Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos, Sem: 2021-1, 3CV1, Práctica 4, 25-11-2020

Práctica 4: Divide y vencerás.

De la Cruz Sierra Diana Paola
Raya Chávez Samuel Antonio
dianapaodcs@gmail.com, samiraya1323@gmail.com

Resumen. Se manejará el concepto 'Divide y vencerás' aplicado a diferentes algoritmos de ordenamiento de datos. Más adelante se analizarán los algoritmos QuickSort y MergeSort.
Palabras clave. Divide y vencerás, Quicksort, Mergesort.

1. Introducción

Para adentrarnos a estrategias de algoritmos debemos primero recordar que:

- Un algoritmo es una secuencia finita y bien definida de pasos utilizada para resolver un problema bien definido.
- Para una estrategia se da un enfoque para resolver un problema e incluso pueden combinarse varios enfoques.
- En un algoritmo recursivo la función se llama a si misma.

Divide y Vencerás es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de subproblemas del mismo tipo, pero de menor tamaño. Si los subproblemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subproblemas lo suficientemente pequeños para ser solucionados directamente. Para esto se sugiere el uso de la recursión.
La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

1. **Dividir:** el problema se plantea de forma que pueda ser descompuesto en k subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k subproblemas ($donde 1 \leq k \leq n$), cada uno con una entrada de tamaño nk y donde $0 \leq nk < n$. A esta tarea se le conoce como división
2. **Vencer:** se resuelven independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos base.
3. **Combinar:** por último, combinar las soluciones obtenidas en el paso anterior para construir la solución del problema original.

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de Divide y Vencerás van a heredar las ventajas e inconvenientes que la recursión plantea:

- El diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.

- Los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Desde un punto de vista de la eficiencia de los algoritmos Divide y Vencerás, es muy importante conseguir que los subproblemas sean independientes, es decir, que no exista solapamiento entre ellos. De lo contrario el tiempo de ejecución de estos algoritmos será exponencial.

En cuanto a la eficiencia hay que tener en también en consideración un factor importante durante el diseño del algoritmo: el número de subproblemas y su tamaño, pues esto influye de forma notable en la complejidad del algoritmo resultante.

2. Conceptos básicos

A continuación, se denotarán algunos algoritmos de ordenamiento de datos que trabajan bajo el concepto de divide y vencerás y su funcionalidad general.

- **QuickSort:** Se elige un elemento como el pivote. Se acomoda el pivote de modo que todos los elementos a la izquierda de este sean menores y todos los elementos a la derecha sean mayores. De esta manera se localiza la posición a la que pertenece el pivote. Se manda llamar la función QuickSort recursivamente para la parte izquierda y la parte derecha del pivote. La función terminará cuando el tamaño del subarreglo es de un elemento. (Arreola, A., 2017).

- **MergeSort:** Mergesort desglosa repetidamente una lista en varias sublistas hasta que cada sublista conste de un solo elemento, y fusiona esas sublistas de manera que resulte en una lista ordenada. (InterviewBit, 2019).

- **Problema del máximo subarreglo:** Es un problema en el cuál se quiere obtener, de una lista de números, la parte de la lista en donde la suma de los elementos de la lista es la más grande de toda la lista. (Pozos, A., Trinidad, R. & Rodriguez A., 2018).

- **Algoritmo de Strassen:** La idea del algoritmo de Strassen es dividir las matrices en cuatro partes iguales, y resolver el producto original en base a operaciones sobre estas partes. (Fillottrani, P, 2017).

2.1. Pseudocódigo del algoritmo Quicksort

Para poder llevar a cabo el algoritmo QuickSort por medio de la estrategia divide y vencerás, es necesario implementar una función que llamaremos Partition, cuya función es dividir un arreglo en dos sub arreglos. El pseudocódigo de esta función puede observarse en el algoritmo 1

Algorithm 1 Pseucódigo: Partition

```
1: procedure PARTITION(A[p,...,r], p, q)
2:     x ← A[r]
3:     i ← p − 1
4:     for j = p, j ≤ r − 1 do
5:         if A[j] ≤ x then
6:             i + +
7:         end if
8:     end for
9:     exchangue(A[i], A[j])
10:    return i + 1
11: end procedure
```

Este procedimiento se repetirá de manera recursiva hasta obtener sub arreglos de tamaño uno, lo cuál se llevará a cabo por medio de la función QuickSort, cuyo pseudocódigo puede observarse en el algoritmo 2

Algorithm 2 Pseucódigo: Partition

```
1: procedure QUICKSORT(A[p,...,n], p, n)
2:     if p < n then
3:         q ← Partition(A, p, n)
4:         QuickSort(A, p, q − 1)
5:         QuickSort(A, q + 1, n)
6:     end if
7: end procedure
```

Al analizar el algoritmo de QuickSort observamos que la función se llama recursivamente 2 veces, una para el sub arreglo del lado izquierdo que es de tamaño q y otro para el lado derecho que es de tamaño $n - q$, esto se detiene hasta que los sub arreglos sean de tamaño 1, lo que significara que en automático el arreglo estará ordenado.

2.2. Pseudocódigo del algoritmo Mergesort

Para este algoritmo de ordenamiento, será necesario crear una función llamada Merge, la cual se encargará de dividir el arreglo que reciba como parámetro en 2 subarreglos. A continuación se presenta el pseudocódigo correspondiente al algoritmo Merge.

Algorithm 3 Pseudocódigo: Merge

```
1: procedure MERGE(A[p, ..., q, ...r], p, q, r)
2:     n1 ← q − p + 1
3:     n2 ← r − q
4:     L[0, ..., n1 − 1], R[0, ..., n2 − 1]
5:     for i ← 0, i <= n1 − 1 do
6:         L[i] = A[p + i]
7:     end for
8:     for j = 0, j <= n2 − 1 do
9:         R[j] = A[q + j + 1]
10:    end for
11:    i = 0
12:    j = 0
13:    for k = p, k <= r do
14:        if L[I] <= R[j] then
15:            A[k] = L[i]
16:            i + +
17:        else
18:            A[k] = R[j]
19:            j + +
20:        end if
21:    end for
22: end procedure
```

El algoritmo 3 se repetirá hasta que solo queden subarreglos de un elemento, y con ayuda del algoritmo MergeSort se empezarán a ordenar estos elementos.

El pseudocódigo para el algoritmo MergeSort se presenta a continuación.

Algorithm 4 Pseudocódigo: MergeSort

```
1: procedure MergeSort(A[p, ...r], p, r)
2:     if p<r then
3:         q = (p + r)/2
4:         MergeSort(A, p, q)
5:         MergeSort(A, q + 1, r)
6:         Merge(A, p, q, r
7:     end if
8: end procedure
```

De aquí se puede observar que el algoritmo 4 Se llama recursivamente 2 veces, la primera vez para el primer subarreglo formado (lado izquierdo) de tamaño $'p'$ a $'q'$, y la segunda vez para el lado derecho de tamaño $'q + 1'$ a $'r'$. Una vez que los subarreglos solo contengan un elemento, se empiezan a ordenar los elementos.

3. Experimentación y resultados

3.1. Algoritmo quickSort

Comenzamos mostrando dos ejemplos del algoritmo 2 con dos arreglos de tamaño 9 y 10 respectivamente y en desorden mostrados en la figura 1.

```
C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica4>g++ -o QuickSort QuickSort.cpp

C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica4>QuickSort
input size of vector: 9
input vector: 9 0 5 2 6 8 2 1 3
0 1 2 2 3 5 6 8 9

C:\Users\Diana Paola\Documents\5to Semestre\Algoritmos\Practica4>QuickSort
input size of vector: 10
input vector: 2 56 47 85 23 14 5 1 2 2 17
1 2 2 2 5 14 23 47 56 85
```

Figura 1: Ejemplos del algoritmo 2 programado

A continuación se mostraran ejemplos del número de pasos para ciertos arreglos en distinto orden donde solamente se hará una partición del arreglo utilizando el algoritmo 1. Este análisis se aproximo suponiendo que ejecuta 2 pasos cada que entra al for (inicialización y condición) y otro cada que entra al if y hace el cambio, ya que cada una de estas lineas es $\theta(1)$. Este análisis puede observarse mejor en la tabla 1

Arreglo	**Tamaño**	**No. de pasos**
9 0 5 2 6 8 2 1 3	9	21
9 8 7 6 5 4 3 2 1	9	17
5 4 3 2 1	5	9
4 2 1 5 3	5	11
1 2 3 4 5 6 7	7	19
1 7 5 4 2 3 6	7	18
2 56 47 85 23 14 5 1 2 2 17	10	22

Cuadro 1: Análisis a posteriori del algoritmo 1

Los puntos graficados de la tabla 1 pueden observarse en la imagen 2

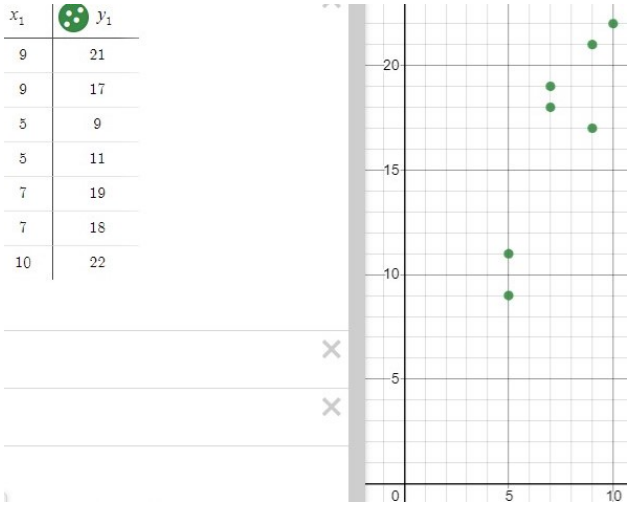


Figura 2: puntos correspondientes al algoritmo 1

Analizando más a detalle este comportamiento, se observa que incluso para varios arreglos del mismo tamaño n puede variar la cantidad de pasos dependiendo de la manera en que este ordenado el arreglo. Sin embargo por la forma de los puntos si graficamos la ecuación: $3n$ se observa que esta funge como cota superior. Esto se ve de manera gráfica en la imagen 3

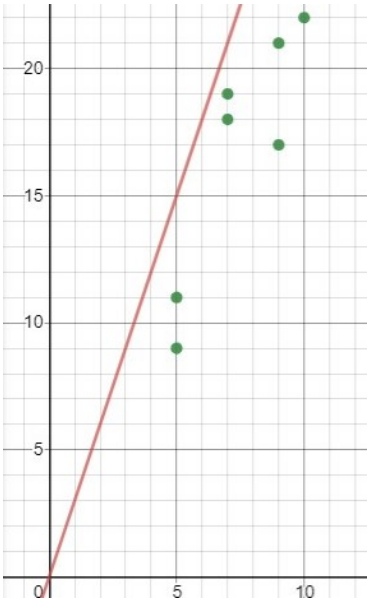


Figura 3: Cota superior T(n)=3n y los puntos del análisis a posteriori del algoritmo 1

Dicho lo anterior puede concluirse que para un análisis A posteriori del algoritmo 1: Partition $\in \Theta(n)$

En la tabla 2 se muestra un análisis para calcular el orden de complejidad por medio de bloques del algoritmo 1.

Partition(A[p,...,r]			
x = A[r]	(1)		**(n)**
i = p-1			
for j=p to j≤ r-1	(n)	(n)	
if($A[j] \leq x$)	(1)		
i++			
exchangue(A[i],A[j])			
exchangue (A[i], A[j])	(1)		
return i+1			

Cuadro 2: Complejidad del algortimo 1

Una vez calcundo el algortimo 1 por bloques se tiene que :

$$T(n) = n \tag{1}$$

De manera que Partition $\in \theta(n)$.

Ahora vamos a hacer un pequeño análisis A posteriori del algoritmo 2, para esto solamente se hizo uso de varios arreglos ordenados de manera distinta pero con un mismo tamaño n. Esto se puede ver en la tabla 3

A	**Tamaño**	**#pasos**
9 0 5 2 6 8 2 1 3	9	42
9 8 7 6 5 4 3 2 1	9	88
1 5 9 0 2 4 3 7 6	9	48
12 0 9 85 64 12 45 3 7	9	56
5 9 1 8 2 7 3 6 4	9	44
1 1 1 2 2 2 0 25 25	9	74

Cuadro 3: Pasos para arreglos de tamaño n en distinto orden

La gráfica de los puntos en la tabla 3 pueden verse en la imagen 4.

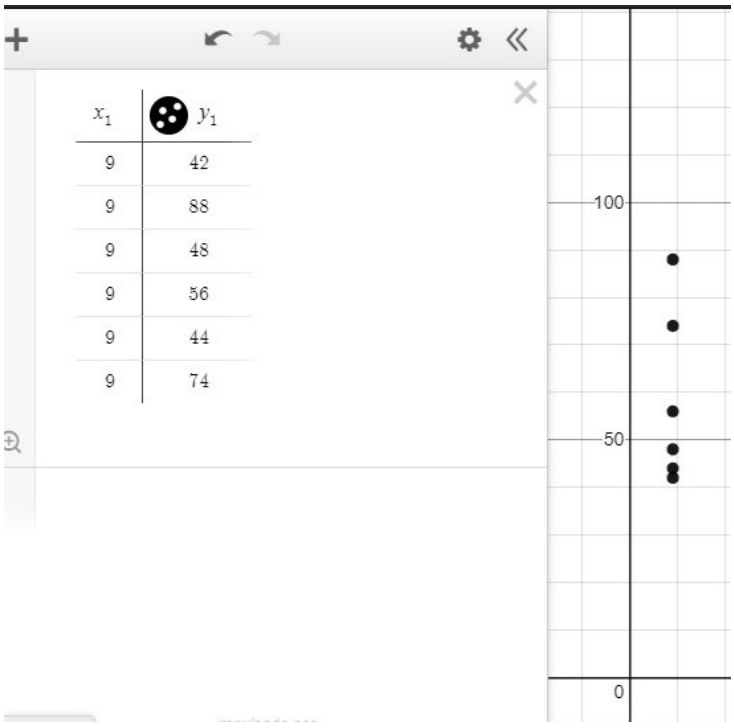


Figura 4: Puntos de la tabla 3

Si a dichos puntos también le graficamos la ecuación 2:

$$4nlogn \tag{2}$$

Se observa que está ecuación acota por arriba a los puntos gráficados de la tabla 3. Dicho resultado se muestra en la imagen 5

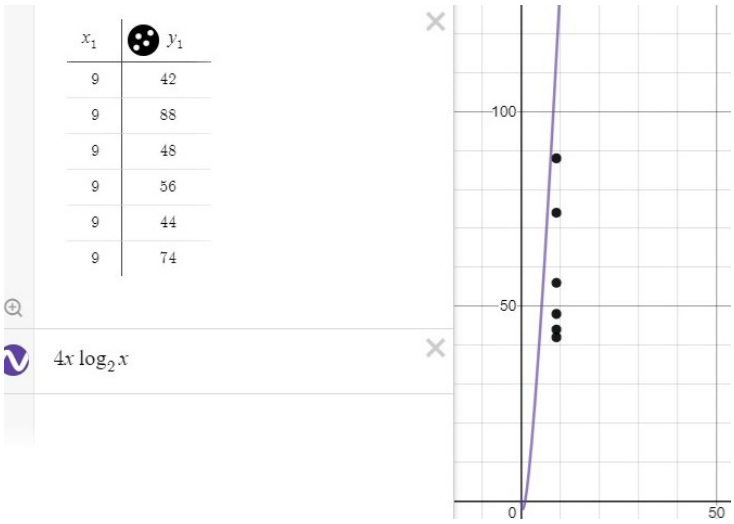


Figura 5: Análisis a posteriori del algoritmo 2.

Entonces de manera gráfica QuickSort $\in \Theta(nlogn)$.

Ahora calculando análiticamente el algoritmo QuickSort puede observarse en la tabla 4 un pequeño análisis a bloques. De ahí se tiene que

$$T(n) = T(q) + T(n-q) + cn \tag{3}$$

QuickSort(A[p,...,n], p, n)	
if p <n	(1)
q = Partition(A, p, n)	(n)
QuickSort(A, p, q-1)	T(q)
QuickSort(A, q+1, n)	T(n-q)

Cuadro 4: Complejidad del algoritmo 2

Sin embargo si este análisis lo hacemos para el mejor caso donde la función partition tiene a pivote (q) exactamente a la mitad del arreglo Se observa que la llamada recursiva QuickSort ahora es: T(n/2), por tanto el orden de complejidad que teníamos en la ecuación 3 ahora es:

$$T(n) = 2T(n/2) + cn \tag{4}$$

Resolviendo esta recurrencia mostrada en la ecuación 4 se tiene que:
Utilizando el método maestro
a=2, b=2, f(n)=cn y $n^{log_b a}$=n
Puesto que
f(n) = cn $\in \theta(n^{log_b a})=\theta(n)$
Entonces por el Teorema maestro caso II:
T(n) $\in \theta(n^{log_b a} logn)$ i.e
T(n) $\in \theta(nlogn)$

Ahora para calcular aproximadamente el orden de complejidad que tendría el algoritmo 2 si todos sus elementos son distintos y además están ordenados de forma descendente haremos primero uso de la tabla 5

Arreglo	**tamaño**	**No. pasos**
5 4 3 2 1	5	28
9 8 7 6 5 4 3 2 1	9	88
12 11 10 9 8 7 6 5 4 3 2 1	12	154
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	15	288

Cuadro 5: No pasos de arreglos en forma decreciente usando el algoritmo 2.

En la figura se muestra la gráfica de la tabla 5

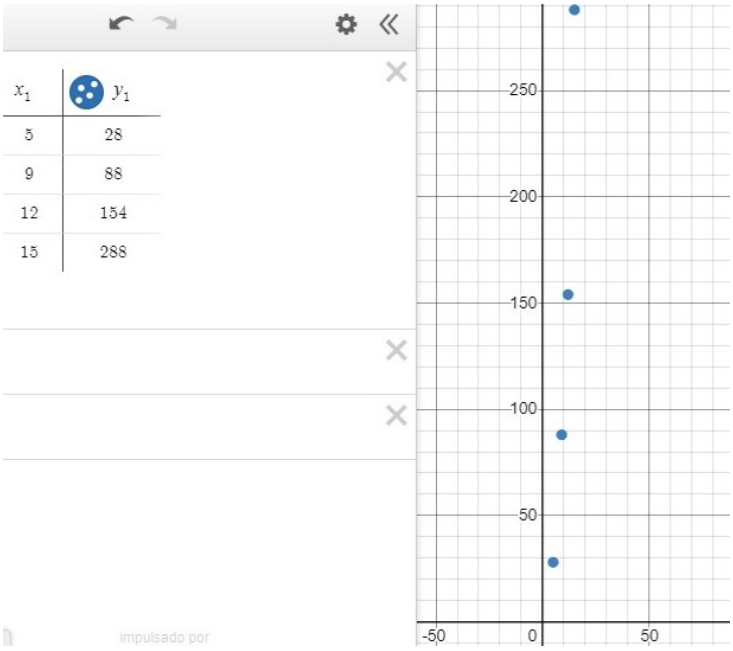


Figura 6: Gráfica de la tabla 5.

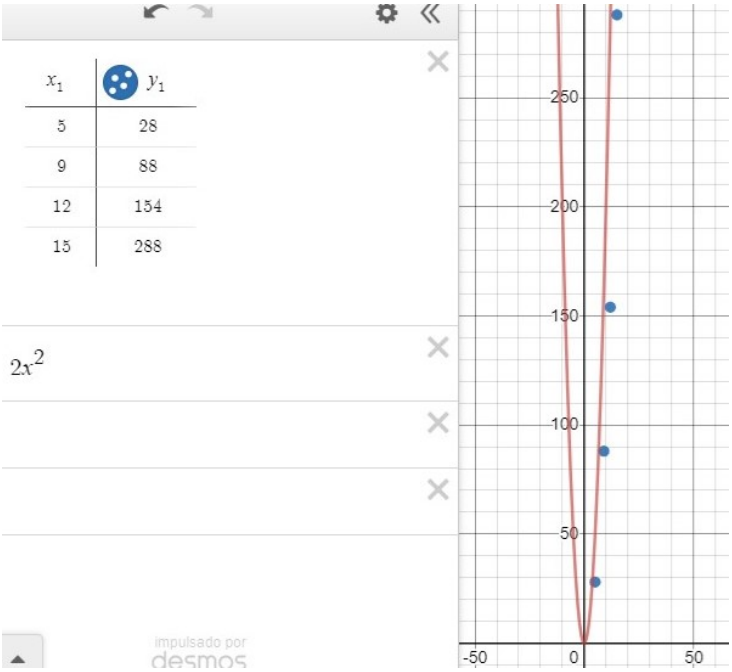


Figura 7: Puntos graficados de la tabla 5 y ecuación propuesta

Y en la figura 7 se muestra también la gráfica $2n^2 \in \theta(n^2)$ en color rojo, esto nos indica un peor caso, en donde, si el algoritmo está ordenado de forma decreciente, entonces QuickSort $\in \theta(n^2)$.

3.2. Algoritmo MergeSort

Ejecutando el programa, el resulta obtenido, por ejemplo, son los siguientes resultados.

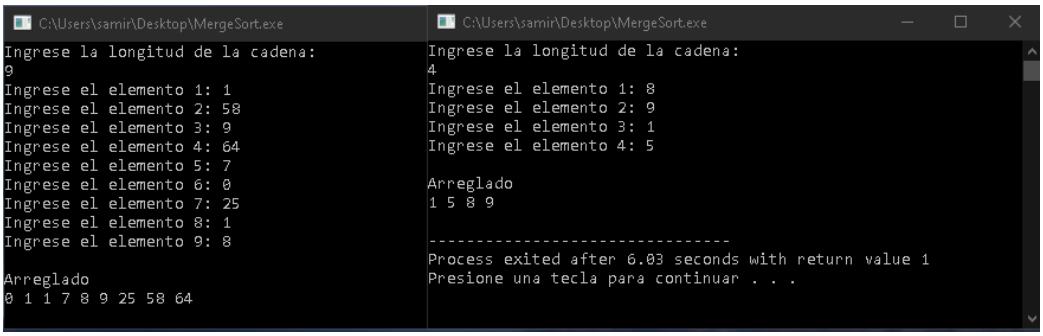


Figura 8: Pruebas del algoritmo 4

De la figura 8 se puede observar que los arreglos dados no tienen un orden

especifico y el programa se encarga de arreglos estos datos.

Ahora, se mostrarán algunos arreglos propuestos y el número de pasos realizados para llegar realizados dentro del algoritmo 3.

Arreglo	**Tamaño**	**No. de pasos**
15 78 0	3	32
1 7 8 9 3	5	72
4 0 8 36 14	5	75
5 4 3 2 1	5	72
8 1 1 5 9 3 1	7	116
19 82 14 36 0 2 1 1	8	141
7 3	2	14

Cuadro 6: Número de pasos del algoritmo 3

Tomando en cuenta los resultados obtenidos, se ingresan estos datos a una tabla y se propone una recta donde se obtiene lo siguiente:

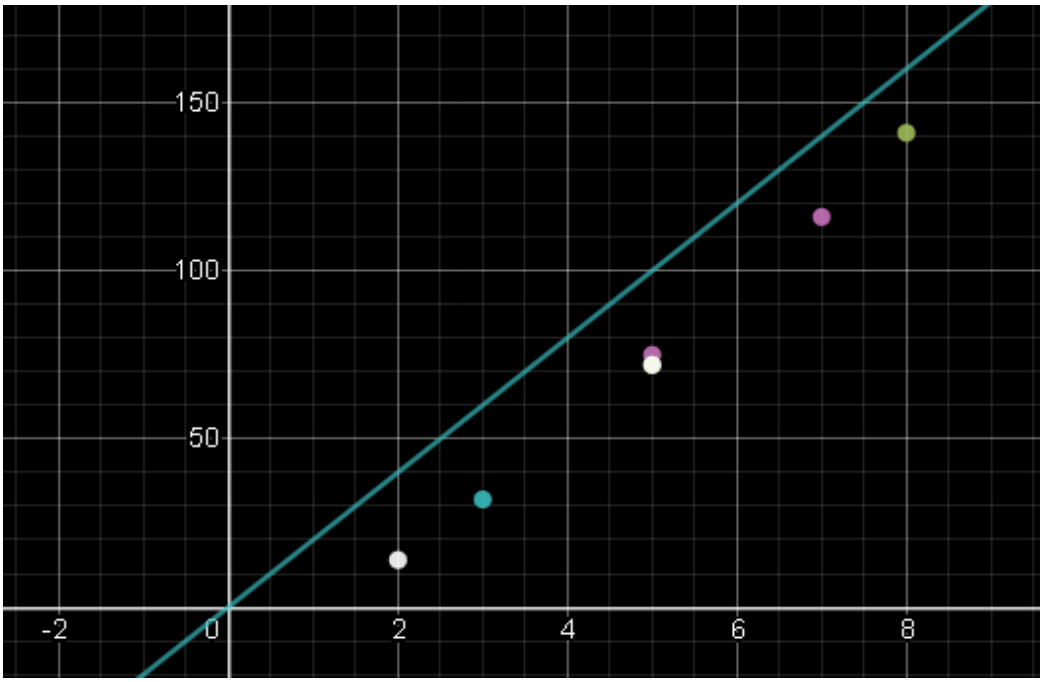


Figura 9: Recta propuesta T(n)=20n que es cota superior de los puntos propuestos

Con lo anterior se concluye que el algoritmo 3: Merge $\in \Theta(n)$.

De igual forma, haciendo un análisis para el orden de complejidad del algoritmo 3, se toma lo siguiente:

Merge(A[p,...,q,...,r],p,q,r)			
n1=q-p+1	(1)		**(n)**
n2=r-q			
L[0,...,n1-1], R[0,...,n2-1]			
for i=0 to i $\leq$ n1-1	(n1)	**(n)**	
L[i]=A[p+i]			
for j=0 to j $\leq$ n2-1	(n2)		
R[j]=A[p+i]			
i=0	(1)		
j=0			
for k=p to k $\leq$ r	(r-p+1)	**(n)**	
if($L[i] \leq R[j]$)			
A[k]=L[i]			
i++			
if(else)			
A[k]=R[j]			
j++			

Cuadro 7: Complejidad del algoritmo 3

Mediante un análisis a bloques del algoritmo 3, se observa que éste tiene un orden de complejidad T(n)=n, es decir, Merge $\in \Theta$(n).

Ahora analizando el algoritmo 4, primero se realizará un análisis a-posteriori del número de pasos que éste ejecutará.

A	**Tamaño**	**#pasos**
999 445 124 587 36 21 56	7	126
489 156 147 259 354 269 19	7	129
10 20 30 40 50 60 70	7	135
50 40 30 20 10	5	83
49 28 75 34 58	5	80

Cuadro 8: Arreglos y la cantidad de pasos realizados

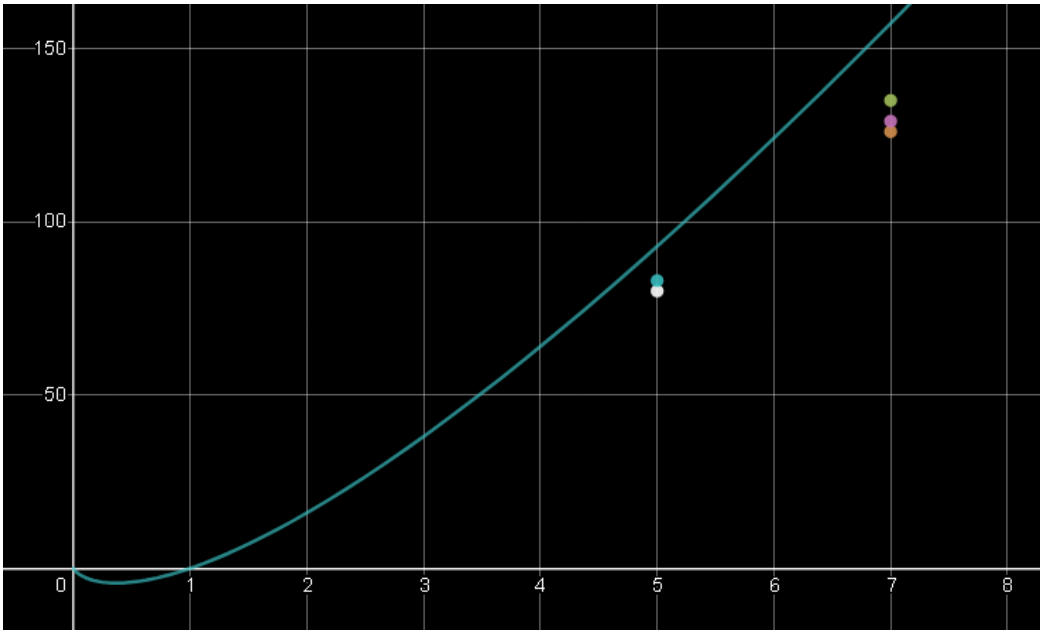


Figura 10: Cota superior T(n)=8nlog(n)

Y así se obtiene que T(n)=8nlog(n), o mejor dicho como:

$$\text{MergeSort} \in \Theta\ (\text{nlog(n)}).$$

Ahora, realizando un análisis a-posteriori del algoritmo 4 se tiene lo siguiente:

MergeSort(A[p,...,r],p,r)		
if(p<r)		
q=p+r/2	(1)	
MergeSprt(A,p,q)	T(n)	2T(n)
MergeSort(A,q+1,r)	T(n)	
Merge(A,p,q,r)	(n)	

Cuadro 9: Complejidad del algoritmo 4

Como se puede ver, cuando se llama a Merge éste tiene un orden de complejidad *cn* lo cuál ya se vio previamente en el cuadro 7, y al llamar a la función Mergesort se tendrá un tiempo de T(n/2). Teniendo así un tiempo de complejidad de T(n)=2T(n/2)+cn.

Haciendo uso del método sustitución hacia atrás para resolver la recurrencia anterior se tiene que:

Sea n=2^k; (k=log(n))
Entonces, T(2^k) = T(2^{k-1})+cn
[T(2^{k-2})+cn] + cn = T(2^{k-2})+2cn
[T(2^{k-3})+cn] + 2cn = T(2^{k-3})+3cn
...
En (i) = T(2^{k-i} + icn
...
En la condición frontera, que es cuando k=i
(i=k) = T(2^0) + kcn
= 0 + c(kn)
= c(kn); y como k=log(n)
= c(nlog(n))
Por lo tanto se tiene que T(n)$\in \Theta$(nlog(n))

4. Conclusiones

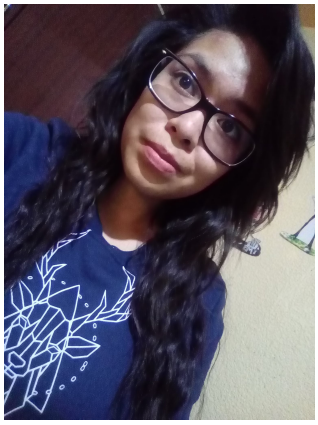
- Conclusiones generales:

  En cuanto al algoritmo quickSort se observó que en el peor caso su complejidad es cuadrática, sin embargo es un algoritmo muy utilizado debido a que el peor caso es menos probable que caso y fuera de ese caso es de los algoritmos más rápidos.

  Para el caso de Mergesort, se obtuvo que tanto para el peor caso como para el mejor caso el orden de complejidad es nlog(n), lo que la hace una opción viable si se busca una ordenamiento de datos.

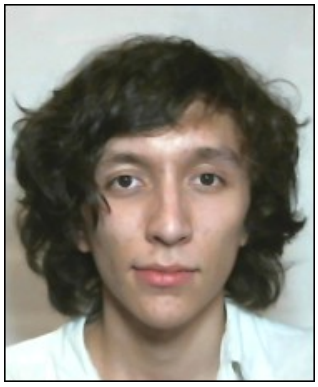
- Diana Paola De la Cruz Sierra

  El método divide y vencerás es gran utilidad si se sabe aplicar de manera adecuada, hacer esto hará que el tiempo de ejecución disminuya o que la manera de dar solución a un problema dado sea más sencillo de implementar debido a que su entendimiento será mejor.



- Raya Chávez Samuel Antonio

  Al hablar de algoritmos lo primero que nos suelen decir es usar el algoritmo divide y vencerás ya que éste es muy fácil de entender y aplicar para cualquier tipo de problemas, solo que está en uno el saber identificar hasta cuando hay que dividir el problema para poder empezarlo a solucionar.



5. Fuentes de información

- Universidad Carlos III Madrid. (S.F). *Tema7: estrategías algortimicas: divide y venceras.* [PDF]. Disponible en: http://ocw.uc3m.es/ingenieria.informatica/estructura.datos.algoritmos/material.de.clase.1
- uma.es. (S.F). *Capitulo 3: divide y venceras.* [PDF]. Disponible en: http://www.lcc.uma.es/ av/Libro/CAP3.pdf
- Arreola, A.. (2017). Quicksort. noviembre 30, 2020, de Universidad Autonoma de Baja California Sitio web: https://quicksortweb.wordpress.com/2017/10/07/func

- InterviewBit. (2019). Merge Sort Algorithm. diciembre 01, 2020, de Interview Bit Sitio web: https://www.interviewbit.com/tutorial/merge-sort-algorithm/
- Pozos, A., Trinidad, R. y Rodriguez A. (2018). Suma máxima de un sub-arreglo. diciembre 01, 2020, de Github Sitio web: https://andreandyp.github.io/mss/
- Fillottrani, P. (2017). Depto. Ciencias e Ingeniería de la Computación Universidad Nacional del Sur. diciembre 01, 2020, de Algoritmos y Complejidad Sitio web: http://www.cs.uns.edu.ar/ prf/teaching/AyC17/downloads/Teoria/1x1.pdf
- Durán, J. (2016). Cómo manejar números grandes. diciembre 04, 2020, de Somos Binarios Sitio web: https://www.somosbinarios.es/como-manejar-numeros-grandes/
- Hartnett, K.,2019). La forma perfecta de multiplicar. diciembre 04, 2020, de Investigación y ciencia Sitio web: https://www.investigacionyciencia.es/noticias/la-forma-perfecta-de-multiplicar-17404

6. Anexos

6.1. Algoritmo de Karatsuba

1. ¿Cómo se pueden implementar números enteros muy grandes?

   Para estos casos se encuentran dos soluciones:

   • Utilizar números reales para guardar números enteros, si la precisión no es importante.

   • Utilizar "Enteros Grandes", para no perder precisión.

   Los Enteros Grandes son un tipo de datos, que no pertenecen a los tipos de datos del lenguaje, pero se pueden encontrar en muchas librerías, estos enteros grandes permiten utilizar enteros que ocupan cientos de bits. (Durán, J., 2016).

2. Determine el orden de complejidad del producto usual de números enteros.

   Ya que para realizar una multiplicación elemental se requiere de una pareja de dígitos, uno para el multiplicador y otro para el multiplicando, se tiene que existen n^2 parejas; mientras que lo demás por resolver es de orden menor.

   Por lo que se llega a que el orden de complejidad es $\leq cn^2$.

3. Determine el orden de complejidad del algoritmo de Karatsuba.

   Ya que el algoritmo de Karatsuba cambia las multiplicaciones por sumas, se ahorra tiempo porque la adición solo requiere 2n pasos en lugar de n^2. Por lo que el orden de complejidad es de $n^{log_2 3}$, lo que se traduce como $n^{1,58}$ multiplicaciones de un solo dígito.

   Por lo tanto, el algoritmo $\in \Theta(n^{1,58})$. (Hartnett, K., 2019).

4. Muestre dos ejemplos aplicados al algoritmo de Karatsuba.

   - 25 x 63

   A. Se dividen los números: 25 ->2,5; 63->6,3

   B. Se multiplican las decenas: 6 x 2 = 12

   C. Se multiplican las unidades: 5 x 3 = 15

   D. Se suman los dígitos: 2+5=7; 6+3=9

   E. Se multiplican las sumas: 7 x 9 = 63

   F. Se restan B y C de E: 63-12-15=36

   G. Se ensamblan los números: 12(00)+(0)36(0)+(00)15 = 1575

   - 2531 x 1467

   A. 25,31; 14,67

   B. 25 x 14 = 350

   C. 31 x 67 = 2077

   D. 25+31=56; 14+67=81

   E. 56x81=4536

   F. 4536-2077-350=2109

   G. 350(0000)+(0)2109(00)+(000)2077 = 3, 712, 977

6.2. Resolución de problemas

1. ¿Qué valor de q retorna Partition cuando todos los elementos en el arreglo A[p, ..., r] tienen el mismo valor?

   Respuesta: q = n-1, donde n es la cantidad de elementos en el vector

2. ¿Cúal es el tiempo de ejecución de QuickSort cuando todos los elementos del arreglo tienen el mismo valor?

   Respuesta: n^2

Este es un peor caso en donde partition divide al arreglo en dos subarreglos de tamaño 1 y n-1 respectivamente, por lo que si hacemos uso de la ecuación 3 la recurrencia queda como:

$$T(n) = T(1) + T(n+1) + cn = T(n-1) + c(n+1) \tag{5}$$

Resolviendo la recurrencia 5: si aplicamos decremento por uno se tiene que: T(n) $\in \Theta(n^2)$ q.e.d.

3. ¿Qué retorna la función de máximo subArreglo cuando todos los elementos de arreglo tiene el mismo valor?

   Respuesta: Depende, en este caso si los valores valores son negativos entonces solamente regresará el único valor p que tengan los valores que es lo mismo a A[0], esto debido a que si se suman el valor sería un negativo más grande.

   Primero análizando MaxCrossingSubArray se tiene que regresa (mitad, mitad+1, sumaIzq+sumaDer) y empleando este resultado en MaxSubArray se observa que el tercer parametro de MaxCrossingSubArray siempre será menor que la sumaIzq y la sumaDer. Entonces finalmente regresa (ind, ind, p) en donde ind son valores idénticos y p es el valor que tienen los elementos del arreglo. (Osea, no regresa ninguna suma adicional).

   Y si los valores son mayores o igual a cero entonces regresara los valores (ini, fin, $\sum_{i=1}^{i=n} p$), en donde, el primer parametro regresa 0 por que es la posición del primer elemento, el segundo parámetro regresa $n-1$ por ser la última posición del arreglo y en el tercer parámetro se suman todos los valores del arreglo.

4. Calcule el producto de las siguientes matrices mediante el algoritmo Strassen (paso a paso)

$$\begin{bmatrix} 1 & 2 & 3 & 1 \\ 3 & 4 & 4 & 2 \\ 5 & 1 & 2 & 1 \\ 2 & 3 & 0 & 1 \end{bmatrix} X \begin{bmatrix} 1 & 4 & 5 & 1 \\ 2 & 1 & 3 & 2 \\ 1 & 4 & 0 & 3 \\ 4 & 5 & 2 & 2 \end{bmatrix} \tag{6}$$

   Solución:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 1 \\ 3 & 4 & 4 & 2 \\ 5 & 1 & 2 & 1 \\ 2 & 3 & 0 & 1 \end{bmatrix} X \begin{bmatrix} 1 & 4 & 5 & 1 \\ 2 & 1 & 3 & 2 \\ 1 & 4 & 0 & 3 \\ 4 & 5 & 2 & 2 \end{bmatrix} \tag{7}$$

vamos a dividir/separar las matrices en tamaños de 2:

$$C_{11} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} X \begin{bmatrix} 1 & 4 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix} X \begin{bmatrix} 1 & 4 \\ 4 & 5 \end{bmatrix} \tag{8}$$

$$C_{12} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} X \begin{bmatrix} 5 & 1 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix} X \begin{bmatrix} 0 & 3 \\ 2 & 2 \end{bmatrix} \tag{9}$$

$$C_{21} = \begin{bmatrix} 5 & 1 \\ 2 & 3 \end{bmatrix} X \begin{bmatrix} 1 & 4 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} X \begin{bmatrix} 1 & 4 \\ 4 & 5 \end{bmatrix} \tag{10}$$

$$C_{22} = \begin{bmatrix} 5 & 1 \\ 2 & 3 \end{bmatrix} X \begin{bmatrix} 5 & 1 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} X \begin{bmatrix} 0 & 3 \\ 2 & 2 \end{bmatrix} \tag{11}$$

Ahora vamos a resolver por partes: $C_{11}, C_{12}, C_{21} y C_{22}$:

- C_{11}
  - 1.-

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} X \begin{bmatrix} 1 & 4 \\ 2 & 1 \end{bmatrix} = \tag{12}$$

$C_{11} = [1]X[1] + [2]X[2] = 5$
$C_{12} = [1]X[4] + [2]X[1] = 6$
$C_{21} = [3]X[1] + [4]X[2] = 11$
$C_{22} = [3]X[4] + [4]X[1] = 16$

  - 2.-

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix} X \begin{bmatrix} 1 & 4 \\ 4 & 5 \end{bmatrix} = \tag{13}$$

$C_{11} = [3]X[1] + [1]X[4] = 7$
$C_{12} = [3]X[4] + [1]X[5] = 17$
$C_{21} = [4]X[1] + [2]X[4] = 12$
$C_{22} = [4]X[4] + [2]X[5] = 26$

de modo que:

$$C_{11} = \begin{bmatrix} 5 & 6 \\ 11 & 16 \end{bmatrix} + \begin{bmatrix} 7 & 17 \\ 12 & 26 \end{bmatrix} = \begin{bmatrix} 12 & 23 \\ 23 & 42 \end{bmatrix} \tag{14}$$

- C_{12}
  - 1.-

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} X \begin{bmatrix} 5 & 1 \\ 3 & 2 \end{bmatrix} \tag{15}$$

$C_{11} = [1]X[5] + [2]X[3] = 11$
$C_{12} = [1]X[1] + [2]X[2] = 5$
$C_{21} = [3]X[5] + [4]X[3] = 27$
$C_{22} = [3]X[1] + [4]X[2] = 11$

- 2.-

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix} X \begin{bmatrix} 0 & 3 \\ 2 & 2 \end{bmatrix} \tag{16}$$

$C_{11} = [3]X[0] + [1]X[2] = 2$
$C_{12} = [3]X[3] + [1]X[2] = 11$
$C_{21} = [4]X[0] + [2]X[2] = 4$
$C_{22} = [4]X[3] + [2]X[2] = 16$

de modo que:

$$C_{12} = \begin{bmatrix} 11 & 5 \\ 27 & 11 \end{bmatrix} + \begin{bmatrix} 2 & 11 \\ 4 & 16 \end{bmatrix} = \begin{bmatrix} 13 & 16 \\ 31 & 27 \end{bmatrix} \tag{17}$$

- C_{21}
  - 1.-

$$\begin{bmatrix} 5 & 1 \\ 2 & 3 \end{bmatrix} X \begin{bmatrix} 1 & 4 \\ 2 & 1 \end{bmatrix} \tag{18}$$

$C_{11} = [5]X[1] + [1]X[2] = 7$
$C_{12} = [5]X[4] + [1]X[1] = 21$
$C_{21} = [2]X[1] + [3]X[2] = 8$
$C_{22} = [2]X[4] + [3]X[1] = 11$

  - 2.-

$$\begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} X \begin{bmatrix} 1 & 4 \\ 4 & 5 \end{bmatrix} \tag{19}$$

$C_{11} = [2]X[1] + [1]X[4] = 6$
$C_{12} = [2]X[4] + [1]X[5] = 13$
$C_{21} = [0]X[1] + [1]X[4] = 4$
$C_{22} = [0]X[4] + [1]X[5] = 5$

de modo que:

$$C_{21} = \begin{bmatrix} 7 & 21 \\ 8 & 11 \end{bmatrix} + \begin{bmatrix} 6 & 13 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 13 & 34 \\ 12 & 16 \end{bmatrix} \tag{20}$$

- C_{22}
  - 1.-

$$\begin{bmatrix} 5 & 1 \\ 2 & 3 \end{bmatrix} X \begin{bmatrix} 5 & 1 \\ 3 & 2 \end{bmatrix} \tag{21}$$

$C_{11} = [5]X[5] + [1]X[3] = 28$
$C_{12} = [5]X[1] + [1]X[2] = 7$
$C_{21} = [2]X[5] + [3]X[3] = 19$
$C_{22} = [2]X[1] + [3]X[2] = 8$

- 2.-

$$\begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} X \begin{bmatrix} 0 & 3 \\ 2 & 2 \end{bmatrix} \tag{22}$$

$C_{11} = [2]X[0] + [1]X[2] = 2$
$C_{12} = [2]X[3] + [1]X[2] = 8$
$C_{21} = [0]X[0] + [1]X[2] = 2$
$C_{22} = [0]X[3] + [1]X[2] = 2$

de modo que:

$$C_{22} = \begin{bmatrix} 28 & 7 \\ 19 & 8 \end{bmatrix} + \begin{bmatrix} 2 & 8 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 30 & 15 \\ 21 & 10 \end{bmatrix} \tag{23}$$

Así finalmente se tiene que

$$C = \begin{bmatrix} 12 & 23 & 13 & 16 \\ 23 & 42 & 31 & 27 \\ 13 & 34 & 30 & 15 \\ 12 & 16 & 21 & 10 \end{bmatrix} \tag{24}$$

5. Calculando el orden de complejidad de maxSubArray

Dando solución a la ecuación de recurrencia 25

$$T(n) = 2T(n/2) + \Theta(n) \tag{25}$$

Solución:

Aplicando divide y venceras se tiene que:

$n = 2^k$, $(k = logn)$, $f(n) = n$, $a = 2$, $b = 2$, luego

$$n^{log_b a} = n^{log_2 2} = n \tag{26}$$

y

$$\sum_{n=1}^{log_b n} \frac{f(b^j)}{a^j} = \sum_{n=1}^{log_2 n} \frac{f(2^j)}{2^j} = \sum_{n=1}^{log_2 n} \frac{2^j}{2^j} = \sum_{n=1}^{log_2 n} 1 = logn \tag{27}$$

multiplicando 26 y 27 se tiene:

$T(n) \in \Theta(nlogn)$

6. Demostrar mediante el método maestro que quickSort tiene orden de complejidad (nlogn)

   $T(n) = 2T(n/2) + cn$

   Utilizando el método maestro

   a=2, b=2, f(n)=cn y $n^{log_b a}$=n

   Puesto que

   f(n) = cn $\in \theta(n^{log_b a})$=$\theta(n)$

   Entonces por el Teorema maestro caso II:

   T(n) $\in \theta(n^{log_b a} logn)$ i.e

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