GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: BUSCAR UN PAR DE SEGMENTOS QUE SE CRUZAN

1. Introducción

Dado n segmentos de línea en el plano. Se requiere verificar si al menos dos de ellos se cruzan entre sí. Si la respuesta es sí, imprima este par de segmentos que se cruzan; basta con elegir cualquiera de ellas entre varias respuestas.

2. Conocimientos previos

2.1. Punto 2D

El punto en la geometría es uno de los entes fundamentales de la geometría, junto con la recta y el plano, pues son considerados conceptos primarios, es decir, que solo es posible describirlos en relación con otros elementos similares o parecidos. El punto carece de largo, espesor o grosor. Se suelen describir apoyándose en los postulados característicos, que determinan las relaciones entre los entes geométricos fundamentales. El punto es la unidad más simple, irreductiblemente mínima, de la comunicación visual; es una figura geométrica sin dimensión, tampoco tiene longitud, área, volumen, ni otro ángulo dimensional. No es un objeto físico. Describe una posición en el plano, determinada respecto de un sistema de coordenadas preestablecidas.

2.2. Segmento

En geometría, el segmento es un fragmento de la recta que está comprendido entre dos puntos, llamados puntos extremos o finales. Así, dado dos puntos A y B, se llama segmento AB a la intersección de la semirrecta de origen A que contiene al punto B con la semirrecta de origen B que contiene al punto A. Los puntos A y B son extremos del segmento y los puntos sobre la recta a la que pertenece el segmento.

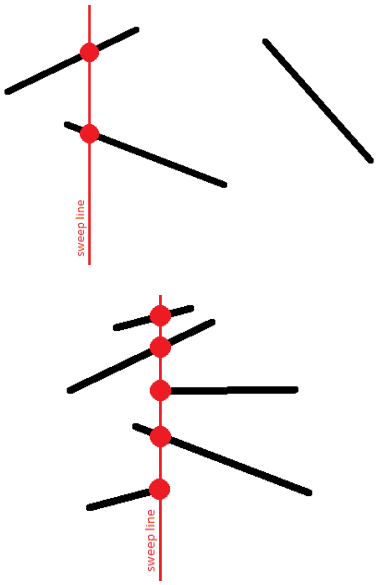
3. Desarrollo

El algoritmo de solución ingenua es iterar sobre todos los pares de segmentos en $O(n^2)$ y verifique para cada par si se cruzan o no. Vamos a elaborar un algoritmo que sea mas eficiente que la solución ingenua basado en el **algoritmo de línea de barrido (*sweep line*)**.

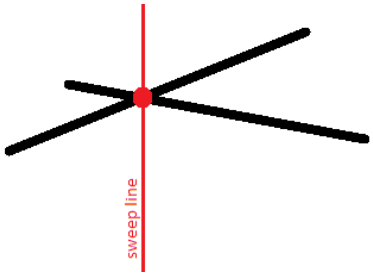
Dibujemos una línea vertical $x = -\infty$ mentalmente y comience a mover esta línea hacia la derecha. En el curso de su movimiento, esta línea se encontrará con segmentos, y cada vez que un segmento se interseca con nuestra línea, se interseca exactamente en un punto (supondremos que no hay segmentos verticales).

Así, para cada segmento, en algún momento, su punto aparecerá en la línea de barrido, luego, con el movimiento de la línea, este punto se moverá y, finalmente, en algún momento, el segmento desaparecerá de la línea.

Estamos interesados en el **orden relativo de los segmentos** a lo largo de la vertical. Es decir, almacenaremos una lista de segmentos que cruzan la línea de barrido en un momento dado, donde los segmentos se ordenarán por su y-coordenada en la línea de barrido.



Este orden es interesante porque los segmentos que se cruzan tendrán el mismo y-coordinar al menos en un momento:



Formulamos declaraciones clave:

- Para encontrar un par que se interseque, es suficiente considerar **solo segmentos adyacentes** en cada posición fija de la línea de barrido.
- Basta con considerar la línea de barrido no en todas las posiciones reales posibles $(-\infty \ldots + \infty)$, pero **sólo en aquellas posiciones en las que aparecen nuevos segmentos o desaparecen los antiguos**. En otras palabras, es suficiente limitarse solo a las posiciones iguales a las abscisas de los puntos finales de los segmentos.
- Cuando aparece un nuevo segmento de línea, basta con **insertarlo** en la ubicación deseada en la lista obtenida para la línea de barrido anterior. Solo debemos verificar la intersección del **segmento agregado con sus vecinos inmediatos en la lista de arriba y abajo**.
- Si el segmento desaparece, basta con **eliminarlo** de la lista actual. Después de eso, es necesario **verificar la intersección de los vecinos superior e inferior en la lista**.
- No existen otros cambios en la secuencia de segmentos de la lista, excepto los descritos. No

se requieren otros controles de intersección.

Para comprender la veracidad de estas afirmaciones bastan las siguientes observaciones:

1. Dos segmentos disjuntos nunca cambian su orden relativo. De hecho, si un segmento fue primero más alto que el otro y luego se volvió más bajo, entonces entre estos dos momentos hubo una intersección de estos dos segmentos.
2. Dos segmentos que no se intersecan tampoco pueden tener el mismo y-coordenadas.
3. De esto se deduce que en el momento de la aparición del segmento podemos encontrar la posición de este segmento en la cola, y ya no tendremos que reorganizar este segmento en la cola: **su orden relativo a otros segmentos en la cola no cambiará**.
4. Dos segmentos que se intersecan en el momento de su punto de intersección serán vecinos entre sí en la cola.
5. Por lo tanto, para encontrar pares de segmentos de línea que se intersecan, es suficiente verificar la intersección de todos y solo aquellos pares de segmentos que en algún momento durante el movimiento de la línea de barrido al menos una vez fueron vecinos entre sí.

   Es fácil notar que solo basta con verificar el segmento agregado con sus vecinos superior e inferior, así como al eliminar el segmento, sus vecinos superior e inferior (que después de la eliminación se convertirán en vecinos entre sí).
6. Cabe señalar que en una posición fija de la línea de barrido, primero debemos **agregar todos los segmentos** que comienzan en esta coordenada x, y solo **luego eliminar todos los segmentos** que terminan aquí.

   Por lo tanto, no perdemos la intersección de los segmentos en el vértice: es decir, los casos en que dos segmentos tienen un vértice común.
7. Tenga en cuenta que **los segmentos verticales** en realidad no afectan la corrección del algoritmo. Estos segmentos se distinguen por el hecho de que aparecen y desaparecen al mismo tiempo. Sin embargo, debido al comentario anterior, sabemos que todos los segmentos se agregarán primero a la cola y solo luego se eliminarán. Por lo tanto, si el segmento vertical se cruza con algún otro segmento abierto en ese momento (incluido el vertical), será detectado. **¿En qué lugar de la cola colocar los segmentos verticales?** Después de todo, un segmento vertical no tiene un segmento específico. y-coordenada, se extiende por todo un segmento a lo largo de la y-coordinar. Sin embargo, es fácil entender que cualquier coordenada de este segmento puede tomarse como y-coordinar.

4. Implementación

4.1. C++

```
const double EPS = 1E-9;

struct Point { double x, y; };
```

```
struct Segment {
   Point p, q;
   int id;

   double get_y(double x) const {
      if (abs(p.x - q.x) < EPS) return p.y;
      return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
   }
};

bool intersect1d(double l1, double r1, double l2, double r2) {
   double t;
   if (l1 > r1){
      t = l1; l1 = r1; r1 =t; }
   if (l2 > r2){
      t = l2; l2 = r2; r2 = t;}
   return max(l1, l2) <= min(r1, r2) + EPS;
}

int vec(const Point &a, const Point &b, const Point &c) {
   double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
   return abs(s) < EPS ? 0 : s > 0 ? +1 : -1;
}

bool intersect(const Segment &a, const Segment &b) {
   return intersect1d(a.p.x, a.q.x, b.p.x, b.q.x)
      && intersect1d(a.p.y, a.q.y, b.p.y, b.q.y)
      && vec(a.p, a.q, b.p) * vec(a.p, a.q, b.q) <= 0
      && vec(b.p, b.q, a.p) * vec(b.p, b.q, a.q) <= 0;
}

bool operator<(const Segment &a, const Segment &b) {
   double x = max(min(a.p.x, a.q.x), min(b.p.x, b.q.x));
   return a.get_y(x) < b.get_y(x) - EPS;
}

struct Event {
   double x; int tp, id;
   Event() {}
   Event(double x, int tp, int id):x(x),tp(tp),id(id){}

   bool operator<(const Event &e) const {
      if (abs(x - e.x) > EPS) return x < e.x;
      return tp > e.tp;
   }
};

set<Segment> s;
vector<set<Segment>::iterator> where;
```

```
set<Segment>::iterator prev(set<Segment>::iterator it) {
   return it == s.begin() ? s.end() : --it;
}

set<Segment>::iterator next(set<Segment>::iterator it){return ++it;}

pair<int, int> solve(const vector<Segment> &a) {
   int n = (int) a.size();
   vector<Event> e;
   for (int i = 0; i < n; ++i) {
      e.push_back(Event(min(a[i].p.x, a[i].q.x), +1, i));
      e.push_back(Event(max(a[i].p.x, a[i].q.x), -1, i));
   }
   sort(e.begin(), e.end());
   s.clear();
   where.resize(a.size());
   for (size_t i = 0; i < e.size(); ++i) {
      int id = e[i].id;
      if (e[i].tp == +1) {
         set<Segment>::iterator nxt = s.lower_bound(a[id]), prv = prev(nxt);
         if (nxt != s.end() && intersect(*nxt, a[id]))
            return make_pair(nxt->id, id);
         if (prv != s.end() && intersect(*prv, a[id]))
            return make_pair(prv->id, id);
                 where[id] = s.insert(nxt, a[id]);
       } else {
          set<Segment>::iterator nxt = next(where[id]), prv = prev(where[id]);
          if (nxt != s.end() && prv != s.end() && intersect(*nxt, *prv))
             return make_pair(prv->id, nxt->id);
          s.erase(where[id]);
       }
   }
   return make_pair(-1, -1);
}
```

La función principal aquí es $solve()$, que devuelve el número de segmentos que se cruzan encontrados,o $(-1, -1)$, si no hay intersecciones. La verificación de la intersección de dos segmentos se lleva a cabo mediante la intersect ()función, utilizando un algoritmo basado en el área orientada del triángulo .

La cola de segmentos es la variable global s, a $set < event >$. Los iteradores que especifican la posición de cada segmento en la cola (para la eliminación conveniente de segmentos de la cola) se almacenan en el vector global where.

También se introducen dos funciones auxiliares $prev()$ y $next()$, que devuelven iteradores a los elementos anterior y siguiente (o end(), si no existe).

La constante EPS denota el error de comparar dos números reales (se usa principalmente

cuando se verifica la intersección de dos segmentos).

4.2. Java

```
public Segment[] findIntersection(Segment[] s) {
   int n = s.length;
   Event[] events = new Event[n * 2];
   for (int i = 0, cnt = 0; i < n; ++i) {
      events[cnt++] = new Event(s[i].x1, s[i].y1, 1, s[i]);
      events[cnt++] = new Event(s[i].x2, s[i].y2, -1, s[i]);
   }
   Arrays.sort(events, eventComparator);
   NavigableSet<Segment> set = new TreeSet<>(segmentComparator);
   for (Event event : events) {
      Segment cur = event.segment;
      if (event.type == 1) {
         Segment floor = set.floor(cur);
         if (floor != null && isCrossOrTouchIntersect(cur, floor))
            return new Segment[] { cur, floor };
         Segment ceiling = set.ceiling(cur);
         if (ceiling != null && isCrossOrTouchIntersect(cur, ceiling))
            return new Segment[] { cur, ceiling };
         set.add(cur);
      } else {
        Segment lower = set.lower(cur);
        Segment higher = set.higher(cur);
        if (lower != null && higher != null && isCrossOrTouchIntersect(lower,
           higher))
          return new Segment[] { lower, higher };
        set.remove(cur);
      }
   }
   return null;
}

public class Segment {
   final int x1, y1, x2, y2;

   public Segment(int x1, int y1, int x2, int y2) {
      if (x1 < x2 || x1 == x2 && y1 < y2) {
         this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 = y2;
      } else {
         this.x1 = x2; this.y1 = y2; this.x2 = x1; this.y2 = y1;
      }
   }
}

static final Comparator<Segment> segmentComparator = (a, b)->{
   if (a.x1 < b.x1) {
```

```
      long v = cross(a.x1, a.y1, a.x2, a.y2, b.x1, b.y1);
      if (v != 0) return v > 0 ? -1 : 1;
   } else if (a.x1 > b.x1) {
     long v = cross(b.x1, b.y1, b.x2, b.y2, a.x1, a.y1);
     if (v != 0) return v < 0 ? -1 : 1;
   }
   return Integer.compare(a.y1, b.y1);
};

private class Event {
   final int x, y, type;
   final Segment segment;

   public Event(int x, int y, int type, Segment segment) {
      this.x = x; this.y = y; this.type = type;
      this.segment = segment;
   }
}

static final Comparator<Event> eventComparator = Comparator.<Event>
    comparingInt(e->e.x)
.thenComparingInt(e -> -e.type).thenComparingInt(e->e.y);

static long cross(long ax, long ay, long bx, long by, long cx, long cy){
  return (bx-ax) * (cy-ay) - (by-ay) * (cx-ax);
}

public boolean isCrossOrTouchIntersect(Segment s1, Segment s2) {
  long x1 = s1.x1; long y1 = s1.y1; long x2 = s1.x2;
  long y2 = s1.y2; long x3 = s2.x1; long y3 = s2.y1;
  long x4 = s2.x2; long y4 = s2.y2;
  if (Math.max(x1, x2) < Math.min(x3, x4) || Math.max(x3, x4) < Math.min(x1,
      x2)
     || Math.max(y1, y2) < Math.min(y3, y4) || Math.max(y3, y4) < Math.min(y1,
        y2))
      return false;
  long z1 = (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1);
  long z2 = (x2 - x1) * (y4 - y1) - (y2 - y1) * (x4 - x1);
  long z3 = (x4 - x3) * (y1 - y3) - (y4 - y3) * (x1 - x3);
  long z4 = (x4 - x3) * (y2 - y3) - (y4 - y3) * (x2 - x3);
  return (z1<=0 || z2<=0) && (z1>=0 || z2>=0) && (z3<=0 || z4<=0) && (z3>=0 ||
      z4>=0);
}

public boolean hasIntersection(Segment[] s) {
   for (int i = 0; i < s.length; i++)
      for (int j = i + 1; j < s.length; j++)
        if (isCrossOrTouchIntersect(s[i], s[j])) return true;
   return false;
}
```

5. Complejidad

Por lo tanto, todo el algoritmo no realizará más de $2n$ pruebas en la intersección de un par de segmentos, y realizará $O(n)$ operaciones con una cola de segmentos($O(1)$ operaciones en el momento de aparición y desaparición de cada segmento). Por lo tanto, el comportamiento asintótico final del algoritmo es $O(n \log n)$.

6. Aplicaciones

Bueno la aplicación de este algoritmo es que permite conocer si dado un cojunto de segmentos existen al menos dos que se intersecan.

7. Ejercicios propuestos

A continuación una lista de jercicios que se resuleven aplicando este algoritmo

- TIMUS - 1469. No Smoking!