

Columbia University: CU Later Team Reference Document

Kevin Yang, Innokentiy Kaurov, Eric Yuang Shao

May 21th 2024

Contents

Templates	2
Ken's template	2
Kevin's template	2
Kevin's Template Extended	2
Geometry	2
Strings	4
Manacher's algorithm	4
Flows	4
$O(N^2M)$, on unit networks $O(N^{1/2}M)$	4
MCMF – maximize flow, then minimize its cost. $O(Fmn)$	5
Graphs	6
Kuhn's algorithm for bipartite matching . . .	6
Hungarian algorithm for Assignment Problem	6
Dijkstra's Algorithm	6
Eulerian Cycle DFS	6
SCC and 2-SAT	6
Finding Bridges	7
Virtual Tree	7
HLD on Edges DFS	7
Centroid Decomposition	7
Math	8
Binary exponentiation	8
Matrix Exponentiation: $O(n^3 \log b)$	8
Extended Euclidean Algorithm	8
Linear Sieve	8
Gaussian Elimination	8
is_prime	9
Berlekamp-Massey	9
Calculating k-th term of a linear recurrence .	9
Partition Function	10
NTT	10
FFT	10
MIT's FFT/NTT, Polynomial mod/log/exp Template	10
Data Structures	13
Fenwick Tree	13
Lazy Propagation SegTree	13
Sparse Table	13

Suffix Array and LCP array	14
Aho Corasick Trie	14
Convex Hull Trick	15
Li-Chao Segment Tree	15
Persistent Segment Tree	15
Miscellaneous	16
Ordered Set	16
Measuring Execution Time	16
Setting Fixed D.P. Precision	16
Common Bugs and General Advice	16

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 const char nl = '\n';
7 #define forn(i, n) for (int i = 0; i < int(n); i++)
8 ll k, n, m, u, v, w, x, y, z;
9 string s, t;
10
11 bool multiTest = 1;
12 void solve(int tt){
13 }
14
15 int main(){
16     ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
17     cout<<fixed<< setprecision(14);
18
19     int t = 1;
20     if (multiTest) cin >> t;
21     forn(ii, t) solve(ii);
22 }
```

Kevin's Template Extended

- to type after the start of the contest

```
1 typedef pair<double, double> pdd;
2 const ld PI = acosl(-1);
3 const ll mod7 = 1e9 + 7;
4 const ll mod9 = 998244353;
5 const ll INF = 2*1024*1024*1023;
6 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
7 #include <ext/pb_ds/assoc_container.hpp>
8 #include <ext/pb_ds/tree_policy.hpp>
9 using namespace __gnu_pbds;
10 template<class T> using ordered_set = tree<T, null_type,
11     ↪ less<T>, rb_tree_tag,
12     ↪ tree_order_statistics_node_update>;
13 vi d4x = {1, 0, -1, 0};
```

```
12 vi d4y = {0, 1, 0, -1};
13 vi d8x = {1, 0, -1, 0, 1, 1, -1, -1};
14 vi d8y = {0, 1, 0, -1, 1, -1, 1, -1};
15 mt19937
```

Geometry

- Basic stuff

```
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_),
8     ↪ id(-1) {}
9     TPoint(const T& x_, const T& y_, const int id_) :
10     ↪ x(x_), y(y_), id(id_) {}
11
12     TPoint operator + (const TPoint& rhs) const {
13         return TPoint(x + rhs.x, y + rhs.y);
14     }
15     TPoint operator - (const TPoint& rhs) const {
16         return TPoint(x - rhs.x, y - rhs.y);
17     }
18     TPoint operator * (const T& rhs) const {
19         return TPoint(x * rhs, y * rhs);
20     }
21     TPoint operator / (const T& rhs) const {
22         return TPoint(x / rhs, y / rhs);
23     }
24     TPoint ort() const {
25         return TPoint(-y, x);
26     }
27     T abs2() const {
28         return x * x + y * y;
29     }
30     T len() const {
31         return sqrtl(abs2());
32     }
33     TPoint unit() const {
34         return TPoint(x, y) / len();
35     }
36 }
37 template<typename T>
38 bool operator< (TPoint<T>& A, TPoint<T>& B){
39     return make_pair(A.x, A.y) < make_pair(B.x, B.y);
40 }
41 template<typename T>
42 bool operator==(TPoint<T>& A, TPoint<T>& B){
43     return abs(A.x - B.x) <= TPoint<T>::eps && abs(A.y -
44     ↪ B.y) <= TPoint<T>::eps;
```

```
45     T a, b, c;
46     TLine() : a(0), b(0), c(0) {}
47     TLine(const T& a_, const T& b_, const T& c_) : a(a_),
48     ↪ b(b_), c(c_) {}
49     TLine(const TPoint<T>& p1, const TPoint<T>& p2){
50         a = p1.y - p2.y;
51         b = p2.x - p1.x;
52         c = -a * p1.x - b * p1.y;
53     }
54     template<typename T>
55     T det(const T& a11, const T& a12, const T& a21, const T&
56     ↪ a22){
57         return a11 * a22 - a12 * a21;
58     }
59     template<typename T>
60     T sq(const T& a){
61         return a * a;
62     }
63     template<typename T>
64     T smul(const TPoint<T>& a, const TPoint<T>& b){
65         return a.x * b.x + a.y * b.y;
66     }
67     template<typename T>
68     T vmul(const TPoint<T>& a, const TPoint<T>& b){
69         return det(a.x, a.y, b.x, b.y);
70     }
71     template<typename T>
72     bool parallel(const TLine<T>& l1, const TLine<T>& l2){
73         return abs(vmul(TPoint<T>(l1.a, l1.b), TPoint<T>(l2.a,
74     ↪ l2.b))) <= TPoint<T>::eps;
75     }
76     template<typename T>
77     bool equivalent(const TLine<T>& l1, const TLine<T>& l2){
78         return parallel(l1, l2) &&
79         ↪ abs(det(l1.b, l1.c, l2.b, l2.c)) <= TPoint<T>::eps &&
80         ↪ abs(det(l1.a, l1.c, l2.a, l2.c)) <= TPoint<T>::eps;
81     }
82 }
```

- Intersection

```
1 template<typename T>
2 TPoint<T> intersection(const TLine<T>& l1, const
3     ↪ TLine<T>& l2){
4     return TPoint<T>{
5         det(-l1.c, l1.b, -l2.c, l2.b) / det(l1.a, l1.b,
6     ↪ l2.a, l2.b),
7         det(l1.a, -l1.c, l2.a, -l2.c) / det(l1.a, l1.b,
8     ↪ l2.a, l2.b)
9     };
10 }
11 template<typename T>
12 int sign(const T& x){
13     if (abs(x) <= TPoint<T>::eps) return 0;
14     return x > 0? +1 : -1;
15 }
```

- Area

```

1  template<typename T>
2  T area(const vector<TPoint<T>>& pts){
3      int n = sz(pts);
4      T ans = 0;
5      for (int i = 0; i < n; i++){
6          ans += vmul(pts[i], pts[(i + 1) % n]);
7      }
8      return abs(ans) / 2;
9  }
10 template<typename T>
11 T dist_pp(const TPoint<T>& a, const TPoint<T>& b){
12     return sqrt(sq(a.x - b.x) + sq(a.y - b.y));
13 }
14 template<typename T>
15 TLine<T> perp_line(const TLine<T>& l, const TPoint<T>&
    ↪ p){
16     T na = -l.b, nb = l.a, nc = - na * p.x - nb * p.y;
17     return TLine<T>(na, nb, nc);
18 }

```

• Projection

```

1  template<typename T>
2  TPoint<T> projection(const TPoint<T>& p, const TLine<T>&
    ↪ l){
3      return intersection(l, perp_line(l, p));
4  }
5  template<typename T>
6  T dist_pl(const TPoint<T>& p, const TLine<T>& l){
7      return dist_pp(p, projection(p, l));
8  }
9  template<typename T>
10 struct TRay{
11     TLine<T> l;
12     TPoint<T> start, dirvec;
13     TRay() : l(), start(), dirvec() {}
14     TRay(const TPoint<T>& p1, const TPoint<T>& p2){
15         l = TLine<T>(p1, p2);
16         start = p1, dirvec = p2 - p1;
17     }
18 };
19 template<typename T>
20 bool is_on_line(const TPoint<T>& p, const TLine<T>& l){
21     return abs(l.a * p.x + l.b * p.y + l.c) <=
    ↪ TPoint<T>::eps;
22 }
23 template<typename T>
24 bool is_on_ray(const TPoint<T>& p, const TRay<T>& r){
25     if (is_on_line(p, r.l)){
26         return sign(smul(r.dirvec, TPoint<T>(p - r.start)))
    ↪ != -1;
27     }
28     else return false;
29 }
30 template<typename T>
31 bool is_on_seg(const TPoint<T>& P, const TPoint<T>& A,
    ↪ const TPoint<T>& B){
32     return is_on_ray(P, TRay<T>(A, B)) && is_on_ray(P,
    ↪ TRay<T>(B, A));

```

```

33 }
34 template<typename T>
35 T dist_pr(const TPoint<T>& P, const TRay<T>& R){
36     auto H = projection(P, R.l);
37     return is_on_ray(H, R)? dist_pp(P, H) : dist_pp(P,
    ↪ R.start);
38 }
39 template<typename T>
40 T dist_ps(const TPoint<T>& P, const TPoint<T>& A, const
    ↪ TPoint<T>& B){
41     auto H = projection(P, TLine<T>(A, B));
42     if (is_on_seg(H, A, B)) return dist_pp(P, H);
43     else return min(dist_pp(P, A), dist_pp(P, B));
44 }

```

• acw

```

1  template<typename T>
2  bool acw(const TPoint<T>& A, const TPoint<T>& B){
3      T mul = vmul(A, B);
4      return mul > 0 || abs(mul) <= TPoint<T>::eps;
5  }

```

• CW

```

1  template<typename T>
2  bool cw(const TPoint<T>& A, const TPoint<T>& B){
3      T mul = vmul(A, B);
4      return mul < 0 || abs(mul) <= TPoint<T>::eps;
5  }

```

• Convex Hull

```

1  template<typename T>
2  vector<TPoint<T>> convex_hull(vector<TPoint<T>> pts){
3      sort(all(pts));
4      pts.erase(unique(all(pts)), pts.end());
5      vector<TPoint<T>> up, down;
6      for (auto p : pts){
7          while (sz(up) > 1 && acw(up.end()[-1] -
    ↪ up.end()[-2], p - up.end()[-2])) up.pop_back();
8          while (sz(down) > 1 && cw(down.end()[-1] -
    ↪ down.end()[-2], p - down.end()[-2]))
    ↪ down.pop_back();
9          up.pb(p), down.pb(p);
10     }
11     for (int i = sz(up) - 2; i >= 1; i--) down.pb(up[i]);
12     return down;
13 }

```

• in_triangle

```

1  template<typename T>
2  bool in_triangle(TPoint<T>& P, TPoint<T>& A, TPoint<T>&
    ↪ B, TPoint<T>& C){
3      if (is_on_seg(P, A, B) || is_on_seg(P, B, C) ||
    ↪ is_on_seg(P, C, A)) return true;
4      return cw(P - A, B - A) == cw(P - B, C - B) &&
    ↪ cw(P - A, B - A) == cw(P - C, A - C);
5  }

```

• prep_convex_poly

```

1  template<typename T>
2  void prep_convex_poly(vector<TPoint<T>>& pts){
3      rotate(pts.begin(), min_element(all(pts)), pts.end());
4  }

```

• in_convex_poly:

```

1  // 0 - Outside, 1 - Exclusively Inside, 2 - On the
    ↪ Border
2  template<typename T>
3  int in_convex_poly(TPoint<T>& p, vector<TPoint<T>>&
    ↪ pts){
4      int n = sz(pts);
5      if (!n) return 0;
6      if (n <= 2) return is_on_seg(p, pts[0], pts.back());
7      int l = 1, r = n - 1;
8      while (r - l > 1){
9          int mid = (l + r) / 2;
10         if (acw(pts[mid] - pts[0], p - pts[0])) l = mid;
11         else r = mid;
12     }
13     if (!in_triangle(p, pts[0], pts[l], pts[l + 1]))
    ↪ return 0;
14     if (is_on_seg(p, pts[l], pts[l + 1]) ||
15         is_on_seg(p, pts[0], pts.back()) ||
16         is_on_seg(p, pts[0], pts[l]))
17         return 2;
18     return 1;
19 }

```

• in_simple_poly

```

1  // 0 - Outside, 1 - Exclusively Inside, 2 - On the
    ↪ Border
2  template<typename T>
3  int in_simple_poly(TPoint<T> p, vector<TPoint<T>>& pts){
4      int n = sz(pts);
5      bool res = 0;
6      for (int i = 0; i < n; i++){
7          auto a = pts[i], b = pts[(i + 1) % n];
8          if (is_on_seg(p, a, b)) return 2;
9          if (((a.y > p.y) - (b.y > p.y)) * vmul(b - p, a - p)
    ↪ > TPoint<T>::eps){
10              res ^= 1;
11          }
12     }
13     return res;
14 }

```

• minkowski_rotate

```

1  template<typename T>
2  void minkowski_rotate(vector<TPoint<T>>& P){
3      int pos = 0;
4      for (int i = 1; i < sz(P); i++){
5          if (abs(P[i].y - P[pos].y) <= TPoint<T>::eps){

```

```

6         if (P[i].x < P[pos].x) pos = i;
7     }
8     else if (P[i].y < P[pos].y) pos = i;
9 }
10 rotate(P.begin(), P.begin() + pos, P.end());
11 }

    • minkowski_sum

1 // P and Q are strictly convex, points given in
  ↪ counterclockwise order
2 template<typename T>
3 vector<TPoint<T>> minkowski_sum(vector<TPoint<T>> P,
  ↪ vector<TPoint<T>> Q){
4     minkowski_rotate(P);
5     minkowski_rotate(Q);
6     P.pb(P[0]);
7     Q.pb(Q[0]);
8     vector<TPoint<T>> ans;
9     int i = 0, j = 0;
10    while (i < sz(P) - 1 || j < sz(Q) - 1){
11        ans.pb(P[i] + Q[j]);
12        T curmul;
13        if (i == sz(P) - 1) curmul = -1;
14        else if (j == sz(Q) - 1) curmul = +1;
15        else curmul = vmul(P[i + 1] - P[i], Q[j + 1] -
  ↪ Q[j]);
16        if (abs(curmul) < TPoint<T>::eps || curmul > 0) i++;
17        if (abs(curmul) < TPoint<T>::eps || curmul < 0) j++;
18    }
19    return ans;
20 }
21 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 }

```

Manacher's algorithm

```

1 string longest_palindrome(string& s) {
2     // init "abc" -> "~$a#b#c$"
3     vector<char> t{'~', '#'};
4     for (char c : s) t.push_back(c), t.push_back('#');
5     t.push_back('$');
6     // manacher
7     int n = t.size(), r = 0, c = 0;
8     vector<int> p(n, 0);
9     for (int i = 1; i < n - 1; i++) {
10        if (i < r + c) p[i] = min(p[2 * c - i], r + c - i);
11        while (t[i + p[i] + 1] == t[i - p[i] - 1]) p[i]++;
12        if (i + p[i] > r + c) r = p[i], c = i;
13    }
14    // s[i] -> p[2 * i + 2] (even), p[2 * i + 2] (odd)
15    // output answer
16    int index = 0;
17    for (int i = 0; i < n; i++)
18        if (p[index] < p[i]) index = i;
19    return s.substr((index - p[index]) / 2, p[index]);
20 }

```

Flows

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

```

1 struct FlowEdge {
2     int v, u;
3     ll cap, flow = 0;
4     FlowEdge(int v, int u, ll cap) : v(v), u(u),
  ↪ cap(cap) {}
5 };
6 struct Dinic {
7     const ll 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, ll 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     ll dfs(int v, ll 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(); cid++) {
47             int id = adj[v][cid];
48             int u = edges[id].u;
49             if (level[v] + 1 != level[u] ||
  ↪ edges[id].cap - edges[id].flow < 1)
50                 continue;
51             ll tr = dfs(u, min(pushed, edges[id].cap -
  ↪ edges[id].flow));
52             if (tr == 0)
53                 continue;
54             edges[id].flow += tr;
55             edges[id ^ 1].flow -= tr;
56             return tr;
57         }
58         return 0;
59     }
60     ll flow() {
61         ll f = 0;
62         while (true) {
63             fill(level.begin(), level.end(), -1);

```

```

64         level[s] = 0;
65         q.push(s);
66         if (!bfs())
67             break;
68         fill(ptr.begin(), ptr.end(), 0);
69         while (ll pushed = dfs(s, flow_inf)) {
70             f += pushed;
71         }
72     }
73     return f;
74 }
75 };
76 // To recover flow through original edges: iterate over
    even 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),
    its(n), pe(n) {}
26
27     int add(int from, int to, T forward_cap, C edge_cost,
    T backward_cap = 0) {
28         assert(0 <= from && from < n && 0 <= to && to < n);
29         assert(forward_cap >= 0 && backward_cap >= 0);
30         int id = static_cast<int>(edges.size());
31         g[from].push_back(id);
32         edges.push_back({from, to, forward_cap, 0,
    edge_cost});
33         g[to].push_back(id + 1);
34         edges.push_back({to, from, backward_cap, 0,
    -edge_cost});
35         return id;

```

```

36     }
37
38     void expath(int st) {
39         fill(d.begin(), d.end(), INF_C);
40         q.clear();
41         fill(its.begin(), its.end(), q.end());
42         its[st] = q.push({pot[st], st});
43         d[st] = 0;
44         while (!q.empty()) {
45             int i = q.top().second;
46             q.pop();
47             its[i] = q.end();
48             for (int id : g[i]) {
49                 const edge &e = edges[id];
50                 int j = e.to;
51                 if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
52                     d[j] = d[i] + e.cost;
53                     pe[j] = id;
54                     if (its[j] == q.end()) {
55                         its[j] = q.push({pot[j] - d[j], j});
56                     } else {
57                         q.modify(its[j], {pot[j] - d[j], j});
58                     }
59                 }
60             }
61         }
62         swap(d, pot);
63     }
64
65     pair<T, C> max_flow(int st, int fin) {
66         T flow = 0;
67         C cost = 0;
68         bool ok = true;
69         for (auto& e : edges) {
70             if (e.c - e.f > eps && e.cost + pot[e.from] -
    pot[e.to] < 0) {
71                 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 <
    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);
166 // To recover flow through original edges: iterate over
167 // 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
4  ↪ FASTER!!!
5  */
6
7  vector<int> g[N]; // Stores edges from left half to
8  ↪ right.
9  bool used[N]; // Stores if vertex from left half is
10 ↪ used.
11 int mt[N]; // For every vertex in right half, stores to
12 ↪ which vertex in left half it's matched (-1 if not
13 ↪ matched).
14
15 bool try_dfs(int v){
16     if (used[v]) return false;
17     used[v] = 1;
18     for (auto u : g[v]){
19         if (mt[u] == -1 || try_dfs(mt[u])){
20             mt[u] = v;
21             return true;
22         }
23     }
24     return false;
25 }
26
27 int main(){
28     // .....
29     for (int i = 1; i <= n2; i++) mt[i] = -1;
30     for (int i = 1; i <= n1; i++) used[i] = 0;
31     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 -
39 ↪ # of edges in matching.
40 // To construct: launch Kuhn-like DFS from unmatched
41 ↪ nodes in the left half.
42 // Independent set = visited nodes in left half +
43 ↪ unvisited in right half.
44 // Finding minimal vertex cover: complement of maximal
45 ↪ 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
2 ↪ the matrix
3 vector<int> u(n+1), v(m+1), p(m+1), way(m+1);
4 for (int i=1; i<=n; ++i) {
5     p[0] = i;
6     int j0 = 0;
7     vector<int> minv (m+1, INF);
8     vector<bool> used (m+1, false);
9     do {
10         used[j0] = true;
11         int i0 = p[j0], delta = INF, j1;
12         for (int j=1; j<=m; ++j)
13             if (!used[j]) {
14                 int cur = A[i0][j]-u[i0]-v[j];
15                 if (cur < minv[j])
16                     minv[j] = cur, way[j] = j0;
17                 if (minv[j] < delta)
18                     delta = minv[j], j1 = j;
19             }
20         for (int j=j0; j<=m; ++j)
21             if (used[j])
22                 u[p[j]] += delta, v[j] -= delta;
23             else
24                 minv[j] -= delta;
25         j0 = j1;
26     } while (p[j0] != 0);
27     do {
28         int j1 = way[j0];
29         p[j0] = p[j1];
30         j0 = j1;

```

```

30     } while (j0);
31 }
32 vector<int> ans (n+1); // ans[i] stores the column
33 ↪ selected for row i
34 for (int j=1; j<=m; ++j)
35     ans[p[j]] = j;
36 int cost = -v[0]; // the total cost of the matching

```

Dijkstra's Algorithm

```

1 priority_queue<pair<ll, ll>, vector<pair<ll, ll>>,
2     ↪ greater<pair<ll, ll>>> q;
3 dist[start] = 0;
4 q.push({0, start});
5 while (!q.empty()){
6     auto [d, v] = q.top();
7     q.pop();
8     if (d != dist[v]) continue;
9     for (auto [u, w] : g[v]){
10         if (dist[u] > dist[v] + w){
11             dist[u] = dist[v] + w;
12             q.push({dist[u], u});
13         }
14     }
15 }

```

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>>&
45     ↪ clauses) {
46     vector<int> ans(n);
47     vector<vector<int>> g(2*n + 1);
48     for(auto [x, y] : clauses) {
49         x = x < 0 ? -x + n : x;
50         y = y < 0 ? -y + n : y;
51         int nx = x <= n ? x + n : x - n;
52         int ny = y <= n ? y + n : y - n;
53         g[nx].push_back(y);
54         g[ny].push_back(x);
55     }
56     int idx[2*n + 1];
57     scc(g, idx);
58     for(int i = 1; i <= n; i++) {
59         if(idx[i] == idx[i + n]) return {0, {}};
60         ans[i - 1] = idx[i + n] < idx[i];
61     }
62     return {1, ans};
63 }

```

Finding Bridges

```

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

```

```

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[{v, u}] = is_bridge[{u, v}] = 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] <
3  ↪ tin[v]});
4  int m = sz(order);
5  for (int i = 1; i < m; i++){
6      order.pb(lca(order[i], order[i - 1]));
7  }
8  sort(all(order), [&](int u, int v){return tin[u] <
9  ↪ tin[v]});
10 order.erase(unique(all(order)), order.end());
11 vector<int> stk{order[0]};
12 for (int i = 1; i < sz(order); i++){
13     int v = order[i];
14     while (tout[stk.back()] < tout[v]) stk.pop_back();
15     int u = stk.back();
16     vg[u].pb({v, dep[v] - dep[u]});
17     stk.pb(v);
18 }

```

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 }

```

```

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]],
33 ↪ pos[v]));
34     }
35     if (pos[u] > pos[v]) swap(u, v);
36     return max(res, rmq(0, 0, n - 1, pos[u] + 1, pos[v]));
37 }

```

Centroid Decomposition

```

1  vector<char> res(n), seen(n), sz(n);
2  function<int(int, int)> get_size = [&](int node, int fa)
3  ↪ {
4      sz[node] = 1;
5      for (auto& ne : g[node]) {
6          if (ne == fa || seen[ne]) continue;
7          sz[node] += get_size(ne, node);
8      }
9      return sz[node];
10 };
11 function<int(int, int, int)> find_centroid = [&](int
12 ↪ node, int fa, int t) {
13     for (auto& ne : g[node])
14         if (ne != fa && !seen[ne] && sz[ne] > t / 2) return
15 ↪ find_centroid(ne, node, t);
16     return node;
17 };
18 function<void(int, char)> solve = [&](int node, char
19 ↪ cur) {
20     get_size(node, -1); auto c = find_centroid(node, -1,
21 ↪ sz[node]);
22     seen[c] = 1, res[c] = cur;
23     for (auto& ne : g[c]) {
24         if (seen[ne]) continue;
25         solve(ne, char(cur + 1)); // we can pass c here to
26 ↪ build tree
27     }
28 };

```


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 }
```

Matrix Exponentiation: $O(n^3 \log b)$

```
1 const int N = 100, MOD = 1e9 + 7;
2
3 struct matrix{
4     ll m[N][N];
5     int n;
6     matrix(){
7         n = N;
8         memset(m, 0, sizeof(m));
9     };
10    matrix(int n){
11        n = n;
12        memset(m, 0, sizeof(m));
13    };
14    matrix(int n_, ll val){
15        n = n_;
16        memset(m, 0, sizeof(m));
17        for (int i = 0; i < n; i++) m[i][i] = val;
18    };
19
20    matrix operator* (matrix oth){
21        matrix res(n);
22        for (int i = 0; i < n; i++){
23            for (int j = 0; j < n; j++){
24                for (int k = 0; k < n; k++){
25                    res.m[i][j] = (res.m[i][j] + m[i][k] *
26                    oth.m[k][j]) % MOD;
27                }
28            }
29            return res;
30        }
31    };
32
33    matrix power(matrix a, ll b){
34        matrix res(a.n, 1);
35        for (; b; a = a * a, b >>= 1){
36            if (b & 1) res = res * a;
37        }
38        return res;
39    }
```

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
20                divide i
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]){
```

```
        prime.push_back(i);
        phi[i] = i - 1; //i is prime
    }
    for (int j = 0; j < prime.size() && i * prime[j] < n; j++){
        is_composite[i * prime[j]] = true;
        if (i % prime[j] == 0){
            phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
            divides i
            break;
        } else {
            phi[i * prime[j]] = phi[i] * phi[prime[j]];
            //prime[j] does not divide i
        }
    }
}
```

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 =>
6 // multiple solutions
7 template <typename T>
8 int gaussian_elimination(vector<vector<T>> &a, int
9 limit) {
10    if (a.empty() || a[0].empty()) return -1;
11    int h = (int)a.size(), w = (int)a[0].size(), r = 0;
12    for (int c = 0; c < limit; c++) {
13        int id = -1;
14        for (int i = r; i < h; i++) {
15            if (!is_0(a[i][c]) && (id == -1 || abs(a[id][c]) <
16            abs(a[i][c]))) {
17                id = i;
18            }
19        }
20        if (id == -1) continue;
21        if (id > r) {
22            swap(a[r], a[id]);
23            for (int j = c; j < w; j++) a[id][j] = -a[id][j];
24        }
25        vector<int> nonzero;
26        for (int j = c; j < w; j++) {
27            if (!is_0(a[r][j])) nonzero.push_back(j);
28        }
29        T inv_a = 1 / a[r][c];
30        for (int i = r + 1; i < h; i++) {
31            if (is_0(a[i][c])) continue;
32            T coeff = -a[i][c] * inv_a;
33            for (int j : nonzero) a[i][j] += coeff * a[r][j];
34        }
35        ++r;
36    }
37    for (int row = h - 1; row >= 0; row--) {
38        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 } // not-free variables: only it on its line
46 for(int i = r; i < h; i++) if(!is_0(a[i][limit]))
47     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,
53     const vector<T> &b, int w) {
54     int h = (int)a.size();
55     for (int i = 0; i < h; i++) a[i].push_back(b[i]);
56     int sol = gaussian_elimination(a, w);
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 }

```

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,
12     19, 23};
13     int s = __builtin_ctzll(n - 1);
14     ll d = (n - 1) >> s;
15     for (auto a : A) {
16         if (a == n) return true;
17         ll x = (ll)power(a, d, n);
18         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 }
30
31 typedef __int128_t i128;
32
33 ll pollard_rho(ll x) {
34     ll s = 0, t = 0, c = rng() % (x - 1) + 1;
35     ll stp = 0, goal = 1, val = 1;
36     for (goal = 1; goal <= 2, s = t, val = 1) {
37         for (stp = 1; stp <= goal; ++stp) {
38             t = ll(((i128)t * t + c) % x);
39             val = ll(((i128)val * abs(t - s) % x);
40             if ((stp % 127) == 0) {
41                 ll d = gcd(val, x);
42                 if (d > 1) return d;
43             }
44         }
45         ll d = gcd(val, x);
46         if (d > 1) return d;
47     }
48 }
49
50 ll get_max_factor(ll _x) {
51     ll max_factor = 0;
52     function<void(ll)> fac = [&](ll x) {
53         if (x <= max_factor || x < 2) return;
54         if (is_prime(x)) {
55             max_factor = max_factor > x ? max_factor : x;
56             return;
57         }
58         ll p = x;
59         while (p >= x) p = pollard_rho(x);
60         while ((x % p) == 0) x /= p;
61         fac(x), fac(p);
62     };
63     fac(_x);
64     return max_factor;
65 }

```

Berlekamp-Massey

- Recovers any n -order linear recurrence relation from the first $2n$ terms of the sequence.
- Input s is the sequence to be analyzed.
- Output c is the shortest sequence c_1, \dots, c_n , such

that

$$s_m = \sum_{i=1}^n c_i \cdot s_{m-i}, \text{ for all } m \geq n.$$

- Be careful since c is returned in 0-based indexation.
- Complexity: $O(N^2)$

```

1 vector<ll> berlekamp_massey(vector<ll> s) {
2     int n = sz(s), l = 0, m = 1;
3     vector<ll> b(n), c(n);
4     ll ldd = b[0] = c[0] = 1;
5     for (int i = 0; i < n; i++, m++) {
6         ll d = s[i];
7         for (int j = 1; j <= l; j++) d = (d + c[j] * s[i -
8     j]) % MOD;
9         if (d == 0) continue;
10        vector<ll> temp = c;
11        ll coef = d * power(ldd, MOD - 2) % MOD;
12        for (int j = m; j < n; j++){
13            c[j] = (c[j] + MOD - coef * b[j - m]) % MOD;
14            if (c[j] < 0) c[j] += MOD;
15        }
16        if (2 * l <= i) {
17            l = i + 1 - l;
18            b = temp;
19            ldd = d;
20            m = 0;
21        }
22    }
23    c.resize(l + 1);
24    c.erase(c.begin());
25    for (ll &x : c)
26        x = (MOD - x) % MOD;
27    return c;
28 }

```

Calculating k-th term of a linear recurrence

- Given the first n terms s_0, s_1, \dots, s_{n-1} and the sequence c_1, c_2, \dots, c_n such that

$$s_m = \sum_{i=1}^n c_i \cdot s_{m-i}, \text{ for all } m \geq n,$$

the function `calc_kth` computes s_k .

- Complexity: $O(n^2 \log k)$

```

vector<ll> poly_mult_mod(vector<ll> p, vector<ll> q,
    vector<ll> &c) {
    vector<ll> ans(sz(p) + sz(q) - 1);
    for (int i = 0; i < sz(p); i++){
        for (int j = 0; j < sz(q); j++){

```

```

5     ans[i + j] = (ans[i + j] + p[i] * q[j]) % MOD;
6 }
7 }
8 int n = sz(ans), m = sz(c);
9 for (int i = n - 1; i >= m; i--){
10     for (int j = 0; j < m; j++){
11         ans[i - 1 - j] = (ans[i - 1 - j] + c[j] * ans[i]) % MOD;
12     }
13 }
14 ans.resize(m);
15 return ans;
16 }

17 ll calc_kth(vector<ll> s, vector<ll> c, ll k){
18     assert(sz(s) >= sz(c)); // size of s can be greater
19     // than c, but not less
20     if (k < sz(s)) return s[k];
21     vector<ll> res{1};
22     for (vector<ll> poly = {0, 1}; k >= 1){
23         poly_mult_mod(poly, poly, c, k >= 1){
24             if (k & 1) res = poly_mult_mod(res, poly, c);
25         }
26         ll ans = 0;
27         for (int i = 0; i < min(sz(res), sz(c)); i++) ans =
28             (ans + s[i] * res[i]) % MOD;
29         return ans;
30 }

```

Partition Function

- Returns number of partitions of n in $O(n^{1.5})$

```

1 int partition(int n) {
2     int dp[n + 1];
3     dp[0] = 1;
4     for (int i = 1; i <= n; i++) {
5         dp[i] = 0;
6         for (int j = 1, r = 1; i - (3 * j * j - j) / 2 >= 0;
7             ++j, r += -1) {
8             dp[i] += dp[i - (3 * j * j - j) / 2] * r;
9             if (i - (3 * j * j + j) / 2 >= 0) dp[i] += dp[i -
10                 (3 * j * j + j) / 2] * r;
11         }
12     }
13     return dp[n];
14 }

```

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) +
6         ((i & 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 *
17                     mid) * j] % MOD;
18                 a[i + j] = (x + y) % MOD, a[i + j + mid] = (x -
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 vector<ll> mul(vector<ll> a, vector<ll> b) {
29     int n = 1, m = (int)a.size() + (int)b.size() - 1;
30     while (n < m) n *= 2;
31     a.resize(n), b.resize(n);
32     ntt(a, 0), ntt(b, 0); // if squaring, you can save one
33     // NTT here
34     for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % MOD;
35     ntt(a, 1);
36     a.resize(m);
37     return a;
38 }

```

FFT

```

1 const ld PI = acos(-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] >
11        1) | ((i & 1) << (bit - 1));
12    auto fft = [&](vector<complex<ld>>& p, int inv) {
13        for (int i = 0; i < len; i++)
14            if (i < rev[i]) swap(p[i], p[rev[i]]);
15        for (int mid = 1; mid < len; mid *= 2) {
16            auto w1 = complex<ld>(cos(PI / mid), (inv ? -1 :
17                1) * sin(PI / mid));
18            for (int i = 0; i < len; i += mid * 2) {
19                auto wk = complex<ld>(1, 0);
20                for (int j = 0; j < mid; j++, wk = wk * w1) {
21                    auto x = p[i + j], y = wk * p[i + j + mid];
22                    p[i + j] = x + y, p[i + j + mid] = x - y;
23                }
24            }
25        }
26    }
27 }

```

```

28 }
29 }
30 }
31 if (inv == 1) {
32     for (int i = 0; i < len; i++)
33         p[i].real(p[i].real() / len);
34 }
35 };
36 fft(a, 0), fft(b, 0);
37 for (int i = 0; i < len; i++) a[i] = a[i] * b[i];
38 fft(a, 1);
39 a.resize(n + m - 1);
40 vector<ld> res(n + m - 1);
41 for (int i = 0; i < n + m - 1; i++) res[i] =
42     a[i].real();
43 return res;
44 };

```

MIT's FFT/NTT, Polynomial mod/log/exp Template

- For integers rounding works if $(|a| + |b|) \max(a, b) < \sim 10^9$, or in theory maybe 10^6
- $\frac{1}{P(x)}$ in $O(n \log n)$, $e^{P(x)}$ in $O(n \log n)$, $\ln(P(x))$ in $O(n \log n)$, $P(x)^k$ in $O(n \log n)$, Evaluates $P(x_1), \dots, P(x_n)$ in $O(n \log^2 n)$, Lagrange Interpolation in $O(n \log^2 n)$

```

1 // use #define FFT 1 to use FFT instead of NTT (default)
2 // Examples:
3 // poly a(n+1); // constructs degree n poly
4 // a[0].v = 10; // assigns constant term a_0 = 10
5 // poly b = exp(a);
6 // poly is vector<num>
7 // for NTT, num stores just one int named v
8 // for FFT, num stores two doubles named x (real), y
9 // (imag)
10
11 #define sz(x) ((int)x.size())
12 #define rep(i, j, k) for (int i = int(j); i < int(k); i++)
13 #define trav(a, x) for (auto& a : x)
14 #define per(i, a, b) for (int i = (b)-1; i >= (a); --i)
15 using ll = long long;
16 using vi = vector<int>;
17
18 namespace fft {
19     #if FFT
20     // FFT
21     using dbl = double;
22     struct num {
23         dbl x, y;
24         num(dbl x_ = 0, dbl y_ = 0): x(x_), y(y_) {}
25     };
26 }

```



```

186     fb[i] = fa[i] * conj(g1);
187     fa[i] = fa[i] * conj(g0);
188 }
189 fft(fa, n);
190 fft(fb, n);
191 vi r(s);
192 rep(i, 0, s) r[i] =
193     int((ll(fa[i].x + 0.5) + (ll(fa[i].y + 0.5) % m < 15) +
194         (ll(fb[i].x + 0.5) % m < 15) +
195         (ll(fb[i].y + 0.5) % m < 30)) %
196         m);
197     return r;
198 }
199 #endif
200 } // namespace fft
201 // For multiply_mod, use num = modnum, poly =
202     vector<num>
203 using fft::num;
204 using poly = fft::vn;
205 using fft::multiply;
206 using fft::inverse;
207
208 poly& operator+=(poly& a, const poly& b) {
209     if (sz(a) < sz(b)) a.resize(b.size());
210     rep(i, 0, sz(b)) a[i] = a[i] + b[i];
211     return a;
212 }
213
214 poly operator+(const poly& a, const poly& b) {
215     poly r = a;
216     r += b;
217     return r;
218 }
219
220 poly& operator-=(poly& a, const poly& b) {
221     if (sz(a) < sz(b)) a.resize(b.size());
222     rep(i, 0, sz(b)) a[i] = a[i] - b[i];
223     return a;
224 }
225
226 poly operator-(const poly& a, const poly& b) {
227     poly r = a;
228     r -= b;
229     return r;
230 }
231
232 poly& operator*(const poly& a, const poly& b) {
233     return multiply(a, b);
234 }
235
236 poly& operator*=(poly& a, const poly& b) { return a = a * b; }
237
238 poly& operator*=(poly& a, const num& b) { // Optional
239     trav(x, a) x = x * b;
240     return a;
241 }
242
243 poly& operator*(const poly& a, const num& b) {
244     poly r = a;
245     r *= b;
246     return r;
247 }
248
249 // Polynomial floor division; no leading 0's please
250 poly operator/(poly a, poly b) {
251     if (sz(a) < sz(b)) return {};
252     int s = sz(a) - sz(b) + 1;
253     reverse(a.begin(), a.end());
254     reverse(b.begin(), b.end());
255     a.resize(s);
256     b.resize(s);
257     a = a * inverse(move(b));
258     a.resize(s);
259     reverse(a.begin(), a.end());
260     return a;
261 }
262
263 poly& operator/=(poly& a, const poly& b) { return a = a / b; }
264
265 poly& operator%=(poly& a, const poly& b) {
266     if (sz(a) >= sz(b)) {
267         poly c = (a / b) * b;
268         a.resize(sz(b) - 1);
269         rep(i, 0, sz(a)) a[i] = a[i] - c[i];
270     }
271     return a;
272 }
273
274 poly operator%(const poly& a, const poly& b) {
275     poly r = a;
276     r %= b;
277     return r;
278 }
279
280 // Log/exp/pow
281 poly deriv(const poly& a) {
282     if (a.empty()) return {};
283     poly b(sz(a) - 1);
284     rep(i, 1, sz(a)) b[i - 1] = a[i] * i;
285     return b;
286 }
287
288 poly integ(const poly& a) {
289     poly b(sz(a) + 1);
290     b[1] = 1; // mod p
291     rep(i, 2, sz(b)) b[i] =
292         b[fft::mod % i] * (-fft::mod / i); // mod p
293     rep(i, 1, sz(b)) b[i] = a[i - 1] * b[i]; // mod p
294     //rep(i, 1, sz(b)) b[i] = a[i - 1] * inv(num(i)); // else
295     return b;
296 }
297
298 poly log(const poly& a) { // MUST have a[0] == 1
299     poly b = integ(deriv(a) * inverse(a));
300     b.resize(a.size());
301     return b;
302 }
303
304 poly exp(const poly& a) { // MUST have a[0] == 0
305     poly b(1, num(1));
306     if (a.empty()) return b;
307     while (sz(b) < sz(a)) {
308         int n = min(sz(b) * 2, sz(a));
309         b.resize(n);
310         poly v = poly(a.begin(), a.begin() + n) - log(b);
311         v[0] = v[0] + num(1);
312         b *= v;
313     }
314 }
315
316 b.resize(n);
317 }
318 return b;
319 }
320
321 poly pow(const poly& a, int m) { // m >= 0
322     poly b(a.size());
323     if (!m) {
324         b[0] = 1;
325         return b;
326     }
327     int p = 0;
328     while (p < sz(a) && a[p].v == 0) ++p;
329     if (ll * m * p >= sz(a)) return b;
330     num mu = pow(a[p], m), di = inv(a[p]);
331     poly c(sz(a) - m * p);
332     rep(i, 0, sz(c)) c[i] = a[i + p] * di;
333     c = log(c);
334     trav(v, c) v = v * m;
335     c = exp(c);
336     rep(i, 0, sz(c)) b[i + m * p] = c[i] * mu;
337     return b;
338 }
339
340 // Multipoint evaluation/interpolation
341
342 vector<num> eval(const poly& a, const vector<num>& x) {
343     int n = sz(x);
344     if (!n) return {};
345     vector<poly> up(2 * n);
346     rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
347     per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
348     vector<poly> down(2 * n);
349     down[1] = a % up[1];
350     rep(i, 2, 2 * n) down[i] = down[i / 2] % up[i];
351     vector<num> y(n);
352     rep(i, 0, n) y[i] = down[i + n][0];
353     return y;
354 }
355
356 poly interp(const vector<num>& x, const vector<num>& y)
357     {
358     int n = sz(x);
359     assert(n);
360     vector<poly> up(n * 2);
361     rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
362     per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
363     vector<num> a = eval(deriv(up[1]), x);
364     vector<poly> down(2 * n);
365     rep(i, 0, n) down[i + n] = poly({y[i] * inv(a[i])});
366     per(i, 1, n) down[i] =
367         down[i * 2] * up[i * 2 + 1] + down[i * 2 + 1] * up[i
368         * 2];
369     return down[1];
370 }

```

Data Structures

Fenwick Tree

```
1 ll sum(int r) {
2     ll ret = 0;
3     for (; r >= 0; r = (r & r + 1) - 1) ret += bit[r];
4     return ret;
5 }
6 void add(int idx, ll delta) {
7     for (; idx < n; idx |= idx + 1) bit[idx] += delta;
8 }
```

Lazy Propagation SegTree

```
1 // Clear: clear() or build()
2 const int N = 2e5 + 10; // Change the constant!
3 template<typename T>
4 struct LazySegTree{
5     T t[4 * N];
6     T lazy[4 * N];
7     int n;
8
9     // Change these functions, default return, and lazy
10    ↪ mark.
11    T default_return = 0, lazy_mark =
12    ↪ numeric_limits<T>::min();
13    // Lazy mark is how the algorithm will identify that
14    ↪ no propagation is needed.
15    function<T(T, T)> f = [&] (T a, T b){
16        return a + b;
17    };
18    // f_on_seg calculates the function f, knowing the
19    ↪ lazy value on segment,
20    // segment's size and the previous value.
21    // The default is segment modification for RSQ. For
22    ↪ increments change to:
23    // return cur_seg_val + seg_size * lazy_val;
24    // For RMQ. Modification: return lazy_val;
25    ↪ Increments: return cur_seg_val + lazy_val;
26    function<T(T, int, T)> f_on_seg = [&] (T cur_seg_val,
27    ↪ int seg_size, T lazy_val){
28        return cur_seg_val + lazy_val * seg_size;
29    };
30    // upd_lazy updates the value to be propagated to
31    ↪ child segments.
32    // Default: modification. For increments change to:
33    // lazy[v] = (lazy[v] == lazy_mark? val : lazy[v]
34    ↪ + val);
35    function<void(int, T)> upd_lazy = [&] (int v, T val){
36        lazy[v] = val;
37    };
38    // Tip: for "get element on single index" queries, use
39    ↪ max() on segment: no overflows.
40
41    LazySegTree(int n_) : n(n_) {
42        clear(n);
43    }
```

```
33 }
34
35 void build(int v, int tl, int tr, vector<T>& a){
36     if (tl == tr) {
37         t[v] = a[tl];
38         return;
39     }
40     int tm = (tl + tr) / 2;
41     // left child: [tl, tm]
42     // right child: [tm + 1, tr]
43     build(2 * v + 1, tl, tm, a);
44     build(2 * v + 2, tm + 1, tr, a);
45     t[v] = f(t[2 * v + 1], t[2 * v + 2]);
46 }
47
48 LazySegTree(vector<T>& a){
49     build(a);
50 }
51
52 void push(int v, int tl, int tr){
53     if (lazy[v] == lazy_mark) return;
54     int tm = (tl + tr) / 2;
55     t[2 * v + 1] = f_on_seg(t[2 * v + 1], tm - tl + 1,
56    ↪ lazy[v]);
57     t[2 * v + 2] = f_on_seg(t[2 * v + 2], tr - tm,
58    ↪ lazy[v]);
59     upd_lazy(2 * v + 1, lazy[v]), upd_lazy(2 * v + 2,
60    ↪ lazy[v]);
61     lazy[v] = lazy_mark;
62 }
63
64 void modify(int v, int tl, int tr, int l, int r, T
65    ↪ val){
66     if (l > r) return;
67     if (tl == l && tr == r){
68         t[v] = f_on_seg(t[v], tr - tl + 1, val);
69         upd_lazy(v, val);
70         return;
71     }
72     push(v, tl, tr);
73     int tm = (tl + tr) / 2;
74     modify(2 * v + 1, tl, tm, l, min(r, tm), val);
75     modify(2 * v + 2, tm + 1, tr, max(l, tm + 1), r,
76    ↪ val);
77     t[v] = f(t[2 * v + 1], t[2 * v + 2]);
78 }
79
80 T query(int v, int tl, int tr, int l, int r) {
81     if (l > r) return default_return;
82     if (tl == l && tr == r) return t[v];
83     push(v, tl, tr);
84     int tm = (tl + tr) / 2;
85     return f(
86         query(2 * v + 1, tl, tm, l, min(r, tm)),
87         query(2 * v + 2, tm + 1, tr, max(l, tm + 1), r)
88     );
89 }
```

```
86 void modify(int l, int r, T val){
87     modify(0, 0, n - 1, l, r, val);
88 }
89
90 T query(int l, int r){
91     return query(0, 0, n - 1, l, r);
92 }
93
94 T get(int pos){
95     return query(pos, pos);
96 }
97
98 // Change clear() function to t.clear() if using
99 ↪ unordered_map for SegTree!!!
100 void clear(int n_){
101     n = n_;
102     for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
103    ↪ lazy_mark;
104 }
105
106 void build(vector<T>& a){
107     n = sz(a);
108     clear(n);
109     build(0, 0, n - 1, a);
110 };
```

Sparse Table

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

```

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:
13         ↪ minimum. For maximum change "min" to "max".
14         hull.pop_back();
15     }
16     while (sz(hull) > 1){
17         auto& l1 = hull.end()[-2], l2 = hull.back();
18         if ((nl.b - l1.b) * (l2.k - nl.k) >= (nl.b - l2.b) *
19         ↪ (l1.k - nl.k)) hull.pop_back(); // Default:
20         ↪ decreasing gradient k. For increasing k change the
21         ↪ sign to <=.
22         else break;
23     }
24     hull.pb(nl);
25 }
26
27 ll get(ll x){
28     int l = 0, r = sz(hull);
29     while (r - l > 1){
30         int mid = (l + r) / 2;
31         if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid;
32         ↪ // Default: minimum. For maximum change the sign to
33         ↪ <=.
34         else r = mid;
35     }
36 }

```

```

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()

```

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;

    void clear(){
        for (auto& l : t) l.k = 0, l.b = minimum? INF :
        ↪ -INF;
    }

    LiChaoTree(int n_, bool min_){ // This is a default
        ↪ constructor for numbers in range [0, n - 1].
        n = n_, minimum = min_, on_points = false;
        t.resize(4 * n);
        clear();
    };

    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.
        pts = pts_, minimum = min_;
        sort(all(pts));
        pts.erase(unique(all(pts)), pts.end());
        on_points = true;
        n = sz(pts);
        t.resize(4 * n);
        clear();
    };

    void add_line(int v, int l, int r, line nl){
        // Adding on segment [l, r)

```

```

int m = (l + r) / 2;
ll lval = on_points? pts[l] : 1, mval = on_points?
↪ pts[m] : m;
if ((minimum && nl.f(mval) < t[v].f(mval)) ||
↪ (!minimum && nl.f(mval) > t[v].f(mval))) swap(t[v],
↪ nl);
if (r - l == 1) return;
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);
else add_line(2 * v + 2, m, r, nl);
}

ll get(int v, int l, int r, int x){
    int m = (l + r) / 2;
    if (r - l == 1) return t[v].f(on_points? pts[x] :
    ↪ x);
    else{
        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));
        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));
    }
}

void add_line(ll k, ll b){
    add_line(0, 0, n, line(k, b));
}

ll get(ll x){
    return get(0, 0, n, on_points? lower_bound(all(pts),
    ↪ x) - pts.begin() : x);
}; // Always pass the actual value of x, even if LCT
↪ is on points.
}

```

Persistent Segment Tree

- for RSQ

```

struct Node {
    ll val;
    Node *l, *r;

    Node(ll x) : val(x), l(nullptr), r(nullptr) {}
    Node(Node *ll, Node *rr) {
        l = ll, r = rr;
        val = 0;
        if (l) val += l->val;
        if (r) val += r->val;
    }
    Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
};

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!