

Competitive Programmer's Reference

Zhongtang Luo

October 22, 2023

MIT License

Copyright (c) 2023 Zhongtang Luo

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

1 Environment

1.1 Vimrc

2 Data Structure

2.1 RMQ

2.2 Link-Cut Tree

2.3 KD Tree

3 Geometry

3.1 2D-Geometry

3.1.1 Triangle Center

3.1.2 Fermat Point

3.1.3 Convex Hull

3.1.4 Half-Plane Intersection

1 Environment

1.1 Vimrc

```
1 set ru nu ts=4 sts=4 sw=4 si sm hls is ar bs=2
  mouse=a
2 syntax on
3 nm <F3> :vsplit %<.in <CR>
4 nm <F4> :!gedit % <CR>
5 au BufEnter *.cpp set cin
6 au BufEnter *.cpp nm <F5> :!time ./%< <CR>|nm <F7>
  :!gdb ./%< <CR>|nm <F8> :!time ./%< <CR>|nm <F9> :!g++ % -o %< -g -std=gnu++14 -O2
  -DLOCAL -Wall -Wconversion && size %< <CR>
7 au BufEnter *.java nm <F5> :!time java %< <CR>|nm
  <F8> :!time java %< <CR>|nm <F9> :!javac
  % <CR>
```

2 Data Structure

2.1 RMQ

```
1 for (int st = 1; st < 20; ++st)
2   for (int i = 0; i < N; ++i)
3     if (i + (1 << st - 1) < N)
4       rmq[st][i] = std::min(rmq[st - 1][i],
5                             rmq[st - 1][i + (1 << st - 1)]);
6 int len = 31 - __builtin_clz(r - 1 + 1);
7 return std::min(
8   rmq[len][l], rmq[len][r - (1 << len) + 1]);
```

2.2 Link-Cut Tree

```
1 struct Node { int son[2], fa, num, pos, rev;
2 } node[maxn]; int n, m, ans, top, q[maxn];
3 inline bool root(int x) {
4   return node[node[x].fa].son[0] != x &&
5         node[node[x].fa].son[1] != x; }
6 void update(int x) {
7   int left = node[x].son[0], right = node[x].son[1];
8   node[x].pos = x; if (node[node[left].pos].num >
9     node[node[x].pos].num)
10     node[x].pos = node[left].pos;
11   if (node[node[right].pos].num >
12     node[node[x].pos].num)
13     node[x].pos = node[right].pos; }
14 void down(int x) {
15   int left = node[x].son[0], right = node[x].son[1];
16   if (node[x].rev) { node[x].rev ^= 1;
17     node[left].rev ^= 1; node[right].rev ^= 1;
18     std::swap(node[x].son[0], node[x].son[1]); } }
19 void rotate(int x) {
```

```
20   int y = node[x].fa, z = node[y].fa, left, right;
21   if (node[y].son[0] == x) left = 0; else left = 1;
22   right = left ^ 1; if (!root(y)) {
23     if (node[z].son[0] == y) node[z].son[0] = x;
24     else node[z].son[1] = x; }
25   node[x].fa = z; node[y].fa = x;
26   if (node[x].son[right] != 0)
27     node[node[x].son[right]].fa = y;
28   node[y].son[left] = node[x].son[right];
29   node[x].son[right] = y; update(y); update(x); }
30 void splay(int x) { top = 0; q[++top] = x;
31   for (int i = x; !root(i); i = node[i].fa)
32     q[++top] = node[i].fa;
33   for (int i = top; i; i--) down(q[i]);
34   while (!root(x)) {
35     int y = node[x].fa, z = node[y].fa;
36     if (!root(y)) {
37       if (node[y].son[0] == x ^ node[z].son[0] == y)
38         rotate(x); else rotate(y); } rotate(x); }
39   update(x); }
40 void access(int x) { int t = 0; while (x) {
41   splay(x); node[x].son[1] = t; t = x;
42   x = node[x].fa; } }
43 void makeroot(int x) { access(x); splay(x);
44   node[x].rev ^= 1; }
45 void link(int x, int y) { makeroot(x);
46   node[x].fa = y; }
47 void cut(int x, int y) { makeroot(x); access(y);
48   splay(y); node[node[y].son[0]].fa = 0;
49   node[y].son[0] = 0; update(y); }
```

2.3 KD Tree

Find the k -th closest/farthest point in $O(kn^{1-\frac{1}{k}})$.

Usage: 1. Store the data in `p[]`. 2. Execute `init`. 3. Execute `min_kth` or `max_kth` for queries (k is 1-based).

Note: Switch to the commented code for Manhattan distance.

```
1 template <int MAXN = 200000, int MAXK = 2>
2 struct kd_tree { int k, size; struct point {
3   int data[MAXK], id; } p[MAXN];
4   struct kd_node { int l, r; point p, dmin, dmax;
5     kd_node() {} kd_node(const point &rhs)
6       : l(-1), r(-1), p(rhs), dmin(rhs),
7         dmax(rhs) {}
8   void merge(const kd_node &rhs, int k) {
9     for (register int i = 0; i < k; ++i) {
10       dmin.data[i] =
11         std::min(dmin.data[i], rhs.dmin.data[i]);
12       dmax.data[i] =
13         std::max(dmax.data[i], rhs.dmax.data[i]);
14     } }
15   long long min_dist(
16     const point &rhs, int k) const {
17     register long long ret = 0;
18     for (register int i = 0; i < k; ++i) {
19       if (dmin.data[i] <= rhs.data[i] &&
20         rhs.data[i] <= dmax.data[i]) continue;
21       ret += std::min(1ll *
22         (dmin.data[i] - rhs.data[i]) *
23         (dmin.data[i] - rhs.data[i]),
24         1ll * (dmax.data[i] - rhs.data[i]) *
25         (dmax.data[i] - rhs.data[i]));
26       // ret += std::max(0, rhs.data[i]
27       // - dmax.data[i]) + std::max(0,
28       // dmin.data[i] - rhs.data[i]);
29     } return ret; }
30   long long max_dist(const point &rhs, int k) {
31     long long ret = 0;
32     for (int i = 0; i < k; ++i) {
33       int tmp = std::max(
34         std::abs(dmin.data[i] - rhs.data[i]),
35         std::abs(dmax.data[i] - rhs.data[i]));
36       ret += 1ll * tmp * tmp; }
```

```

37 // ret += std::max(std::abs
38 // (rhs.data[i] - dmax.data[i]) + std::abs
39 // (rhs.data[i] - dmin.data[i])); }
40 return ret; } } tree[MAXN * 4];
41 struct result { long long dist; point d;
42 result() {}
43 result(const long long &dist, const point &d)
44 : dist(dist), d(d) {}
45 bool operator>(const result &rhs) const {
46 return dist > rhs.dist ||
47 (dist == rhs.dist && d.id > rhs.d.id); }
48 bool operator<(const result &rhs) const {
49 return dist < rhs.dist ||
50 (dist == rhs.dist && d.id < rhs.d.id); } };
51 long long sqrdist(
52 const point &a, const point &b) {
53 long long ret = 0; for (int i = 0; i < k; ++i)
54 ret += 1ll * (a.data[i] - b.data[i]) *
55 (a.data[i] - b.data[i]);
56 // for (int i = 0; i < k; ++i) ret +=
57 // std::abs(a.data[i] - b.data[i]);
58 return ret; }
59 int alloc() { tree[size].l = tree[size].r = -1;
60 return size++; }
61 void build(const int &depth, int &rt,
62 const int &l, const int &r) { if (l > r) return;
63 register int middle = (l + r) >> 1;
64 std::nth_element(p + l, p + middle, p + r + 1,
65 [=](const point &a, const point &b) {
66 return a.data[depth] < b.data[depth]; });
67 tree[rt = alloc()] = kd_node(p[middle]);
68 if (l == r) return; build(
69 (depth + 1) % k, tree[rt].l, l, middle - 1);
70 build(
71 (depth + 1) % k, tree[rt].r, middle + 1, r);
72 if (~tree[rt].l)
73 tree[rt].merge(tree[tree[rt].l], k);
74 if (~tree[rt].r)
75 tree[rt].merge(tree[tree[rt].r], k); }
76 std::priority_queue<result, std::vector<result>,
77 std::less<result>> heap_l;
78 std::priority_queue<result, std::vector<result>,
79 std::greater<result>> heap_r;
80 void _min_kth(const int &depth, const int &rt,
81 const int &m, const point &d) { result tmp =
82 result(sqrdist(tree[rt].p, d), tree[rt].p);
83 if ((int)heap_l.size() < m) heap_l.push(tmp);
84 else if (tmp < heap_l.top()) { heap_l.pop();
85 heap_l.push(tmp); }
86 int x = tree[rt].l, y = tree[rt].r;
87 if (~x && ~y &&
88 sqrdist(d, tree[x].p) > sqrdist(d, tree[y].p))
89 std::swap(x, y);
90 if (~x && ((int)heap_l.size() < m ||
91 tree[x].min_dist(d, k) < heap_l.top().dist))
92 _min_kth((depth + 1) % k, x, m, d);
93 if (~y && ((int)heap_l.size() < m ||
94 tree[y].min_dist(d, k) < heap_l.top().dist))
95 _min_kth((depth + 1) % k, y, m, d); }
96 void _max_kth(const int &depth, const int &rt,
97 const int &m, const point &d) { result tmp =
98 result(sqrdist(tree[rt].p, d), tree[rt].p);
99 if ((int)heap_r.size() < m) heap_r.push(tmp);
100 else if (tmp > heap_r.top()) { heap_r.pop();
101 heap_r.push(tmp); }
102 int x = tree[rt].l, y = tree[rt].r;
103 if (~x && ~y &&
104 sqrdist(d, tree[x].p) < sqrdist(d, tree[y].p))
105 std::swap(x, y);
106 if (~x && ((int)heap_r.size() < m ||
107 tree[x].max_dist(d, k) >=
108 heap_r.top().dist))
109 _max_kth((depth + 1) % k, x, m, d);
110 if (~y && ((int)heap_r.size() < m ||
111 tree[y].max_dist(d, k) >=
112 heap_r.top().dist))

```

```

113 _max_kth((depth + 1) % k, y, m, d); }
114 void init(int n, int k) { this->k = k; size = 0;
115 int rt = 0; build(0, rt, 0, n - 1); }
116 result min_kth(const point &d, const int &m) {
117 heap_l = decltype(heap_l)();
118 _min_kth(0, 0, m, d); return heap_l.top(); }
119 result max_kth(const point &d, const int &m) {
120 heap_r = decltype(heap_r)();
121 _max_kth(0, 0, m, d); return heap_r.top(); } };

```

3 Geometry

```

1 #define cd const double &
2 const double EPS = 1E-8, PI = acos(-1);
3 int sgn(cd x) { return x < -EPS ? -1 : x > EPS; }
4 int cmp(cd x, cd y) { return sgn(x - y); }
5 double sqr(cd x) { return x * x; }
6 double msqrt(cd x) {
7 return sgn(x) <= 0 ? 0 : sqrt(x); }

```

3.1 2D-Geometry

1. point::rot90: Counter-clockwise rotation.
2. line_circle_intersect: Ordered w.r.t. the direction of a.
3. circle_intersect: Counter-clockwise w.r.t. O_a .
4. tangent: Counter-clockwise w.r.t. a.
5. extangent: Counter-clockwise w.r.t. O_a .
6. intangent: Counter-clockwise w.r.t. O_a .

```

1 #define cp const point &
2 struct point { double x, y;
3 explicit point(cd x = 0, cd y = 0) : x(x), y(y) {}
4 int dim() const {
5 return sgn(y) == 0 ? sgn(x) > 0 : sgn(y) > 0; }
6 point unit() const {
7 double l = msqrt(x * x + y * y);
8 return point(x / l, y / l); }
9 point rot90() const { return point(-y, x); }
10 point _rot90() const { return point(y, -x); }
11 point rot(cd t) const {
12 double c = cos(t), s = sin(t);
13 return point(x * c - y * s, x * s + y * c); } };
14 bool operator==(cp a, cp b) {
15 return cmp(a.x, b.x) == 0 && cmp(a.y, b.y) == 0; }
16 bool operator!=(cp a, cp b) {
17 return cmp(a.x, b.x) != 0 || cmp(a.y, b.y) != 0; }
18 bool operator<(cp a, cp b) {
19 return cmp(a.x, b.x) == 0 ? cmp(a.y, b.y) < 0
20 : cmp(a.x, b.x) < 0; }
21 point operator-(cp a) { return point(-a.x, -a.y); }
22 point operator+(cp a, cp b) {
23 return point(a.x + b.x, a.y + b.y); }
24 point operator-(cp a, cp b) {
25 return point(a.x - b.x, a.y - b.y); }
26 point operator*(cp a, cd b) {
27 return point(a.x * b, a.y * b); }
28 point operator/(cp a, cd b) {
29 return point(a.x / b, a.y / b); }
30 double dot(cp a, cp b) {
31 return a.x * b.x + a.y * b.y; }
32 double det(cp a, cp b) {
33 return a.x * b.y - a.y * b.x; }
34 double dis2(cp a, cp b = point()) {
35 return sqr(a.x - b.x) + sqr(a.y - b.y); }
36 double dis(cp a, cp b = point()) {
37 return msqrt(dis2(a, b)); }
38 #define cl const line &
39 struct line { point s, t;
40 explicit line(cp s = point(), cp t = point())
41 : s(s), t(t) {} };

```

```

42 bool point_on_segment(cp a, cl b) {
43     return sgn(det(a - b.s, b.t - b.s)) == 0 &&
44         sgn(dot(b.s - a, b.t - a)) <= 0; }
45 bool two_side(cp a, cp b, cl c) {
46     return sgn(det(a - c.s, c.t - c.s)) *
47         sgn(det(b - c.s, c.t - c.s)) < 0; }
48 bool intersect_judgment(cl a, cl b) {
49     if (point_on_segment(b.s, a) ||
50         point_on_segment(b.t, a)) return true;
51     if (point_on_segment(a.s, b) ||
52         point_on_segment(a.t, b)) return true;
53     return two_side(a.s, a.t, b) &&
54         two_side(b.s, b.t, a); }
55 point line_intersect(cl a, cl b) {
56     double s1 = det(a.t - a.s, b.s - a.s),
57         s2 = det(a.t - a.s, b.t - a.s);
58     return (b.s * s2 - b.t * s1) / (s2 - s1); }
59 double point_to_line(cp a, cl b) {
60     return std::abs(det(b.t - b.s, a - b.s)) /
61         dis(b.s, b.t); }
62 point project_to_line(cp a, cl b) { return b.s +
63     (b.t - b.s) *
64     (dot(a - b.s, b.t - b.s) / dis2(b.t, b.s)); }
65 double point_to_segment(cp a, cl b) {
66     if (sgn(dot(b.s - a, b.t - b.s)) *
67         dot(b.t - a, b.t - b.s)) <= 0)
68         return std::abs(det(b.t - b.s, a - b.s)) /
69             dis(b.s, b.t);
70     return std::min(dis(a, b.s), dis(a, b.t)); }
71 bool in_polygon(
72     cp p, const std::vector<point> &po) {
73     int n = (int)po.size(), counter = 0;
74     for (int i = 0; i < n; ++i) {
75         point a = po[i], b = po[(i + 1) % n];
76         // Modify the next line if necessary.
77         if (point_on_segment(p, line(a, b)))
78             return true;
79         int x = sgn(det(p - a, b - a)),
80             y = sgn(a.y - p.y), z = sgn(b.y - p.y);
81         if (x > 0 && y <= 0 && z > 0) counter++;
82         if (x < 0 && z <= 0 && y > 0) counter--; }
83     return counter != 0; }
84 double polygon_area(const std::vector<point> &a) {
85     double ans = 0.0;
86     for (int i = 0; i < (int)a.size(); ++i)
87         ans += det(a[i], a[(i + 1) % a.size()]) / 2.0;
88     return ans; }
89 #define cc const circle &
90 struct circle { point c; double r;
91     explicit circle(point c = point(), double r = 0)
92         : c(c), r(r) {} };
93 bool operator==(cc a, cc b) {
94     return a.c == b.c && cmp(a.r, b.r) == 0; }
95 bool operator!=(cc a, cc b) { return !(a == b); }
96 bool in_circle(cp a, cc b) {
97     return cmp(dis(a, b.c), b.r) <= 0; }
98 circle make_circle(cp a, cp b) {
99     return circle((a + b) / 2, dis(a, b) / 2); }
100 circle make_circle(cp a, cp b, cp c) {
101     point p = circumcenter(a, b, c);
102     return circle(p, dis(p, a)); }
103 std::vector<point> line_circle_intersect(
104     cl a, cc b) {
105     if (cmp(point_to_line(b.c, a), b.r) > 0)
106         return std::vector<point>();
107     double x =
108         msqrt(sqr(b.r) - sqr(point_to_line(b.c, a)));
109     point s = project_to_line(b.c, a),
110         u = (a.t - a.s).unit();
111     if (sgn(x) == 0) return std::vector<point>({s});
112     return std::vector<point>({s - u * x, s + u * x}); }
113 double circle_intersect_area(cc a, cc b) {
114     double d = dis(a.c, b.c);
115     if (sgn(d - (a.r + b.r)) >= 0) return 0;
116     if (sgn(d - std::abs(a.r - b.r)) <= 0) {
117         double r = std::min(a.r, b.r);

```

```

118         return r * r * PI; }
119 double x = (d * d + a.r * a.r - b.r * b.r) /
120     (2 * d), t1 = acos(
121         std::min(1., std::max(-1., x / a.r))),
122     t2 = acos(std::min(
123         1., std::max(-1., (d - x) / b.r)));
124     return a.r * a.r * t1 + b.r * b.r * t2 -
125         d * a.r * sin(t1); }
126 std::vector<point> circle_intersect(cc a, cc b) {
127     if (a.c == b.c ||
128         cmp(dis(a.c, b.c), a.r + b.r) > 0 ||
129         cmp(dis(a.c, b.c), std::abs(a.r - b.r)) < 0)
130         return std::vector<point>();
131     point r = (b.c - a.c).unit();
132     double d = dis(a.c, b.c);
133     double x = ((sqr(a.r) - sqr(b.r)) / d + d) / 2,
134         h = msqrt(sqr(a.r) - sqr(x));
135     if (sgn(h) == 0)
136         return std::vector<point>({a.c + r * x});
137     return std::vector<point>({
138         a.c + r * x - r.rot90() * h,
139         a.c + r * x + r.rot90() * h}); }
140 std::vector<point> tangent(cp a, cc b) {
141     circle p = make_circle(a, b.c);
142     return circle_intersect(p, b); }
143 std::vector<line> extangent(cc a, cc b) {
144     std::vector<line> ret;
145     if (cmp(dis(a.c, b.c), std::abs(a.r - b.r)) <= 0)
146         return ret;
147     if (sgn(a.r - b.r) == 0) { point dir = b.c - a.c;
148         dir = (dir * a.r / dis(dir)).rot90();
149         ret.push_back(line(a.c - dir, b.c - dir));
150         ret.push_back(line(a.c + dir, b.c + dir));
151     } else {
152         point p = (b.c * a.r - a.c * b.r) / (a.r - b.r);
153         std::vector<point> pp = tangent(p, a),
154             qq = tangent(p, b);
155         if (pp.size() == 2 && qq.size() == 2) {
156             if (cmp(a.r, b.r) < 0)
157                 std::swap(pp[0], pp[1]),
158                 std::swap(qq[0], qq[1]);
159             ret.push_back(line(pp[0], qq[0]));
160             ret.push_back(line(pp[1], qq[1])); } }
161     return ret; }
162 std::vector<line> intangent(cc a, cc b) {
163     std::vector<line> ret;
164     point p = (b.c * a.r + a.c * b.r) / (a.r + b.r);
165     std::vector<point> pp = tangent(p, a),
166         qq = tangent(p, b);
167     if (pp.size() == 2 && qq.size() == 2) {
168         ret.push_back(line(pp[0], qq[0]));
169         ret.push_back(line(pp[1], qq[1])); }
170     return ret; }

```

3.1.1 Triangle Center

```

1 point incenter(cp a, cp b, cp c) {
2     double p = dis(a, b) + dis(b, c) + dis(c, a);
3     return (a * dis(b, c) + b * dis(c, a) +
4         c * dis(a, b)) / p; }
5 point circumcenter(cp a, cp b, cp c) {
6     point p = b - a, q = c - a,
7         s(dot(p, p) / 2, dot(q, q) / 2);
8     return a + point(det(s, point(p.y, q.y)),
9         det(point(p.x, q.x), s)) / det(p, q); }
10 point orthocenter(cp a, cp b, cp c) {
11     return a + b + c - circumcenter(a, b, c) * 2; }

```

3.1.2 Fermat Point

Find a point P that minimizes $|PA| + |PB| + |PC|$.

```

1 point fermat_point(cp a, cp b, cp c) {
2     if (a == b) return a; if (b == c) return b;

```

```

3  if (c == a) return c;
4  double ab = dis(a, b), bc = dis(b, c),
5         ca = dis(c, a);
6  double cosa = dot(b - a, c - a) / ab / ca;
7  double cosb = dot(a - b, c - b) / ab / bc;
8  double cosc = dot(b - c, a - c) / ca / bc;
9  double sq3 = PI / 3.0; point mid;
10 if (sgn(cosa + 0.5) < 0) mid = a;
11 else if (sgn(cosb + 0.5) < 0) mid = b;
12 else if (sgn(cosc + 0.5) < 0) mid = c;
13 else if (sgn(det(b - a, c - a)) < 0) mid =
14     line_intersect(line(a, b + (c - b).rot(sq3)),
15                   line(b, c + (a - c).rot(sq3)));
16 else mid =
17     line_intersect(line(a, c + (b - c).rot(sq3)),
18                   line(c, b + (a - b).rot(sq3)));
19 return mid; }

```

3.1.3 Convex Hull

Counter-clockwise, starting with the smallest point and with the minimum number of points. Modify `!= -s` to `== s` in turn to preserve all points on the hull.

`convex_tan` finds the covering `[s..t]` of a certain point.

```

1 bool turn(cp a, cp b, cp c, int s) {
2     return sgn(det(b - a, c - a)) != -s; }
3 std::pair<std::vector<point>, int> convex_hull(
4     std::vector<point> a) { int cnt = 0;
5     std::sort(a.begin(), a.end());
6     static std::vector<point> ret;
7     ret.resize(a.size() << 1);
8     for (int i = 0; i < (int)a.size(); ++i) {
9         while (cnt > 1 &&
10             turn(ret[cnt - 2], a[i], ret[cnt - 1], 1))
11             --cnt; ret[cnt++] = a[i]; }
12     int fixed = cnt;
13     for (int i = (int)a.size() - 1; i >= 0; --i) {
14         while (cnt > fixed &&
15             turn(ret[cnt - 2], a[i], ret[cnt - 1], 1))
16             --cnt; ret[cnt++] = a[i]; }
17     return std::make_pair(std::vector<point>(
18         ret.begin(), ret.begin() + cnt - 1),
19         fixed - 1); }
20 int lb(cp x, const std::vector<point> &v, int l,
21     int r, int s) { if (l > r) l = r; while (l != r) {
22     int m = (l + r) / 2;
23     if (sgn(det(v[m % v.size()] - x,
24         v[(m + 1) % v.size()] - x)) == s)
25         r = m; else l = m + 1; }
26     return r % v.size(); }
27 std::pair<int, int> convex_tan(
28     cp x, const std::vector<point> &v, int rp) {
29     if (cmp(x.x, v[0].x) < 0) return std::make_pair(
30         lb(x, v, rp, v.size(), -1),
31         lb(x, v, 0, rp, 1));
32     else if (cmp(x.x, v[rp].x) > 0)
33         return std::make_pair(lb(x, v, 0, rp, -1),
34             lb(x, v, rp, v.size(), 1));
35     else { int id = std::lower_bound(
36         v.begin(), v.begin() + rp, x) -
37         v.begin();
38     if (id == 0 ||
39         sgn(det(v[id - 1] - x, v[id] - x)) < 0)
40         return std::make_pair(
41             lb(x, v, 0, id, -1), lb(x, v, id, rp, 1));
42     id = std::lower_bound(v.begin() + rp, v.end(),
43         x, std::greater<point>()) -

```

```

44     v.begin();
45     if (id == rp || sgn(det(
46         v[id - 1] - x, v[id % v.size()] - x)) < 0)
47         return std::make_pair(lb(x, v, rp, id, -1),
48             lb(x, v, id, v.size(), 1));
49     return std::make_pair(-1, -1); } }

```

3.1.4 Half-Plane Intersection

1. cut: Online in $O(n^2)$.
2. half_plane_intersect: Offline in $O(m \log m)$.

```

1 std::vector<point> cut(
2     const std::vector<point> &c, line p) {
3     std::vector<point> ret; if (c.empty()) return ret;
4     for (int i = 0; i < (int)c.size(); ++i) {
5         int j = (i + 1) % (int)c.size();
6         if (turn_left(p.s, p.t, c[i]))
7             ret.push_back(c[i]);
8         if (two_side(c[i], c[j], p)) ret.push_back(
9             line_intersect(p, line(c[i], c[j]))); }
10    return ret; }
11 bool turn_left(cl l, cp p) {
12     return sgn(det(l.t - l.s, p - l.s)) >= 0; }
13 int cmp(cp a, cp b) { return a.dim() != b.dim()
14     ? (a.dim() < b.dim() ? -1 : 1)
15     : -sgn(det(a, b)); }
16 std::vector<point> half_plane_intersect(
17     std::vector<line> h) {
18     typedef std::pair<point, line> polar;
19     std::vector<polar> g; g.resize(h.size());
20     for (int i = 0; i < (int)h.size(); ++i)
21         g[i] = std::make_pair(h[i].t - h[i].s, h[i]);
22     sort(g.begin(), g.end(),
23         [&](const polar &a, const polar &b) {
24             if (cmp(a.first, b.first) == 0)
25                 return sgn(det(a.second.t - a.second.s,
26                     b.second.t - a.second.s)) < 0;
27             else return cmp(a.first, b.first) < 0; });
28     h.resize(std::unique(g.begin(), g.end(),
29         [&](const polar &a, const polar &b) {
30             return cmp(a.first, b.first) == 0;
31         }) -
32         g.begin());
33     for (int i = 0; i < (int)h.size(); ++i)
34         h[i] = g[i].second;
35     int fore = 0, rear = -1;
36     std::vector<line> ret(h.size(), line());
37     for (int i = 0; i < (int)h.size(); ++i) {
38         while (fore < rear && !turn_left(h[i],
39             line_intersect(ret[rear - 1], ret[rear])))
40             --rear;
41         while (fore < rear && !turn_left(h[i],
42             line_intersect(ret[fore], ret[fore + 1])))
43             ++fore; ret[++rear] = h[i]; }
44     while (rear - fore > 1 && !turn_left(ret[fore],
45         line_intersect(ret[rear - 1], ret[rear])))
46         --rear;
47     while (rear - fore > 1 && !turn_left(ret[rear],
48         line_intersect(ret[fore], ret[fore + 1])))
49         ++fore;
50     if (rear - fore < 2) return std::vector<point>();
51     std::vector<point> ans; ans.resize(rear + 1);
52     for (int i = 0; i < rear + 1; ++i)
53         ans[i] = line_intersect(
54             ret[i], ret[(i + 1) % (rear + 1)]);
55     return ans; }

```