Instituto Politécnico Nacional
Escuela Superior de Computo

Ejercicio 12

"Ejercicio 12 Diseño de Soluciones mediante programación Voraz (Greedy)"

Mora Ayala José Antonio

Análisis de Algoritmos

CONTENIDO

INSTRUCCIONES

[1] 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 1 SCARECROW

DESCRIPCIÓN DEL PROBLEMA [2]

Taso owns a very long field. He plans to grow different types of crops in the upcoming growing season. The area, however, is full of crows and Taso fears that they might feed on most of the crops. For this reason, he has decided to place some scarecrows at different locations of the field. The field can be modeled as a 1×N grid. Some parts of the field are infertile and that means you cannot grow any crops on them. A scarecrow, when placed on a spot, covers the cell to its immediate left and right along with the cell it is on. Given the description of the field, what is the minimum number of scarecrows that needs to be placed so that all the useful section of the field is covered? Useful section refers to cells where crops can be grown.

Input

Input starts with an integer T (≤ 100), denoting the number of test cases. Each case starts with a line containing an integer N (0 < N < 100). The next line contains N characters that describe the field.

A dot (.) indicates a crop-growing spot and a hash (#) indicates an infertile region.

Output

For each case, output the case number first followed by the number of scarecrows that need to be placed.

Sample

Input
3
3
.#.
11
...##....##
2
##

Sample Output

Case 1: 1

Case 2:

3 Case 3: 0

Tal como el problema nos solicita comenzaremos solicitando al usuario que ingresa una cantidad de casos que quiera probar, posteriormente una cadena que nos representa las zonas fértiles y las zonas infértiles, las cuales estan denotadas por "." y "# respectivamente. El objetivo es colocar a los espantapájaros de tal forma que se cubran todas aquellas zonas que son fértiles, considerando que cuando un espantapájaros es colocado cubre la celda de la izquierda y la derecha, así como en la que fue colocado.

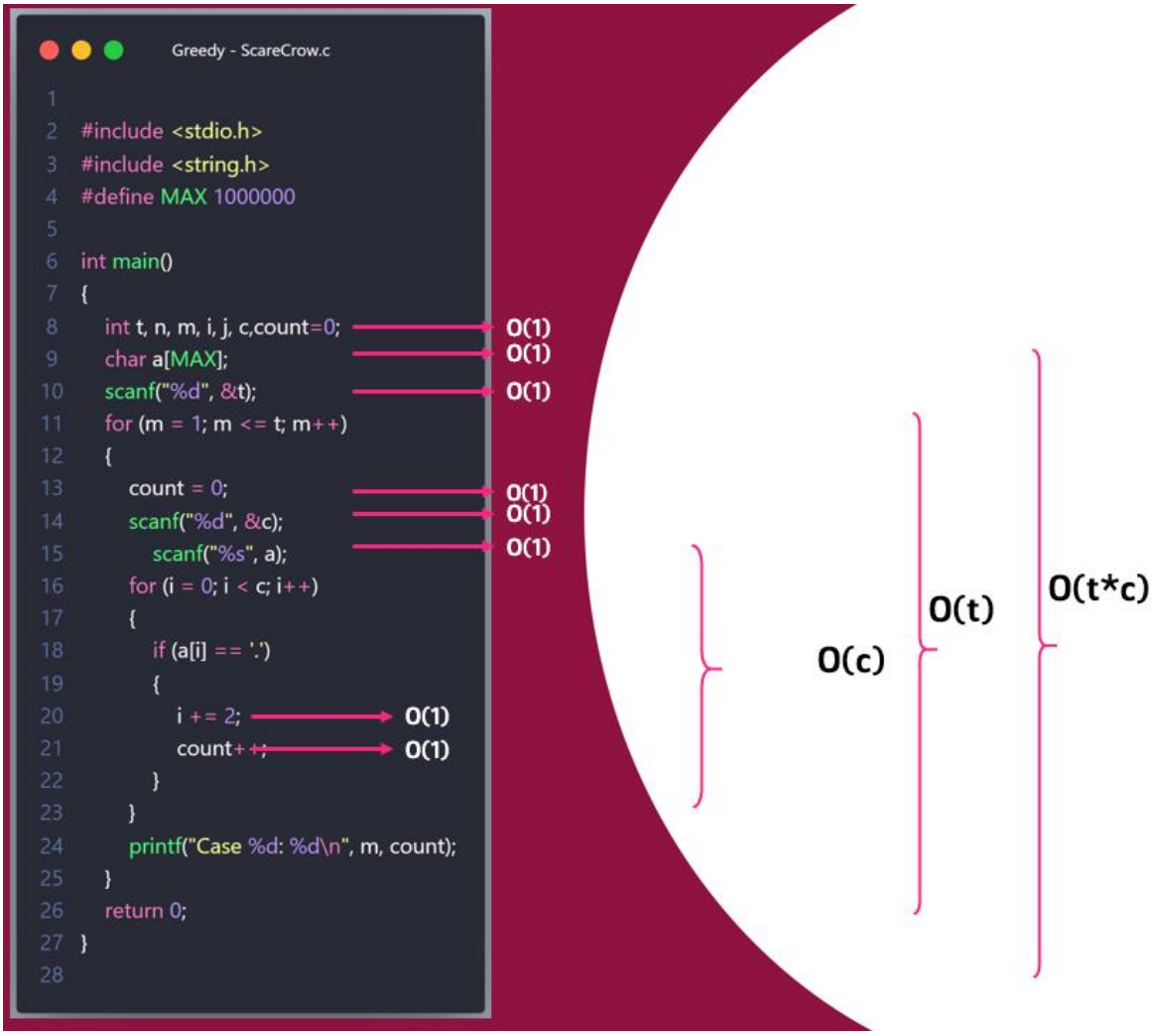
Teniendo estas consideraciones en cuenta, en si el algoritmo que nos da la respuesta no es muy complejo de desarrollar, pues nos valdremos de un ciclo for para la distinta cantidad de casos que van a ser analizados, dentro del mismo solicitamos la cantidad de caracteres que habrá en el campo y posteriormente el caso correspondiente, una vez realizado esto comenzamos con el segundo ciclo, el cual se encarga de analizar toda la cadena ingresada, cuando en una iteración encuentra un punto nuestro iterador de ciclo (i) incrementara en 2 unidades, así como la cantidad de espantapájaros que debe ser colocada, pues cada que encontramos un punto estamos considerando que el espantapájaros se colocara en la posición centras de las 3 en cuestión garantizando que cubre el espacio donde el punto fue encontrado así como el del centro en caso de que haya y el de la siguiente posición

"Análisis en Cota O

Como podemos ver para este tipo de algoritmos el análisis a ser realizado se torna bastante sencillo, pues todo es ejecutado desde la parte principal del programa, sin hacer uso de la recursión o alguna clase de función especial, al menos para este caso, podemos ver asignaciones y comparaciones las cuales ya hemos determinado que simplemente serán de un costo constante.

Ahora partiremos desde la parte mas interna de los ciclos, en el cual se realiza una condición y en caso de que sea cumplida se realizan dos operaciones aritméticas, así como una asignación, este ciclo es determinado por el tamaño del campo que se va a someter a evaluación para poder colocar los espantapájaros, al salir de este ciclo nos encontramos con una impresión la cual nos dará el número de caso analizado, así como la cantidad mínima de espantapájaros necesarios, el ciclo mas grande, el cual contiene al anteriormente descrito simplemente se iterara dependiendo de la cantidad de casos que el usuario quiera someter a evaluación, por lo que en el mejor caso será uno solo.

Cota O (t*c)



#	Problem	Verdict	Language	Run Time	Submission Date
27043018	12405 Scarecrow	Accepted	ANSI C	0.000	2021-12-10 18:08:50

ALGORITMO 2 "BEAR AND ROW"

DESCRIPCIÓN DEL PROBLEMA [3]

Limak is a little polar bear. He is playing a video game and he needs your help.

There is a row with N cells, each either empty or occupied by a soldier, denoted by '0' and '1' respectively. The goal of the game is to move all soldiers to the right (they should occupy some number of rightmost cells).

The only possible command is choosing a soldier and telling him to move to the right as far as possible. Choosing a soldier takes 1 second, and a soldier moves with the speed of a cell per second. The soldier stops immediately if he is in the last cell of the row or the next cell is already occupied. Limak isn't allowed to choose a soldier that can't move at all (the chosen soldier must move at least one cell to the right).

Limak enjoys this game very much and wants to play as long as possible. In particular, he doesn't start a new command while the previously chosen soldier moves. Can you tell him, how many seconds he can play at most?

Input

The first line of the input contains an integer T denoting the number of test cases. The description of T test cases follows.

The only line of each test case contains a string S describing the row with N cells. Each character is either '0' or '1', denoting an empty cell or a cell with a soldier respectively.

Output

For each test case, output a single line containing one integer — the maximum possible number of seconds Limak will play the game.

Constraints

- $1 \le T \le 5$
- $1 \le N \le 105$ (N denotes the length of the string S)

Subtasks

- Subtask #1 (25 points): $1 \le N \le 10$
- Subtask #2 (25 points): $1 \le N \le 2000$
- Subtask #3 (50 points): Original constraints.

Sample Input 1

4

10100

1100001

000000000111

001110100011010

Sample Output 1

8

10

0

48

Tal como el problema nos está solicitando primero comenzamos solicitando al usuario que ingresa una n cantidad de casos para las cuales el algoritmo se ira iterando, en donde dada una cadena (la cual nos indica de cierta forma el espacio que debemos considerar, donde un cero significa una celda vacía y un 1 una posición donde se encuentra un soldado).

En si lo que el problema nos esta solicitando es realizar el conteo de los segundos que Limak podrá jugar el juego, el cual esta determinado por el movimiento que se tendría que hacer para poder pasar a todos los "soldados" al extremo derecho.

Debemos tener unas consideraciones, cada posición recorrida a la derecha es un incremento a la cantidad de segundos que tiene para jugar dicho juego, así como que debemos incrementar la cantidad de segundos cuando toma uno de los soldados (ósea al momento de encontrar un numero 1 dentro de nuestra cadena o bien campo de juego el contador incrementa)

Así mismo si un soldado en la posición siguiente se encuentra con un soldado este parara su recorrido, no se lo puede saltar, en caso de que hubiese más casillas vacías después del segundo soldado en cuestión.

Una vez que hemos comprendido el problema, este se torna bastante simple de resolver

Comenzaremos solicitando el numero de casos que serán analizados o bien la cantidad de veces que Limak quiera jugar, en cada iteración de este ciclo se ira pidiendo un campo de juego distinto, en el cual se podrán introducir las diferentes posiciones de los soldados y las celdas vacías, recordemos que **1 = soldado, 0 = vacío** por lo cual estos son los caracteres validos que se consideraran dentro de la cadena para el resultado proporcionado

Dado que nuestro objetivo es mover a los soldados al extremo derecho y considerando que cuando se encuentra un soldado el soldado en cuestión se pararía, nos conviene comenzar el ciclo desde la última posición, en donde procederemos a analizar cada posición de la cadena de 0 y 1 dada, así cuando encontremos un 0 realizaremos el incremento a dicho contador, pues quiere decir que habrá una casilla que puede ser ocupada.

Cuando nos hemos encontrado con un soldado incrementamos nuestra variable y, la cual determina la cantidad de segundos tomada para ese soldado en cuestión mas aparte debemos realizar el incremento en 1 que es la cantidad de segundos que toma agarrar el soldado, actualizamos nuestros segundos y continuamos con el análisis de la cadena, debemos reiniciar la cantidad de celdas vacías, para que en caso de que haya mas celdas vacías esta se reinicie desde la última posición del primer soldado encontrado, pues la variable y ya nos almacena la cantidad de celdas vacías a considerar y la cantidad de segundos que toma agarrar el soldado.

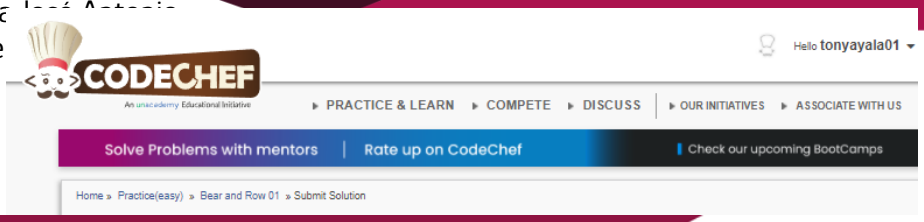
Greedy - BearandRow.c

```c
int main()
{
    int casos,
    // Variables de ciclos
    i,j;
    long long seg,y,tam,
    // Celda vacia
    vacio,soldado;
    // Cadena de caracteres recibida
    char a[MAX];
    scanf("%d", &casos);

    while(casos--){
        scanf("%s", a);
        // Longitud de la cadena
        tam= strlen(a);
        // Reiniciando las variables para las distintas partidas
        seg=vacio=y=soldado = 0;
        for(i=tam-1;i>=0;--i){
            if(a[i]=='0'){
                vacio++;
            }
            else if(a[i]=='1' && vacio>0){
                y= vacio+y+1;
                seg+= y;
                vacio=0;
            }
            else if(a[i]=='1' && vacio==0){
                seg+=y;
            }
        }
        printf("%lld\n",seg);
    }

    return 0;
}
```

Análisis en Cota O

El análisis en cota O para este algoritmo s bastante sencillo, pues en su mayoría nos encontramos con asignaciones y algunas operaciones aritméticas, las cuales consisten principalmente en incremento en una unidad, así mismo podemos observar la presencia de comparaciones las cuales presentan todas un orden constante, el ciclo que se encuentra dentro del while (recordemos que para el análisis en cota O debemos partir desde lo más interno a lo más externo), este ciclo for se iterara dependiendo del tamaño del campo de juego que fue ingresado, tal como se menciono en la explicación de la solución debemos ir de derecha a izquierda por mera conveniencia, es por eso que realizamos el ciclo de esa forma, finalmente el while se ejecutara dependiendo de la cantidad de casos que el usuario quiera someter a evaluación dicho algoritmo con distintos campos de juego, por lo que nuestro mejor caso sería que solo quisiera hacerlo 1 vez

Cota O (tam*casos)



Greedy - BearandRow.c

```
1  int main()
2  {
3      int casos,                          → O(1)
4      // Variables de ciclos
5      i,j;
6      long long seg,y,tam,                → O(1)
7      // Celda vacia
8      vacio,soldado;
9      // Cadena de caracteres recibida
10     char a[MAX];                        → O(1)
11     scanf("%d", &casos);                → O(1)
12
13     while(casos--){
14         scanf("%s", a);                 → O(1)
15         // Longitud de la cadena
16         tam= strlen(a);                 → O(1)
17         // Reiniciando las variables para las distintas partidas
18         seg=vacio=y=soldado = 0;
19         for(i=tam-1;i>=0;--i){
20             if(a[i]=='0'){              → O(1)
21                 vacio++;                → O(1)
22             }
23             else if(a[i]=='1' && vacio>0){
24                 y= vacio+y+1;           → O(1)
25                 seg+= y;                → O(1)
26                 vacio=0;                → O(1)
27             }
28             else if(a[i]=='1' && vacio==0){  → O(1)
29                 seg+=y;                 → O(1)
30             }
31         }
32         printf("%lld\n",seg);
33     }
34
35     return 0;
36 }
```

O(tam)

O(tam*casos)

ALGORITMO 3

DESCRIPCIÓN DEL PROBLEMA [2]

Given several segments of line (int the X axis) with coordinates [Li , Ri]. 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 ≤ M ≤ 5000), followed by pairs "Li Ri" (|Li |, |Ri | ≤ 50000, i ≤ 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 (Li), 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 '0' (without quotes).

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

EXPLICACIÓN

Como podemos observar en nuestro código comenzamos solicitando el numero total de casos que serán sometidos a evaluación en nuestro programa, los cuales determinan que tantas veces se realizara este algoritmo, posteriormente nos encontramos con 3 variables las cuales nos indican el número de líneas totales que se tienen para cubrir la línea principal, la variable "res" es la que almacena el resultado obtenido finalmente, mientras que la de posición actual nos indica el lugar de la línea que debe ser cubierta en la cual nos encontremos en ese momento. Posterior a esto solicitamos que nos indiquen el tamaño de la línea que debe ser cubierta, para la cual este algoritmo está siendo desarrollado, posterior a esto comenzamos a solicitar que ingresen el valor de las líneas que tenemos para poder cubrir la del mayor tamaño.

Una vez que tenemos los valores de todas las líneas, debemos ordenarlo mediante el método de merge, ya que hemos realizado este ordenamiento realizamos la ejecución del ciclo que se encargara de analizar las líneas, tendremos un ciclo for que se ejecutara hasta que se haya acabado de analizar todas las líneas y se haya alcanzado a cubrir la línea principal. Primero, lo que haremos es verificar que la línea analizarse válida para ello. Esta línea tiene que empezar antes o en la posición en la que estamos, o bien si estamos en cero, que es el inicio, pero si la

Greedy - MinimalCoverage.c

```c
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;
}
```

línea empieza a partir de 5, esta línea no será válida. Pero usted la línea empieza en menos 10 y estamos en cero. Esta línea contará como válida.

Ya que tenemos una línea pálida, lo que haremos será meter en nuestras reglas de líneas resultados donde tendremos todas las líneas que usaremos para cubrir nuestra línea principal. De esta forma sumaremos uno al número del resultado, el cual nos indica el número total de líneas de las cuales tendremos que disponer.

Nuestra variable de avance tendrá como límite el tamaño máximo de nuestra línea, es decir, si tenemos una línea que va desde menos 2 hasta 3, nuestro avance tendrá que ser igual a 3.

Continuando con nuestro análisis y conforme vamos viendo el código, ahora nos encontramos con un ciclo anidado que se ejecutará al menos 1 máximo o igual al número de líneas que se analizarán. Esto no es cuadrático el algoritmo, ya que este ciclo busca una mejor línea, es decir, si tenemos una línea más cercana al origen y que habrá más 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 que sea mayor que el avance actual. Usaremos una

Y la guardaremos en la posición correspondiente que reemplazará a la línea analizada anteriormente debido a que no ha cumplido con las. Condiciones propuestas. Una vez que hayamos escaneado y encontrado la línea, mejor lo que haremos es actualizar nuestra posición actual, la cual será igual al avance y seguiremos analizando las líneas siguientes.

Greedy - MinimalCoverage.c

```
1   if (posActual < tamLinea)
2   {
3       printf("0\n");
4   }
5   else
6   {
7       printf("%d\n", resultado);
8       for (i = 0; i < resultado; i++)
9           printf("%d %d\n", lineasResultado[i][0], lineasResultado[i][1]);
10  }
11  if (numCasos)
12      printf("\n");
13 }
14 return 0;
```

Ya que ya hemos analizado todas nuestras líneas, lo que haremos será comprobar si hemos cubierto toda la línea principal en caso de que no. Nuestro algoritmo imprimirá un cero que significa que no hay una solución válida con respecto al número de líneas y las mediciones que se tienen para poder cubrir la línea principal en caso contrario. Nos indicará cuántas líneas fueron necesarias para realizar la cobertura principal.

Análisis en Cota O

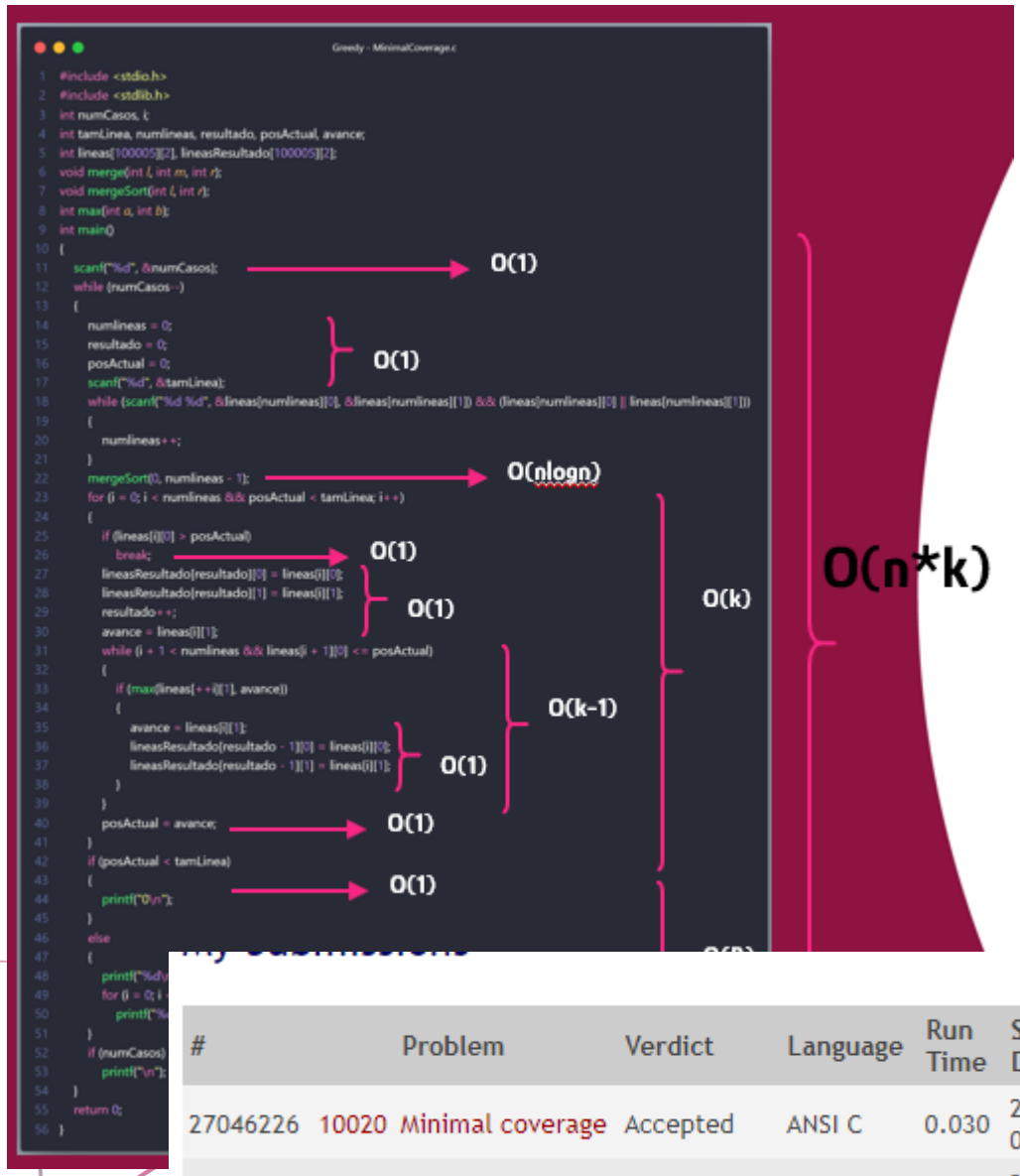
Este algoritmo en particular, tiene muchos ciclos anidados, por lo cual podría resultar de cierta forma engañoso al momento de realizar sus análisis, pero tenemos que notar que tiene ciertas peculiaridades. Por ejemplo, nuestro primer ciclo anidado (while) que escanea los valores para nuestras líneas que es de orden. K igual al número de las líneas que sean ingresadas, nuestro siguiente ciclo es el for, que es de orden k, dentro podemos observar otro, pero este no es de orden K, ya que solo se realizar una vez el recorrido de la lista de líneas.

El ciclo posterior que ya no se encuentra anidad es de orden r, el cual es el orden de la línea que vamos a utilizar, en el peor caso que se puede obtener es que el resultado sea igual a la cantidad de líneas ingresadas.

Por último tenemos el ciclo principal, el cual se ejecutará n veces, ósea la cantidad de veces que se quiera someter al algoritmo a análisis (cantidad de casos)

De igual forma debemos tomar en cuenta la complejidad de nuestro algoritmo de ordenamiento, pues este juega un papel importante.

Cota final O(n*(k*log(k)))



#	Problem	Verdict	Language	Run Time	Submission Date
27046226	10020 Minimal coverage	Accepted	ANSI C	0.030	2021-12-12 04:19:26

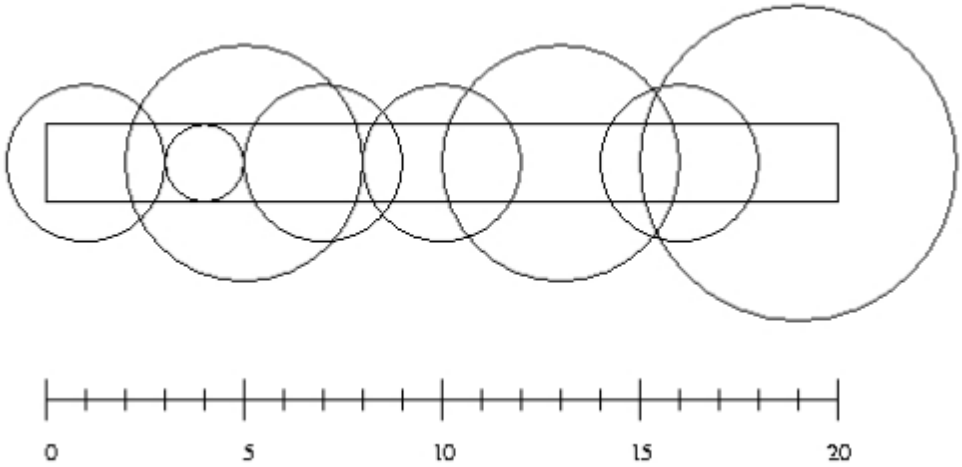
ALGORITMO 4

DESCRIPCIÓN DEL PROBLEMA

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 ≤ 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

Comenzamos con la declaración y previa inicialización de las variables que debemos ocupar para poder disponer de ellas como debe de ser, de igual forma tendremos un arreglo bidimensional, donde todas las primeras posiciones serán los valores de los lugares en los cuales colocaremos los aspersores, mientras que la otra parte de nuestro arreglo corresponde al valor del radio de nuestro aspersor.

De igual forma tendremos el numero de aspersores, el largo y ancho de nuestro campo además de las posiciones y por ultimo el radio de nuestro aspersor

De igual forma nos encontramos con otras dos variables que nos ayudaran para la obtención de las distancias que obtendremos gracias a la utilización del teorema de Pitágoras, además de nuestras variables auxiliar y avance las cuales nos ayudan a determinar si las posiciones donde nos encontramos son validas y podemos proceder a usar el aspersor que se esté sometiendo a análisis.

Nuestro ciclo principal consiste en que mientras haya casos que analizar con respecto a los aspersores, en donde realizamos la obtención del numero de aspersor, largo y ancho, este será ejecutado mientras haya valores que deban ser analizados

Greedy - wateringGrass.c

```c
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;
      }
   }
```

Una vez que ya tenemos las primeras variables procedemos a escanear las posiciones y los radios pertinentes de cada aspersor, esto gracias a la ayuda del teorema de Pitágoras. Dado que resulta algo complejo poder trabajar con círculos dentro de un terreno con forma cuadrática, nos auxiliaremos de la obtención de rectángulos inscritos, pero para saber la distancia "dx", la cual será la distancia que ocuparemos en nuestro programa, debemos obtener primero el cateto adyacente teniendo como hipotenusa el radio de nuestro circulo 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 fórmula para obtener la medida de este cateto la formula despejada quedaría de la siguiente manera.

Greedy - wateringGrass.c

```
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;
    }
}
```

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

Ya que hemos obtenido esta distancia podemos obtener la distancia que nuestro circulo estará ocupando dada la posición en la cual se encuentra. En nuestra primera posición "0" de nuestro arreglo tenemos la **ubicación del aspersor menos la distancia "dx"** y en la posición numero 1 **tenemos la posición más la distancia "dx",** ósea que tenemos la distancia del largo del rectángulo inscrito en la circunferencia.

Ya que tenemos todas las distancias calculadas procederemos a utilizar merge para poder realizar el ordenamiento pertinente de las mimas con respecto a **posición menos distancia obtenida**

Ahora tenemos un ciclo que se ejecutara desde 0 hasta que hayamos alcanzado el máximo de aspersores de los cuales disponemos, de igual forma es ejecutado mientras no se haya excedido el tamaño total del largo de nuestro terreno.

Debemos considerar que si el aspersor desde su posición menos la distancia es mayor que nuestro auxiliar la posición no será válida, es decir, para que una posición sea válida tendrá que acabar desde la parte del área del rectángulo anterior.

Si empezamos de 1-4 el siguiente debe ser de 4-x y así sucesivamente.

Si el aspersor es valido simplemente realizaremos un incremento a nuestra variable de resultado, y mientras que a nuestra variable de avance le daremos el valor de la posición mas la distancia del aspersor que estamos analizando.

Aquí analizamos que aspersor es mas grande en el que nos encontramos o el siguiente, terminado este ciclo a nuestra variable auxiliar la actualizaremos a el avance.

Una vez analizados todos los aspersores realizaremos la comprobación con respecto a auxiliar, si esta logro superar el tamaño del campo imprimiremos el total de aspersores, en caso contrario imprimiremos -1

Análisis en Cota O

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 número 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 dos líneas de código consisten en igualaciones y asignaciones las cuales son constantes, además de que tenemos la llamada de nuestro algoritmo de ordenamiento que sabemos que es de orden **nlogn** 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 de greedy.

Posterior a esto tendemos 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 nos encontramos con otro ciclo while pero este ciclo anidado no hace que el algoritmo se llegue a cuadrático, debido a que cuando estamos analizando el aspersor mas grande, continuaremos con 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, encontrando los aspersores ams grandes y partiendo a partir de ese.

Por lo tanto este segundo ciclo for es de orden lineal por lo cual en el peor caso se ejecutara una k cantidad de veces

Ya que tenemos nuestro análisis y por regla de la multiplicación tendremos una cota de:

Cota Obtenida O(n*k)

Considerando la complejidad de merge:

O(n*(k*log(k)))

My Submissions

#		Problem	Verdict	Language	Run Time	Submission Date
27046191	10382	Watering Grass	Accepted	ANSI C	0.000	2021-12-12 04:07:00


Greedy - wateringGrass.c
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;
O(1)
while (scanf("%d %d %d", &numAsper, &largo, &ancho) != EOF)
{
for (i = 0; i < numAsper; i++)
{
scanf("%d %d", &posicion, &radio);
O(1)
O(k)
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;
O(1)
}
}
mergeSort(0, numAsper - 1);
O(klogn)
aux = 0;
resultado = 0;
for (i = 0; i < numAsper && aux < largo; i++)
{
if (aspersores[i][0] > aux)
break;
resultado++;
O(1)
O(k)
avance = aspersores[i][1];
while (i + 1 < numAsper && aspersores[i + 1][0] <= aux)
{
avance = max(aspersores[++i][1], avance);
}
O(k-1)
aux = avance;
O(1)
}
if (aux < largo)
resultado = -1;
printf("%d\n", resultado);
for (i = 0; i < numAsper; i++)
{
aspersores[i][0] = 0;
aspersores[i][1] = 0;
O(1)
O(K)
}
}
return 0;
}
O(n*k)

ANEXO CÓDIGOS

CÓDIGO SCARECROW

```c
#include <stdio.h>
#include <string.h>
#define MAX 1000000

int main()
{
    int t, n, m, i, j, c,count=0;
    char a[MAX];
    scanf("%d", &t);
    for (m = 1; m <= t; m++)
    {
        count = 0;
        scanf("%d", &c);
            scanf("%s", a);
        for (i = 0; i < c; i++)
        {
            if (a[i] == '.')
            {
                i += 2;
                count++;
            }
        }
        printf("Case %d: %d\n", m, count);
    }
    return 0;
}
```

Código Bear and Row

```c
#define MAX 1000000
#include <stdio.h>
#include <string.h>


int main()
{
    int casos,
    // Variables de ciclos
    i,j;
    long long seg,y,tam,
    // Celda vacia
    vacio,soldado;
    // Cadena de caracteres recibida
    char a[MAX];
    scanf("%d", &casos);

    while(casos--){
        scanf("%s", a);
        // Longitud de la cadena
        tam= strlen(a);
        // Reiniciando las variables para las distintas partidas
        seg=vacio=y=soldado = 0;
        for(i=tam-1;i>=0;--i){
            if(a[i]=='0'){
                vacio++;
            }
            else if(a[i]=='1' && vacio>0){
                y= vacio+y+1;
                seg+= y;
                vacio=0;
            }
            else if(a[i]=='1' && vacio==0){
                seg+=y;
            }
        }
        printf("%lld\n",seg);
    }

    return 0;
}
```

Codigo Minimal 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;
}
```

Código 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

[1] E. A. F. Martinez, «EaFranco.Eakdemy,» [En línea]. Available: https://eafranco.eakdemy.com/mod/scorm/player.php. [Último acceso: 07 Octubre 2021].

[2] «Online Judge,» [En línea]. Available: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=657&page=show_problem&problem=3836. [Último acceso: 10 Diciemrbe 2021].

[3] ROSWOLD, «CodeChef,» CodeChef, [En línea]. Available: https://www.codechef.com/problems/ROWSOLD. [Último acceso: 10 Diciembre 2021].

[4] «Online Judge,» UvaHunting, [En línea]. Available: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=657&page=show_problem&problem=3836. [Último acceso: 10 Diciembre 2021].