Tema 2. Divide y vencerás

Grado en Ingeniería Informática en Tecnologías de la Información

Departamento de Ingeniería de Sistemas Informáticos y Telemáticos

Área de Lenguajes y Sistemas Informáticos

Índice

Preliminares

- Divide y Vencerás es la técnica de diseño algorítmico más simple y conocida.
- Es una técnica básica tanto desde el punto de vista de la recursividad, como desde el de la estrategia de diseño descendente.
- Existen algunos algoritmos sumamente eficientes cuya estructura se ajusta a la estrategia de Divide y Vencerás.

El Todo y las Partes

Sea lo que sea aquello por lo que se reza, siempre se pide un milagro. Cualquier plegaria se reduce a esto - *Señor, concédeme que dos veces dos no sea cuatro*.

Iván Turgenev (1818{1883), novelista ruso

Un Ejemplo Ilustrativo (pero no práctico)

- Queremos sumar los valores almacenados en un array de *n* posiciones. El enfoque de fuerza bruta es hacer simplemente:

```
func Suma (↓A: array[1..n] de ℕ):ℕ
variables i, s: ℕ
inicio
   s ← A[1]
   para i ← 2 hasta n hacer
      s ← s+A[i]
   finpara
   devolver s
fin
```

- ¿Podemos hacerlo de otro modo?

Un Ejemplo Ilustrativo (pero no práctico)

- Si $n = 1$ la suma es trivial, y si $n > 1$ dividimos el problema en dos instancias del mismo problema, pero de menor tamaño:

```
func Suma (↓A: array[1..n] de ℕ, ↓izq, der: ℕ):ℕ
variables m, s: ℕ
inicio
   si izq=der entonces s ← A[izq]
   en otro caso
      m ← (izq+der)/2
      s ← Suma(A, izq, m) + Suma(A, m+1, der)
   finsi
   devolver s
fin
```

Un Ejemplo Ilustrativo (pero no práctico)

- La complejidad (número de sumas) del algoritmo de fuerza bruta es t(n) = n - 1 $\in \theta$(n). Para el algoritmo de Divide y Vencerás tenemos:

$$t(n) = \begin{cases} 0 & n = 1 \\ 2t\left(\frac{n}{2}\right) + 3 & n > 1 \end{cases}$$

- Por Fórmula Maestra, a = 2, b = 3 y c = 2, luego t(n) $\in$ $\theta(n^{\log_2 2})$ = θ(n) también, aunque la constante multiplicativa es mayor: t(n) = 3n - 3.
- En este problema la estrategia de Divide y Vencerás no ha proporcionado ganancia en eficiencia, pero en otros problemas puede hacerlo.

Esquema General

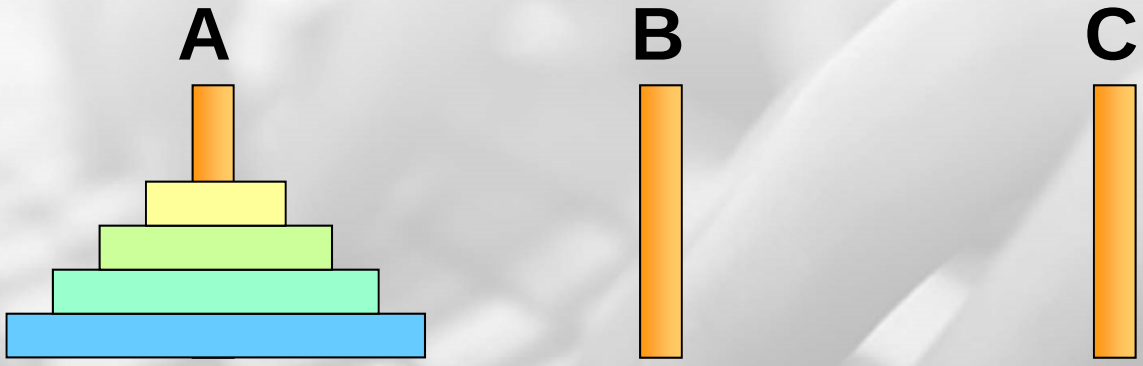
- Los algoritmos de Divide y Vencerás abordan la resolución de problemas del siguiente modo:
  1. La instancia del problema se divide en varias instancias de menor tamaño.
  2. Estas instancias se resuelven.
  3. Las soluciones obtenidas para las instancias más pequeñas se combinan para obtener la solución a la instancia original.

Esquema General

```
DivideVencerás (p: problema)
   Dividir (p, p1, p2, ..., pk)
   para i:= 1, 2, ..., k
      si:= Resolver (pi)
   solución:= Combinar (s1, s2, ..., sk)
```

- Normalmente para resolver los subproblemas se utilizan llamadas recursivas al mismo algoritmo (aunque no necesariamente).
- Ejemplo: Torres de Hanoi

Esquema general



- **Ejemplo.** Problema de las torres de Hanoi. Mover **n** discos del poste A al C:
  - Mover **n-1** discos de A a B
  - Mover 1 disco de A a C
  - Mover **n-1** discos de B a C

Esquema general

```
Hanoi (n, A, B, C: entero)
    si  n==1 entonces
          mover (A, C)
    sino
          Hanoi (n-1, A, C, B)
          mover (A, C)
          Hanoi (n-1, B, A, C)
    finsi
```

- Si el problema es “pequeño”, entonces se puede resolver de forma directa.

Esquema general

- Requisitos para aplicar divide y vencerás:
  - Necesitamos un método (más o menos directo) de resolver los problemas de tamaño pequeño.
  - El problema original debe poder dividirse fácilmente en un conjunto de sub-problemas, del mismo tipo que el problema original pero con una resolución más sencilla (menos costosa).
  - Los sub-problemas deben ser disjuntos: la solución de un sub-problema debe obtenerse independientemente de los otros.
  - Es necesario tener un método de combinar los resultados de los sub-problemas.

Método general

- Normalmente los sub-problemas deben ser de tamaños parecidos.
- Como mínimo necesitamos que hayan dos sub-problemas.
- Si sólo tenemos un sub-problema entonces hablamos de técnicas de **reducción** (o **simplificación**).

- **Ejemplo sencillo:** Cálculo del factorial.
  `Fact(n) = n*Fact(n-1)`

Análisis de tiempos de ejecución

- Para el esquema recursivo, con división en dos sub-problemas con la mitad de tamaño:

$$t(n) = \begin{cases} g(n) & \text{si } n \leq n_0 \text{ (caso base)} \\ 2*t(n/2) + f(n) & \text{en otro caso} \end{cases}$$

- **t(n)**: tiempo de ejecución del algoritmo DV.
- **g(n)**: tiempo de calcular la solución para el caso base, algoritmo directo.
- **f(n)**: tiempo de dividir el problema y combinar los resultados.

Análisis de tiempos de ejecución

- Resolver suponiendo que **n** es potencia de 2
  n = 2^k y $\mathbf{n_0}$ = **1**

$$t(n) = n \cdot g(1) + \sum_{i=0}^{k-1} (2^i f(n / 2^i))$$

- Aplicando expansión de recurrencias:

$$t(n) = 2^m g(n / 2^m) + \sum_{i=0}^{m-1} (2^i f(n / 2^i))$$

- Para $\mathbf{n_0} \neq 1$, siendo **m** tal que $\mathbf{n_0} \geq \mathbf{n}/2^{\mathbf{m}}$

Análisis de tiempos de ejecución

- **Ejemplo 1.** La resolución directa se puede hacer en un tiempo constante y la división y combinación de resultados también.

  $g(n) = c$; $f(n) = d$

  $\Rightarrow t(n) \in \Theta(n)$

- **Ejemplo 2.** La solución directa se calcula en $O(n^2)$ y la combinación en $O(n)$.

  $g(n) = c \cdot n^2$; $f(n) = d \cdot n$

  $\Rightarrow t(n) \in \Theta(n \log n)$

Análisis de tiempos de ejecución.

- En general, si se realizan **a** llamadas recursivas de tamaño **n/b**, y la división y combinación requieren $f(n) = d \cdot n^p \in O(n^p)$, entonces:

$$t(n) = a \cdot t(n/b) + d \cdot n^p$$

Suponiendo $n = b^k \Rightarrow k = \log_b n$

$$t(b^k) = a \cdot t(b^{k-1}) + d \cdot b^{pk}$$

Podemos deducir que:

$$t(n) \in \begin{cases} O(n^{\log_b a}) & \text{Si } a > b^p \\ O(n^p \cdot \log n) & \text{Si } a = b^p \\ O(n^p) & \text{Si } a < b^p \end{cases}$$

Fórmula maestra

Análisis de tiempos de ejecución

- **Ejemplo 3.** Dividimos en 2 trozos de tamaño n/2, con $f(n) \in O(n)$:

  $a = b = 2$

  $t(n) \in O(n \cdot \log n)$

- **Ejemplo 4.** Realizamos 4 llamadas recursivas con trozos de tamaño n/2, con $f(n) \in O(n)$:

  $a = 4; b = 2$

  $t(n) \in O(n^{\log_2 4}) = O(n^2)$

Ejemplos de aplicación

Ordenación por Mezcla

- Mergesort es un ejemplo claro de aplicación con éxito de la técnica de Divide y Vencerás.
- Se desea ordenar un array A[1..n]. Para ello:
  - Se divide el array en dos mitades A[1..$\lfloor n/2 \rfloor$] y A[$\lfloor n/2 \rfloor$ + 1..n].
  - Se ordenan recursivamente estas mitades. El caso base es la ordenación de una mitad con un único elemento.
  - Se mezclan las dos mitades ordenadas para tener un único array ordenado.

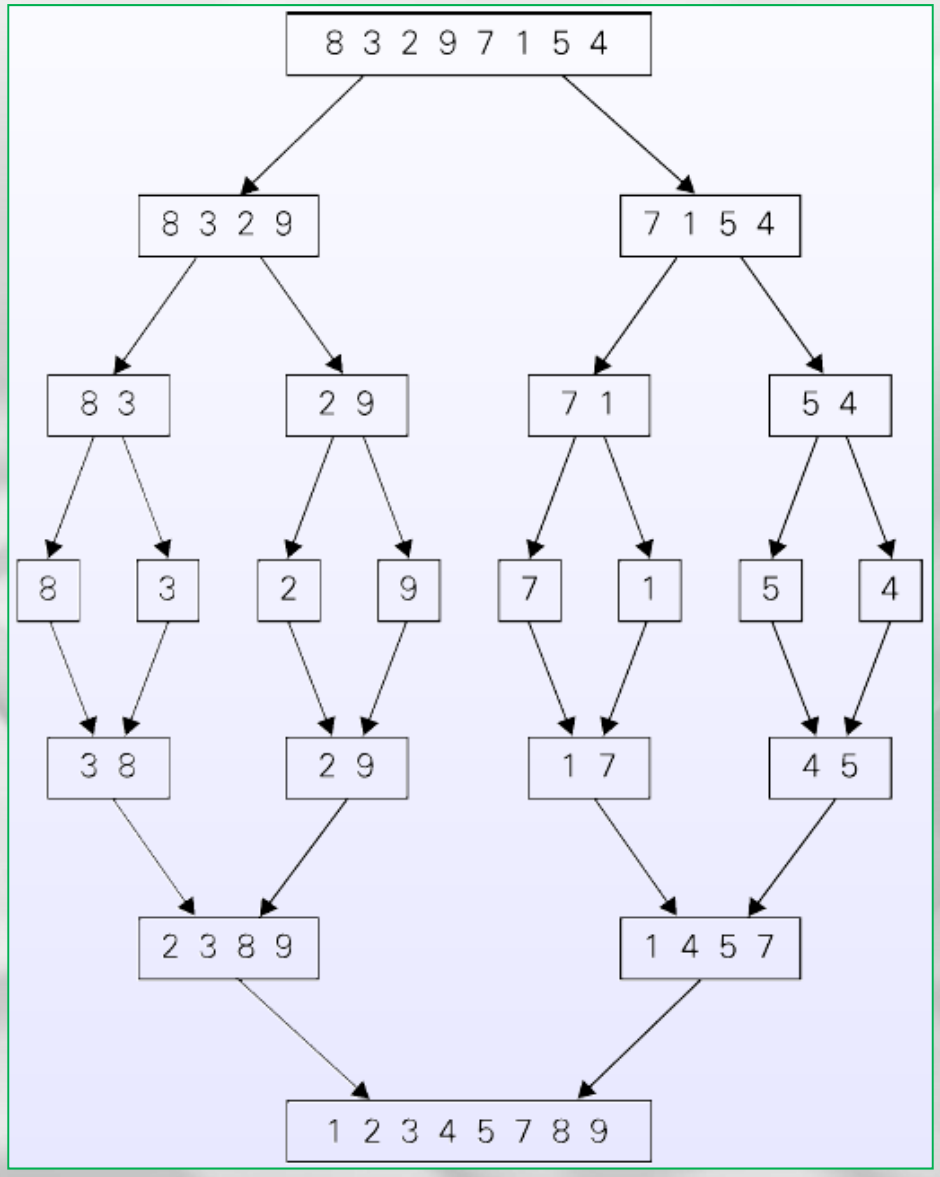
Ordenación por mezcla

```
proc Mergesort (↓↑A: array[1..n]de ℕ)
variables B, C: array[1.. ⌊n/2⌋]de ℕ
inicio
   si n>1 entonces
      B[1..⌊n/2⌋] ← A[1..⌊n/2⌋]
      C[1..⌊n/2⌋] ← A[⌊n/2⌋+1..n]
      Mergesort(B)
      Mergesort(C)
      Mezclar(A, B, C)
   finsi
fin
```

Ordenación por Mezcla

```
proc Mezclar (↓↑A: array[1..n]de ℕ,
              ↓B: array[1..p] de ℕ,
              ↓C: array[1..q] de ℕ)
variables i, j, k: ℕ
inicio
   i ← 1; j ← 1; k ← 1
   mientras (j≤p) ∧ (k≤q) hacer
      si B[j]≤C[k] entonces A[i] ← B[j]; j ← j+1
      en otro caso A[i] ← C[k]; k ← k+1
      finsi
      i ← i+1
   finmientras
   si j>p entonces A[i..n] ← C[k..q]
   en otro caso A[i..n] ← B[j..p]
   finsi
fin
```

Mergesort en Funcionamiento



Eficiencia de Mergesort

- Consideremos en primer lugar la complejidad temporal del algoritmo. Hay dos posibles operaciones básicas:
  1. Comparaciones entre elementos
  2. Copia de elementos
- En relación a las comparaciones se realizan en exclusiva dentro del procedimiento Mezcla.
- Cada iteración del bucle de Mezcla conlleva una comparación. En el peor caso se realizan p + q - 1 = n - 1 iteraciones, luego

$$t(n) = \begin{cases} 0 & n \leq 1 \\ 2t\left(\frac{n}{2}\right) + n - 1 & n > 1 \end{cases}$$

- De acuerdo con la Fórmula Maestra tenemos que

  $t(n) \in \theta(n \log n)$

Eficiencia de Mergesort

- Si consideramos las copias de elementos, en Mergesort se realizan *n* copias, y en Mezcla otras n. Por lo tanto:

$$t(n) = \begin{cases} 0 & n \leq 1 \\ 2t\left(\frac{n}{2}\right) + 2n & n > 1 \end{cases}$$

- De acuerdo con la Fórmula Maestra tenemos nuevamente que t(n) $\in \theta$(n log n).

Eficiencia de Mergesort

- En relación al espacio, dentro de Mergesort se emplean dos arrays cuyo tamaño combinado es n. El consumo total de espacio viene pues dado por:

$$t(n) = \begin{cases} 1 & n = 1 \\ t\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

- De acuerdo con el Teorema Maestro tenemos $\log_2 1 = 0$, $f(n) \in \theta(n) \in \Omega(n^{\epsilon})$ para algún $\epsilon > 0$ (concretamente para todo $0 < \epsilon < 1$), y $n/2 \leq cn$ para algún $c < 1$ (concretamente para todo $1/2 < c < 1$). Por lo tanto, $t(n) \in \theta(n)$.

Ordenación rápida

- Quicksort es otro ejemplo de éxito de la técnica de Divide y Vencerás.
- Se trata de un algoritmo de ordenación extremadamente eficiente descubierto por Sir C.A.R. “Tony” Hoare.
- Se desea ordenar un array A[1..n]. Para ello:
  - Se divide el array en dos mitades A[1..m - 1] y A[m + 1..n], previa reorganización de los valores del array de manera que
    - $\forall i < m : A[i] < A[m]$
    - $\forall i > m : A[i] > A[m]$
  - Se ordenan recursivamente estas mitades. El caso base es la ordenación de una mitad con un único elemento.
- No es necesario realizar ninguna acción posterior a las llamadas recursivas. El array está ordenado tras las mismas.

Ordenación rápida

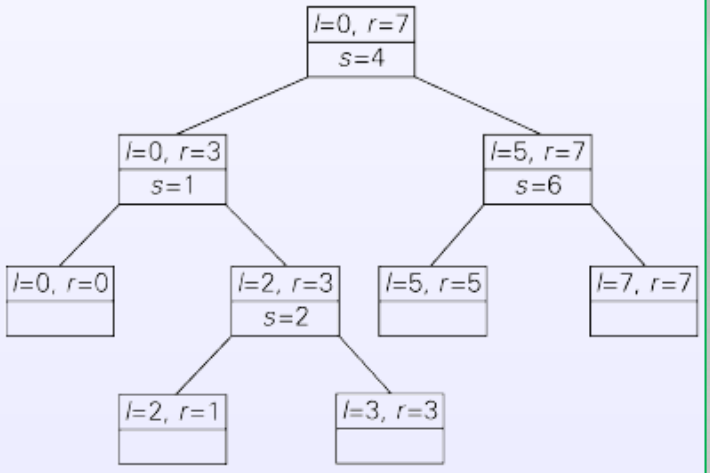
```
proc Quicksort (↓↑A: array[1..n] de ℕ, ↓i, d: ℕ)
variables m: ℕ
inicio
   si d>i entonces
      Partir(A, i , d, m)
      Quicksort(A, i , m - 1)
      Quicksort(A, m + 1, d)
   finsi
fin
```

Ordenación rápida

```
proc Partir (↓↑A: array[1..n] de ℕ, ↓i, d: ℕ, ↑m: ℕ)
variables p: ℕ
inicio
   p ← i
   repetir
      mientras (i ≤ d) ∧ (A[i] ≤ A[p]) hacer i ← i + 1
      finmientras
      mientras (i ≤ d) ∧ (A[d] > A[p]) hacer d ← d - 1
      finmientras
      si i < d entonces intercambiar(A[i], A[d])
      finsi
   hasta que i ≥ d
   intercambiar(A[p], A[d])
   m ← d
fin
```

Quicksort en funcionamiento

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			j			*i*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				j	*i*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			i j				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9



(b)

Eficiencia del Quicksort

- Consideremos la complejidad temporal del algoritmo. En relación a las comparaciones de elementos, se realizan en exclusiva dentro del procedimiento `Partir`.
- Cada invocación del procedimiento `Partir` conlleva θ(n) comparaciones (*n* es el número de elementos). **En el mejor caso**, cada llamada recursiva divide el array por la mitad, luego:

$$t(n) = \begin{cases} 0 & n \leq 1 \\ 2t\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

- De acuerdo con la Formula Maestra tenemos que t(n) $\in \theta$(n log n).

Eficiencia del Quicksort

- En el peor caso, la partición resulta en dos mitades de tamaños extremadamente asimétricos, i.e., una de tamaño 0, y otra de tamaño n - 1:

$$t(n) = \begin{cases} 0 & n \leq 1 \\ t(n-1) + n & n > 1 \end{cases}$$

- Si se resuelve la ecuación recurrente obtenemos que $t(n) \in \theta(n^2)$ en el peor caso.
- La ecuación recurrente anterior también representa el peor caso en relación al número de copias de elementos realizados, por lo que ésta es también $\theta(n^2)$.

Eficiencia de Quicksort

- En la práctica, la complejidad media de Quicksort es del mismo orden que la del mejor caso, i.e., $\theta(n \log n)$. Concretamente, puede demostrarse que

$$t_{avg}(n) \approx 2n \ln n \approx 1{,}38n \log_2 n$$

- Existen diferentes estrategias para minimizar los efectos del peor caso: elección "inteligente" del elemento pivote, aleatorización, etc.

Multiplicación de Enteros

- Consideremos el problema de multiplicar dos enteros de gran tamaño.
- El signo se gestiona de manera independiente, por lo que nos concentraremos en enteros positivos.
- Sea, por ejemplo, A = 23 y B = 14. Para calcular el producto A · B hacemos:

$$A \cdot B = 23 \cdot 14 = 23 \cdot (1 \cdot 10^1 + 4 \cdot 10^0) = 23 \cdot 1 \cdot 10^1 + 23 \cdot 4 \cdot 10^0 =$$
$$= (2 \cdot 10^1 + 3 \cdot 10^0) \cdot 1 \cdot 10^1 + (2 \cdot 10^1 + 3 \cdot 10^0) \cdot 4 \cdot 10^0 =$$
$$= 2 \cdot 1 \cdot 10^2 + 3 \cdot 1 \cdot 10^1 + 2 \cdot 4 \cdot 10^1 + 3 \cdot 4 \cdot 10^0 =$$
$$= (2 \cdot 1) \cdot 10^2 + (2 \cdot 4 + 3 \cdot 1) \cdot 10^1 + (3 \cdot 4) \cdot 10^0 =$$
$$= 200 + 110 + 12 = 322$$

- Si la operación básica es la multiplicación de dos dígitos, se realizan obviamente n^2 operaciones.

Una Versión Diferente

- Consideremos $A = a_1a_0$ y $B = b_1b_0$. Su producto $C = AB$ puede expresarse como

$$C = c_2 10^2 + c_1 10^1 + c_0 10^0$$

donde

$$c_2 = a_1 b_1$$
$$c_1 = a_1 b_0 + a_0 b_1$$
$$c_0 = a_0 b_0$$

- Nótese ahora que

$$c_1 = a_1 b_0 + a_0 b_1 = a_1 b_0 + a_0 b_1 + (a_1 b_1 + a_0 b_0) - (a_1 b_1 + a_0 b_0) =$$
$$= (a_1 + a_0) \cdot (b_1 + b_0) - (a_1 b_1 + a_0 b_0) =$$
$$= (a_1 + a_0) \cdot (b_1 + b_0) - (c_2 + c_0)$$

- Podemos emplear este hecho para optimizar el proceso con números más grandes.

El Caso General: Algoritmo de Karatsuba

- Consideremos A = $a_{n-1}a_{n-2}\ldots a_0$ y B = $b_{n-1}b_{n-2}\ldots b_0$.
- Dividamos A por la mitad:

$$A = \overbrace{a_{n-1}a_{n-2}\cdots a_{n/2}}\,\overbrace{a_{\frac{n}{2}-1}\cdots a_1a_0}$$

- Análogamente, B_1 es la mitad izquierda de los dígitos de B y B_0 es la mitad derecha. Puede verse que

$$A = A_1 10^{n/2} + A_0,\ B = B_1 10^{n/2} + B_0$$

- La misma relación anterior sigue cumpliéndose:

$$C = AB = (A_1 10^{n/2} + A_0) \cdot (B_1 10^{n/2} + B_0) =$$
$$= (A_1B_1)\, 10^n + (A_1B_0 + A_0B_1)\, 10^{n/2} + (A_0B_0) =$$
$$= C_2 10^n + C_1 10^{n/2} + C_0$$

El Caso General: Algoritmo de Karatsuba

- Una vez calculado C_2 y C_0, se puede calcular C_1 como

$$C_1 = (A_1 + A_0)(B_1 + B_0) - (C_2 + C_0)$$

- Todos los cómputos se realizan mediante sumas y multiplicaciones de números de n/2 dígitos.
- Estas multiplicaciones pueden realizarse siguiendo el mismo procedimiento de manera recursiva.
- La recursión se detiene cuando los números tienen un único dígito (o cuando el número de dígitos es lo suficientemente pequeño como para poder multiplicarlos directamente).

El Caso General: Variante de Knuth

- En el algoritmo de Karatsuba, el cálculo de C_1 como

$$C_1 = (A_1 + A_0)(B_1 + B_0) - (C_2 + C_0)$$

  puede conducir a alguna irregularidad, ya que si A_0, A_1 tienen n/2 dígitos, su suma puede tener n/2 + 1 dígitos (y lo mismo con B_0, B_1).
- Knuth propuso una variante que solventa esta problemática:

$$C_1 = C_0 + C_2 - (A_0 - A_1)(B_0 - B_1)$$

- La resta $A_0 - A_1$ tiene exactamente n/2 dígitos, aunque puede ser negativa, lo que hay que tener en cuenta durante el cómputo.

Complejidad del Método

- Si medimos la complejidad en términos del número de multiplicaciones de un dígito tenemos que

$$t(n) = \begin{cases} 1 & n = 1 \\ 3t(\frac{n}{2}) & n > 1 \end{cases}$$

- Aplicando el Teorema Maestro tenemos que a = 3, b = 2, d = $\log_b a = \log_2 3 \approx 1{,}585$, y f(n) = 0 $\in \theta(1)$. Como existe un $\epsilon > 0$ tal que $f(n) \in O(n^{d-\epsilon})$ (cualquier $\epsilon \in (0, d]$ vale), se tiene que $t(n) \in \theta(n^{1,585})$.
- Esto supone una notable ganancia con relación al algoritmo de fuerza bruta que tiene complejidad $\theta(n^2)$.

Multiplicación rápida de matrices

- Supongamos el problema de multiplicar dos matrices cuadradas A, B de tamaños **n**x**n**. C = AxB

$$C(i, j) = \sum_{k=1..n} A(i, k)\cdot B(k, j);\ \text{Para todo } i, j = 1..n$$

- Método clásico de multiplicación:

```
for i:= 1 to N do
   for j:= 1 to N do
      suma:= 0
      for k:= 1 to N do
         suma:= suma + a[i,k]*b[k,j]
      end
      c[i, j]:= suma
   end
end
```

- El método clásico de multiplicación requiere $\Theta(n^3)$.

Multiplicación rápida de matrices.

- Aplicamos **divide y vencerás**:
  Cada matriz de **n**x**n** es dividida en cuatro submatrices de tamaño (**n**/2)x(**n**/2): A_{ij}, B_{ij} y C_{ij}.

A_{11}	A_{12}
A_{21}	A_{22}

X

B_{11}	B_{12}
B_{21}	B_{22}

=

C_{11}	C_{12}
C_{21}	C_{22}

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- Es necesario resolver 8 problemas de tamaño **n**/2.
- La combinación de los resultados requiere un O(**n**^{2}).

$$t(n) = 8 \cdot t(n/2) + a \cdot n^2$$

- Resolviéndolo obtenemos que t(n) es O(n^3).
- Podríamos obtener una mejora si hiciéramos 7 multiplicaciones (o menos)...

Multiplicación rápida de matrices

- **Multiplicación rápida de matrices (Strassen):**

$P = (A_{11}+A_{22})(B_{11}+B_{22})$
$Q = (A_{12}+A_{22})\ B_{11}$
$R = A_{11}\ (B_{12}-B_{22})$
$S = A_{22}(B_{21}-B_{11})$
$T = (A_{11}+A_{12})B_{22}$
$U = (A_{21}-A_{11})(B_{11}+B_{12})$
$V = (A_{12}-A_{22})(B_{21}+B_{22})$

$C_{11} = P + S - T + U$
$C_{12} = R + T$
$C_{21} = Q + S$
$C_{22} = P + R - Q + U$

- Tenemos 7 sub-problemas de la mitad de tamaño.
- ¿Cuánto es el tiempo de ejecución?

Multiplicación rápida de matrices

- El tiempo de ejecución será:

$$t(n) = 7 \cdot t(n/2) + a \cdot n^2$$

- Resolviéndolo, tenemos que:

$$t(n) \in O(n^{\log_2 7}) \approx O(n^{2.807}).$$

- Las constantes que multiplican al polinomio son mucho mayores (tenemos muchas sumas y restas), por lo que sólo es mejor cuando la entrada es muy grande (empíricamente, para valores en torno a **n**>120).

- ¿Cuál es el tamaño óptimo del caso base?

Multiplicación rápida de matrices

- Aunque el algoritmo es más complejo e inadecuado para tamaños pequeños, se demuestra que la **cota de complejidad del problema** es menor que $O(n^3)$.

- **Cota de complejidad de un problema:** tiempo del algoritmo más rápido posible que resuelve el problema.

- Algoritmo clásico → $O(n^3)$
- V. Strassen (1969) → $O(n^{2.807})$
- V. Pan (1984) → $O(n^{2.795})$
- D. Coppersmith y S. Winograd (1990) → $O(n^{2.376})$
- ...

Bibliografía Complementaria

- A. Mohammed and M. Othman
  Comparative Analysis of Some Pivot Selection Schemes for Quicksort Algorithm
  Inform Technol J 6:424-427, 2007
  http://dx.doi.org/10.3923/itj.2007.424.427