

Divide y conquista, y método maestro

Alejandro ANZOLA ÁVILA

2024-2

Algoritmos y Estructuras de Datos – Grupo 4

Quiz

1. Describa las tres características de la estrategia de divide y conquista
 - Divide (*Divide*)
 - Conquista (*Conquer*)
 - Combina (*Combine*)
2. Describa brevemente en que consiste el problema del máximo subarreglo.
3. ¿Qué es el método maestro? ¿Para que se usa?

Divide y conquista

Divide y conquista y estrategia incremental

La estrategia de *divide y conquista* consiste en dividir un problema en partes más simples, resolverlos y luego mezclar la solución.

Divide y conquista y estrategia incremental

La estrategia de *divide y conquista* consiste en **dividir**, un problema en partes mas simples, **resolverlos** y luego **mezclar** la solución.

Versus

La estrategia *incremental* resuelve un problema al construir progresivamente su solución (ej. **INSERTION-SORT** ordena desde 2 elementos hasta ordenar los n elementos de una lista).

Pasos de D&C

Se realizan tres pasos recursivamente:

1. **Divide** el problema en a subproblemas
2. **Conquista**/resuelve cada problema de tamaño n/b , incluso subdividiendo en más partes hasta que se llega a problemas triviales.
3. **Combina**/mezcla las a sub-soluciones, para resolver el problema inicial.

Ordenamiento

Entrada Una secuencia de n números $\langle a_1, a_2, \dots, a_n \rangle$

Salida Una permutación $\langle a'_1, a'_2, \dots, a'_n \rangle$, tal que $a'_1 \leq a'_2 \leq \dots \leq a'_n$

MERGE-SORT: Diseño

El algoritmo de MERGE-SORT aplica la estrategia de *divide y conquista*.

MERGE-SORT: Diseño

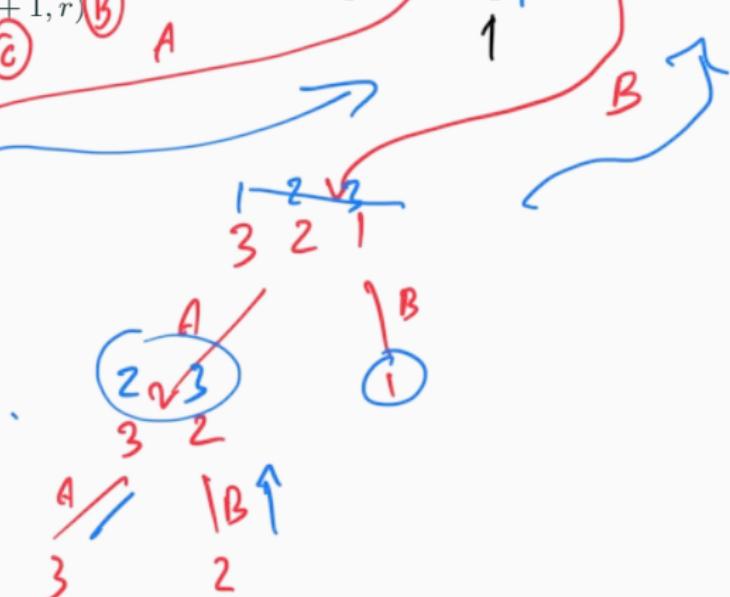
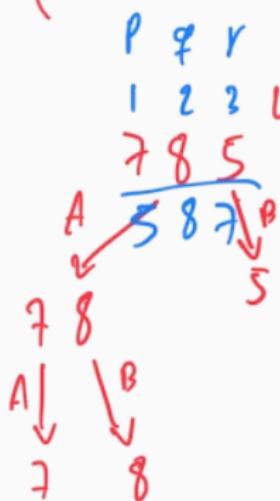
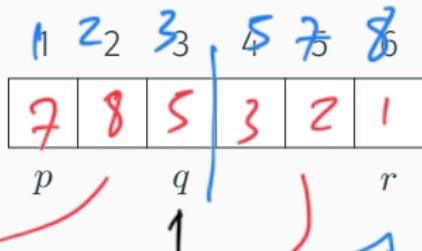
El algoritmo de MERGE-SORT aplica la estrategia de *divide y conquista*.

- Este *subdivide* su entrada $\langle A_1, \dots, A_n \rangle$ en subproblemas al dividirlo en $a = 2$ partes de tamaño n/b donde $b = 2$.
- Se subdividen hasta llegar a subproblemas de tamaño $n = 1$, donde la solución es trivial.
- Luego se *mezclan* las sub-soluciones hasta tener el arreglo ordenado.

Árbol de recursion para MERGE-SORT

MERGE-SORT(A, p, r) $\rightarrow 1, 6$

- 1 if $p < r$
- 2 $q = \lfloor (p + r)/2 \rfloor$
- 3 - MERGE-SORT(A, p, q) A
- 4 - MERGE-SORT($A, q + 1, r$) B
- 5 - MERGE(A, p, q, r) C



Combina para MERGE-SORT: MERGE

MERGE(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

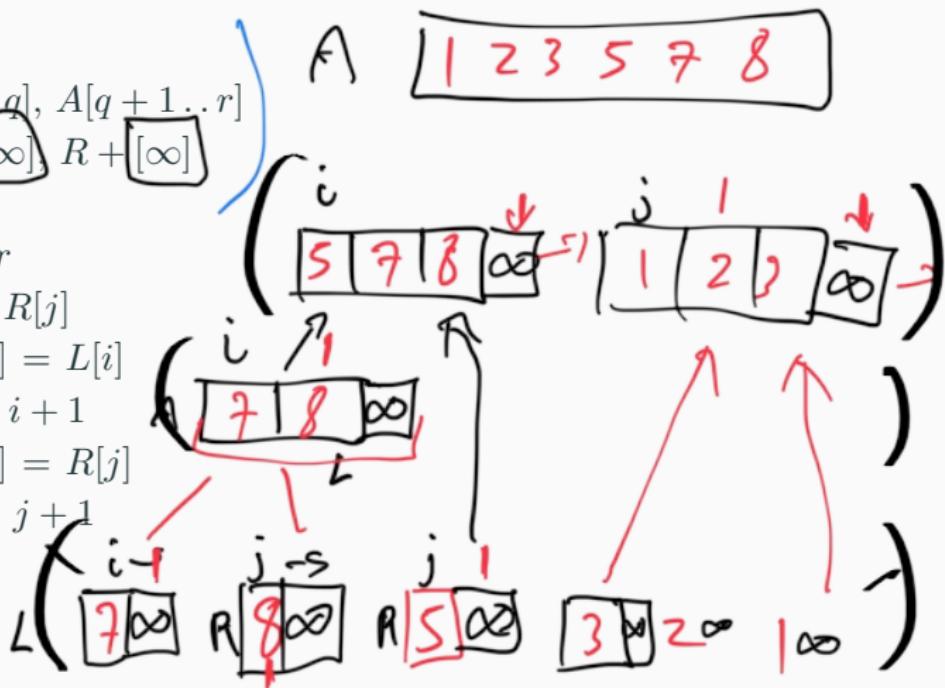
Árbol de recursion para MERGE-SORT: MERGE

MERGE(A, p, q, r)

```

1    $L, R = A[p..q], A[q+1..r]$ 
2    $L, R = L + [\infty], R + [\infty]$ 
3    $i, j = 1, 1$ 
4   for  $k = p$  to  $r$ 
5     if  $L[i] \leq R[j]$ 
6        $A[k] = L[i]$ 
7        $i = i + 1$ 
8     else  $A[k] = R[j]$ 
9        $j = j + 1$ 

```



Árbol de recursion para MERGE-SORT: MERGE

MERGE(A, p, q, r)

```
1   $L, R = A[p..q], A[q+1..r]$ 
2   $L, R = L + [\infty], R + [\infty]$ 
3   $i, j = 1, 1$ 
4  for  $k = p$  to  $r$ 
5    if  $L[i] \leq R[j]$ 
6       $A[k] = L[i]$ 
7       $i = i + 1$ 
8    else  $A[k] = R[j]$ 
9       $j = j + 1$ 
```

```
1  def merge( $A, p, q, r$ ):
2     $\text{infty} = \text{float}(\text{'inf'})$ 
3     $L = A[p : q+1]$ 
4     $R = A[q+1 : r+1]$ 
5     $L, R = L + [\text{infty}], R + [\text{infty}]$ 
6
7     $i, j = 0, 0$ 
8    for  $k$  in range( $p, r+1$ ):
9      if  $L[i] \leq R[j]$ :
10         $A[k] = L[i]$ 
11         $i += 1$ 
12      else:
13         $A[k] = R[j]$ 
14         $j += 1$ 
```

```

def merge_sort(A, p, r):↓
    if p < r:↓
        q = (p + r) // 2↓
        merge_sort(A, p, q)↓
        merge_sort(A, q + 1, r)↓
        merge(A, p, q, r)↓
    ↓
    ↓

def merge(A, p, q, r):↓
    infnty = float('inf')↓
    L = A[p : q + 1]↓
    R = A[q + 1 : r + 1]↓
    L, R = L + [infnty], R + [infnty]↓
    i, j = 0, 0↓
    for k in range(p, r + 1):↓
        if L[i] <= R[j]:↓
            A[k] = L[i]↓
            i += 1↓
        else:↓
            A[k] = R[j]↓
            j += 1↓
    ↓
    ↓

def main():↓
    A = [8, 3, 2, 6, 7, 1, 5, 4]↓
    print(A)↓
    merge_sort(A, 0, len(A) - 1)↓
    print(A)↓
    ↓
    ↓
main()↓

```

➤ python3 mergesort.py

```
[8, 3, 2, 6, 7, 1, 5, 4]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Tiempo de ejecución de MERGE-SORT

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ D(n) + aT(n/b) + C(n) & \text{si } n > 1 \end{cases}$$


Tiempo de ejecución de MERGE-SORT

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ D(n) + \underbrace{aT(n/b)}_{\vdots} + C(n) & \text{si } n > 1 \end{cases}$$

Divide $D(n) =$

Tiempo de ejecución de MERGE-SORT

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ D(n) + aT(n/b) + C(n) & \text{si } n > 1 \end{cases}$$

Divide $D(n) = \Theta(1)$ porque solo operamos con los índices p, r, q y esas operaciones son de tiempo constante.

Conquista $aT(n/b) =$

Tiempo de ejecución de MERGE-SORT

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ D(n) + aT(n/b) + C(n) & \text{si } n > 1 \end{cases}$$

Divide $D(n) = \Theta(1)$ porque solo operamos con los índices p, r, q y esas operaciones son de tiempo constante.

Conquista $aT(n/b) = 2T(n/2)$ porque dividimos en $a = 2$ subproblemas y cada uno sobre una entrada $b = 2$ veces mas pequeña ($1/b$).

Combina $C(n) =$

Tiempo de ejecución de MERGE-SORT

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ D(n) + aT(n/b) + C(n) & \text{si } n > 1 \end{cases}$$

Divide $D(n) = \Theta(1)$ porque solo operamos con los índices p, r, q y esas operaciones son de tiempo constante.

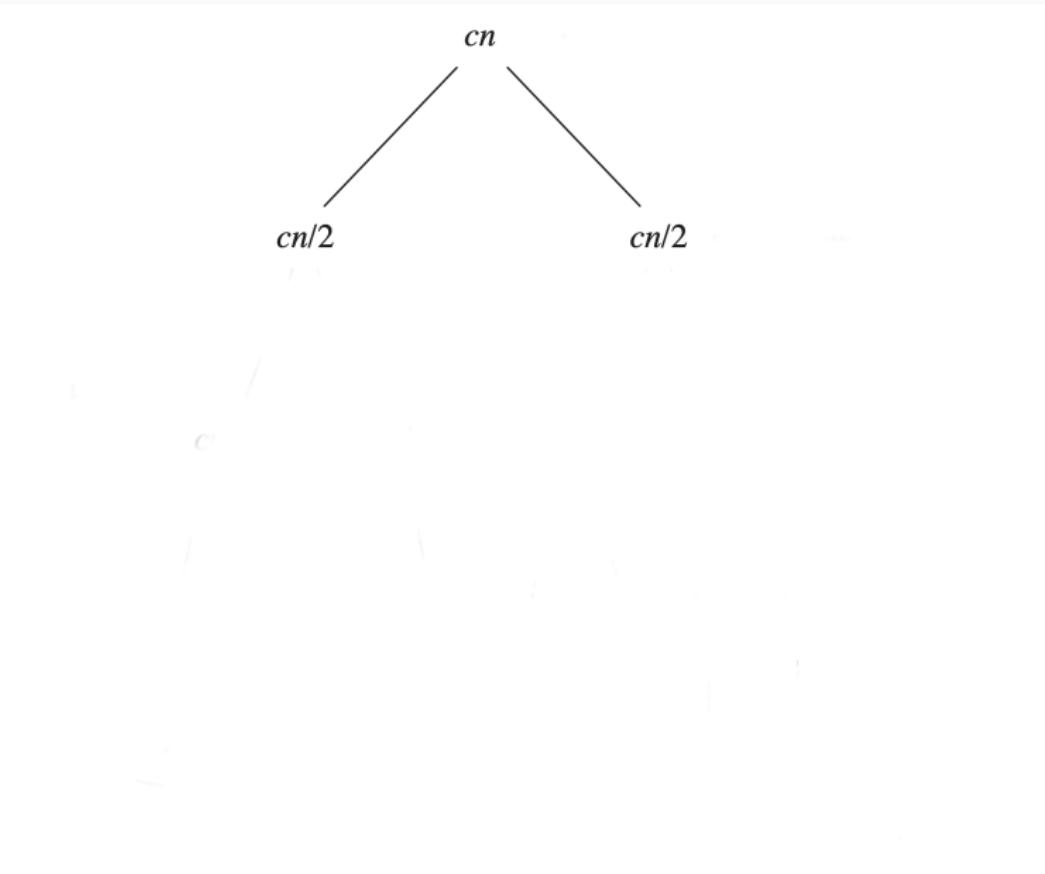
Conquista $aT(n/b) = 2T(n/2)$ porque dividimos en $a = 2$ subproblemas y cada uno sobre una entrada $b = 2$ veces mas pequeña ($1/b$).

Combina $C(n) = \Theta(n)$ porque al hacer MERGE se opera sobre todos los n elementos en cada nivel del árbol.

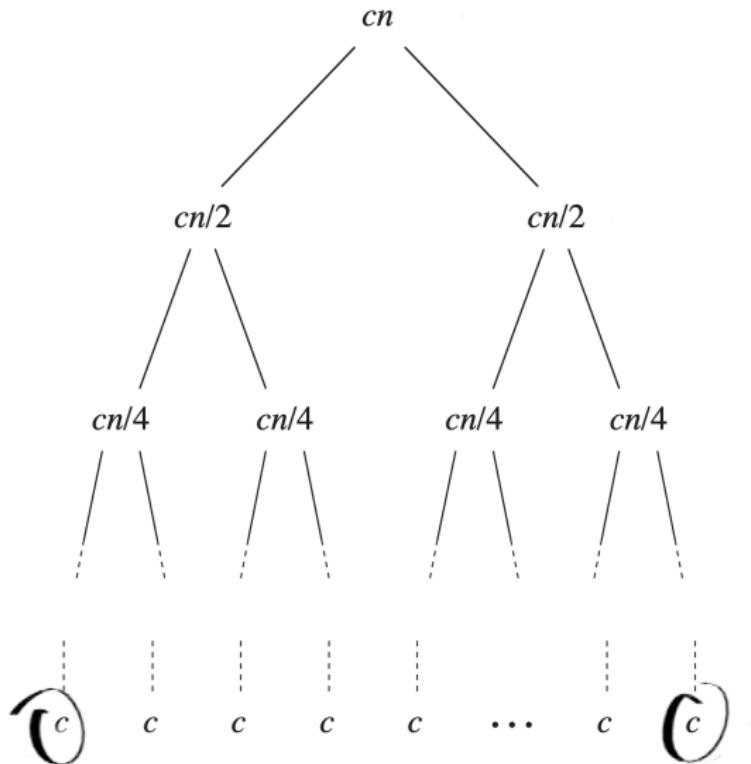
Tiempo de ejecución de MERGE-SORT

$$\begin{aligned}T(n) &= D(n) + aT(n/b) + C(n) \\&= \Theta(1) + 2T(n/2) + \Theta(n) \\&= \underbrace{2T(n/2) + \Theta(n)}_{\mathcal{O}(?)}\end{aligned}$$

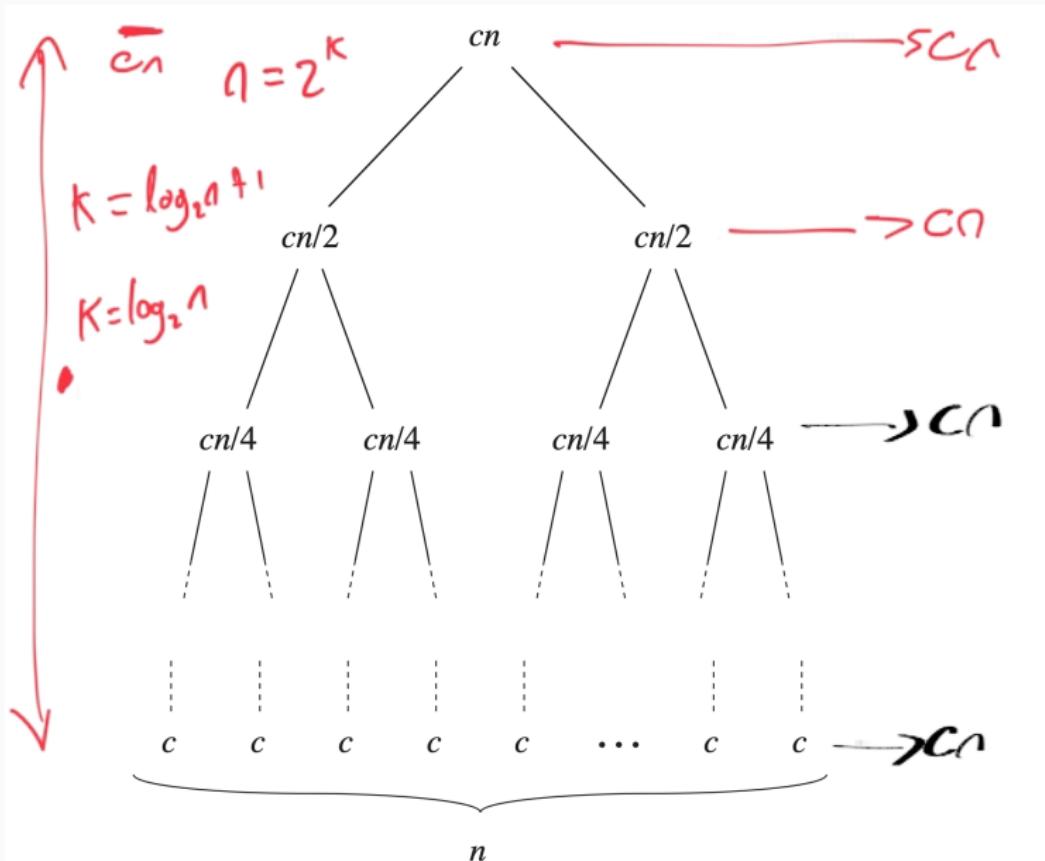
Visualización del tiempo de ejecución de MERGE-SORT



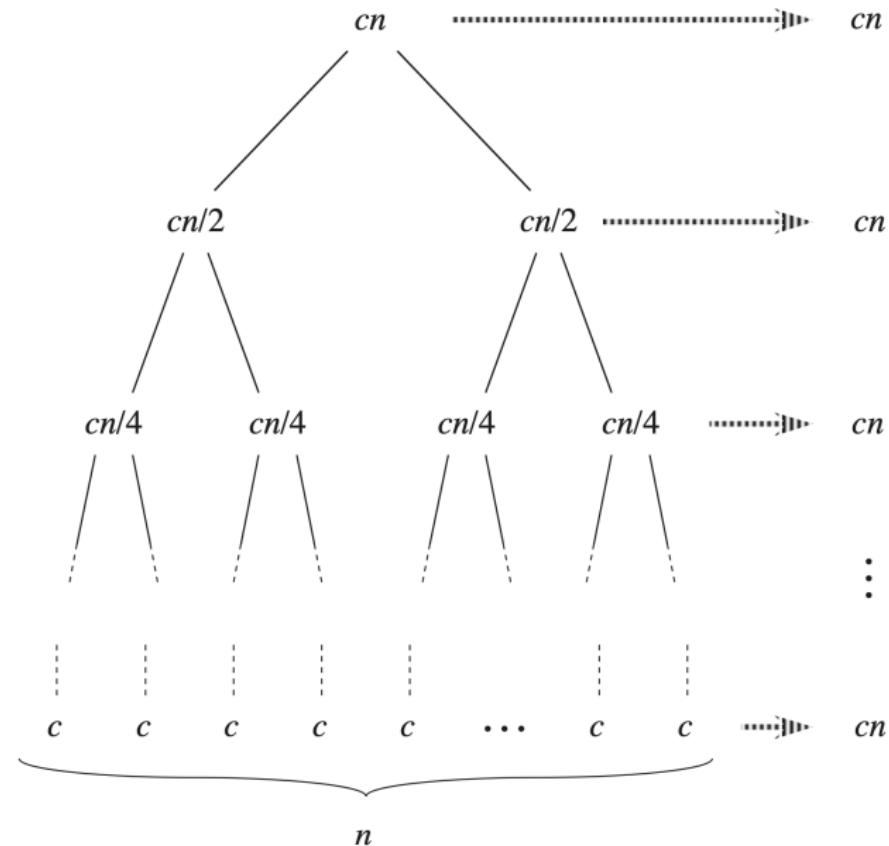
Visualización del tiempo de ejecución de MERGE-SORT



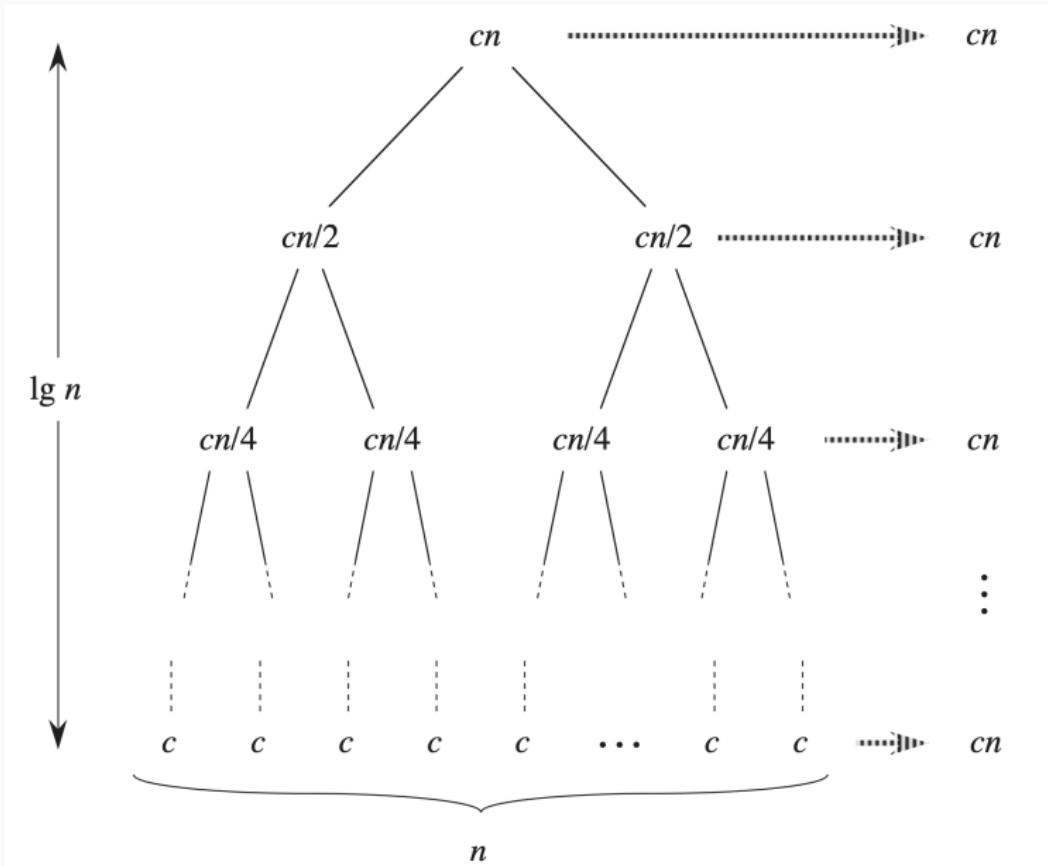
Visualización del tiempo de ejecución de MERGE-SORT



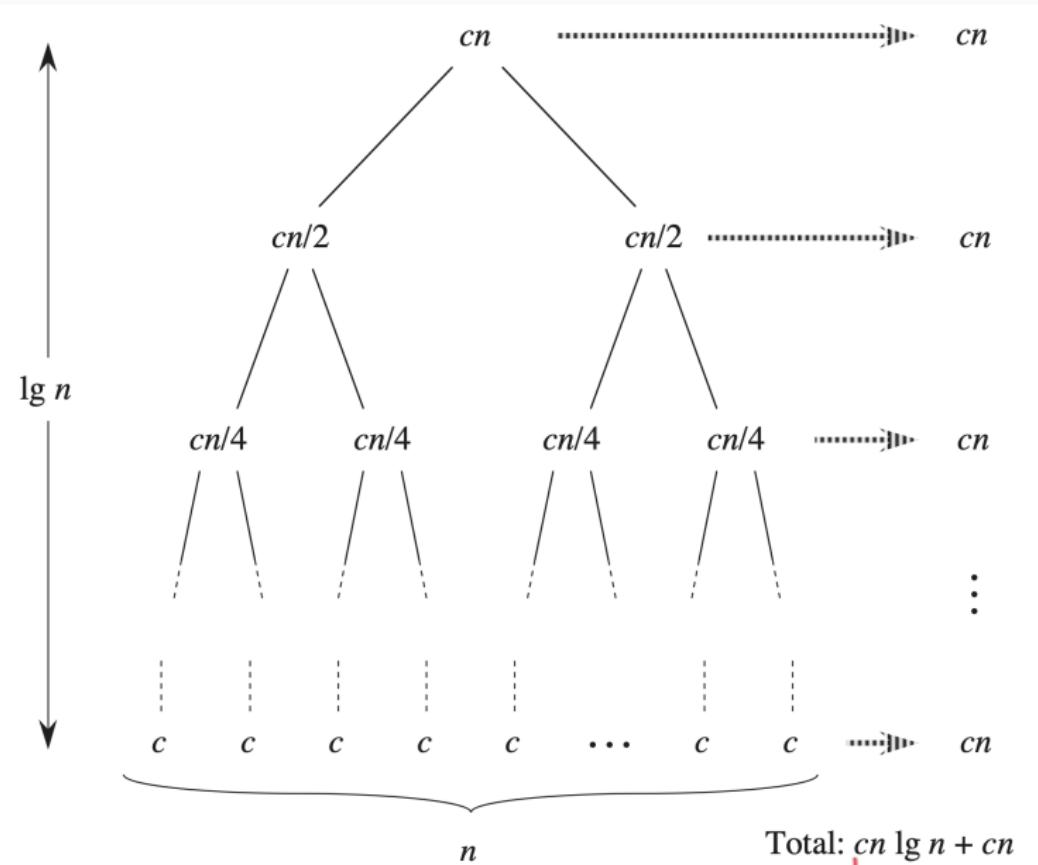
Visualización del tiempo de ejecución de MERGE-SORT



Visualización del tiempo de ejecución de MERGE-SORT



Visualización del tiempo de ejecución de MERGE-SORT



Método maestro

Método maestro

El *método maestro* nos provee de una receta para resolver recurrencias de la forma

$$T(n) = \underbrace{aT(n/b)}_{\downarrow} + f(n)$$

donde $a \geq 1$ y $b > 1$ son constantes y $f(n)$ es una función asintótica positiva.

Teorema maestro

Teorema maestro

Sean $a > 0$ y $b > 1$ constantes, y sea $f(n)$ una función asintótica positiva.

Sea también $T(n)$ definida por la recurrencia

$$T(n) = aT(n/b) + f(n)$$

Sea $\epsilon > 0$, $k \geq 0$, y $c < 1$.

#	Si	Entonces
1	$f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a} \log^k n)$	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3	$f(n) = \Omega(n^{\log_b a + \epsilon})$ y $af(n/b) \leq cf(n)$	$T(n) = \Theta(f(n))$

Error en Cormen, 3ra edición

En el libro no esta bien definido el caso #2.

Ejemplos

Sea $\epsilon > 0$, $k \geq 0$, y $c < 1$.

#	Si	Entonces
1	$f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a} \log^k n)$	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3	$f(n) = \Omega(n^{\log_b a + \epsilon})$ y $af(n/b) \leq cf(n)$	$T(n) = \Theta(f(n))$

$$T(n) = 9T(n/3) + n$$

$a=9$; $b=3$; $f(n)=n$ Caso 1

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

~~n^2~~ $f(n) = n^2$ $T(n) = \Theta(n^2)$

Ejemplos

Sea $\epsilon > 0$, $k \geq 0$, y $c < 1$.

#	Si	Entonces
1	$f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a} \log^k n)$	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3	$f(n) = \Omega(n^{\log_b a + \epsilon})$ y $a f(n/b) \leq c f(n)$	$T(n) = \Theta(f(n))$

Caso 2

$$q=1 ; b=\frac{3}{2} ; f(n)=\underset{n^0}{\cancel{1}} \quad T(n)=T(2n/3)+1$$
$$n^{\log_b a}=n^{\log_{3/2} 1}=n^0 \quad T(n)=\Theta(n^0 \cdot \log^{k+1} n)=\Theta(\log n)$$

Ejemplos

Sea $\epsilon > 0$, $k \geq 0$, y $c < 1$.

#	Si	Entonces
1	$f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a} \log^k n)$	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3	$f(n) = \Omega(n^{\log_b a + \epsilon})$ y $af(n/b) \leq cf(n)$	$T(n) = \Theta(f(n))$

$$a=3 \text{ ; } b=4 \text{ ; } f(n)=n \log n$$

$$T(n) = 3T(n/4) + n \log n$$

$$\log_4 3 = \log_2 3 / \log_2 4 = \log_2 3$$

$$f(n) = n^{\log_2 3 + \epsilon} = n$$

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

$$\log_2 3 < \log_2 4 < 1$$

$$\frac{1}{2} < \log_2 3 < 1$$

$$Caso 3$$

$$\begin{cases} af(n/b) \leq cf(n) \\ 3f(n/4) \leq cf(n) \\ 3 \cdot \left(\frac{n}{4} \log_2 \left(\frac{n}{4}\right)\right) \leq Cn \log n \end{cases}$$

Ejemplos

Sea $\epsilon > 0$, $k \geq 0$, y $c < 1$.

#	Si	Entonces
1	$f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a} \log^k n)$	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3	$f(n) = \Omega(n^{\log_b a + \epsilon})$ y $a f(n/b) \leq c f(n)$	$T(n) = \Theta(f(n))$

$a=7$; $b=2$; $f(n)=\Theta(n^2)$

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$n^{\log_2 7}$ vs Cn^2

$$n^{\log_2 7 - \epsilon} = n^2$$

$\log_2 7 < \log_2 8$

$\frac{1}{2} < \log_2 7 < \frac{1}{3}$

$T(n) = \Theta(n^{\log_2 7})$

Subarreglo máximo

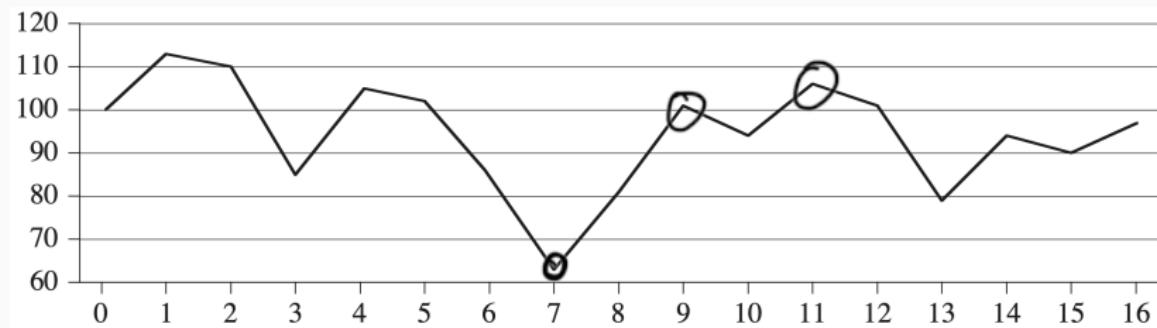
Formulación

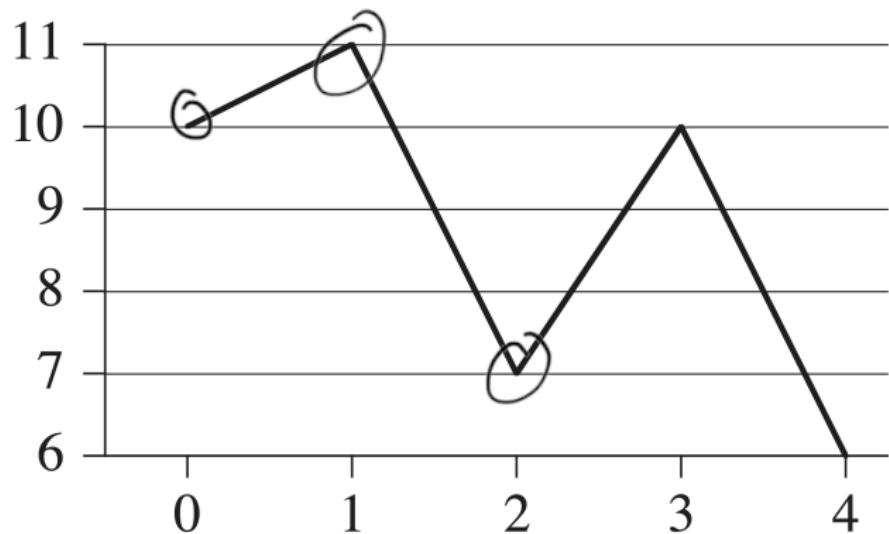
Supongamos que tenemos la oportunidad de entrar en el mercado bursátil.

Queremos:

- Comprar cuando la acción este mas **barata**
- Vender cuando la acción este mas **cara**

También supongamos que sabemos como estará la acción en el futuro:





Tal vez es posible maximizar la ganancia si se compra al **menor** precio o si se vende al **mayor** precio.

Fuerza bruta

Podemos probar todas las $\binom{n}{2}$ posibilidades para comprar y vender, donde la fecha de compra sea antes de la fecha de venta.

$$(i, j) = (j, i)$$

Fuerza bruta

Podemos probar todas las $\binom{n}{2}$ posibilidades para comprar y vender, donde la fecha de compra sea antes de la fecha de venta.

Esta estrategia nos costaría

$$\begin{aligned}\binom{n}{2} &= \frac{n!}{2!(n-2)!} \\ &= \frac{n(n-1)(n-2)!}{2(n-2)!} \\ &= \frac{n(n-1)}{2} \\ &= \Theta(n^2)\end{aligned}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Dividir y conquistar: Formulación

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change	X	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Podemos buscar una secuencia de días sobre los que el cambio neto desde el primer día hasta el ultimo es *máximo*.

En vez de pensar en los precios individuales, consideraremos el **cambio** diario en el precio.

$$\text{Precios} = \langle p_1, \dots, p_n \rangle$$

$$\text{Cambios} = \langle p_2 - p_1, p_3 - p_2, \dots, p_n - p_{n-1} \rangle$$

$$\|\text{Cambios}\| = n - 1$$

Dividir y conquistar: Diseño

Aplicamos dividir y conquistar por medio de subdividir el problema en $a = 2$ subarreglos.

Dividir y conquistar: Diseño

Aplicamos dividir y conquistar por medio de subdividir el problema en $a = 2$ subarreglos.

1. Supóngase que queremos encontrar el máximo subarreglo en $A[low .. high]$.



Dividir y conquistar: Diseño

Aplicamos dividir y conquistar por medio de subdividir el problema en $a = 2$ subarreglos.

1. Supóngase que queremos encontrar el máximo subarreglo en $A[low .. high]$.
2. Encontramos un punto medio $mid = \lfloor \frac{n}{2} \rfloor$ y tenemos los dos arreglos $A[low .. mid]$ y $A[mid + 1 .. high]$



Dividir y conquistar: Diseño

Aplicamos dividir y conquistar por medio de subdividir el problema en $a = 2$ subarreglos.

1. Supóngase que queremos encontrar el máximo subarreglo en $A[low .. high]$.
2. Encontramos un punto medio $mid = \lfloor \frac{n}{2} \rfloor$ y tenemos los dos arreglos $A[low .. mid]$ y $A[mid + 1 .. high]$

Existen **tres** casos para encontrar el máximo subarreglo.

Dividir y conquistar: FIND-MAX-SUBARRAY



El máximo subarreglo $A[i..j]$ debe estar:

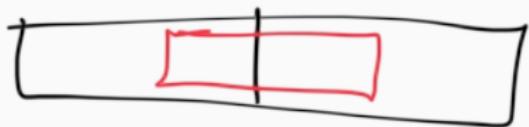
1. Enteramente en $A[low..mid]$ tal que $low \leq i \leq j \leq mid$

El máximo subarreglo $A[i..j]$ debe estar:

1. Enteramente en $A[low..mid]$ tal que $low \leq i \leq j \leq mid$
2. Enteramente en $A[mid + 1..high]$ tal que $mid < i \leq j \leq high$

El máximo subarreglo $A[i..j]$ debe estar:

1. Enteramente en $A[low..mid]$ tal que $low \leq i \leq j \leq mid$
2. Enteramente en $A[mid + 1..high]$ tal que $mid < i \leq j \leq high$
3. Cruzando el punto medio mid tal que $low \leq i \leq mid < j \leq high$



Algoritmo de FIND-MAX-CROSSING-SUBARRAY

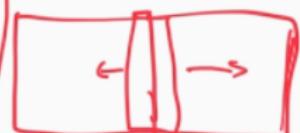
FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)

```
1  left-sum = -∞ →  
2  sum = 0 ←  
3  for  $i = mid$  downto  $low$   $i=mid \rightarrow low$   
4      sum = sum +  $A[i]$  ]  $\Theta(1) (mid-low)$   
5      if sum > left-sum  
6          left-sum = sum  
7          max-left = i  
8  right-sum = -∞ ←  
9  sum = 0  
10 for  $j = mid + 1$  to  $high$   $j=mid+1 \rightarrow high$   
11     sum = sum +  $A[j]$   $\Theta(1) (high-(mid+1))$   
12     if sum > right-sum  
13         right-sum = sum  
14         max-right = j  
15 return (max-left, max-right, left-sum + right-sum)
```



$\Theta(1)$

¿Complejidad Θ ?



$high - low$

Algoritmo de FIND-MAX-SUBARRAY

FIND-MAXIMUM-SUBARRAY($A, low, high$)

```
1  if  $high == low$ 
2      return ( $low, high, A[low]$ )           // base case: only one element
3  else  $mid = \lfloor (low + high)/2 \rfloor$ 
4      ( $left-low, left-high, left-sum$ ) =
5          { FIND-MAXIMUM-SUBARRAY( $A, low, mid$ ) } A
6      ( $right-low, right-high, right-sum$ ) =
7          { FIND-MAXIMUM-SUBARRAY( $A, mid + 1, high$ ) } B
8      ( $cross-low, cross-high, cross-sum$ ) =
9          { FIND-MAX-CROSSING-SUBARRAY( $A, low, mid, high$ ) } C
10     if  $left-sum \geq right-sum$  and  $left-sum \geq cross-sum$ 
11         return ( $left-low, left-high, left-sum$ )
12     elseif  $right-sum \geq left-sum$  and  $right-sum \geq cross-sum$ 
13         return ( $right-low, right-high, right-sum$ )
14     else return ( $cross-low, cross-high, cross-sum$ )
```

$$T(n) = O(1) + \alpha T(n/2) + O(1) + O(n)$$

$$2T(n/2)$$

¿Complejidad Θ ?

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

Búsqueda

Búsqueda

Entrada Una secuencia de n números $\langle a_1, a_2, \dots, a_n \rangle$ ordenados $a_1 \leq a_2 \leq \dots \leq a_n$, y un número k .

Salida Posición $i \in [1 : n]$ tal que $a_i = k$. Si no existe, entonces $i = -1$.

Búsqueda secuencial

LINEAR-SEARCH(A, k)

```
1   $i = 1$ 
2  while  $i < A.length$            ¿Complejidad  $\Theta$ ?
3      if  $A[i] == k$ 
4          return  $i$ 
5       $i = i + 1$ 
6  return  $-1$ 
```

Búsqueda secuencial

LINEAR-SEARCH(A, k)

```
1   $i = 1$ 
2  while  $i < A.length$ 
3      if  $A[i] == k$ 
4          return  $i$ 
5       $i = i + 1$ 
6  return  $-1$ 
```

```
1  def linear_search(A, k):
2      i = 0
3      while i < len(A):
4          if A[i] == k:
5              return i
6          i += 1
7      return -1
```

Búsqueda binaria

Ya que sabemos que la secuencia esta ordenada, podemos optimizarlo.

`BINARY-SEARCH(A, k)`

```
1   $low, high = 1, A.length$ 
2  while  $low < high$ 
3       $mid = \lfloor (low + high)/2 \rfloor$ 
4      if  $A[mid] == k$ 
5          return  $mid$ 
6      if  $A[mid] < k$ 
7           $low = mid + 1$ 
8      else  $high = mid$ 
```

Búsqueda binaria

Ya que sabemos que la secuencia esta ordenada, podemos optimizarlo.

BINARY-SEARCH(A, k)

```
1  low, high = 1, A.length
2  while low < high
3      mid = ⌊(low + high)/2⌋          ¿Complejidad  $\Theta$ ?
4      if A[mid] == k                  ¿Recursivo vs Iterativo?
5          return mid
6      if A[mid] < k
7          low = mid + 1
8      else high = mid
```

Búsqueda binaria

Ya que sabemos que la secuencia esta ordenada, podemos optimizarlo.

BINARY-SEARCH(A, k)

```
1  low, high = 1, A.length
2  while low < high
3      mid = ⌊(low + high)/2⌋
4      if A[mid] == k
5          return mid
6      if A[mid] < k
7          low = mid + 1
8      else high = mid
```

```
1  def binary_search(A, k):
2      low, high = 0, len(A)-1
3      while low < high:
4          mid = (low + high) // 2
5          if A[mid] == k:
6              return mid
7          if A[mid] < k:
8              low = mid+1
9          else:
10             high = mid
11
12     return -1
```

Ejercicios

Ejercicios

1. Diseñe un algoritmo D&Q para calcular el exponente de un numero a^n .
2. Use el método maestro para calcular los siguientes limites asintóticos:
 - a. $T(n) = 2T(n/4) + 1$
 - b. $T(n) = 2T(n/4) + \sqrt{n}$
 - c. $T(n) = 2T(n/4) + \sqrt{n} \log^2 n$
 - d. $T(n) = 2T(n/4) + n$
 - e. $T(n) = 2T(n/4) + n^2$
3. Encuentre la formula de recursion $T(n)$ del algoritmo de FIND-MAX-SUBARRAY, y calcule su limite asintótico Θ con el método maestro.
4. ★ ¿Es $(x + y)^2 = O(x^2 + y^2)$?