CU-Later Code Library

yangster67, ikaurov, serichaoo

May 21th 2024

Contents

Templates

Ken's template

```cpp
#include <bits/stdc++.h>
using namespace std;
#define all(v) (v).begin(), (v).end()
typedef long long ll;
typedef long double ld;
#define pb push_back
#define sz(x) (int)(x).size()
#define fi first
#define se second
#define endl '\n'
```

Kevin's template

```cpp
// paste Kaurov's Template, minus last line
typedef vector<int> vi;
typedef vector<ll> vll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef pair<double, double> pdd;
const ld PI = acosl(-1);
const ll mod7 = 1e9 + 7;
const ll mod9 = 998244353;
const ll INF = 2*1024*1024*1023;
const char nl = '\n';
#define forn(i, n) for (int i = 0; i < int(n); i++)
ll k, n, m, u, v, w;
string s, t;

bool multiTest = 1;
void solve(int tt){
}

int main(){
  ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
  cout<<fixed<< setprecision(14);

  int t = 1;
  if (multiTest) cin >> t;
  forn(ii, t) solve(ii);
}
```

Kevin's Template Extended

- to type after the start of the contest

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

Geometry

```cpp
template<typename T>
struct TPoint{
  T x, y;
  int id;
  static constexpr T eps = static_cast<T>(1e-9);
  TPoint() : x(0), y(0), id(-1) {}
  TPoint(const T& x_, const T& y_) : x(x_), y(y_), id(-1) {}
  TPoint(const T& x_, const T& y_, const int id_) : x(x_),
  y(y_), id(id_) {}

  TPoint operator + (const TPoint& rhs) const {
```

```cpp
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) <=
    ↪  TPoint<T>::eps;
36  }
37  template<typename T>
38  struct TLine{
39    T a, b, c;
40    TLine() : a(0), b(0), c(0) {}
41    TLine(const T& a_, const T& b_, const T& c_) : a(a_), b(b_),
    ↪  c(c_) {}
42    TLine(const TPoint<T>& p1, const TPoint<T>& p2){
43      a = p1.y - p2.y;
44      b = p2.x - p1.x;
45      c = -a * p1.x - b * p1.y;
46    }
47  };
48  template<typename T>
49  T det(const T& a11, const T& a12, const T& a21, const T& a22){
50    return a11 * a22 - a12 * a21;
51  }
52  template<typename T>
53  T sq(const T& a){
54    return a * a;
55  }
56  template<typename T>
57  T smul(const TPoint<T>& a, const TPoint<T>& b){
58    return a.x * b.x + a.y * b.y;
59  }
60  template<typename T>
61  T vmul(const TPoint<T>& a, const TPoint<T>& b){
62    return det(a.x, a.y, b.x, b.y);
63  }
64  template<typename T>
65  bool parallel(const TLine<T>& l1, const TLine<T>& l2){
66    return abs(vmul(TPoint<T>(l1.a, l1.b), TPoint<T>(l2.a,
    ↪  l2.b))) <= TPoint<T>::eps;
67  }
68  template<typename T>
69  bool equivalent(const TLine<T>& l1, const TLine<T>& l2){
70    return parallel(l1, l2) &&
71    abs(det(l1.b, l1.c, l2.b, l2.c)) <= TPoint<T>::eps &&
72    abs(det(l1.a, l1.c, l2.a, l2.c)) <= TPoint<T>::eps;
73  }
74  template<typename T>
75  TPoint<T> intersection(const TLine<T>& l1, const TLine<T>&
    ↪  l2){
76    return TPoint<T>(
77      det(-l1.c, l1.b, -l2.c, l2.b) / det(l1.a, l1.b, l2.a,
    ↪  l2.b),
78      det(l1.a, -l1.c, l2.a, -l2.c) / det(l1.a, l1.b, l2.a,
    ↪  l2.b)
79    );
80  }
81  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
    ↪  TPoint<T>& B){
135   return is_on_ray(P, TRay<T>(A, B)) && is_on_ray(P,
    ↪  TRay<T>(B, A));
136 }
137 template<typename T>
138 T dist_pr(const TPoint<T>& P, const TRay<T>& R){
139   auto H = projection(P, R.l);
140   return is_on_ray(H, R)? dist_pp(P, H) : dist_pp(P, R.start);
141 }
142 template<typename T>
143 T dist_ps(const TPoint<T>& P, const TPoint<T>& A, const
    ↪  TPoint<T>& B){
144   auto H = projection(P, TLine<T>(A, B));
145   if (is_on_seg(H, A, B)) return dist_pp(P, H);
146   else return min(dist_pp(P, A), dist_pp(P, B));
147 }
148 template<typename T>
149 bool acw(const TPoint<T>& A, const TPoint<T>& B){
150   T mul = vmul(A, B);
151   return mul > 0 || abs(mul) <= TPoint<T>::eps;
152 }
153 template<typename T>
154 bool cw(const TPoint<T>& A, const TPoint<T>& B){
155   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 -
    ↪ up.end()[-2])) up.pop_back();
165            while (sz(down) > 1 && cw(down.end()[-1] - down.end()[-2],
    ↪ p - down.end()[-2])) down.pop_back();
166            up.pb(p), down.pb(p);
167        }
168        for (int i = sz(up) - 2; i >= 1; i--) down.pb(up[i]);
169        return down;
170    }
171
172    template<typename T>
173    bool in_triangle(TPoint<T>& P, TPoint<T>& A, TPoint<T>& B,
    ↪ TPoint<T>& C){
174        if (is_on_seg(P, A, B) || is_on_seg(P, B, C) || is_on_seg(P,
    ↪ C, A)) return true;
175        return cw(P - A, B - A) == cw(P - B, C - B) &&
176        cw(P - A, B - A) == cw(P - C, A - C);
177    }
178    template<typename T>
179    void prep_convex_poly(vector<TPoint<T>>& pts){
180        rotate(pts.begin(), min_element(all(pts)), pts.end());
181    }
182    // 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
183    template<typename T>
184    int in_convex_poly(TPoint<T>& p, vector<TPoint<T>>& pts){
185        int n = sz(pts);
186        if (!n) return 0;
187        if (n <= 2) return is_on_seg(p, pts[0], pts.back());
188        int l = 1, r = n - 1;
189        while (r - l > 1){
190            int mid = (l + r) / 2;
191            if (acw(pts[mid] - pts[0], p - pts[0])) l = mid;
192            else r = mid;
193        }
194        if (!in_triangle(p, pts[0], pts[l], pts[l + 1])) return 0;
195        if (is_on_seg(p, pts[l], pts[l + 1]) ||
196        is_on_seg(p, pts[0], pts.back()) ||
197        is_on_seg(p, pts[0], pts[1])
198        ) return 2;
199        return 1;
200    }
201    // 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
202    template<typename T>
203    int in_simple_poly(TPoint<T> p, vector<TPoint<T>>& pts){
204        int n = sz(pts);
205        bool res = 0;
206        for (int i = 0; i < n; i++){
207            auto a = pts[i], b = pts[(i + 1) % n];
208            if (is_on_seg(p, a, b)) return 2;
209            if (((a.y > p.y) - (b.y > p.y)) * vmul(b - p, a - p) >
    ↪ TPoint<T>::eps){
210                res ^= 1;
211            }
212        }
213        return res;
214    }
215    template<typename T>
216    void minkowski_rotate(vector<TPoint<T>>& P){
217        int pos = 0;
218        for (int i = 1; i < sz(P); i++){
219            if (abs(P[i].y - P[pos].y) <= TPoint<T>::eps){
220                if (P[i].x < P[pos].x) pos = i;
221            }
222            else if (P[i].y < P[pos].y) pos = i;
223        }
224        rotate(P.begin(), P.begin() + pos, P.end());
225    }
226    // P and Q are strictly convex, points given in
    ↪  counterclockwise order
```

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

$O(N^2 M)$, 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),
    ↪ cap(cap) {}
5    };
```

```cpp
struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;
    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }
    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }
    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)
                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }
    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size();
            cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap -
                edges[id].flow < 1)
                continue;
            long long tr = dfs(u, min(pushed, edges[id].cap -
                edges[id].flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }
    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
// To recover flow through original edges: iterate over even
//   indices in edges.
```

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

```cpp
#include <ext/pb_ds/priority_queue.hpp>
template <typename T, typename C>
class MCMF {
 public:
    static constexpr T eps = (T) 1e-9;

    struct edge {
        int from;
        int to;
        T c;
        T f;
        C cost;
    };

    int n;
    vector<vector<int>> g;
    vector<edge> edges;
    vector<C> d;
    vector<C> pot;
    __gnu_pbds::priority_queue<pair<C, int>> q;
    vector<typename decltype(q)::point_iterator> its;
    vector<int> pe;
    const C INF_C = numeric_limits<C>::max() / 2;

    explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0),
        its(n), pe(n) {}

    int add(int from, int to, T forward_cap, C edge_cost, T
        backward_cap = 0) {
        assert(0 <= from && from < n && 0 <= to && to < n);
        assert(forward_cap >= 0 && backward_cap >= 0);
        int id = static_cast<int>(edges.size());
        g[from].push_back(id);
        edges.push_back({from, to, forward_cap, 0, edge_cost});
        g[to].push_back(id + 1);
        edges.push_back({to, from, backward_cap, 0, -edge_cost});
        return id;
    }

    void expath(int st) {
        fill(d.begin(), d.end(), INF_C);
        q.clear();
        fill(its.begin(), its.end(), q.end());
        its[st] = q.push({pot[st], st});
        d[st] = 0;
        while (!q.empty()) {
            int i = q.top().second;
            q.pop();
            its[i] = q.end();
            for (int id : g[i]) {
                const edge &e = edges[id];
                int j = e.to;
                if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
                    d[j] = d[i] + e.cost;
                    pe[j] = id;
                    if (its[j] == q.end()) {
                        its[j] = q.push({pot[j] - d[j], j});
                    } else {
                        q.modify(its[j], {pot[j] - d[j], j});
                    }
                }
            }
        }
        swap(d, pot);
    }

    pair<T, C> max_flow(int st, int fin) {
        T flow = 0;
        C cost = 0;
        bool ok = true;
        for (auto& e : edges) {
            if (e.c - e.f > eps && e.cost + pot[e.from] - pot[e.to]
                < 0) {
                ok = false;
```

4

```
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 <
     ↪ pot[e.to]) {
130                 pot[e.to] = pot[i] + e.cost;
131                 pe[e.to] = id;
132                 if (!in_queue[e.to]) {
133                   que.push_back(e.to);
134                   in_queue[e.to] = true;
135                 }
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);
     ↪ g.max_flow(s,t).
166   // To recover flow through original edges: iterate over even
     ↪ 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
     ↪ FASTER!!!
4    */
5    const int N = 305;
6
7    vector<int> g[N]; // Stores edges from left half to right.
8    bool used[N]; // Stores if vertex from left half is used.
9    int mt[N]; // For every vertex in right half, stores to which
     ↪ vertex in left half it's matched (-1 if not matched).
10
11   bool try_dfs(int v){
12     if (used[v]) return false;
13     used[v] = 1;
14     for (auto u : g[v]){
15       if (mt[u] == -1 || try_dfs(mt[u])){
16         mt[u] = v;
17         return true;
18       }
19     }
20     return false;
21   }
22
23   int main(){
24   // ......
25     for (int i = 1; i <= n2; i++) mt[i] = -1;
26     for (int i = 1; i <= n1; i++) used[i] = 0;
27     for (int i = 1; i <= n1; i++){
28       if (try_dfs(i)){
29         for (int j = 1; j <= n1; j++) used[j] = 0;
30       }
31     }
32     vector<pair<int, int>> ans;
33     for (int i = 1; i <= n2; i++){
34       if (mt[i] != -1) ans.pb({mt[i], i});
35     }
36   }
37
38   // Finding maximal independent set: size = # of nodes - # of
     ↪ edges in matching.
39   // To construct: launch Kuhn-like DFS from unmatched nodes in
     ↪ the left half.
40   // Independent set = visited nodes in left half + unvisited in
     ↪ right half.
41   // Finding minimal vertex cover: complement of maximal
     ↪ independent set.
```

Dijkstra's Algorithm

```cpp
priority_queue<pair<ll, ll>, vector<pair<ll, ll>>,
↪  greater<pair<ll, ll>>> q;
dist[start] = 0;
q.push({0, start});
while (!q.empty()){
    auto [d, v] = q.top();
    q.pop();
    if (d != dist[v]) continue;
    for (auto [u, w] : g[v]){
      if (dist[u] > dist[v] + w){
        dist[u] = dist[v] + w;
        q.push({dist[u], u});
      }
    }
}
```

EULERIAN CYCLE DFS

```cpp
void dfs(int v){
  while (!g[v].empty()){
    int u = g[v].back();
    g[v].pop_back();
    dfs(u);
    ans.pb(v);
  }
}
```

Strongly Connected Components: Kosaraju's Algorithm

```cpp
vector<vector<int>> adj, adj_rev;
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
    used[v] = true;

    for (auto u : adj[v])
        if (!used[u])
            dfs1(u);

    order.push_back(v);
}

void dfs2(int v) {
    used[v] = true;
    component.push_back(v);

    for (auto u : adj_rev[v])
        if (!used[u])
            dfs2(u);
}

int main(){
// ......
    used.assign(n, false);

    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i);
    used.assign(n, false);
    reverse(order.begin(), order.end());
    for (auto v : order)
        if (!used[v]) {
            dfs2(v);
            // process
            component.clear();
        }
}
```

Finding Bridges

```cpp
/*
Bridges.
Results are stored in a map "is_bridge".
For each connected component, call "dfs(starting vertex,
↪  starting vertex)".
*/
const int N = 2e5 + 10; // Careful with the constant!

vector<int> g[N];
int tin[N], fup[N], timer;
map<pair<int, int>, bool> is_bridge;

void dfs(int v, int p){
  tin[v] = ++timer;
  fup[v] = tin[v];
  for (auto u : g[v]){
    if (!tin[u]){
      dfs(u, v);
      if (fup[u] > tin[v]){
        is_bridge[{u, v}] = is_bridge[{v, u}] = true;
      }
      fup[v] = min(fup[v], fup[u]);
    }
    else{
      if (u != p) fup[v] = min(fup[v], tin[u]);
    }
  }
}
```

Virtual Tree

```cpp
// order stores the nodes in the queried set
sort(all(order), [&] (int u, int v){return tin[u] < tin[v];});
int m = sz(order);
for (int i = 1; i < m; i++){
    order.pb(lca(order[i], order[i - 1]));
}
sort(all(order), [&] (int u, int v){return tin[u] < tin[v];});
order.erase(unique(all(order)), order.end());
vector<int> stk{order[0]};
for (int i = 1; i < sz(order); i++){
    int v = order[i];
    while (tout[stk.back()] < tout[v]) stk.pop_back();
    int u = stk.back();
    vg[u].pb({v, dep[v] - dep[u]});
    stk.pb(v);
}
```

HLD ON EDGES DFS

```cpp
void dfs1(int v, int p, int d){
  par[v] = p;
  for (auto e : g[v]){
    if (e.fi == p){
      g[v].erase(find(all(g[v]), e));
      break;
    }
  }
  dep[v] = d;
  sz[v] = 1;
  for (auto [u, c] : g[v]){
    dfs1(u, v, d + 1);
    sz[v] += sz[u];
  }
  if (!g[v].empty()) iter_swap(g[v].begin(),
↪  max_element(all(g[v]), comp));
}
void dfs2(int v, int rt, int c){
  pos[v] = sz(a);
  a.pb(c);
  root[v] = rt;
  for (int i = 0; i < sz(g[v]); i++){
    auto [u, c] = g[v][i];
    if (!i) dfs2(u, rt, c);
```

```
24        else dfs2(u, u, c);
25    }
26  }
27  int getans(int u, int v){
28    int res = 0;
29    for (; root[u] != root[v]; v = par[root[v]]){
30      if (dep[root[u]] > dep[root[v]]) swap(u, v);
31      res = max(res, rmq(0, 0, n - 1, pos[root[v]], pos[v]));
32    }
33    if (pos[u] > pos[v]) swap(u, v);
34    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
    ↪ fa, int t) {
11    for (auto& ne : g[node])
12      if (ne != fa && !seen[ne] && sz[ne] > t / 2) return
    ↪ find_centroid(ne, node, t);
13    return node;
14  };
15  function<void(int, char)> solve = [&](int node, char cur) {
16    get_size(node, -1); auto c = find_centroid(node, -1,
    ↪ sz[node]);
17    seen[c] = 1, res[c] = cur;
18    for (auto& ne : g[c]) {
19      if (seen[ne]) continue;
20      solve(ne, char(cur + 1)); // we can pass c here to build
    ↪ tree
21    }
22  };
```

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

```
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, false);
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]
    ↪ divides i
17          break;
18        } else {
19          phi[i * prime[j]] = phi[i] * phi[prime[j]]; //prime[j]
    ↪ does not divide i
20        }
21      }
22    }
23  }
```

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
    ↪ solutions
6   template <typename T>
7   int gaussian_elimination(vector<vector<T>> &a, int limit) {
8       if (a.empty() || a[0].empty()) return -1;
9     int h = (int)a.size(), w = (int)a[0].size(), r = 0;
10    for (int c = 0; c < limit; c++) {
11      int id = -1;
12      for (int i = r; i < h; i++) {
13        if (!is_0(a[i][c]) && (id == -1 || abs(a[id][c]) <
    ↪ abs(a[i][c]))) {
14          id = i;
15        }
16      }
17      if (id == -1) continue;
18      if (id > r) {
19        swap(a[r], a[id]);
20        for (int j = c; j < w; j++) a[id][j] = -a[id][j];
21      }
22      vector<int> nonzero;
23      for (int j = c; j < w; j++) {
24        if (!is_0(a[r][j])) nonzero.push_back(j);
25      }
26      T inv_a = 1 / a[r][c];
27      for (int i = r + 1; i < h; i++) {
28        if (is_0(a[i][c])) continue;
29        T coeff = -a[i][c] * inv_a;
30        for (int j : nonzero) a[i][j] += coeff * a[r][j];
31      }
32      ++r;
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 *
    ↪ a[row][j];
42          }
43          break;
44        }
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    template <typename T>
52    pair<int,vector<T>> solve_linear(vector<vector<T>> a, const
   ↪   vector<T> &b, int w) {
53      int h = (int)a.size();
54      for (int i = 0; i < h; i++) a[i].push_back(b[i]);
55      int sol = gaussian_elimination(a, w);
56      if(!sol) return {0, vector<T>()};
57      vector<T> x(w, 0);
58      for (int i = 0; i < h; i++) {
59        for (int j = 0; j < w; j++) {
60          if (!is_0(a[i][j])) {
61            x[j] = a[i][w] / a[i][j];
62            break;
63          }
64        }
65      }
66      return {sol, x};
67    }
```

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
   ↪   & 1) * (n / 2));
6      for (int i = 0; i < n; i++) {
7        if (i < rev[i]) swap(a[i], a[rev[i]]);
8      }
9      ll wn = power(f ? (MOD + 1) / 3 : 3, (MOD - 1) / n);
10     w[0] = 1;
11     for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn % MOD;
12     for (int mid = 1; mid < n; mid *= 2) {
13       for (int i = 0; i < n; i += 2 * mid) {
14         for (int j = 0; j < mid; j++) {
15           ll x = a[i + j], y = a[i + j + mid] * w[n / (2 * mid)
   ↪   * j] % MOD;
16           a[i + j] = (x + y) % MOD, a[i + j + mid] = (x + MOD -
   ↪   y) % MOD;
17         }
18       }
19     }
20     if (f) {
21       ll iv = power(n, MOD - 2);
22       for (auto& x : a) x = x * iv % MOD;
23     }
24   }
25   vector<ll> mul(vector<ll> a, vector<ll> b) {
26     int n = 1, m = (int)a.size() + (int)b.size() - 1;
27     while (n < m) n *= 2;
28     a.resize(n), b.resize(n);
29     ntt(a, 0), ntt(b, 0); // if squaring, you can save one NTT
   ↪   here
30     for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % MOD;
31     ntt(a, 1);
32     a.resize(m);
33     return a;
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; 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
↪   built on the same string.
51    int n = sz(s);
52    h.resize(n - 1);
53    int k = 0;
54    for (int i = 0; i < n; i++){
55      if (c[i] == n){
56        k = 0;
57        continue;
58      }
59      int j = p[c[i]];
60      while (i + k < n && j + k < n && s[i + k] == s[j + k])
↪   k++;
61      h[c[i] - 1] = k;
62      if (k) k--;
63    }
64    /*
65    Then an RMQ Sparse Table can be built on array h
66    to calculate LCP of 2 non-consecutive suffixes.
67    */
68  }
69
70  void buildSparse(){
71    st.build(h);
72  }
73
74  // l and r must be in 0-BASED INDEXATION
75  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).

```cpp
const int S = 26;

// Function converting char to int.
int ctoi(char c){
  return c - 'a';
}

// To add terminal links, use DFS
struct Node{
  vector<int> nxt;
  int link;
  bool terminal;

  Node() {
    nxt.assign(S, -1), link = 0, terminal = 0;
  }
};

vector<Node> trie(1);

// add_string returns the terminal vertex.
int add_string(string& s){
  int v = 0;
  for (auto c : s){
    int cur = ctoi(c);
    if (trie[v].nxt[cur] == -1){
      trie[v].nxt[cur] = sz(trie);
      trie.emplace_back();
    }
    v = trie[v].nxt[cur];
  }
  trie[v].terminal = 1;
  return v;
}

/*
Suffix links are compressed.
This means that:
  If vertex v has a child by letter x, then:
    trie[v].nxt[x] points to that child.
  If vertex v doesn't have such child, then:
    trie[v].nxt[x] points to the suffix link of that child
    if we would actually have it.
*/
void add_links(){
  queue<int> q;
  q.push(0);
  while (!q.empty()){
    auto v = q.front();
    int u = trie[v].link;
    q.pop();
    for (int i = 0; i < S; i++){
      int& ch = trie[v].nxt[i];
      if (ch == -1){
        ch = v? trie[u].nxt[i] : 0;
      }
      else{
        trie[ch].link = v? trie[u].nxt[i] : 0;
        q.push(ch);
      }
    }
  }
}

bool is_terminal(int v){
  return trie[v].terminal;
}

int get_link(int v){
  return trie[v].link;
}

int go(int v, char c){
  return trie[v].nxt[ctoi(c)];
}
```

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.

```cpp
struct line{
  ll k, b;
  ll f(ll x){
    return k * x + b;
  };
};

vector<line> hull;

void add_line(line nl){
  if (!hull.empty() && hull.back().k == nl.k){
    nl.b = min(nl.b, hull.back().b); // Default: minimum. For
  // maximum change "min" to "max".
    hull.pop_back();
  }
  while (sz(hull) > 1){
    auto& l1 = hull.end()[-2], l2 = hull.back();
    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 <=.
    else break;
  }
  hull.pb(nl);
}

ll get(ll x){
  int l = 0, r = sz(hull);
  while (r - l > 1){
    int mid = (l + r) / 2;
    if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid; //
  // Default: minimum. For maximum change the sign to <=.
    else r = mid;
  }
  return hull[l].f(x);
}
```

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()

```cpp
const ll INF = 1e18; // Change the constant!
struct LiChaoTree{
  struct line{
    ll k, b;
    line(){
      k = b = 0;
    };
    line(ll k_, ll b_){
      k = k_, b = b_;
    };
    ll f(ll x){
      return k * x + b;
    };
  };
  int n;
  bool minimum, on_points;
  vector<ll> pts;
  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.
```