

# CU-Later Code Library

yangster67, ikaurov, serichao

May 21th 2024

# Contents

## Templates

Ken's template . . . . .	1
Kevin's template . . . . .	1
Kevin's Template Extended . . . . .	1

## Geometry

## Strings

## Flows

$O(N^2M)$ , on unit networks $O(N^{1/2}M)$ . . . . .	3
MCMF – maximize flow, then minimize its cost. $O(Fmn)$ . . . . .	3

## Graphs

Kuhn's algorithm for bipartite matching . . . . .	5
Hungarian algorithm for Assignment Problem . . .	5
Dijkstra's Algorithm . . . . .	5
Eulerian Cycle DFS . . . . .	6
SCC and 2-SAT . . . . .	6
Finding Bridges . . . . .	6
Virtual Tree . . . . .	6
HLD on Edges DFS . . . . .	7
Centroid Decomposition . . . . .	7

## Math

Binary exponentiation . . . . .	7
Extended Euclidean Algorithm . . . . .	7
Linear Sieve . . . . .	7
Gaussian Elimination . . . . .	8
NTT . . . . .	8
FFT . . . . .	9
is_prime . . . . .	9

## Data Structures

Sparse Table . . . . .	9
Suffix Array and LCP array . . . . .	9
Aho Corasick Trie . . . . .	10
Convex Hull Trick . . . . .	11
Li-Chao Segment Tree . . . . .	11
Persistent Segment Tree . . . . .	11

## Miscellaneous

Ordered Set . . . . .	12
Measuring Execution Time . . . . .	12
Setting Fixed D.P. Precision . . . . .	12
Common Bugs and General Advice . . . . .	12

# Templates

## Ken's template

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define all(v) (v).begin(), (v).end()
4 typedef long long ll;
5 typedef long double ld;
6 #define pb push_back
7 #define sz(x) (int)(x).size()
8 #define fi first
9 #define se second
10 #define endl '\n'
```

## Kevin's template

```
1 // paste Kaurov's Template, minus last line
2 typedef vector<int> vi;
3 typedef vector<ll> vll;
4 typedef pair<int, int> pii;
5 typedef pair<ll, ll> pll;
6 typedef pair<double, double> pdd;
7 const ld PI = acos(-1);
8 const ll mod7 = 1e9 + 7;
9 const ll mod9 = 998244353;
10 const ll INF = 2*1024*1024*1023;
11 const char nl = '\n';
12 #define forn(i, n) for (int i = 0; i < int(n); i++)
13 ll k, n, m, u, v, w;
14 string s, t;
15
16 bool multiTest = 1;
17 void solve(int tt){
18 }
19
20 int main(){
21     ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
22     cout<<fixed<< setprecision(14);
23
24     int t = 1;
25     if (multiTest) cin >> t;
26     forn(ii, t) solve(ii);
27 }
```

## Kevin's Template Extended

- to type after the start of the contest

```
1 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4 using namespace __gnu_pbds;
5 template<class T> using ordered_set = tree<T, null_type,
6     ⇨ less<T>, rb_tree_tag, tree_order_statistics_node_update>;
7 vi d4x = {1, 0, -1, 0};
8 vi d4y = {0, 1, 0, -1};
9 vi d8x = {1, 0, -1, 0, 1, 1, -1, -1};
10 vi d8y = {0, 1, 0, -1, 1, -1, 1, -1};
11 mt19937
12     ⇨ rng(chrono::steady_clock::now().time_since_epoch().count());
```

## Geometry

```
1 template<typename T>
2 struct TPoint{
3     T x, y;
4     int id;
5     static constexpr T eps = static_cast<T>(1e-9);
6     TPoint() : x(0), y(0), id(-1) {}
7     TPoint(const T& x_, const T& y_) : x(x_), y(y_), id(-1) {}
8     TPoint(const T& x_, const T& y_, const int id_) : x(x_),
9     ⇨ y(y_), id(id_) {}
10
11     TPoint operator + (const TPoint& rhs) const {
```

```

11     return TPoint(x + rhs.x, y + rhs.y);
12 }
13 TPoint operator - (const TPoint& rhs) const {
14     return TPoint(x - rhs.x, y - rhs.y);
15 }
16 TPoint operator * (const T& rhs) const {
17     return TPoint(x * rhs, y * rhs);
18 }
19 TPoint operator / (const T& rhs) const {
20     return TPoint(x / rhs, y / rhs);
21 }
22 TPoint ort() const {
23     return TPoint(-y, x);
24 }
25 T abs2() const {
26     return x * x + y * y;
27 }
28 };
29 template<typename T>
30 bool operator< (TPoint<T>& A, TPoint<T>& B){
31     return make_pair(A.x, A.y) < make_pair(B.x, B.y);
32 }
33 template<typename T>
34 bool operator== (TPoint<T>& A, TPoint<T>& B){
35     return abs(A.x - B.x) <= TPoint<T>::eps && abs(A.y - B.y) <=
36     ↪ TPoint<T>::eps;
37 }
38 template<typename T>
39 struct TLine{
40     T a, b, c;
41     TLine() : a(0), b(0), c(0) {}
42     TLine(const T& a_, const T& b_, const T& c_) : a(a_), b(b_),
43     ↪ c(c_) {}
44     TLine(const TPoint<T>& p1, const TPoint<T>& p2){
45         a = p1.y - p2.y;
46         b = p2.x - p1.x;
47         c = -a * p1.x - b * p1.y;
48     }
49 };
50 template<typename T>
51 T det(const T& a11, const T& a12, const T& a21, const T& a22){
52     return a11 * a22 - a12 * a21;
53 }
54 template<typename T>
55 T sq(const T& a){
56     return a * a;
57 }
58 template<typename T>
59 T smul(const TPoint<T>& a, const TPoint<T>& b){
60     return a.x * b.x + a.y * b.y;
61 }
62 template<typename T>
63 T vmul(const TPoint<T>& a, const TPoint<T>& b){
64     return det(a.x, a.y, b.x, b.y);
65 }
66 template<typename T>
67 bool parallel(const TLine<T>& l1, const TLine<T>& l2){
68     return abs(vmul(TPoint<T>(l1.a, l1.b), TPoint<T>(l2.a,
69     ↪ l2.b))) <= TPoint<T>::eps;
70 }
71 template<typename T>
72 bool equivalent(const TLine<T>& l1, const TLine<T>& l2){
73     return parallel(l1, l2) &&
74     ↪ abs(det(l1.b, l1.c, l2.b, l2.c)) <= TPoint<T>::eps &&
75     ↪ abs(det(l1.a, l1.c, l2.a, l2.c)) <= TPoint<T>::eps;
76 }
77 template<typename T>
78 TPoint<T> intersection(const TLine<T>& l1, const TLine<T>&
79     ↪ l2){
80     return TPoint<T>(
81         det(-l1.c, l1.b, -l2.c, l2.b) / det(l1.a, l1.b, l2.a,
82         ↪ l2.b),
83         det(l1.a, -l1.c, l2.a, -l2.c) / det(l1.a, l1.b, l2.a,
84         ↪ l2.b)
85     );
86 }
87 template<typename T>

```

```

82 int sign(const T& x){
83     if (abs(x) <= TPoint<T>::eps) return 0;
84     return x > 0? +1 : -1;
85 }
86 template<typename T>
87 T area(const vector<TPoint<T>>& pts){
88     int n = sz(pts);
89     T ans = 0;
90     for (int i = 0; i < n; i++){
91         ans += vmul(pts[i], pts[(i + 1) % n]);
92     }
93     return abs(ans) / 2;
94 }
95 template<typename T>
96 T dist_pp(const TPoint<T>& a, const TPoint<T>& b){
97     return sqrt(sq(a.x - b.x) + sq(a.y - b.y));
98 }
99 template<typename T>
100 TLine<T> perp_line(const TLine<T>& l, const TPoint<T>& p){
101     T na = -l.b, nb = l.a, nc = -na * p.x - nb * p.y;
102     return TLine<T>(na, nb, nc);
103 }
104 template<typename T>
105 TPoint<T> projection(const TPoint<T>& p, const TLine<T>& l){
106     return intersection(l, perp_line(l, p));
107 }
108 template<typename T>
109 T dist_pl(const TPoint<T>& p, const TLine<T>& l){
110     return dist_pp(p, projection(p, l));
111 }
112 template<typename T>
113 struct TRay{
114     TLine<T> l;
115     TPoint<T> start, dirvec;
116     TRay() : l(), start(), dirvec() {}
117     TRay(const TPoint<T>& p1, const TPoint<T>& p2){
118         l = TLine<T>(p1, p2);
119         start = p1, dirvec = p2 - p1;
120     }
121 };
122 template<typename T>
123 bool is_on_line(const TPoint<T>& p, const TLine<T>& l){
124     return abs(l.a * p.x + l.b * p.y + l.c) <= TPoint<T>::eps;
125 }
126 template<typename T>
127 bool is_on_ray(const TPoint<T>& p, const TRay<T>& r){
128     if (is_on_line(p, r.l)){
129         return sign(smul(r.dirvec, TPoint<T>(p - r.start))) != -1;
130     }
131     else return false;
132 }
133 template<typename T>
134 bool is_on_seg(const TPoint<T>& P, const TPoint<T>& A, const
135     ↪ TPoint<T>& B){
136     return is_on_ray(P, TRay<T>(A, B)) && is_on_ray(P,
137     ↪ TRay<T>(B, A));
138 }
139 template<typename T>
140 T dist_pr(const TPoint<T>& P, const TRay<T>& R){
141     auto H = projection(P, R.l);
142     return is_on_ray(H, R)? dist_pp(P, H) : dist_pp(P, R.start);
143 }
144 template<typename T>
145 T dist_ps(const TPoint<T>& P, const TPoint<T>& A, const
146     ↪ TPoint<T>& B){
147     auto H = projection(P, TLine<T>(A, B));
148     if (is_on_seg(H, A, B)) return dist_pp(P, H);
149     else return min(dist_pp(P, A), dist_pp(P, B));
150 }
151 template<typename T>
152 bool acw(const TPoint<T>& A, const TPoint<T>& B){
153     T mul = vmul(A, B);
154     return mul > 0 || abs(mul) <= TPoint<T>::eps;
155 }
156 template<typename T>
157 bool cw(const TPoint<T>& A, const TPoint<T>& B){
158     T mul = vmul(A, B);

```

```

156     return mul < 0 || abs(mul) <= TPoint<T>::eps;
157 }
158 template<typename T>
159 vector<TPoint<T>> convex_hull(vector<TPoint<T>> pts){
160     sort(all(pts));
161     pts.erase(unique(all(pts)), pts.end());
162     vector<TPoint<T>> up, down;
163     for (auto p : pts){
164         while (sz(up) > 1 && acw(up.end()[-1] - up.end()[-2], p -
165         ↪ up.end()[-2])) up.pop_back();
166         while (sz(down) > 1 && cw(down.end()[-1] - down.end()[-2],
167         ↪ p - down.end()[-2])) down.pop_back();
168         up.pb(p), down.pb(p);
169     }
170     for (int i = sz(up) - 2; i >= 1; i--) down.pb(up[i]);
171     return down;
172 }
173 template<typename T>
174 bool in_triangle(TPoint<T>& P, TPoint<T>& A, TPoint<T>& B,
175     ↪ TPoint<T>& C){
176     if (is_on_seg(P, A, B) || is_on_seg(P, B, C) || is_on_seg(P,
177     ↪ C, A)) return true;
178     return cw(P - A, B - A) == cw(P - B, C - B) &&
179     ↪ cw(P - A, B - A) == cw(P - C, A - C);
180 }
181 template<typename T>
182 void prep_convex_poly(vector<TPoint<T>>& pts){
183     rotate(pts.begin(), min_element(all(pts)), pts.end());
184 }
185 // 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
186 template<typename T>
187 int in_convex_poly(TPoint<T>& p, vector<TPoint<T>>& pts){
188     int n = sz(pts);
189     if (!n) return 0;
190     if (n <= 2) return is_on_seg(p, pts[0], pts.back());
191     int l = 1, r = n - 1;
192     while (r - l > 1){
193         int mid = (l + r) / 2;
194         if (acw(pts[mid] - pts[0], p - pts[0])) l = mid;
195         else r = mid;
196     }
197     if (!in_triangle(p, pts[0], pts[l], pts[l + 1])) return 0;
198     if (is_on_seg(p, pts[l], pts[l + 1]) ||
199     ↪ is_on_seg(p, pts[0], pts.back()) ||
200     ↪ is_on_seg(p, pts[0], pts[l])
201     ) return 2;
202     return 1;
203 }
204 // 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
205 template<typename T>
206 int in_simple_poly(TPoint<T> p, vector<TPoint<T>>& pts){
207     int n = sz(pts);
208     bool res = 0;
209     for (int i = 0; i < n; i++){
210         auto a = pts[i], b = pts[(i + 1) % n];
211         if (is_on_seg(p, a, b)) return 2;
212         if ((a.y > p.y) - (b.y > p.y)) * vmul(b - p, a - p) >
213         ↪ TPoint<T>::eps){
214             res ^= 1;
215         }
216     }
217     return res;
218 }
219 template<typename T>
220 void minkowski_rotate(vector<TPoint<T>>& P){
221     int pos = 0;
222     for (int i = 1; i < sz(P); i++){
223         if (abs(P[i].y - P[pos].y) <= TPoint<T>::eps){
224             if (P[i].x < P[pos].x) pos = i;
225         }
226         else if (P[i].y < P[pos].y) pos = i;
227     }
228     rotate(P.begin(), P.begin() + pos, P.end());
229 }
230 // P and Q are strictly convex, points given in
231 ↪ counterclockwise order

```

```

227 template<typename T>
228 vector<TPoint<T>> minkowski_sum(vector<TPoint<T>> P,
229     ↪ vector<TPoint<T>> Q){
230     minkowski_rotate(P);
231     minkowski_rotate(Q);
232     P.pb(P[0]);
233     Q.pb(Q[0]);
234     vector<TPoint<T>> ans;
235     int i = 0, j = 0;
236     while (i < sz(P) - 1 || j < sz(Q) - 1){
237         ans.pb(P[i] + Q[j]);
238         T curmul;
239         if (i == sz(P) - 1) curmul = -1;
240         else if (j == sz(Q) - 1) curmul = +1;
241         else curmul = vmul(P[i + 1] - P[i], Q[j + 1] - Q[j]);
242         if (abs(curmul) < TPoint<T>::eps || curmul > 0) i++;
243         if (abs(curmul) < TPoint<T>::eps || curmul < 0) j++;
244     }
245     return ans;
246 }
247 using Point = TPoint<ll>; using Line = TLine<ll>; using Ray =
248     ↪ TRay<ll>; const ld PI = acos(-1);

```

## Strings

```

1 vector<int> prefix_function(string s){
2     int n = sz(s);
3     vector<int> pi(n);
4     for (int i = 1; i < n; i++){
5         int k = pi[i - 1];
6         while (k > 0 && s[i] != s[k]){
7             k = pi[k - 1];
8         }
9         pi[i] = k + (s[i] == s[k]);
10    }
11    return pi;
12 }
13 vector<int> kmp(string s, string k){
14     string st = k + "#" + s;
15     vector<int> res;
16     auto pi = pf(st);
17     for (int i = 0; i < sz(st); i++){
18         if (pi[i] == sz(k)){
19             res.pb(i - 2 * sz(k));
20         }
21     }
22     return res;
23 }
24 vector<int> z_function(string s){
25     int n = sz(s);
26     vector<int> z(n);
27     int l = 0, r = 0;
28     for (int i = 1; i < n; i++){
29         if (r >= i) z[i] = min(z[i - l], r - i + 1);
30         while (i + z[i] < n && s[z[i]] == s[i + z[i]]){
31             z[i]++;
32         }
33         if (i + z[i] - 1 > r){
34             l = i, r = i + z[i] - 1;
35         }
36     }
37     return z;
38 }

```

## Flows

$O(N^2M)$ , on unit networks  $O(N^{1/2}M)$

```

1 struct FlowEdge {
2     int v, u;
3     long long cap, flow = 0;
4     FlowEdge(int v, int u, long long cap) : v(v), u(u),
5     ↪ cap(cap) {}
6 };

```

```

6 struct Dinic {
7     const long long flow_inf = 1e18;
8     vector<FlowEdge> edges;
9     vector<vector<int>> adj;
10    int n, m = 0;
11    int s, t;
12    vector<int> level, ptr;
13    queue<int> q;
14    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
15        adj.resize(n);
16        level.resize(n);
17        ptr.resize(n);
18    }
19    void add_edge(int v, int u, long long cap) {
20        edges.emplace_back(v, u, cap);
21        edges.emplace_back(u, v, 0);
22        adj[v].push_back(m);
23        adj[u].push_back(m + 1);
24        m += 2;
25    }
26    bool bfs() {
27        while (!q.empty()) {
28            int v = q.front();
29            q.pop();
30            for (int id : adj[v]) {
31                if (edges[id].cap - edges[id].flow < 1)
32                    continue;
33                if (level[edges[id].u] != -1)
34                    continue;
35                level[edges[id].u] = level[v] + 1;
36                q.push(edges[id].u);
37            }
38        }
39        return level[t] != -1;
40    }
41    long long dfs(int v, long long pushed) {
42        if (pushed == 0)
43            return 0;
44        if (v == t)
45            return pushed;
46        for (int& cid = ptr[v]; cid < (int)adj[v].size();
47             ↪ cid++) {
48            int id = adj[v][cid];
49            int u = edges[id].u;
50            if (level[v] + 1 != level[u] || edges[id].cap -
51             ↪ edges[id].flow < 1)
52                continue;
53            long long tr = dfs(u, min(pushed, edges[id].cap -
54             ↪ edges[id].flow));
55            if (tr == 0)
56                continue;
57            edges[id].flow += tr;
58            edges[id ^ 1].flow -= tr;
59            return tr;
60        }
61        return 0;
62    }
63    long long flow() {
64        long long f = 0;
65        while (true) {
66            fill(level.begin(), level.end(), -1);
67            level[s] = 0;
68            q.push(s);
69            if (!bfs())
70                break;
71            fill(ptr.begin(), ptr.end(), 0);
72            while (long long pushed = dfs(s, flow_inf)) {
73                f += pushed;
74            }
75        }
76        return f;
77    }
78    // To recover flow through original edges: iterate over even
79    ↪ indices in edges.

```

MCMF – maximize flow, then minimize its cost.  $O(Fmn)$ .

```

1 #include <ext/pb_ds/priority_queue.hpp>
2 template <typename T, typename C>
3 class MCMF {
4 public:
5     static constexpr T eps = (T) 1e-9;
6
7     struct edge {
8         int from;
9         int to;
10        T c;
11        T f;
12        C cost;
13    };
14
15    int n;
16    vector<vector<int>> g;
17    vector<edge> edges;
18    vector<C> d;
19    vector<C> pot;
20    __gnu_pbds::priority_queue<pair<C, int>> q;
21    vector<typename decltype(q)::point_iterator> its;
22    vector<int> pe;
23    const C INF_C = numeric_limits<C>::max() / 2;
24
25    explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0),
26    ↪ its(n), pe(n) {}
27
28    int add(int from, int to, T forward_cap, C edge_cost, T
29    ↪ backward_cap = 0) {
30        assert(0 <= from && from < n && 0 <= to && to < n);
31        assert(forward_cap >= 0 && backward_cap >= 0);
32        int id = static_cast<int>(edges.size());
33        g[from].push_back(id);
34        edges.push_back({from, to, forward_cap, 0, edge_cost});
35        g[to].push_back(id + 1);
36        edges.push_back({to, from, backward_cap, 0, -edge_cost});
37        return id;
38    }
39
40    void expath(int st) {
41        fill(d.begin(), d.end(), INF_C);
42        q.clear();
43        fill(its.begin(), its.end(), q.end());
44        its[st] = q.push({pot[st], st});
45        d[st] = 0;
46        while (!q.empty()) {
47            int i = q.top().second;
48            q.pop();
49            its[i] = q.end();
50            for (int id : g[i]) {
51                const edge &e = edges[id];
52                int j = e.to;
53                if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
54                    d[j] = d[i] + e.cost;
55                    pe[j] = id;
56                    if (its[j] == q.end()) {
57                        its[j] = q.push({pot[j] - d[j], j});
58                    } else {
59                        q.modify(its[j], {pot[j] - d[j], j});
60                    }
61                }
62            }
63        }
64        swap(d, pot);
65    }
66
67    pair<T, C> max_flow(int st, int fin) {
68        T flow = 0;
69        C cost = 0;
70        bool ok = true;
71        for (auto& e : edges) {
72            if (e.c - e.f > eps && e.cost + pot[e.from] - pot[e.to]
73            ↪ < 0) {
74                ok = false;

```

```

72     break;
73 }
74 }
75 if (ok) {
76     expath(st);
77 } else {
78     vector<int> deg(n, 0);
79     for (int i = 0; i < n; i++) {
80         for (int eid : g[i]) {
81             auto& e = edges[eid];
82             if (e.c - e.f > eps) {
83                 deg[e.to] += 1;
84             }
85         }
86     }
87     vector<int> que;
88     for (int i = 0; i < n; i++) {
89         if (deg[i] == 0) {
90             que.push_back(i);
91         }
92     }
93     for (int b = 0; b < (int) que.size(); b++) {
94         for (int eid : g[que[b]]) {
95             auto& e = edges[eid];
96             if (e.c - e.f > eps) {
97                 deg[e.to] -= 1;
98                 if (deg[e.to] == 0) {
99                     que.push_back(e.to);
100                 }
101             }
102         }
103     }
104     fill(pot.begin(), pot.end(), INF_C);
105     pot[st] = 0;
106     if (static_cast<int>(que.size()) == n) {
107         for (int v : que) {
108             if (pot[v] < INF_C) {
109                 for (int eid : g[v]) {
110                     auto& e = edges[eid];
111                     if (e.c - e.f > eps) {
112                         if (pot[v] + e.cost < pot[e.to]) {
113                             pot[e.to] = pot[v] + e.cost;
114                             pe[e.to] = eid;
115                         }
116                     }
117                 }
118             }
119         }
120     } else {
121         que.assign(1, st);
122         vector<bool> in_queue(n, false);
123         in_queue[st] = true;
124         for (int b = 0; b < (int) que.size(); b++) {
125             int i = que[b];
126             in_queue[i] = false;
127             for (int id : g[i]) {
128                 const edge &e = edges[id];
129                 if (e.c - e.f > eps && pot[i] + e.cost <
130 ↪ pot[e.to]) {
131                     pot[e.to] = pot[i] + e.cost;
132                     pe[e.to] = id;
133                     if (!in_queue[e.to]) {
134                         que.push_back(e.to);
135                         in_queue[e.to] = true;
136                     }
137                 }
138             }
139         }
140     }
141     while (pot[fin] < INF_C) {
142         T push = numeric_limits<T>::max();
143         int v = fin;
144         while (v != st) {
145             const edge &e = edges[pe[v]];
146             push = min(push, e.c - e.f);
147             v = e.from;

```

```

148     }
149     v = fin;
150     while (v != st) {
151         edge &e = edges[pe[v]];
152         e.f += push;
153         edge &back = edges[pe[v] ^ 1];
154         back.f -= push;
155         v = e.from;
156     }
157     flow += push;
158     cost += push * pot[fin];
159     expath(st);
160 }
161 return {flow, cost};
162 }
163 };
164
165 // Examples: MCMF<int, int> g(n); g.add(u,v,c,w,0);
166 ↪ g.max_flow(s,t).
167 // To recover flow through original edges: iterate over even
168 ↪ indices in edges.

```

## Graphs

### Kuhn's algorithm for bipartite matching

```

1 /*
2  The graph is split into 2 halves of n1 and n2 vertices.
3  Complexity: O(n1 * m). Usually runs much faster. MUCH
4  ↪ FASTER!!!
5 */
6
7 const int N = 305;
8
9 vector<int> g[N]; // Stores edges from left half to right.
10 bool used[N]; // Stores if vertex from left half is used.
11 int mt[N]; // For every vertex in right half, stores to which
12 ↪ vertex in left half it's matched (-1 if not matched).
13
14 bool try_dfs(int v){
15     if (used[v]) return false;
16     used[v] = 1;
17     for (auto u : g[v]){
18         if (mt[u] == -1 || try_dfs(mt[u])){
19             mt[u] = v;
20             return true;
21         }
22     }
23     return false;
24 }
25
26 int main(){
27     // .....
28     for (int i = 1; i <= n2; i++) mt[i] = -1;
29     for (int i = 1; i <= n1; i++) used[i] = 0;
30     for (int i = 1; i <= n1; i++){
31         if (try_dfs(i)){
32             for (int j = 1; j <= n1; j++) used[j] = 0;
33         }
34     }
35     vector<pair<int, int>> ans;
36     for (int i = 1; i <= n2; i++){
37         if (mt[i] != -1) ans.pb({mt[i], i});
38     }
39
40     // Finding maximal independent set: size = # of nodes - # of
41     ↪ edges in matching.
42     // To construct: launch Kuhn-like DFS from unmatched nodes in
43     ↪ the left half.
44     // Independent set = visited nodes in left half + unvisited in
45     ↪ right half.
46     // Finding minimal vertex cover: complement of maximal
47     ↪ independent set.

```

## Hungarian algorithm for Assignment Problem

- Given a 1-indexed  $(n \times m)$  matrix  $A$ , select a number in each row such that each column has at most 1 number selected, and the sum of the selected numbers is minimized.

```
1 int INF = 1e9; // constant greater than any number in the
  ↪ matrix
2 vector<int> u(n+1), v(m+1), p(m+1), way(m+1);
3 for (int i=1; i<=n; ++i) {
4     p[0] = i;
5     int j0 = 0;
6     vector<int> minv (m+1, INF);
7     vector<bool> used (m+1, false);
8     do {
9         used[j0] = true;
10        int i0 = p[j0], delta = INF, j1;
11        for (int j=1; j<=m; ++j)
12            if (!used[j]) {
13                int cur = A[i0][j]-u[i0]-v[j];
14                if (cur < minv[j])
15                    minv[j] = cur, way[j] = j0;
16                if (minv[j] < delta)
17                    delta = minv[j], j1 = j;
18            }
19        for (int j=0; j<=m; ++j)
20            if (used[j])
21                u[p[j]] += delta, v[j] -= delta;
22            else
23                minv[j] -= delta;
24        j0 = j1;
25    } while (p[j0] != 0);
26    do {
27        int j1 = way[j0];
28        p[j0] = p[j1];
29        j0 = j1;
30    } while (j0);
31 }
32 vector<int> ans (n+1); // ans[i] stores the column selected
  ↪ for row i
33 for (int j=1; j<=m; ++j)
34     ans[p[j]] = j;
35 int cost = -v[0]; // the total cost of the matching
```

## Dijkstra's Algorithm

```
1 priority_queue<pair<ll, ll>, vector<pair<ll, ll>>,
  ↪ greater<pair<ll, ll>>> q;
2 dist[start] = 0;
3 q.push({0, start});
4 while (!q.empty()){
5     auto [d, v] = q.top();
6     q.pop();
7     if (d != dist[v]) continue;
8     for (auto [u, w] : g[v]){
9         if (dist[u] > dist[v] + w){
10            dist[u] = dist[v] + w;
11            q.push({dist[u], u});
12        }
13    }
14 }
```

## Eulerian Cycle DFS

```
1 void dfs(int v){
2     while (!g[v].empty()){
3         int u = g[v].back();
4         g[v].pop_back();
5         dfs(u);
6         ans.pb(v);
7     }
8 }
```

## SCC and 2-SAT

```
1 void scc(vector<vector<int>>& g, int* idx) {
2     int n = g.size(), ct = 0;
3     int out[n];
4     vector<int> ginv[n];
5     memset(out, -1, sizeof out);
6     memset(idx, -1, n * sizeof(int));
7     function<void(int)> dfs = [&](int cur) {
8         out[cur] = INT_MAX;
9         for(int v : g[cur]) {
10            ginv[v].push_back(cur);
11            if(out[v] == -1) dfs(v);
12        }
13        ct++; out[cur] = ct;
14    };
15    vector<int> order;
16    for(int i = 0; i < n; i++) {
17        order.push_back(i);
18        if(out[i] == -1) dfs(i);
19    }
20    sort(order.begin(), order.end(), [&](int& u, int& v) {
21        return out[u] > out[v];
22    });
23    ct = 0;
24    stack<int> s;
25    auto dfs2 = [&](int start) {
26        s.push(start);
27        while(!s.empty()) {
28            int cur = s.top();
29            s.pop();
30            idx[cur] = ct;
31            for(int v : ginv[cur])
32                if(idx[v] == -1) s.push(v);
33        }
34    };
35    for(int v : order) {
36        if(idx[v] == -1) {
37            dfs2(v);
38            ct++;
39        }
40    }
41 }
42
43 // 0 => impossible, 1 => possible
44 pair<int, vector<int>> sat2(int n, vector<pair<int, int>>&
  ↪ clauses) {
45     vector<int> ans(n);
46     vector<vector<int>> g(2*n + 1);
47     for(auto [x, y] : clauses) {
48         x = x < 0 ? -x + n : x;
49         y = y < 0 ? -y + n : y;
50         int nx = x <= n ? x + n : x - n;
51         int ny = y <= n ? y + n : y - n;
52         g[nx].push_back(y);
53         g[ny].push_back(x);
54     }
55     int idx[2*n + 1];
56     scc(g, idx);
57     for(int i = 1; i <= n; i++) {
58         if(idx[i] == idx[i + n]) return {0, {}};
59         ans[i - 1] = idx[i + n] < idx[i];
60     }
61     return {1, ans};
62 }
```

## Finding Bridges

```
1 /*
2 Bridges.
3 Results are stored in a map "is_bridge".
4 For each connected component, call "dfs(starting vertex,
  ↪ starting vertex)".
5 */
6 const int N = 2e5 + 10; // Careful with the constant!
7
8 vector<int> g[N];
```



```

9  int tin[N], fup[N], timer;
10 map<pair<int, int>, bool> is_bridge;
11
12 void dfs(int v, int p){
13     tin[v] = ++timer;
14     fup[v] = tin[v];
15     for (auto u : g[v]){
16         if (!tin[u]){
17             dfs(u, v);
18             if (fup[u] > tin[v]){
19                 is_bridge[{u, v}] = is_bridge[{v, u}] = true;
20             }
21             fup[v] = min(fup[v], fup[u]);
22         }
23         else{
24             if (u != p) fup[v] = min(fup[v], tin[u]);
25         }
26     }
27 }

```

## Virtual Tree

```

1  // order stores the nodes in the queried set
2  sort(all(order), [&] (int u, int v){return tin[u] < tin[v]});
3  int m = sz(order);
4  for (int i = 1; i < m; i++){
5      order.pb(lca(order[i], order[i - 1]));
6  }
7  sort(all(order), [&] (int u, int v){return tin[u] < tin[v]});
8  order.erase(unique(all(order)), order.end());
9  vector<int> stk{order[0]};
10 for (int i = 1; i < sz(order); i++){
11     int v = order[i];
12     while (tout[stk.back()] < tout[v]) stk.pop_back();
13     int u = stk.back();
14     vg[u].pb({v, dep[v] - dep[u]});
15     stk.pb(v);
16 }

```

## HLD on Edges DFS

```

1  void dfs1(int v, int p, int d){
2      par[v] = p;
3      for (auto e : g[v]){
4          if (e.fi == p){
5              g[v].erase(find(all(g[v]), e));
6              break;
7          }
8      }
9      dep[v] = d;
10     sz[v] = 1;
11     for (auto [u, c] : g[v]){
12         dfs1(u, v, d + 1);
13         sz[v] += sz[u];
14     }
15     if (!g[v].empty()) iter_swap(g[v].begin(),
16     ↪ max_element(all(g[v]), comp));
17 }
18 void dfs2(int v, int rt, int c){
19     pos[v] = sz(a);
20     a.pb(c);
21     root[v] = rt;
22     for (int i = 0; i < sz(g[v]); i++){
23         auto [u, c] = g[v][i];
24         if (!i) dfs2(u, rt, c);
25         else dfs2(u, u, c);
26     }
27 }
28 int getans(int u, int v){
29     int res = 0;
30     for (; root[u] != root[v]; v = par[root[v]]){
31         if (dep[root[u]] > dep[root[v]]) swap(u, v);
32         res = max(res, rmq(0, 0, n - 1, pos[root[v]], pos[v]));
33     }
34     if (pos[u] > pos[v]) swap(u, v);
35     return max(res, rmq(0, 0, n - 1, pos[u] + 1, pos[v]));

```

```

35 }

```

## Centroid Decomposition

```

1  vector<char> res(n), seen(n), sz(n);
2  function<int(int, int)> get_size = [&](int node, int fa) {
3      sz[node] = 1;
4      for (auto& ne : g[node]) {
5          if (ne == fa || seen[ne]) continue;
6          sz[node] += get_size(ne, node);
7      }
8      return sz[node];
9  };
10 function<int(int, int, int)> find_centroid = [&](int node, int
11     ↪ fa, int t) {
12     for (auto& ne : g[node])
13         if (ne != fa && !seen[ne] && sz[ne] > t / 2) return
14         ↪ find_centroid(ne, node, t);
15     return node;
16 };
17 function<void(int, char)> solve = [&](int node, char cur) {
18     get_size(node, -1); auto c = find_centroid(node, -1,
19     ↪ sz[node]);
20     seen[c] = 1, res[c] = cur;
21     for (auto& ne : g[c]) {
22         if (seen[ne]) continue;
23         solve(ne, char(cur + 1)); // we can pass c here to build
24         ↪ tree
25     }
26 };

```

## Math

### Binary exponentiation

```

1  ll power(ll a, ll b){
2      ll res = 1;
3      for (; b; a = a * a % MOD, b >>= 1){
4          if (b & 1) res = res * a % MOD;
5      }
6      return res;
7  }

```

## Extended Euclidean Algorithm

```

1  // gives (x, y) for ax + by = g
2  // solutions given (x0, y0): a(x0 + kb/g) + b(y0 - ka/g) = g
3  int gcd(int a, int b, int& x, int& y) {
4      x = 1, y = 0; int sum1 = a;
5      int x2 = 0, y2 = 1, sum2 = b;
6      while (sum2) {
7          int q = sum1 / sum2;
8          tie(x, x2) = make_tuple(x2, x - q * x2);
9          tie(y, y2) = make_tuple(y2, y - q * y2);
10         tie(sum1, sum2) = make_tuple(sum2, sum1 - q * sum2);
11     }
12     return sum1;
13 }

```

## Linear Sieve

### • Mobius Function

```

1  vector<int> prime;
2  bool is_composite[MAX_N];
3  int mu[MAX_N];
4
5  void sieve(int n){
6      fill(is_composite, is_composite + n, 0);
7      mu[1] = 1;
8      for (int i = 2; i < n; i++){
9          if (!is_composite[i]){
10             prime.push_back(i);
11             mu[i] = -1; //i is prime

```



```

12     }
13     for (int j = 0; j < prime.size() && i * prime[j] < n; j++){
14         is_composite[i * prime[j]] = true;
15         if (i % prime[j] == 0){
16             mu[i * prime[j]] = 0; //prime[j] divides i
17             break;
18         } else {
19             mu[i * prime[j]] = -mu[i]; //prime[j] does not divide i
20         }
21     }
22 }
23 }

```

### • Euler's Totient Function

```

1 vector<int> prime;
2 bool is_composite[MAX_N];
3 int phi[MAX_N];
4
5 void sieve(int n){
6     fill(is_composite, is_composite + n, 0);
7     phi[1] = 1;
8     for (int i = 2; i < n; i++){
9         if (!is_composite[i]){
10             prime.push_back(i);
11             phi[i] = i - 1; //i is prime
12         }
13         for (int j = 0; j < prime.size() && i * prime[j] < n; j++){
14             is_composite[i * prime[j]] = true;
15             if (i % prime[j] == 0){
16                 phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
17                 divides i
18                 break;
19             } else {
20                 phi[i * prime[j]] = phi[i] * phi[prime[j]]; //prime[j]
21                 does not divide i
22             }
23         }
24     }
25 }

```

## Gaussian Elimination

```

1 bool is_0(Z v) { return v.x == 0; }
2 Z abs(Z v) { return v; }
3 bool is_0(double v) { return abs(v) < 1e-9; }
4
5 // 1 => unique solution, 0 => no solution, -1 => multiple
6 solutions
7 template <typename T>
8 int gaussian_elimination(vector<vector<T>> &a, int limit) {
9     if (a.empty() || a[0].empty()) return -1;
10    int h = (int)a.size(), w = (int)a[0].size(), r = 0;
11    for (int c = 0; c < limit; c++) {
12        int id = -1;
13        for (int i = r; i < h; i++) {
14            if (!is_0(a[i][c]) && (id == -1 || abs(a[id][c]) <
15            abs(a[i][c]))) {
16                id = i;
17            }
18        }
19        if (id == -1) continue;
20        if (id > r) {
21            swap(a[r], a[id]);
22            for (int j = c; j < w; j++) a[id][j] = -a[id][j];
23        }
24        vector<int> nonzero;
25        for (int j = c; j < w; j++) {
26            if (!is_0(a[r][j])) nonzero.push_back(j);
27        }
28        T inv_a = 1 / a[r][c];
29        for (int i = r + 1; i < h; i++) {
30            if (is_0(a[i][c])) continue;
31            T coeff = -a[i][c] * inv_a;
32            for (int j : nonzero) a[i][j] += coeff * a[r][j];
33        }
34        ++r;
35    }
36    return r;
37 }

```

```

33     }
34     for (int row = h - 1; row >= 0; row--) {
35         for (int c = 0; c < limit; c++) {
36             if (!is_0(a[row][c])) {
37                 T inv_a = 1 / a[row][c];
38                 for (int i = row - 1; i >= 0; i--) {
39                     if (is_0(a[i][c])) continue;
40                     T coeff = -a[i][c] * inv_a;
41                     for (int j = c; j < w; j++) a[i][j] += coeff *
42                     a[row][j];
43                 }
44                 break;
45             }
46         } // not-free variables: only it on its line
47         for (int i = r; i < h; i++) if (!is_0(a[i][limit])) return 0;
48         return (r == limit) ? 1 : -1;
49     }
50 }
51
52 template <typename T>
53 pair<int, vector<T>> solve_linear(vector<vector<T>> a, const
54 vector<T> &b, int w) {
55     int h = (int)a.size();
56     for (int i = 0; i < h; i++) a[i].push_back(b[i]);
57     int sol = gaussian_elimination(a, w);
58     if (!sol) return {0, vector<T>()};
59     vector<T> x(w, 0);
60     for (int i = 0; i < h; i++) {
61         for (int j = 0; j < w; j++) {
62             if (!is_0(a[i][j])) {
63                 x[j] = a[i][w] / a[i][j];
64                 break;
65             }
66         }
67     }
68     return {sol, x};
69 }

```

## NTT

```

1 void ntt(vector<ll> &a, int f) {
2     int n = (int)a.size();
3     vector<ll> w(n);
4     vector<int> rev(n);
5     for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] / 2) | ((i
6     & 1) * (n / 2));
7     for (int i = 0; i < n; i++) {
8         if (i < rev[i]) swap(a[i], a[rev[i]]);
9     }
10    ll wn = power(f ? (MOD + 1) / 3 : 3, (MOD - 1) / n);
11    w[0] = 1;
12    for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn % MOD;
13    for (int mid = 1; mid < n; mid *= 2) {
14        for (int i = 0; i < n; i += 2 * mid) {
15            for (int j = 0; j < mid; j++) {
16                ll x = a[i + j], y = a[i + j + mid] * w[n / (2 * mid)
17                * j] % MOD;
18                a[i + j] = (x + y) % MOD, a[i + j + mid] = (x + MOD -
19                y) % MOD;
20            }
21        }
22    }
23    if (f) {
24        ll iv = power(n, MOD - 2);
25        for (auto& x : a) x = x * iv % MOD;
26    }
27 }
28
29 vector<ll> mul(vector<ll> a, vector<ll> b) {
30     int n = 1, m = (int)a.size() + (int)b.size() - 1;
31     while (n < m) n *= 2;
32     a.resize(n), b.resize(n);
33     ntt(a, 0), ntt(b, 0); // if squaring, you can save one NTT
34     here
35     for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % MOD;
36     ntt(a, 1);
37     a.resize(m);
38     return a;
39 }

```

```
34 }
```

## FFT

```
1 const ld PI = acosl(-1);
2 auto mul = [&](const vector<ld>& aa, const vector<ld>& bb) {
3     int n = (int)aa.size(), m = (int)bb.size(), bit = 1;
4     while ((1 << bit) < n + m - 1) bit++;
5     int len = 1 << bit;
6     vector<complex<ld>> a(len), b(len);
7     vector<int> rev(len);
8     for (int i = 0; i < n; i++) a[i].real(aa[i]);
9     for (int i = 0; i < m; i++) b[i].real(bb[i]);
10    for (int i = 0; i < len; i++) rev[i] = (rev[i >> 1] >> 1) |
    ↪ ((i & 1) << (bit - 1));
11    auto fft = [&](vector<complex<ld>>& p, int inv) {
12        for (int i = 0; i < len; i++)
13            if (i < rev[i]) swap(p[i], p[rev[i]]);
14        for (int mid = 1; mid < len; mid *= 2) {
15            auto w1 = complex<ld>(cos(PI / mid), (inv ? -1 : 1) *
    ↪ sin(PI / mid));
16            for (int i = 0; i < len; i += mid * 2) {
17                auto wk = complex<ld>(1, 0);
18                for (int j = 0; j < mid; j++, wk = wk * w1) {
19                    auto x = p[i + j], y = wk * p[i + j + mid];
20                    p[i + j] = x + y, p[i + j + mid] = x - y;
21                }
22            }
23        }
24        if (inv == 1) {
25            for (int i = 0; i < len; i++) p[i].real(p[i].real() /
    ↪ len);
26        }
27    };
28    fft(a, 0), fft(b, 0);
29    for (int i = 0; i < len; i++) a[i] = a[i] * b[i];
30    fft(a, 1);
31    a.resize(n + m - 1);
32    vector<ld> res(n + m - 1);
33    for (int i = 0; i < n + m - 1; i++) res[i] = a[i].real();
34    return res;
35 };
```

## is\_prime

- (Miller–Rabin primality test)

```
1 typedef __int128_t i128;
2
3 i128 power(i128 a, i128 b, i128 MOD = 1, i128 res = 1) {
4     for (; b /= 2, (a *= a) %= MOD)
5         if (b & 1) (res *= a) %= MOD;
6     return res;
7 }
8
9 bool is_prime(ll n) {
10     if (n < 2) return false;
11     static constexpr int A[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
12     int s = __builtin_ctzll(n - 1);
13     ll d = (n - 1) >> s;
14     for (auto a : A) {
15         if (a == n) return true;
16         ll x = (ll)power(a, d, n);
17         if (x == 1 || x == n - 1) continue;
18         bool ok = false;
19         for (int i = 0; i < s - 1; ++i) {
20             x = ll((i128)x * x % n); // potential overflow!
21             if (x == n - 1) {
22                 ok = true;
23                 break;
24             }
25         }
26         if (!ok) return false;
27     }
28     return true;
29 }
```

```
1 typedef __int128_t i128;
2
3 ll pollard_rho(ll x) {
4     ll s = 0, t = 0, c = rng() % (x - 1) + 1;
5     ll stp = 0, goal = 1, val = 1;
6     for (goal = 1;; goal *= 2, s = t, val = 1) {
7         for (stp = 1; stp <= goal; ++stp) {
8             t = ll(((i128)t * t + c) % x);
9             val = ll(((i128)val * abs(t - s) % x);
10            if ((stp % 127) == 0) {
11                ll d = gcd(val, x);
12                if (d > 1) return d;
13            }
14        }
15        ll d = gcd(val, x);
16        if (d > 1) return d;
17    }
18 }
19
20 ll get_max_factor(ll _x) {
21     ll max_factor = 0;
22     function<void(ll)> fac = [&](ll x) {
23         if (x <= max_factor || x < 2) return;
24         if (is_prime(x)) {
25             max_factor = max_factor > x ? max_factor : x;
26             return;
27         }
28         ll p = x;
29         while (p >= x) p = pollard_rho(x);
30         while ((x % p) == 0) x /= p;
31         fac(x), fac(p);
32     };
33     fac(_x);
34     return max_factor;
35 }
```

## Data Structures

### Sparse Table

```
1 const int N = 2e5 + 10, LOG = 20; // Change the constant!
2 template<typename T>
3 struct SparseTable{
4     int lg[N];
5     T st[N][LOG];
6     int n;
7
8     // Change this function
9     function<T(T, T)> f = [&](T a, T b){
10         return min(a, b);
11     };
12
13     void build(vector<T>& a){
14         n = sz(a);
15         lg[1] = 0;
16         for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
17
18         for (int k = 0; k < LOG; k++){
19             for (int i = 0; i < n; i++){
20                 if (!k) st[i][k] = a[i];
21                 else st[i][k] = f(st[i][k - 1], st[min(n - 1, i + (1 <<
    ↪ (k - 1)))][k - 1]);
22             }
23         }
24     }
25
26     T query(int l, int r){
27         int sz = r - l + 1;
28         return f(st[l][lg[sz]], st[r - (1 << lg[sz]) + 1][lg[sz]]);
29     }
30 };
```

### Suffix Array and LCP array

- (uses SparseTable above)

```

1 struct SuffixArray{
2     vector<int> p, c, h;
3     SparseTable<int> st;
4     /*
5      In the end, array c gives the position of each suffix in p
6      using 1-based indexation!
7      */
8
9     SuffixArray() {}
10
11     SuffixArray(string s){
12         buildArray(s);
13         buildLCP(s);
14         buildSparse();
15     }
16
17     void buildArray(string s){
18         int n = sz(s) + 1;
19         p.resize(n), c.resize(n);
20         for (int i = 0; i < n; i++) p[i] = i;
21         sort(all(p), [&] (int a, int b){return s[a] < s[b];});
22         c[p[0]] = 0;
23         for (int i = 1; i < n; i++){
24             c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
25         }
26         vector<int> p2(n), c2(n);
27         // w is half-length of each string.
28         for (int w = 1; w < n; w <= 1){
29             for (int i = 0; i < n; i++){
30                 p2[i] = (p[i] - w + n) % n;
31             }
32             vector<int> cnt(n);
33             for (auto i : c) cnt[i]++;
34             for (int i = 1; i < n; i++) cnt[i] += cnt[i - 1];
35             for (int i = n - 1; i >= 0; i--){
36                 p[--cnt[c[p2[i]]]] = p2[i];
37             }
38             c2[p[0]] = 0;
39             for (int i = 1; i < n; i++){
40                 c2[p[i]] = c2[p[i - 1]] +
41                     (c[p[i]] != c[p[i - 1]] ||
42                      c[(p[i] + w) % n] != c[(p[i - 1] + w) % n]);
43             }
44             c.swap(c2);
45         }
46         p.erase(p.begin());
47     }
48
49     void buildLCP(string s){
50         // The algorithm assumes that suffix array is already
51         // built on the same string.
52         int n = sz(s);
53         h.resize(n - 1);
54         int k = 0;
55         for (int i = 0; i < n; i++){
56             if (c[i] == n){
57                 k = 0;
58                 continue;
59             }
60             int j = p[c[i]];
61             while (i + k < n && j + k < n && s[i + k] == s[j + k])
62                 k++;
63             h[c[i] - 1] = k;
64             if (k) k--;
65         }
66         /*
67          Then an RMQ Sparse Table can be built on array h
68          to calculate LCP of 2 non-consecutive suffixes.
69          */
70     }
71
72     void buildSparse(){
73         st.build(h);
74     }
75
76     // l and r must be in 0-BASED INDEXATION
77     int lcp(int l, int r){

```

```

76         l = c[l] - 1, r = c[r] - 1;
77         if (l > r) swap(l, r);
78         return st.query(l, r - 1);
79     }
80 };

```

## Aho Corasick Trie

- For each node in the trie, the suffix link points to the longest proper suffix of the represented string. The terminal-link tree has square-root height (can be constructed by DFS).

```

1 const int S = 26;
2
3 // Function converting char to int.
4 int ctoi(char c){
5     return c - 'a';
6 }
7
8 // To add terminal links, use DFS
9 struct Node{
10     vector<int> nxt;
11     int link;
12     bool terminal;
13
14     Node() {
15         nxt.assign(S, -1), link = 0, terminal = 0;
16     }
17 };
18
19 vector<Node> trie(1);
20
21 // add_string returns the terminal vertex.
22 int add_string(string& s){
23     int v = 0;
24     for (auto c : s){
25         int cur = ctoi(c);
26         if (trie[v].nxt[cur] == -1){
27             trie[v].nxt[cur] = sz(trie);
28             trie.emplace_back();
29         }
30         v = trie[v].nxt[cur];
31     }
32     trie[v].terminal = 1;
33     return v;
34 }
35
36 /*
37 Suffix links are compressed.
38 This means that:
39     If vertex v has a child by letter x, then:
40         trie[v].nxt[x] points to that child.
41     If vertex v doesn't have such child, then:
42         trie[v].nxt[x] points to the suffix link of that child
43         if we would actually have it.
44 */
45 void add_links(){
46     queue<int> q;
47     q.push(0);
48     while (!q.empty()){
49         auto v = q.front();
50         int u = trie[v].link;
51         q.pop();
52         for (int i = 0; i < S; i++){
53             int& ch = trie[v].nxt[i];
54             if (ch == -1){
55                 ch = v? trie[u].nxt[i] : 0;
56             }
57             else{
58                 trie[ch].link = v? trie[u].nxt[i] : 0;
59                 q.push(ch);
60             }
61         }
62     }
63 }

```

```

64
65 bool is_terminal(int v){
66     return trie[v].terminal;
67 }
68
69 int get_link(int v){
70     return trie[v].link;
71 }
72
73 int go(int v, char c){
74     return trie[v].nxt[ctoi(c)];
75 }

```

## Convex Hull Trick

- Allows to insert a linear function to the hull in (1) and get the minimum/maximum value of the stored function at a point in  $O(\log n)$ .
- NOTE: The lines must be added in the order of decreasing/increasing gradients. CAREFULLY CHECK THE SETUP BEFORE USING!
- IMPORTANT: THE DEFAULT VERSION SURELY WORKS. IF MODIFIED VERSIONS DON'T WORK, TRY TRANSFORMING THEM TO THE DEFAULT ONE BY CHANGING SIGNS.

```

1 struct line{
2     ll k, b;
3     ll f(ll x){
4         return k * x + b;
5     };
6 };
7
8 vector<line> hull;
9
10 void add_line(line nl){
11     if (!hull.empty() && hull.back().k == nl.k){
12         nl.b = min(nl.b, hull.back().b); // Default: minimum. For
        ↪ maximum change "min" to "max".
13         hull.pop_back();
14     }
15     while (sz(hull) > 1){
16         auto& l1 = hull.end()[-2], l2 = hull.back();
17         if ((nl.b - l1.b) * (l2.k - nl.k) >= (nl.b - l2.b) * (l1.k
        ↪ - nl.k)) hull.pop_back(); // Default: decreasing gradient
        ↪ k. For increasing k change the sign to <=.
18         else break;
19     }
20     hull.pb(nl);
21 }
22
23 ll get(ll x){
24     int l = 0, r = sz(hull);
25     while (r - l > 1){
26         int mid = (l + r) / 2;
27         if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid; //
        ↪ Default: minimum. For maximum change the sign to <=.
28         else r = mid;
29     }
30     return hull[l].f(x);
31 }

```

## Li-Chao Segment Tree

- allows to add linear functions in any order and query minimum/maximum value of those at a point, all in  $O(\log n)$ .
- Clear: clear()

```

1 const ll INF = 1e18; // Change the constant!
2 struct LiChaoTree{
3     struct line{
4         ll k, b;

```

```

5         line(){
6             k = b = 0;
7         };
8         line(ll k_, ll b_){
9             k = k_, b = b_;
10        };
11        ll f(ll x){
12            return k * x + b;
13        };
14    };
15    int n;
16    bool minimum, on_points;
17    vector<ll> pts;
18    vector<line> t;
19
20    void clear(){
21        for (auto& l : t) l.k = 0, l.b = minimum? INF : -INF;
22    }
23
24    LiChaoTree(int n_, bool min_){ // This is a default
        ↪ constructor for numbers in range [0, n - 1].
25        n = n_, minimum = min_, on_points = false;
26        t.resize(4 * n);
27        clear();
28    };
29
30    LiChaoTree(vector<ll> pts_, bool min_){ // This constructor
        ↪ will build LCT on the set of points you pass. The points
        ↪ may be in any order and contain duplicates.
31        pts = pts_, minimum = min_;
32        sort(all(pts));
33        pts.erase(unique(all(pts)), pts.end());
34        on_points = true;
35        n = sz(pts);
36        t.resize(4 * n);
37        clear();
38    };
39
40    void add_line(int v, int l, int r, line nl){
41        // Adding on segment [l, r)
42        int m = (l + r) / 2;
43        ll lval = on_points? pts[l] : l, mval = on_points? pts[m]
        ↪ : m;
44        if ((minimum && nl.f(mval) < t[v].f(mval)) || (!minimum &&
        ↪ nl.f(mval) > t[v].f(mval))) swap(t[v], nl);
45        if (r - l == 1) return;
46        if ((minimum && nl.f(lval) < t[v].f(lval)) || (!minimum &&
        ↪ nl.f(lval) > t[v].f(lval))) add_line(2 * v + 1, l, m, nl);
47        else add_line(2 * v + 2, m, r, nl);
48    }
49
50    ll get(int v, int l, int r, int x){
51        int m = (l + r) / 2;
52        if (r - l == 1) return t[v].f(on_points? pts[x] : x);
53        else{
54            if (minimum) return min(t[v].f(on_points? pts[x] : x), x
        ↪ < m? get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
55            else return max(t[v].f(on_points? pts[x] : x), x < m?
        ↪ get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
56        }
57    }
58
59    void add_line(ll k, ll b){
60        add_line(0, 0, n, line(k, b));
61    }
62
63    ll get(ll x){
64        return get(0, 0, n, on_points? lower_bound(all(pts), x) -
        ↪ pts.begin() : x);
65        // Always pass the actual value of x, even if LCT is on
        ↪ points.
66    };

```

## Persistent Segment Tree

- for RSQ

```

1 struct Node {
2     ll val;
3     Node *l, *r;
4
5     Node(ll x) : val(x), l(nullptr), r(nullptr) {}
6     Node(Node *ll, Node *rr) {
7         l = ll, r = rr;
8         val = 0;
9         if (l) val += l->val;
10        if (r) val += r->val;
11    }
12    Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
13 };
14 const int N = 2e5 + 20;
15 ll a[N];
16 Node *roots[N];
17 int n, cnt = 1;
18 Node *build(int l = 1, int r = n) {
19     if (l == r) return new Node(a[l]);
20     int mid = (l + r) / 2;
21     return new Node(build(l, mid), build(mid + 1, r));
22 }
23 Node *update(Node *node, int val, int pos, int l = 1, int r =
    ↪ n) {
24     if (l == r) return new Node(val);
25     int mid = (l + r) / 2;
26     if (pos > mid)
27         return new Node(node->l, update(node->r, val, pos, mid
    ↪ + 1, r));
28     else return new Node(update(node->l, val, pos, l, mid),
    ↪ node->r);
29 }
30 ll query(Node *node, int a, int b, int l = 1, int r = n) {
31     if (l > b || r < a) return 0;
32     if (l >= a && r <= b) return node->val;
33     int mid = (l + r) / 2;
34     return query(node->l, a, b, l, mid) + query(node->r, a, b,
    ↪ mid + 1, r);
35 }

```

## Miscellaneous

### Ordered Set

```

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

```

### Measuring Execution Time

```

1 ld tic = clock();
2 // execute algo...
3 ld tac = clock();
4 // Time in milliseconds
5 cerr << (tac - tic) / CLOCKS_PER_SEC * 1000 << endl;
6 // No need to comment out the print because it's done to cerr.

```

### Setting Fixed D.P. Precision

```

1 cout << setprecision(d) << fixed;
2 // Each number is rounded to d digits after the decimal point,
    ↪ and truncated.

```

### Common Bugs and General Advice

- Check overflow, array bounds
- Check variable overloading
- Check special cases (n=1?)
- Do something instead of nothing, stay organized
- Write stuff down!
- Don't get stuck on one approach!