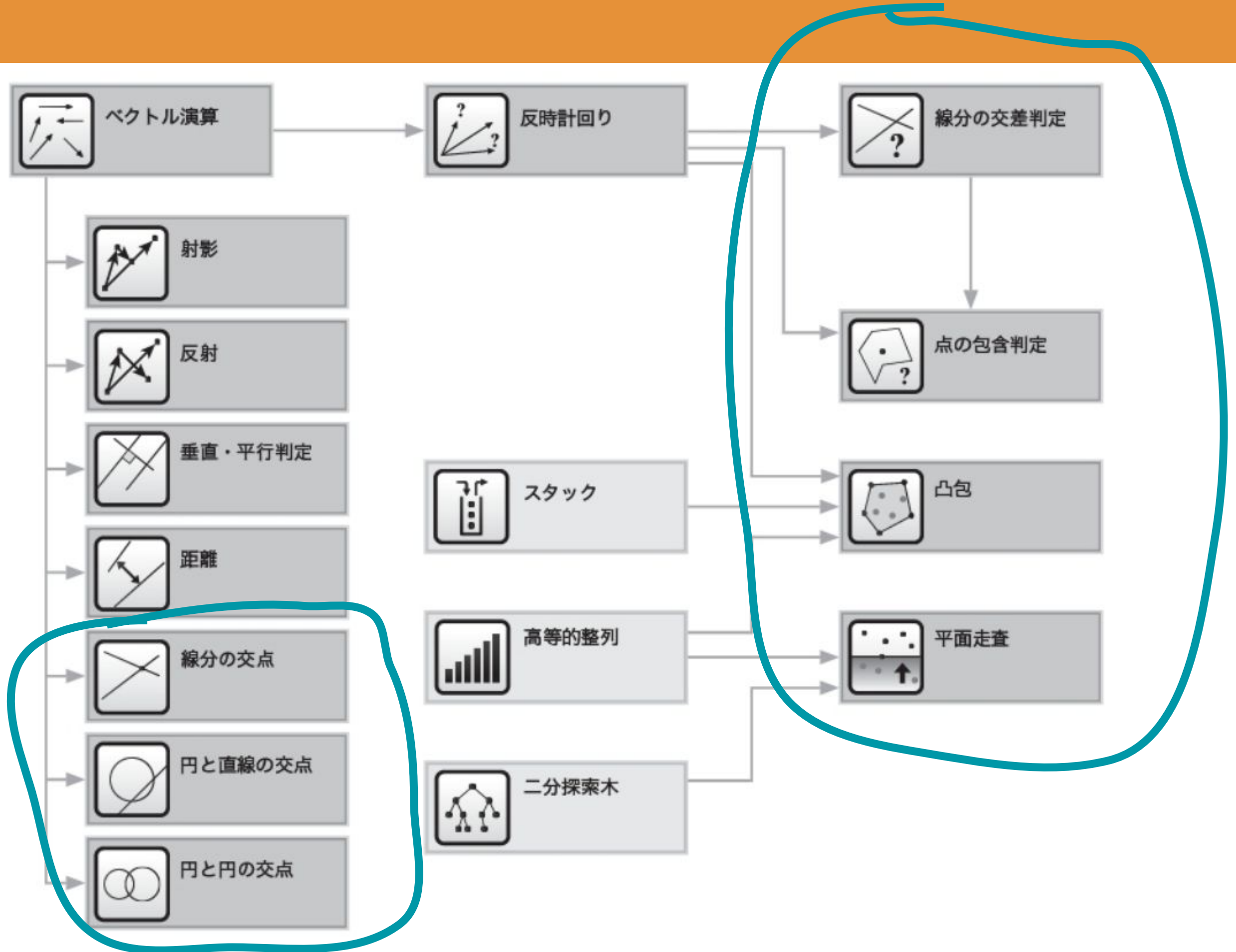
16.8~16.14章

計算幾何学（後半）

sean

スキル



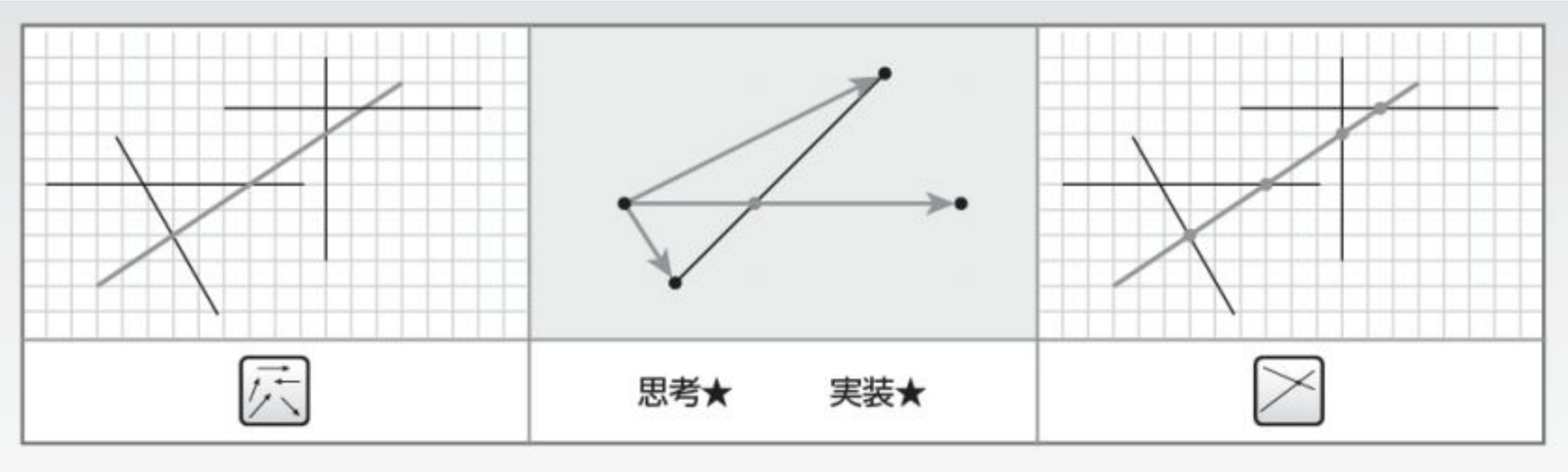
ベクトル演算
反時計回り
線分の交差判定
射影
反射
垂直・平行判定
距離
線分の交点
円と直線の交点
円と円の交点
点の包含判定
スタック
凸包
高等的整列
平面走査
二分探索木

目次

目次

線分の交点

交わる２本の線分の交点



線分の交点

制約 $1 \le q \le 1{,}000$ クエリ数

$-10{,}000 \le x_{p_i}, y_{p_i} \le 10{,}000$ 座標の範囲

$p0, p1$ は同一でない。

$p2, p3$ は同一でない。

$s1, s2$ は交点を持ち、線分で重なることはない。

線分 s1=(p0, p1)
線分 s2=(p2, p3)

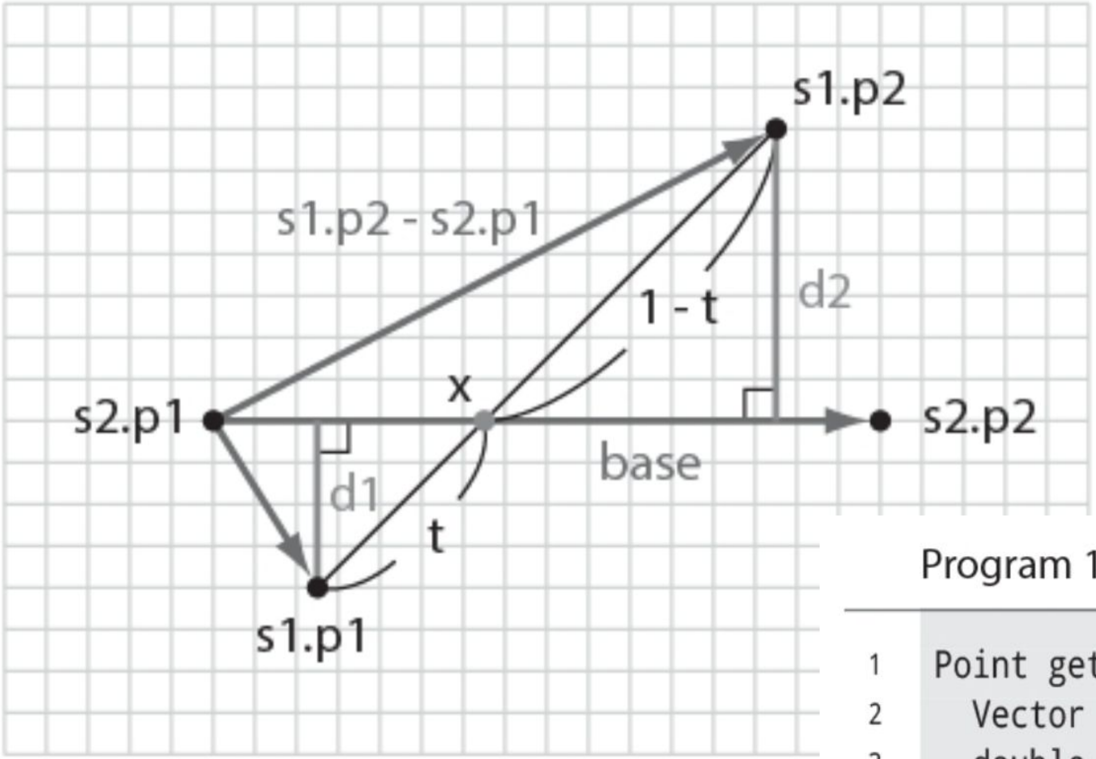
入力例

```
3 クエリ数
0 0 2 0 1 1 1 -1 p0 p1 p2 p3
0 0 1 1 0 1 1 0
0 0 1 1 1 0 0 1
```

出力例

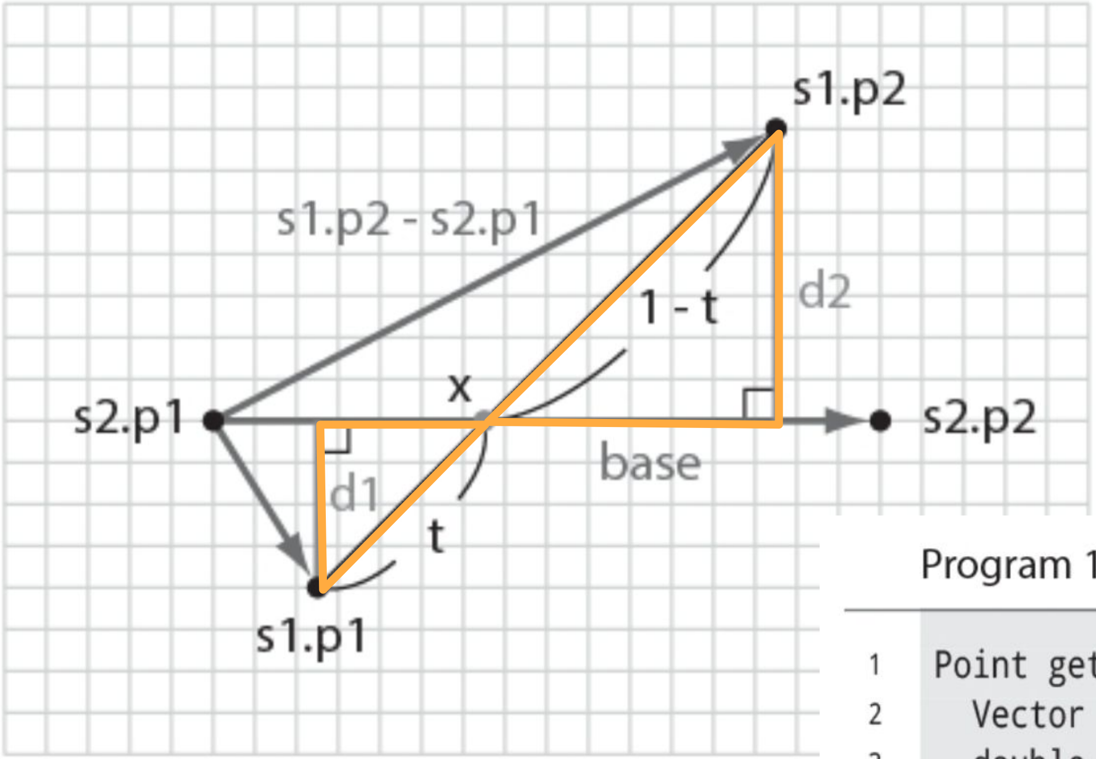
```
1.0000000000 0.0000000000 交点
0.5000000000 0.5000000000
0.5000000000 0.5000000000
```

入力：２つの線分
出力：交わる座標



Program 16.23: 線分 *s*1 と線分 *s*2 の交点

```
Point getCrossPoint(Segment s1, Segment s2) {
  Vector base = s2.p2 - s2.p1;
  double d1 = abs(cross(base, s1.p1 - s2.p1));
  double d2 = abs(cross(base, s1.p2 - s2.p1));
  double t = d1 / (d1 + d2);
  return s1.p1 + (s1.p2 - s1.p1) * t;
}
```



$$t:(1-t) = d1:d2$$

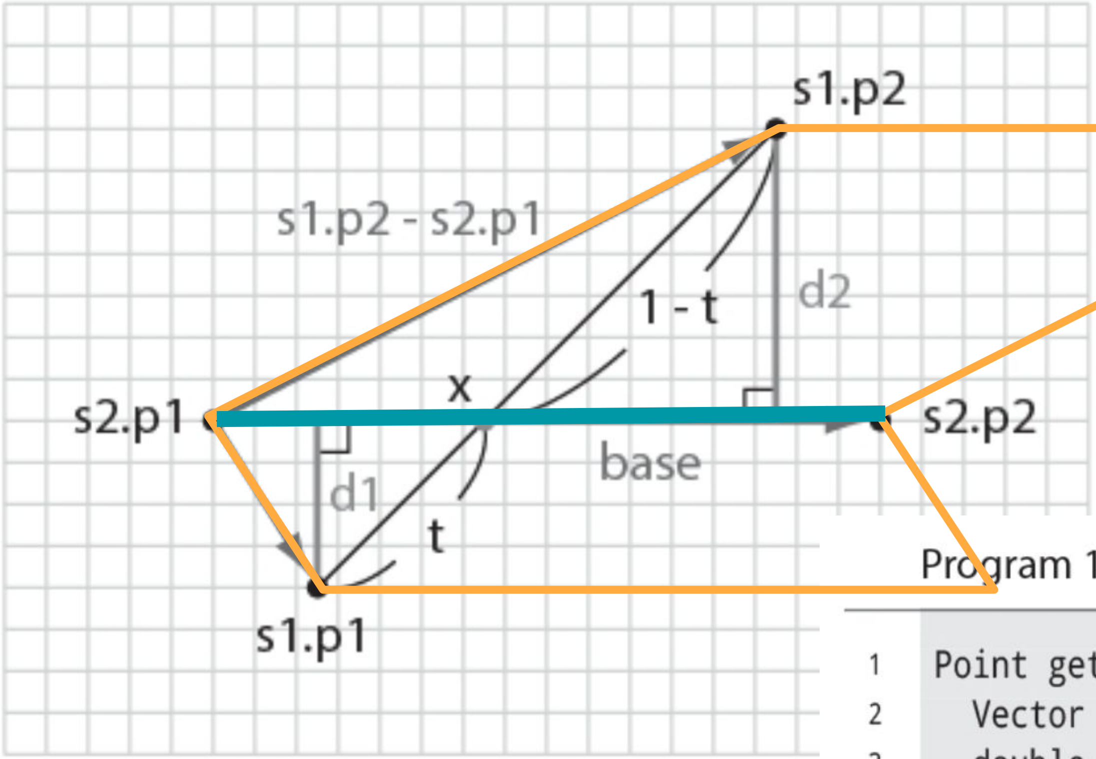
$$t = d1/(d1+d2)$$

Program 16.23: 線分 $s1$ と線分 $s2$ の交点

```
Point getCrossPoint(Segment s1, Segment s2) {
  Vector base = s2.p2 - s2.p1;
  double d1 = abs(cross(base, s1.p1 - s2.p1));
  double d2 = abs(cross(base, s1.p2 - s2.p1));
  double t = d1 / (d1 + d2);
  return s1.p1 + (s1.p2 - s1.p1) * t;
}
```

$$t:(1-t) = d1:d2$$

$$t = d1/(d1+d2)$$



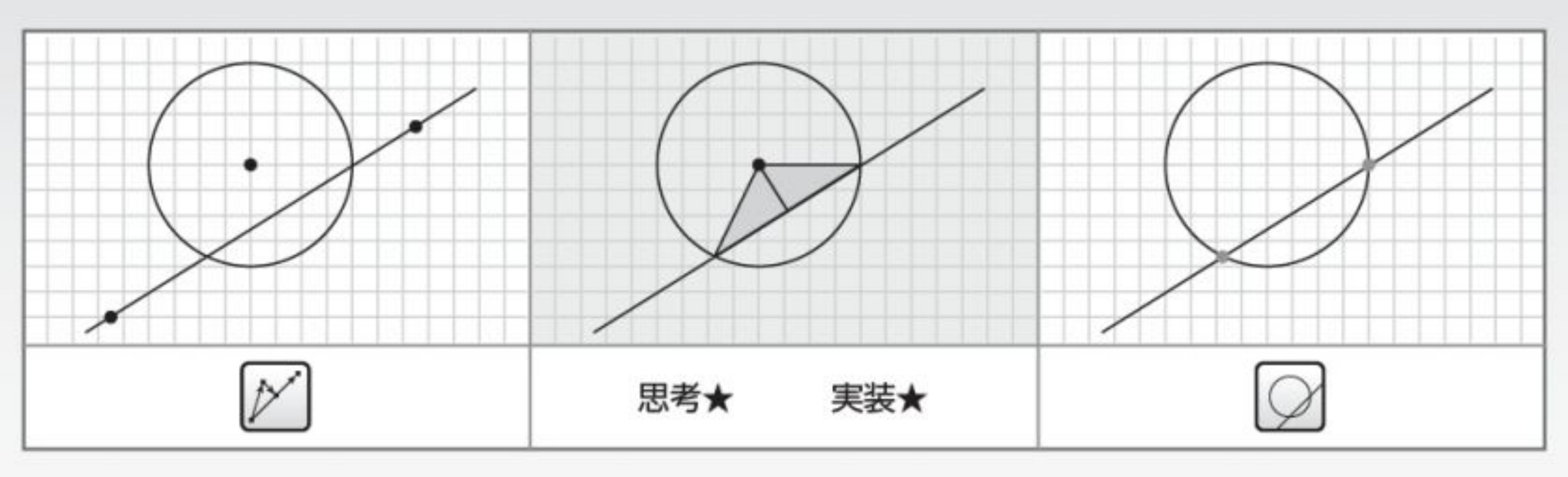
Program 16.23: 線分 $s1$ と線分 $s2$ の交点

```
Point getCrossPoint(Segment s1, Segment s2) {
  Vector base = s2.p2 - s2.p1;
  double d1 = abs(cross(base, s1.p1 - s2.p1));
  double d2 = abs(cross(base, s1.p2 - s2.p1));
  double t = d1 / (d1 + d2);
  return s1.p1 + (s1.p2 - s1.p1) * t;
}
```

目次

円と直線の交点

交点は２つあると前提



円と直線の交点

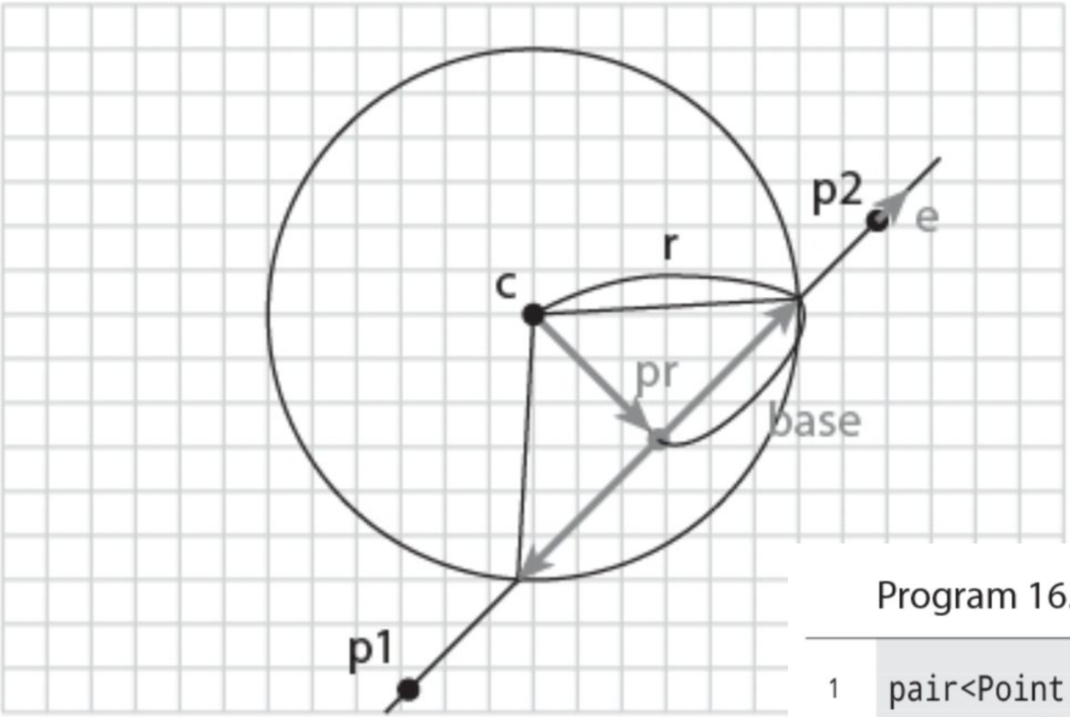
制約　$p1$ と $p2$ は異なる。
円と直線は必ず交点を持つ。
$1 \le q \le 1{,}000$
$-10{,}000 \le cx, cy, x_1, y_1, x_2, y_2 \le 10{,}000$
$1 \le r \le 10{,}000$　半径の範囲

入力例

```
2 1 1 cx cy r
2 クエリ数
0 1 4 1 x1 y1 x2 y2
3 0 3 3
```

出力例

```
1.000000 1.000000 3.000000 1.000000  ２つの交点
3.000000 1.000000 3.000000 1.000000
```



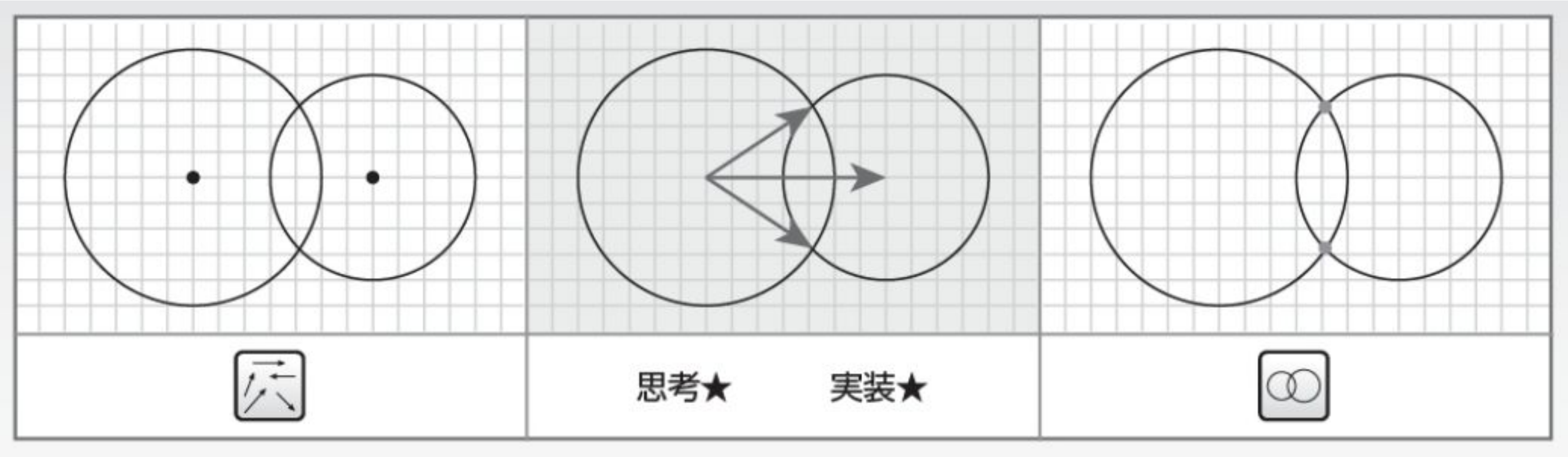
Program 16.24: 円 c と線分 l の交点

```
pair<Point, Point> getCrossPoints(Circle c, Line l) {
  assert(intersect(c, l));
  Vector pr = project(l, c.c);
  Vector e = (l.p2 - l.p1) / abs(l.p2 - l.p1);
  double base = sqrt(c.r * c.r - norm(pr - c.c));
  return make_pair(pr + e * base, pr - e * base);
}
```

目次

円と円の交点

交点は２つあると前提



円と円の交点

制約　2つの円は交点を持ち、中心の座標は異なる。

$-10{,}000 \le c1x, c1y, c2x, c2y \le 10{,}000$

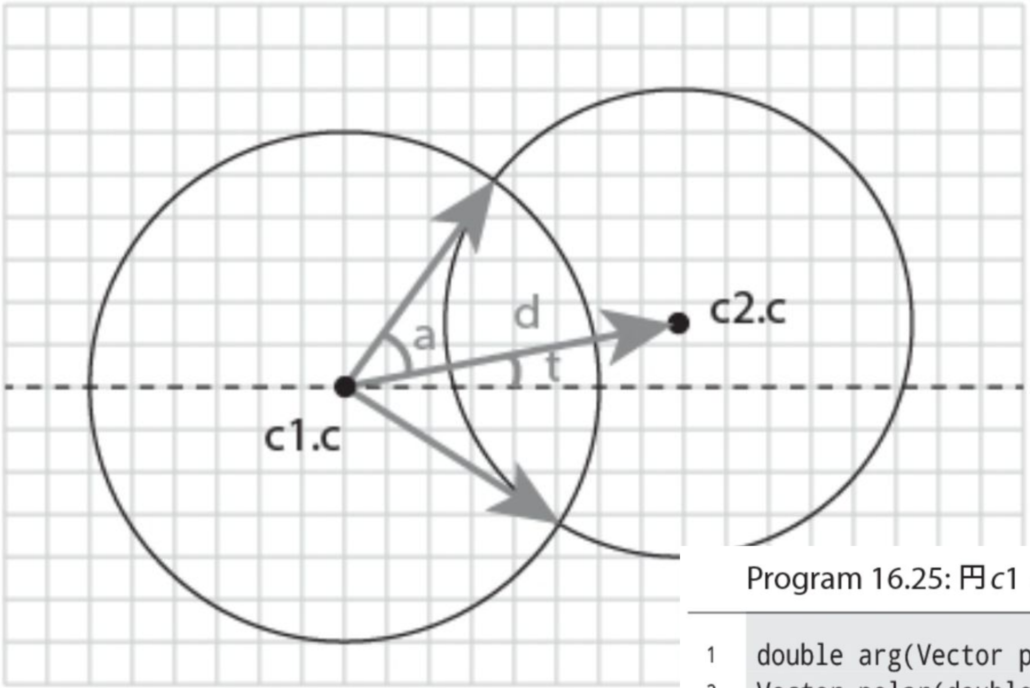
$1 \le c1r, c2r \le 10{,}000$

入力例 1

```
0 0 2 c1x c1y c1r
2 0 2 c2x c2y c2r
```

出力例 1

```
1.0000000 -1.7320508 1.0000000 1.7320508 2つの交点
```



Program 16.25: 円$c1$と円$c2$の交点

```
double arg(Vector p) { return atan2(p.y, p.x); }
Vector polar(double a, double r) { return Point(cos(r) * a, sin(r) * a); }

pair<Point, Point> getCrossPoints(Circle c1, Circle c2) {
  assert(intersect(c1, c2));
  double d = abs(c1.c - c2.c);
  double a = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2 * c1.r * d));
  double t = arg(c2.c - c1.c);
  return make_pair(c1.c + polar(c1.r, t + a), c1.c + polar(c1.r, t - a));
}
```

目次

点の内包

点が多角形の中か辺上か外かを求める

点の内包

制約　$3 \le n \le 100$　多角形の点の数

$1 \le q \le 1{,}000$

$-10{,}000 \le x_i, y_i, x, y \le 10{,}000$

多角形の点の座標はすべて異なる

多角形の辺は共有する端点のみで交差する

入力例

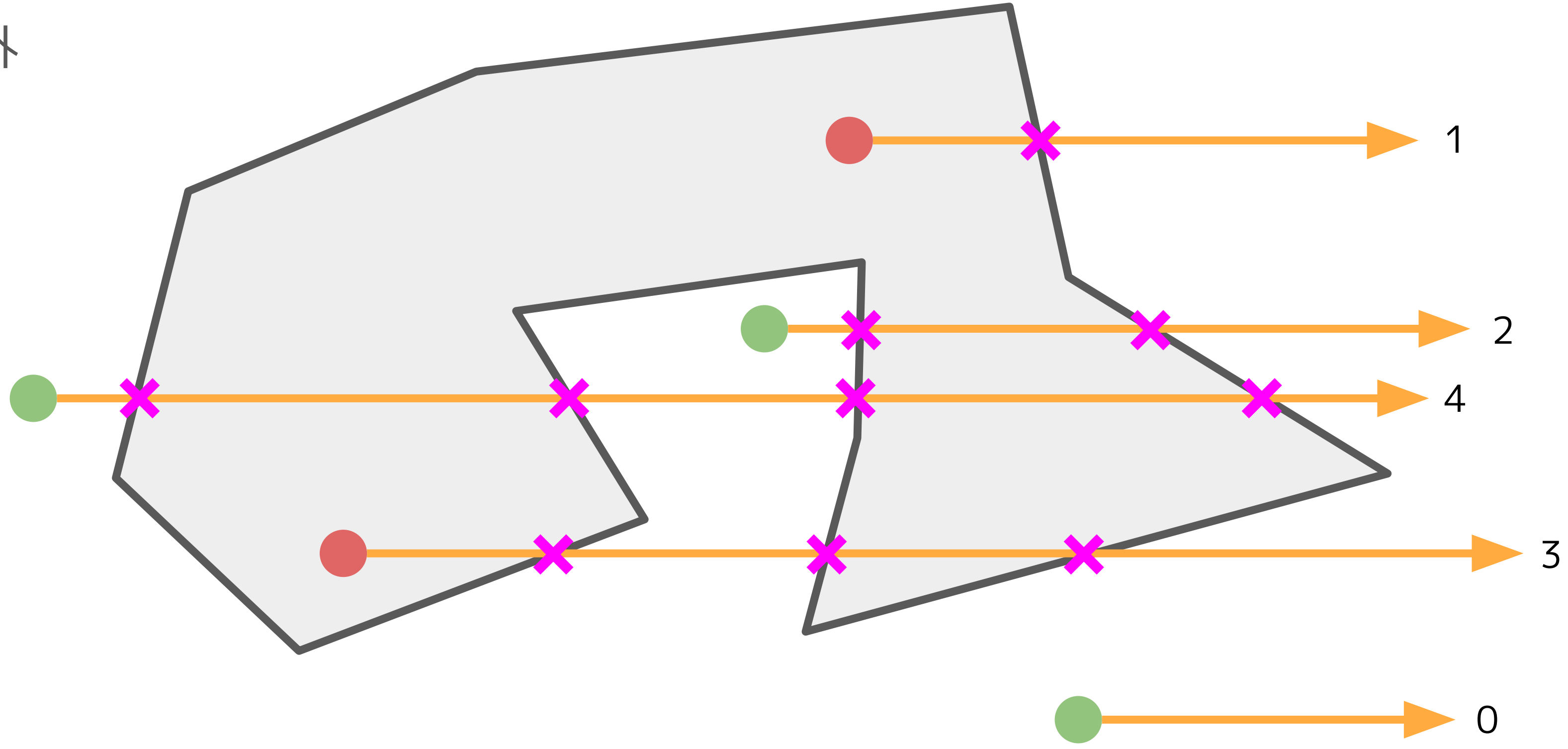
```
4     多角形の点の数
0 0   点の座標
3 1
2 3
0 3
3     クエリの数
2 1   クエリ点
0 2
3 2
```

出力例

```
2     2=多角形の中
1     1=多角形の辺の上
0     0=それ以外（外）
```

点の内包

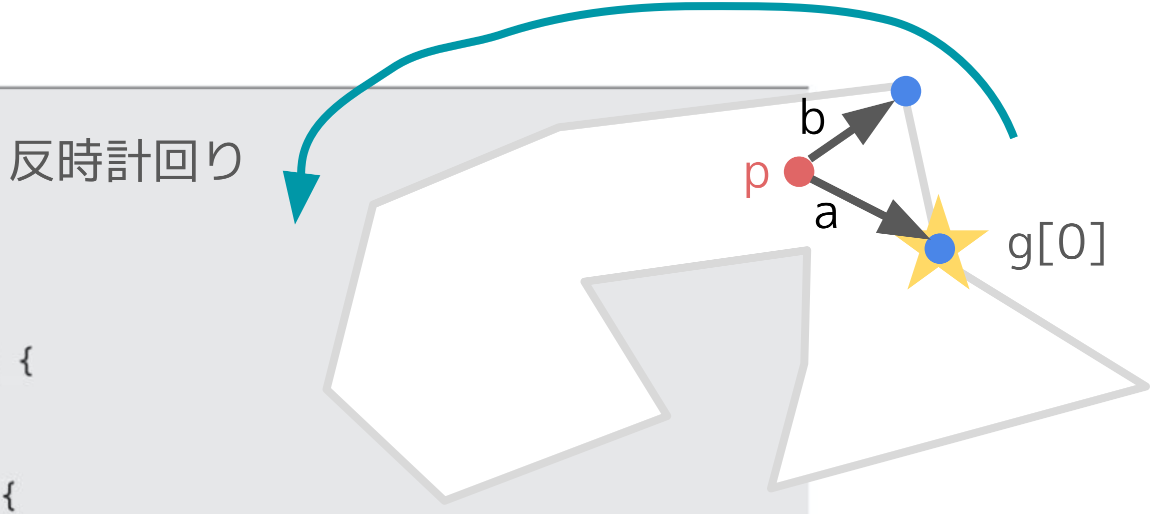
奇数→中
偶数→外



点の内包

Program 16.26: 点の内包

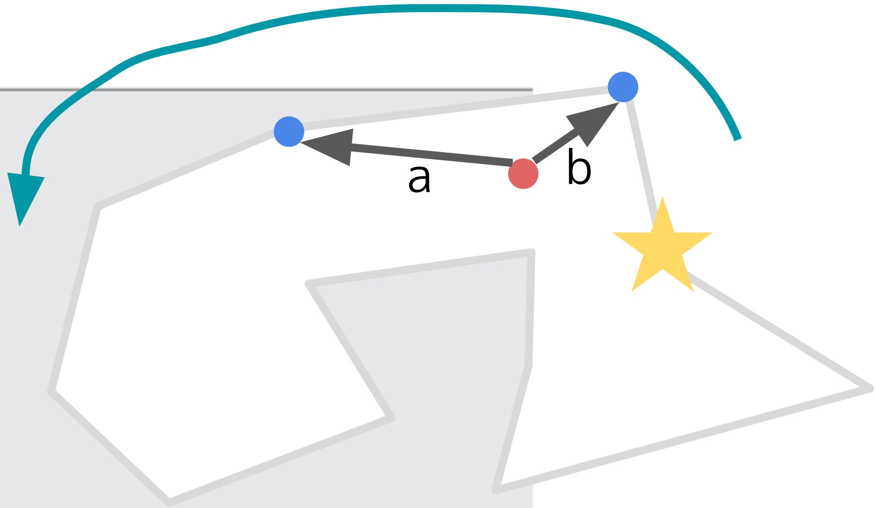
```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```



点の内包

Program 16.26: 点の内包

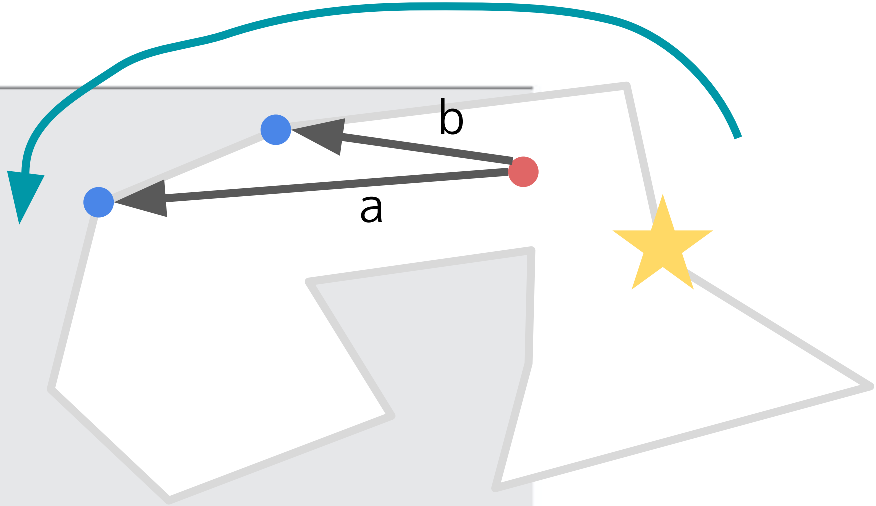
```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```



点の内包

Program 16.26: 点の内包

```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```

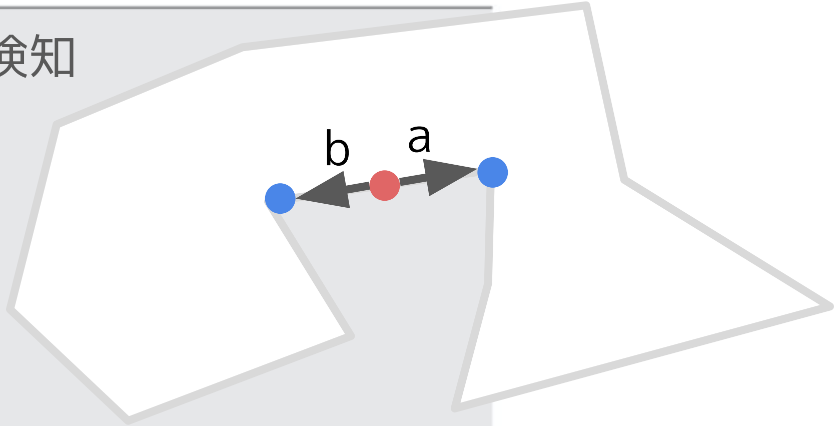


点の内包

Program 16.26: 点の内包

```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```

辺上の検知

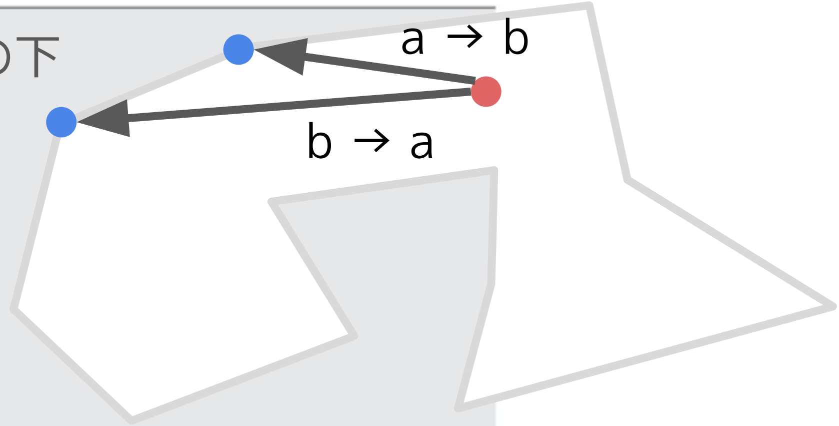


点の内包

Program 16.26: 点の内包

```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```

aは必ずbの下



点の内包

Program 16.26: 点の内包

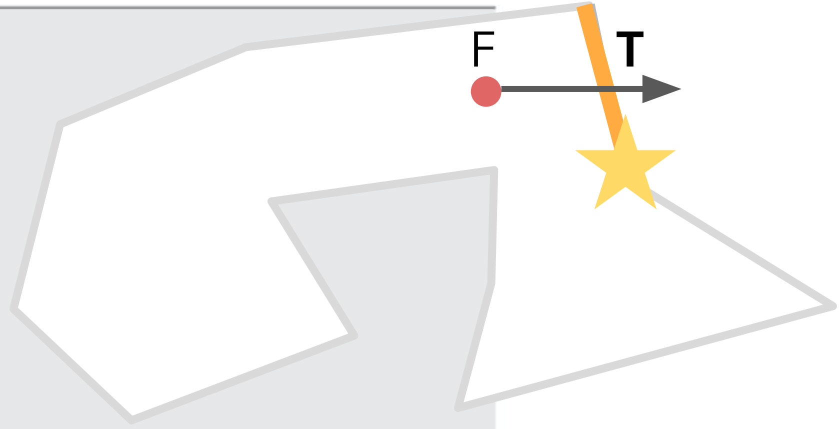
```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```

aからbは反時計回り

点の内包

Program 16.26: 点の内包

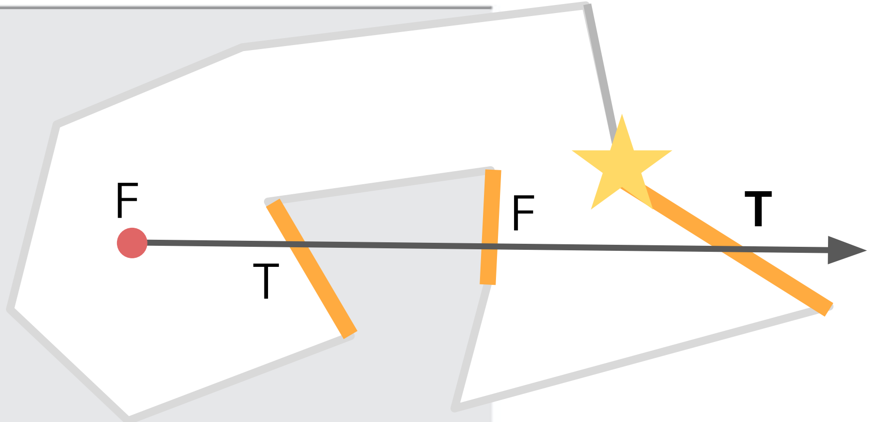
```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```



点の内包

Program 16.26: 点の内包

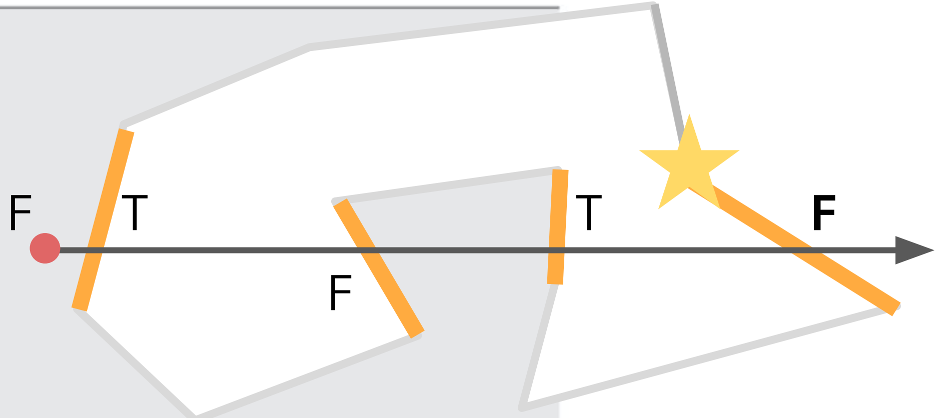
```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```



点の内包

Program 16.26: 点の内包

```
/*
  IN 2
  ON 1
  OUT 0
  */
int contains(Polygon g, Point p) {
  int n = g.size();
  bool x = false;
  for ( int i = 0; i < n; i++ ) {
    Point a = g[i] - p, b = g[(i + 1) % n] - p;
    if ( abs(cross(a, b)) < EPS && dot(a, b) < EPS ) return 1;
    if ( a.y > b.y ) swap(a, b);
    if ( a.y < EPS && EPS < b.y && cross(a, b) > EPS ) x = !x;
  }
  return ( x ? 2 : 0 );
}
```



目次

凸包

与えられた点の集合を包む多角形を求める

凸包

制約 $3 \leq n \leq 100{,}000$ 点の数

$-10{,}000 \leq x_i, y_i \leq 10{,}000$

点の座標はすべて異なる

入力例 1

```
7   点の数
2 1   点
0 0
1 2
2 2
4 2
1 3
3 3
```

出力例 1

```
5   多角形の点の数
0 0   点
2 1
4 2
3 3
1 3
```

アンドリューのアルゴリズム

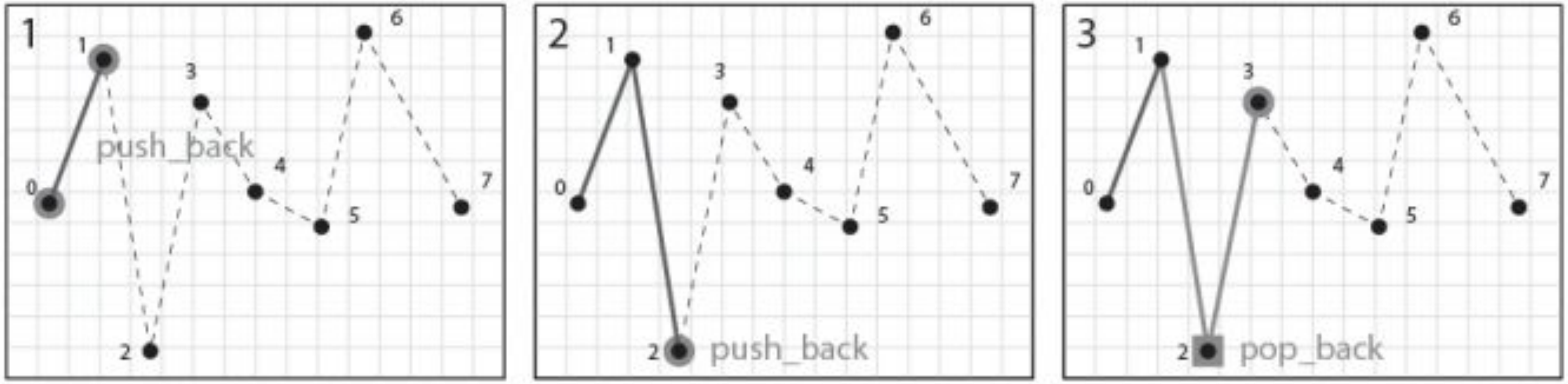
与えられた点をx軸においてソート

ベクトル・リスト的なデータ構造を使用

最後に得られる点のリストが凸になっている

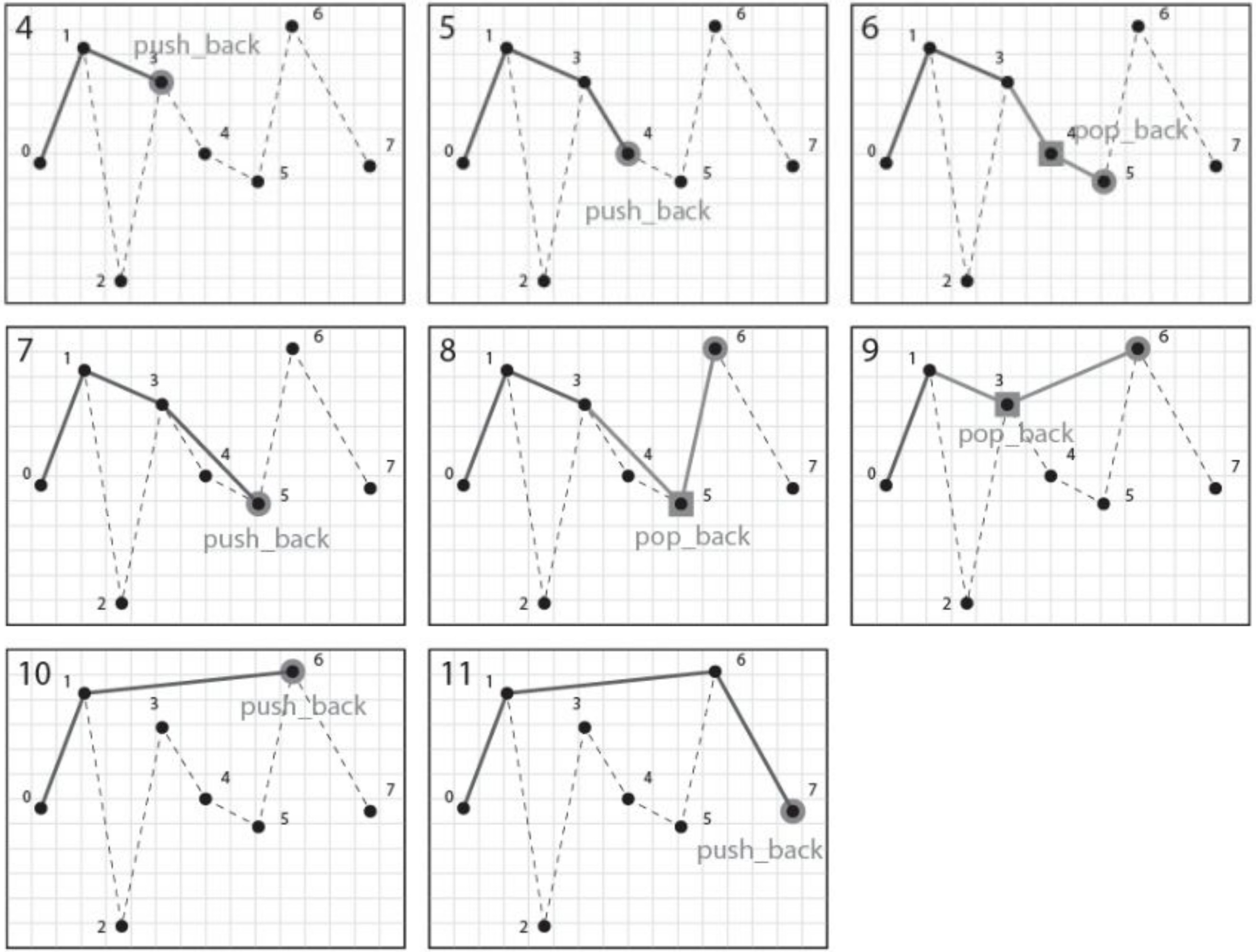
アンドリューのアルゴリズム

凸苞の上部を獲得する例



0→1
0→2
時計回り

1→2
1→3
反時計回り
凸に戻るまで後ろから削除
（２がpopされる）

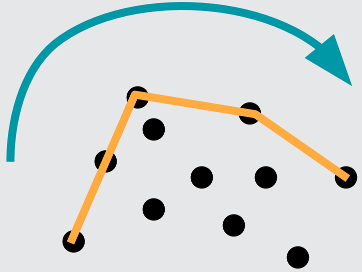

4
push_back
5
push_back
6
pop_back
7
push_back
8
pop_back
9
pop_back
10
push_back
11
push_back

Program 16.27: アンドリューのアルゴリズム

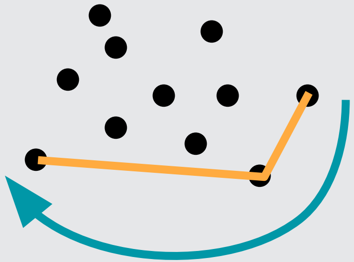
```
Polygon andrewScan( Polygon s ) {
  Polygon u, l;
  if ( s.size() < 3 ) return s;
  sort(s.begin(), s.end()); // x, y を基準に昇順にソート
  // x が小さいものから 2 つ u に追加
  u.push_back(s[0]);
  u.push_back(s[1]);
  // x が大きいものから 2 つ l に追加
  l.push_back(s[s.size() - 1]);
  l.push_back(s[s.size() - 2]);

  // 凸包の上部を生成
  for ( int i = 2; i < s.size(); i++ ) {
    for ( int n = u.size(); n >= 2 && ccw(u[n-2], u[n-1], s[i]) != CLOCKWISE; n--) {
      u.pop_back();
    }
    u.push_back(s[i]);
  }

```



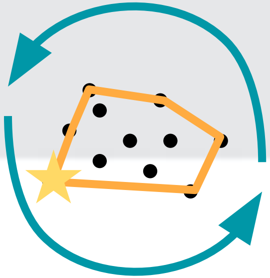
```
  // 凸包の下部を生成
  for ( int i = s.size() - 3; i >= 0; i-- ) {
    for ( int n = l.size(); n >= 2 && ccw(l[n-2], l[n-1], s[i]) != CLOCKWISE; n--) {
      l.pop_back();
    }
    l.push_back(s[i]);
  }
```



反？

```
  // 時計回りになるように凸包の点の列を生成
  reverse(l.begin(), l.end());
  for ( int i = u.size() - 2; i >= 1; i-- ) l.push_back(u[i]);

  return l;
}
```



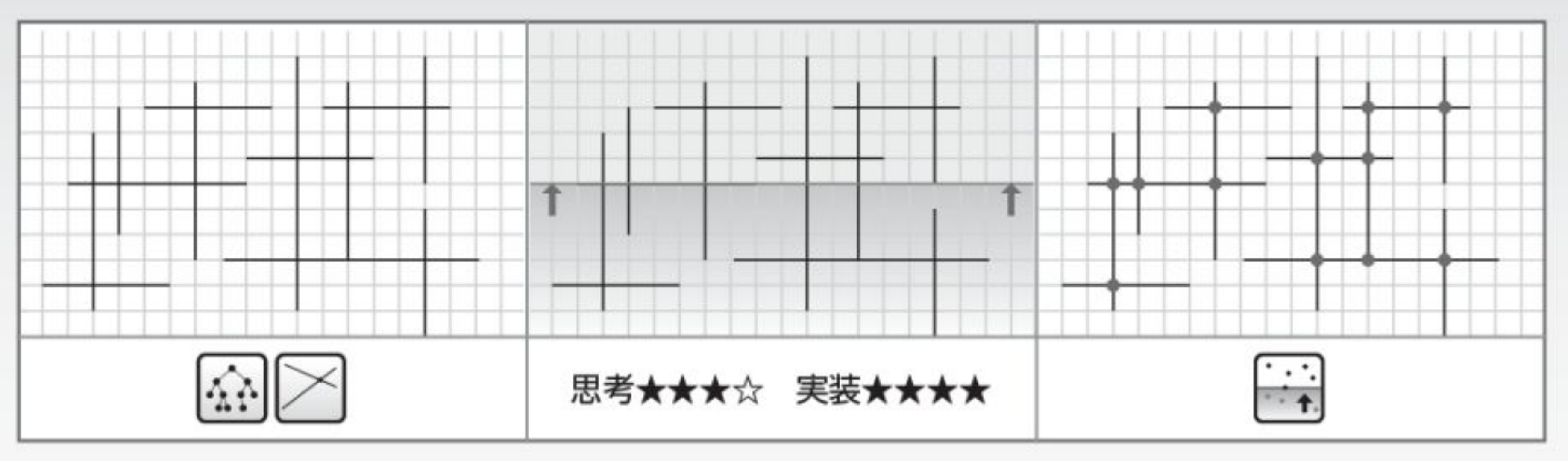
考察

点を一つ一つ処理するのはO(n)だが,
事前のソートが必要であるためO(nlogn)

目次

線分交差問題

軸に平行な線の集合の交点の総数



線分交差問題

制約　$1 \leq n \leq 100{,}000$　線分の数

互いに平行な2つ以上の線分が、線分あるいは点で重なることはない。

交点の数は1,000,000を超えない。

$-1{,}000{,}000{,}000 \leqq x_1, y_1, x_2, y_2 \leqq 1{,}000{,}000{,}000$

入力例

```
6  線分の数
2 2 2 5  線分の x1 y1 x2 y2
1 3 5 3
4 1 4 4
5 2 7 2
6 1 6 3
6 5 6 7
```

出力例

```
3  交点の数
```

平面走査 (sweep)

全ての線分同士で交点の確認を行うと$O(n^2)$

一つ一つ独立に確認するのではなく，
一遍に処理を行っていくかたち

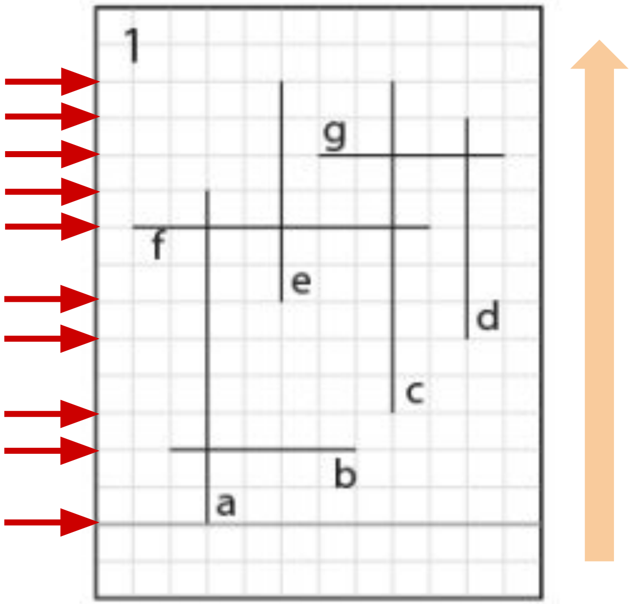
平面走査

今回は下から上へと走査する

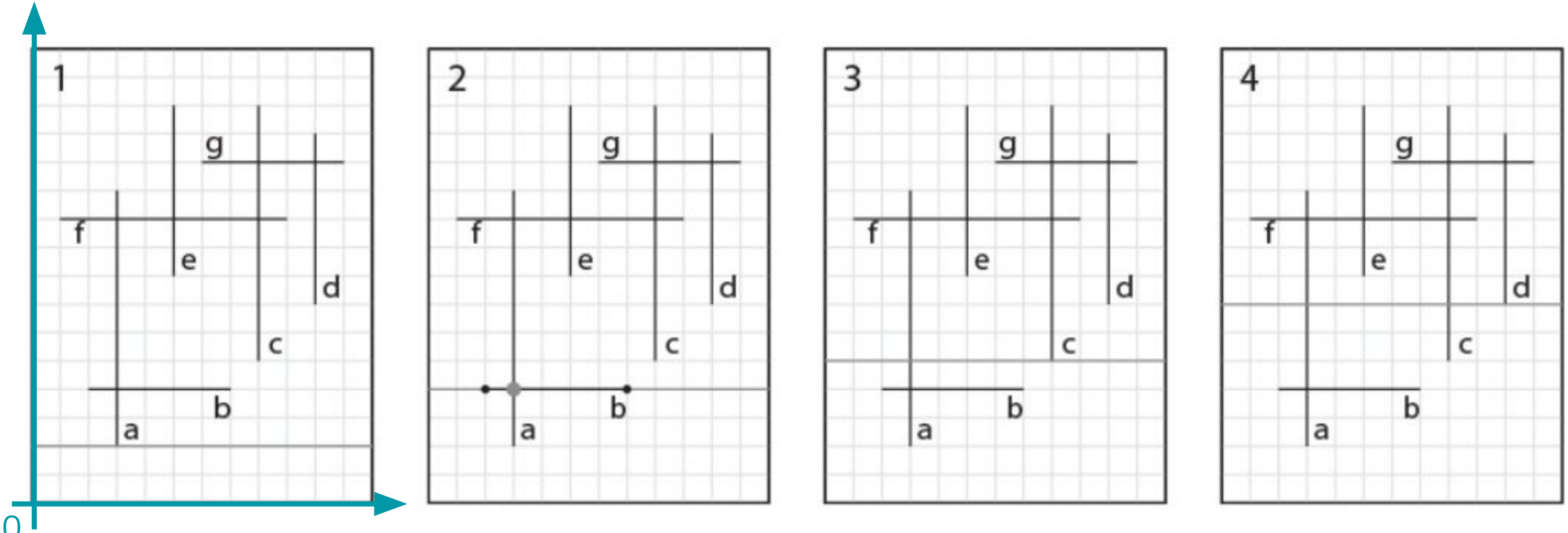
線分の端点のリストEPをyでソートし，
途中で走査線が停まるところが決まる

範囲における確認を頻繁に行うため，
空の二分探索木Tを用意

EPから順に取り出し，
上端点，下端点，左端点であるかにより処理が決まる



平面走査



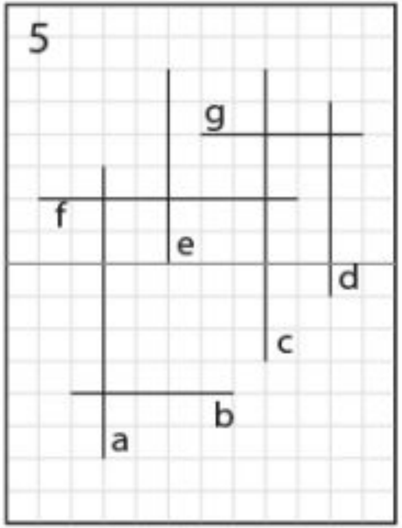
下端点@y=2：
Tに3を追加

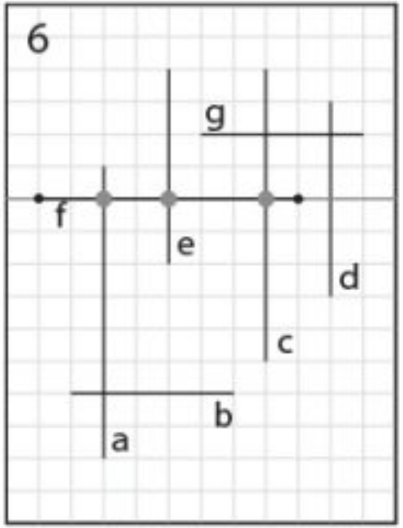
左端点@y=4：
2<=x<=7をTで確認
→3
なのでcount+=1

下端点@y=5：
Tに8を追加

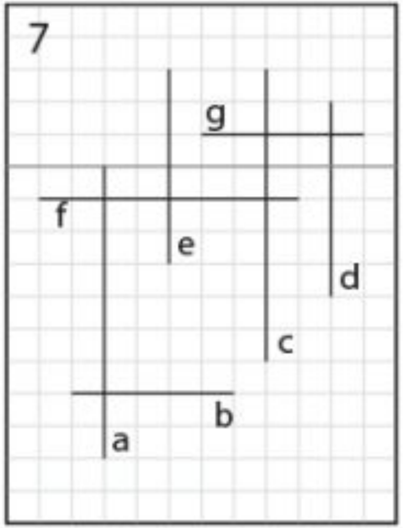
…

平面走査

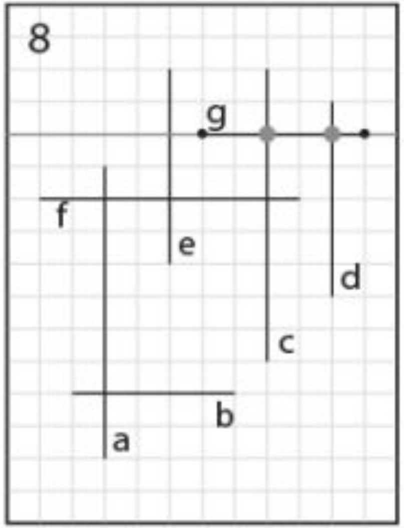




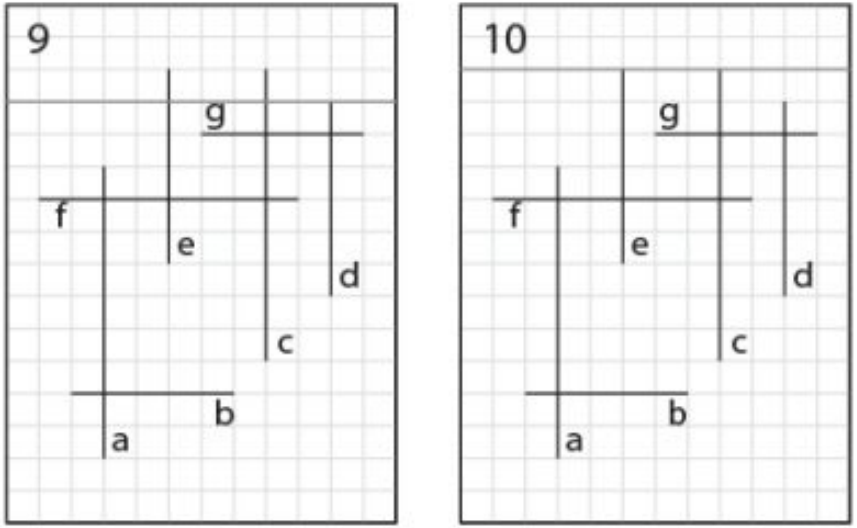
左端点@y=10：
1<=x<=9をTで確認
→3,5,8
なので count+=3



上端点@y=11：
Tから3を削除



平面走査



完了時：
Tは空で
全ての交点を検出

考察

探索操作がO(logn)で，端点は2n個あるので
O(2nlogn)→O(nlogn)

k個の交点の記録も含むとO(nlogn + k)

Program 16.28: 平面走査

```
// 端点の種類
#define BOTTOM 0
#define LEFT 1
#define RIGHT 2
#define TOP 3

class EndPoint {
public:
  Point p;
  int seg, st; // 入力線分のID, 端点の種類
  EndPoint() {}
  EndPoint(Point p, int seg, int st): p(p), seg(seg), st(st) {}

  bool operator < (const EndPoint &ep) const {
    // y 座標が小さい順に整列
    if ( p.y == ep.p.y ) {
      return st < ep.st; // y が同一の場合は、下端点、左端点、右端点、上端点の順に並べる
    } else return p.y < ep.p.y;
  }
};

EndPoint EP[2 * 100000]; // 端点のリスト

// 線分交差問題：マンハッタン幾何
```

```
int manhattanIntersection(vector&lt;Segment&gt; S) {
  int n = S.size();

  for ( int i = 0, k = 0; i < n; i++ ) {
    // 端点 p1, p2 が左下を基準に並ぶように調整
    if ( S[i].p1.y == S[i].p2.y ) {
      if ( S[i].p1.x > S[i].p2.x ) swap(S[i].p1, S[i].p2);
    } else if ( S[i].p1.y > S[i].p2.y ) swap(S[i].p1, S[i].p2);

    if ( S[i].p1.y == S[i].p2.y ) { // 水平線分を端点リストに追加
      EP[k++] = EndPoint(S[i].p1, i, LEFT);
      EP[k++] = EndPoint(S[i].p2, i, RIGHT);
    } else {                         // 垂直線分を端点リストに追加
      EP[k++] = EndPoint(S[i].p1, i, BOTTOM);
      EP[k++] = EndPoint(S[i].p2, i, TOP);
    }
  }

```

```
  set<int> BT;              // 二分探索木
  BT.insert(1000000001);  // 番兵を設置
  int cnt = 0;

  for ( int i = 0; i < 2 * n; i++ ) {
    if ( EP[i].st == TOP ) {
      BT.erase(EP[i].p.x);  // 上端点を削除
    } else if ( EP[i].st == BOTTOM ) {
      BT.insert(EP[i].p.x); // 下端点を追加
    } else if ( EP[i].st == LEFT ) {
      set<int>::iterator b = lower_bound(BT.begin(), BT.end(), S[EP[i].seg].p1.x); // O(log n)
      set<int>::iterator e = upper_bound(BT.begin(), BT.end(), S[EP[i].seg].p2.x); // O(log n)
      cnt += distance(b, e); // b と e の距離（点の数）を加算, O(k)
    }
  }

  return cnt;
}
```

16.14 その他の問題

- Closest Pair
- Diameter of Convex Polygon
  - 多角形の最遠頂点対間距離
- Convex Cut
  - 多角形を線分で（２つに？）切断

おわり

16.8　線分の交点
16.9　円と直線の交点
16.10 円と円の交点
16.11 点の内包
16.12 凸包
16.13 線分交差問題
16.14 その他の問題