Análisis de Algoritmos, Sem: 2018-2, 3CV1, Práctica 5, 4 de Octubre de 2018

Práctica 5: Problema del Máximo Subarreglo

Reyes Valenzuela Alejandro, Gutiérrez Povedano Pablo.

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
*areyesv*11@*gmail.com*, *gpovedanop*@*gmail.com*

Resumen: En este reporte se aboradará el algoritmo del máximo subarreglo para determinar el orden de complejidad mediante el paradigma de divide y vencerás, tanto del algoritmo de subarreglo cruzado como el del subarreglo en sí. **Palabras Clave:** Divide, Vence, Combina, Subarreglo.

1 Introducción

En la práctica anterior utilizamos el paradigma de divide y vencerás para analizar la complejidad del algortimo Quicksort, ahora reutilizaremos el paradigma para analizar el algoritmo del máximo subarreglo, el cuál ya no es un arreglo de ordenamiento, pero que también se apoya de una función auxiliar para funcionar, por lo que calcularemos la complejidad de ambos tanto analíticamente como gráficamente. Adicionalmente analizaremos una implementación con realizada con fuerza bruta, en donde veremos que el orden de complejidad es mayor con respecto a los otros dos algoritmos.

2 Conceptos Básicos

El paradigma de Divide y Vencerás separa un problema en subproblemas que se parecen al problema original, de manera recursiva resuelve los subproblemas y, por último, combina las soluciones de los subproblemas para resolver el problema original.

Como divide y vencers resuelve subproblemas de manera recursiva, cada subproblema debe ser más pequeo que el problema original, y debe haber un caso base para los subproblemas.

Para ello recurrimos a tres pasos fundamentales:

- **Divide:** Dividir el problema en un número de subproblemas que son instancias más pequeas del mismo problema.
- **Vence:** Resolver los subproblemas de manera recursiva. Si son los suficientemente pequeños, resolver los subproblemas como casos base.
- **Combina:** Juntar las soluciones de los subproblemas en la solución para el problema original.

Haremos uso del paradigma anterior para encontrar el orden de complejidad del algoritmo del máximo subarreglo cruzado y del máximo subarreglo. Dichos algoritmos nos sirven para encontrar la suma máxima que puede encontrarse en un arreglo de enteros, el cruzado toma en cuenta que pase por la mitad mientras que el general busca en ambos lados del mismo.

Algoritmo Maximo subarreglo Cruzado.

$$
\begin{aligned}
&MSC(A[bajo, ..., medio, ..., alto], alto, medio, bajo)\\
&\quad suma\_izq = -\infty\\
&\quad suma = max\_der = max\_izq = 0\\
&\quad for(i = medio; i \leq bajo + 1; i - -)\\
&\quad\quad suma = suma + A[i]\\
&\quad\quad if suma \geq suma\_izq\\
&\quad\quad\quad max\_izq = i\\
&\quad suma\_der = -\infty\\
&\quad suma = 0\\
&\quad for(i = medio + 1; i \leq alto + 1; i + +)\\
&\quad\quad suma = suma + A[i]\\
&\quad\quad if suma \geq suma\_der\\
&\quad\quad\quad max\_der = i\\
&\quad return(max\_izq, max\_der, suma\_der + suma\_izq)
\end{aligned}
$$

Algoritmo Maximo Subarreglo

$MS(A[bajo, ..., alto], alto, bajo)$
$if\ alto = bajo$
$return(bajo, alto, a[bajo])$
$else$
$mitad = (alto + bajo)/2$
$(bajo\_izq, alto\_izq, suma\_izq) = MS(A, bajo, mitad)$
$(bajo\_der, alto\_der, suma\_der) = MS(A, mitad + 1, alto)$
$(bajo\_cruz, alto\_cruz, suma\_cruz) = MSC(A, bajo, mitad, alto)$
$if\ (suma\_der \geq suma\_izq$ and $suma\_der \geq suma\_cruz)$
$return(bajo\_der, alto\_der, suma\_der)$
$else\ if(suma\_izq \geq suma\_der$ and $suma\_izq \geq suma\_cruz)$
$return(bajo\_izq, alto\_izq, suma\_izq)$
$else$
$return(bajo\_cruz, alto\_cruz, suma\_cruz)$

Implementación usando fuerza bruta:

$MSFB(A[bajo, .., alto], alto, bajo)$
$suma\_max = -\infty$
$max\_der = max\_izq = 0$
$for(i = 0; i \leq alto; i++)$
$suma = 0$
$for(j = i; j \leq alto; i++)$
$suma = suma + A[j]$
$if suma \geq suma\_max$
$suma\_max = suma$
$max\_izq = i$
$max\_der = j$
$return(max\_izq, max\_der, suma\_max)$

A continuación calcularemos el orden de complejidad de los tres algoritmos presentados anteriormente.

3 Experimentación y Resultados

Cálculo del orden de complejidad del algoritmo de máximo subarreglo cruzado:

$MSC(A[bajo, ..., medio, ..., alto], alto, medio, bajo)$	
$suma\_izq = -\infty$	$\Theta(1)$
$suma = max\_der = max\_izq = 0$	—
$for(i = medio; i \leq bajo + 1; i - -)$	$\Theta(mid - bajo + 2)$
$\quad suma = suma + A[i]$	$\Theta(1)$
$\quad if suma \geq suma\_izq$	—
$\quad\quad max\_izq = i$	$\Theta(1)$
$suma\_der = -\infty$	—
$suma = 0$	—
$for(i = medio + 1; i \leq alto + 1; i + +)$	$\Theta(alto - bajo - 1 + 2)$
$\quad suma = suma + A[i]$	$\Theta(1)$
$\quad if suma \geq suma\_der$	—
$\quad\quad max\_der = i$	—
$return(max\_izq, max\_der, suma\_der + suma\_izq)$	—

Podemos observar que la complejidad radica en los ciclos for, los cuales tienen rangos asignados con base a la parte del arreglo donde buscarán, es por eso que se puede juntar la solución de la complejidad al sumar los elementos de dichas complejidades, lo cual nos queda como $alto - bajo - 3$. Sin embargo, $alto - bajo = n$, por eso la complejidad de este algoritmo es $\Theta(n)$.

Calculo de la complejidad del algoritmo del máximo subarreglo:

$MS(A[bajo, ..., alto], alto, bajo)$
$\quad if\ alto = bajo$
$\quad\quad return(bajo, alto, a[bajo])$ $\Theta(1)$
$\quad else$
$\quad\quad mitad = (alto + bajo)/2$
$\quad\quad (bajo\_izq, alto\_izq, suma\_izq) = MS(A, bajo, mitad)$ $T(\frac{n}{2})$
$\quad\quad (bajo\_der, alto\_der, suma\_der) = MS(A, mitad + 1, alto)$ $T(\frac{n}{2})$
$\quad\quad (bajo\_cruz, alto\_cruz, suma\_cruz) = MSC(A, bajo, mitad, alto)$ $\Theta(n)$
$\quad if\ (suma\_der \geq suma\_izq$ and $suma\_der \geq suma\_cruz)$
$\quad\quad return(bajo\_der, alto\_der, suma\_der)$ $\Theta(1)$
$\quad else\ if(suma\_izq \geq suma\_der$ and $suma\_izq \geq suma\_cruz)$
$\quad\quad return(bajo\_izq, alto\_izq, suma\_izq)$ $\Theta(1)$
$\quad else$
$\quad\quad return(bajo\_cruz, alto\_cruz, suma\_cruz)$ $\Theta(1)$

Podemos observar que existe recursividad en nuestra función, por lo que procederemos a calcular la ecuación de recurrencia.

$$T(n) = \begin{cases} \Theta(1) & si \quad n = 1 \\ \\ 2T(\frac{n}{2}) + \Theta(n) & si \quad n > 1 \end{cases}$$

Ahora resolveremos la recurrencia, como tenemos un cociente, debemos asegurarnos que nuestro resultado sea entero, para ello se propone el cambio de variable $n = 2^k$, donde $k = \log_2(n)$, por lo que ahora resolveremos:

$$T(n) = 2T(\tfrac{n}{2}) + C(n) = 2T(2^{k-1}) + C(2^k)$$

Haciendo iteraciones:

$$T(2^k) = 2(2T(2^{k-2}) + C(2^{k-1})) + C(2^k)$$

Llegando al término i:

$$T(2^k) = 2^i((T(2^{k-i}) + iC(2^k))$$

Llegando a la condición de frontera:

$$T(2^k) = 2^k((T(1) + kC(2^k))$$

Finalmente:

$$T(2^k) = c(n) + \log_2(n) + c(n)$$

Por lo que este algoritmo tiene complejidad $\Theta(n \log_{10}(n))$.

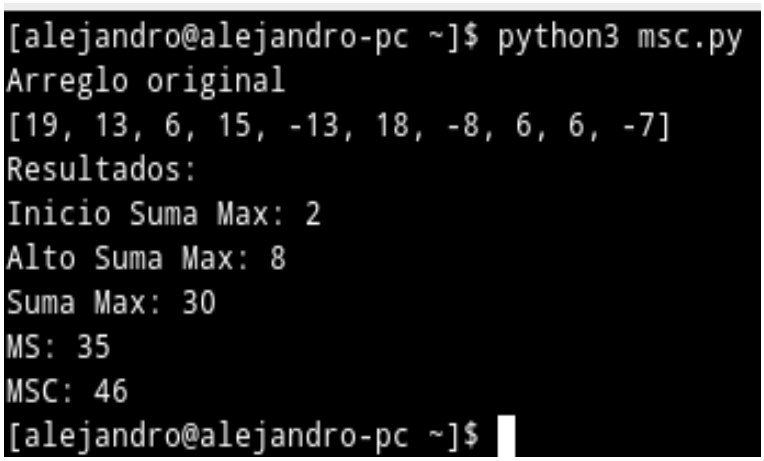
Calculo de complejidad del algoritmo por fuerza bruta:

$MSFB(A[bajo, .., alto], alto, bajo)$
$suma\_max = -\infty$
$max\_der = max\_izq = 0$
$for(i = 0; i \leq alto; i++)$
$suma = 0$
$for(j = i; j \leq alto; i++)$
$suma = suma + A[j]$
$if suma \geq suma\_max$
$suma\_max = suma$
$max\_izq = i$
$max\_der = j$
$return(max\_izq, max\_der, suma\_max)$

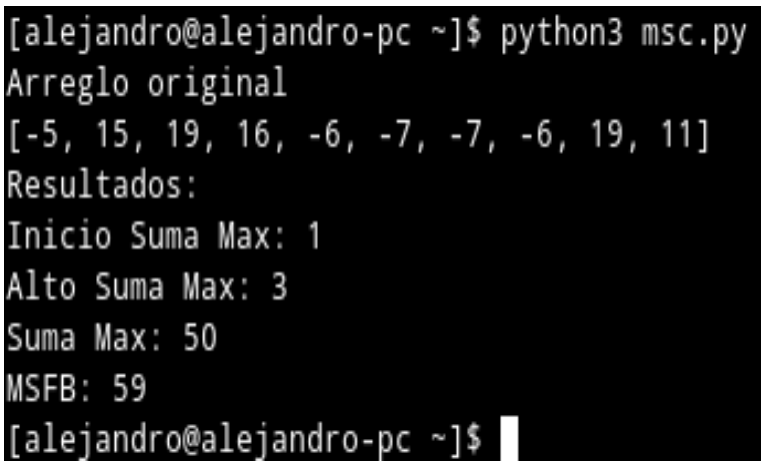
Podemos observar que tenemos dos for anidados que recorren al mismo tiempo el arreglo, por lo que puede deducirse fácilmente que la complejidad de este algoritmo es $\Theta(n^2)$.

Para el desarrollo del programa se manejaron arreglos llenados de manera aleatoria con enteros de rango de entre -20 a 20.

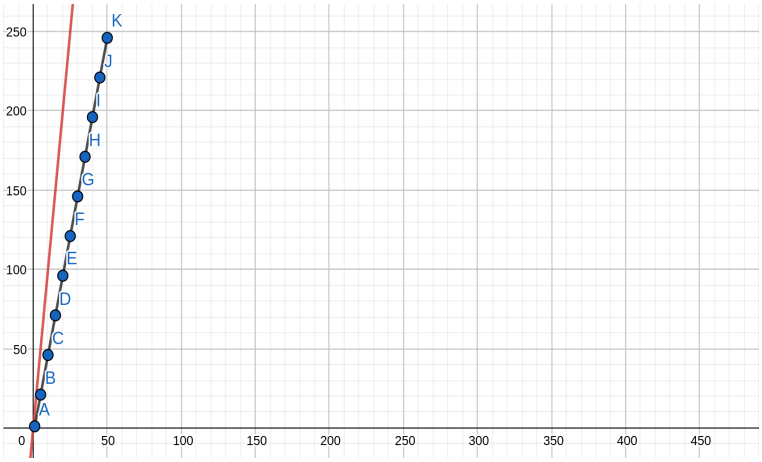
Captura de pantalla del programa original.



Captura de pantalla del programa con fuerza bruta.



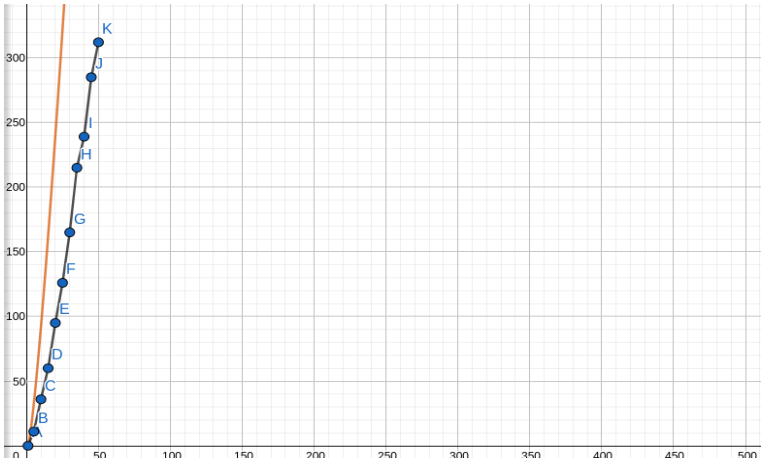
Gráfica de la función del máximo subarreglo, acotando con la función 10n (mostrada en color rojo).



Valores de función MSC:

Longitud	**Valor**
1	1
5	21
10	46
15	71
20	96
25	121
30	146
35	171
40	196
45	221
50	246

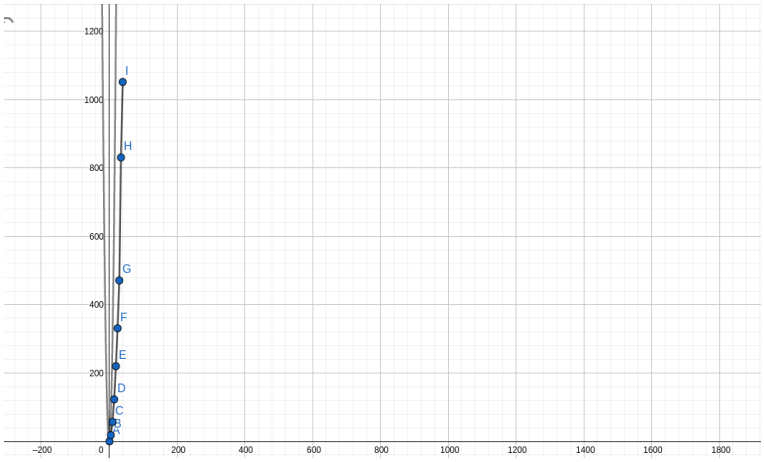
Gráfica de la función del máximo subarreglo, acotando con la función 4n*log(n) (mostrada en color naranja).



Valores de función MS:

Longitud	**Valor**
1	0
5	11
10	36
15	60
20	95
25	126
30	165
35	215
40	239
45	285
50	312

Gráfica de la función del máximo subarreglo implementado con fuerza bruta acotando con la función n^2 (mostrada en color gris).



Valores de función MSFB:

Longitud	**Valor**
1	0
5	18
10	57
15	123
20	220
25	331
30	471
35	831
40	1052

4 Conclusiones

Conclusión General: Pudimos ver que las implementaciones tuvieron distintos tiempos de ejecución, en el caso de la fuerza bruta a pesar de ser un sólo código el que se ejecutaba, al recorrer todo el arreglo hace que tenga mayor complejidad, en caso de las funciones recursivas, recorrian todo el arreglo pero mediante bloques, por lo que se reduce la comnplejidad considerablemente.

Conclusión Gutiérrez Povedano: Al final de la práctica puede constatar que un problema puede ser resuelto por un algoritmo y éste puede ser optimizado para reducir considerablemente su orden de complejidad mejorando así el tiempo que tarda en resolver el problema y reduciendo el número de instrucciones a ejecutadas, como en los algoritmos suma de sub arreglo máximo resuelto por fuerza bruta con orden de complejidad cuadrtica y el ptimo con complejidad lneal.

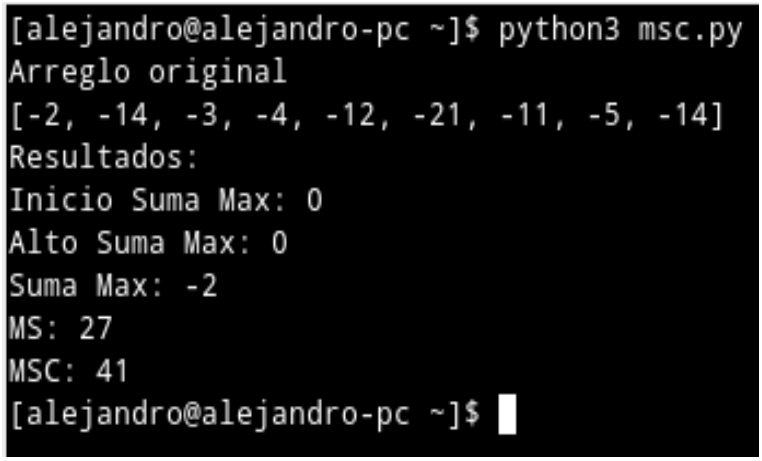
Conclusión Reyes Valenzuela: Pude notar que en este caso tuvimos que asociar más de un valor de retorno a las funciones. Esto nos permite conocer toda la información de la suma, como el inicio de la suma como el final. Además, es importante que todos los valores estén alternados con positivos y negativos, ya que en caso de ser todos positivos, la función devolverá el arreglo completo, mientras que en el anexo, se tratará el caso donde todos los valores son negativos.

5 Anexo

1. Qué retorna la función de máximo subarreglo cuando todos los valores del arreglo son valores enteros negativos?.

Al ser todos negativos, una simple suma hará que el valor de esta sea más negativa que la anterior, por lo que si se quiere encontrar una suma máxima, la función debe de regresar el valor del elemento más próximo a cero (el menor negativo).

Captura de pantalla de la ejecución del programa dado un arreglo con puros negativos. Se puede observar que nos devuelve como suma máxima el menor elemento del arreglo, en este caso el -2.



6 Bibliografía

Khanacademy.org (2018). [online] Available at: https://es.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms. [Accessed 29 Sep. 2018].

Slideshare.net (2018). [online] Available at: https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/. [Accessed 30 Sep. 2018].