

Análisis de Algoritmos, Sem: 2018-1, 3CV2 Práctica 2, 7 de  
Septiembre del 2017

# Práctica 3: Divide y Venceras: Algoritmo Merge Sort.

Luis Daniel Martinez Berumen



Escuela Superior de Computo  
Instituto Politecnico Nacional  
dany.berumen@gmail.com



## Abstract

En esta practica vamos a demostrar tanto de manera analitica como de manera practica, con graficas de los experimentos, la complejidad de dos algoritmos de ordenamiento, Merge y MergeSort, siendo el primero de complejidad lineal y el segundo de complejidad  $\theta(n \log(n))$ .

Valiendonos del paradigma de "Divide y venceras", el cual nos ayudara a simplificar el trabajo. Ademas de mostrar la resolucion de 2 problemas con propiedades de los algoritmos vistos en clase.

## Palabras Clave

- Algoritmo
- Fucion
- Recursividad
- Paradigma
- Dividir

## 1. Introducción

Divide y Vencerás es una frase que hemos escuchado todos, al menos, una vez en nuestra vida, para nosotros es técnica de diseño de algoritmos, siendo de gran utilidad para nuestra carrera, ya que, los problemas a los que nos enfrentamos día con día son mas faciles de resolver si aplciamos una tecnica de este tipo. De hecho, suele ser considerada una filosofía general para resolver problemas, no solo del termino informatico, sino que también se utiliza en muchos otros ámbitos

## 2. Conceptos Básicos

Para la correcta comprensión de este trabajo, es necesario definir algunos términos tales como  $\theta$ ,  $O$  y  $\Omega$ .

$\theta(n)$ :

Sea  $g(n)$  una función. Se define  $\theta(g(n))$  como:

$$\theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0 \ \& \ n_0 > 0 \mid \forall n \geq n_0 \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$O(n)$ :

Sea  $g(n)$  una función,  $O(n)$  (el peor de los casos) se define como:

$$O(n) = \{f(n) \mid \exists c > 0 \ \& \ n_0 > 0 \mid f(n) \leq C g(n) \ \forall n \geq n_0\}$$

$\Omega(n)$ :

Sea  $g(n)$  una función. Se define  $\Omega(g(n))$  (el mejor de los casos) como:

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0 \ \& \ n_0 > 0 \mid 0 \leq c g(n) \leq f(n) \ \forall n \geq n_0\}$$

El algoritmo de MergeSort es un método de ordenamiento por mezcla, la idea general es que: Dados dos conjuntos ordenados, A y B, si los mezclamos entre ambos, tomando los valores de ellos en orden, entonces nos devolverá el conjunto ordenado C, con los elementos de A y B. Partiendo

de este punto, aplicaremos la estrategia Divide y Vencerás. La idea es, que si inicialmente tenemos la lista desordenada, y la dividimos a la mitad, nos quedaremos con 2 sub-listas desordenadas, entonces, realizamos otra vez la misma acción: dividimos las sub-listas resultantes en 4 nuevas sub-listas, y así sucesivamente. Esta operación se realizará hasta que lleguemos a una sub-lista con

1 elemento en ella, que por defecto va a estar ordenada, y como dicha sub-lista ya está ordenada, la mezclamos con la de al lado, que está ordenada también, y así continuamente vamos ordenando las sub-listas hacia arriba para llegar al caso base.

Si representamos esas operaciones en una imagen, puedes notar que se forma un árbol, donde, partiendo de las hojas (son los elementos terminales del árbol), se van a ordenar los nodos padres, por medio de la mezcla entre sus dos hijos, hasta llegar al nodo raíz.

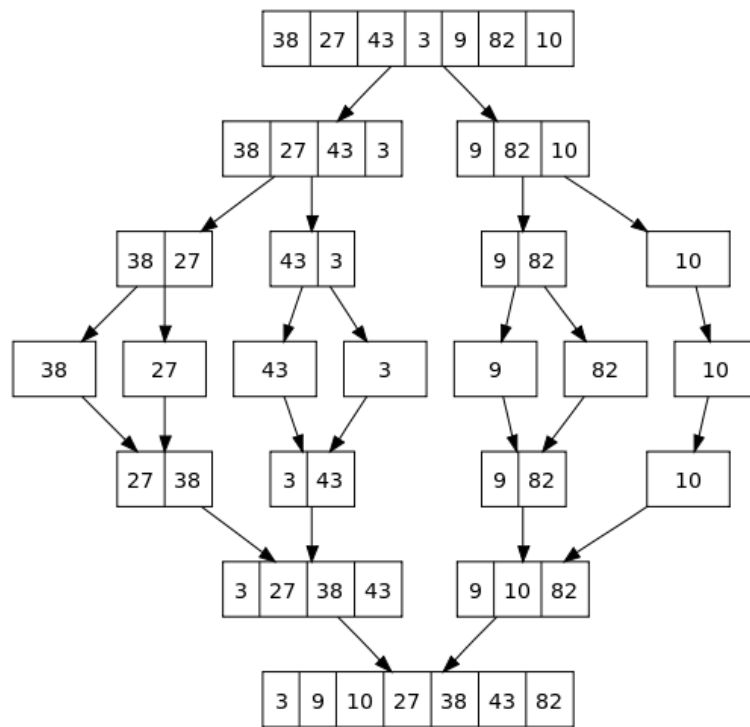


Figura 1: Arbol

### 3. Experimentación y Resultados

#### 3.1. Implemente el algoritmo MergeSort

i) Mediante graficas, muestre que el algoritmo Merge tiene complejidad lineal.

Colocando un contador para cada vez que se llama a la funcion merge, la cual se ejecuta para cada uno de los elementos de nuestro arreglo, se obtubo la siguiente grafica:

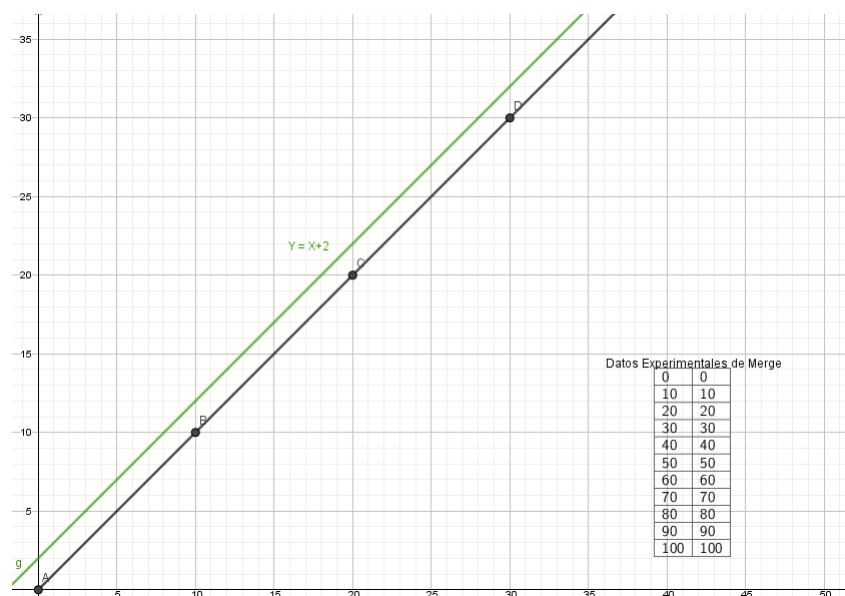


Figura 2: Complejidad del algoritmo Merge mediante graficas

Por lo siguiente, debido a que para cada uno de los elementos la funcion Merge es llamada una vez, y cortejandola con una funcion del tipo:  $\theta(n)$  se concluye que la complejidad del algoritmo Merge es lineal.

ii) Demostrar analíticamente que el algoritmo Merge tiene complejidad lineal.

merge(A, p, q, r)	Costo	Pasos
1.- $n1 \leftarrow p-q+1$	C1	1
2.- $n2 \leftarrow r-q$	C2	1
4.- for $i=0$ to $i<n1$ do:	C3	$n$
5.- $L[i] \leftarrow A[p+i]$	C4	$n-1$
6.- for $j=0$ to $i<n2$ do:	C5	$n$
7.- $R[j] \leftarrow A[q+1+j]$	C6	$n-1$
8.- $i \leftarrow 0$	C7	1
9.- $j \leftarrow 0$	C8	1
10.- for $k=p$ to $k<r$ do:	C9	$n$
11.-     if $L[i] < R[j]$	C10	$n-1$
12.- $A[k] = L[j]$	C11	$n-1$
13.- $i++$	C12	$n-1$
14.-     else	C13	$n-1$
15.- $A[k] = R[j]$	C14	$n-1$
16.- $j++$	C15	$n-1$
8.- return a	C16	1

#### **Demostración.**

Tenemos que:

$$T(n) = C1 + C2 + C3n + C4(n-1) + C5n + C6(n-1) + C7 + C8 + C9n + C10(n-1) + C11(n-1) + C12(n-1) + C13(n-1) + C14(n-1) + C15(n-1) + C16$$

$$T(n) = C1 + C2 + C3n + C4n - C4 + C5n + C6n - C6 + C7 + C8 + C9n + C10n - C10 + C11n - C11 + C12n - C12 + C13n - C13 + C14n - C14 + C15n - C15 + C16$$

Al factorizar nos queda:

$$T(n) = (C3 + C4 + C5 + C6 + C7 + C9 + C10 + C11 + C12 + C13 + C14)n + (C1 + C2 + C7 + C8 + C16 - C4 - C6 - C10 - C11 - C12 - C13 - C14 - C16)$$

$\therefore$

$$T(n) \in \theta(n)$$

iii)Mostrar mediante graficas, que el algoritmo MergeSort tiene complejidad  $\theta(n \log n)$ .

Colocando un contador para cada vez que se llama a la funcion mergeSort, la cual se ejecuta para cada uno de los elementos de nustro arreglo, se obtubo la siguiente tabla:

1	0	46	258	91	600
2	2	47	265	92	608
3	5	48	272	93	616
4	8	49	279	94	624
5	12	50	286	95	632
6	16	51	293	96	640
7	20	52	300	97	648
8	24	53	307	98	656
9	29	54	314	99	664
10	34	55	321	100	672
11	39	56	328		
12	44	57	335		
13	49	58	342		
14	54	59	349		
15	59	60	356		
16	64	61	363		
17	70	62	370		
18	76	63	377		
19	82	64	384		
20	88	65	392		
21	94	66	400		
22	100	67	408		
23	106	68	416		
24	112	69	424		
25	118	70	432		
26	124	71	440		
27	130	72	448		
28	136	73	456		
29	142	74	464		
30	148	75	472		
31	154	76	480		
32	160	77	488		
33	167	78	496		
34	174	79	504		
35	181	80	512		
36	188	81	520		
37	195	82	528		
38	202	83	536		
39	209	84	544		
40	216	85	552		
41	223	86	560		
42	230	87	568		
43	237	88	576		
44	244	89	584		
		90	592		
		91	600		

Figura 3: Resultados obtenidos tras la ejecucion de MergeSort

Para este algoritmo se propone la funcion  $\theta(n \log(n))$ , quedando graficada de color verde, siendo la linea de color obscuro, la grafica de la tabla anterior.

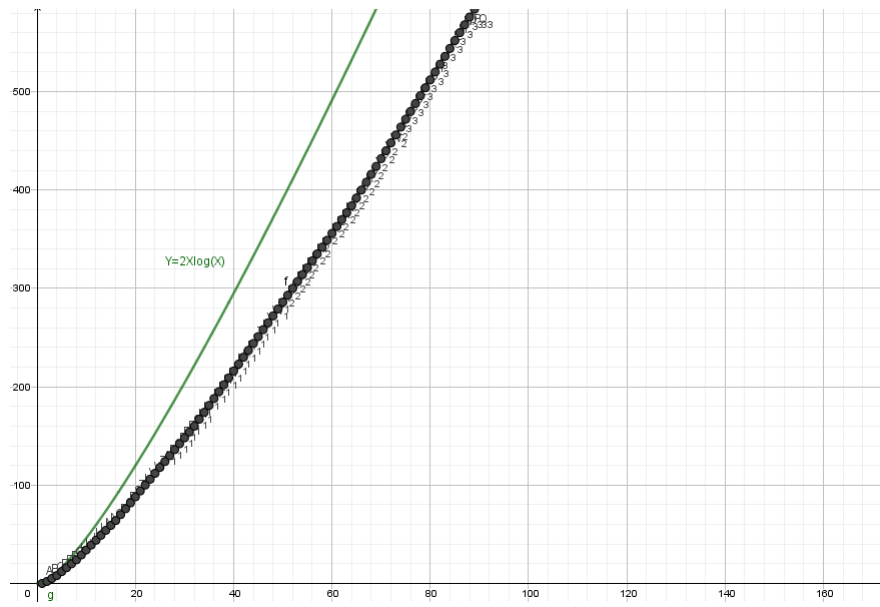


Figura 4: Complejidad del algoritmo MergeSort mediante graficas.

Debido a los resultados obtenidos tras la experimentacion, se concluye que el algoritmo MergeSort, tiene complejidad  $n \log(n)$

iv) Demostrar analíticamente que el algoritmo MergeSort tiene complejidad  $\theta(n \log n)$ .

mergesort(A, p, r)	Costo
1.- if $p < r$	1
2.- $q = \frac{(p+r)}{2}$	$D(n)$
3.- Mergesort(A, p, q)	$T(\frac{n}{2})$
4.- Mergesort(A, q+1, r)	$T(\frac{n}{2})$
5.- Merge (A, p, q, r)	$C(n)$

Sea  $D(n)$  el costo computacional de dividir el arreglo de tamaño  $n$ .

Sea  $C(n)$  el costo computacional de combinar la solución de subproblemas para obtener la solución del problema original.

luego

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

$$T(n) = \begin{cases} C_1 & \text{si } n = 1 \\ 2T(\frac{n}{2}) + \theta(n) & \text{si } n > 1 \end{cases} \quad (2)$$

$$T(n) = \begin{cases} C & \text{si } n = 1 \\ 2T(\frac{n}{2}) + C_n & \text{si } n > 1 \end{cases} \quad (3)$$

Sea  $n$  tal que  $K = \log_2 n$  ( $n = 2^K$ ) luego

$C$  si  $n = 2^k = 1$

$2T(2^{k-1}) + C2^k$  si  $n = 2^n > 1$

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

$= 2[2T(2^{K-3}) + 2^{K-2}C] + 2^{K-1}C$

...

$= 2^i T(2^{K-i}) + iC2^k$   $k-i=0 \quad k=i$

$= 2^K T(1) + KC2^K$

$= C2^K + K2^K$

$= (K+1)C2^K$

$= \log_2(n+1)Cn$

$= Cn \log_2 n + Cn \in \theta(n \log_2 n)$



## 4. Conclusión

En esta practica logre darme cuenta que ordenar un arreglo es mas complejo de lo que parece, que es nuestro deber saber implementar bien cada uno de los algoritmos que vamos conociendo para así lograr obtener el mejor resultado, al principio tuve algunos problemas con las graficas, debido a que no sabia que el MergeSort era logaritmico, yo pense que sería exponencial, me agrado lograr determinar despues que sun complejidad es logartimica ya que eso era lo que indicaban mis resultado.

De igual manera me parece que el escoger Python para programar estos ejercicios fue la desicion correcta, es mas sencillo trabajar con los datos en este lenguaje y asi obtener los resultados esperados.

## 5. Anexos:

**5.1. Calcular el orden de complejidad de los siguientes algoritmos en el mejor ( $\Omega$ ) y en el peor de los casos ( $O$ ) (no es necesario hacer el analisis linea por linea, en este caso, puede aplicar propiedades de los algoritmos vistos en clase):**

A

Función1( n par )

1. i=0
2. mientras i ≤ n hacer
3.     para j=1 hasta j=10 hacer
4.         Accion(i)
5.         j++
6. i+=2

Suponga que Accion  $\in \theta(1)$

Solución

- A) 1.i=0  $\leftarrow \theta(1)$   
B) 2. mientras i ≤ n hacer  $\leftarrow \theta(n)$   
C) 3.     para j=1 hasta j=10  $\leftarrow \theta(n)$   
D) por el teorema 1.4 para 2. y 3.  $\leftarrow \theta(n^2)$   
E) 4.         Accion(i)  $\leftarrow \theta(1)$   
F) 5.         j++  $\leftarrow \theta(1)$   
G) 6. i+=2  $\leftarrow \theta(1)$   
H)  $T(n) = \theta(1) + \theta(n^2) + \theta(1) + \theta(1) + \theta(1) = \theta(1) + \theta(n^2)$   
Por los teoremas 1.2 y 1.4 :  
 $T(n) = \theta(n^2)$

## B

Función2(A[0,...n-1,x entero])

1. for i=0 to i=n do
2.     if(A[i]≤x)
3.         A[i]=min(A[0,...n-1])
4.     else if(A[i]≥x)
5.         A[i]=max(A[0,...n-1])
6.     else
7.         exit

Solución

En el mejor de los casos es cuando  $A[i]=X$ , lo cual deja  $T(n) = \theta(1)$

El peor de los casos es cuando  $A[i] \neq x$ , por lo que

- A) 1. *for*  $i = 0$   $i < n$  *do*  $\leftarrow \theta(n)$
- B) 2.     *if* ( $A[i] < x$ )  $\leftarrow \theta(1)$
- C) 3.          $A[i] = \min(A[0, \dots, n-1]) \leftarrow \theta(n)$
- D)         *Por el teorema 1,4 en 2. y 3*  $\leftarrow \theta(n)$
- E) 4.     *elseif* ( $A[i] > X$ )  $\leftarrow \theta(1)$
- F) 5.          $A[i] = \max(A[0, \dots, n-1]) \leftarrow \theta(n)$
- G)         *Por el teorema 1,4 en 4. y 5*  $\leftarrow \theta(n)$
- H) 6.             *else*  $\leftarrow \theta(1)$ .
- I) 7.     *exit*  $\leftarrow \theta(1)$
- J)         *Por el teorema 1,2 en 6. y 7*  $\leftarrow \theta(1)$

$$T(n) = \theta(n) + \theta(n) + \theta(n) + \theta(1)$$

Por los teoremas 1.2 y 1.4 :

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