Competitive Programmer’s Reference

Zhongtang Luo

October 22, 2023

Contents

1 Environment

1.1 Vimrc

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

2 Data Structure

2.1 RMQ

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

2.2 Link-Cut Tree

```
struct Node { int son[2], fa, num, pos, rev;
} node[maxn]; int n, m, ans, top, q[maxn];
inline bool root(int x) {
  return node[node[x].fa].son[0] != x &&
    node[node[x].fa].son[1] != x; }
void update(int x) {
  int left = node[x].son[0], right = node[x].son[1];
  node[x].pos = x; if (node[node[left].pos].num >
    node[node[x].pos].num)
    node[x].pos = node[left].pos;
  if (node[node[right].pos].num >
    node[node[x].pos].num)
    node[x].pos = node[right].pos; }
void down(int x) {
  int left = node[x].son[0], right = node[x].son[1];
  if (node[x].rev) { node[x].rev ^= 1;
    node[left].rev ^= 1; node[right].rev ^= 1;
    std::swap(node[x].son[0], node[x].son[1]); } }
void rotate(int x) {
  int y = node[x].fa, z = node[y].fa, left, right;
  if (node[y].son[0] == x) left = 0; else left = 1;
  right = left ^ 1; if (!root(y)) {
    if (node[z].son[0] == y) node[z].son[0] = x;
    else node[z].son[1] = x; }
  node[x].fa = z; node[y].fa = x;
  if (node[x].son[right] != 0)
    node[node[x].son[right]].fa = y;
  node[y].son[left] = node[x].son[right];
  node[x].son[right] = y; update(y); update(x); }
void splay(int x) { top = 0; q[++top] = x;
  for (int i = x; !root(i); i = node[i].fa)
    q[++top] = node[i].fa;
  for (int i = top; i; i--) down(q[i]);
  while (!root(x)) {
    int y = node[x].fa, z = node[y].fa;
    if (!root(y)) {
      if (node[y].son[0] == x ^ node[z].son[0] == y)
        rotate(x); else rotate(y); } rotate(x); }
  update(x); }
void access(int x) { int t = 0; while (x) {
    splay(x); node[x].son[1] = t; t = x;
    x = node[x].fa; } }
void makeroot(int x) { access(x); splay(x);
  node[x].rev ^= 1; }
void link(int x, int y) { makeroot(x);
  node[x].fa = y; }
void cut(int x, int y) { makeroot(x); access(y);
  splay(y); node[node[y].son[0]].fa = 0;
  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.

```
template <int MAXN = 200000, int MAXK = 2>
struct kd_tree { int k, size; struct point {
    int data[MAXK], id; } p[MAXN];
  struct kd_node { int l, r; point p, dmin, dmax;
    kd_node() {} kd_node(const point &rhs)
        : l(-1), r(-1), p(rhs), dmin(rhs),
          dmax(rhs) {}
    void merge(const kd_node &rhs, int k) {
      for (register int i = 0; i < k; ++i) {
        dmin.data[i] =
          std::min(dmin.data[i], rhs.dmin.data[i]);
        dmax.data[i] =
          std::max(dmax.data[i], rhs.dmax.data[i]);
      } }
    long long min_dist(
      const point &rhs, int k) const {
      register long long ret = 0;
      for (register int i = 0; i < k; ++i) {
        if (dmin.data[i] <= rhs.data[i] &&
          rhs.data[i] <= dmax.data[i]) continue;
        ret += std::min(1ll *
            (dmin.data[i] - rhs.data[i]) *
            (dmin.data[i] - rhs.data[i]),
          1ll * (dmax.data[i] - rhs.data[i]) *
            (dmax.data[i] - rhs.data[i]));
        // ret += std::max (0, rhs.data[i]
        //- dmax.data[i]) + std::max (0,
        // dmin.data[i] - rhs.data[i]);
      } return ret; }
    long long max_dist(const point &rhs, int k) {
      long long ret = 0;
      for (int i = 0; i < k; ++i) {
        int tmp = std::max(
          std::abs(dmin.data[i] - rhs.data[i]),
          std::abs(dmax.data[i] - rhs.data[i]));
        ret += 1ll * tmp * tmp; }
```

```
      // ret += std::max (std::abs
      //(rhs.data[i] - dmax.data[i]) + std::abs
      //(rhs.data[i] - dmin.data[i])); }
      return ret; } } tree[MAXN * 4];
struct result { long long dist; point d;
  result() {}
  result(const long long &dist, const point &d)
      : dist(dist), d(d) {}
  bool operator>(const result &rhs) const {
    return dist > rhs.dist ||
      (dist == rhs.dist && d.id > rhs.d.id); }
  bool operator<(const result &rhs) const {
    return dist < rhs.dist ||
      (dist == rhs.dist && d.id < rhs.d.id); } };
long long sqrdist(
  const point &a, const point &b) {
  long long ret = 0; for (int i = 0; i < k; ++i)
    ret += 1ll * (a.data[i] - b.data[i]) *
      (a.data[i] - b.data[i]);
  // for (int i = 0; i < k; ++i) ret +=
  // std::abs (a.data[i] - b.data[i]);
  return ret; }
int alloc() { tree[size].l = tree[size].r = -1;
  return size++; }
void build(const int &depth, int &rt,
  const int &l, const int &r) { if (l > r) return;
  register int middle = (l + r) >> 1;
  std::nth_element(p + l, p + middle, p + r + 1,
    [=](const point &a, const point &b) {
      return a.data[depth] < b.data[depth]; });
  tree[rt = alloc()] = kd_node(p[middle]);
  if (l == r) return; build(
    (depth + 1) % k, tree[rt].l, l, middle - 1);
  build(
    (depth + 1) % k, tree[rt].r, middle + 1, r);
  if (~tree[rt].l)
    tree[rt].merge(tree[tree[rt].l], k);
  if (~tree[rt].r)
    tree[rt].merge(tree[tree[rt].r], k); }
std::priority_queue<result, std::vector<result>,
  std::less<result>> heap_l;
std::priority_queue<result, std::vector<result>,
  std::greater<result>> heap_r;
void _min_kth(const int &depth, const int &rt,
  const int &m, const point &d) { result tmp =
    result(sqrdist(tree[rt].p, d), tree[rt].p);
  if ((int)heap_l.size() < m) heap_l.push(tmp);
  else if (tmp < heap_l.top()) { heap_l.pop();
    heap_l.push(tmp); }
  int x = tree[rt].l, y = tree[rt].r;
  if (~x && ~y &&
    sqrdist(d, tree[x].p) > sqrdist(d, tree[y].p))
    std::swap(x, y);
  if (~x && ((int)heap_l.size() < m ||
      tree[x].min_dist(d, k) < heap_l.top().dist))
    _min_kth((depth + 1) % k, x, m, d);
  if (~y && ((int)heap_l.size() < m ||
      tree[y].min_dist(d, k) < heap_l.top().dist))
    _min_kth((depth + 1) % k, y, m, d); }
void _max_kth(const int &depth, const int &rt,
  const int &m, const point &d) { result tmp =
    result(sqrdist(tree[rt].p, d), tree[rt].p);
  if ((int)heap_r.size() < m) heap_r.push(tmp);
  else if (tmp > heap_r.top()) { heap_r.pop();
    heap_r.push(tmp); }
  int x = tree[rt].l, y = tree[rt].r;
  if (~x && ~y &&
    sqrdist(d, tree[x].p) < sqrdist(d, tree[y].p))
    std::swap(x, y);
  if (~x && ((int)heap_r.size() < m ||
      tree[x].max_dist(d, k) >=
        heap_r.top().dist))
    _max_kth((depth + 1) % k, x, m, d);
  if (~y && ((int)heap_r.size() < m ||
      tree[y].max_dist(d, k) >=
        heap_r.top().dist))
    _max_kth((depth + 1) % k, y, m, d); }
  void init(int n, int k) { this->k = k; size = 0;
    int rt = 0; build(0, rt, 0, n - 1); }
  result min_kth(const point &d, const int &m) {
    heap_l = decltype(heap_l)();
    _min_kth(0, 0, m, d); return heap_l.top(); }
  result max_kth(const point &d, const int &m) {
    heap_r = decltype(heap_r)();
    _max_kth(0, 0, m, d); return heap_r.top(); } };
```

3 Geometry

```
#define cd const double &
const double EPS = 1E-8, PI = acos(-1);
int sgn(cd x) { return x < -EPS ? -1 : x > EPS; }
int cmp(cd x, cd y) { return sgn(x - y); }
double sqr(cd x) { return x * x; }
double msqrt(cd x) {
  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.

```
#define cp const point &
struct point { double x, y;
  explicit point(cd x = 0, cd y = 0) : x(x), y(y) {}
  int dim() const {
    return sgn(y) == 0 ? sgn(x) > 0 : sgn(y) > 0; }
  point unit() const {
    double l = msqrt(x * x + y * y);
    return point(x / l, y / l); }
  point rot90() const { return point(-y, x); }
  point _rot90() const { return point(y, -x); }
  point rot(cd t) const {
    double c = cos(t), s = sin(t);
    return point(x * c - y * s, x * s + y * c); } };
bool operator==(cp a, cp b) {
  return cmp(a.x, b.x) == 0 && cmp(a.y, b.y) == 0; }
bool operator!=(cp a, cp b) {
  return cmp(a.x, b.x) != 0 || cmp(a.y, b.y) != 0; }
bool operator<(cp a, cp b) {
  return cmp(a.x, b.x) == 0 ? cmp(a.y, b.y) < 0
                            : cmp(a.x, b.x) < 0; }
point operator-(cp a) { return point(-a.x, -a.y); }
point operator+(cp a, cp b) {
  return point(a.x + b.x, a.y + b.y); }
point operator-(cp a, cp b) {
  return point(a.x - b.x, a.y - b.y); }
point operator*(cp a, cd b) {
  return point(a.x * b, a.y * b); }
point operator/(cp a, cd b) {
  return point(a.x / b, a.y / b); }
double dot(cp a, cp b) {
  return a.x * b.x + a.y * b.y; }
double det(cp a, cp b) {
  return a.x * b.y - a.y * b.x; }
double dis2(cp a, cp b = point()) {
  return sqr(a.x - b.x) + sqr(a.y - b.y); }
double dis(cp a, cp b = point()) {
  return msqrt(dis2(a, b)); }
#define cl const line &
struct line { point s, t;
  explicit line(cp s = point(), cp t = point())
      : s(s), t(t) {} };
```

```
bool point_on_segment(cp a, cl b) {
  return sgn(det(a - b.s, b.t - b.s)) == 0 &&
    sgn(dot(b.s - a, b.t - a)) <= 0; }
bool two_side(cp a, cp b, cl c) {
  return sgn(det(a - c.s, c.t - c.s)) *
    sgn(det(b - c.s, c.t - c.s)) < 0; }
bool intersect_judgment(cl a, cl b) {
  if (point_on_segment(b.s, a) ||
    point_on_segment(b.t, a)) return true;
  if (point_on_segment(a.s, b) ||
    point_on_segment(a.t, b)) return true;
  return two_side(a.s, a.t, b) &&
    two_side(b.s, b.t, a); }
point line_intersect(cl a, cl b) {
  double s1 = det(a.t - a.s, b.s - a.s),
         s2 = det(a.t - a.s, b.t - a.s);
  return (b.s * s2 - b.t * s1) / (s2 - s1); }
double point_to_line(cp a, cl b) {
  return std::abs(det(b.t - b.s, a - b.s)) /
    dis(b.s, b.t); }
point project_to_line(cp a, cl b) { return b.s +
    (b.t - b.s) *
    (dot(a - b.s, b.t - b.s) / dis2(b.t, b.s)); }
double point_to_segment(cp a, cl b) {
  if (sgn(dot(b.s - a, b.t - b.s) *
        dot(b.t - a, b.t - b.s)) <= 0)
    return std::abs(det(b.t - b.s, a - b.s)) /
      dis(b.s, b.t);
  return std::min(dis(a, b.s), dis(a, b.t)); }
bool in_polygon(
  cp p, const std::vector<point> &po) {
  int n = (int)po.size(), counter = 0;
  for (int i = 0; i < n; ++i) {
    point a = po[i], b = po[(i + 1) % n];
    // Modify the next line if necessary.
    if (point_on_segment(p, line(a, b)))
      return true;
    int x = sgn(det(p - a, b - a)),
        y = sgn(a.y - p.y), z = sgn(b.y - p.y);
    if (x > 0 && y <= 0 && z > 0) counter++;
    if (x < 0 && z <= 0 && y > 0) counter--; }
  return counter != 0; }
double polygon_area(const std::vector<point> &a) {
  double ans = 0.0;
  for (int i = 0; i < (int)a.size(); ++i)
    ans += det(a[i], a[(i + 1) % a.size()]) / 2.0;
  return ans; }
#define cc const circle &
struct circle { point c; double r;
  explicit circle(point c = point(), double r = 0)
      : c(c), r(r) {} };
bool operator==(cc a, cc b) {
  return a.c == b.c && cmp(a.r, b.r) == 0; }
bool operator!=(cc a, cc b) { return !(a == b); }
bool in_circle(cp a, cc b) {
  return cmp(dis(a, b.c), b.r) <= 0; }
circle make_circle(cp a, cp b) {
  return circle((a + b) / 2, dis(a, b) / 2); }
circle make_circle(cp a, cp b, cp c) {
  point p = circumcenter(a, b, c);
  return circle(p, dis(p, a)); }
std::vector<point> line_circle_intersect(
  cl a, cc b) {
  if (cmp(point_to_line(b.c, a), b.r) > 0)
    return std::vector<point>();
  double x =
    msqrt(sqr(b.r) - sqr(point_to_line(b.c, a)));
  point s = project_to_line(b.c, a),
        u = (a.t - a.s).unit();
  if (sgn(x) == 0) return std::vector<point>({s});
  return std::vector<point>({s - u * x, s + u * x});
} double circle_intersect_area(cc a, cc b) {
  double d = dis(a.c, b.c);
  if (sgn(d - (a.r + b.r)) >= 0) return 0;
  if (sgn(d - std::abs(a.r - b.r)) <= 0) {
    double r = std::min(a.r, b.r);
    return r * r * PI; }
  double x = (d * d + a.r * a.r - b.r * b.r) /
    (2 * d), t1 = acos(
          std::min(1., std::max(-1., x / a.r))),
        t2 = acos(std::min(
          1., std::max(-1., (d - x) / b.r)));
  return a.r * a.r * t1 + b.r * b.r * t2 -
    d * a.r * sin(t1); }
std::vector<point> circle_intersect(cc a, cc b) {
  if (a.c == b.c ||
    cmp(dis(a.c, b.c), a.r + b.r) > 0 ||
    cmp(dis(a.c, b.c), std::abs(a.r - b.r)) < 0)
    return std::vector<point>();
  point r = (b.c - a.c).unit();
  double d = dis(a.c, b.c);
  double x = ((sqr(a.r) - sqr(b.r)) / d + d) / 2,
         h = msqrt(sqr(a.r) - sqr(x));
  if (sgn(h) == 0)
    return std::vector<point>({a.c + r * x});
  return std::vector<point>(
    {a.c + r * x - r.rot90() * h,
      a.c + r * x + r.rot90() * h}); }
std::vector<point> tangent(cp a, cc b) {
  circle p = make_circle(a, b.c);
  return circle_intersect(p, b); }
std::vector<line> extangent(cc a, cc b) {
  std::vector<line> ret;
  if (cmp(dis(a.c, b.c), std::abs(a.r - b.r)) <= 0)
    return ret;
  if (sgn(a.r - b.r) == 0) { point dir = b.c - a.c;
    dir = (dir * a.r / dis(dir)).rot90();
    ret.push_back(line(a.c - dir, b.c - dir));
    ret.push_back(line(a.c + dir, b.c + dir));
  } else {
    point p = (b.c * a.r - a.c * b.r) / (a.r - b.r);
    std::vector<point> pp = tangent(p, a),
                       qq = tangent(p, b);
    if (pp.size() == 2 && qq.size() == 2) {
      if (cmp(a.r, b.r) < 0)
        std::swap(pp[0], pp[1]),
          std::swap(qq[0], qq[1]);
      ret.push_back(line(pp[0], qq[0]));
      ret.push_back(line(pp[1], qq[1])); } }
  return ret; }
std::vector<line> intangent(cc a, cc b) {
  std::vector<line> ret;
  point p = (b.c * a.r + a.c * b.r) / (a.r + b.r);
  std::vector<point> pp = tangent(p, a),
                     qq = tangent(p, b);
  if (pp.size() == 2 && qq.size() == 2) {
    ret.push_back(line(pp[0], qq[0]));
    ret.push_back(line(pp[1], qq[1])); }
  return ret; }
```

3.1.1 Triangle Center

```
point incenter(cp a, cp b, cp c) {
  double p = dis(a, b) + dis(b, c) + dis(c, a);
  return (a * dis(b, c) + b * dis(c, a) +
          c * dis(a, b)) / p; }
point circumcenter(cp a, cp b, cp c) {
  point p = b - a, q = c - a,
        s(dot(p, p) / 2, dot(q, q) / 2);
  return a + point(det(s, point(p.y, q.y)),
      det(point(p.x, q.x), s)) / det(p, q); }
point orthocenter(cp a, cp b, cp c) {
  return a + b + c - circumcenter(a, b, c) * 2; }
```

3.1.2 Fermat Point

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

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

```
  if (c == a) return c;
  double ab = dis(a, b), bc = dis(b, c),
         ca = dis(c, a);
  double cosa = dot(b - a, c - a) / ab / ca;
  double cosb = dot(a - b, c - b) / ab / bc;
  double cosc = dot(b - c, a - c) / ca / bc;
  double sq3 = PI / 3.0; point mid;
  if (sgn(cosa + 0.5) < 0) mid = a;
  else if (sgn(cosb + 0.5) < 0) mid = b;
  else if (sgn(cosc + 0.5) < 0) mid = c;
  else if (sgn(det(b - a, c - a)) < 0) mid =
      line_intersect(line(a, b + (c - b).rot(sq3)),
        line(b, c + (a - c).rot(sq3)));
  else mid =
      line_intersect(line(a, c + (b - c).rot(sq3)),
        line(c, b + (a - b).rot(sq3)));
  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.

```
bool turn(cp a, cp b, cp c, int s) {
  return sgn(det(b - a, c - a)) != -s; }
std::pair<std::vector<point>, int> convex_hull(
  std::vector<point> a) { int cnt = 0;
  std::sort(a.begin(), a.end());
  static std::vector<point> ret;
  ret.resize(a.size() << 1);
  for (int i = 0; i < (int)a.size(); ++i) {
    while (cnt > 1 &&
      turn(ret[cnt - 2], a[i], ret[cnt - 1], 1))
      --cnt; ret[cnt++] = a[i]; }
  int fixed = cnt;
  for (int i = (int)a.size() - 1; i >= 0; --i) {
    while (cnt > fixed &&
      turn(ret[cnt - 2], a[i], ret[cnt - 1], 1))
      --cnt; ret[cnt++] = a[i]; }
  return std::make_pair( std::vector<point>(
      ret.begin(), ret.begin() + cnt - 1),
    fixed - 1); }
int lb(cp x, const std::vector<point> &v, int l,
  int r, int s) { if (l > r) l = r; while (l != r) {
    int m = (l + r) / 2;
    if (sgn(det(v[m % v.size()] - x,
          v[(m + 1) % v.size()] - x)) == s)
      r = m; else l = m + 1; }
  return r % v.size(); }
std::pair<int, int> convex_tan(
  cp x, const std::vector<point> &v, int rp) {
  if (cmp(x.x, v[0].x) < 0) return std::make_pair(
      lb(x, v, rp, v.size(), -1),
      lb(x, v, 0, rp, 1));
  else if (cmp(x.x, v[rp].x) > 0)
    return std::make_pair(lb(x, v, 0, rp, -1),
      lb(x, v, rp, v.size(), 1));
  else { int id = std::lower_bound(
                v.begin(), v.begin() + rp, x) -
      v.begin();
    if (id == 0 ||
      sgn(det(v[id - 1] - x, v[id] - x)) < 0)
      return std::make_pair(
        lb(x, v, 0, id, -1), lb(x, v, id, rp, 1));
    id = std::lower_bound(v.begin() + rp, v.end(),
          x, std::greater<point>()) -
      v.begin();
    if (id == rp || sgn(det(
        v[id - 1] - x, v[id % v.size()] - x)) < 0)
      return std::make_pair(lb(x, v, rp, id, -1),
        lb(x, v, id, v.size(), 1));
    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 $O(m \log m)$.

```
std::vector<point> cut(
  const std::vector<point> &c, line p) {
  std::vector<point> ret; if (c.empty()) return ret;
  for (int i = 0; i < (int)c.size(); ++i) {
    int j = (i + 1) % (int)c.size();
    if (turn_left(p.s, p.t, c[i]))
      ret.push_back(c[i]);
    if (two_side(c[i], c[j], p)) ret.push_back(
        line_intersect(p, line(c[i], c[j]))); }
  return ret; }
bool turn_left(cl l, cp p) {
  return sgn(det(l.t - l.s, p - l.s)) >= 0; }
int cmp(cp a, cp b) { return a.dim() != b.dim()
    ? (a.dim() < b.dim() ? -1 : 1)
    : -sgn(det(a, b)); }
std::vector<point> half_plane_intersect(
  std::vector<line> h) {
  typedef std::pair<point, line> polar;
  std::vector<polar> g; g.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i)
    g[i] = std::make_pair(h[i].t - h[i].s, h[i]);
  sort(g.begin(), g.end(),
    [&](const polar &a, const polar &b) {
      if (cmp(a.first, b.first) == 0)
        return sgn(det(a.second.t - a.second.s,
              b.second.t - a.second.s)) < 0;
      else return cmp(a.first, b.first) < 0; });
  h.resize(std::unique(g.begin(), g.end(),
          [](const polar &a, const polar &b) {
            return cmp(a.first, b.first) == 0;
          }) -
    g.begin());
  for (int i = 0; i < (int)h.size(); ++i)
    h[i] = g[i].second;
  int fore = 0, rear = -1;
  std::vector<line> ret(h.size(), line());
  for (int i = 0; i < (int)h.size(); ++i) {
    while (fore < rear && !turn_left(h[i],
        line_intersect(ret[rear - 1], ret[rear])))
      --rear;
    while (fore < rear && !turn_left(h[i],
        line_intersect(ret[fore], ret[fore + 1])))
      ++fore; ret[++rear] = h[i]; }
  while (rear - fore > 1 && !turn_left(ret[fore],
      line_intersect(ret[rear - 1], ret[rear])))
    --rear;
  while (rear - fore > 1 && !turn_left(ret[rear],
      line_intersect(ret[fore], ret[fore + 1])))
    ++fore;
  if (rear - fore < 2) return std::vector<point>();
  std::vector<point> ans; ans.resize(rear + 1);
  for (int i = 0; i < rear + 1; ++i)
    ans[i] = line_intersect(
      ret[i], ret[(i + 1) % (rear + 1)]);
  return ans; }
```