Analisis y diseño de algoritmos. División y Conquista: parte 2

Juan Gutiérrez

September 2019

1 Problema del subarreglo máximo

- Entrada: vector $A[1..n]$ de números enteros (no necesariamente positivos).
- Salida: índices i, j tal que la suma de los elementos en $A[i..j]$ es la máxima posible

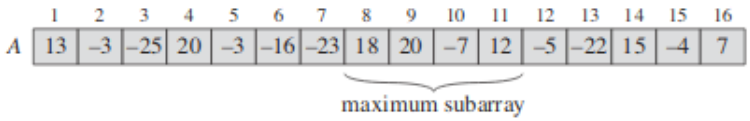

Figure 1: Tomada del libro Cormen, Introduction to Algorithms

Solución de fuerza bruta: probar todas las posibles subsecuencias. Como existen $\binom{n}{2}$ posibilidades, el algoritmo es $\Theta(n^2)$.

Podemos hacer algo mejor?

Note que, dado un arreglo $A[low..high]$, un subarreglo de suma máxima tiene tres posibilidades. El está

- Enteramente en $A[low..mid]$
- Enteramente en $A[mid + 1..high]$
- Con una parte en $A[low..mid]$ y la otra en $A[mid + 1..high]$

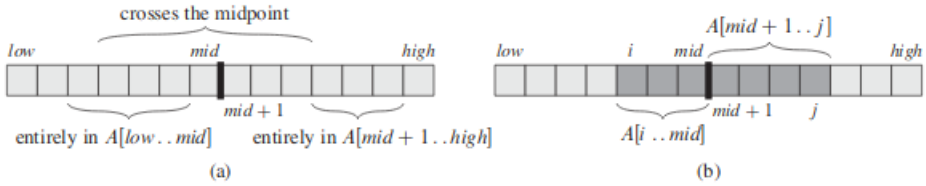

Figure 2: Tomada del libro Cormen, Introduction to Algorithms

Esta observación nos permite diseñar un algoritmo de división y conquista para el problema.

- **Dividir** Dividimos A en dos subarreglos de tamaño $(high - low + 1)/2$
- **Conquistar**: En cada subarreglo encontramos el correspondiente subarreglo máximo.
- **Combinar** Hallamos cual de los dos subarreglos de las llamadas recursivas tienen suma máxima.

  Después debemos comparar esta suma con un tercer candidato, que es el arreglo de suma máxima con una parte en $A[low..mid]$ y la otra en $A[mid + 1..high]$

A continuación veremos como hallar este candidato mencionado en la operación de "Combinar".

Recibe: Un arreglo A de números enteros, y tres índices $low \leq mid < high$.
Devuelve: Tres enteros $i, j, \sum_{k=i}^{j} A[k]$ tales que $\sum_{k=i}^{j} A[k]$ tiene valor máximo para todos los i, j que cumplen que $i \leq mid < j$

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 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
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

Figure 3: Tomada del libro Cormen, Introduction to Algorithms

Tiempo de ejecución:
$T(n) = c_1 + (low - mid + 1)c_2 + (high - mid)c_3 = c_1 + c_4(low - mid + 1) = c_1 + 4n = \Theta(n)$
Ahora analizaremos el algoritmo principal

Recibe: Un arreglo de números enteros $A[low..high]$.

Devuelve: Tres enteros $i, j, \sum_{k=i}^{j} A[k]$ tales que $\sum_{k=i}^{j} A[k]$ tiene valor máximo para todos los i, j

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

Figure 4: Tomada del libro Cormen, Introduction to Algorithms

Tiempo de ejecución de FIND-MAXIMUM-SUBARRAY. Cuando $n = 1$, se ejecutan las lineas 1 y 2: tiempo $c_1 + c_2$. Cuando $n > 1$, tenemos $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_3 + c_7 + c_8 + c_9 + c_{10} + c_{11} + k_1 n$

$$T(n) = \begin{cases} c & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + k_1 n + k_2 & \text{caso contrário} \end{cases}$$

Portanto $T(n) = \Theta(n \lg n)$. (Para ver esto, aplique teorema maestro, ya que cuando n es potencia de 2, tenemos que $T(n) = 2T(n/2) + kn$), tenemos que $a = b = 2, k = 1$, segundo caso de teorema maestro)

2 Multiplicación de números naturales

- Entrada: Dos números naturales a y b de n dígitos
- Salida: El producto $a \cdot b$

El algoritmo usual:

```
    9999     A
    7777     B
   -----
   69993     C
  69993      D
 69993       E
69993        F
-----------
77762223     G
```

En la figura anterior, $A \cdot B = C + D + E + F = G$. Tiempo de ejecución es proporcional a $4 + 4 + 4 + 4 = 16 = 4^2$.

Este es un algoritmo simple que puede ser descrito de la siguiente manera:

Recibe: Dos números enteros representados por $a[1..n], b[1..n]$

Devuelve: el producto $a \cdot b$

MULTIPLICACION-BASICA(a, b, n)	*cost*	*times*
1: total $= 0$	c_1	1
2: **for** $j = 1$ **to** n	c_2	$n + 1$
3: sum $= 0$	c_3	n
4: **for** $i = 1$ **to** n	c_4	$(n + 1) \cdot n$
5: sum $=$ sum $+ b[j] \cdot a[i]$	c_5	$n \cdot n$
6: total $=$ total $\cdot 10 +$ sum	c_6	n
7: **return** total	c_7	1

Es claro que el tiempo de ejecución es $\Theta(n^2)$. Veremos como mejorar este algoritmo.

2.1 Un algoritmo de división y conquista

Note que, dados dos números naturales a y b con n dígitos, podemos expresarlos como

$$a = a_1 \cdot 10^m + a_2,$$

$$b = b_1 \cdot 10^m + b_2.$$

Donde $m = \lceil n/2 \rceil$.

Por ejemplo, $3213209842 = 32132 \cdot 10^5 + 09842$ o $953421412 = 9534 \cdot 10^5 + 21412$.

Por lo tanto

$$\begin{aligned} a \cdot b &= (a_1 \cdot 10^m + a_2) \cdot (b_1 \cdot 10^m + b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + (a_1 b_2 + a_2 b_1) \cdot 10^m + (a_2 b_2) \end{aligned}$$

De esta manera, podemos dividir nuestro problema original en cuatro subproblemas: multiplicar a_1 con b_1, multiplicar a_1 con b_2, multiplicar a_2 con b_1 y multiplicar a_2 con b_2.

Obtenemos el siguiente algoritmo de división y conquista:

Recibe: Dos números enteros a y b de $n-1$ o n dígitos

Devuelve: el producto $a \cdot b$

MULTIPLICACION-DC (a, b, n)	*cost*	*times*
1: **if** $n = 1$	c_1	1
2: **return** $a \cdot b$	c_2	1
3: $m = \lceil n/2 \rceil$	$c_3 n$	1
4: $a_1 = \lfloor a/10^m \rfloor$	$c_4 n$	1
5: $a_2 = a \mod 10^m$	$c_5 n$	1
6: $b_1 = \lfloor b/10^m \rfloor$	$c_6 n$	1
7: $b_2 = b \mod 10^m$	$c_7 n$	1
8: $a_1 b_1 =$ MULTIPLICACION-DC(a_1, b_1, m)	$T(m)$	1
9: $a_1 b_2 =$ MULTIPLICACION-DC(a_1, b_2, m)	$T(m)$	1
10: $a_2 b_1 =$ MULTIPLICACION-DC(a_2, b_1, m)	$T(m)$	1
11: $a_2 b_2 =$ MULTIPLICACION-DC(a_2, b_2, m)	$T(m)$	1
12: **return** $a_1 b_1 \cdot 10^{2m} + (a_1 b_2 + a_2 b_1) \cdot 10^m + a_2 b_2$	$c_{12} n$	1

Tiempo de ejecución:

$$T(n) = \begin{cases} k_1 & n = 1 \\ 4T(\lceil n/2 \rceil) + k_2 n & \text{caso contrário} \end{cases}$$

Por Teorema Maestro, $\lg 4/lg2 = 2 > 1$. Luego $T(n) = \Theta(n^{\lg 4/lg2}) = \Theta(n^2)$.

Vemos que MULTIPLICACION-DC no mejora MULTIPLICACION-BASICA.

2.2 Algoritmo de Karatsuba

Este algoritmo mejorará la recurrencia anterior haciendo únicamente 3 llamadas recursivas. Por lo tanto tendremos un tiempo de ejecución de $\Theta(n^{\lg 3/\lg 2}) = \Theta(n^{1.59})$.

La observación principal es la siguiente.

Dados dos números naturales a y b con n dígitos, los expresamos como en la subsección anterior:

$$a = a_1 \cdot 10^m + a_2,$$

$$b = b_1 \cdot 10^m + b_2.$$

Donde $m = \lceil n/2 \rceil$.

Tenemos

$$\begin{aligned} a \cdot b &= (a_1 \cdot 10^m + a_2) \cdot (b_1 \cdot 10^m + b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + (a_1 b_2 + a_2 b_1) \cdot 10^m + (a_2 b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + ((a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2) \cdot 10^m + (a_2 b_2) \end{aligned}$$

De esta manera, solo debemos calcular tres productos: $a_1 b_1$, $a_2 b_2$ y $(a_1 + a_2)(b_1 + b_2)$. El producto $a_1 b_2 + a_2 b_1$ puede ser calculado por $(a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2$.

Recibe: Dos números enteros de $n-1$ o n dígitos
Devuelve: el producto $a \cdot b$

KARATSUBA (a, b, n)	*cost*	*times*
1: **if** $n \leq 3$	c_1	1
2: **return** $a \cdot b$	c_2	1
3: $m = \lceil n/2 \rceil$	$c_3 n$	1
4: $a_1 = \lfloor a/10^m \rfloor$	$c_4 n$	1
5: $a_2 = a \mod 10^m$	$c_5 n$	1
6: $b_1 = \lfloor b/10^m \rfloor$	$c_6 n$	1
7: $b_2 = b \mod 10^m$	$c_7 n$	1
8: $a_1 b_1 =$ KARATSUBA(a_1, b_1, m)	$T(m)$	1
9: $a_2 b_2 =$ KARATSUBA(a_2, b_2, m)	$T(m)$	1
10: $a_1 a_2 b_1 b_2 =$ KARATSUBA$(a_1 + a_2, b_1 + b_2, m + 1)$	$T(m+1)$	1
11: **return** $a_1 b_1 \cdot 10^{2m} + (a_1 a_2 b_1 b_2 - a_1 b_1 - a_2 b_2) \cdot 10^m + a_2 b_2$	$c_{12} n$	1

Tiempo de ejecución:

$$T(n) = \begin{cases} k_1 & n = 1 \\ 2T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil + 1) + k_2 n & \text{caso contrário} \end{cases}$$

Por Teorema Maestro, $\lg 4/lg3 = 2 > 1$. Luego $T(n) = \Theta(n^{\lg 4/lg3}) = \Theta(n^{1.59\ldots})$.

3 Contar inversiones

- Entrada: vector $A[1..n]$ de números enteros diferentes
- Salida: Número de inversiones. Donde una *inversión* es un par (i, j) tal que $i < j$ y $A[i] > A[j]$

Ejemplo, sea $A = [2, 4, 1, 3, 5]$. Las inversiones son $(1, 3), (2, 3), (2, 4)$

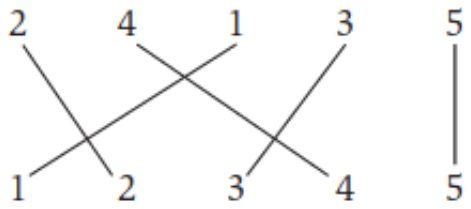

Figure 5: Tomada del libro Kleinberg-Tardos, Algorithm Design

Algoritmo ingenuo: Evaluar todos los posibles pares ordenados y verificar si son inversiones.

Recibe: Un vector de números enteros diferentes $A[1..n]$

Devuelve: El número de inversiones en A.

INVERSIONES-INGENUO(A, n)	*cost*	*times*
1: total = 0	c_1	1
2: **for** $i = 1$ **to** $n - 1$	c_2	n
3: **for** $j = i + 1$ **to** n	c_3	$\sum_{i=1}^{n-1} n - i + 1$
4: **if** $A[i] > A[j]$	c_4	$\sum_{i=1}^{n-1} n - i$
5: total = total + 1	c_5	$\sum_{i=1}^{n-1} n - i$
6: **return** total	c_6	1

Este algoritmo, tanto en el peor caso y mejor caso consume tiempo $\Theta(n^2)$. (Un ejemplo de peor caso ocurre si el vector está ordenado de manera decreciente). A continuación veremos un algoritmo de división y conquista para este problema.

3.1 División y Conquista

Observamos que, dado el vector $A[1..n]$, una inversión (i, j) puede estar en exactamente una de estas posibilidades:

- $i, j \in \{1, \ldots, \lfloor n/2 \rfloor\}$
- $i, j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$
- $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$, $j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$

Eso nos da la idea de un algoritmo de división y conquista.

- **Dividir** Dividimos A en dos subarreglos de tamaños $\lfloor n/2 \rfloor, \lceil n/2 \rceil$
- **Conquistar**: En cada subarreglo encontramos el número de inversiones
- **Combinar** Sumamos dicho número con el número de inversiones que tienen un elemento antes o igual de $\lfloor n/2 \rfloor$ y el otro despues de $\lfloor n/2 \rfloor$. Eso nos daria el total de inversiones

Vemos entonces, informalmente que $T(n) = 2T(n/2) + C(n)$, donde $C(n)$ es el tiempo para combinar. Si queremos que la recurrencia sea $\Theta(n \lg n)$ entonces $C(n)$ debe ser $\Theta(n)$.

Pero si intentamos contar los (i, j) tales que $1 \leq i \leq \lfloor n/2 \rfloor$ y $\lfloor n/2 \rfloor + 1 \leq j \leq n$ con fuerza bruta, podemos gastar tiempo proporcional a $n/2 \cdot n/2 = \Theta(n^2)$. Necesitamos un algoritmo más eficiente.

Note que si los subarreglos $A[1..\lfloor n/2 \rfloor]$ y $A[\lfloor n/2 \rfloor + 1 \ldots n]$ ya estuviesen ordenados, entonces solo gastariamos tiempo $\Theta(n)$ por la siguiente observación:

Si (i, j) es una inversión con $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$ y $j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$
entonces (i', j) también es una inversión para todo $i' > i$ (1)

Para probar (1), basta observar que $A[i] > A[j]$ porque (i, j) es una inversión y que $A[i'] > A[i]$ porque A está ordenado. El siguiente subprograma se encarga de hacer este conteo. Nos recuerda a la función Merge del Mergesort. Luego, el siguiente subprograma haría lo pedido.

Recibe: Un vector de números enteros diferentes $A[1..n]$ y tres índices p, q, r tales que $A[p..q]$ y $A[q+1..r]$ están ordenados.

Devuelve: El número de inversiones (i, j) en A tales que $i \in \{p, \ldots, q\}$, $j \in \{q+1, \ldots, r\}$. Además, ordena el vector $A[p..r]$.

INVERSIONES-CENTRADAS(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] = ∞
 9: L[n_2 + 1] = ∞
10: i = 1
11: j = 1
12: total = 0
13: for k = p to r
14:     if L[i] ≤ R[j]
15:         A[k] = L[i]
16:         i = i + 1
17:         total = total + j - 1
18:     else
19:         A[k] = R[j]
20:         j = j + 1
21: return total
```

Note que INVERSIONES-CENTRADAS es escencialmente la subrutina Merge del Mergesort, la cual consume tiempo $\Theta(n)$.

Finalmente, con la ayuda de esta subrutina, diseñamos nuestro algoritmo principal.

Recibe: Un vector de números enteros diferentes $A[p..r]$

Devuelve: El número de inversiones en A.

INVERSIONES-DC(A, p, r)	*cost*	*times*
1: **if** $(p == r)$	c_1	*1*
2: **return** 0	c_2	*0*
3: $q = \lfloor \frac{r-p+1}{2} \rfloor$	c_3	1
4: $total_1 =$ INVERSIONES-DC(A, p, q)	$T(\lfloor n/2 \rfloor)$	1
5: $total_2 =$ INVERSIONES-DC$(A, q+1, r)$	$T(\lceil n/2 \rceil)$	1
6: $total_3 =$ INVERSIONES-CENTRADAS(A, p, q, r)	kn	1
7: **return** $total_1$+$total_2$+$total_3$	c_5	1

Note que $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + kn$. Luego, por teorema maestro, $T(n) = \Theta(n \lg n)$.