DAA-C.Electric Charges
Problem 623/C

Abel Molina Sánchez C411

Julio 2022

1 Descripción del problema

Descripción completa: `https://codeforces.com/problemset/problem/623/C`

Problema Se tienen n puntos con coordenadas enteras (x_i, y_i). Cada punto se proyectará en uno de los ejes $X(x_i, 0)$ o $Y(y_i, 0)$. Se quiere conocer cual es el cuadrado del menor diámetro posible entre los puntos después de esta operación.

Entrada:

Línea 1: n - $(1 \leq n \leq 100000)$: número de puntos

Siguientes n líneas: en cada uno dos enteros x_i y_i las coordenadas del i-ésimo punto. No hay dos puntos coincidentes.

Salida:

Línea 1: un entero, el cuadrado del mínimo diámetro posible entre los puntos.

2 Análisis del problema

Notación: Denotaremos $D_2(p_1, p_2)$ al cuadrado del diámetro entre dos puntos p_1 y p_2.

Cota de la solución: Tenemos que cada punto será proyectado en alguno de los ejes, con los cual, hay dos distribuciones de puntos que siempres son posibles: 1- colocarlos todos en el eje $X(x_i, 0)$, o colocarlos todos en el eje $Y(0, y_i)$. Con lo cual si tomamos:
$x\_min = min\ x_i \forall i$
$x\_max = max\ x_i \forall i$
$y\_min = min\ y_i \forall i$
$y\_max = max\ y_i \forall i$
tenemos que los siguientes D_2 serán siempre posibles de alcanzar:
$D_2y = (y\_max - y\_min)^2$
$D_2x = (x\_max - x\_min)^2$

Con lo cual si hacemos $MIND_2 = min(D_2y, D_2x)$, tenemos que cualquier respuesta óptima estará en el intervalo $[0, MIND_2]$.

Distancia entre los ejes: A la hora de calcular los diámetros habrá dos casos, el diámtro entre elementos del mismo eje $(X - X,\ Y - Y)$, o entre ejes distintos$(X - Y)$. En el caso de un mismo eje, como se anula una coordenada, de la fórmula de distancia nos queda: $D_2 = (p_1.c - p_2.c)^2$. En el caso de cruce de ejes, tendremos $D_2 = (p_1.c1)^2 + (p_2.c2)^2$, donde en este último es válido señalar que es la suma de las distancias respectivas al $(0, 0)$(Más adelante se menciona este elemento)

Relación de los diámetros: En una distribución cualquiera de los puntos luego de proyectarlos en sus respectivos ejes, vamos a tener los siguientes diámetros que definen el diámetro del conjunto:
Con los puntos definidos en el apartado anterior:
$D1 = D_2(max_x, min_x) = d(max_x, min_x)^2$
$D2 = D_2(max_y, min_y) = d(max_y, min_y)^2$
$D3 = D_2(max_x, min_y) = d(max_x, min_y)^2$
$D4 = D_2(max_x, max_y) = d(max_x, max_y)^2$
$D5 = D_2(min_x, min_y) = d(min_x, min_y)^2$

$D6 = D_2(min_x, max_y) = d(min_x, max_y)^2$

Tenemos que el diámetro del conjunto va a ser el $max(di)$, ya que: supongamos que existe un punto que no es de los mencionados que forma parte de un diámetro más grande. Tenemos que ese punto tiene que estar proyectado en alguno de los ejes, sin pérdida de generalidad digamos que lo está en el eje X. Luego, es claro que este punto no puede tener una distancia mayor con ningún punto del eje X que la que tienen x_min, x_max, precisamente porque de tenerla sería una contradicción con el hecho de que x_min y x_max son extremos de este conjunto. Entonces, queda el caso de que forme un diámetro con un punto del eje Y tal que este diámetro sea mayor que los mencionados. Pero ningún punto del eje Y va a estar a más distancia del eje X que el máximo entre y_min, y_max, porque nuevamente sería una contradicción con la extremalidad de dichos puntos en el eje Y. Luego, dicho diámtero tendría que ser entre el punto y el máximo(absoluto) de y_max y y_min, pero este diámetro es menor que los diámetros que forman el mayor extremo de X con dicha y, porque de lo contrario, $|punto| > |max_x|$ lo cual es una contradicción. Con lo cual, el mayor diámetro del conjunto está dado por los diámetros mencionados anteriormente.

3 Soluciones

Fuerza bruta: La primera solución de fuerza bruta pasa por explorar el universo de soluciones al completo. Para esto es necesario generar todas las posibles distribuciones de los puntos en los ejes. Por cada punto se tienen dos opciones, eje X o eje Y, como tenemos n puntos, por el principio de la multiplicación sabemos que en total serán 2^n distribuciones que a su vez, es el número de cadenas binarias de tamaño n. Precisamente, en el algoritmo de *fuerza bruta* lo que se hace es generar todas las cadenas binarias y luego, a partir de cada una, computar la distancia máxima en el conjunto resultante de proyectar los puntos $0's$ en Y y los puntos $1's$ en X. Como se tienen 2^n cadenas y por cada una se recorre el conjunto de n puntos para computar el diámetro máximo, entonces, el algoritmo de *fuerza bruta* es $O(n * 2^n)$.
Código en: brute_force.py

Ahora, para empezar a mejorar la solución, vamos a demostrar el siguiente teorema sobre la respuesta óptima al problema.

Teorema-Estructura óptima: *Existe una distribución de puntos que garantiza el óptimo, tal que, si se ordenan por las x, tiene la forma $S_yS_xS'_y$, siendo S_y, S_x, S'_y, subconjuntos contiguos de puntos.*
Demostración: Sea Op una distribución de puntos que garantiza el óptimo. Si Op no tiene puntos en X, hacemos $S_x = \emptyset$ y dividimos los puntos Y entre S_y y S'_y, eventualmente vacíos y tenemos la estructura buscada.
Si Op tiene un solo punto en las X, hacemos $S_x = \{x_i\}$, S_y igual a los puntos menores que x_i y S_y igual a los mayores que x_i y tenemos la estructura buscada.
Ahora, si hay al menos dos puntos que se colocan en el eje X, seleccionemos Xl y Xr los de mayor diámetro entre si. El resto de puntos proyectados en el eje X está contenido en el intervalo $[Xl, Xr]$ porque: supongamos un punto proyectado en las X, x_{out}, que pertenece a Op y no está contenido en el intervalo anterior. Tenemos dos casos: $x_{out} < Xl$ o $x_{out} > Xr$. Si $x_{out} < Xl$, tenemos que $d(x_{out}, Xr) = d(x_{out}, Xl) + d(x_{out}, Xr) > d(Xl, Xr)$ lo cual contradice que el intervalo $[Xl, Xr]$ fuera el de mayor diámetro en Op. El otro caso es análogo.
Tenemos que todos los puntos X están contenidos en el intervalo $Ix = [Xl, Xr]$. Si no hay elementos Y en Ix entonces decimos, $S_x = Ix$, S_y las y menores que Xl(por las x) y S'_y las y mayores que Xr. Ahora supongamos que en Op hay algún punto Y, digamos y_i tal que y_i está en Ix. Como tenemos $[Xl, Xr]$ dentro del Op, sabemos que $d(Op) \geq d(Xl, Xr)$. Sabemos, por lo que se plantea en *Relación de diámetros* cuáles diámetros definen el diámetro de Op, a saber: $d(max_x, min_x) = d(Xl, Xr)$, $d(max_y, min_y)$, $d(max_x, min_y)$, $d(min_x, min_y)$, $d(max_x, max_y)$ y $d(min_x, max_y)$. Tenemos entonces y_i contenido en el intervalo $[Xl, Xr]$, construyamos Op' tal que el punto y_i se proyecta en las X. Comprobemos que Op' sigue siendo óptimo. Proyectar y_i en X, no afectará el tamaño de $d(Xl, Xr)$ porque está contenido entre ambos extremos. Directamente tenemos que $d(Op) = d(Op')$ si $y_i \neq y\_max$ y $y_i \neq y\_min$ porque y_i no estaría involucrado en ninguno de los diámetros que definen en el máximo. Vemos entonces, que si en otro caso $d(Op) \geq d(Op')$ porque eventualmente hemos disminuido uno de los extremos que definen los diámetros mayores, pero como Op es óptimo, no puede pasar que $d(Op) > d(Op')$ porque sería una contradicción.
Entonces, proyectar y_i en X en cualquier caso mantiene $d(Op) = d(Op')$. Luego, tenemos que todo punto contenido en el intervalo $[Xl, Xr]$ puede ser proyectado en el eje X sin afectar el óptimo, y, por tanto, siguiendo una secuencia finita de la operación hecha con y_i anteriormente, podemos convertir Op en un Op^*, también óptimo, tal que en Op^* el intervalo $[Xl, Xr]$ contiene solo puntos proyectados en el eje X y, todos los puntos proyectados en el eje X, están contenidos en $[Xl, Xr]$. Luego, hacemos $S_x = [Xl, Xr]$, S_y las y menores que Xl y S'_y las y mayores que Xr. De esta forma tenemos la estructura planteada, con lo cual queda demostrado que existe un óptimo con esta estructura, y que, de haber un óptimo que no cumple dicha estructura, con una secuencia fininta de pasos se puede reducir a este.

Segunda solución: Como sabemos que hay una distribución óptima que tiene la estructura descrita en el *Teorema-Estructura óptima*, entonces la idea de esta solución es encontrar el intervalo $[Xl, Xr]$, ya que, una vez determinado los límites de S_x, se pueden sabe que elementos pertencen a Sy y Sy'.

```
1-Ordenar los puntos por las x
2-Computar los arreglos de prefijos y sufijos máximos y mínimos de las y.
3-min_sd = min((max_x-min_x)**2,(max_y-min_y)**2)
4-Por cada punto i de 0..n, elegirlo como Xl y:
    4.1 por cada punto de i a n elegirlo como XR:
        4.1.1 computar los diámetros de la distribución resultante.
        4.1.2 elegir el máximo y compararlo con min_sd y si es menor actualizar
5-Devolver min_sd
```

Código en: n2_solution.py

La solución hace lo siguiente: ordenar los puntos por su coordenada x, y marcar cada uno de ellos como inicio posible del intervalo $[Xl, Xr]$, o sea, como Xl, y explorar los restante puntos a su derecha como posibles Xr. Por cada uno de los intervalos que se forman, calculamos los diámetros que describen el conjunto que se forma, y nos quedamos con el mayor de ellos que será el diámetro de esa distribución. Todo el tiempo se mantiene una variable *min_sd*, que guarda el menor diámetro de distribución que se ha visto. Como sabemos que existe un intervalo óptimo, este tiene que ser el que da lugar a la distribución de menor diámetro de entre todas los que siguen esta estructura. Como por cada intervalo es necesario calcular los diámetros, es necesario conocer los puntos extremos en el eje Y en esa distribución. Para poder obtener estos valores sin necesidad de iterar nuevamenente por el conjunto, luego de ordenar los puntos, se precomputan 4 arreglos con los posibles valores de las y para cada intervalo: *prefix_min, prefix_max, sufix_min, sufix_max* , que conitienen, dado un índice i, el valor mínimo, máximo, mínimo, máximo de las y en los intervalos, $[0, i]$, $[0, i]$, $[[i, n]$ e $[i, n]$ respectivamente. Precomputar cada uno de estos arreglos se hace en $O(n)$, iterando, para el caso de los prefijos, de 1 a $n-1$ y haciendo $p[i] = min/max(a[i].y, p[i-1])$, luego de setear en $p[0]$ a $a[0].y$. Para el caso de los sufijos, se hace de $n-2$ hasta 0 haciendo en cado paso $p[i] = min/max(a[i].y, p[i+1])$, luego de setear $p[n-1] = a[-1].y$. De esta forma, conocer max/min y, se hace en $O(1)$ preguntando en los prefijos max/min y sufijos max/min por los elementos $Xl-1$ y $Xr+1$ respectivamente, y luego quedándose con el max entre los max y el min entre los min obtenidos.

Complejidad temporal: Tenemos el conjunto de puntos de tamaño n. Ordenarlos por las x es $O(nlogn)$, luego, precomputar los arreglos de prefijos y sufijos max/min de las y se hace en una iteración $1..n$ con operaciones $O(1)$, con lo cual tenemos que este precómputo se hace en $O(n)$. La exploración del óptimo se hace en dos ciclos for anidados, uno que itera en i de $0...(n-1)$, y otro que lo hace desde $j = i...(n-1)$, con lo cual, tenemos que se hacen:

$\sum_{i=0}^{n-1} n - i = n + (n-1) + ... + 1 = n(n+1)/2 = (n^2+n)/2 = n^2/2 + n/2$
iteraciones con operaciones constantes $O(1)$. Con lo cual determinamos el óptimo en $O(n^2)$, y tenemos que el algoritmo al completo va a hacer $O(nlogn + n + n^2) = O(n^2)$.

Enfoque para la tercera solución: Como tenemos la solución acotada, o sea, tenemos el intervalo $D = [0, MAX]$, $MAX = min(D_2(min\_y, max\_y), D_2(min\_x, max\_x))$ donde está contenido el óptimo, se puede realizar una búsqueda binaria sobre mismo respondiendo al siguiente Predicado(D): El diámetro al cuadrado D puede contener al conjunto?, O lo que es lo mimso: Hay alguna distribución tal que el D_2 del conjunto es menor igual que D?. Es necesario probar dos cosas, que el predicado es de la forma "no","no",....,"no","si","si",...,"si" y que, en ese caso el primer "si" es exactamente el D_2 óptimo del conjunto.

Demostración del predicado: Sea d_2 un valor para el D_2, y sea Opd el óptimo del conjunto. Ningún $D_2 \leq Opd$ puede contener el conjunto, porque entonces sería una contradicción con que Opd sea el óptimo. A su vez, tenemos que para cualquier $d_2 \geq Opd$ es posible encontrar una distribución del conjunto tal que el d_2 lo contiene, en especial podemos encontrar la de Opd que verifica que el D_2 del conjunto, igual a Opd es menor igual que d_2. Con lo cual, el predicado es de la forma "no","no",....,"no","si","si",...,"si" sobre los valores para el D_2 del conjunto y, en especial, hemos demostrado que Opd es el primer "si", ya que es el menor valor que contiene al conjunto y, por tanto, el primero que lo hace.

Luego, la búsqueda binaria sobre el conjunto de posibles valores para el óptimo es posible. Ahora, es necesario poder evaluar el Predicado de forma tal que se mejore la solución anterior.

Para evaluar el predicado necesitamos encontrar una distribución tal que que su D_2 sea $\leq D$, siendo D el valor de la evaluación.

Las siguientes dos soluciones al problema siguen el mismo enfoque a la hora de evaluar el predicado, así que, vamos a ver la parte teórica y luego el detalle de la implementación de cada una. Ambas parten de haber ordenado previamente el conjunto por las X y de computar los arreglos de prefijos y sufijos max/min de las y, tal como se hizo en la solución anterior.

```
1-Ordenar los puntos por las x
2-Computar los arreglos de prefijos y sufijos máximos y mínimos de las y.
3-min_sd = min((max_x-min_x)**2,(max_y-min_y)**2)
4-Busqueda binaria sobre el intervalo [0,min_sd]. Actualizar min_sd
5-Devolver min_sd
```

Tenemos por el *Teorema-Estructura óptima*, que existen las distribuciones que siguen la estructura $S_y S_x S'_y$, y que entre ellas hay una, la de menor D_2 que es óptima para el problema. Entonces, y también como resultado de la solución anterior, tenemos que si construimos todas las distribuciones que siguen esta estructura, eventualmente tendremos una que es óptima y que por tanto su $D_2 = Opd$. Como no podemos iterar nuevamente de la misma forma que en la solución anterior, vamos a hacer lo siguiente: Sabemos que, en cualquier distribución de la forma descrita tendremos un intervalo $[Xl, Xr]$ y un $x\_min = min(|Xl|, |Xr|)$ y un $x\_max = max(|Xl|, |Xr|)$. Entonces, vamos a marcar la posicion de x_max y para ella vamos a buscar la mejor x_min posible que valide a D siendo D el valor por el que se pregunta en el Predicado. Esto siempre va a ser posible para alguna x_max, siempre y cuando, $D >= Opd$, ya que hay una x_max que genera el óptimo. Si $D < Opd$, no va a ser posible, porque entonces habría una distribución que genera un D_2 mejor que el óptimo, lo cual es una contradicción.
Vamos a analizar la relación x_min x_max para una x_max arbitraria:
Tenemos que 1) $|x\_max| \geq |x\_min|$ por definición y que para que valide a D, 2)$D_2(x\_max, x\_min) <= D$.
También el hecho de definir x_max, hace que el menor D_2 que se pueda generar en el conjunto esté definido por los máximo entre $D_2(x\_max, x\_min)$, $D_2(y\_max, y\_min)$ y $D_2(x\_max, y\_max)$, ya que la distancia de x_min a y siempre será menor que la de x_max, ya que tiene mayor valor absoluto y por tanto mayor distancia con el $(0, 0)$ que, como vimos al principio, está contenido en la distancia desde el eje X hasta cualquier punto proyectado en las y. Con lo cual, tenemos que el mejor x_min para x_max va a ser el mayor que cumple 1 y 2, ya que no afecta los D_2 de las distribuciones.
A la hora de fijar las x_max va a tener dos posibles escenarios, o bien $x\_max = Xl$ y, por tanto, < 0, o $x\_max = Xr$ y por tanto $>= 0$. Esto se debe a que, si estamos en el caso $Xl = max\_x$, si Xl fuera > 0, todos los puntos a su derecha cumplen que $|Xl| < |x_i|$ lo cual contradice que sea el max_x. Análogamente ocurre con el caso Xr. Por tanto siempre será necesario manejar ambos escenarios a la hora de utilizar los valores de x_max.

Solución con Búsqueda Binaria: Para validar el predicado para D, o sea, comprobar si $D >= D_2$ para algún D_2 posible en el conjunto lo que se hace es lo siguiente:
Se itera por el arreglo ordenados de puntos, y se va eligiendo cada uno como Xl mientras sea posible($points[l].x < 0$), luego tenemos que buscar a su x_min tal y como se describió antes. Para esto se hace una búsqueda binaria en todos los puntos a su derecha. Si no se encunetra una distribución que valida a D, entonces, análogamente, se hace lo mismo pero iterando desde el extremo derecho $n - 1$ a 0, para el caso que en que $Xr = max\_x$
Analicemos ahora cual será el predicado de esta nueva búsqueda binaria. Sabemos que los x_min para un x_max cumplen las condiciones 1 y 2 descritas anteriormente, entonces, podemos ver que si se tiene un punto xr que es válido para Xl, un punto xr' a la izquierda de xr también lo será, ya que, si $|Xl| >= |xr|$ se cumple y $D_2(Xl, xr) <= D$ también se tiene que se cumple, tenemos directo que $D_2(Xl, xr')$ se cumple(está a la izquierda de x_r), y, para ver que $|Xl| >= |xr'|$, supongamos que eso no es cierto, entonces tendríamos $|Xl| < |xr'|$ pero como $|Xl| < 0$ y xr está a la derecha de Xl, entonces tendría que ser $xr' > 0$ y por tanto, esto implicaría que $xr > 0$ porque x'_r está a la izquierda de x_r, y entonces se tiene $xr > xr'$ y por tanto $|Xl| >= |x_r| > |x'_r| > |Xl|$, lo cual es una contradicción.
Se tiene que, si x_r es válido, x'_r a la izquierda de x_r también es válido.
Entonces tengamos un x_r a la derecha de Xl tal que Xl no es válido para x_min respecto a Xl y con D. Esto implica que $|Xl| < |xr|$ o $D_2(Xl, xr) > D$. Sea xr' un punto a la derecha de xr, tenemos que si $D_2(Xl, xr) > D$, $D_2(Xl, xr')$ también lo sería. Así que, supongamos que esa no es al condición que invalida a xr, tiene que ser, por tanto, que $|Xl| < |xr|$, pero como tenemos que $Xl < 0$, entonces $xr > 0$ y por tanto xr' a la derecha de xr cumple que $xr' > xr > |Xl|$, con lo cual $|Xl| < |xr'|$ y se tendría que en cualquier caso, si xr no es válido, ningún xr' a la derecha de xr será válido.
Luego, uniendo ambas situaciones tenemos que ante el predicado: Es xr a la derecha de Xl un x_min para Xl que valida D?, la evaluación es de la forma "si","si",...,"si","no","no",..,"no". Entonces en el último "si" tenemos el mejor x_min para Xl que valida D.

El caso para cuando se elige $Xr = x\_max$, es simétrico a este.

```
#source va a ser el Xl/Xr elegido y target el xl/xr a #validar
def valid(source, target, D, points):
    if D_2(source, target) > D:
        return False
    if abs(source)<abs(target):
        return False
    return True

def check(D, n, points):
    min_sd = INF
```

```
        for Xl in points[1....n]:
            if Xl > 0: break
            xr = Xl
            end = n
            while xr+ 1 < end:
                mid = (xr+end)//2
                if valid(Xl, mid, D, points):
                    xr = mid
                else:
                    end = mid
            Calcular los D_2.
            Actualizar min_sd
            if min_sd <= D:
                return True
        for Xr in points[1....n]:
            if Xr < 0: break
            xl = Xr
            end = -1
            while end+ 1 < xl:
                mid = (xl+end)//2
                if valid(Xr, mid, D, points):
                    xl = mid
                else:
                    end = mid
            Calcular los D_2.
            Actualizar min_sd
            if min_sd <= D:
                return True
        return False
```

Código en: double_bs_solution.py

Complejidad temporal: Tenemos ya el $O(nlogn)$ de la ordenación, luego, hacemos una búsqueda binaria en el intervalo [0,MAX], y en cada paso hacemos *check*(). En *check*(), recorremos todas las $xl < 0$ y se eligen como x_max, y se hace una búsqueda por los restantes puntos. Si no se ha encontrado una distribución válida aún, se hace el análisis en el sentido inverso para las $xr > 0$. En los casos que no haya validez para D que son los peores escenarios para el algoritmo, se realizara la iteración sobre los n puntos y se hará una búsqueda binaria en los subconjuntos candidatos de máximo $n-1$ puntos, los cual está acotado por n. Entonces tenemos que buscar el óptimo se hará en $O(logMAX * nlogn)$.

Solución con Dos punteros: El mismo análisis que se siguió en la solución anterior se sigue en esta, lo que en este caso se utiliza en vez de una búsqueda binaria sobre los extremos, un segundo puntero para iterar por el x_min. Vamos a ver el caso iterando desde Xl(el caso desde Xr es simétrico) .

```
xr = 1
for Xl in range(1,n+1):
    while xr<n && D2(points[Xl].x-points[xr+1].x)<= D &&|points[Xl].x|>=|points[xr+1].x|:
        xr+=1
    while |points[Xl].x|<|points[xr].x|:
        xr-=1
    Calcular los D_2.
    if max(D2s) <= D:
        return True
xl = n
for Xr in range(n,0,-1):
    while xl>1 && D2(points[xl-1].x-points[xr].x)<=D &&|points[xl-1].x|<=|points[Xr].x|:
        xl-=1
    while |points[xl].x|>|points[Xr].x|:
        xl+=1
    Calcular los D_2.
    if max(D2s) <= D:
        return True
```

Código en: two_pointers_solution.py

Sabemos que el mejor x_min para Xl va a ser el último que es válido cumpliendo las dos condiones expresadas:
1) $|x\_max| \geq |x\_min|$
2)$D_2(x\_max, x\_min) \leq D$. para que valide a D
Demostramos esto en el predicado de la búsqueda binaria.
Tenemos también que, a medida que avance el puntero que selecciona Xl, los valores de Xl_i se comportan de manera tal de que $|Xl_i| > |Xl_{i+1}|$ $\forall i$, porque se avanza hacia la derecha y solo en los negativos en un arreglo ordenado.
Entonces la cuestión está en ver como ocurren las transiciones en el puntero xr que apunta a x_min.
Inicialmente se va a mover hasta el mejor xr posible de Xl_1. Analicemos los casos en los que se detendría el puntero.
Primera condición de parada, se alcanzó el final de los puntos, con lo cual no habría que analizar más nada porque significa que el conjunto está abarcado por las X a partir de Xl.
Si se llega a la segunda condición de parada esta ocurrirá porque, si se mueve el puntero xr a xr_{+1}, el D_2 de los extremos sobrepasará D. Pero en este caso como el puntero comprueba que no va a poder cumplir la condición para D, no avanza.
Por tanto el que avanza es el puntero Xl, que al moverse desde i a $i+1$ se mantendrá cumpliendo que la $D_2(Xl, xr) \leq D$ porque $|Xl_i| > |Xl_{i+1}|$ $\forall i$. Tenemos entonces que la condición del diámetro al cuadrado nunca causa retroceso en el puntero de xr(derecha).
Entonces, hay retroceso, cuando al avanzar Xl_i a Xl_{i+1} ocurre que $|Xl_{i+1}| < |xr|$, con lo cual el puntero retrocede hasta el primer $xr' < xr$(por la derecha) que valida a Xl_{i+1}.
Pero una vez que el puntero retrocede ya nunca avanza más, se mantiene o retrocede.

Esto se cumple, porque, como hemos visto, $|Xl_i| > |Xl_{i+1}|$, y como el puntero retrocedió a xr' que cumplía $|Xl_{i+1}| \geq |xr'|$, y cualquier otro $xr_k^{'}$ a la derecha de de xr' va a cumplir que $|xr_k^{'}| > |xr'|$ y se tiene entonces que $|xr'| < |Xl_{i+1}| \leq |xr|$ y que cualquier Xl' a la derecha de Xl_{i+1} cumple que $|Xl_{i+1}| > |Xl'|$ con lo cual, una vez retrocede, no avanza de nuevo.

La operación es análoga cuando se hace desde la derecha(positivos), buscando el x_min hacia los negativos.

Complejidad temporal: Tenemos que el puntero no va avanzar más de n elementos ni tampoco va a retroceder más de n elementos. Con lo cual, tenemos que la cantidad de moviemientos del puntero para mantener los xr es $O(n)$. Los D_2 se calculan en tiempo constante, con lo cual, el $check(D)$ ocurre en $O(n)$.

Luego tenemos que la complejidad temporal del algoritmo va a ser $O(nlogn + nlogMAX)$, con lo cual queda en $O(nlog(max(n, MAX))$.

4 Generador y tester

En la carpeta *.test/* se encuentran 2 subcarpetas *in/, out/*, cada una contiene repectivamente, las entradas generadas y las salidas obtenidas por el algoritmo de fuerza bruta.
También se incluyen algunos casos tomados del tester del codeforce, estos se identifican por su nombre: $t < num > cf$.

El generador de casos *test/test_generator.py*, genera juegos de datos de entrada de forma completamente aleatoria, siempre dentro de las restricciones del problema, o permite ingresar cotas manualmente, así como especificar si se desea que todos los valores generados sean de un signo específico(+ o -). La cantidad de puntos a generar está restringida a 22 por el 2^n de la fuerza bruta.

El generador coloca los archivos *.in* y *.out* en sus carpetas correspondientes.

Para validar las soluciones de los algoritmos, se utiliza el *checker.py* que permite elegir la implementación a validar y da dos opciones para los casos de prueba: 1) Correr un caso específico o 2) correr todos los casos.

La estrategia seguida para validar las soluciones fue la siguiente: Se leen los archivos de entrada y salida del test hechos por el generador y para cada uno se computa la solución del algoritmo seleccionado, se comprueba si el valor dado como óptimo coincide.