(sa[i - 1], sa[i]);

r = nr;

}

return sa;

}

vector<int> buildLCP(string& str, vector<int>& sa) {

int n = str.size();

vector<int> lcp(n + 1), isa(n + 1);

for (int i = 0; i < n; i++) isa[sa[i]] = i;

for (int i = 0, k = 0; i < n; i++) {

if (isa[i]) {

for (int j = sa[isa[i] - 1]; str[i + k] == str[j + k]; k++);

lcp[isa[i]] = (k ? k-- : 0);

}

}

return lcp;

}

**5. Geometry**

**5.1 Line-Segment Intersection**

//BOJ 12555, by shwldus067

typedef pair<int, int> pi;

#define x first

#define y second

struct Line { pi s, e; };

int ccw(pi a, pi b, pi c) {

int ret = (b.x - a.x) \* (c.y - a.y) - (b.y - a.y) \* (c.x - a.x);

if (ret > 0) return 1;

else if (ret == 0) return 0;

return -1;

}

int cross(pi a, pi b, Line l) {

int p = ccw(a, b, l.s), q = ccw(a, b, l.e);

int r = ccw(l.s, l.e, a), s = ccw(l.s, l.e, b);

if (p == 0 && q == 0 && r == 0 && s == 0) {

if (a > b) swap(a, b);

if (l.s > l.e) swap(l.s, l.e);

if (l.s == b || a == l.e) return 1;

return (a < l.e&& l.s < b) << 3;

}

return p \* q <= 0 && r \* s <= 0;

}

int rectCross(pi s, pi e, Line l) {

vector<pi> sq = { {s.x, s.y}, {s.x, e.y}, {e.x, e.y}, {e.x, s.y}, {s.x, s.y} };

int res = 0;

for (int i = 0; i < 4; i++) {

res += cross(sq[i], sq[i + 1], l);

if (cross(sq[i], sq[i], l)) res--;

}

return min(res, 4);

}

**5.2 Convex Hull**

typedef long long ll;

typedef pair<ll, ll> pi;

#define x first

#define y second

ll ccw(pi a, pi b, pi c, bool area = false) {

ll \_ = (b.x - a.x) \* (c.y - a.y) - (c.x - a.x) \* (b.y - a.y);

if (area) return \_;

if (\_ < 0) return -1;

else if (\_ > 0) return 1;

return 0;

}

ll pw(ll x) { return x \* x; }

ll dst(pi a, pi b) { return pw(a.x - b.x) + pw(a.y - b.y); }

vector<pi> hull(vector<pi>& vec) {

swap(vec[0], \*min\_element(vec.begin(), vec.end()));

sort(vec.begin() + 1, vec.end(), [&](auto& a, auto& b) -> bool {

ll cw = ccw(vec[0], a, b);

if (cw != 0) return cw > 0;

return dst(vec[0], a) < dst(vec[0], b);

});

vector<pi> ans;

for (auto& p : vec) {

while (ans.size() > 1 && ccw(ans[ans.size() - 2], ans.back(), p) <= 0)

ans.pop\_back();

ans.push\_back(p);

}

return ans;

}

**5.3 Smallest Enclosing Circle**

#include <random>

namespace cover\_2d {

//https://www.secmem.org/blog/2019/04/08/Smallest-Enclosing-Circle/

double eps = 1e-9;

using Point = complex<double>;

struct Circle { Point p; double r; };

double dist(Point p, Point q) { return abs(p - q); }

double area2(Point p, Point q) { return (conj(p) \* q).imag(); }

bool in(const Circle& c, Point p) { return dist(c.p, p) < c.r + eps; }

Circle INVAL = Circle{ Point(0, 0), -1 };

Circle mCC(Point a, Point b, Point c) {

b -= a; c -= a;

double d = 2 \* (conj(b) \* c).imag(); if (abs(d) < eps) return INVAL;

Point ans = (c \* norm(b) - b \* norm(c)) \* Point(0, -1) / d;

return Circle{ a + ans, abs(ans) };

}

Circle solve(vector<Point> p) {

mt19937 gen(0x94949); shuffle(p.begin(), p.end(), gen);

Circle c = INVAL;

for (int i = 0; i < p.size(); ++i) if (c.r < 0 || !in(c, p[i])) {

c = Circle{ p[i], 0 };

for (int j = 0; j <= i; ++j) if (!in(c, p[j])) {

Circle ans{ (p[i] + p[j]) \* 0.5, dist(p[i], p[j]) \* 0.5 };

if (c.r == 0) { c = ans; continue; }

Circle l, r; l = r = INVAL;

Point pq = p[j] - p[i];

for (int k = 0; k <= j; ++k) if (!in(ans, p[k])) {

double a2 = area2(pq, p[k] - p[i]);

Circle c = mCC(p[i], p[j], p[k]);

if (c.r < 0) continue;

else if (a2 > 0 && (l.r<0 || area2(pq, c.p - p[i]) > area2(pq, l.p - p[i]))) l = c;

else if (a2 < 0 && (r.r < 0 || area2(pq, c.p - p[i]) < area2(pq, r.p - p[i]))) r = c;

}

if (l.r < 0 && r.r < 0) c = ans;

else if (l.r < 0) c = r;

else if (r.r < 0) c = l;

else c = l.r <= r.r ? l : r;

}

}

return c;

}

};

**5.4 Point In Convex Polygon Check**

typedef long long ll;

typedef pair<ll, ll> pi;

#define x first

#define y second

bool f(vector<pi>& cv, pi p) {

int n = cv.size();

if (ccw(cv[0], cv[1], p) < 0 ||

ccw(cv[0], cv.back(), p) > 0) return false;

int lo = 1, hi = n - 1, ans = 1;

while (lo <= hi) {

int mid = lo + hi >> 1;

if (ccw(cv[0], cv[mid], p) > 0)

lo = mid + 1, ans = mid;

else hi = mid - 1;

}

return ccw(cv[ans], cv[(ans + 1) % n], p) >= 0;

}

**5.5 Point In Non-Convex Polygon Check**

typedef long long ll;

typedef pair<ll, ll> pi;

#define x first

#define y second

bool pointInRect(pi& p, vector<pi>& pos) {

int cnt = 0;

for (int i = 0; i < pos.size(); i++) {

int nxt = (i + 1) % pos.size();

double sx = pos[i].x, sy = pos[i].y;

double ex = pos[nxt].x, ey = pos[nxt].y;

if ((sy > p.y) != (ey > p.y)) {

double x = (ex - sx) \* (p.y - sy) / (ey - sy) + sx;

if (p.x < x) cnt++;

}

}

return cnt % 2;

}

**5.6 Rotating Calipers**

pi operator-(pi a, pi b) {

return { a.x - b.x, a.y - b.y };

}

ll get(vector<pi>& arr) {

vector<pi> cv = hull(arr);

int l = 0, r = 0;

for (int i = 0; i < cv.size(); i++) {

if (cv[l].x > cv[i].x) l = i;

if (cv[r].x < cv[i].x) r = i;

}

pi line = { 0, 1 };

ll ans = dst(cv[l], cv[r]);

int sz = cv.size();

for (int i = 0; i < sz; i++) {

if (ccw(cv[(l + 1) % sz] - cv[l], cv[r] - cv[(r + 1) % sz]) > 0)

l = (l + 1) % sz;

else

r = (r + 1) % sz;

ans = max(ans, dst(cv[l], cv[r]));

}

return ans;

}

**5.7 Half Plane Intersection**

//https://www.secmem.org/blog/2019/09/17/Half-Plane-Intersection/

#define sz(x) ((int)x.size())

typedef long long ll;

typedef long double ld;

struct point {

ld x, y;

point() {}

point(ld x, ld y) :x(x), y(y) {}

};

struct line {

point s, t;

line() {}

line(point s, point t) : s(s), t(t) {}

};

inline bool equals(ld a, ld b) { return abs(a - b) < 1e-9; }

bool line\_intersect(point& s1, point& e1, point& s2, point& e2, point& v) {

ld vx1 = e1.x - s1.x, vy1 = e1.y - s1.y;

ld vx2 = e2.x - s2.x, vy2 = e2.y - s2.y;

ld det = vx1 \* (-vy2) - (-vx2) \* vy1;

if (equals(det, 0)) return 0;

ld s = (ld)((s2.x - s1.x) \* (-vy2) + (s2.y - s1.y) \* vx2) / det;

v.x = s1.x + vx1 \* s;

v.y = s1.y + vy1 \* s;

return 1;

}

bool bad(line& a, line& b, line& c) {

point v;

if (!line\_intersect(a.s, a.t, b.s, b.t, v)) return 0;

ld crs = (c.t.x - c.s.x) \* (v.y - c.s.y) - (c.t.y - c.s.y) \* (v.x - c.s.x);

return crs < 0 || equals(crs, 0);

}

vector<point> HPI(vector<line>& ln) {

auto lsgn = [&](const line& a) {

if (a.s.y == a.t.y) return a.s.x > a.t.x;

return a.s.y > a.t.y;

};

sort(ln.begin(), ln.end(), [&](const line& a, const line& b) {

if (lsgn(a) != lsgn(b)) return lsgn(a) < lsgn(b);

return (a.t.x - a.s.x) \* (b.t.y - b.s.y) - (a.t.y - a.s.y) \* (b.t.x - b.s.x) > 0;

});

deque<line> dq;

for (int i = 0; i < sz(ln); i++) {

while (dq.size() >= 2 && bad(dq[dq.size() - 2], dq.back(), ln[i]))

dq.pop\_back();

while (dq.size() >= 2 && bad(dq[0], dq[1], ln[i]))

dq.pop\_front();

if (dq.size() < 2 || !bad(dq.back(), ln[i], dq[0]))

dq.push\_back(ln[i]);

}

vector<point> res;

if (dq.size() >= 3) for (int i = 0; i < sz(dq); i++) {

int j = (i + 1) % sz(dq);

point v;

if (!line\_intersect(dq[i].s, dq[i].t, dq[j].s, dq[j].t, v)) continue;

res.push\_back(v);

}

return res;

}

**6. Math**

**6.1 FFT, XOR-FFT**

namespace FFT {

using ll = long long;

using cpx = complex<double>;

const double PI = acos(-1);

void FFT(vector<cpx>& v, bool inv) {

ll S = v.size();

for (ll i = 1, j = 0; i < S; i++) {

ll bit = S / 2;

while (j >= bit) {

j -= bit;

bit /= 2;

}

j += bit;

if (i < j) swap(v[i], v[j]);

}

for (ll k = 1; k < S; k \*= 2) {

double angle = (inv ? PI / k : -PI / k);

cpx w(cos(angle), sin(angle));

for (ll i = 0; i < S; i += k \* 2) {

cpx z(1, 0);

for (ll j = 0; j < k; j++) {

cpx even = v[i + j];

cpx odd = v[i + j + k];

v[i + j] = even + z \* odd;

v[i + j + k] = even - z \* odd;

z \*= w;

}

}

}

if (inv)

for (ll i = 0; i < S; i++) v[i] /= S;

}

vector<ll> multiply(vector<ll>& v, vector<ll>& u) {

vector<cpx> vc(v.begin(), v.end());

vector<cpx> uc(u.begin(), u.end());

ll S = 2;

while (S < v.size() + u.size()) S \*= 2;

vc.resize(S); FFT(vc, false);

uc.resize(S); FFT(uc, false);

for (ll i = 0; i < S; i++) vc[i] \*= uc[i];

FFT(vc, true);

vector<ll> w(S);

for (ll i = 0; i < S; i++) w[i] = round(vc[i].real());

return w;

}

}

namespace XORFFT {

using ll = long long;

using cpx = complex<double>;

const double PI = acos(-1);

void XORFFT(vector<ll>& v, bool inv) {

ll S = v.size();

for (ll i = 1, j = 0; i < S; i++) {

ll bit = S / 2;

while (j >= bit) {

j -= bit;

bit /= 2;

}

j += bit;

if (i < j) swap(v[i], v[j]);

}

for (ll k = 1; k < S; k \*= 2) {

for (ll i = 0; i < S; i += k \* 2) {

for (ll j = 0; j < k; j++) {

ll even = v[i + j];

ll odd = v[i + j + k];

v[i + j] = even + odd;

v[i + j + k] = even - odd;

}

}

}

if (inv)

for (ll i = 0; i < S; i++) v[i] /= S;

}

vector<ll> XORmultiply(std::vector<ll>& v, std::vector<ll>& u) {

vector<ll> vc(v.begin(), v.end());

vector<ll> uc(u.begin(), u.end());

ll S = 2;

while (S < v.size() + u.size()) S \*= 2;

vc.resize(S); XORFFT(vc, false);

uc.resize(S); XORFFT(uc, false);

for (ll i = 0; i < S; i++) vc[i] \*= uc[i];

XORFFT(vc, true);

vector<ll> w(S);

for (ll i = 0; i < S; i++) w[i] = vc[i];

return w;

}

}

**6.2 Extended Euclidean**

typedef long long ll;

struct Euclid {

ll g, x, y;

};

//ax+by=1, get a, b

Euclid egcd(ll a, ll b) {

if (b == 0) return { a, 1, 0 };

Euclid ret = egcd(b, a % b);

return { ret.g, ret.y, ret.x - (a / b) \* ret.y };

}

**6.3 Z2 Matrix**

namespace Z2mat {

const int MAX = 501;

int n;

bitset<2 \* MAX> mat[MAX];

void init() { for (int i = 0; i < MAX; i++) mat[i].reset(); }

void input(vector<vector<bool>> arr) {

assert(arr.size() == arr[0].size());

init();

n = arr.size();

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++) {

mat[i][j] = arr[i][j];

if (i == j) mat[i][j + n] = true;

}

}

vector<vector<bool>> rev() {

for (int i = 0; i < n; i++) {

if (!mat[i][i])

for (int j = i + 1; j < n; j++)

if (mat[j][i]) swap(mat[i], mat[j]);

assert(mat[i][i]);

for (int j = 0; j < n; j++)

if (i != j && mat[j][i]) mat[j] ^= mat[i];

}

vector ans(n, vector<bool>(n));

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++) ans[i][j] = mat[i][j + n];

return ans;

}

}

**6.4 Miller-Rabin**

typedef unsigned long long ull;

vector<int> test = { 2, 7, 61 };

ull mypow(ull x, ull cnt, ull mod) {

x %= mod;

ull ans = 1LL;

for (; cnt; ans = (ans \* ans) % mod, cnt >>= 1LL)

if (cnt & 1) ans = (ans \* x) % mod;

return ans;

}

bool miller\_rabin(ull n, int a) {

if (a % n == 0) return true;

ull d = n - 1;

while (d) {

ull k = mypow(a, d, n);

if (k == n - 1) return true;

if (d & 1) return k == n - 1 || k == 1;

d >>= 1;

}

}

bool prime(ull x) {

for (auto& l : test)

if (!miller\_rabin(x, l)) return false;

return true;

}

**6.5 Euler phi Function**

typedef long long ll;

const int MAX = 100001;

ll phi[MAX], low[MAX];

//phi(n) is equal to the number of integers from 1 to n that are prime to n.

ll f(ll n) {

ll i; ll ret = n;

for (i = 2; i \* i <= n; i++) {

if (n % i == 0) {

ret -= ret / i;

while (n % i == 0) n /= i;

}

}

if (n != 1) ret -= ret / n;

return ret;

}

void fillPhi(int n) {

phi[1] = 1;

for (int i = 2; i <= n; i++) {

for (int j = i; j <= n; j += i) {

if (!low[j]) low[j] = i;

}

phi[i] = i;

for (int j = i; j != 1; ) {

int p = low[j];

while (j % p == 0) {

j /= p;

}

phi[i] = (1ll \* phi[i] \* (p - 1)) / p;

}

}

}

**6.6 Mobius Function**

typedef long long ll;

const ll MAX = 1000000;

ll u[MAX + 1];

//find the number of square-free numbers less than or eqaul to num

ll func(ll num) {

ll cnt = 0;

for (ll i = 1; i \* i <= num; i++)

cnt += u[i] \* (num / (i \* i));

return cnt;

}

void init() {

u[1] = 1;

for (int i = 1; i <= MAX; i++)

for (int j = 2 \* i; j <= MAX; j += i)

u[j] -= u[i];

}

**6.7 Modular Integer**

template <int MOD = 998'244'353>

struct Modular {

int value;

static const int MOD\_value = MOD;

Modular(long long v = 0) { value = v % MOD; if (value < 0) value += MOD; }

Modular(long long a, long long b) : value(0) { \*this += a; \*this /= b; }

Modular& operator+=(Modular const& b) { value += b.value; if (value >= MOD) value -= MOD; return \*this; }

Modular& operator-=(Modular const& b) { value -= b.value; if (value < 0) value += MOD; return \*this; }

Modular& operator\*=(Modular const& b) { value = (long long)value \* b.value % MOD; return \*this; }

friend Modular mexp(Modular a, long long e) {

Modular res = 1; while (e) { if (e & 1) res \*= a; a \*= a; e >>= 1; }

return res;

}

friend Modular inverse(Modular a) { return mexp(a, MOD - 2); }

Modular& operator/=(Modular const& b) { return \*this \*= inverse(b); }

friend Modular operator+(Modular a, Modular const b) { return a += b; }

friend Modular operator-(Modular a, Modular const b) { return a -= b; }

friend Modular operator-(Modular const a) { return 0 - a; }

friend Modular operator\*(Modular a, Modular const b) { return a \*= b; }

friend Modular operator/(Modular a, Modular const b) { return a /= b; }

friend std::ostream& operator<<(std::ostream& os, Modular const& a) { return os << a.value; }

friend bool operator==(Modular const& a, Modular const& b) { return a.value == b.value; }

friend bool operator!=(Modular const& a, Modular const& b) { return a.value != b.value; }

};

**7. Miscellaneous**

**7.1 CHT, DNC dp optimization**

**7.2 Lucas Theorem**

**7.3 Burnside’s Lemma**

**7.4 Hall’s Theorem**