

University of Tartu ICPC Team Notebook  
(2018-2019) March 13, 2019

- 1 crc.sh
- 2 gcc ordered set
- 3 Triangle centers
- 4 2D geometry
- 5 3D geometry
- 6 Seg-Seg intersection, halfplane intersection area
- 7 Convex polygon algorithms
- 8 Delaunay triangulation  $\mathcal{O}(n \log n)$
- 9 Aho Corasick  $\mathcal{O}(|\alpha| \sum \text{len})$
- 10 Suffix automaton and tree  $\mathcal{O}((n + q) \log(|\alpha|))$
- 11 Dinic
- 12 Min Cost Max Flow with Cycle Cancelling  $\mathcal{O}(\text{cap}^2 \cdot n \log(n))$
- 13 DMST  $\mathcal{O}(E \log V)$
- 14 Bridges  $\mathcal{O}(n)$
- 15 2-Sat  $\mathcal{O}(n)$  and SCC  $\mathcal{O}(n)$
- 16 Generic persistent compressed lazy segment tree
- 17 Templatized HLD  $\mathcal{O}(M(n) \log n)$  per query
- 18 Templatized multi dimensional BIT  $\mathcal{O}(\log(n)^{\text{dim}})$  per query
- 19 Treap  $\mathcal{O}(\log n)$  per query
- 20 Radixsort 50M 64 bit integers as single array in  $\mathcal{O}(n \log n)$
- 21 FFT 5M length/sec

	22	Fast mod mult, Rabbin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$	21
1	23	Symmetric Submodular Functions; Queyranne's algorithm	23
1	24	Berlekamp-Massey $O(\mathcal{L}N)$	23
2			
3	1	alias g++='g++ -g -Wall -Wshadow -DCDEBBUG' #.basrc	
3	2	alias a='setxkbmap us -option'	
3	3	alias m='setxkbmap us -option caps:escape'	
3	4	alias ma='setxkbmap us -variant dvp -option caps:escape'	
3	5	#settings	
4	6	gsettings set	
4	7	→ org.compiz.core:/org/compiz/profiles/Default/plugins/core/ hsize 4	
4	8	gsettings set org.gnome.desktop.wm.preferences focus-mode 'slippy'	
4	9	set si cin #.vimrc	
4	10	set ts=4 sw=4 noet	
4	11	set cb=unnamed	
5	12	(global-set-key (kbd "C-x <next>") 'other-window) #.emacs	
5	13	(global-set-key (kbd "C-x <prior>") 'previous-multiframe-window)	
5	14	(global-set-key (kbd "C-M-z") 'ansi-term)	
5	15	(global-linum-mode 1)	
5	16	(column-number-mode 1)	
5	17	(show-paren-mode 1)	
5	18	(setq-default indent-tabs-mode nil)	
5	19	valgrind --vgdb-error=0 ./a <inp & #valgrind	
5	20	gdb a	
6		target remote   vgdb	
7	1	crc.sh	
7	1	#!/bin/env bash	
7	2	for j in `seq 1 1 200` ; do	
7	3	sed '/^\$\s*\$/.d' \$1   head -\$j   tr -d '[:space:]'   cksum   cut -f1	
7	4	→ -d ' '   tail -c 5 #whistespaces don't matter.	
7	5	done #there shouldn't be any COMMENTS.	
7	6	#copy lines being checked to separate file.	
7	6	# \$ ./crc.sh tmp.cpp   grep XXXX	
8	2	gcc ordered set	
8	1	#define DEBUG(...) cerr << __VA_ARGS__ << endl;	
8	2	#ifndef CDEBUG	
8	3	#undef DEBUG	
8	4	#define DEBUG(...) ((void)0);	
8	5	#define NDEBUG	
8	6	#endif	
8	7	#define ran(i, a, b) for (auto i = (a); i < (b); i++)	
8	8	#include<bits/stdc++.h>	
8	9	typedef long long ll	
8	10	typedef long double ld;	
8	11	using namespace std; #include<ext/pb_ds/assoc_container.hpp>	#1736

```

12 #include<ext/pb_ds/tree_policy.hpp>
13 using namespace __gnu_pbds
14 template <typename T>
15 using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
16   tree_order_statistics_node_update>;
17 intmain{
18   ordered_set<int> cur
19   cur.insert(1);
20   cur.insert(3);
21   cout << cur.order_of_key(2)
22     << endl;// the number of elements in the set less than 2
23   cout << *cur.find_by_order(0
24     << endl;// the 0-th smallest number in the set(0-based)
25   cout << *cur.find_by_order(1)
26     << endl;// the 1-th smallest number in the set(0-based)

```

#5119

#3802

#0578

---

### 3 Triangle centers

```

1 const double min_delta = 1e-13;
2 const double coord_max = 1e6;
3 typedef complex<double> point;
4 point A, B, C;// vertixes of the triangle
5 boolcollinear(
6   double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
7   if (min_diff < coord_max * min_delta) return true;
8   point sp = (B - A) / (C - A);
9   double ang = M_PI / 2 - abs(abs(arg(sp)) - M_PI / 2);
10  return ang < min_delta;// positive angle with the real line
11  #8446
12 pointcircum_center(
13   if (collinear()) return point(NAN, NAN);
14   // squared lengths of sides
15   double a2 = norm(B - C);
16   double b2 = norm(A - C);
17   double c2 = norm(A - B)
18   // barycentric coordinates of the circumcenter
19   double c_A = a2 * (b2 + c2 - a2);// sin(2 * alpha) works also
20   double c_B = b2 * (a2 + c2 - b2);
21   double c_C = c2 * (a2 + b2 - c2);
22   double sum = c_A + c_B + c_C;
23   c_A /= sum
24   c_B /= sum;
25   c_C /= sum;
26   return c_A * A + c_B * B + c_C * C;// cartesian
27
28 pointcentroid({// center of mass
29   return (A + B + C) / 3.0;
30 }
31 point ortho_center() {// euler line
32   point O = circum_center()
33   return O + 3.0 * (centroid() - O);
34 }
35 pointnine_point_circle_center({// euler line

```

%6856

#3895

```

36   point O = circum_center();
37   return O + 1.5 * (centroid() - O)
38 }
39 pointin_center({
40   if (collinear()) return point(NAN, NAN);
41   double a = abs(B - C);// side lenghts
42   double b = abs(A - C);
43   double c = abs(A - B)
44   // trilinear coordinates are (1,1,1)
45   double sum = a + b + c;
46   a /= sum;
47   b /= sum;
48   c /= sum;
49   return a * A + b * B + c * C;// cartesian

```

#8193

%3031

#5954

## 4 2D geometry

Define  $\text{orient}(A, B, C) = \overline{AB} \times \overline{AC}$ . CCW iff  $> 0$ . Define  $\text{perp}((a, b)) = (-b, a)$ . The vectors are orthogonal.

For line  $ax + by = c$  def  $\bar{v} = (-b, a)$ .

Line through  $P$  and  $Q$  has  $\bar{v} = \overline{PQ}$  and  $c = \bar{v} \times P$ .  $\text{side}_l(P) = \bar{v}_l \times P - c_l$  sign determines which side  $P$  is on from  $l$ .

$\text{dist}_l(P) = \text{side}_l(P)/\|\bar{v}_l\|$  squared is integer.

Sorting points along a line: comparator is  $\bar{v} \cdot A < \bar{v} \cdot B$ .

Translating line by  $\bar{t}$ : new line has  $c' = c + \bar{v} \times \bar{t}$ .

Line intersection: is  $(c_l \bar{v}_m - c_m \bar{v}_l)/(\bar{v}_l \times \bar{v}_m)$ .

Project  $P$  onto  $l$ : is  $P - \text{perp}(v) \text{side}_l(P)/\|v\|^2$ .

Angle bisectors:  $\bar{v} = \bar{v}_l/\|\bar{v}_l\| + \bar{v}_m/\|\bar{v}_m\|$

$c = c_l/\|\bar{v}_l\| + c_m/\|\bar{v}_m\|$ .

$P$  is on segment  $AB$  iff  $\text{orient}(A, B, P) = 0$  and  $\overline{PA} \cdot \overline{PB} \leq 0$ .

Proper intersection of  $AB$  and  $CD$  exists iff  $\text{orient}(C, D, A)$  and  $\text{orient}(C, D, B)$  have opp. signs and  $\text{orient}(A, B, C)$  and  $\text{orient}(A, B, D)$  have opp. signs. Coordinates:

$$\frac{A \text{orient}(C, D, B) - B \text{orient}(C, D, A)}{\text{orient}(C, D, B) - \text{orient}(C, D, A)}.$$

Circumcircle center:

```
pt circumCenter(pt a, pt b, pt c) {
    b = b-a, c = c-a; // consider coordinates relative to A
    assert(cross(b,c) != 0); // no circumcircle if A,B,C aligned
    return a + perp(b*sq(c) - c*sq(b))/cross(b,c)/2;
```

Circle-line intersect:

```
int circleLine(pt o, double r, line l, pair<pt, pt> &out) {
    double h2 = r*r - l.sqDist(o);
    if (h2 >= 0) { // the line touches the circle
        pt p = l.proj(o); // point P
        pt h = l.v*sqrt(h2)/abs(l.v); // vector parallel to l, of len h
        out = {p-h, p+h};
    }
    return 1 + sgn(h2);
```

Circle-circle intersect:

```
int circleCircle(pt o1, double r1, pt o2, double r2, pair<pt, pt> &out) {
    pt d=o2-o1; double d2=sq(d);
```

```
if (d2 == 0) {assert(r1 != r2); return 0;} // concentric circles
double pd = (d2 + r1*r1 - r2*r2)/2; // = |0_1P| * d
double h2 = r1*r1 - pd*pd/d2; // = h^2
if (h2 >= 0) {
    pt p = o1 + d*pd/d2, h = perp(d)*sqrt(h2/d2);
    ;
    out = {p-h, p+h};}
return 1 + sgn(h2);
```

Tangent lines:

```
int tangents(pt o1, double r1, pt o2, double r2,
    bool inner, vector<pair<pt, pt>> &out) {
    if (inner) r2 = -r2;
    pt d = o2-o1;
    double dr = r1-r2, d2 = sq(d), h2 = d2-dr*dr;
    if (d2 == 0 || h2 < 0) {assert(h2 != 0);
        return 0;}
    for (double sign : {-1,1}) {
        pt v = (d*dr + perp(d)*sqrt(h2)*sign)/d2;
        out.push_back({o1 + v*r1, o2 + v*r2});}
    return 1 + (h2 > 0);
```

## 5 3D geometry

$\text{orient}(P, Q, R, S) = (\overline{PQ} \times \overline{PR}) \cdot \overline{PS}$ .

$S$  above  $PQR$  iff  $> 0$ .

For plane  $ax + by + cz = d$  def  $\bar{n} = (a, b, c)$ .

Line with normal  $\bar{n}$  through point  $P$  has  $d = \bar{n} \cdot P$ .

$\text{side}_\Pi(P) = \bar{n} \cdot P - d$  sign determines side from  $\Pi$ .

$\text{dist}_\Pi(P) = \text{side}_\Pi(P)/\|\bar{n}\|$ .

Translating plane by  $\bar{t}$  makes  $d' = d + \bar{n} \cdot \bar{t}$ .

Plane-plane intersection of has direction  $\bar{n}_1 \times \bar{n}_2$  and goes through  $((d_1 \bar{n}_2 - d_2 \bar{n}_1) \times \bar{d})/\|\bar{d}\|^2$ .

Line-line distance:

```
double dist(line3d l1, line3d l2) {
    p3 n = l1.d*l2.d;
    if (n == zero) // parallel
        return l1.dist(l2.o);
    return abs((l2.o-l1.o)|n)/abs(n);
```

Spherical to Cartesian:

$(r \cos \varphi \cos \lambda, r \cos \varphi \sin \lambda, r \sin \varphi)$ .

Sphere-line intersection:

```
int sphereLine(p3 o, double r, line3d l, pair<p3, p3> &out) {
    double h2 = r*r - l.sqDist(o);
    if (h2 < 0) return 0; // the line doesn't touch the sphere
    p3 p = l.proj(o); // point P
    p3 h = l.d*sqrt(h2)/abs(l.d); // vector parallel to l, of length h
    out = {p-h, p+h};
```

```
return 1 + (h2 > 0);
```

Great-circle distance between points  $A$  and  $B$  is  $r\angle AOB$ .

Spherical segment intersection:

```
bool properInter(p3 a, p3 b, p3 c, p3 d, p3 &out)
    ) {
    p3 ab = a*b, cd = c*d; // normals of planes OAB and OCD
    int oa = sgn(cd|a),
        ob = sgn(cd|b),
        oc = sgn(ab|c),
        od = sgn(ab|d);
    out = ab*cd*od; // four multiplications => careful with overflow !
    return (oa != ob && oc != od && oa != oc);
}
bool onSphSegment(p3 a, p3 b, p3 p) {
    p3 n = a*b;
    if (n == zero)
        return a*p == zero && (a|p) > 0;
    return (n|p) == 0 && (n|a*p) >= 0 && (n|b*p) <= 0;
}
struct directionSet : vector<p3> {
    using vector::vector; // import constructors
    void insert(p3 p) {
        for (p3 q : *this) if (p*q == zero) return;
        push_back(p);
    }
};
directionSet intersSph(p3 a, p3 b, p3 c, p3 d) {
    assert(validSegment(a, b) && validSegment(c, d));
    p3 out;
    if (properInter(a, b, c, d, out)) return {out};
    directionSet s;
    if (onSphSegment(c, d, a)) s.insert(a);
    if (onSphSegment(c, d, b)) s.insert(b);
    if (onSphSegment(a, b, c)) s.insert(c);
    if (onSphSegment(a, b, d)) s.insert(d);
    return s;
}
```

Angle between spherical segments  $AB$  and  $AC$  is angle between  $A \times B$  and  $A \times C$ .

Oriented angle: subtract from  $2\pi$  if mixed product is negative.

Area of a spherical polygon:

$$r^2[\text{sum of interior angles} - (n-2)\pi].$$

## 6 Seg-Seg intersection, halfplane intersection area

```

1 struct Seg {
2     Vec a, b;
3     Vecd({ return b - a; })
4 };
5 Vecintersection(Seg l, Seg r
6     Vec dl = l.d(), dr = r.d();
7     if (cross(dl, dr) == 0) return {nanl""}, nanl""});
8     double h = cross(dr, l.a - r.a) / len(dr);
9     double dh = cross(dr, dl) / len(dr);
10    return l.a + dl * (h / -dh)
11 } // Returns the area bounded by halfplanes
12 double calc_area(const vector<Seg>& lines{
13     double lb = -HUGE_VAL, ub = HUGE_VAL;
14     vector<Seg> slines[2];
15     for (auto line : lines)
16         if (line.b.y == line.a.y) {
17             if (line.a.x < line.b.x) {
18                 lb = max(lb, line.a.y);
19             } else {
20                 ub = min(ub, line.a.y)
21             }
22         } else if (line.a.y < line.b.y) {
23             slines[1].push_back(line);
24         } else {
25             slines[0].push_back({line.b, line.a})
26         }
27     }
28     ran(i, 0, 2) {
29         sort(slines[i].begin(), slines[i].end(), [&](Seg l, Seg r) {
30             if (cross(l.d(), r.d()) == 0) #4919
31                 return normal(l.d()) * l.a > normal(r.d()) * r. ;
32             return (1 - 2 * i) * cross(l.d(), r.d()) < 0;
33         });
34     }
35 // Now find the application area of the lines and clean up redundant
36 // ones
37 vector<double> ap_s[2]
38 ran(side, 0, 2) { #9949
39     vector<double>& apply = ap_s[side];
40     vector<Seg> clines;
41     for (auto line : slines[side]) {
42         while (clines.size() > 0)
43             Seg other = clines.back();
44             if (cross(line.d(), other.d()) != 0) {
45                 double start = intersection(line, other).y;
46                 if (start > apply.back()) break;
47
48                 clines.pop_back();
49                 apply.pop_back();
50             }

```

#6327

#8893

#1804

#6288

#3607

#4919

#9949

#3099

#7856

```

51     if (clines.size() == 0) {
52         apply.push_back(-HUGE_VAL) #0868
53     } else {
54         apply.push_back(intersection(line, clines.back()).y);
55     }
56     clines.push_back(line);
57
58     slines[side] = clines; #8545
59 }
60 ap_s[0].push_back(HUGE_VALL);
61 ap_s[1].push_back(HUGE_VALL);
62 double result = 0 #3234
63 {
64     double lb = -HUGE_VALL, ub; #4531
65     for (int i = 0, j = 0;
66         i < (int)slines[0].size() && j < (int)slines[1].size();
67         lb = ub)
68         ub = min(ap_s[0][i + 1], ap_s[1][j + 1]);
69     double alb = lb, aub = ub;
70     Seg l[2] = {slines[0][i], slines[1][j]};
71     if (cross(l[1].d(), l[0].d()) > 0) { #2627
72         alb = max(alb, intersection(l[0], l[1]).y);
73     } else if (cross(l[1].d(), l[0].d()) < 0)
74         aub = min(aub, intersection(l[0], l[1]).y);
75
76     alb = max(alb, lb);
77     aub = min(aub, ub);
78     aub = max(aub, alb) #8493
79     ran(k, 0, 2) {
80         double x1 =
81             l[0].a.x + (alb - l[0].a.y) / l[0].d().y * l[0].d().x;
82         double x2 =
83             l[0].a.x + (aub - l[0].a.y) / l[0].d().y * l[0].d().x;
84         result += (-1 + 2 * k) * (aub - alb) * (x1 + x2) / 2; #9267
85
86         if (ap_s[0][i + 1] < ap_s[1][j + 1]) {
87             i++;
88         } else { #3074
89             j++;
90         }
91     }
92 }
93 return result;

```

## 7 Convex polygon algorithms

```

1 typedef pair<int, int> Vec;
2 typedef pair<Vec, Vec> Seg;
3 typedef vector<Seg>::iterator SegIt; #define F first
4 #define S second
5 #define MP(x, y) make_pair(x, y)
6 lldot(Vec &v1, Vec &v2){ return (ll)v1.F * v2.F + (ll)v1.S * v2.S; }
7 llcross(Vec &v1, Vec &v2) #6913

```

```

8   return (l1)v1.F * v2.S - (l1)v2.F * v1.S;
9 }
10 lldist_sq(Vec &p1, Vec &p2{
11   return (l1)(p2.F - p1.F) * (p2.F - p1.F) +
12     (l1)(p2.S - p1.S) * (p2.S - p1.S);
13
14 struct Hull {
15   vector<Seg> hull;
16   SegIt up_beg;
17   template <typename It>
18   void extend(It beg, It end) { // O(n)
19     vector<Vec> r
20     for (auto it = beg; it != end; ++it) {
21       if (r.empty() || *it != r.back()) {
22         while (r.size() >= 2) {
23           int n = r.size();
24           Vec v1 = {r[n - 1].F - r[n - 2].F, r[n - 1].S - r[n - 2].S};
25           Vec v2 = {it->F - r[n - 2].F, it->S - r[n - 2].S};
26           if (cross(v1, v2) > 0) break; #3588
27           r.pop_back();
28         }
29         r.push_back(*it);
30       }
31     } #6639
32     ran(i, 0, (int)r.size() - 1) hull.emplace_back(r[i], r[i + 1]);
33   }
34   Hull(vector<Vec> &vert) { // atleast 2 distinct points
35     sort(vert.begin(), vert.end()); // O(n log(n))
36     extend(vert.begin(), vert.end()); #6560
37     int diff = hull.size();
38     extend(vert.rbegin(), vert.rend());
39     up_beg = hull.begin() + diff;
40   } #0722
41   bool contains(Vec p){ // O(log(n))
42     if (p < hull.front().F || p > up_beg->F) return false;
43     {
44       auto it_low = lower_bound(
45         hull.begin(), up_beg, MP(MP(p.F, (int)-2e9), MP(0, 0)));
46       if (it_low != hull.begin()) --it_low; #3373
47       Vec a = {it_low->S.F - it_low->F.F, it_low->S.S - it_low->F.S};
48       Vec b = {p.F - it_low->F.F, p.S - it_low->F.S};
49       if (cross(a, b) < 0) // < 0 is inclusive, <=0 is exclusive
50         return false;
51     } #2197
52     {
53       auto it_up = lower_bound(hull.rbegin(),
54         hull.rbegin() + (hull.end() - up_beg),
55         MP(MP(p.F, (int)2e9), MP(0, 0)));
56       if (it_up - hull.rbegin() == hull.end() - up_beg) --it_up;
57       Vec a = {it_up->F.F - it_up->S.F, it_up->F.S - it_up->S.S};
58       Vec b = {p.F - it_up->S.F, p.S - it_up->S.S};
59       if (cross(a, b) > 0) // > 0 is inclusive, >=0 is exclusive #7227
60         return false;
61     } #1826
62     return true;
63
64   // The function can have only one local min and max
65   // and may be constant only at min and max.
66   template <typename T>
67   SegIt max(function<T(Seg &>) f) { // O(log(n))
68     auto l = hull.begin();
69     auto r = hull.end();
70     SegIt b = hull.end(); #8566
71     T b_v;
72     while (r - l > 2) {
73       auto m = l + (r - l) / 2;
74       T l_v = f(*l);
75       T l_n_v = f(*(l + 1)); #3586
76       T m_v = f(*m);
77       T m_n_v = f(*(m + 1));
78       if (b == hull.end() || l_v > b_v) {
79         b = l; // If max is at l we may remove it from the range. #7332
80         b_v = l_v
81       }
82       if (l_n_v > l_v) {
83         if (m_v < l_v) {
84           r = m; #7279
85         } else {
86           if (m_n_v > m_v) {
87             l = m + 1; #0656
88           } else {
89             r = m + 1;
90           }
91         }
92       } else {
93         if (m_v < l_v) {
94           l = m + 1; #7311
95         } else {
96           if (m_n_v > m_v) {
97             l = m + 1; #4469
98           } else {
99             r = m + 1;
100           }
101         }
102       }
103     }
104     T l_v = f(*l); #9864
105     if (b == hull.end() || l_v > b_v)
106       b = l;
107     b_v = l_v;
108     if (r - l > 1) {
109

```

```

110 T l_n_v = f(*(l + 1))
111 if (b == hull.end() || l_n_v > b_v) {
112     b = l + 1;
113     b_v = l_n_v;
114 }
115
116     return b;
117
118 SegIt closest(Vec p{// p can't be internal(can be on border),
119                 // hull must have atleast 3 points
120 Seg &ref_p = hull.front();// O(log(n))
121 return max(function<double>(Seg &>(
122     [&p, &ref_p](
123         Seg &seg){// accuracy of used type should be coord^-2
124         if (p == seg.F) return 10 - M_PI
125         Vec v1 = {seg.S.F - seg.F.F, seg.S.S - seg.F.S};
126         Vec v2 = {p.F - seg.F.F, p.S - seg.F.S};
127         ll c_p = cross(v1, v2);
128         if (c_p > 0) {// order the backside by angle
129             Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
130             Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
131             ll d_p = dot(v1, v2);
132             ll c_p = cross(v2, v1);
133             return atan2(c_p, d_p / ;
134         }
135         ll d_p = dot(v1, v2);
136         double res = atan2(d_p, c_p)
137         if (d_p <= 0 && res > 0) res = -M_PI;
138         if (res > 0) {
139             res += 20;
140         } else {
141             res = 10 - res
142         }
143         return res;
144     )));
145
146 template <int DIRECTION>// 1 or -1
147 Vectan_point(Vec p{ // can't be internal or on border
148 // -1 iff CCW rotation of ray from p to res takes it away from
149 // polygon?
150 Seg &ref_p = hull.front();// O(log(n))
151 auto best_seg = max(function<double>(Seg &>(
152     [&p, &ref_p]
153         Seg &seg) { // accuracy of used type should be coord^-2
154         Vec v1 = {ref_p.F.F - p.F, ref_p.F.S - p.S};
155         Vec v2 = {seg.F.F - p.F, seg.F.S - p.S};
156         ll d_p = dot(v1, v2);
157         ll c_p = DIRECTION * cross(v2, v1)
158         return atan2(c_p, d_p); // order by signed angle
159     )));
160     return best_seg->F;

```

#5972	161 SegIt max_in_dir(Vec v{// first is the ans. O(log(n)) 162     return max( 163         function<ll>([&v](Seg &seg){ return dot(v, seg.F); })); 164     %9596 165 166 pair<SegIt, SegIt> intersections(Seg l) {// O(log(n)) 167     int x = l.S.F - l.F.F; 168     int y = l.S.S - l.F.S; 169     Vec dir = {-y, x}; 170     auto it_max = max_in_dir(dir) 171     auto it_min = max_in_dir(MP(y, -x)); 172     ll opt_val = dot(dir, l.F); 173     if (dot(dir, it_max->F) < opt_val    174         dot(dir, it_min->F) > opt_val) 175         return MP(hull.end(), hull.end()); 176     SegIt it_r1, it_r2; 177     function<bool>(Seg &, Seg &> inc_c([&dir](Seg &lft, Seg &rgt) { 178         return dot(dir, lft.F) < dot(dir, rgt.F); 179     }); 180     function<bool>(Seg &, Seg &> dec_c([&dir](Seg &lft, Seg &rgt) { 181         return dot(dir, lft.F) > dot(dir, rgt.F); 182     }); 183     if (it_min <= it_max) { 184         it_r1 = upper_bound(it_min, it_max + 1, l, inc_c) - 1; 185         if (dot(dir, hull.front().F) >= opt_val) { 186             it_r2 = upper_bound(hull.begin(), it_min + 1, l, dec_c) - 1; 187         } else 188             it_r2 = upper_bound(it_max, hull.end(), l, dec_c) - 1; 189     } else { 190         it_r1 = upper_bound(it_max, it_min + 1, l, dec_c) - 1; 191         if (dot(dir, hull.front().F) <= opt_val) 192             it_r2 = upper_bound(hull.begin(), it_max + 1, l, inc_c) - 1; 193         } else { 194             it_r2 = upper_bound(it_min, hull.end(), l, inc_c) - 1; 195         } 196     } 197     return MP(it_r1, it_r2); 198 199 Segdiameter({// O(n) 200     Seg res; 201     ll dia_sq = 0; 202     auto it1 = hull.begin(); 203     auto it2 = up_beg 204     Vec v1 = {hull.back().S.F - hull.back().F.F, 205               hull.back().S.S - hull.back().F.S}; 206     while (it2 != hull.begin()) { 207         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 208                   (it2 - 1)->S.S - (it2 - 1)->F.S} 209         if (cross(v1, v2) > 0) break; 210         --it2; 211     } 212 213     return dia_sq; 214 } 215     %1498 216 217 Segdiameter({// O(n) 218     Seg res; 219     ll dia_sq = 0; 220     auto it1 = hull.begin(); 221     auto it2 = up_beg 222     Vec v1 = {hull.back().S.F - hull.back().F.F, 223               hull.back().S.S - hull.back().F.S}; 224     while (it2 != hull.begin()) { 225         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 226                   (it2 - 1)->S.S - (it2 - 1)->F.S} 227         if (cross(v1, v2) > 0) break; 228         --it2; 229     } 230 231     return dia_sq; 232 } 233     %2632 234 235 Segdiameter({// O(n) 236     Seg res; 237     ll dia_sq = 0; 238     auto it1 = hull.begin(); 239     auto it2 = up_beg 240     Vec v1 = {hull.back().S.F - hull.back().F.F, 241               hull.back().S.S - hull.back().F.S}; 242     while (it2 != hull.begin()) { 243         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 244                   (it2 - 1)->S.S - (it2 - 1)->F.S} 245         if (cross(v1, v2) > 0) break; 246         --it2; 247     } 248 249     return dia_sq; 250 } 251     %5150 252 253 Segdiameter({// O(n) 254     Seg res; 255     ll dia_sq = 0; 256     auto it1 = hull.begin(); 257     auto it2 = up_beg 258     Vec v1 = {hull.back().S.F - hull.back().F.F, 259               hull.back().S.S - hull.back().F.S}; 260     while (it2 != hull.begin()) { 261         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 262                   (it2 - 1)->S.S - (it2 - 1)->F.S} 263         if (cross(v1, v2) > 0) break; 264         --it2; 265     } 266 267     return dia_sq; 268 } 269     %5037 270 271 Segdiameter({// O(n) 272     Seg res; 273     ll dia_sq = 0; 274     auto it1 = hull.begin(); 275     auto it2 = up_beg 276     Vec v1 = {hull.back().S.F - hull.back().F.F, 277               hull.back().S.S - hull.back().F.S}; 278     while (it2 != hull.begin()) { 279         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 280                   (it2 - 1)->S.S - (it2 - 1)->F.S} 281         if (cross(v1, v2) > 0) break; 282         --it2; 283     } 284 285     return dia_sq; 286 } 287     %5037 288 289 Segdiameter({// O(n) 290     Seg res; 291     ll dia_sq = 0; 292     auto it1 = hull.begin(); 293     auto it2 = up_beg 294     Vec v1 = {hull.back().S.F - hull.back().F.F, 295               hull.back().S.S - hull.back().F.S}; 296     while (it2 != hull.begin()) { 297         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 298                   (it2 - 1)->S.S - (it2 - 1)->F.S} 299         if (cross(v1, v2) > 0) break; 300         --it2; 301     } 302 303     return dia_sq; 304 } 305     %5037 306 307 Segdiameter({// O(n) 308     Seg res; 309     ll dia_sq = 0; 310     auto it1 = hull.begin(); 311     auto it2 = up_beg 312     Vec v1 = {hull.back().S.F - hull.back().F.F, 313               hull.back().S.S - hull.back().F.S}; 314     while (it2 != hull.begin()) { 315         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 316                   (it2 - 1)->S.S - (it2 - 1)->F.S} 317         if (cross(v1, v2) > 0) break; 318         --it2; 319     } 320 321     return dia_sq; 322 } 323     %5037 324 325 Segdiameter({// O(n) 326     Seg res; 327     ll dia_sq = 0; 328     auto it1 = hull.begin(); 329     auto it2 = up_beg 330     Vec v1 = {hull.back().S.F - hull.back().F.F, 331               hull.back().S.S - hull.back().F.S}; 332     while (it2 != hull.begin()) { 333         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 334                   (it2 - 1)->S.S - (it2 - 1)->F.S} 335         if (cross(v1, v2) > 0) break; 336         --it2; 337     } 338 339     return dia_sq; 340 } 341     %5037 342 343 Segdiameter({// O(n) 344     Seg res; 345     ll dia_sq = 0; 346     auto it1 = hull.begin(); 347     auto it2 = up_beg 348     Vec v1 = {hull.back().S.F - hull.back().F.F, 349               hull.back().S.S - hull.back().F.S}; 350     while (it2 != hull.begin()) { 351         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 352                   (it2 - 1)->S.S - (it2 - 1)->F.S} 353         if (cross(v1, v2) > 0) break; 354         --it2; 355     } 356 357     return dia_sq; 358 } 359     %5037 360 361 Segdiameter({// O(n) 362     Seg res; 363     ll dia_sq = 0; 364     auto it1 = hull.begin(); 365     auto it2 = up_beg 366     Vec v1 = {hull.back().S.F - hull.back().F.F, 367               hull.back().S.S - hull.back().F.S}; 368     while (it2 != hull.begin()) { 369         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 370                   (it2 - 1)->S.S - (it2 - 1)->F.S} 371         if (cross(v1, v2) > 0) break; 372         --it2; 373     } 374 375     return dia_sq; 376 } 377     %5037 378 379 Segdiameter({// O(n) 380     Seg res; 381     ll dia_sq = 0; 382     auto it1 = hull.begin(); 383     auto it2 = up_beg 384     Vec v1 = {hull.back().S.F - hull.back().F.F, 385               hull.back().S.S - hull.back().F.S}; 386     while (it2 != hull.begin()) { 387         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 388                   (it2 - 1)->S.S - (it2 - 1)->F.S} 389         if (cross(v1, v2) > 0) break; 390         --it2; 391     } 392 393     return dia_sq; 394 } 395     %5037 396 397 Segdiameter({// O(n) 398     Seg res; 399     ll dia_sq = 0; 400     auto it1 = hull.begin(); 401     auto it2 = up_beg 402     Vec v1 = {hull.back().S.F - hull.back().F.F, 403               hull.back().S.S - hull.back().F.S}; 404     while (it2 != hull.begin()) { 405         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 406                   (it2 - 1)->S.S - (it2 - 1)->F.S} 407         if (cross(v1, v2) > 0) break; 408         --it2; 409     } 410 411     return dia_sq; 412 } 413     %5037 414 415 Segdiameter({// O(n) 416     Seg res; 417     ll dia_sq = 0; 418     auto it1 = hull.begin(); 419     auto it2 = up_beg 420     Vec v1 = {hull.back().S.F - hull.back().F.F, 421               hull.back().S.S - hull.back().F.S}; 422     while (it2 != hull.begin()) { 423         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 424                   (it2 - 1)->S.S - (it2 - 1)->F.S} 425         if (cross(v1, v2) > 0) break; 426         --it2; 427     } 428 429     return dia_sq; 430 } 431     %5037 432 433 Segdiameter({// O(n) 434     Seg res; 435     ll dia_sq = 0; 436     auto it1 = hull.begin(); 437     auto it2 = up_beg 438     Vec v1 = {hull.back().S.F - hull.back().F.F, 439               hull.back().S.S - hull.back().F.S}; 440     while (it2 != hull.begin()) { 441         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 442                   (it2 - 1)->S.S - (it2 - 1)->F.S} 443         if (cross(v1, v2) > 0) break; 444         --it2; 445     } 446 447     return dia_sq; 448 } 449     %5037 450 451 Segdiameter({// O(n) 452     Seg res; 453     ll dia_sq = 0; 454     auto it1 = hull.begin(); 455     auto it2 = up_beg 456     Vec v1 = {hull.back().S.F - hull.back().F.F, 457               hull.back().S.S - hull.back().F.S}; 458     while (it2 != hull.begin()) { 459         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 460                   (it2 - 1)->S.S - (it2 - 1)->F.S} 461         if (cross(v1, v2) > 0) break; 462         --it2; 463     } 464 465     return dia_sq; 466 } 467     %5037 468 469 Segdiameter({// O(n) 470     Seg res; 471     ll dia_sq = 0; 472     auto it1 = hull.begin(); 473     auto it2 = up_beg 474     Vec v1 = {hull.back().S.F - hull.back().F.F, 475               hull.back().S.S - hull.back().F.S}; 476     while (it2 != hull.begin()) { 477         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 478                   (it2 - 1)->S.S - (it2 - 1)->F.S} 479         if (cross(v1, v2) > 0) break; 480         --it2; 481     } 482 483     return dia_sq; 484 } 485     %5037 486 487 Segdiameter({// O(n) 488     Seg res; 489     ll dia_sq = 0; 490     auto it1 = hull.begin(); 491     auto it2 = up_beg 492     Vec v1 = {hull.back().S.F - hull.back().F.F, 493               hull.back().S.S - hull.back().F.S}; 494     while (it2 != hull.begin()) { 495         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 496                   (it2 - 1)->S.S - (it2 - 1)->F.S} 497         if (cross(v1, v2) > 0) break; 498         --it2; 499     } 500 501     return dia_sq; 502 } 503     %5037 504 505 Segdiameter({// O(n) 506     Seg res; 507     ll dia_sq = 0; 508     auto it1 = hull.begin(); 509     auto it2 = up_beg 510     Vec v1 = {hull.back().S.F - hull.back().F.F, 511               hull.back().S.S - hull.back().F.S}; 512     while (it2 != hull.begin()) { 513         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 514                   (it2 - 1)->S.S - (it2 - 1)->F.S} 515         if (cross(v1, v2) > 0) break; 516         --it2; 517     } 518 519     return dia_sq; 520 } 521     %5037 522 523 Segdiameter({// O(n) 524     Seg res; 525     ll dia_sq = 0; 526     auto it1 = hull.begin(); 527     auto it2 = up_beg 528     Vec v1 = {hull.back().S.F - hull.back().F.F, 529               hull.back().S.S - hull.back().F.S}; 530     while (it2 != hull.begin()) { 531         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 532                   (it2 - 1)->S.S - (it2 - 1)->F.S} 533         if (cross(v1, v2) > 0) break; 534         --it2; 535     } 536 537     return dia_sq; 538 } 539     %5037 540 541 Segdiameter({// O(n) 542     Seg res; 543     ll dia_sq = 0; 544     auto it1 = hull.begin(); 545     auto it2 = up_beg 546     Vec v1 = {hull.back().S.F - hull.back().F.F, 547               hull.back().S.S - hull.back().F.S}; 548     while (it2 != hull.begin()) { 549         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 550                   (it2 - 1)->S.S - (it2 - 1)->F.S} 551         if (cross(v1, v2) > 0) break; 552         --it2; 553     } 554 555     return dia_sq; 556 } 557     %5037 558 559 Segdiameter({// O(n) 560     Seg res; 561     ll dia_sq = 0; 562     auto it1 = hull.begin(); 563     auto it2 = up_beg 564     Vec v1 = {hull.back().S.F - hull.back().F.F, 565               hull.back().S.S - hull.back().F.S}; 566     while (it2 != hull.begin()) { 567         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 568                   (it2 - 1)->S.S - (it2 - 1)->F.S} 569         if (cross(v1, v2) > 0) break; 570         --it2; 571     } 572 573     return dia_sq; 574 } 575     %5037 576 577 Segdiameter({// O(n) 578     Seg res; 579     ll dia_sq = 0; 580     auto it1 = hull.begin(); 581     auto it2 = up_beg 582     Vec v1 = {hull.back().S.F - hull.back().F.F, 583               hull.back().S.S - hull.back().F.S}; 584     while (it2 != hull.begin()) { 585         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 586                   (it2 - 1)->S.S - (it2 - 1)->F.S} 587         if (cross(v1, v2) > 0) break; 588         --it2; 589     } 590 591     return dia_sq; 592 } 593     %5037 594 595 Segdiameter({// O(n) 596     Seg res; 597     ll dia_sq = 0; 598     auto it1 = hull.begin(); 599     auto it2 = up_beg 600     Vec v1 = {hull.back().S.F - hull.back().F.F, 601               hull.back().S.S - hull.back().F.S}; 602     while (it2 != hull.begin()) { 603         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 604                   (it2 - 1)->S.S - (it2 - 1)->F.S} 605         if (cross(v1, v2) > 0) break; 606         --it2; 607     } 608 609     return dia_sq; 610 } 611     %5037 612 613 Segdiameter({// O(n) 614     Seg res; 615     ll dia_sq = 0; 616     auto it1 = hull.begin(); 617     auto it2 = up_beg 618     Vec v1 = {hull.back().S.F - hull.back().F.F, 619               hull.back().S.S - hull.back().F.S}; 620     while (it2 != hull.begin()) { 621         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 622                   (it2 - 1)->S.S - (it2 - 1)->F.S} 623         if (cross(v1, v2) > 0) break; 624         --it2; 625     } 626 627     return dia_sq; 628 } 629     %5037 630 631 Segdiameter({// O(n) 632     Seg res; 633     ll dia_sq = 0; 634     auto it1 = hull.begin(); 635     auto it2 = up_beg 636     Vec v1 = {hull.back().S.F - hull.back().F.F, 637               hull.back().S.S - hull.back().F.S}; 638     while (it2 != hull.begin()) { 639         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 640                   (it2 - 1)->S.S - (it2 - 1)->F.S} 641         if (cross(v1, v2) > 0) break; 642         --it2; 643     } 644 645     return dia_sq; 646 } 647     %5037 648 649 Segdiameter({// O(n) 650     Seg res; 651     ll dia_sq = 0; 652     auto it1 = hull.begin(); 653     auto it2 = up_beg 654     Vec v1 = {hull.back().S.F - hull.back().F.F, 655               hull.back().S.S - hull.back().F.S}; 656     while (it2 != hull.begin()) { 657         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 658                   (it2 - 1)->S.S - (it2 - 1)->F.S} 659         if (cross(v1, v2) > 0) break; 660         --it2; 661     } 662 663     return dia_sq; 664 } 665     %5037 666 667 Segdiameter({// O(n) 668     Seg res; 669     ll dia_sq = 0; 670     auto it1 = hull.begin(); 671     auto it2 = up_beg 672     Vec v1 = {hull.back().S.F - hull.back().F.F, 673               hull.back().S.S - hull.back().F.S}; 674     while (it2 != hull.begin()) { 675         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 676                   (it2 - 1)->S.S - (it2 - 1)->F.S} 677         if (cross(v1, v2) > 0) break; 678         --it2; 679     } 680 681     return dia_sq; 682 } 683     %5037 684 685 Segdiameter({// O(n) 686     Seg res; 687     ll dia_sq = 0; 688     auto it1 = hull.begin(); 689     auto it2 = up_beg 690     Vec v1 = {hull.back().S.F - hull.back().F.F, 691               hull.back().S.S - hull.back().F.S}; 692     while (it2 != hull.begin()) { 693         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 694                   (it2 - 1)->S.S - (it2 - 1)->F.S} 695         if (cross(v1, v2) > 0) break; 696         --it2; 697     } 698 699     return dia_sq; 700 } 701     %5037 702 703 Segdiameter({// O(n) 704     Seg res; 705     ll dia_sq = 0; 706     auto it1 = hull.begin(); 707     auto it2 = up_beg 708     Vec v1 = {hull.back().S.F - hull.back().F.F, 709               hull.back().S.S - hull.back().F.S}; 710     while (it2 != hull.begin()) { 711         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 712                   (it2 - 1)->S.S - (it2 - 1)->F.S} 713         if (cross(v1, v2) > 0) break; 714         --it2; 715     } 716 717     return dia_sq; 718 } 719     %5037 720 721 Segdiameter({// O(n) 722     Seg res; 723     ll dia_sq = 0; 724     auto it1 = hull.begin(); 725     auto it2 = up_beg 726     Vec v1 = {hull.back().S.F - hull.back().F.F, 727               hull.back().S.S - hull.back().F.S}; 728     while (it2 != hull.begin()) { 729         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 730                   (it2 - 1)->S.S - (it2 - 1)->F.S} 731         if (cross(v1, v2) > 0) break; 732         --it2; 733     } 734 735     return dia_sq; 736 } 737     %5037 738 739 Segdiameter({// O(n) 740     Seg res; 741     ll dia_sq = 0; 742     auto it1 = hull.begin(); 743     auto it2 = up_beg 744     Vec v1 = {hull.back().S.F - hull.back().F.F, 745               hull.back().S.S - hull.back().F.S}; 746     while (it2 != hull.begin()) { 747         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 748                   (it2 - 1)->S.S - (it2 - 1)->F.S} 749         if (cross(v1, v2) > 0) break; 750         --it2; 751     } 752 753     return dia_sq; 754 } 755     %5037 756 757 Segdiameter({// O(n) 758     Seg res; 759     ll dia_sq = 0; 760     auto it1 = hull.begin(); 761     auto it2 = up_beg 762     Vec v1 = {hull.back().S.F - hull.back().F.F, 763               hull.back().S.S - hull.back().F.S}; 764     while (it2 != hull.begin()) { 765         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 766                   (it2 - 1)->S.S - (it2 - 1)->F.S} 767         if (cross(v1, v2) > 0) break; 768         --it2; 769     } 770 771     return dia_sq; 772 } 773     %5037 774 775 Segdiameter({// O(n) 776     Seg res; 777     ll dia_sq = 0; 778     auto it1 = hull.begin(); 779     auto it2 = up_beg 780     Vec v1 = {hull.back().S.F - hull.back().F.F, 781               hull.back().S.S - hull.back().F.S}; 782     while (it2 != hull.begin()) { 783         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 784                   (it2 - 1)->S.S - (it2 - 1)->F.S} 785         if (cross(v1, v2) > 0) break; 786         --it2; 787     } 788 789     return dia_sq; 790 } 791     %5037 792 793 Segdiameter({// O(n) 794     Seg res; 795     ll dia_sq = 0; 796     auto it1 = hull.begin(); 797     auto it2 = up_beg 798     Vec v1 = {hull.back().S.F - hull.back().F.F, 799               hull.back().S.S - hull.back().F.S}; 800     while (it2 != hull.begin()) { 801         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 802                   (it2 - 1)->S.S - (it2 - 1)->F.S} 803         if (cross(v1, v2) > 0) break; 804         --it2; 805     } 806 807     return dia_sq; 808 } 809     %5037 810 811 Segdiameter({// O(n) 812     Seg res; 813     ll dia_sq = 0; 814     auto it1 = hull.begin(); 815     auto it2 = up_beg 816     Vec v1 = {hull.back().S.F - hull.back().F.F, 817               hull.back().S.S - hull.back().F.S}; 818     while (it2 != hull.begin()) { 819         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 820                   (it2 - 1)->S.S - (it2 - 1)->F.S} 821         if (cross(v1, v2) > 0) break; 822         --it2; 823     } 824 825     return dia_sq; 826 } 827     %5037 828 829 Segdiameter({// O(n) 830     Seg res; 831     ll dia_sq = 0; 832     auto it1 = hull.begin(); 833     auto it2 = up_beg 834     Vec v1 = {hull.back().S.F - hull.back().F.F, 835               hull.back().S.S - hull.back().F.S}; 836     while (it2 != hull.begin()) { 837         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 838                   (it2 - 1)->S.S - (it2 - 1)->F.S} 839         if (cross(v1, v2) > 0) break; 840         --it2; 841     } 842 843     return dia_sq; 844 } 845     %5037 846 847 Segdiameter({// O(n) 848     Seg res; 849     ll dia_sq = 0; 850     auto it1 = hull.begin(); 851     auto it2 = up_beg 852     Vec v1 = {hull.back().S.F - hull.back().F.F, 853               hull.back().S.S - hull.back().F.S}; 854     while (it2 != hull.begin()) { 855         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 856                   (it2 - 1)->S.S - (it2 - 1)->F.S} 857         if (cross(v1, v2) > 0) break; 858         --it2; 859     } 860 861     return dia_sq; 862 } 863     %5037 864 865 Segdiameter({// O(n) 866     Seg res; 867     ll dia_sq = 0; 868     auto it1 = hull.begin(); 869     auto it2 = up_beg 870     Vec v1 = {hull.back().S.F - hull.back().F.F, 871               hull.back().S.S - hull.back().F.S}; 872     while (it2 != hull.begin()) { 873         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 874                   (it2 - 1)->S.S - (it2 - 1)->F.S} 875         if (cross(v1, v2) > 0) break; 876         --it2; 877     } 878 879     return dia_sq; 880 } 881     %5037 882 883 Segdiameter({// O(n) 884     Seg res; 885     ll dia_sq = 0; 886     auto it1 = hull.begin(); 887     auto it2 = up_beg 888     Vec v1 = {hull.back().S.F - hull.back().F.F, 889               hull.back().S.S - hull.back().F.S}; 890     while (it2 != hull.begin()) { 891         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 892                   (it2 - 1)->S.S - (it2 - 1)->F.S} 893         if (cross(v1, v2) > 0) break; 894         --it2; 895     } 896 897     return dia_sq; 898 } 899     %5037 900 901 Segdiameter({// O(n) 902     Seg res; 903     ll dia_sq = 0; 904     auto it1 = hull.begin(); 905     auto it2 = up_beg 906     Vec v1 = {hull.back().S.F - hull.back().F.F, 907               hull.back().S.S - hull.back().F.S}; 908     while (it2 != hull.begin()) { 909         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 910                   (it2 - 1)->S.S - (it2 - 1)->F.S} 911         if (cross(v1, v2) > 0) break; 912         --it2; 913     } 914 915     return dia_sq; 916 } 917     %5037 918 919 Segdiameter({// O(n) 920     Seg res; 921     ll dia_sq = 0; 922     auto it1 = hull.begin(); 923     auto it2 = up_beg 924     Vec v1 = {hull.back().S.F - hull.back().F.F, 925               hull.back().S.S - hull.back().F.S}; 926     while (it2 != hull.begin()) { 927         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 928                   (it2 - 1)->S.S - (it2 - 1)->F.S} 929         if (cross(v1, v2) > 0) break; 930         --it2; 931     } 932 933     return dia_sq; 934 } 935     %5037 936 937 Segdiameter({// O(n) 938     Seg res; 939     ll dia_sq = 0; 940     auto it1 = hull.begin(); 941     auto it2 = up_beg 942     Vec v1 = {hull.back().S.F - hull.back().F.F, 943               hull.back().S.S - hull.back().F.S}; 944     while (it2 != hull.begin()) { 945         Vec v2 = {(it2 - 1)->S.F - (it2 - 1)->F.F, 946                   (it2 - 1)->S.S - (it2 - 1)->F.S} 947         if (cross(v1, v2) > 0) break; 948         --it2; 949     } 950 951     return dia_sq; 952 } 953
-------	--

```

212 }
213 while (it2 != hull.end()) { // check all antipodal pairs
214     if (dist_sq(it1->F, it2->F) > dia_sq)
215         res = {it1->F, it2->F};
216     dia_sq = dist_sq(res.F, res.S);
217 }
218 Vec v1 = {it1->S.F - it1->F.F, it1->S.S - it1->F.S}; #1246
219 Vec v2 = {it2->S.F - it2->F.F, it2->S.S - it2->F.S};
220 if (cross(v1, v2) == 0) #9381
221     if (dist_sq(it1->S, it2->F) > dia_sq) {
222         res = {it1->S, it2->F};
223         dia_sq = dist_sq(res.F, res.S);
224     }
225     if (dist_sq(it1->F, it2->S) > dia_sq) #7011
226         res = {it1->F, it2->S};
227         dia_sq = dist_sq(res.F, res.S);
228 } // report cross pairs at parallel lines.
229 ++it1;
230 ++it2; #5626
231 } else if (cross(v1, v2) < 0) {
232     ++it1;
233 } else {
234     ++it2; #4406
235 }
236 }
237 return res;
238 }

```

## 8 Delaunay triangulation $\mathcal{O}(n \log n)$

```

1 const int max_co = (1 << 28) - 5;
2 struct Vec {
3     int x, y;
4     bool operator==(const Vec &oth) { return x == oth.x && y == oth.y; }
5     bool operator!=(const Vec &oth) { return !operator==(oth); }
6     Vec operator-(const Vec &oth) { return {x - oth.x, y - oth.y}; } #2919
7 }
8 llcross(Vec a, Vec b{ return (ll)a.x * b.y - (ll)a.y * b.x; }
9 lldot(Vec a, Vec b{ return (ll)a.x * b.x + (ll)a.y * b.y; }
10 struct Edge {
11     Vec tar;
12     Edge *nxt; #8008
13     Edge *inv = NULL;
14     Edge *rep = NULL;
15     bool vis = false;
16 };
17 struct Seg #7311
18     Vec a, b;
19     bool operator==(const Seg &oth) { return a == oth.a && b == oth.b; }
20     bool operator!=(const Seg &oth) { return !operator==(oth); }
21 };
22 llorient(Vec a, Vec b, Vec c #6432
23     return (ll)a.x * (b.y - c.y) + (ll)b.x * (c.y - a.y) +

```

```

24             (ll)c.x * (a.y - b.y); %6334
25
26 bool_in_c_circle(Vec *arr, Vec d{ #4264
27     if (cross(arr[1] - arr[0], arr[2] - arr[0]) == 0)
28         return true; // degenerate
29     ll m[3][3];
30     ran(i, 0, 3)
31     m[i][0] = arr[i].x - d.x;
32     m[i][1] = arr[i].y - d.y;
33     m[i][2] = m[i][0] * m[i][0];
34     m[i][2] += m[i][1] * m[i][1];
35
36     __int128 res = 0; #7305
37     res += (__int128)(m[0][0] * m[1][1] - m[0][1] * m[1][0]) * m[2][2];
38     res += (__int128)(m[1][0] * m[2][1] - m[1][1] * m[2][0]) * m[0][2];
39     res -= (__int128)(m[0][0] * m[2][1] - m[0][1] * m[2][0]) * m[1][2];
40     return res > 0 #1845
41
42 Edge add_triangle(Edge *a, Edge *b, Edge *c{ #6793
43     Edge *old[] = {a, b, c};
44     Edge *tmp = new Edge[3];
45     ran(i, 0, 3) {
46         old[i]->rep = tmp + i #8219
47         tmp[i] = {old[i]->tar, tmp + (i + 1) % 3, old[i]->inv};
48         if (tmp[i].inv) tmp[i].inv->inv = tmp + i;
49     }
50     return tmp;
51
52 Edge add_point(Vec p, Edge *cur{// returns outgoing edge #8178
53     Edge *triangle[] = {cur, cur->nxt, cur->nxt->nxt};
54     ran(i, 0, 3) {
55         if (orient(triangle[i]->tar, triangle[(i + 1) % 3]->tar, p) < 0)
56             return NULL #0233
57     }
58     ran(i, 0, 3) {
59         if (triangle[i]->rep) {
60             Edge *res = add_point(p, triangle[i]->rep); #5636
61             if (res
62                 return res; // unless we are on last layer we must exit here
63             }
64         }
65         Edge p_as_e{p};
66         Edge tmp{cur->tar} #1432
67         tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
68         Edge *res = tmp.inv->nxt;
69         tmp.tar = cur->tar;
70         tmp.inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt); #8359
71         tmp.tar = cur->tar
72         res->inv = add_triangle(&p_as_e, &tmp, cur = cur->nxt);
73         res->inv->inv = res;
74         return res;

```

```

75 }
76 Edge *delaunay(vector<Vec> &points) #3029
77 random_shuffle(points.begin(), points.end());
78 Vec arr[] = {{4 * max_co, 4 * max_co}, {-4 * max_co, max_co},
79   {max_co, -4 * max_co}};
80 Edge *res = new Edge[3];
81 ran(i, 0, 3) res[i] = {arr[i], res + (i + 1) % 3};
82 for (Vec &cur : points) #4575
83   Edge *loc = add_point(cur, res);
84   Edge *out = loc;
85   arr[0] = cur;
86   while (true) {
87     arr[1] = out->tar #3471
88     arr[2] = out->nxt->tar;
89     Edge *e = out->nxt->inv;
90     if (e && in_c_circle(arr, e->nxt->tar)) {
91       Edge tmp{cur};
92       tmp.inv = add_triangle(&tmp, out, e->nxt);
93       tmp.tar = e->nxt->tar #9851
94       tmp.inv->inv = add_triangle(&tmp, e->nxt->nxt, out->nxt->nxt);
95       out = tmp.inv->nxt;
96       continue;
97     }
98     out = out->nxt->nxt->inv #0151
99     if (out->tar == loc->tar) break;
100   }
101 }
102 return res; #6769 %6769
103 void extract_triangles(Edge *cur, vector<vector<Seg> > &res{ #3769
104   if (!cur->vis) {
105     bool inc = true;
106     Edge *it = cur;
107     do
108       it->vis = true;
109     if (it->rep) {
110       extract_triangles(it->rep, res);
111       inc = false;
112     }
113     it = it->nxt;
114   } while (it != cur);
115   if (inc) {
116     Edge *triangle[3] = {cur, cur->nxt, cur->nxt->nxt};
117     res.resize(res.size() + 1) #6207
118     vector<Seg> &tar = res.back();
119     ran(i, 0, 3) {
120       if ((abs(triangle[i]->tar.x) < max_co &&
121           abs(triangle[(i + 1) % 3]->tar.x) < max_co))
122         tar.push_back
123           {triangle[i]->tar, triangle[(i + 1) % 3]->tar}; #3011
124     }
125     if (tar.empty()) res.pop_back();
126   }

```

```

127 }
128 #8602



---



## 9 Aho Corasick $\mathcal{O}(|\alpha| \sum \text{len})$



---



```

1 const int alpha_size = 26;
2 struct Node {
3     Node *nxt[alpha_size]; // May use other structures to move in trie
4     Node *suffix;
5     Node() { memset(nxt, 0, alpha_size * sizeof(Node *)); }
6     int cnt = 0
7 };
8 Node aho_corasick(vector<vector<char>> &dict{
9     Node *root = new Node;
10    root->suffix = 0;
11    vector<pair<vector<char>, Node *>> state
12    for (vector<char> &s : dict) state.emplace_back(&s, root);
13    for (int i = 0; !state.empty(); ++i) {
14        vector<pair<vector<char>, Node *>> nstate;
15        for (auto &cur : state) {
16            Node *nxt = cur.second->nxt[(*cur.first)[i]];
17            if (nxt)
18                cur.second = nxt;
19            } else {
20                nxt = new Node;
21                cur.second->nxt[(*cur.first)[i]] = nxt;
22                Node *suf = cur.second->suffix
23                cur.second = nxt;
24                nxt->suffix = root; // set correct suffix link
25                while (suf) {
26                    if (suf->nxt[(*cur.first)[i]]) {
27                        nxt->suffix = suf->nxt[(*cur.first)[i]];
28                        break
29                    }
30                    suf = suf->suffix;
31                }
32            }
33            if (cur.first->size() > i + 1) nstate.push_back(cur);
34        }
35        state = nstate;
36    }
37    return root;
38    %2882 // auxilary functions for searching and counting
39 Node walk(Node *cur,
40 char c{ // longest prefix in dict that is suffix of walked string.
41 while (true) {
42     if (cur->nxt[c]) return cur->nxt[c];
43     if (!cur->suffix) return cur
44     cur = cur->suffix;
45 }
46 voidcnt_matches(Node *root, vector<char> &match_in{
47
%
```


```

```

48 Node *cur = root;
49 for (char c : match_in) {
50     cur = walk(cur, c);
51     ++cur->cnt
52 }
53 %8156
54 voidadd_cnt(Node *root{ // After counting matches propagate ONCE to
55     // suffixes for final counts
56     vector<Node *> to_visit = {root};
57     ran(i, 0, to_visit.size()) {
58         Node *cur = to_visit[i];
59         ran(j, 0, alpha_size)
60         if (cur->nxt[j]) to_visit.push_back(cur->nxt[j]);
61     }
62 }
63 for (int i = to_visit.size() - 1; i > 0; --i)
64     to_visit[i]->suffix->cnt += to_visit[i]->cnt
65 %0488
66 intmain{
67     int n, len;
68     scanf "%d %d", &len, &n);
69     vector<char> a(len + 1);
70     scanf "%s", a.data());
71     a.pop_back();
72     for (char &c : a) c -= 'a';
73     vector<vector<char>> dict(n);
74     ran(i, 0, n) {
75         scanf "%d", &len);
76         dict[i].resize(len + 1);
77         scanf "%s", dict[i].data());
78         dict[i].pop_back();
79         for (char &c : dict[i]) c -= 'a';
80     }
81     Node *root = aho_corasick(dict);
82     cnt_matches(root, a);
83     add_cnt(root);
84     ran(i, 0, n) {
85         Node *cur = root;
86         for (char c : dict[i]) cur = walk(cur, c);
87         printf "%d\n", cur->cnt);
88 }

```

## 10 Suffix automaton and tree $\mathcal{O}((n + q) \log(|\alpha|))$

```

1 struct Node {
2     map<char, Node *> nxt_char;
3     // Map is faster than hashtable and unsorted arrays
4     int len; // Length of longest suffix in equivalence class.
5     Node *suf;
6     boolhas_nxt(char c const{ return nxt_char.count(c); }
7     Node nxt(char c
8         if (!has_nxt(c)) return NULL;
9         return nxt_char[c];
#9664

```

```

10    }
11    voidset_nxt(char c, Node *node{ nxt_char[c] = node; }
12    Node split(int new_len, char c
#8305
13        Node *new_n = new Node;
14        new_n->nxt_char = nxt_char;
15        new_n->len = new_len;
16        new_n->suf = suf;
17        suf = new_n
18        return new_n;
19
20    // Extra functions for matching and counting
21    Node lower(int depth{
22        // move to longest suf of current with a maximum length of depth.
23        if (suf->len >= depth) return suf->lower(depth);
24        return this;
25    }
26    Node *walk(char c, int depth, int &match_len)
#2130
27        // move to longest suffix of walked path that is a substring
28        match_len = min(match_len, len);
29        // includes depth limit(needed for finding matches)
30        if (has_nxt(c)) { // as suffixes are in classes match_len must be
31            // tracked externally
32            ++match_len;
33            return nxt(c)->lower(depth);
34        }
35        if (suf) return suf->walk(c, depth, match_len);
36        return this;
37
38    int paths_to_end = 0;
39    voidset_as_end{// All suffixes of current node are marked as
40        // ending nodes.
41    paths_to_end += 1;
42    if (suf) suf->set_as_end();
43
44    bool vis = false;
45    voidcalc_paths(){
46        /* Call ONCE from ROOT. For each node calculates number of ways
47         * to reach an end node. paths_to_end is occurrence count for any
48         * strings in current suffix equivalence class. */
49        if (!vis) {
50            vis = true;
51            for (auto cur : nxt_char)
#2404
52                cur.second->calc_paths();
53                paths_to_end += cur.second->paths_to_end;
54            }
55        }
#7906
56        // Transform into suffix tree of reverse string
57        map<char, Node *> tree_links;
58        int end_dist = 1 << 30;
59        intcalc_end_dist({
60

```

```

61 if (end_dist == 1 << 30) {
62     if (nxt_char.empty()) end_dist = 0 #7524
63     for (auto cur : nxt_char)
64         end_dist = min(end_dist, 1 + cur.second->calc_end_dist());
65 }
66 return end_dist;
67 #2021
68 bool vis_t = false;
69 void build_suffix_tree(string &s{// Call ONCE from ROOT.
70     if (!vis_t) {
71         vis_t = true;
72         if (suf #6270
73             suf->tree_links[s[s.size() - end_dist - suf->len - 1]] = this;
74             for (auto cur : nxt_char) cur.second->build_suffix_tree(s);
75     }
76 }
77 }
78 struct SufAuto {
79     Node *last;
80     Node *root;
81     void extend(char new_c{
82         Node *nlast = new Node #4696
83         nlast->len = last->len + 1;
84         Node *swn = last;
85         while (swn && !swn->has_nxt(new_c)) {
86             swn->set_nxt(new_c, nlast);
87             swn = swn->suf
88         }
89         if (!swn) {
90             nlast->suf = root;
91         } else {
92             Node *max_sbstr = swn->nxt(new_c) #7000
93             if (swn->len + 1 == max_sbstr->len) {
94                 nlast->suf = max_sbstr;
95             } else {
96                 Node *eq_sbstr = max_sbstr->split(swn->len + 1, new_c);
97                 nlast->suf = eq_sbstr #2075
98                 Node *x = swn;
99                 while (x != 0 && x->nxt(new_c) == max_sbstr) {
100                     x->set_nxt(new_c, eq_sbstr);
101                     x = x->suf;
102                 }
103             }
104             last = nlast;
105         }
106     }
107     SufAuto(string &s) {
108         root = new Node;
109         root->len = 0;
110         root->suf = NULL;
111         last = root;
112         for (char c : s) extend(c); #9604
113     }
114 }
```

```

113     root->calc_end_dist(); // To build suffix tree use reversed string
114     root->build_suffix_tree(s);
115 }
```

## 11 Dinic

```

1 struct MaxFlow {
2     const static ll INF = 1e18;
3     int source, sink;
4     ll sink_pot = 0;
5     vector<int> start, now, lvl, adj, rcap, cap_loc, bfs;
6     vector<bool> visited;
7     vector<ll> cap, orig_cap/*ly*/, cost;
8     priority_queue<pair<ll, int>, vector<pair<ll, int> >, greater<pair<ll, int> > >
9         dist_que; /*rg*/
10    void add_flow(int idx, ll flow, bool cont = true{
11        cap[idx] -= flow;
12        if (cont) add_flow(rcap[idx], -flow, false);
13    }
14    MaxFlow(
15        const vector<tuple<int, int, ll/*ly*/, ll/*ry*/> > &edges) {
16        for (auto &cur : edges) { // from, to, cap, rcap/*ly*/, cost/*ry*/
17            start.resize(
18                max(max(get<0>(cur), get<1>(cur)) + 2, (int)start.size()));
19            ++start[get<0>(cur) + 1];
20            ++start[get<1>(cur) + 1];
21        }
22        for (int i = 1; i < start.size(); ++i) start[i] += start[i - 1];
23        now = start;
24        adj.resize(start.back());
25        cap.resize(start.back());
26        rcap.resize(start.back());
27        /*ly*/ cost.resize(start.back()); /*ry*/
28        for (auto &cur : edges) {
29            int u, v;
30            ll c, rc/*ly*/, c_cost/*ry*/;
31            tie(u, v, c, rc/*ly*/, c_cost/*ry*/) = cur;
32            assert(u != v);
33            adj[now[u]] = v;
34            adj[now[v]] = u;
35            rcap[now[u]] = now[v];
36            rcap[now[v]] = now[u];
37            cap_loc.push_back(now[u]);
38            /*ly*/ cost[now[u]] = c_cost;
39            cost[now[v]] = -c_cost; /*ry*/
40            cap[now[u]++] = c;
41            cap[now[v]++] = rc;
42            orig_cap.push_back(c);
43        }
44    }
45    bool dinic_bfs() {
46
```

```

47 lvl.clear();
48 lvl.resize(start.size());
49 bfs.clear();
50 bfs.resize(1, source);
51 now = start;
52 lvl[source] = 1;
53 for (int i = 0; i < bfs.size(); ++i) {
54     int u = bfs[i];
55     while (now[u] < start[u + 1]) {
56         int v = adj[now[u]];
57         if /*ly*/ cost[now[u]] == 0 && /*ry*/ cap[now[u]] > 0 &&
58             lvl[v] == 0) {
59             lvl[v] = lvl[u] + 1;
60             bfs.push_back(v);
61         }
62         ++now[u];
63     }
64     return lvl[sink];
65 }
66 ll dinic_dfs(int u, ll flow) {
67     if (u == sink) return flow;
68     while (now[u] < start[u + 1]) {
69         int v = adj[now[u]];
70         if (lvl[v] == lvl[u] + 1/*ly*/ && cost[now[u]] == 0/*ry*/ &&
71             cap[now[u]] != 0) {
72             ll res = dinic_dfs(v, min(flow, cap[now[u]]));
73             if (res) {
74                 add_flow(now[u], res);
75                 return res;
76             }
77         }
78         ++now[u];
79     }
80     return 0;
81 }
82 /*ly*/ bool recalc_dist(bool check_imp = false{
83     now = start;
84     visited.clear();
85     visited.resize(start.size());
86     dist_que.emplace(0, source);
87     bool imp = false;
88     while (!dist_que.empty()) {
89         int u;
90         ll dist;
91         tie(dist, u) = dist_que.top();
92         dist_que.pop();
93         if (!visited[u]) {
94             visited[u] = true;
95             if (check_imp && dist != 0) imp = true;
96             if (u == sink) sink_pot += dist;
97             while (now[u] < start[u + 1]) {
98
99             int v = adj[now[u]];
100            if (!visited[v] && cap[now[u]])
101                dist_que.emplace(dist + cost[now[u]], v);
102                cost[now[u]] += dist;
103                cost[rcap[now[u]++]] -= dist;
104            }
105        }
106        if (check_imp) return imp;
107        return visited[sink];
108    }
109 }/*ry*/
110 /*lp*/ bool recalc_dist_bellman_ford() { // return whether there is
111 // a negative cycle
112     int i = 0;
113     for (; i < (int)start.size() - 1 && recalc_dist(true); ++i) {
114     }
115     return i == (int)start.size() - 1;
116 }/*rp*/
117 /*ly*/ pair<ll,/*ry*/ ll/*ly*/> /*ry*/ calc_flow(
118     int _source, int _sink) {
119     source = _source;
120     sink = _sink;
121     assert(max(source, sink) < start.size() - 1);
122     ll tot_flow = 0;
123     ll tot_cost = 0;
124     /*lp*/ if (recalc_dist_bellman_ford()) {
125         assert(false);
126     } else {
127         /*ly*/ while (recalc_dist()) { /*ry*/
128             ll flow = 0;
129             while (dinic_bfs()) {
130                 now = start;
131                 ll cur;
132                 while (cur = dinic_dfs(source, INF)) flow += cur;
133             }
134             tot_flow += flow;
135             /*ly*/ tot_cost += sink_pot * flow; /*ry*/
136         }
137         return /*ly*/ { /*ry*/ tot_flow/*ly*/, tot_cost} /*ry*/;
138     }
139 }
140 ll flow_on_edge(int idx) {
141     assert(idx < cap.size());
142     return orig_cap[idx] - cap[cap_loc[idx]];
143 }
144 };
145 const int nmax = 1055;
146 int main(){
147     // arguments source and sink, memory usage 0(largest node index
148     // +input size)
149     int t;

```

```

150 scanf("%d", &t);
151 for (int i = 0; i < t; ++i) {
152     vector<tuple<int, int, ll, ll, ll>> edges;
153     int n;
154     scanf("%d", &n);
155     for (int j = 1; j <= n; ++j) {
156         edges.emplace_back(j, 2 * n + 1, 1, 0, 0);
157     }
158     for (int j = 1; j <= n; ++j) {
159         int card;
160         scanf("%d", &card);
161         edges.emplace_back(0, card, 1, 0, 0);
162     }
163     int ex_c;
164     scanf("%d", &ex_c);
165     for (int j = 0; j < ex_c; ++j) {
166         int a, b;
167         scanf("%d %d", &a, &b);
168         if (b < a) swap(a, b);
169         edges.emplace_back(a, b, nmax, 0, 1);
170         edges.emplace_back(b, n + b, nmax, 0, 0);
171         edges.emplace_back(n + b, a, nmax, 0, 1);
172     }
173     int v = 2 * n + 2;
174     MaxFlowmf(edges);
175     printf("%d\n", (int)mf.calc_flow(0, v - 1).second);
176 // cout << mf.flow_on_edge(edge_index) << endl;
177 }

#1577
#5057
#5123
#0927
#7358
#0078
#7871
#3940
#3693
#5398

```

## 12 Min Cost Max Flow with Cycle Cancelling $\mathcal{O}(\text{cap} \cdot nm)$

```

1 struct Network {
2     struct Node;
3     struct Edge {
4         Node *u, *v;
5         int f, c, cost;
6         Node*from(Node* pos{ #2965
7             if (pos == u) return v;
8             return u;
9         }
10        int getCap(Node* pos { #4145
11            if (pos == u) return c - f;
12            return f;
13        }
14        int addFlow(Node* pos, int toAdd) { #6369
15            if (pos == u)
16                f += toAdd;
17                return toAdd * cost;
18            } else {
19                f -= toAdd;
20                return -toAdd * cost
21            }
22        }

#8987

```

```

74     valid = true;
75     vector<Edge*> path;
76     int cap = 1000000000;
77     Node* cur = &nodes[i];
78     int clev = n
79     vector<bool> expr(n, false);
80     while (!expr[cur->index]) {
81         expr[cur->index] = true;
82         State cstate = state[clev][cur->index];
83         cur = cstate.used->from(cur)
84         path.push_back(cstate.used);
85     }
86     reverse(path.begin(), path.end());
87     {
88         int i = 0
89         Node* cur2 = cur;
90         do {
91             cur2 = path[i]->from(cur2);
92             i++;
93         } while (cur2 != cur)
94         path.resize(i);
95     }
96     for (auto edge : path) {
97         cap = min(cap, edge->getCap(cur));
98         cur = edge->from(cur)
99     }
100    for (auto edge : path) {
101        result += edge->addFlow(cur, cap);
102        cur = edge->from(cur);
103    }
104    if (!valid) break;
105 }
106 return result;
107 #4029


---


13 DMST  $\mathcal{O}(E \log V)$ 
1 struct EdgeDesc {
2     int from, to, w;
3 };
4 struct DMST {
5     struct Node
6     struct Edge {
7         Node *from;
8         Node *tar;
9         int w;
10        bool inc
11    };
12    struct Circle {
13        bool vis = false;
14        vector<Edge *> cont;
15        voidclean(int idx
#6091
#2186
#4353
#6663
#3984
#9784
#9838
#8867
#4467
#4029
#6612
#7005
#9916
#0564
#0300
#0747
#3927
#2561
#8600
#6612
#7005
};

16     };
17     const static greater<pair<ll, Edge *>> comp;
18     static vector<Circle> to_proc;
19     static bool no_dmst;
20     static Node *root; // Can use inline static since C++17
21     struct Node
22     {
23         Node *par = NULL;
24         vector<pair<int, int>> out_cands; // Circ, edge idx
25         vector<pair<ll, Edge *>> con;
26         bool in_use = false;
27         ll w = 0; // extra to add to edges in con
28         Node anc()
29         {
30             if (!par) return this;
31             while (par->par) par = par->par;
32             return par;
33         }
34         voidclean()
35         {
36             if (!no_dmst) {
37                 in_use = false;
38                 for (auto &cur : out_cands)
39                     to_proc[cur.first].clean(cur.second);
40             }
41             Node con_to_root(){
42                 if (anc() == root) return root;
43                 in_use = true;
44                 Node *super = this; // Will become root or the first Node
45                                         // encountered in a loop.
46                 while (super == this)
47                     while (
48                         !con.empty() && con.front().second->tar->anc() == anc() {
49                             pop_heap(con.begin(), con.end(), comp);
50                             con.pop_back();
51                         if (con.empty()) {
52                             no_dmst = true;
53                             return root;
54                         }
55                         pop_heap(con.begin(), con.end(), comp)
56                         auto nxt = con.back();
57                         con.pop_back();
58                         w = -nxt.first;
59                         if (nxt.second->tar
60                             ->in_use) { // anc() wouldn't change anything
61                             super = nxt.second->tar->anc()
62                             to_proc.resize(to_proc.size() + 1);
63                         } else {
64                             super = nxt.second->tar->con_to_root();
65                         }
66                         if (super != root)
67                             to_proc.back().cont.push_back(nxt.second);
68                     }
69             }
70         }
71     };
72 }
```

```

67     out_cands.emplace_back(
68         to_proc.size() - 1, to_proc.back().cont.size() - 1);
69 } else { // Clean circles
70     nxt.second->inc = true; #1096
71     nxt.second->from->clean();
72 }
73 if (super != root) { // we are some loops non first Node.
74     if (con.size() > super->con.size()) #2844
75         swap(con,
76             super->con); // Largest con in loop should not be copied.
77     swap(w, super->w);
78 }
79 for (auto cur : con) #3498
80     super->con.emplace_back(
81         cur.first - super->w + w, cur.second);
82     push_heap(super->con.begin(), super->con.end(), comp);
83 }
84 } #6348
85 par = super; // root or anc() of first Node encountered in a
86         // loop
87 return super;
88 }
89 Node *croot #0309
90 vector<Node> graph;
91 vector<Edge> edges;
92 DMST(int n, vector<EdgeDesc> &desc,
93       int r) { // Self loops and multiple edges are okay.
94     graph.resize(n) #8100
95     croot = &graph[r];
96     for (auto &cur : desc) // Edges are reversed internally
97         edges.push_back(Edge{&graph[cur.to], &graph[cur.from], cur.w});
98     for (int i = 0; i < desc.size(); ++i)
99         graph[desc[i].to].con.emplace_back(desc[i].w, &edges[i]);
100    for (int i = 0; i < n; ++i) #8811
101        make_heap(graph[i].con.begin(), graph[i].con.end(), comp);
102    }
103 bool find() {
104     root = croot;
105     no_dmst = false #5307
106     for (auto &cur : graph) {
107         cur.con_to_root();
108         to_proc.clear();
109         if (no_dmst) return false;
110     }
111     return true;
112 }
113 ll weight({ #1568
114     ll res = 0;
115     for (auto &cur : edges) {
116         if (cur.inc) res += cur.w;
117     }
118 }
```

```

119     return res; #6369
120 }
121 }
122 void DMST::Circle::clean(int idx) { %1477
123     if (!vis) {
124         vis = true;
125         for (int i = 0; i < cont.size(); ++i) #6503
126             if (i != idx) {
127                 cont[i]->inc = true;
128                 cont[i]->from->clean();
129             }
130     }
131 }
132 }
133 }
134 const greater<pair<ll, DMST::Edge *>> DMST::comp; #8144
135 vector<DMST::Circle> DMST::to_proc;
136 bool DMST::no_dmst #2354
```

## 14 Bridges $\mathcal{O}(n)$

```

1 struct vert;
2 struct edge {
3     bool exists = true;
4     vert *dest;
5     edge *rev;
6     edge(vert *_dest) : dest(_dest) { rev = NULL; } #8922
7     vert &operator*() { return *dest; }
8     vert *operator->() { return dest; }
9     bool is_bridge(); #0116
10 }
11 struct vert {
12     deque<edge> con;
13     int val = 0;
14     int seen;
15     inttdfs(int upd, edge *ban){ // handles multiple edges #1288
16         if (!val)
17             val = upd;
18             seen = val;
19             for (edge &nxt : con) {
20                 if (nxt.exists && (&nxt) != ban)
21                     seen = min(seen, nxt->dfs(upd + 1, nxt.rev)); #8194
22             }
23     }
24     return seen; #8624
25 }
26 void remove_adj_bridges({ #7106
27     for (edge &nxt : con) {
28         if (nxt.is_bridge()) nxt.exists = false;
29     }
30 }
31 intcnt_adj_bridges({ #7106
32 }
```

```

32     int res = 0;
33     for (edge &nxt : con) res += nxt.is_bridge();
34     return res;
35   };
36   bool edge::is_bridge() {
37     return exists &&
38       (dest->seen > rev->dest->val || dest->val < rev->dest->seen);
39     #5223
40   } %5223
41 vert graph[nmax];
42 intmain{// Mechanics Practice BRIDGES
43   int n, m;
44   cin >> n >> m;
45   for (int i = 0; i < m; ++i) {
46     int u, v;
47     scanf("%d %d", &u, &v);
48     graph[u].con.emplace_back(graph + v);
49     graph[v].con.emplace_back(graph + u);
50     graph[u].con.back().rev = &graph[v].con.back();
51     graph[v].con.back().rev = &graph[u].con.back();
52   }
53   graph[1].dfs(1, NULL);
54   int res = 0;
55   for (int i = 1; i <= n; ++i) res += graph[i].cnt_adj_bridges();
56   cout << res / 2 << endl;

```

### 15 2-Sat $\mathcal{O}(n)$ and SCC $\mathcal{O}(n)$

```

1 struct Graph {
2   int n;
3   vector<vector<int> > con;
4   Graph(int nsize) {
5     n = nsize
6     con.resize(n);
7   }
8   void add_edge(int u, int v) { con[u].push_back(v); }
9   void top_dfs(int pos, vector<int> &result, vector<bool> &explr,
10    vector<vector<int> > &revcon) #2422
11   if (explr[pos]) return;
12   explr[pos] = true;
13   for (auto next : revcon[pos])
14     top_dfs(next, result, explr, revcon);
15   result.push_back(pos)
16   #2081
17   %7763
18   vector<int> topsort() {
19     vector<vector<int> > revcon(n);
20     ran(u, 0, n) {
21       for (auto v : con[u]) revcon[v].push_back(u);
22     }
23     vector<int> result;
24     vector<bool> explr(n, false);
25     ran(i, 0, n) top_dfs(i, result, explr, revcon);
26     reverse(result.begin(), result.end());
#3875

```

```

26   return result
27
28   void ddfs(int pos, vector<int> &result, vector<bool> &explr{ #7568
29     if (explr[pos]) return;
30     explr[pos] = true;
31     for (auto next : con[pos]) ddfs(next, result, explr);
32     result.push_back(pos)
33
34   vector<vector<int> > scc() { #5339
35     vector<int> order = topsort();
36     reverse(order.begin(), order.end());
37     vector<bool> explr(n, false);
38     vector<vector<int> > res
39     for (auto it = order.rbegin(); it != order.rend(); ++it) { #6880
40       vector<int> comp;
41       top_dfs(*it, comp, explr, con);
42       sort(comp.begin(), comp.end());
43       res.push_back(comp)
44     }
45     sort(res.begin(), res.end());
46     return res;
47   }
48 } #0543 %0543
49 intmain{
50   int n, m;
51   cin >> n >> m;
52   Graphg(2 * m);
53   ran(i, 0, n) {
54     int a, sa, b, sb;
55     cin >> a >> sa >> b >> sb;
56     a--, b--;
57     g.add_edge(2 * a + 1 - sa, 2 * b + sb);
58     g.add_edge(2 * b + 1 - sb, 2 * a + sa);
59   }
60   vector<int> state(2 * m, 0);
61   {
62     vector<int> order = g.topsort();
63     vector<bool> explr(2 * m, false);
64     for (auto u : order) {
65       vector<int> traversed;
66       g.dfs(u, traversed, explr);
67       if (traversed.size() > 0 && !state[traversed[0] ^ 1]) {
68         for (auto c : traversed) state[c] = 1;
69       }
70     }
71   }
72   ran(i, 0, m) {
73     if (state[2 * i] == state[2 * i + 1]) {
74       cout << "IMPOSSIBLE\n";
75       return 0;
76     }

```

```

77 }
78 ran(i, 0, m) cout << state[2 * i + 1] << '\n';
79 return 0;


---


16 Generic persistent compressed lazy segment tree


---


1 struct Seg {
2     ll sum = 0;
3     void recalc(const Seg &lhs_seg, int lhs_len, const Seg &rhs_seg,
4             int rhs_len{
5         sum = lhs_seg.sum + rhs_seg.sum
6     } #7684
7 } __attribute__((packed));
8 struct Lazy {
9     ll add;
10    ll assign_val; // LLONG_MIN if no assign;
11    void init()
12        add = 0;
13        assign_val = LLONG_MIN;
14    }
15 Lazy() { init(); }
16 void split(Lazy &lhs_lazy, Lazy &rhs_lazy, int len) {
17     lhs_lazy = *this
18     rhs_lazy = *this;
19     init();
20 }
21 void merge(Lazy &oth, int len) {
22     if (oth.assign_val != LLONG_MIN)
23         add = 0;
24         assign_val = oth.assign_val;
25     }
26     add += oth.add;
27 }
28 void apply_to_seg(Seg &cur, int len const{
29     if (assign_val != LLONG_MIN) {
30         cur.sum = len * assign_val;
31     }
32     cur.sum += len * add
33 }
34 } __attribute__((packed)); %0625 struct Node { // Following code should
35     ~ not need to be modified
36     int ver;
37     bool is_lazy = false;
38     Seg seg;
39     Lazy lazy
40     Node *lc = NULL, *rc = NULL;
41     void init(){
42         if (!lc)
43             lc = new Node{ver};
44             rc = new Node{ver}
45     }
46     Node upd(int L, int R, int l, int r, Lazy &val, int tar_ver{
#7883
#7654
#0050
#2924
#6280
#6321
#5313
#8874
#2138
#8209
#8104
#8581
#9373
#6654
#2185
#4770
#4470
#96
#97
#16

```

```

98 int len #4873
99 SegTree(int _len) : len(_len) { roots.push_back(new Node{0}); }
100 int upd(int l, int r, Lazy &val, bool new_ver = false) {
101     Node *cur_root =
102         roots.back()->upd(0, len, l, r, val, roots.size() - !new_ver);
103     if (cur_root != roots.back()) roots.push_back(cur_root);
104     return roots.size() - 1 #1461
105 }
106 Seg get(int l, int r, int ver = -1) {
107     if (ver == -1) ver = roots.size() - 1;
108     Seg seg1, seg2;
109     Seg *pres = &seg1, *ptmp = &seg2 #9427
110     roots[ver]->get(0, len, l, r, pres, ptmp, roots.size() - 1);
111     return *pres;
112 }
113 }; %7542 intmain(){
114 int n, m; // solves Mechanics Practice LAZY
115 cin >> n >> m;
116 SegTree seg_tree(1 << 17);
117 for (int i = 0; i < n; ++i) {
118     Lazy tmp;
119     scanf("%lld", &tmp.assign_val);
120     seg_tree.upd(i, i + 1, tmp);
121 }
122 for (int i = 0; i < m; ++i) {
123     int o;
124     int l, r;
125     scanf("%d %d %d", &o, &l, &r);
126     --l;
127     if (o == 1) {
128         Lazy tmp;
129         scanf("%lld", &tmp.add);
130         seg_tree.upd(l, r, tmp);
131     } else if (o == 2) {
132         Lazy tmp;
133         scanf("%lld", &tmp.assign_val);
134         seg_tree.upd(l, r, tmp);
135     } else {
136         Seg res = seg_tree.get(l, r);
137         printf("%lld\n", res.sum);
138     }
139 }



---


17 Templatized HLD  $\mathcal{O}(M(n)\log n)$  per query


---


1 class dummy {
2 public:
3     dummy() {}
4     dummy(int, int) {}
5     void set(int, int) {} #9531
6     intquery(int left, int right{
7         cout << this << ' ' << left << ' ' << right << endl;
8     }
9     %7932 /*  $T$  should be the type of the data stored in each vertex;
10    *  $DS$  should be the underlying data structure that is used to perform
11    * the group operation. It should have the following methods:
12    * *  $DS()$  - empty constructor
13    * *  $DS(int size, T initial)$  - constructs the structure with the
14    * given size, initially filled with initial.
15    * *  $void set(int index, T value)$  - set the value at index `index` to
16    * `value`
17    * *  $T query(int left, int right)$  - return the "sum" of elements
18    * between left and right, inclusive.
19 */
20 template <typename T, class DS>
21 class HLD {
22     int vertexc;
23     vector<int> *adj;
24     vector<int> subtree_size #6178
25     DS structure;
26     DS aux;
27     void build_sizes(int vertex, int parent{
28         subtree_size[vertex] = 1;
29         for (int child : adj[vertex]) #2037
30             if (child != parent) {
31                 build_sizes(child, vertex);
32                 subtree_size[vertex] += subtree_size[child];
33             }
34     } #6759
35     int cur;
36     vector<int> ord;
37     vector<int> chain_root;
38     vector<int> par #9593
39     void build_hld(int vertex, int parent, int chain_source{
40         cur++;
41         ord[vertex] = cur;
42         chain_root[vertex] = chain_source;
43         par[vertex] = parent #0432
44         if (adj[vertex].size() > 1 ||
45             (vertex == 1 && adj[vertex].size() == 1)) {
46             int big_child, big_size = -1;
47             for (int child : adj[vertex]) {
48                 if ((child != parent) && (subtree_size[child] > big_size)) {
49                     big_child = child #9151
50                     big_size = subtree_size[child];
51                 }
52             }
53             build_hld(big_child, vertex, chain_source);
54             for (int child : adj[vertex]) #3027
55                 if ((child != parent) && (child != big_child))
56                     build_hld(child, vertex, child);
57             }
58         }
59     }

```

```

60
61     public:
62     HLD(int _vertexc) {
63         vertexc = _vertexc;
64         adj = new vector<int>[vertexc + 5];
65
66     void add_edge(int u, int v{
67         adj[u].push_back(v);
68         adj[v].push_back(u);
69     }
70     void build(T initial
71     subtree_size = vector<int>(vertexc + 5);
72     ord = vector<int>(vertexc + 5);
73     chain_root = vector<int>(vertexc + 5);
74     par = vector<int>(vertexc + 5);
75     cur = 0
76     build_sizes(1, -1);
77     build_hld(1, -1, 1);
78     structure = DS(vertexc + 5, initial);
79     aux = DS(50, initial);
80
81     void set(int vertex, int value{
82         structure.set(ord[vertex], value);
83     }
84     T query_path(
85         int u, int v/* returns the "sum" of the path u->v */
86         int cur_id = 0
87         while (chain_root[u] != chain_root[v]) {
88             if (ord[u] > ord[v]) {
89                 cur_id++;
90                 aux.set(cur_id, structure.query(ord[chain_root[u]], ord[u]));
91                 u = par[chain_root[u]] #4538
92             } else {
93                 cur_id++;
94                 aux.set(cur_id, structure.query(ord[chain_root[v]], ord[v]));
95                 v = par[chain_root[v]];
96             }
97             cur_id++;
98             aux.set(cur_id,
99                 structure.query(min(ord[u], ord[v]), max(ord[u], ord[v])));
100            return aux.query(1, cur_id) #7150
101            %1905
102
103    void print(){
104        for (int i = 1; i <= vertexc; i++)
105            cout << i << ' ' << ord[i] << ' ' << chain_root[i] << ' '
106            << par[i] << endl;
107    }
108 };
109 int main(){
110     int vertexc;
111     cin >> vertexc;

```

```

#8562
112     HLD<int, dummy> hld(vertexc);
113     for (int i = 0; i < vertexc - 1; i++) {
114         int u, v;
115         cin >> u >> v;
116         hld.add_edge(u, v);
117     }
118     hld.build();
119     hld.print();
120     int queryc;
121     cin >> queryc;
122     for (int i = 0; i < queryc; i++) {
123         int u, v;
124         cin >> u >> v;
125         hld.query_path(u, v);
126         cout << endl;
127     }

#3486
#4566
#2693
#7758
#4754
#1595
#7150
#1905
#8505
#7303
#18
18  Tempered multi dimensional BIT  $\mathcal{O}(\log(n)^{\dim})$  per query
1 // Fully overloaded any dimensional BIT, use any type for coordinates,
2 // elements, return_value. Includes coordinate compression.
3 template <typename E_T, typename C_T, C_T n_inf, typename R_T>
4 struct BIT {
5     vector<C_T> pos;
6     vector<E_T> elems;
7     bool act = false
8     BIT() { pos.push_back(n_inf); } #3273
9     void init() {
10         if (act) {
11             for (E_T &c_elem : elems) c_elem.init();
12         } else
13             act = true;
14             sort(pos.begin(), pos.end());
15             pos.resize(unique(pos.begin(), pos.end()) - pos.begin());
16             elems.resize(pos.size()); #2594
17     }
18     template <typename... loc_form>
19     void update(C_T cx, loc_form... args) {
20         if (act) {
21             int x = lower_bound(pos.begin(), pos.end(), cx) - pos.begin();
22             for (; x < (int)pos.size(); x += x & -x) #7303
23                 elems[x].update(args...);
24         } else {
25             pos.push_back(cx);
26         }
27     }
28     template <typename... loc_form>
29     R_T query(C_T cx, loc_form... args) { // sum in (-inf, cx)
30         R_T res = 0;
31         int x = lower_bound(pos.begin(), pos.end(), cx) - pos.begin() - 1;
32         for (; x > 0; x -= x & -x) res += elems[x].query(args...);
33     }

```

```

34     return res
35   }
36 };
37 template <typename I_T>
38 struct wrapped {
39   I_T a = 0
40   void update(I_T b{ a += b; })
41   I_T query({ return a; })
42   // Should never be called, needed for compilation
43   void init({ DEBUG'i' })
44   void update() { DEBUG'u' }
45 } #2858
46 // return type should be same as type inside wrapped
47 BIT<BIT<wrapped<ll>, int, INT_MIN, ll>, int, INT_MIN, ll> fenwick;
48 int dim = 2;
49 vector<tuple<int, int, ll> > to_insert;
50 to_insert.emplace_back(1, 1, 1);
51 // set up all pos that are to be used for update
52 for (int i = 0; i < dim; ++i) {
53   for (auto &cur : to_insert)
54     fenwick.update(get<0>(cur), get<1>(cur));
55   // May include value which won't be used
56   fenwick.init();
57 }
58 // actual use
59 for (auto &cur : to_insert)
60   fenwick.update(get<0>(cur), get<1>(cur), get<2>(cur));
61 cout << fenwick.query(2, 2) << '\n';

```

### 19 Treap $\mathcal{O}(\log n)$ per query

```

1 mt19937 randgen;
2 struct Treap {
3   struct Node {
4     int key;
5     int value
6     unsigned int priority;
7     long long total;
8     Node* lch;
9     Node* rch;
10    Node(int new_key, int new_value)
11      key = new_key;
12      value = new_value;
13      priority = randgen();
14      total = new_value;
15      lch = 0
16      rch = 0;
17  }
18  void update() {
19    total = value;
20    if (lch) total += lch->total
21    if (rch) total += rch->total;
22  }

```

```

#2526
23   };
24   deque<Node> nodes;
25   Node* root = 0
26   pair<Node*, Node*> split(int key, Node* cur) {
27     if (cur == 0) return {0, 0};
28     pair<Node*, Node*> result;
29     if (key <= cur->key) {
30       auto ret = split(key, cur->lch)
31       cur->lch = ret.second;
32       result = {ret.first, cur};
33     } else {
34       auto ret = split(key, cur->rch);
35       cur->rch = ret.first
36       result = {cur, ret.second};
37     }
38     cur->update();
39     return result;
40   }
41   Node* merge(Node* left, Node* right{
42     if (left == 0) return right;
43     if (right == 0) return left;
44     Node* top;
45     if (left->priority < right->priority)
46       left->rch = merge(left->rch, right);
47       top = left;
48     } else {
49       right->lch = merge(left, right->lch);
50       top = right
51     }
52     top->update();
53     return top;
54   }
55   void insert(int key, int value) #8918
56     nodes.push_back(Node(key, value));
57     Node* cur = &nodes.back();
58     pair<Node*, Node*> ret = split(key, root);
59     cur = merge(ret.first, cur);
60     cur = merge(cur, ret.second)
61     root = cur;
62   }
63   void erase(int key) {
64     Node *left, *mid, *right;
65     tie(left, mid) = split(key, root)
66     tie(mid, right) = split(key + 1, mid);
67     root = merge(left, right);
68   }
69   long long sum_upto(int key, Node* cur) { #7634
70     if (cur == 0) return 0
71     if (key <= cur->key) {
72       return sum_upto(key, cur->lch);
73     } else {

```

```

74     long long result = cur->value + sum_upto(key, cur->rch);
75     if (cur->lch) result += cur->lch->total
76     return result;
77 }
78 }
79 long long get(int l, int r) {
80     return sum_upto(r + 1, root) - sum_upto(l, root);
81 } #0094 %4959 // Solution for:
82 // http://codeforces.com/group/U01GDa2Gwb/contest/219104/problem/TREAP
83 intmain({
84     ios_base::sync_with_stdio(false);
85     cin.tie(0);
86     int m;
87     Treap treap;
88     cin >> m;
89     for (int i = 0; i < m; i++) {
90         int type;
91         cin >> type;
92         if (type == 1) {
93             int x, y;
94             cin >> x >> y;
95             treap.insert(x, y);
96         } else if (type == 2) {
97             int x;
98             cin >> x;
99             treap.erase(x);
100        } else {
101            int l, r;
102            cin >> l >> r;
103            cout << treap.get(l, r) << endl;
104        }
105    }
106 } #5681
107 return 0;

```

## 20 Radixsort 50M 64 bit integers as single array in 1 sec

```

1 template <typename T>
2 void rsort(T *a, T *b, int size, int d = sizeof(T) - 1) {
3     int b_s[256]{};
4     ran(i, 0, size) { ++b_s[(a[i] >> (d * 8)) & 255]; }
5     // ++b_s[*((uchar *) (a + i) + d)]; #5369
6     T *mem[257];
7     mem[0] = b;
8     T **l_b = mem + 1;
9     l_b[0] = b;
10    ran(i, 0, 255) { l_b[i + 1] = l_b[i] + b_s[i]; }
11    for (T *it = a; it != a + size; ++it) #6813
12        T id = ((*it) >> (d * 8)) & 255;
13        *(l_b[id]++) = *it;
14    }
15    l_b = mem;
16    if (d)

```

```

17    T *l_a[256];
18    l_a[0] = a;
19    ran(i, 0, 255) l_a[i + 1] = l_a[i] + b_s[i];
20    ran(i, 0, 256) { #1162
21        if (l_b[i + 1] - l_b[i] < 100)
22            sort(l_b[i], l_b[i + 1]);
23            if (d & 1) copy(l_b[i], l_b[i + 1], l_a[i]);
24        } else {
25            rsort(l_b[i], l_a[i], b_s[i], d - 1); #7759
26        }
27    }
28 }
29 const int nmax = 5e7;
30 ll arr[nmax], tmp[nmax];
31 intmain({
32     for (int i = 0; i < nmax; ++i) arr[i] = ((ll)rand() << 32) | rand(); #0571
33     rsort(arr, tmp, nmax);
34     assert(is_sorted(arr, arr + nmax));

```

## 21 FFT 5M length/sec

integer  $c = a * b$  is accurate if  $c_i < 2^{49}$

```

1 struct Complex {
2     double a = 0, b = 0;
3     Complex &operator/=(const int &oth) {
4         a /= oth;
5         b /= oth; #1139
6         return *this;
7     }
8 };
9 Complex operator+(const Complex &lft, const Complex &rgt) {
10    return Complex{lft.a + rgt.a, lft.b + rgt.b}; #8384
11 }
12 Complex operator-(const Complex &lft, const Complex &rgt) {
13    return Complex{lft.a - rgt.a, lft.b - rgt.b}; #5371
14 }
15 Complex operator*(const Complex &lft, const Complex &rgt) {
16    return Complex #7637
17        lft.a * rgt.a - lft.b * rgt.b, lft.a * rgt.b + lft.b * rgt.a};
18 }
19 Complex conj(const Complex &cur) { return Complex{cur.a, -cur.b}; }
20 void fft_rec(Complex *arr, Complex *root_pow, int len) { #0670
21     if (len != 1) #0670
22         fft_rec(arr, root_pow, len >> 1);
23         fft_rec(arr + len, root_pow, len >> 1);
24     }
25     root_pow += len;
26     for (int i = 0; i < len; ++i)
27         Complex tmp = arr[i] + root_pow[i] * arr[i + len];
28         arr[i + len] = arr[i] - root_pow[i] * arr[i + len];
29         arr[i] = tmp;

```

```

30 }
31
32 void fft(vector<Complex> &arr, int ord, bool invert{
33     assert(arr.size() == 1 << ord);
34     static vector<Complex> root_pow(1);
35     static int inc_pow = 1;
36     static bool is_inv = false
37     if (inc_pow <= ord) {
38         int idx = root_pow.size();
39         root_pow.resize(1 << ord);
40         for (; inc_pow <= ord; ++inc_pow) {
41             for (int idx_p = 0; idx_p < 1 << (ord - 1) #3349
42                 idx_p += 1 << (ord - inc_pow), ++idx) {
43                 root_pow[idx] = Complex{cos(-idx_p * M_PI / (1 << (ord - 1))), #7078
44                     sin(-idx_p * M_PI / (1 << (ord - 1)))};
45                 if (is_inv) root_pow[idx].b = -root_pow[idx].b;
46             }
47         }
48     }
49     if (invert != is_inv) {
50         is_inv = invert;
51         for (Complex &cur : root_pow) cur.b = -cur.b #7526
52     }
53     for (int i = 1, j = 0; i < (1 << ord); ++i) {
54         int m = 1 << (ord - 1);
55         bool cont = true;
56         while (cont) #0510
57             cont = j & m;
58             j ^= m;
59             m >>= 1;
60         if (i < j) swap(arr[i], arr[j]) #0506
61     }
62     fft_rec(arr.data(), root_pow.data(), 1 << (ord - 1));
63     if (invert)
64         for (int i = 0; i < (1 << ord); ++i) arr[i] /= (1 << ord); #4380
65     #4380
66 void mult_poly_mod()
67     vector<int> &a, vector<int> &b, vector<int> &c{// c += a*b
68     static vector<Complex>
69         arr[4];// correct upto 0.5-2M elements(mod ~ 1e9)
70     if (c.size() < 400) #8811
71         for (int i = 0; i < a.size(); ++i)
72             for (int j = 0; j < b.size() && i + j < c.size(); ++j)
73                 c[i + j] = ((ll)a[i] * b[j] + c[i + j]) % mod;
74     } else {
75         int fft_ord = 32 - __builtin_clz(c.size())
76         if (arr[0].size() != 1 << fft_ord) #4629
77             for (int i = 0; i < 4; ++i) arr[i].resize(1 << fft_ord);
78         for (int i = 0; i < 4; ++i)
79             fill(arr[i].begin(), arr[i].end(), Complex{});
80         for (int &cur : a
81             #9591

```

```

82         if (cur < 0) cur += mod;
83         for (int &cur : b)
84             if (cur < 0) cur += mod;
85         const int shift = 15;
86         const int mask = (1 << shift) - 1 #2625
87         for (int i = 0; i < min(a.size(), c.size()); ++i) {
88             arr[0][i].a = a[i] & mask;
89             arr[1][i].a = a[i] >> shift;
90         }
91         for (int i = 0; i < min(b.size(), c.size()); ++i) #3501
92             arr[0][i].b = b[i] & mask
93             arr[1][i].b = b[i] >> shift;
94         }
95         for (int i = 0; i < 2; ++i) fft(arr[i], fft_ord, false);
96         for (int i = 0; i < 2; ++i) {
97             for (int j = 0; j < 2; ++j) #9971
98                 int tar = 2 + (i + j) / 2;
99                 Complex mult = {0, -0.25};
100                if (i ^ j) mult = {0.25, 0};
101                for (int k = 0; k < (1 << fft_ord); ++k) {
102                    int rev_k = ((1 << fft_ord) - k) % (1 << fft_ord);
103                    Complex ca = arr[i][k] + conj(arr[i][rev_k]);
104                    Complex cb = arr[j][k] - conj(arr[j][rev_k]);
105                    arr[tar][k] = arr[tar][k] + mult * ca * cb;
106                }
107            }
108        }
109        for (int i = 2; i < 4; ++i) #4471
110            fft(arr[i], fft_ord, true);
111            for (int k = 0; k < (int)c.size(); ++k) #8403
112                c[k] = (c[k] + (((ll)(arr[i][k].a + 0.5) % mod)
113                                << (shift * 2 * (i - 2)))) %
114                                mod;
115                c[k] = (c[k] + (((ll)(arr[i][k].b + 0.5) % mod)
116                                << (shift * (2 * (i - 2) + 1)))) %
117                                mod
118            }
119        }
120    }

```

## 22 Fast mod mult, Rabin Miller prime check, Pollard rho factorization $\mathcal{O}(\sqrt{p})$

```

1 struct ModArithm {
2     ull n;
3     ld rec;
4     ModArithm(ull _n) : n(_n) {// n in [2, 1<<63)
5         rec = 1.0L / n #0237
6     }
7     ull multf(ull a, ull b) {// a, b in [0, min(2*n, 1<<63))
8         ull mult = (ld)a * b * rec + 0.5L;
9         ll res = a * b - mult * n;

```

```

10     if (res < 0) res += n
11     return res;// in [0, n-1)
12 }
13 ull sqp1(ull a) { return multf(a, a) + 1; }
14 }
15 ull pow_mod(ull a, ull n, ModArithm &arithm{
16     ull res = 1;
17     for (ull i = 1; i <= n; i <= 1) {
18         if (n & i) res = arithm.multf(res, a);
19         a = arithm.multf(a, a)
20     }
21     return res;
22 }
23 vector<char> small_primes = {
24     2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
25 bool is_prime(ull n{// n <= 1<<63, 1M rand/s
26     ModArithm arithm(n);
27     if (n == 2 || n == 3) return true;
28     if (!(n & 1) || n == 1) return false;
29     ull s = __builtin_ctz(n - 1);
30     ull d = (n - 1) >> s;
31     for (ull a : small_primes) {
32         if (a >= n) break;
33         a = pow_mod(a, d, arithm);
34         if (a == 1 || a == n - 1) continue;
35         for (ull r = 1; r < s; ++r) {
36             a = arithm.multf(a, a);
37             if (a == 1) return false;
38             if (a == n - 1) break;
39         }
34         if (a != n - 1) return false;
41     }
42     return true;
43 }
44 ll pollard_rho(ll n{
45     ModArithm arithm(n);
46     int cum_cnt = 64 - __builtin_clz(n);
47     cum_cnt *= cum_cnt / 5 + 1;
48     while (true)
49         ll lv = rand() % n;
50         ll v = arithm.sqp1(lv);
51         int idx = 1;
52         int tar = 1;
53         while (true)
54             ll cur = 1;
55             ll v_cur = v;
56             int j_stop = min(cum_cnt, tar - idx);
57             for (int j = 0; j < j_stop; ++j) {
58                 cur = arithm.multf(cur, abs(v_cur - lv));
59                 v_cur = arithm.sqp1(v_cur);
60                 ++idx;
61             }
62             if (cur == 1) return 1;
63             if (cur == v) return 0;
64             v = cur;
65         }
66     }
67 }
```

```

#0780
61     }
62     if (!cur) {
63         for (int j = 0; j < cum_cnt; ++j)
64             ll g = __gcd(abs(v - lv), n);
65             if (g == 1) {
66                 v = arithm.sqp1(v);
67             } else if (g == n) {
68                 break;
69             } else {
70                 return g;
71             }
72     }
73     break;
74 } else {
75     ll g = __gcd(cur, n);
76     if (g != 1) return g;
77 }
78 v = v_cur
79 idx += j_stop;
80 if (idx == tar) {
81     lv = v;
82     tar *= 2;
83     v = arithm.sqp1(v)
84     ++idx;
85 }
86 }
87 }

#0876                                         #3542
88 map<ll, int> prime_factor(ll n,
89 map<ll, int> *res = NULL) { // n <= 1<<61, ~1000/s (<500/s on CF)
90 if (!res) {
91     map<ll, int> res_act;
92     for (int p : small_primes)
93         while (!(n % p)) {
94             ++res_act[p];
95             n /= p;
96         }
97 }

#4806                                         #3770
%0975
98     if (n != 1) prime_factor(n, &res_act);
99     return res_act;
100 }

#2118                                         #4612
101     if (is_prime(n)) {
102         ++(*res)[n];
103     } else {
104         ll factor = pollard_rho(n);
105         prime_factor(factor, res);
106         prime_factor(n / factor, res);
107     }
108 }

#5290                                         #1963
109     return map<ll, int>();

```

## 23 Symmetric Submodular Functions; Queyrannes's algorithm

**SSF:** such function  $f : V \rightarrow R$  that satisfies  $f(A) = f(V/A)$  and for all  $x \in V, X \subseteq Y \subseteq V$  it holds that  $f(X+x) - f(X) \leq f(Y+x) - f(Y)$ . **Hereditary family:** such set  $I \subseteq 2^V$  so that  $X \subset Y \wedge Y \in I \Rightarrow X \in I$ . **Loop:** such  $v \in V$  so that  $v \notin I$ . breaklines

```

1 def minimize():
2     s = merge_all_loops()
3     while size >= 3:
4         t, u = find_pp()
5         {u} is a possible minimizer
6         tu = merge(t, u)
7         if tu not in I:
8             s = merge(tu, s)
9     for x in V:
10        {x} is a possible minimizer
11 def find_pp():
12     W = {s} # s as in minimizer()
13     todo = V/W
14     ord = []
15     while len(todo) > 0:
16         x = min(todo, key=lambda x: f(W+{x}) - f({x}))
17         W += {x}
18         todo -= {x}
19         ord.append(x)
20     return ord[-1], ord[-2]
21 def enum_all_minimal_minimizers(X):
22     # X is a inclusionwise minimal minimizer
23     s = merge(s, X)
24     yield X
25     for {v} in I:
26         if f({v}) == f(X):
27             yield X
28             s = merge(v, s)
29     while size(V) >= 3:
30         t, u = find_pp()
31         tu = merge(t, u)
32         if tu not in I:
33             s = merge(tu, s)
34         elif f({tu}) == f(X):
35             yield tu
36             s = merge(tu, s)

```

## 24 Berlekamp-Massey $O(\mathcal{LN})$

```

1 template <typename K>
2 static vector<K> berlekamp_massey(vector<K> ss) {
3     vector<K> ts(ss.size());
4     vector<K> cs(ss.size());
5     cs[0] = K::unity
6     fill(cs.begin() + 1, cs.end(), K::zero);
7     vector<K> bs = cs;
8     int l = 0, m = 1;

```

```

9     K b = K::unity;
10    for (int k = 0; k < (int)ss.size(); k++)
11        K d = ss[k];
12        assert(l <= k);
13        for (int i = 1; i <= l; i++) d += cs[i] * ss[k - i];
14        if (d == K::zero) {
15            m++;
16        } else if (2 * l <= k) {
17            K w = d / b;
18            ts = cs;
19            for (int i = 0; i < (int)cs.size() - m; i++)
20                cs[i + m] -= w * bs[i];
21            l = k + 1 - l;
22            swap(bs, ts);
23            b = d;
24            m = 1;
25        } else {
26            K w = d / b;
27            for (int i = 0; i < (int)cs.size() - m; i++)
28                cs[i + m] -= w * bs[i];
29            m++;
30        }
31    cs.resize(l + 1);
32    while (cs.back() == K::zero) cs.pop_back();
33    return cs;
34

```