

PRÁCTICA 5: PROBLEMA DEL MÁXIMO SUBARREGLO.

Hernández Castellanos César Uriel, Aguilar Garcia Mauricio

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
uuriel12009u@gmail.com, mauricio.aguilar.garcia,90@gmail.com

Resumen: En la practica calculamos analítica y experimentalmente la complejidad de los algoritmos de maximo subarreglo cruzado y del maximo subarreglo comparandolos con el algoritmo de fuerza bruta para resolver el mismo problema.

Palabras Clave: Algoritmo, Complejidad, Máximo, Subarreglo.

1. Introducción

En la cultura popular, divide y vencerás hace referencia a un refrán que implica resolver un problema difícil, dividiéndolo en partes más simples tantas veces como sea necesario, hasta que la resolución de las partes se torna obvia. La solución del problema principal se construye con las soluciones encontradas.

En las ciencias de la computación, el término divide y vencerás (DYV) hace referencia a uno de los más importantes paradigmas de diseño algorítmico. El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original.

Esta técnica es la base de los algoritmos eficientes para casi cualquier tipo de problema como, por ejemplo, algoritmos de ordenamiento (quicksort, mergesort, entre muchos otros), multiplicar números grandes (Karatsuba), análisis sintácticos (análisis sintáctico top-down) y la transformada discreta de Fourier.

Por otra parte, analizar y diseñar algoritmos de DyV son tareas que lleva tiempo dominar. Al igual que en la inducción, a veces es necesario sustituir el problema original por uno más complejo para conseguir realizar la recursión, y no hay un método sistemático de generalización.

El nombre divide y vencerás también se aplica a veces a algoritmos que reducen cada problema a un único subproblema, como la búsqueda binaria para encontrar un elemento en una lista ordenada (o su equivalente en computación numérica, el algoritmo de bisección para búsqueda de raíces). Estos algoritmos pueden ser implementados más eficientemente que los algoritmos generales de “divide y vencerás”; en particular, si es usando una serie de recursiones que lo convierten en simples bucles. Bajo esta amplia definición, sin embargo, cada algoritmo que usa recursión o bucles puede ser tomado como un algoritmo de “divide

y vencerás”. El nombre decrements y vencerás ha sido propuesta para la subclase simple de problemas.

La corrección de un algoritmo de “divide y vencerás”, está habitualmente probada una inducción matemática, y su coste computacional se determina resolviendo relaciones de recurrencia.[?]

2. Conceptos básicos

2.1. Cota ajustada asintótica

En análisis de algoritmos una cota ajustada asintótica es una función que sirve de cota tanto superior como inferior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación $\theta(g(x))$ para referirse a las funciones acotadas por la función $g(x)$. [0]

2.2. Cota inferior asintótica

En análisis de algoritmos una cota inferior asintótica es una función que sirve de cota inferior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación $\Omega(g(x))$ para referirse a las funciones acotadas inferiormente por la función $g(x)$. [0]

2.3. Cota superior asintótica

En análisis de algoritmos una cota superior asintótica es una función que sirve de cota superior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación de Landau: $O(g(x))$, Orden de $g(x)$, coloquialmente llamada Notación O Grande, para referirse a las funciones acotadas superiormente por la función $g(x)$. [0]

2.4. Máximo Subarreglo

El problema del subvector de suma máxima consiste en encontrar un subvector de una determinada longitud m cuya suma sea máxima dentro de un vector de longitud n , con $m \leq n$.

Este problema se puede resolver aplicando la técnica del algoritmo divide y vencerás, formando problemas cada vez más pequeños hasta alcanzar un caso base y posteriormente combinando las soluciones obtenidas. En concreto, la forma de aplicar el método algorítmico citado consiste en obtener los segmentos de suma máxima correspondientes a las mitades izquierda y derecha del vector y a la parte central para, una vez calculados, elegir el máximo de los tres. El algoritmo tiene un coste lineal respecto al tiempo.

3. Experimentación y Resultados

i) Mediante gráficas, muestre que el algoritmo del máximo subarreglo cruzado tiene complejidad lineal.

```
def obtenerMaximoSubArregloCruzado(array,bajo,medio,alto):  
    suma_izq = max_izq = suma = 0  
    for x in range(medio,bajo-1,-1):  
        suma = suma + array[x]  
        if(suma > suma_izq):  
            suma_izq = suma  
            max_izq = x  
    suma_der = max_der = suma = 0  
    for y in range(medio+1,alto+1):  
        suma = suma + array[y]  
        if(suma >= suma_der):  
            suma_der = suma  
            max_der = y  
    return suma_der+suma_izq
```

Figura 1: Algoritmo implementado que encuentra el máximo subarreglo

En la figura 2 se muestra la gráfica de la complejidad del algoritmo del máximo subarreglo(roja) y además se muestra la función que acota por arriba a dicha función (azul)

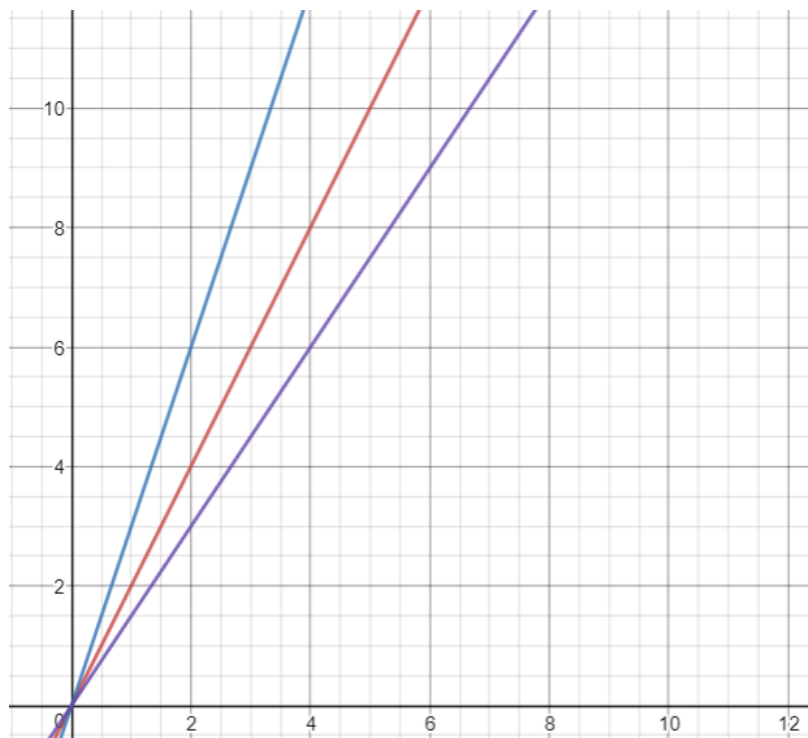


Figura 2: Gráfica de la complejidad del algoritmo de máximo subarreglo

Las funciones que se muestran en la figura 2 son $f1(n) = 3n$, $f2(n) = 2n$ y $f1(n) = 1.5n$

En la imagen que se muestra a continuación se muestran las coordenadas que nos arrojó el algoritmo (n vs k), donde n es el tamaño del arreglo y k es el número de pasos.

| N | K |
|----|----|
| 0 | 0 |
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 4 | 8 |
| 5 | 10 |
| 6 | 12 |
| 7 | 14 |
| 8 | 16 |
| 9 | 18 |
| 10 | 20 |

Figura 3: Gráfica de la complejidad del algoritmo de máximo subarreglo

II) Demuestre analíticamente que el algoritmo del máximo subarreglo cruzado tiene complejidad lineal.

Encontrando las sumatorias del algoritmo de máximo subarreglo cruzado:

$$T(n) = C_1 + C_2(n + 2) + (C_3 + C_4)(n + 1)$$

$$T(n) = (C_2 + C_3 + C_4)n + C_1 + 2C_2 + C_3 + C_4$$

$$T(n) = 3n + 5 \therefore T(n) \in \theta(n)$$

III) Mediante gráficas, muestre que el algoritmo del máximo subarreglo tiene complejidad $n \log(n)$.

```
def obtenerMaximoSubArreglo(array,bajo,alto):
    if(bajo == alto):
        return array[bajo]
    else:
        medio = (bajo + alto)/2
        return max(obtenerMaximoSubArreglo(array,bajo,medio),
                   obtenerMaximoSubArreglo(array,medio+1,alto),
                   obtenerMaximoSubArregloCruzado(array,bajo,medio,alto))
```

Figura 4: Algoritmo implementado en Python que encuentra el máximo subarreglo

En la figura siguiente se muestra la gráfica de complejidad del algoritmo anteriormente descrito (roja), además de las funciones que la acotan por la parte superior e inferior (azul y verde)

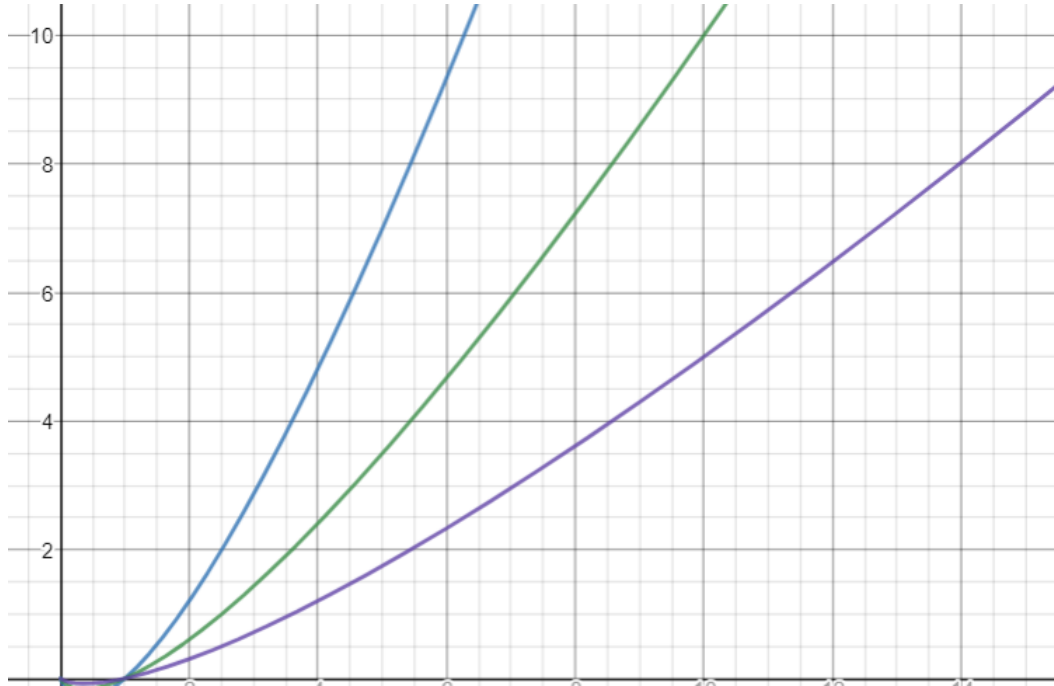


Figura 5: Algoritmo implementado que encuentra el máximo subarreglo

IV) Demuestre analíticamente que el algoritmo del máximo subarreglo tiene complejidad

$n \log(n)$.

$$T(n) \begin{cases} 1; & \text{bajo} \geq \text{alto} \Rightarrow n == 1 \\ 2T\left(\frac{n}{2}\right) + 4n + 7; & \text{alto} > \text{bajo} \Rightarrow n > 1 \end{cases}$$

$$n = 2^k, k = \log_2 n$$

$$T(2^k)$$

$$2T(2^{k-1}) + 4(2^k) + 7$$

$$2[2T(2^{k-2}) + 4(2^{k-1}) + 7] + 4(2^k) + 7$$

$$2^2 T(2^{k-2}) + 2[4(2^k)] + 3(7)$$

$$2^2 [2T(2^{k-3}) + 4(2^{k-2}) + 7] + 2[4(2^k)] + 3(7)$$

$$2^3 T(2^{k-3}) + 3[4(2^k)] + 7(7)$$

$$2^3 [2T(2^{k-4}) + 4(2^{k-3}) + 7] + 3[4(2^k)] + 7(7)$$

$$2^4 T(2^{k-4}) + 4[4(2^k)] + 15(7)$$

$$2^{(i+1)} T(2^{k-(i+1)}) + (i+1)[4(2^k)] + 7[2^{i+1} - 1]$$

$$k - (i+1) = 0$$

$$i = k - 1$$

$$2^k T(2^0) + (k)[4(2^k)] + 7[2^k - 1]$$

$$2^k + (k)[2(2^k)] + 7[2^k - 1]$$

$$2^{\log_2 n} + (\log_2 n)[4(2^{\log_2 n})] + 7[2^{\log_2 n} - 1]$$

$$n + (\log_2 n)[4n] + 7[n - 1]$$

$$T(n) = n[8 + 4 \log_2 n] - 7$$

$$T(n) = \theta(n \log n)$$

V) Implementar un algoritmo que resuelva el problema del máximo subarreglo utilizando fuerza bruta. Calcule su orden de complejidad analítica y experimentalmente.

Algoritmo del máximo subarreglo por fuerza bruta

En la Figura uno se muestran las coordenadas obtenidas experimentalmente del algoritmo de máximo subarreglo por fuerza bruta.

| N | K |
|----|-----|
| 1 | 10 |
| 2 | 18 |
| 3 | 30 |
| 4 | 42 |
| 5 | 62 |
| 6 | 86 |
| 7 | 134 |
| 8 | 146 |
| 9 | 182 |
| 10 | 222 |

Figura 6: Tabla de los valores graficados

En la figura 2 se muestra la gráfica que se obtuvo a partir de los puntos de la figura anterior, es posible observar que el algoritmo para la obtención del máximo subarreglo por fuerza bruta tiene complejidad cuadrática.

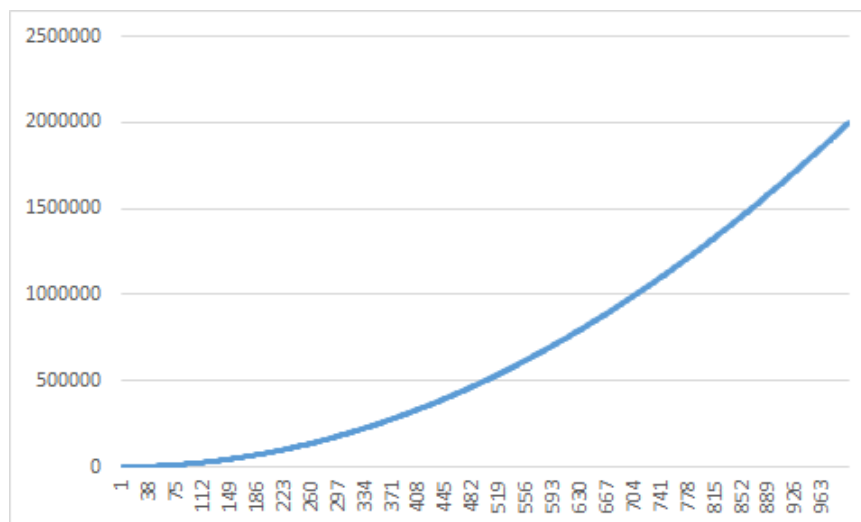


Figura 7: Gráfica de la complejidad del algoritmo del máximo subarreglo por fuerza bruta

En la figura 3 que se muestra a continuación, es posible observar la función que acota por arriba a nuestra función cuadrática.

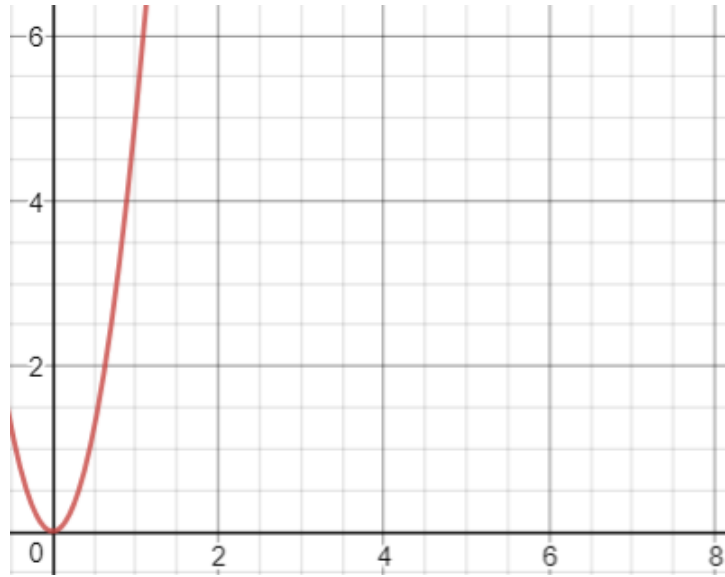


Figura 8: Función que acota por arriba a la función cuadrática $f(n) = 5x^2$

En la figura 4 que se muestra a continuación, es posible observar la función que acota por abajo a nuestra función cuadrática.

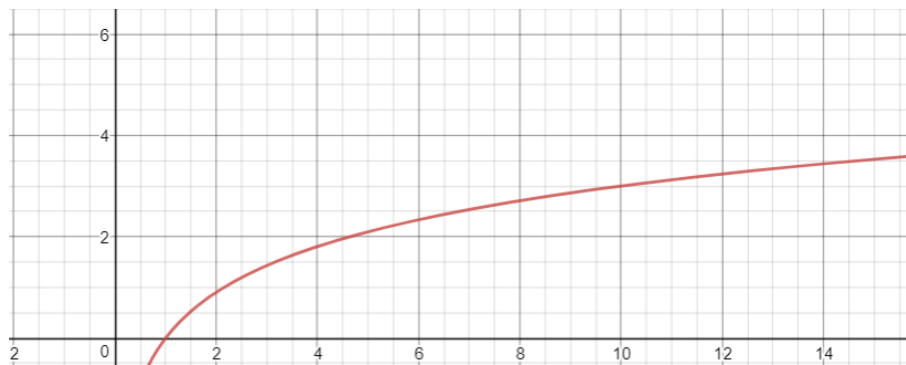


Figura 9: Función que acota por abajo a la función cuadrática $f(n) = 3 \log(n)$

En la figura 9 se muestra la implementación del algoritmo del máximo subarreglo por fuerza bruta, se ha usado el lenguaje de programación Java, para esto se creó una clase con el nombre de MaximoSubArreglo, donde se declararon los atributos sumamax,suma,n,maxizq y maxder, todos estos de tipo entero, además de la implementación del algoritmo mismo.


```

1  package pr05;
2
3  import java.util.Arrays;
4  import java.util.LinkedList;
5
6  public class MaximoSubArreglo {
7
8      private int suma_max, suma;
9      private int n, max_izq, max_der;
10
11     public MaximoSubArreglo() {
12         this.suma_max = Integer.MIN_VALUE;
13         this.suma = 0;
14     }
15
16     public LinkedList obtenerMaximoSubArreglo(int[] arreglo) {
17         int k = 0;
18         n = arreglo.length;
19         k++;
20         for (int i = 0; i < n; i++) {
21             k++;
22             suma = 0;
23             k++;
24             for (int j = 0; j < n; j++) {
25                 k++;
26                 suma += arreglo[j];
27                 k++;
28                 if (suma > suma_max) {
29                     k++;
30                     suma_max = suma;
31                     k++;
32                     max_izq = i;
33                     k++;
34                     max_der = j;
35                     k++;
36                 }
37             }
38         }
39         k++;
40         return new LinkedList(Arrays.asList(max_izq, max_der, suma_max, k));
41     }
42 }

```

Figura 10: Código de implementación en Java del algoritmo del máximo subarreglo por fuerza bruta

En la Figura 10 se puede observar el calculo de la complejidad por bloques del algoritmo.

| Algorithm 1 Algoritmo para obtener el máximo subarreglo (Fuerza bruta) | | |
|--|------------------|--|
| 1: procedure OBTENERMAXIMOSUBARREGLO(<i>Arreglo</i>) | | |
| 2: $n \leftarrow \text{arreglo.lenght}()$ | $\}$ $\Theta(1)$ | |
| 3: $\text{carry} \leftarrow \text{false}$ | | |
| 4: for $i \leftarrow 0, \text{nbits}$ do | | |
| 5: $\text{suma} \leftarrow 0$ | | |
| 6: for $i \leftarrow 0, \text{nbits}$ do | | |
| 7: $\text{suma} \leftarrow \text{arreglo}[j] + \text{suma}$ | | |
| 8: if $\text{suma} > \text{sumamax}$ then | | |
| 9: $\text{sumamax} \leftarrow \text{suma}$ | | |
| 10: $\text{maxizq} \leftarrow i$ | | |
| 11: $\text{maxder} \leftarrow j$ | | |
| 12: end if | | |
| 13: end for | | |
| 14: end for | | |
| 15: end procedure | | |

Figura 11: Calculo de la complejidad del algoritmo del máximo subarreglo por fuerza bruta

4. Conclusiones

Conclusión general

En esta práctica pudimos verificar que hay algoritmos que en su versión recursiva además de proporcionarnos la información que queremos, nos puede dar información adicional gracias al backtracking, como en este caso que en las versiones iterativas se puede encontrar la máxima suma, en cambio, en la recursiva además se obtiene donde inicia y termina el subarreglo y dependiendo de la aplicación en la que lo vayamos a utilizar, podemos seleccionar el mejor algoritmo.

Aguilar Garcia Mauricio

En ésta práctica pudimos ver como se resuelve el problema del máximo subarreglo y como al emplear el paradigma divide y vencerás se puede mejorar enormemente la complejidad algorítmica con diferencia a su contra parte de la fuerza bruta. El algoritmo máximo subarreglo, implementa el paradigma Divide y Vencerás, y la salida depende de que valor de sumatoria sea maximo, ya sea la llamada recursiva para la primer mitad, la segunda o la llamada a maximo subarreglo cruzado. Aparte de que al analizar el caso donde todos los elementos son negativos (análisis en el Anexo), nos ayudo a comprender completamente como funciona el algoritmo.

Hernández Castellanos César Uriel

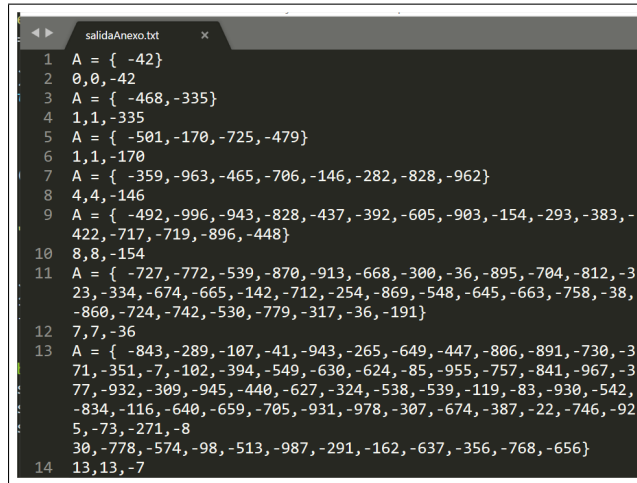
El problema del subarreglo máximo consiste en encontrar un subarreglo de una cierta longitud m cuya suma sea máxima dentro de un arreglo de tamaño n , donde $m \leq n$

En la presente práctica se presentaron tres formas diferentes de resolver la problemática del máximo subarreglo, en primera instancia se implemento un algoritmo con complejidad lineal, el siguiente logaritmica y por último cuadrática, teniendo peor desempeño el algoritmo por fuerza bruta y su contrario siendo la forma recursiva.

5. Anexo

¿Qué retorna la función de máximo subarreglo cuando todos los valores del arreglo son valores enteros negativos?. La función $MS(A, \text{bajo}, \text{alto})$ retorna la posición del mayor elemento en el arreglo, es decir el más cercano a cero.

Esto se puede corroborar con la siguiente figura, en la cual se aprecia la entrada, A , y la salida, índice inferior, índice superior y la suma, que en este caso la suma es igual a el mayor elemento del arreglo.



```
1 A = { -42}
2 0,0,-42
3 A = { -468,-335}
4 1,1,-335
5 A = { -501,-170,-725,-479}
6 1,1,-170
7 A = { -359,-963,-465,-706,-146,-282,-828,-962}
8 4,4,-146
9 A = { -492,-996,-943,-828,-437,-392,-605,-903,-154,-293,-383,-
10 422,-717,-719,-896,-448}
11 8,8,-154
12 A = { -727,-772,-539,-870,-913,-668,-300,-36,-895,-704,-812,-3
13 23,-334,-674,-665,-142,-712,-254,-869,-548,-645,-663,-758,-38,
-860,-724,-742,-530,-779,-317,-36,-191}
14 7,7,-36
15 A = { -843,-289,-107,-41,-943,-265,-649,-447,-806,-891,-730,-3
16 71,-351,-7,-102,-394,-549,-630,-624,-85,-955,-757,-841,-967,-3
17 77,-932,-309,-945,-440,-627,-324,-538,-539,-119,-83,-930,-542,
-834,-116,-640,-659,-705,-931,-978,-307,-674,-387,-22,-746,-92
18 5,-73,-271,-8
19 30,-778,-574,-98,-513,-987,-291,-162,-637,-356,-768,-656}
20 13,13,-7
```

Figura 12: Salida del programa, dando como entrada únicamente números negativos.

6. Bibliografía

Introduction to Algorithms, Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein