

Instrucciones

De los siguientes 11 problemas que se plantean resolver al menos 4 problemas para completar el ejercicio. *Se otorgará calificación adicional al resolver más.

- Incluir portada con los de datos del alumno, datos del trabajo y fotografía del alumno
- Recordar manejar encabezados y pies de página
- Se deberá incluir la captura de pantalla del problema aceptado en el juez online con fecha, hora y nombre de usuario.
- Incluir la redacción de cada ejercicio en un documento con portada que incluye:

Explicación de cada solución Greedy y su análisis del orden de complejidad

Incluir el algoritmo y código de la solución.

- Para que los ejercicios cuenten al 100% deberán de contestarse al menos 4 correctamente.



Algoritmo Minimal Coverage

Given several segments of line (int the X axis) with coordinates $[L_i, R_i]$. You are to choose the minimal amount of them, such they would completely cover the segment $[0, M]$.

Input

The first line is the number of test cases, followed by a blank line.

Each test case in the input should contains an integer M ($1 \leq M \leq 5000$), followed by pairs " $L_i R_i$ " ($|L_i|, |R_i| \leq 50000, i \leq 100000$), each on a separate line. Each test case of input is terminated by pair '0 0'.

Each test case will be separated by a single line.

Output

For each test case, in the first line of output your programm should print the minimal number of line segments which can cover segment $[0, M]$. In the following lines, the coordinates of segments, sorted by their left end (L_i), should be printed in the same format as in the input. Pair '0 0' should not be printed. If $[0, M]$ can not be covered by given line segments, your programm should print 'o' (without quotes).

Print a blank line between the outputs for two consecutive test cases.

Explicación del algoritmo

Este algoritmo se resuelve de una manera bastante concreta, lo que haremos es buscar entre líneas que se ajusten a una gran línea usando el mínimo posible para este algoritmo haremos lo siguiente

```
int main(){
    scanf("%d", &numCasos);
    while (numCasos--){
        numlineas = 0;
        resultado = 0;
        posActual = 0;

        scanf("%d", &tamLinea);
        while(scanf("%d %d", &lineas[numlineas][0], &lineas[numlineas][1]) && (lineas[numlineas][0] || lineas[numlineas][1])){
            numlineas++;
        }
    }
}
```

Para este algoritmo empezaremos escaneando el numero total de casos que se analizaran en nuestro programa, el programa se ejecutara todo en un ciclo hasta haber analizado todos los casos que se ingresen

Posteriormente tendremos 3 variables la cual nos dirán el número de líneas totales que se tienen para cubrir la línea principal, la variable resultado tendrá el número total del líneas que usaremos al final y la variable posActual, determinara en que posición de la línea que se tiene que cubrir estamos

Una vez inicializadas lo que haremos es escanear el tamaño de la línea que tenemos que cubrir para posteriormente mediante un ciclo escanear todas las líneas que tenemos para cubrir la gran línea principal, si estas líneas en su posición de inicio y de final es igual a 0 y 0 ese será el final de las líneas para ese problema, mientras se este ejecutando este ciclo iremos sumando en 1 el total de líneas analizadas para futuros ciclos que tendremos

```
mergeSort(0, numlineas-1);  
  
for (i = 0; i < numlineas && posActual < tamLinea; i++){  
    if (lineas[i][0] > posActual) break;  
  
    lineasResultado[resultado][0] = lineas[i][0];  
    lineasResultado[resultado][1] = lineas[i][1];  
  
    resultado++;  
  
    avance = lineas[i][1];  
  
    while (i + 1 < numlineas && lineas[i + 1][0] <= posActual){  
        if (max(lineas[++i][1], avance)){  
            avance = lineas[i][1];  
            lineasResultado[resultado-1][0] = lineas[i][0];  
            lineasResultado[resultado-1][1] = lineas[i][1];  
        }  
    }  
  
    posActual = avance;
```

Una vez que ya tenemos todas las líneas escaneadas, lo que haremos es ordenarlo mediante el método de merge, una vez ordenada nuestra lista de líneas ejecutaremos el ciclo que se encargara de analizar las líneas, tendremos un ciclo for que se ejecutara hasta que hayamos acabado de analizar todas las líneas y se haya alcanzado a cubrir la línea a cubrir, primero lo que haremos es verificar que la línea a analizar sea válida, para ello esta línea tiene que empezar antes o en la misma posición en la que estamos, si estamos en 0 que es el inicio

pero si la línea empieza a partir de 5, esta línea no será válida, pero si la línea empieza en -10 y estamos en 0 esta línea será válida

Ya que tenemos una línea valida, lo que haremos es meterla en nuestro arreglo de líneas resultados donde tendremos todas las líneas que usaremos para cubrir nuestra línea principal, y sumaremos en 1 el numero resultado que indica el número total de líneas que usaremos

Nuestra variable de avance será igual a hasta dónde llega nuestra línea, es decir si tenemos una línea que va desde -10 hasta 5 el avance será igual a 5

Ahora viene un ciclo anidado que se ejecutara al menos un máximo igual al número de líneas que se analizaran, esto no hace cuadrático al algoritmo, este ciclo busca una mejor línea, es decir si tenemos una línea mas cercana al origen y que abra que mas espacio, usaremos esa línea en lugar de la que tenemos actualmente, para ello veremos si la línea que tenemos sigue siendo menor o igual a la posición actual y sea mayor que el avance actual usaremos esa nueva línea y la guardaremos en la posición correspondiente que reemplaza a la línea analizada anteriormente

Una vez que hayamos escaneado y encontrado la línea mejor lo que haremos es actualizar la posición actual igual al avance y seguiremos analizando las líneas siguientes

```
if (posActual < tamLinea){  
    printf("0\n");  
}  
else{  
    printf("%d\n", resultado);  
    for (i = 0; i < resultado; i++){  
        printf("%d %d\n", lineasResultado[i][0], lineasResultado[i][1]);  
    }  
    if (numCasos) printf("\n");  
}  
return 0;
```

Ya que hayamos analizado todas las líneas lo que haremos es comprobar si hemos cubierto toda la línea, en caso de que no nuestro algoritmo imprimirá un 0 que significa que no hay solución para el problema, en cambio lo que haremos es imprimir el número de líneas que usaremos además de imprimir estas líneas que usaremos para cubrir la línea principal, en caso de que hayamos cubierto este caso lo que haremos es imprimir un salto de línea como lo requiere el juez

Orden de complejidad

```

int main(){
    scanf("%d", &numCasos); → O(1)
    while (numCasos--){
        numlineas = 0;
        resultado = 0;
        posActual = 0;
        scanf("%d", &tamLinea);
        while(scanf("%d %d", &lineas[numlineas][0], &lineas[numlineas][1]) && (lineas[numlineas][0] || lineas[numlineas][1])){
            numlineas++;
        }
        mergeSort(0, numlineas-1); → O(n log n)
        for (i = 0; i < numlineas && posActual < tamLinea; i++){
            if (lineas[i][0] > posActual) break; → O(1)
            lineasResultado[resultado][0] = lineas[i][0];
            lineasResultado[resultado][1] = lineas[i][1];
            resultado++;
            avance = lineas[i][1];
            while (i + 1 < numlineas && lineas[i + 1][0] <= posActual){
                if (max(lineas[i+1][1], avance))
                {
                    avance = lineas[i+1][1];
                    lineasResultado[resultado-1][0] = lineas[i][0];
                    lineasResultado[resultado-1][1] = lineas[i][1];
                }
            }
            posActual = avance; → O(1)
        }
        if (posActual < tamLinea){
            printf("0\n"); → O(1)
        }
        else{
            printf("%d\n", resultado); → O(1)
            for (i = 0; i < resultado; i++){
                printf("%d %d\n", lineasResultado[i][0], lineasResultado[i][1]); → O(n)
            }
            if (numCasos) printf("\n"); → O(1)
        }
    }
    return 0; → O(1)
}

```

Diagrama de complejidad:

- $O(1)$ para `scanf("%d", &numCasos);`
- $O(1)$ para `numlineas = 0; resultado = 0; posActual = 0;`
- $O(1)$ para `scanf("%d", &tamLinea);`
- $O(K)$ para el `while(scanf(...))` (donde K es el número de líneas)
- $O(n \log n)$ para `mergeSort(0, numlineas-1);`
- $O(1)$ para `if (lineas[i][0] > posActual) break;`
- $O(1)$ para `lineasResultado[resultado][0] = lineas[i][0];` y `lineasResultado[resultado][1] = lineas[i][1];`
- $O(1)$ para `resultado++;`
- $O(1)$ para `avance = lineas[i][1];`
- $O(K-1)$ para el `while (i + 1 < numlineas && lineas[i + 1][0] <= posActual){}` (donde K es el número de líneas)
- $O(1)$ para `posActual = avance;`
- $O(1)$ para `if (posActual < tamLinea){}`
- $O(1)$ para `printf("0\n");`
- $O(1)$ para `printf("%d\n", resultado);`
- $O(n)$ para el `for (i = 0; i < resultado; i++){}` (donde n es el número de líneas)
- $O(1)$ para `if (numCasos) printf("\n");`
- $O(1)$ para `return 0;`

Complejidad total: $O(n \cdot K)$

Este algoritmo tiene muchos ciclos anidados pero tiene peculiaridades, por ejemplo el primer ciclo anidado que es el `while` que escanea los valores para nuestras líneas, que es de orden K o igual al número de líneas que se ingresen, el siguiente ciclo es el `for` que es orden K pero dentro tiene otro ciclo anidado por que este orden es K y no cuadrático, esto se debe a que solo se recorre una vez la lista de líneas, es decir si encontramos una línea mejor en el ciclo `while` anidado, no se analizara esa línea de nuevo ya que estamos aumentando el índice de i directamente, por lo tanto este ciclo es de orden lineal,

El ciclo posterior que ya no esta anidado es de orden r , que es el orden de línea que vamos a usar, en el peor caso que se tenga es que el resultado sea igual al número de las líneas ingresadas, pero en este caso le pondremos r para no confundir

Por último, tenemos el ciclo principal que se ejecutara n veces o n número de casos que analicemos, teniendo un orden de complejidad final igual a:


$$O(n * k)$$

Pero no hemos tomado en cuenta el ordenamiento de merge que es de orden superior, en este caso tenemos la cota del algoritmo de ordenamiento en si, pero si tenemos en cuenta el ordenamiento como debe de ser la cota de complejidad mayor es igual a la del ordenamiento y el numero de casos que tenemos que analizar, entonces la cota final completa para nuestro algoritmo es la siguiente

$$O(n * (k * \log(k)))$$

Captura de pantalla

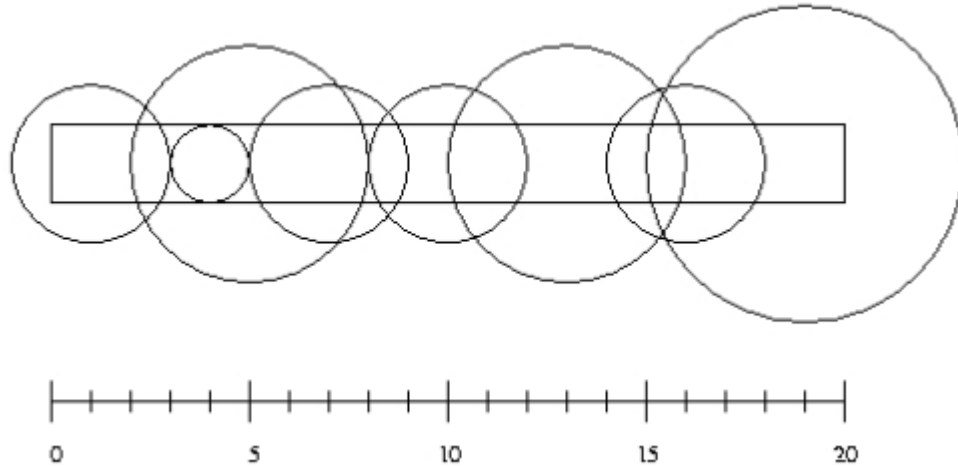
La pagina no permite mostrar si el algoritmo fue aceptado junto al nombre de usuario por ello al final en los anexos tendremos una foto con todos los problemas analizados que se muestra en el perfil donde ahí si se tienen los nombres



My Submissions					
#	Problem	Verdict	Language	Run Time	Submission Date
27040818	10020 Minimal coverage	Accepted	ANSI C	0.030	2021-12-09 20:50:48

Algoritmo Watering Grass

n sprinklers are installed in a horizontal strip of grass l meters long and w meters wide. Each sprinkler is installed at the horizontal center line of the strip. For each sprinkler we are given its position as the distance from the left end of the center line and its radius of operation. What is the minimum number of sprinklers to turn on in order to water the entire strip of grass?



Input

Input consists of a number of cases. The first line for each case contains integer numbers n , l and w with $n \leq 10000$. The next n lines contain two integers giving the position of a sprinkler and its radius of operation. (The picture above illustrates the first case from the sample input.)

Output

For each test case output the minimum number of sprinklers needed to water the entire strip of grass. If it is impossible to water the entire strip output '-1'.

Explicación del algoritmo

Para comenzar con el algoritmo primero debemos tener en cuenta todas las entradas y variables que necesitamos para nuestro programa

```
double aspersores[10000][2];

int main(){

    int numAsper = 0, largo = 0, ancho = 0, posicion = 0, radio = 0;
    int i = 0, resultado = 0;
    double distanciaX = 0, aux = 0, avance = 0;
```

Tendremos 1 arreglo de 2 dimensiones donde todas las primeras posiciones serán los valores de las posiciones de nuestros aspersores, y en las posiciones correspondientes a la segunda línea del arreglo tendremos el radio de cada aspersor

También tendremos el numero de aspersores, el largo y ancho de nuestro campo que tenemos que cubrir con nuestros aspersores, además de las posiciones y el radio de nuestros aspersores

Una variable para nuestros ciclos, y una variable resultado que tendrá el numero de aspersores necesarios para regar nuestro campo

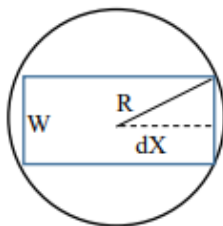
También tendremos otras que nos servirán para la obtención de la distancia que obtendremos con el teorema de Pitágoras, además de nuestras variables auxiliar y avance que nos ayudaran a determinar si las posiciones donde estamos son validas y podamos usar el aspersor que se esté analizando

```
while (scanf("%d %d %d", &numAsper, &largo, &ancho) != EOF){  
    for (i = 0; i < numAsper; i++){  
        scanf("%d %d", &posicion, &radio);  
        distanciaX = sqrt((double)radio*radio-(ancho/2.0)*(ancho/2.0));  
        if (distanciaX == distanciaX){  
            aspersores[i][0] = (double)posicion-distanciaX;  
            aspersores[i][1] = (double)posicion+distanciaX;  
        }  
    }  
}
```

Ahora bien, el ciclo principal de nuestro proyecto que consiste en que mientras existan casos que se falten por analizar no se detendrá el algoritmo, este ciclo se encarga de escanear el número de aspersor el largo y el ancho del campo, y hasta que no se

hayamos escaneado todos los aspersores totales y las medidas de los campos no se detendrá el algoritmo

Una vez que tenemos estas primeras variables lo que haremos es escanear las posiciones y los radios de cada aspersor que tenemos, dichos valores los almacenaremos en las variables correspondientes ahora obtendremos un calculo gracias al teorema de Pitágoras



Como no podemos operar con círculos debido a que es muy complicado interpretar estos círculos en un terreno cuadrado lo que haremos es obtener rectángulos inscritos, pero para saber las distancia Dx que es la que ocuparemos para nuestro problema tendremos que obtener el cateto adyacente teniendo como hipotenusa el radio de nuestro círculo y como cateto opuesto la mitad de nuestro ancho que tenemos en nuestro terreno, para

obtener este valor usaremos el teorema de Pitágoras que nos dice que la suma de los cuadrados de los catetos es igual al cuadrado de la hipotenusa por lo tanto para obtener esta distancia Dx despejaremos la formula para obtener la medida de este cateto la formula despejada quedaría de la siguiente manera

$$Dx = \sqrt{(r)^2 - \left(\frac{ancho}{2}\right)^2}$$

Ya que obtenemos esta distancia podemos obtener que tanto de nuestro círculo ahora rectángulo abarca desde la posición en la que esta

En la primera posición 0 de nuestro arreglo tendremos la posición del aspersor menos la distancia D_x y en la posición 1 tendremos la posición más la distancia de D_x

Es decir, tenemos la distancia del largo del rectángulo inscrito de nuestro aspersor con el cual podemos trabajar de manera más sencilla además de que es más útil que usar o tener en mente círculos

```
mergeSort(0, numAsper - 1);

aux = 0;
resultado = 0;

for (i = 0; i < numAsper && aux < largo; i++){
    if (aspersores[i][0] > aux) break;

    resultado++;

    avance = aspersores[i][1];

    while (i + 1 < numAsper && aspersores[i + 1][0] <= aux){
        avance = max(aspersores[i + 1][1], avance);
    }

    aux = avance;
}

if (aux < largo) resultado = -1;

printf("%d\n", resultado);

for (i = 0; i < numAsper; i++){
    aspersores[i][0] = 0;
    aspersores[i][1] = 0;
}
```

Ya que tenemos todas las distancias calculadas lo que haremos es ordenar estas medidas con respecto al primer elemento del arreglo es decir con respecto a la posición menos la distancia obtenida mediante el algoritmo de ordenamiento por mezcla

Ahora tendremos uso de nuestras variables de auxiliar y el resultado que tendrá el numero de aspersores que usaremos, tendremos un ciclo for que se ejecutara desde 0 hasta que hayamos alcanzado el máximo de aspersores que tenemos para esta instancia del problema y también

mientras no hayamos excedido el tamaño total del largo de nuestro campo que tenemos, ahora tendremos en cuenta lo siguiente si el aspersor desde su posición menos la distancia es mayor que auxiliar esta posición no será válida, es decir para que una posición sea válida tendrá que abarcar parte del área del rectángulo anterior, es decir si tenemos un rectángulo que inicia en 1 y acaba en 10 nuestro siguiente rectángulo valido tendrá que iniciar en una posición menor a 10 para que sea válido, si no cumple este requerimiento no estaremos cubriendo apropiadamente nuestro campo y no tendrá solución este problema para las posiciones de estos aspersores

Si nuestro siguiente aspersor es válido, sumaremos en 1 el numero de aspersores que necesitamos, lo que haremos es ahora es a nuestra variable avance darle el valor de la posición mas la distancia del aspersor que estamos analizando, tendremos un ciclo while que se ejecutara mientras no lleguemos al final de los aspersores y que el aspersor siguiente en su posición siguiente sea menor que aux

Una vez que ya hayamos analizado todos los aspersores lo que haremos es comprobar si nuestra variable auxiliar logro superar el tamaño del campo en caso de que si imprimiremos el numero total de aspersores, pero en caso de que no hayamos logrado cubrir todo el campo imprimiremos menos 1, posteriormente lo que haremos es vaciar las distancias de nuestro problema actual para no tener conflicto de valores con el siguiente

```
int main(){

    int numAsper = 0, largo = 0, ancho = 0, posicion = 0, radio = 0;
    int i = 0, resultado = 0;
    double distanciaX = 0, aux = 0, avance = 0;

    while (scanf("%d %d %d", &numAsper, &largo, &ancho) != EOF){
        for (i = 0; i < numAsper; i++){

            scanf("%d %d", &posicion, &radio);
            distanciaX = sqrt((double)radio*radio-(ancho/2.0)*(ancho/2.0));
            if (distanciaX == distanciaX){
                aspersores[i][0] = (double)posicion-distanciaX;
                aspersores[i][1] = (double)posicion+distanciaX;
            }
        }

        mergeSort(0, numAsper - 1);

        aux = 0;
        resultado = 0;

        for (i = 0; i < numAsper && aux < largo; i++){
            if (aspersores[i][0] > aux) break;
            resultado++;
            avance = aspersores[i][1];
            while (i + 1 < numAsper && aspersores[i + 1][0] <= aux){
                avance = max(aspersores[++i][1], avance);
            }
            aux = avance;
        }

        if (aux < largo) resultado = -1;

        printf("%d\n", resultado);

        for (i = 0; i < numAsper; i++){
            aspersores[i][0] = 0;
            aspersores[i][1] = 0;
        }
    }

    return 0;
}
```

Para este orden de complejidad tendremos en cuenta el principal gran ciclo while que se ejecutara n veces este número de n veces es el número total de casos que analizara el programa, el primer ciclo anidado sabemos que se ejecutara k veces donde k es el numero de aspersores que tenemos por caso, en este caso todas las instrucciones dentro de nuestro ciclo sonden orden lineal por lo tanto este ciclo es de orden lineal con valor de n

Las siguientes líneas son 2 igualaciones de nuestras variables que son constantes además de que tenemos la llamada de nuestro algoritmo de ordenamiento que sabemos que es de orden $n \log n$ por lo tanto de momento es el orden mas grande pero no lo tomaremos en cuenta de momento ya que primero obtendremos la cota del algoritmo greedy en si

Posteriormente tendremos otro ciclo que se ejecutara hasta que hayamos llegado hasta el total de aspersores o hasta que hayamos llegado al final de nuestro campo

Dentro tenemos otro ciclo while pero este ciclo anidado no hace que el algoritmo se llegue a cuadrático, debido a que cuando estamos analizando el aspersor se comparara con los siguientes en caso de que encuentre un aspersor mas grande, continuaremos el análisis desde ese aspersor y no es que recorramos el total de aspersores 2 veces en cada ciclo si no que lo recorreremos una única vez pero encontraremos los aspersores mas grandes y continuaremos a partir de ellos

Por lo tanto, este segundo ciclo anidado for es de orden lineal por que en el peor caso se ejecutara un total k veces

Ya que tenemos todos el análisis de nuestro algoritmo tendremos que todo el interior de nuestro ciclo principal es de orden lineal K mientras que por regla de la multiplicación tendremos que nuestro algoritmo greedy en si es de orden

$$O(n * k)$$

Pero si tomamos en cuenta el ordenamiento de merge tendremos que el orden de complejidad de todo nuestro algoritmo es de orden

$$O(n * (k * \log(k)))$$

Captura de Pantalla

En la página de UVA no muestra el nombre de usuario cuando vemos si el algoritmo es aceptado en el apartado de anexos se verán las ID de los algoritmos resueltos para corroborar que si fueron resueltos

My Submissions					
#	Problem	Verdict	Language	Run Time	Submission Date
27038285	10382 Watering Grass	Accepted	ANSI C	0.000	2021-12-08 20:06:31

Anexos

Codigo Mlimal Coverage

```
#include <stdio.h>
#include <stdlib.h>

int numCasos, i;

int tamLinea, numlineas, resultado, posActual, avance;

int lineas[100005][2], lineasResultado[100005][2];

void merge(int l, int m, int r);

void mergeSort(int l, int r);

int max (int a, int b);

int main(){

    scanf("%d", &numCasos);

    while (numCasos--){

        numlineas = 0;
        resultado = 0;
        posActual = 0;

        scanf("%d", &tamLinea);

        while(scanf("%d %d", &lineas[numlineas][0],
&lineas[numlineas][1]) && (lineas[numlineas][0] ||
lineas[numlineas][1])){
            numlineas++;
        }

        mergeSort(0, numlineas-1);

        for (i = 0; i < numlineas && posActual < tamLinea; i++){

            if (lineas[i][0] > posActual) break;

            lineasResultado[resultado][0] = lineas[i][0];
            lineasResultado[resultado][1] = lineas[i][1];

            resultado++;

            avance = lineas[i][1];

            while (i + 1 < numlineas && lineas[i + 1][0] <= posActual){

                if (max(lineas[++i][1], avance))
                {
                    avance = lineas[i][1];
                    lineasResultado[resultado-1][0] = lineas[i][0];
                    lineasResultado[resultado-1][1] = lineas[i][1];
                }
            }
        }
    }
}
```

```
        }
    }

    posActual = avance;
}

if (posActual < tamLinea){
    printf("0\n");
}
else{
    printf("%d\n", resultado);

    for (i = 0; i < resultado; i++)
        printf("%d %d\n", lineasResultado[i][0],
lineasResultado[i][1]);
}

if (numCasos) printf("\n");
}

return 0;
}

void merge(int l, int m, int r){

    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1][2], R[n2][2];

    for (i = 0; i < n1; i++){

        L[i][0] = lineas[l + i][0];
        L[i][1] = lineas[l + i][1];
    }

    for (j = 0; j < n2; j++){
        R[j][0] = lineas[m + 1 + j][0];
        R[j][1] = lineas[m + 1 + j][1];
    }

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2) {
        if (L[i][0] <= R[j][0]) {
            lineas[k][0] = L[i][0];
            lineas[k][1] = L[i][1];
            i++;
        }
        else {
            lineas[k][0] = R[j][0];
            lineas[k][1] = R[j][1];

```

```
        j++;
    }
    k++;
}

while (i < n1) {
    lineas[k][0] = L[i][0];
    lineas[k][1] = L[i][1];
    i++;
    k++;
}

while (j < n2) {
    lineas[k][0] = R[j][0];
    lineas[k][1] = R[j][1];
    j++;
    k++;
}
}

void mergeSort(int l, int r){

    if (l < r) {

        int m = l + (r - l) / 2;

        mergeSort(l, m);
        mergeSort(m + 1, r);

        merge(l, m, r);
    }
}

int max (int a, int b){

    return (a>b)?1:0;
}
```


Codigo Watering Grass

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void merge(int l, int m, int r);

void mergeSort(int l, int r);

double max (double a, double b);

double aspersores[10000][2];

int main(){

    int numAsper = 0, largo = 0, ancho = 0, posicion = 0, radio = 0;
    int i = 0, resultado = 0;
    double distanciaX = 0, aux = 0, avance = 0;

    while (scanf("%d %d %d", &numAsper, &largo, &ancho) != EOF){

        for (i = 0; i < numAsper; i++){

            scanf("%d %d", &posicion, &radio);

            distanciaX = sqrt((double)radio*radio-
(ancho/2.0)*(ancho/2.0));

            if (distanciaX == distanciaX){
                aspersores[i][0] = (double)posicion-distanciaX;
                aspersores[i][1] = (double)posicion+distanciaX;
            }
        }

        mergeSort(0, numAsper - 1);

        aux = 0;
        resultado = 0;

        for (i = 0; i < numAsper && aux < largo; i++){

            if (aspersores[i][0] > aux) break;

            resultado++;

            avance = aspersores[i][1];

            while (i + 1 < numAsper && aspersores[i + 1][0] <= aux){
                avance = max(aspersores[++i][1], avance);
            }

            aux = avance;
        }

        if (aux < largo) resultado = -1;
    }
}
```

```
printf("%d\n", resultado);

for (i = 0; i < numAsper; i++){
    aspersores[i][0] = 0;
    aspersores[i][1] = 0;
}

return 0;
}

void merge(int l, int m, int r){

    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    double L[n1][2], R[n2][2];

    for (i = 0; i < n1; i++){

        L[i][0] = aspersores[l + i][0];
        L[i][1] = aspersores[l + i][1];

    }
    for (j = 0; j < n2; j++){
        R[j][0] = aspersores[m + 1 + j][0];
        R[j][1] = aspersores[m + 1 + j][1];
    }

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2) {
        if (L[i][0] <= R[j][0]) {
            aspersores[k][0] = L[i][0];
            aspersores[k][1] = L[i][1];
            i++;
        }
        else {
            aspersores[k][0] = R[j][0];
            aspersores[k][1] = R[j][1];
            j++;
        }
        k++;
    }

    while (i < n1) {
        aspersores[k][0] = L[i][0];
        aspersores[k][1] = L[i][1];
        i++;
        k++;
    }

    while (j < n2) {
        aspersores[k][0] = R[j][0];
```

```
        aspersores[k][1] = R[j][1];  
        j++;  
        k++;  
    }  
}  
  
void mergeSort(int l, int r){  
    if (l < r) {  
        int m = l + (r - l) / 2;  
        mergeSort(l, m);  
        mergeSort(m + 1, r);  
        merge(l, m, r);  
    }  
}  
  
double max (double a, double b){  
    return (a>b)?a:b;  
}
```



Bibliografía

