Soluciones entrenamiento OIE nivel experto (17/04 - 01/05)

Xavier Povill

27 de abril de 2026

Problema A: Closest Distance

`https://lightoj.com/problem/closest-distance`

Resumen del enunciado

Dados cuatro puntos A, B, C y D en el plano, dos personas se mueven simultáneamente: una va de A a B y la otra de C a D. Ambas salen al mismo tiempo, avanzan en línea recta a velocidad constante y llegan a sus destinos al mismo tiempo. ¿Cuál será la mínima distancia entre ellas a lo largo del recorrido?

Solución

La idea clave de la solución es darse cuenta que la distancia entre ambas personas en función del tiempo es una función convexa, así que podemos encontrar su mínimo con una búsqueda ternaria. Explicaremos ahora con más detalle los conceptos implicados, pues son de gran utilidad para resolver otros problemas del mismo estilo.

Funciones convexas

Decimos que una función f es *convexa* si su gráfica tiene "forma de U". Matemáticamente, esto lo caracterizamos como que, para todo x e y en el dominio, la función evaluada en el punto medio entre x e y está por debajo de la media entre $f(x)$ y $f(y)$:

$$f\left(\frac{x+y}{2}\right) \leq \frac{f(x)+f(y)}{2}$$

De hecho, esta condición implica que la misma propiedad se cumple para cualquier media ponderada entre los puntos x e y:[1]

$$f\left(tx+(1-t)y\right) \leq t\cdot f(x)+(1-t)\cdot f(y), \quad \text{para cualquier } t\in[0,1]$$

Geométricamente, una función es convexa si, al unir con un segmento dos puntos cualesquiera de su gráfica, la función queda completamente por debajo del segmento (véase figura 1).

Si tenemos una expresión algebraica para la función, podemos comprovar si es convexa calculando su segunda derivada:

$$f \text{ convexa} \iff f''(x)\geq 0 \quad \text{para todo } x$$

[1] Esto solo es cierto para funciones continuas. Si f no es continua, entonces se toma como definición de convexidad esta condición, mientras que la condición anterior, llamada a veces *midpoint-convexity*, es ligeramente más débil.

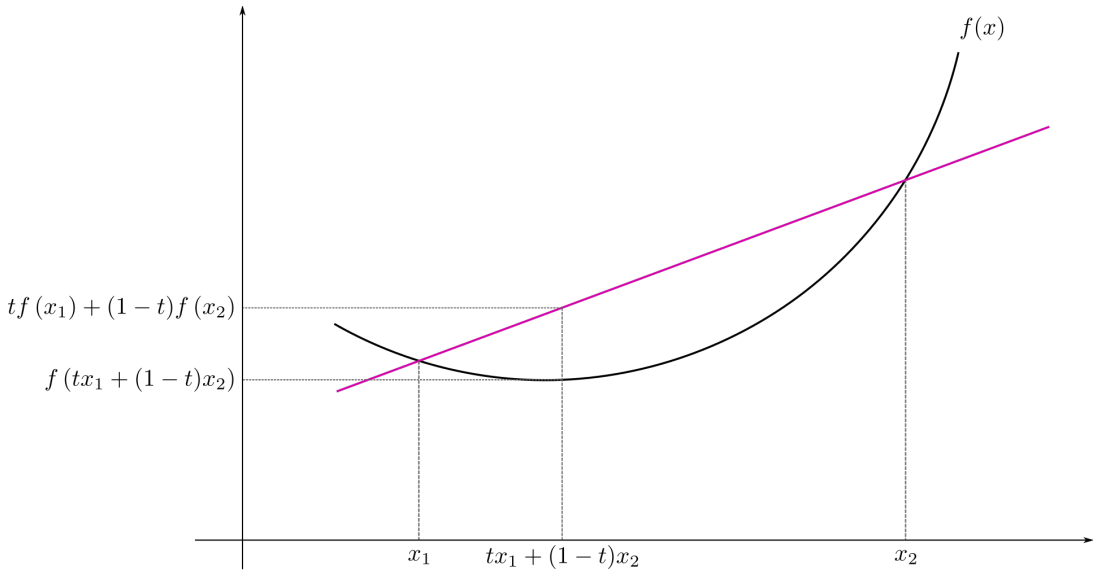

Figura 1: Función convexa

Así mismo, conviene saber que varias operaciones preservan la convexidad:

1. f convexa y $a \in \mathbb{R}^+$, $b \in \mathbb{R} \implies af + b$ convexa
2. f y g convexas $\implies f + g$ convexa
3. f y g convexas $\implies$ máx$\{f, g\}$ convexa
4. f y g convexas y g no decreciente $\implies g \circ f$ convexa (es decir, $h(x) = g(f(x))$ convexa)

Búsqueda ternaria

Una propiedad importante de las funciones convexas es que son unimodales. Decimos que f es *unimodal* si

- primero, **decrece estrictamente**,
- luego, **se mantiene constante** en su valor mínimo (puede ser a lo largo de un intervalo o únicamente en un punto),
- finalmente, **crece estrictamente**.[2]

[2] A veces veréis la misma definición pero pidiendo que f crezca primero y luego decrezca. Esto equivale a considerar que $-f$ es unimodal.

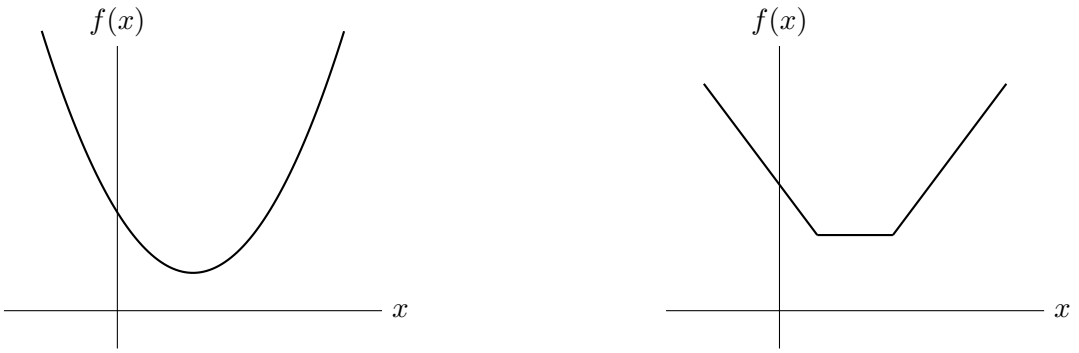

Figura 2: Ejemplos de funciones unimodales

La gracia de las funciones unimodales es que podemos encontrar su mínimo eficientemente con una *búsqueda ternaria*. Dado un interval $I = [x_l, x_r]$, encontramos el mínimo de f en I con el procedimiento siguiente:

Algorithm 1 Búsqueda ternaria

Require: Función unimodal f, intervalo $[x_l, x_r]$, tolerancia ε

while $x_r - x_l > \varepsilon$ **do**
  $d \leftarrow x_r - x_l$
  $m_1 \leftarrow x_l + d/3$
  $m_2 \leftarrow x_l + 2d/3$
  if $f(m_1) > f(m_2)$ **then**
    $x_l \leftarrow m_1$
  else
    $x_r \leftarrow m_2$
  end if
end while
return $(x_l + x_r)/2$

¿Por qué funciona el algoritmo anterior? La idea es que si tenemos tres puntos $x_1 < x_2 < x_3$ tales que $f(x_1) \leq f(x_2) \leq f(x_3)$, entonces sabemos[3] que el mínimo de f en el intervalo $[x_1, x_3]$ se encontrará en el subintervalo $[x_1, x_2]$. Similarmente, si tenemos tres puntos $x_1 < x_2 < x_3$ tales que $f(x_1) \geq f(x_2) \geq f(x_3)$, entonces el mínimo de f en el intervalo $[x_1, x_3]$ se encontrará en el subintervalo $[x_2, x_3]$.

El hecho de evaluar la función en dos puntos interiores m_1 y m_2, nos garantiza que encontraremos uno de estos dos escenarios, y podremos descartar el subintervalo correspondiente. Al coger los puntos equiespaciados, en cada paso reducimos la longitud del intervalo a considerar por un factor $2/3$, así que en $\log_{3/2}(L/\varepsilon)$ pasos habremos pasado de un intervalo de longitud L a un intervalo de longitud ε. Al acabar, es recomendable devolver el valor medio del intervalo que nos quede, para minimizar posibles errores de precisión.

Una cuestión importante es como escoger un valor de ε adecuado. Necesitamos que ε sea suficientemente pequeño para que $|f(x + \varepsilon) - f(x)| < T$, donde T es la tolerancia que nos piden en el problema. Observad que, para ε pequeño, $f(x+\varepsilon) - f(x) \approx \varepsilon f'(x)$, así que podemos tomar $\varepsilon := T/M$, donde M es una cota superior de f'.

[3]Si no lo habéis visto nunca antes, recomiendo que hagáis un dibujo y os convenzáis de esto.

Aplicación al problema

Nos falta únicamente comprovar que la función que nos piden minimizar en el problema es unimodal. En general, aunque la búsqueda ternaria únicamente necesita que la función sea unimodal, es más fácil demostrar que es convexa (una propiedad más fuerte), ya que tenemos mejores herramientas para ello (i.e. la caracterización en función de la segunda derivada y el hecho que se preserva por varias operaciones).

Consideremos que las dos personas empiezan a moverse en el tiempo $t = 0$ y llegan a su destinación en el tiempo $t = 1$. Entonces, la posición de la primera persona en el instante t será $p_1(t) = A + t(B - A)$, mientras que la posición de la segunda persona será $p_2(t) = C + t(D - C)$. Así pues, su distancia será

$$f(t) := \text{dist}(p_1(t), p_2(t)) = \big\|A + t(B - A) - (C + t(D - C))\big\| = \big\|v + tw\big\|,$$

donde $v := A - C$ y $w := B - A + C - D$.

Para ver que es convexa, tenemos que demostrar que $f(\lambda t + (1 - \lambda)s) \leq \lambda f(t) + (1 - \lambda)f(s)$, donde $\lambda \in [0, 1]$ y $s, t \in [0, 1]$ son valores cualesquiera. Esto sale de combinar la desigualdad triangular con las propiedades de las normas de vectores:

$$\begin{aligned} f(\lambda t + (1 - \lambda)s) &= \big\|\lambda(v + tw) + (1 - \lambda)(v + sw)\big\| \\ &\leq \big\|\lambda(v + tw)\big\| + \big\|(1 - \lambda)(v + sw)\big\| \\ &= \lambda\big\|v + tw\| + (1 - \lambda)\|v + sw\| \\ &= \lambda f(t) + (1 - \lambda)f(s) \end{aligned}$$

Implementación

```cpp
#include<bits/stdc++.h>
using namespace std;

using ld = long double;
ld const EPS = 1e-11;

struct Point {
  ld x, y;

  Point operator+(Point const& other) const {
    return Point{x + other.x, y + other.y};
  }

  Point operator*(double t) const {
    return Point{t*x, t*y};
  }
};

 istream& operator>>(istream& in, Point& p) {
  int x0, y0;
  in >> x0 >> y0;
  p.x = x0;
  p.y = y0;
  return in;
}

ld Dist(Point const& a, Point const& b) {
  return sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
}

int main(){
  cin.tie(NULL);
```

```
33    ios_base::sync_with_stdio(false);
34    cout.setf(ios::fixed);
35    cout.precision(10);
36
37    int t;
38    cin >> t;
39    for(int caso = 1; caso <= t; ++caso) {
40      Point a, b, c, d;
41      cin >> a >> b >> c >> d;
42
43      function<ld(ld)> Distancia = [&](ld t) {
44        return Dist(a*t + b*(1-t), c*t + d*(1-t));
45      };
46
47      ld t_left = 0;
48      ld t_right = 1;
49      while(t_right - t_left > EPS) {
50        ld t_mid_1 = (2*t_left + t_right) / 3;
51        ld t_mid_2 = (t_left + 2*t_right) / 3;
52        if(Distancia(t_mid_1) > Distancia(t_mid_2))
53          t_left = t_mid_1;
54        else
55          t_right = t_mid_2;
56      }
57      cout << "Case " << caso << ": " << Distancia((t_left + t_right) / 2) << '\n
          ';
58    }
59  }
```

Problema B: Volatile Kite

`https://codeforces.com/problemset/problem/772/B`

Resumen del enunciado

Dados los vértices de un polígono convexo, tenemos que encontrar el máximo valor de d tal que, si permitimos mover cada vértice a distancia d, el polígono continuará siendo convexo.

Solución

En el ejercicio anterior hemos trabajado con *funciones convexas*. Ahora, en cambio, hablamos de *polígonos convexos*. Decimos que un polígono P (o un conjunto de puntos en general) es *convexo* si, para todo par de puntos $x, y \in P$, se cumple que $\overline{xy} \subseteq P$ (el segmento que los une está totalmente contenido en el polígono P). [4]

En general, conviene pensar en los polígonos convexos como si fueran un n-ágono regular (donde n es el número de vértices) en el cual podemos desplazar los vértices pero no lo suficiente como para provocar una "hendidura".

Observad que si ordenamos los vértices de un polígono convexo en orden antihorario $(P_1, \ldots, P_n)$, entonces tendremos que, para cada trío de vértices consecutivos, el giro del vector $v_i := P_{i+1} - P_i$ al vector $v_{i+1} := P_{i+2} - P_{i+1}$ es antihorario. En otras palabras, el producto vectorial $v_i \times v_{i+1}$ (entendiendo ambos vectores como vectores tridimensionales con tercera coordenada cero) nos da un vector con tercera coordenada positiva.

Esto implica que, para hacer que el polígono deje de ser convexo, hay que conseguir que un vértice P_{i+1} pase al otro lado de la recta $\overline{P_i P_{i+2}}$ (para que el producto vectorial cambie de

[4] Observad que una función f es convexa si el conjunto de puntos que quedan por encima de su gráfica es convexo. Esta es la razón por la que se utiliza el mismo nombre para las dos propiedades.

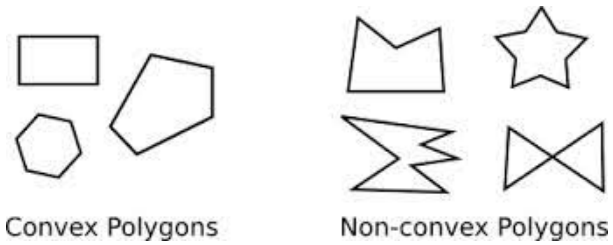

Figura 3: Ejemplos de polígonos convexos y no convexos

orientación). Así pues, el valor de d que queremos será la mitad de la mínima distancia de P_{i+1} a la recta $\overline{P_iP_{i+2}}$, donde minimizamos sobre todo $i \in \{1, \ldots, n\}$.

Entonces, hemos reducido el problema a calcular distancias entre rectas y puntos. Para ello, conviene representar las rectas como una `struct` (o `class`) con tres valores A, B y C, que representan la recta $Ax + By + C = 0$. De este modo, la distancia entre un punto p y la recta $r : Ax + By + C = 0$ vendrá dada por

$$\text{dist}(p, r) = \frac{|Ap_x + Bp_y + C|}{\sqrt{A^2 + B^2}}$$

Para encontrar los coeficientes A, B y C a partir de dos puntos P y Q cualesquiera de la recta, utilizamos el procedimiento siguiente. Observad que, si P y Q pertenecen ambos a la recta, entonces se satisfacen las ecuaciones

$$\begin{cases} Ap_x + Bp_y + C = 0 \\ Aq_x + Bq_y + C = 0 \end{cases},$$

así que $A(q_x - p_x) + B(q_y - p_y) = 0$.[5] Esta ecuación se cumplirá si, por ejemplo, tomamos como valores de A y B el coeficiente que acompaña al otro (cambiando uno de los dos de signo):

$$\begin{aligned} A &:= -(q_y - p_y) \\ B &:= q_x - p_x \end{aligned}$$

Solo nos falta escoger C de manera que se cumplan

$$\begin{cases} Ap_x + Bp_y + C = 0 \\ Aq_x + Bq_y + C = 0 \end{cases}$$

De la primera ecuación obtenemos que $C = p_xq_y - p_yq_x$. Afortunadamente, observamos que este valor también satisface la segunda ecuación, así que hemos encontrado unos coeficientes A, B y C tales que la ecuación $Ax + By + C = 0$ se satisface para los dos puntos P y Q. Dado que una recta queda completamente determinada especificando únicamente dos puntos, sabemos automáticamente que los demás puntos de la recta también satisfacerán la misma ecuación.

Implementación

```
#include<bits/stdc++.h>
using namespace std;

using ld = long double;

struct Point {
  ld x, y;
```

[5] En otras palabras, (A, B) es perpendicular al vector director de la recta $Q - P$.

```
};

struct Line {
  ld A, B, C; // Recta dada por Ax + By + C = 0
  Line(Point const& p, Point const& q) {
    A = p.y - q.y;
    B = q.x - p.x;
    C = p.x*q.y - p.y*q.x;
  }
};


ld Dist(Point const& p, Line const& r) {
  ld norma = sqrt(r.A*r.A + r.B*r.B);
  return abs(r.A*p.x + r.B*p.y + r.C) / norma;
}

int main() {
  cin.tie(NULL);
  ios_base::sync_with_stdio(false);

  int n;
  while(cin >> n) {
    vector<Point> v(n);
    for(Point& p : v)
      cin >> p.x >> p.y;
    ld const INF = 6e9;
    ld ans = INF;
    for(int i = 0; i < n; ++i) {
      Line r{v[i], v[(i+2)%n]};
      ans = min(ans, Dist(v[(i+1)%n], r)/2);
    }
    cout.setf(ios::fixed);
    cout.precision(10);
    cout << ans << endl;
  }
}
```

Problema C: Point in Polygon

`https://cses.fi/problemset/task/2192`

Resumen del enunciado

Dados los vértices de un polígono simple y una lista de puntos, debemos determinar si cada punto de la lista está dentro, fuera, o en el borde del polígono.

Solución

Un *polígono simple* es un polígono que no se interseca consigo mismo.[6] Por ejemplo, los polígonos verde y azul de la Figura 4 son polígonos simples, mientras que el polígono rojo no lo es.

[6]Técnicamente, los lados consecutivos de un polígono siempre se intersecan (en el vértice que los une), así que requerimos que no se interseque ninguna pareja de lados no consecutivos.

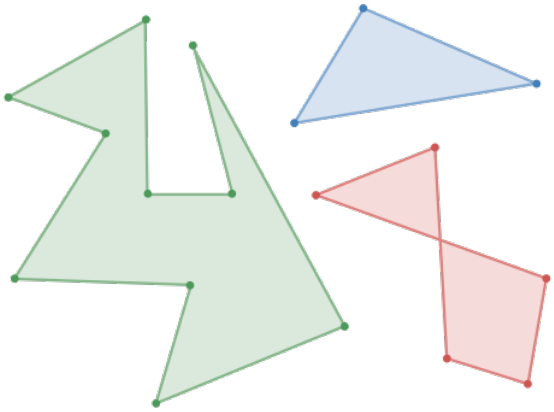

Figura 4: Ejemplos de polígonos simples y no simples

Puntos en el borde

Vamos a ver primero como determinar si un punto se encuentra en el borde del polígono. Dado un polígono simple con vértices $P_1, \ldots, P_n$ y un punto Q cualquiera, comprovaremos para cada $i \in \{1, \ldots, n\}$ si el punto Q se encuentra contenido en el segmento $\overline{P_i P_{i+1}}$.

Podéis comprobar que un punto C se encuentra contenido en el segmento $\overline{AB}$ si, y solo si, se cumplen las tres condiciones siguientes:

- El punto C se encuentra en la recta definida por A y B.
- $\text{mín}\{a_x, b_x\} \leq c_x \leq \text{máx}\{a_x, b_x\}$.
- $\text{máx}\{a_y, b_y\} \leq c_y \leq \text{máx}\{a_y, b_y\}$.

Comprobar la segunda y la tercera condición es trivial, mientras que la primera condición se puede verificar calculando el producto vectorial $(B - A) \times (C - A)$ y verificando que su tercera coordenada es cero.

En caso que no estéis familiarizados con el producto vectorial, esta es una operación que toma dos vectores tridimensionales y devuelve un tercer vector tridimensional que es perpendicular a ambos. Se puede calcular como el determinante de la siguiente matriz:

$$v \times w := \left| \begin{pmatrix} \hat{x} & \hat{y} & \hat{z} \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{pmatrix} \right| = \begin{pmatrix} v_y w_z - v_z w_y \\ v_z w_x - v_x w_z \\ v_x w_y - v_y w_x \end{pmatrix},$$

donde $\hat{x}$, $\hat{y}$, $\hat{z}$ es notación para los vectores unitarios en la dirección de los ejes x, y y z.

Lo importante a saber sobre el producto vectorial para este problema es que su tercera coordenada, $v_x w_y - v_y w_x$, es cero únicamente si v y w son vectores paralelos.

Puntos interiores/exteriores

Una vez sabemos que el punto Q no está en el borde del polígono, tenemos que determinar si está en el interior o en el exterior del polígono. Para ello, utilizaremos la observación siguiente: *Q está en el interior del polígono $\iff$ cualquier semirecta desde Q cruza el borde del polígono un número impar de veces.*

Podemos observar un ejemplo de ello en el polígono de la Figura 5: la semirecta horizontal desde el punto A cruza el borde del polígono 1 vez (un número impar), así que A se encuentra en el interior del polígono. En cambio, la semirecta desde B cruza el borde del polígono 4 veces (un número par), así que B se encuentra en el exterior del polígono.

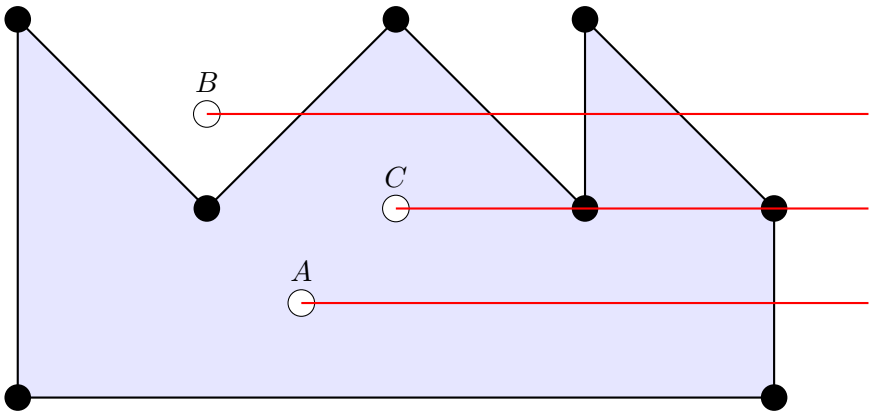

Figura 5

Dado que la caracterización anterior es válida para cualquier semirecta, en la implementación hemos tomado las semirectas horizontales en la dirección positiva del eje x. Para comprobar cuántos lados del polígono interseca, iteramos por todos los lados del polígono y, para cada uno de ellos, calculamos el producto vectorial entre $P - B$ y $B - A$ para ver en qué lado de la recta dada por los puntos A y B cae el punto P.

El caso del punto C es un poco particular, pues la semirecta pasa justamente por un vértice donde se juntan dos aristas que vienen del mismo lado de la semirecta. Estos casos límite complican el código (y es el motivo por el que en una rama de la implementación tenemos la desigualdad no estricta `b.y >= p.y` mientras que en la otra `b.y < p.y`). Un truco práctico para evitar estos problemas es escoger al inicio un ángulo aleatorio y rotar todos los vértices del polígono (y los puntos de cada query) por ese ángulo. Así nos aseguramos que, con alta probabilidad, no habrá dos puntos con la misma coordenada y.

Implementación

```
#include<bits/stdc++.h>
using namespace std;

using ll = long long;

struct Point {
  int x, y;
  Point operator-(Point const& other) const {
    return {x - other.x, y - other.y};
  }
};

// Devuelve el componente perpendicular al plano del producto
// vectorial (q-p) x (r-q).
ll Cross(Point const& p, Point const& q, Point const& r) {
  return ll((q-p).x) * (r-q).y - ll((q-p).y) * (r-q).x;
}

// Devuelve true si el punto p se encuentra en el segmento qr (extremos
    incluidos).
bool OnSegment(Point const& p, Point const& q, Point const& r) {
  return Cross(p, q, r) == 0 and min(q.x, r.x) <= p.x and max(q.x, r.x) >= p.x
                             and min(q.y, r.y) <= p.y and max(q.y, r.y) >= p.y;
}

int main() {
  cin.tie(NULL);
```

```
27    ios_base::sync_with_stdio(false);
28    int n, m;
29    while(cin >> n >> m) {
30      vector<Point> v(n);
31      for(Point& p : v)
32        cin >> p.x >> p.y;
33      while(m--) {
34        Point p;
35        cin >> p.x >> p.y;
36        bool boundary = false;
37        int num_cruces = 0;
38        for(int i = 0; i < n; ++i) {
39          Point a = v[i];
40          Point b = v[(i+1)%n];
41          if(OnSegment(p, a, b))
42            boundary = true;
43          if(a.y < p.y) {
44            if(b.y >= p.y and Cross(a, b, p) > 0)
45              ++num_cruces;
46          }
47          else {
48            if(b.y < p.y and Cross(a, b, p) < 0)
49              ++num_cruces;
50          }
51        }
52        cout << (boundary? "BOUNDARY" : (num_cruces&1 ? "INSIDE" : "OUTSIDE")) <<
              '\n';
53      }
54    }
55  }
```

Problema D: Pandemic Restrictions

`https://codeforces.com/gym/104017/problem/H`

Resumen del enunciado

Gerard tiene tres amigos: Aitor, Blanca y Carlos, con los que quiere organizar tres tipos de encuentros (unos con Aitor, Blanca y Gerard; otros con Blanca, Carlos y Gerard; y otros con Aitor, Carlos y Gerard). El coste de cada encuentro es la suma de las distancias que cada asistente tiene que recorrer hasta el punto de encuentro (que será el punto del plano que minimiza dicha suma de distancias). Sabiendo que Aitor, Blanca y Carlos viven en los puntos A, B y C del plano, ¿en qué punto tiene que construirse su casa Gerard para minimizar el máximo coste de entre los tres encuentros?

Solución

El punto de Fermat

Supongamos que nos dan la localización de la casa de Gerard[7]. Entonces, para calcular el coste de cada una de las tres reuniones, tenemos que encontrar el punto de encuentro que minimiza la suma de distancias recorridas por cada asistente. Este punto, conocido como el *punto de Fermat* (véase `https://en.wikipedia.org/wiki/Fermat_point`) se puede encontrar con argumentos geométricos, dando lugar a una fórmula para la suma de distancias óptima.

[7]Leed el resumen anterior del enunciado, pues he cambiado los nombres respecto al enunciado del problema para que sea más claro.

Ahora bien, esta solución requiere ciertos conocimientos previos de geometría y no es la opción más práctica para resolver el problema en un concurso. Lo más práctico consiste en darse cuenta que, igual que muchas funciones que implican minimizar distancias, la función objetivo es una función convexa.

En efecto, si queremos calcular el punto de encuentro óptimo entre tres puntos A, B y C, tenemos que minimizar la función[8]

$$f(P) = \operatorname{dist}(A, P) + \operatorname{dist}(B, P) + \operatorname{dist}(C, P).$$

Con un argumento parecido al del problema A, podemos ver que la función $P \mapsto \operatorname{dist}(A, P)$ es convexa:

$$\begin{aligned}
\operatorname{dist}(A, tP + (1-t)Q) &= \big\|A - tP - (1-t)Q\big\| \\
&= \big\|t(A-P) + (1-t)(A-Q)\big\| \\
&\leq \big\|t(A-P)\big\| + \big\|(1-t)(A-Q)\big\| \\
&= t\big\|A-P\big\| + (1-t)\big\|A-Q\big\| \\
&= t\operatorname{dist}(A,P) + (1-t)\operatorname{dist}(A,Q)
\end{aligned}$$

Así pues, f, al ser una suma de funciones convexas, también será convexa. Esto implica que podemos encontrar el punto de encuentro óptimo con una búsqueda ternaria en dos dimensiones.

Búsqueda ternaria en dos dimensiones

En el problema A habíamos visto como encontrar el mínimo de una función unimodal de una variable con búsqueda ternaria. Si la función depende de dos variables, podemos usar el procedimiento siguiente.

Sea $f(x, y)$ la función a minimizar. Sea $g(y) := \min_x f(x, y)$ la función de y que nos da el mínimo de f restringido a puntos con esa coordenada y. La idea es realizar una búsqueda ternaria para minimizar $g(y)$ en el intervalo $[y_l, y_r]$, donde cada vez que tenemos que evaluar g en un punto y_0, realizamos una segunda búsqueda ternaria para minimizar la función $x \mapsto f(x, y_0)$.

Algorithm 2 Búsqueda ternaria en dos dimensiones

Require: Función $F(x, y)$, intervalos $[x_l, x_r]$, $[y_l, y_r]$, precisión ε
Ensure: Aproximación del mínimo de F en el rectángulo $[x_l, x_r] \times [y_l, y_r]$

```
function TernarySearch2D(F, x_l, x_r, y_l, y_r, ε)
    function g(y):
        return TernarySearch1D(x ↦ F(x, y), x_l, x_r, ε)
    end function

    return TernarySearch1D(g, y_l, y_r, ε)
end function
```

Si la función $f(x, y)$ es convexa (es decir, informalmente, tiene "forma de cuenco") el procedimiento anterior hallará su mínimo global. Esto se debe a que la función $g(y)$ es también convexa, así que la búsqueda ternaria en y encuentra su mínimo, que se corresponderá con el mínimo de f:

$$\min_{x,y} f(x, y) = \min_y \min_x f(x, y) = \min_y g(y).$$

[8] Cuando escribimos $f(P)$, queremos decir que $f \equiv f(x, y)$ es una función de dos variables, las coordenadas del punto P.

Así pues, la busqueda ternaria en dos dimensiones funciona para minimizar funciones convexas. **Atención:** lo mismo no ocurre para funciones que son únicamente unimodales (¡a diferencia del caso de una variable!). Si únicamente supiéramos que la función f es unimodal en cada coordenada (es decir, que $h_{y_0}(x) := f(x, y_0)$ es unimodal para todo $y_0 \in [y_l, y_r]$, y que $\tilde{h}_{x_0}(y) := f(x_0, y)$ es unimodal para todo $x_0 \in [x_l, x_r]$), entonces la función $g(y)$ puede no ser unimodal, así que el algoritmo anterior no tiene por qué hallar el mínimo de f.

Localización de la casa de Gerard

Con lo anterior ya tenemos la solución del problema suponiendo que conocemos la localización óptima de la casa de Gerard. ¿Y ésta como la encontramos? Hay quien dice que al hombre con un martillo, todo le parece un clavo. Pues bien, en los problemas de geometría, nuestro martillo es la búsqueda ternaria, así que vamos a demostrar que la localización óptima de la casa es el mínimo de una función convexa.

Dados tres puntos X, Y y Z, sea $f_{XYZ}(P)$ la función que nos da la suma de distancias a P, es decir,

$$f_{XYZ}(P) := \operatorname{dist}(X, P) + \operatorname{dist}(Y, P) + \operatorname{dist}(Z, P).$$

Esta es la función que hemos visto que es convexa en el apartado anterior. Ahora, definimos una nueva función f_{XY} como

$$f_{XY}(Z) := \min_P f_{XYZ}(P).$$

Entonces, para encontrar la casa de Gerard, debemos minimizar la función

$$f(G) := \max\left\{f_{AB}(G), f_{BC}(G), f_{AC}(G)\right\},$$

donde A, B y C son los puntos dados en el enunciado, y G se corresponde con el punto donde se ubica la casa de Gerard.

Observad que, para todo X e Y, la función $f_{XY}(Z)$ es una función convexa:

$$\begin{aligned}
f_{XY}(tZ + (1-t)\widetilde{Z}) &= \min_P \left\{ \operatorname{dist}(X, P) + \operatorname{dist}(Y, P) + \operatorname{dist}(tZ + (1-t)\widetilde{Z}, P)\right\} \\
&\leq \min_P \left\{ \operatorname{dist}(X, P) + \operatorname{dist}(Y, P) + t\operatorname{dist}(Z, P) + (1-t)\operatorname{dist}(\widetilde{Z}, P)\right\} \\
&= t \cdot \min_P \left\{ \operatorname{dist}(X, P) + \operatorname{dist}(Y, P) + \operatorname{dist}(Z, P)\right\} \\
&\quad + (1-t) \cdot \min_P \left\{ \operatorname{dist}(X, P) + \operatorname{dist}(Y, P) + \operatorname{dist}(\widetilde{Z}, P)\right\} \\
&= t f_{XY}(Z) + (1-t) f_{XY}(\widetilde{Z})
\end{aligned}$$

Aquí hemos utilizado que $Z \mapsto \operatorname{dist}(Z, P)$ es una función convexa, como hemos visto anteriormente.

En conclusión, $f(G)$ es una función convexa, ya que se puede escribir como el máximo de un número finito de funciones convexas.

Implementación

Dado que tenemos que realizar (¡)cuatro(!) búsquedas ternarias encadenadas, conviene definirse una función `TernarySearch()` que reciba como parámetro la función a optimizar y ejecute una búsqueda ternaria para encontrar su mínimo. Así mismo, definimos una función `TernarySearch_2D` que encadena dos búsquedas ternarias para encontrar el mínimo en un rectángulo.

```
1 #include<bits/stdc++.h>
2 using namespace std;
3
```

```cpp
using ld = long double;

ld const EPS = 1e-4;

struct Pt {
  ld x, y;

  friend istream& operator>>(istream& in, Pt& p) {
    in >> p.x >> p.y;
    return in;
  }
};

ld Dist(Pt const& a, Pt const& b) {
  ld dx = a.x - b.x;
  ld dy = a.y - b.y;
  return sqrt(dx*dx + dy*dy);
}

// Devuelve minimo de F en el intervalo [le, ri]
// Precondicion:  f es unimodal, le <= ri
ld TernarySearch(function<ld(ld)> F, ld le, ld ri) {
  while(ri - le > EPS) {
    ld mid_le = (2*le + ri)/3;
    ld mid_ri = (le + 2*ri)/3;
    if(F(mid_le) > F(mid_ri))
      le = mid_le;
    else
      ri = mid_ri;
  }
  return F((le + ri)/2);
}

// Devuelve minimo de F en el rectangulo [x_le, x_ri] x [y_le, y_ri]
ld TernarySearch_2D(function<ld(ld,ld)> F, ld x_le, ld x_ri, ld y_le, ld y_ri)
    {
  function<ld(ld)> BusquedaHorizontal = [&](ld y0) {
    function<ld(ld)> F_Horizontal = [&](ld x) { return F(x, y0); };
    return TernarySearch(F_Horizontal, x_le, x_ri);
  };
  return TernarySearch(BusquedaHorizontal, y_le, y_ri);
}

// Devuelve la minima suma de distancias entre los puntos p, q, r y un cuarto
    punto por determinar.
ld DistanciaDeEncuentro(Pt const& p, Pt const& q, Pt const& r) {
  function<ld(ld,ld)> SumaDeDistancias = [&](ld x, ld y) {
    Pt z{x, y};
    return Dist(p, z) + Dist(q, z) + Dist(r, z);
  };
  ld x_min = min(p.x, min(q.x, r.x));
  ld x_max = max(p.x, max(q.x, r.x));
  ld y_min = min(p.y, min(q.y, r.y));
  ld y_max = max(p.y, max(q.y, r.y));
  return TernarySearch_2D(SumaDeDistancias, x_min, x_max, y_min, y_max);
}

int main() {
  cin.tie(NULL);
  ios_base::sync_with_stdio(false);

  cout << fixed;
  cout.precision(10);
```

```
65    Pt a, b, c;
66    while(cin >> a >> b >> c) {
67      function<ld(ld,ld)> F = [&](ld x, ld y) {
68        Pt p{x, y};
69        return max(DistanciaDeEncuentro(a, b, p), max(DistanciaDeEncuentro(a, p,
              c), DistanciaDeEncuentro(p, b, c)));
70      };
71
72      ld x_min = min(a.x, min(b.x, c.x));
73      ld x_max = max(a.x, max(b.x, c.x));
74      ld y_min = min(a.y, min(b.y, c.y));
75      ld y_max = max(a.y, max(b.y, c.y));
76      cout << TernarySearch_2D(F, x_min, x_max, y_min, y_max) << '\n';
77    }
78  }
```

¡He aquí el poder de la convexidad! Hemos resuelto un problema geométrico muy difícil sin ninguna fórmula y sin tener que usar ningún conocimiento geométrico (más allá de la distancia entre dos puntos).

Problema E: No Smoking!

`https://acm.timus.ru/problem.aspx?space=1&num=1469`

Resumen del enunciado

Dados n segmentos en el plano, tenemos que encontrar dos que se intersequen (o decir que no hay ninguna pareja de segmentos que se interseque).

Solución

Este problema se puede resolver de manera trivial en $\mathcal{O}(n^2)$: iterando por cada pareja de segmentos y aplicando un procedimiento parecido al del problema C para ver si se intersecan. Sin embargo, existe una solución muy instructiva en $\mathcal{O}(n \log n)$, usando el método de la *sweep line* (línea de barrido).

La idea básica es imaginar que tenemos una recta vertical que "barre" el plano de izquierda a derecha (es decir, de menor a mayor coordenada x). A medida que hacemos esto, mantenemos una lista de los segmentos con los que estamos intersecando en ese momento, ordenados de menor a mayor coordenada y (de su punto de intersección con nuestra recta vertical). Cada vez que nos encontramos el punto de inicio de un segmento, lo añadimos a nuestra lista; y cada vez que encontramos el punto final, lo quitamos de la lista.

La observación clave es que un segmento solo se puede intersecar con segmentos que han sido vecinos suyos en la lista. Esto (junto con alguna observación adicional), nos indica que solo es necesario comprobar si se intersecan los pares de segmentos siguientes:

- Al añadir un segmento en la lista, el segmento añadido con sus dos segmentos vecinos.
- Al quitar un segmento de la lista, sus dos segmentos vecinos.

En `https://cp-algorithms.com/geometry/intersecting_segments.html` podéis encontrar más detalles sobre el porqué de la corrección del algoritmo anterior.

Conviene saber que la técnica de la *sweep line* no es solamente útil para problemas geométricos, sino que también puede aparecer en problemas de programación dinámica y de estructuras de datos (por ejemplo, véase `https://usaco.guide/plat/range-sweep?lang=cpp`).

Implementación

En `https://cp-algorithms.com/geometry/intersecting_segments.html` podéis encontrar una implementación del problema. He intentado simplificar el código y hacerlo más entendible (siendo más verboso en algunos puntos y cambiando un poco la lógica). Pese a ello, continúa siendo delicado de implementar, pues requiere mantener un set con los segmentos activos y un vector de iteradores que apuntan a las posiciones de cada segmento en este set.

Podéis echarle un vistazo, pero os recomiendo que intentéis implementarlo por vuestra cuenta, pues contiene algunas herramientas con las que conviene estar familiarizado (por ejemplo: la función `std::set::lower_bound` (`https://cplusplus.com/reference/set/set/lower_bound/`). Así mismo, conviene entender cómo funcionan `std::next` y `std::prev` en los casos extremos (es decir, cuando se ha llegado al final (o inicio) del contenedor).

```
#include<bits/stdc++.h>
using namespace std;

using ll = long long;
using ld = long double;
ld const EPS = 1e-8;

struct Point {
  int x, y;
};

struct Segment {
  Point a, b;
  int id;

  ld Get_y(int x) const {
    if(a.x == b.x)
      return a.y;
    return a.y + (b.y - a.y) * ld(x - a.x) / (b.x - a.x);
  }

  // Ordena segun altura del primer punto en que se encuentran.
  bool operator<(Segment const& other) const {
    int x = max(min(a.x, b.x), min(other.a.x, other.b.x));
    return Get_y(x) < other.Get_y(x) - EPS;
  }
};

int SignCross(Point const& p, Point const& q, Point const& r) {
  int ans = (q.x - p.x)*(r.y - q.y) - (q.y - p.y)*(r.x - q.x);
  return (ans > 0 ? 1 : (ans < 0 ? -1 : 0));
}

bool Intersect(Segment const& s, Segment const& t) {
  return SignCross(s.a, s.b, t.a) * SignCross(s.a, s.b, t.b) <= 0 and
        SignCross(t.a, t.b, s.a) * SignCross(t.a, t.b, s.b) <= 0 and
        max(s.a.x, s.b.x) >= min(t.a.x, t.b.x) and
        max(t.a.x, t.b.x) >= min(s.a.x, s.b.x) and
        max(s.a.y, s.b.y) >= min(t.a.y, t.b.y) and
        max(t.a.y, t.b.y) >= min(s.a.y, s.b.y);
}

struct Event {
  int x, type, segment_id;
};

int main() {
  int n;
  while(cin >> n) {
```

```
50      vector<Segment> v(n);
51      vector<Event> events;
52      for(int i = 0; i < n; ++i) {
53        cin >> v[i].a.x >> v[i].a.y >> v[i].b.x >> v[i].b.y;
54        v[i].id = i;
55        int xmin = min(v[i].a.x, v[i].b.x);
56        int xmax = max(v[i].a.x, v[i].b.x);
57        events.push_back({xmin, 1, v[i].id});
58        events.push_back({xmax, -1, v[i].id});
59      }
60      sort(events.begin(), events.end(), [&](Event const& lhs, Event const& rhs)
            {
61        if(lhs.x == rhs.x)
62          return lhs.type > rhs.type;
63        return lhs.x < rhs.x;
64      });
65
66      set<Segment> active;
67      vector<set<Segment>::iterator> pointer(n);
68      pair<int,int> intersecting = {-1, -1};
69      for(Event const& event : events) {
70        int id = event.segment_id;
71        if(event.type == 1) {
72          auto it = active.lower_bound(v[id]);
73          // Intersecamos con el siguiente:
74          if(it != active.end() and Intersect(v[id], *it)) {
75            intersecting = {id, it->id};
76            break;
77          }
78          // Intersecamos con el anterior:
79          if(not active.empty() and it != active.begin() and Intersect(v[id], *
                prev(it))) {
80            intersecting = {id, prev(it)->id};
81            break;
82          }
83          // Insertamos el nuevo segmento:
84          pointer[id] = active.insert(it, v[id]);
85        }
86        else {
87          auto it = pointer[id];
88          if(it != active.begin() and next(it) != active.end() and Intersect(*
                prev(it), *next(it))) {
89            intersecting = {prev(it)->id, next(it)->id};
90            break;
91          }
92          active.erase(it);
93        }
94      }
95      if(intersecting.first == -1)
96        cout << "NO\n";
97      else {
98        cout << "YES\n" << intersecting.first + 1 << " " << intersecting.second +
              1 << "\n";
99      }
100   }
101 }
```