CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Recurrencias y el principio de dividir y conquistar

Objetivos

- Repasar los conceptos de recurrencias y algunos métodos de solución
- Estrategia dividir y conquistar para el diseño de algoritmos
- Algoritmo mergesort para ordenamiento y su análisis
- Problema de multiplicación de matrices y algoritmos
- Algoritmo de Strassen para multiplicación de matrices
- Algoritmo de Karatsuba para multiplicación de enteros

Recurrencias

Una función $f(n)$ de los naturales en los reales (o enteros) puede entenderse como una **sucesión** $\langle f_0, f_1, f_2, \ldots \rangle$ donde $f(n) = f_n$

Una recurrencia es una **regla** que permite calcular el n-ésimo valor f_n de la sucesión a partir de los valores anteriores

La recurrencia también debe especificar los **valores borde** de la sucesión: aquellos valores que una vez conocidos junto con la regla inductiva permiten calcular toda la sucesión

En computación nos interesa conocer que tan rápido crecen los valores f_n

Ejemplo de recurrencia

Considere la recurrencia dada por:

– el valor de borde $f_0 = 0$

– la regla inductiva $f(n) = f(n-1) + n$ para $n > 0$

La recurrencia define la sucesión:

$$\langle 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, \ldots \rangle$$

La **solución** a la recurrencia es una **forma cerrada** que permite calcular el valor f_n directamente a partir de n

En el ejemplo, $f(n) = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$ cuya rata de crecimiento es $f(n) = \Theta(n^2)$

Métodos de solución

Existen diversos métodos generales de solución exacta o aproximada de recurrencias. Ningún método es totalmente general y de hecho existen recurrencias para las cuales no se conocen formas cerradas

En este curso consideramos dos métodos de solución:

– método de sustitución o inductivo

– Teorema Maestro

El método de sustitución puede usarse para calcular formas cerradas exactas o aproximadas

Método de sustitución

El método de sustitución es una forma de verificar una forma cerrada obtenida de alguna manera

Por lo general lo usamos para dar una cota sobre la tasa de crecimiento cuando tenemos una pista sobre ella

En esencia, el método de sustitución es una **prueba por inducción** sobre la tasa de crecimiento (o forma cerrada en algunos casos) de la recurrencia

Ejemplo del método de sustitución

Considere la recurrencia:

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ f(n-1) + n & \text{si } n > 0 \end{cases}$$

Suponga que creemos $f(n) = O(n)$ y queremos probarlo

Lo primero es hacer la suposición explícita $f(n) \leq cn$ para alguna constante $c > 0$, y mostrar por inducción que $f(n) \leq cn$ para todos los n en los cuales f está definida:

Base: $f(0) = 0 \leq c \cdot 0 = 0$

Tesis: $f(n) = f(n-1) + n \leq c(n-1) + n = cn + n - c \overset{?}{\leq} cn$

Como no existe ninguna **constante** c tal que $n - c \leq 0$ para todo n "grande", entonces $f(n) = O(n)$ es falso

Ejemplo del método de sustitución

Considere la recurrencia:

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ f(n-1) + n & \text{si } n > 0 \end{cases}$$

Ahora intentemos $f(n) = O(n^2)$. Suponemos $f(n) \leq cn^2$ para alguna constante $c > 0$

Base: $f(0) = 0 \leq c \cdot 0^2 = 0$

Tesis:

$$\begin{aligned} f(n) &= f(n-1) + n \leq c(n-1)^2 + n = cn^2 - 2cn + c + n \\ &= cn^2 - n(2c-1) + c \overset{?}{\leq} cn^2 \end{aligned}$$

Desigualdad válida para n "grande" para cualquier $c \geq 1/2$ □

Dividir y conquistar

Muchos algoritmos tiene una estructura **recursiva**

Ellos se ejecutan así mismos una o varias veces para solucionar subproblemas de menor tamaño que surgen de un problema dado

Estos algoritmos utilizan un enfoque de **dividir y conquistar:**

- **dividen** el problema a resolver en subproblemas más pequeños
- **solucionan** los subproblemas de forma **recursiva**
- **combinan** las soluciones de los subproblemas en una solución al problema original

Mergesort

El algoritmo **mergesort** es un algoritmo de ordenamiento basado en comparaciones de tipo dividir y conquistar:

Divide la secuencia de n objetos en dos secuencias de $n/2$ objetos cada una

Soluciona cada secuencia (i.e. ordena) de forma recursiva

Combina las dos soluciones (mezcla/merge) en una secuencia ordenada

Mergesort

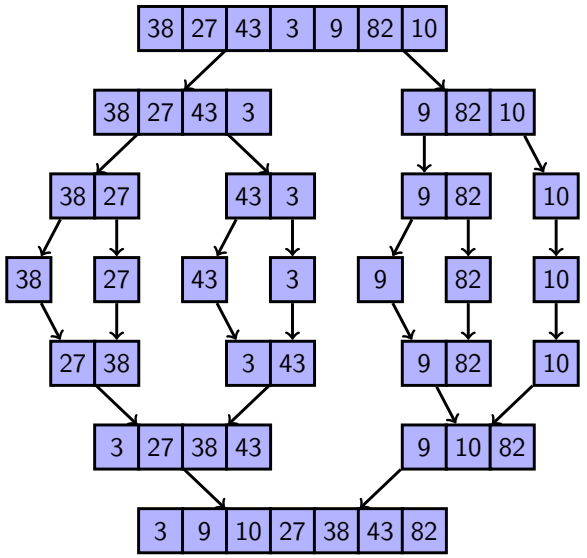
Input: arreglo $A[p \ldots r]$ con $r - p + 1$ elementos

Output: arreglo A con elementos reordenados de menor a mayor

```
1 Mergesort(array A, int p, int 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)
```

La rutina `Merge` se encarga de mezclar (combinar) dos secuencias ordenadas en una secuencia ordenada

Mergesort



Merge (1 de 3)

Suponga que tenemos dos mazos de cartas, cada uno de ellos ordenados de menor a mayor (la primera carta del mazo es la menor del mazo y la última es la mayor), y queremos obtener un mazo con las cartas de ambos mazos ordenadas de menor a mayor

Procedimiento:

1. Comenzar con un nuevo mazo vacío que al finalizar tendra todas las cartas ordenadas
2. Si alguno de los mazos está vacío, colocar el otro mazo al final del mazo resultante y terminar
3. Comparar la primera carta de cada mazo
4. Remover la menor carta y colocarla al final del nuevo mazo
5. Repetir 2–4 hasta terminar

Merge (2 de 3)

Input: arreglo A, índices p, q y r tal que los subarreglos $A[p \ldots q]$ y $A[q+1 \ldots r]$ están ordenados de menor a mayor

Output: arreglo $A[p \ldots r]$ reordenado de menor a mayor

```
1  Merge(array A, int p, int q, int r)
2      n = q - p + 1                                  % tamaño de A[p...q]
3      m = r - q                                      % tamaño de A[q+1...r]
4      let L[1...n+1] y R[1...m+1] nuevos arreglos
5      for i = 1 to n do
6          L[i] = A[p + i - 1]
7      for i = 1 to m do
8          R[i] = A[q + i]
9      L[n+1] = R[m+1] = ∞                            % sentinelas
10     i = j = 1
11     for k = p to r do
12         if L[i] <= R[j]
13             A[k] = L[i]                            % remover primero de L
14             i = i + 1
15         else
16             A[k] = R[j]                            % remover primero de R
17             j = j + 1
```


Merge (3 de 3)

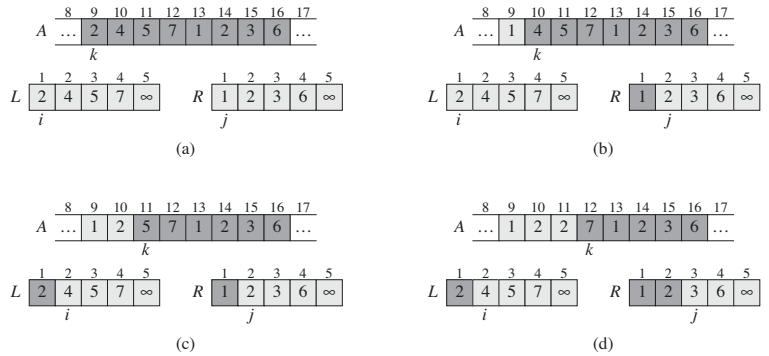


Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Operación de `Merge(A, 9, 12, 16)`. Panel (a) justo después de la inicialización (líneas 2–11)

Correctitud de mergesort

Para analizar la correctitud de un **algoritmo recursivo** utilizamos inducción en el tamaño n de la entrada

Para $n = 1$, $p = r$ y `Mergesort(A,p,r)` no realiza ninguna operación sobre A. Como el arreglo tiene un solo elemento, el algoritmo es correcto

Ahora suponemos que `Mergesort(A,p,r)` es correcto para todas las entradas de tamaño $\leq k$ y considere una entrada de tamaño $k + 1$

`Mergesort(A,p,r)` hace dos llamadas recursivas sobre los arreglos $A[p \ldots q]$ y $A[q + 1 \ldots r]$ respectivamente. Como ambos subarreglos tienen a lo sumo k cartas, y`Mergesort` ordena de forma correcta ambos subarreglos (hipótesis inductiva)

La correctitud de `Mergesort(A,p,r)` sigue de la correctitud del algoritmo `Merge(A,p,q,r)`

Desempeño de merge

`Merge(A,p,q,r)` ejecuta tres lazos:

- el lazo 5–6 de $q - p + 1$ iteraciones
- el lazo 7–8 de $r - q$ iteraciones
- el lazo 11–17 de $r - p + 1$ iteraciones

Cada iteración toma tiempo constante. En total se ejecutan $2n$ iteraciones donde $n = r - p + 1$

`Merge(A,p,q,r)` toma tiempo $\Theta(n)$ **independiente** de q

Desempeño de mergesort

Claramente `Mergesort(A,p,r)`:

- hace recursión sobre los subproblemas $A[p \ldots q]$ y $A[q + 1 \ldots r]$ con $q = \lfloor (p + r)/2 \rfloor$
- mezcla los resultados con `Merge(A,p,q,r)`

El tiempo $T(n)$ para mergesort sobre un arreglo de n elementos se puede describir con la recurrencia:

$$T(n) \;=\; \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Teorema Maestro

Sea $a \geq 1$ y $b > 1$ constantes, $f(n)$ una función y $T(n)$ una recurrencia sobre los enteros no negativos dada por

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

donde n/b se refiere a $\lfloor n/b \rfloor$ ó $\lceil n/b \rceil$. Entonces,

1. Si $f(n) = O(n^{\log_b a - \epsilon})$ para algún $\epsilon > 0$, $T(n) = \Theta(n^{\log_b a})$
2. Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} \log n)$
3. Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ para algún $\epsilon > 0$, y $af(n/b) \leq cf(n)$ para alguna constante $c < 1$ y n grande, entonces $T(n) = \Theta(f(n))$

Análisis de mergesort

Para `Mergesort` obtuvimos la recurrencia

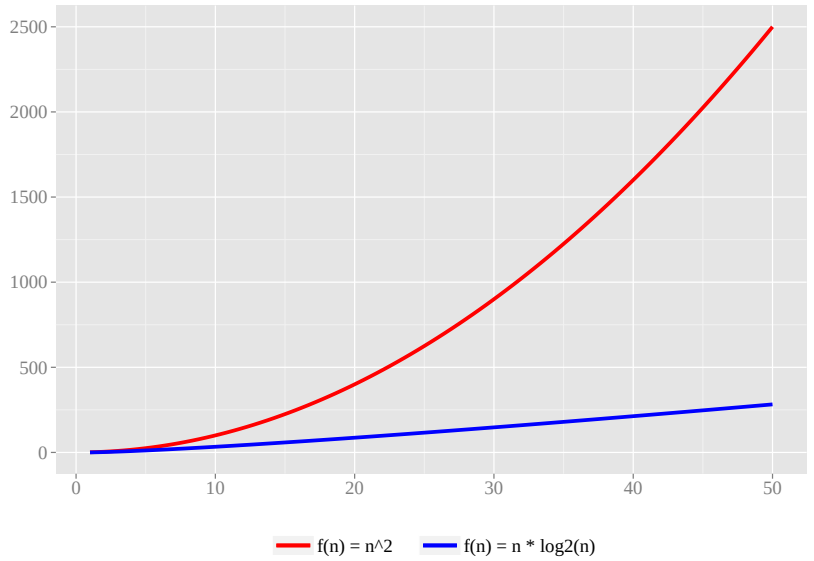
$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Para utilizar el Teorema Maestro calculamos $\log_b a = \log_2 2 = 1$ y **comparamos** $f(n)$ con $n^{\log_b a} = n$. Como $f(n) = \Theta(n)$, estamos en el segundo caso del Teorema Maestro y concluimos

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

Más adelante veremos que `Mergesort` es un algoritmo **asintóticamente óptimo** de ordenamiento basado en comparaciones □

Tiempo: n^2 vs. $n \log n$



Multiplicación de matrices

Ahora consideramos un nuevo problema. El problema de **multiplicar matrices cuadradas**

Considere dos matrices $A = (a_{ij})$ y $B = (b_{ij})$ de dimensión $n \times n$

Queremos diseñar un algoritmo eficiente para calcular el producto $A \times B = C = (c_{ij})$ definido por:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

$$\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

Multiplicación de matrices

Input: dos matrices A y B de dimensión $n \times n$

Output: matriz $C = A \times B$ de dimensión $n \times n$

```
1  Square-Matrix-Multiply(matrix A, matrix B)
2      n = A.nrows
3      let C be new matrix[n][n]
4      for i=1 to n do
5          for j=1 to n do
6              C[i][j] = 0
7              for k=1 to n do
8                  C[i][j] += A[i][k] * B[k][j]
9      return C
```

El algoritmo corre en tiempo $T(n) = \Theta(n^3)$ para matrices $n \times n$

Multiplicación de matrices

Como cada una de las n^2 entradas c_{ij} en $C = A \times B$ está definida por

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

pareciera que se **necesitan** n^3 multiplicaciones para calcular C

¿Existe un algoritmo que realice menos de n^3 multiplicaciones?

El **algoritmo de Strassen** para multiplicación de matrices realiza $\Theta(n^{\log_2 7}) = O(n^{2.81})$ multiplicaciones

El algoritmo de Strassen es un algoritmo de tipo **dividir y conquistar**

Multiplicación de matrices

Asumimos $n = 2^q$ ya que subdivideremos recursivamente las matricies de dimensión $n \times n$ en 4 submatrices de dimensión $n/2 \times n/2$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

El valor de n garantiza que cada submatriz está bien definida

Calculamos el producto usando la descomposición:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\begin{aligned} C_{11} &= A_{11} \times B_{11} + A_{12} \times B_{21} \\ C_{12} &= A_{11} \times B_{12} + A_{12} \times B_{22} \\ C_{21} &= A_{21} \times B_{11} + A_{22} \times B_{21} \\ C_{22} &= A_{21} \times B_{12} + A_{22} \times B_{22} \end{aligned}$$

Multiplicación de matrices

Input: matrices A y B de dimensión $n \times n$ con n potencia de 2

Output: matriz $C = A \times B$

```
1  Square-Matrix-Multiply-Recursive(matrix A, matrix B)
2    n = A.nrows
3    let C be new matrix[n][n]
4    if n == 1
5        C[1][1] = A[1][1] * B[1][1]
6    else
7        particionar A, B y C en 4 submatrices cada una
8        C11 = Square-Matrix-Multiply-Recursive(A11, B11) +
9              Square-Matrix-Multiply-Recursive(A12, B21)
10       C12 = Square-Matrix-Multiply-Recursive(A11, B12) +
11             Square-Matrix-Multiply-Recursive(A12, B22)
12       C21 = Square-Matrix-Multiply-Recursive(A21, B11) +
13             Square-Matrix-Multiply-Recursive(A22, B21)
14       C22 = Square-Matrix-Multiply-Recursive(A21, B12) +
15             Square-Matrix-Multiply-Recursive(A22, B22)
16   return C
```


Análisis

`Square-Matrix-Multiply-Recursive` realiza **8 llamadas recursivas** cada una con submatrices de tamaño $n/2 \times n/2$

El tiempo para particionar las matrices es $\Theta(n^2)$ ya que ese es el número de elementos que deben ser copiados al particionar

Después de la recursión se deben realizar 8 sumas de matrices de dimension $n/2 \times n/2$, lo que toma $\Theta(n^2)$ tiempo

La recurrencia $T(n)$ para el tiempo de ejecución es:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{si } n > 1 \end{cases}$$

Por el primer caso del Teorema Maestro, $T(n) = \Theta(n^3)$

Algoritmo de Strassen (1969)

El algoritmo recursivo no mejora el tiempo de ejecución pero nos da la base para el **Algoritmo de Strassen**

La idea es explotar la recursión junto con manipulaciones aritméticas para **reducir el número de llamadas recursivas de 8 a 7**

El tiempo de corrida del algoritmo de Strassen viene dada por la recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{si } n > 1 \end{cases}$$

Por el primer caso del TM, $T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$

Algoritmo de Strassen

La idea consiste en primero construir las siguientes 10 matrices de dimension $n/2 \times n/2$:

$$\begin{aligned} S_1 &= B_{12} - B_{22} & S_6 &= B_{11} + B_{22} \\ S_2 &= A_{11} + A_{12} & S_7 &= A_{12} - A_{22} \\ S_3 &= A_{21} + A_{22} & S_8 &= B_{21} + B_{22} \\ S_4 &= B_{21} - B_{11} & S_9 &= A_{11} - A_{21} \\ S_5 &= A_{11} + A_{22} & S_{10} &= B_{11} + B_{12} \end{aligned}$$

Para construir cada matriz se requiere de $n^2/4$ operaciones de suma/resta. Las 10 submatrices se calculan en tiempo $\Theta(n^2)$

Algoritmo de Strassen

Una vez que las S_i son calculadas, **utilizamos recursión** para calcular los productos:

$$\begin{aligned} P_1 &= A_{11} \times S_1 = A_{11} \times B_{12} - A_{11} \times B_{22} \\ P_2 &= S_2 \times B_{22} = A_{11} \times B_{22} + A_{12} \times B_{22} \\ P_3 &= S_3 \times B_{11} = A_{21} \times B_{11} + A_{22} \times B_{11} \\ P_4 &= A_{22} \times S_4 = A_{22} \times B_{21} - A_{22} \times B_{11} \\ P_5 &= S_5 \times S_6 = A_{11} \times B_{11} + A_{11} \times B_{22} + A_{22} \times B_{11} + A_{22} \times B_{22} \\ P_6 &= S_7 \times S_8 = A_{12} \times B_{21} + A_{12} \times B_{22} - A_{22} \times B_{21} - A_{22} \times B_{22} \\ P_7 &= S_9 \times S_{10} = A_{11} \times B_{11} + A_{11} \times B_{12} - A_{21} \times B_{11} - A_{21} \times B_{12} \end{aligned}$$

Son 7 matrices P_i de dimensión $n/2 \times n/2$ que se calculan de forma recursiva en **tiempo total** $7T(n/2)$

Algoritmo de Strassen

Finalmente, calculamos las 4 matrices C_{ij} del resultado $C = A \times B$:

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

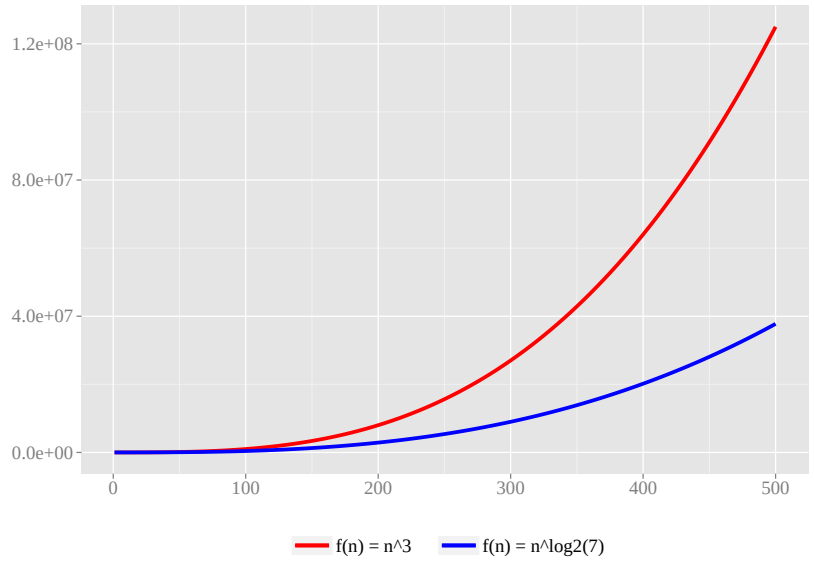
(ver Ejercicio 11)

La matrices C_{ij} se calculan realizando $\Theta(n^2)$ operaciones de suma/resta a partir de las matrices P_k

El tiempo total del algoritmo de Strassen es

$$T(n) = 7\,T(n/2) + \Theta(n^2) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Tiempo: n^3 vs. $n^{\log_2 7}$



Multiplicación de enteros de precisión arbitraria

Considere dos enteros representados por **arreglos de dígitos** sobre una base B (e.g. $B = 2$)

Por ejemplo,

- $x = 87,612,372 = 0101\ 0011\ 1000\ 1101\ 1011\ 1101\ 0100_2$
- $y = 34,239,751 = 0010\ 0000\ 1010\ 0111\ 0101\ 0000\ 0111_2$

Queremos calcular el producto xy. ¿Cómo lo podemos hacer?

El **algoritmo clásico de multiplicación** para enteros de n dígitos toma tiempo $\Theta(n^2)$

¿Es posible hacerlo en menos tiempo?

Algoritmo clásico de multiplicacón

87,612,372	$\times$
34,239,751	
87,612,372	$+$
4,380,618,600	
61,328,660,400	
788,511,348,000	
2,628,371,160,000	
17,522,474,400,000	
350,449,488,000,000	
2,628,371,160,000,000	
2,999,825,801,799,372	

Algoritmo de multiplicación de Karatsuba (1960)

Asumimos que el número de dígitos es $n = 2^q$ y expresamos

$$\begin{aligned} x &= x_1 \times B^{n/2} + x_0 \\ y &= y_1 \times B^{n/2} + y_0 \end{aligned}$$

donde x_0, x_1, y_0, y_1 son de $n/2$ dígitos cada uno

Claramente, $xy = x_1y_1 \times B^n + (x_0y_1 + x_1y_0) \times B^{n/2} + x_0y_0$ cuyo cálculo **requiere 4 multiplicaciones**

Karatsuba define $z_0 = x_0y_0$, $z_1 = x_0y_1 + x_1y_0$ y $z_2 = x_1y_1$ tal que $xy = z_2 \times B^n + z_1 \times B^{n/2} + z_0$ pero **observa** que

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

se puede calcular con **una sola multiplicación** realizando 4 sumas/restas en lugar de 1 suma

Algoritmo de multiplicación de Karatsuba (1960)

Input: arreglos $A[1 \ldots n]$, $B[1 \ldots n]$ y $C[1 \ldots 2n]$ con $n = 2^q$

Output: C contiene el producto ab de los enteros a y b en A y B

```
1  Multiply-Karatsuba(array A, array B, array C, int n)
2      if n == 1 then
3          C[1] = A[1] * B[1]
4          return
5
6      let Z0, Z1, Z2 nuevos arreglos de tamaño n, y T0, T1 de tamaño n/2
7
8      % calcular z0 = a0 x b0 y z2 = a1 x b1
9      Multiply-Karatsuba(A[1..n/2], A[1..n/2], Z0, n/2)
10     Multiply-Karatsuba(A[1+n/2..n], B[1+n/2..n], Z2, n/2)
11
12     % calcular z1 = a0 x b1 + a1 x b0 = (a0 + a1) x (b0 x b1) - z0 - z2
13     Sumar(A[1..n/2], A[1+n/2..n], T0, n/2)
14     Sumar(B[1..n/2], B[1+n/2..n], T1, n/2)
15     Multiply-Karatsuba(T0, T1, Z1 n/2)
16     Restar(Z1, Z0, Z1, n)
17     Restar(Z1, Z2, Z1, n)
18
19     % calcular ab = z2 x B^(2n) + z1 x B^(n/2) + z0
20     for i=1 to 2n do C[i] = 0
21     Sumar(Z0, C[1..n], C[1..n], n)
22     Sumar(Z1, C[n/2..3n/2], C[n/2..3n/2], n)
23     Sumar(Z2, C[n..2n], C[n..2n], n)
```

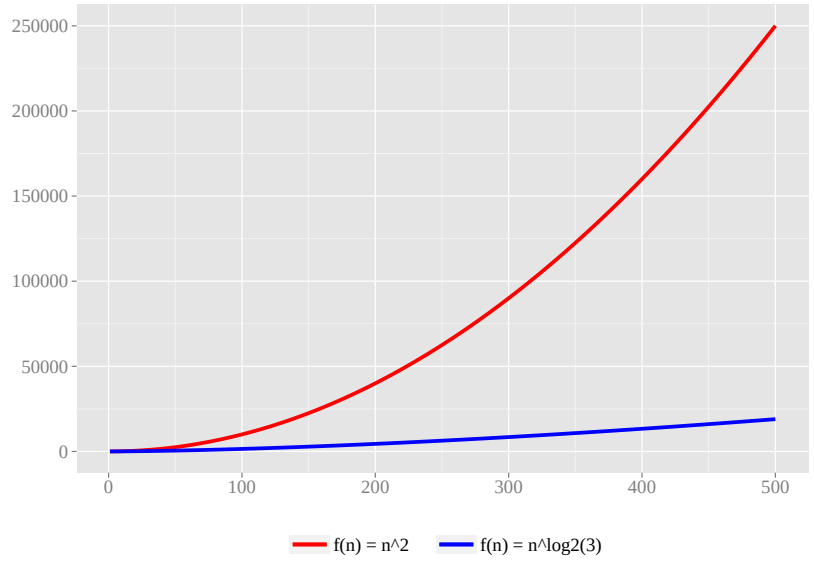

Análisis del algoritmo de Karatsuba

Sea $T(n)$ el tiempo tomado por el algoritmo de Karatsuba para calcular el producto de dos enteros de n dígitos cada uno

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 3T(n/2) + \Theta(n) & \text{si } n = 2^q \end{cases}$$

Por el 1er caso del Teorema Maestro, $T(n) = \Theta(n^{\log_3 2}) = O(n^{1.585})$

Tiempo: n^2 vs. $n^{\log_2 3}$



Resumen

- Recurrencias y métodos de solución: método de sustitución (inductivo) y Teorema Maestro
- Estrategia de dividir y conquistar (descomposición en subproblemas)
- Mergesort y su análisis
- Multiplicación de matrices: algoritmo ingenuo y algoritmo de Strassen
- Algoritmo de Karatsuba para multiplicación de enteros

Ejercicios (1 de 4)

1. (4.3-1) Use el método de sustitución para ver que $T(n) = T(n-1) + n$ es $O(n^2)$
2. (4.3-2) Use el método de sustitución para ver que $T(n) = T(\lceil n/2 \rceil) + 1$ es $O(\log n)$
3. (4.5-3) Use el Teorema Maestro para mostrar que $T(n) = T(n/2) + 1$ es $\Theta(\log n)$
4. (2.3-2) Modifique el procedimiento `Merge` para que no utilice sentinelas
5. Haga una implementación recursiva de búsqueda binaria y analice su tiempo de ejecución con el Teorema Maestro
6. Verifique la correctitud del procedimiento `Merge(A,p,q,r)` que hace el merge de los subarreglos ordenados $A[p \dots q]$ y $A[q+1 \dots r]$

Ejercicios (2 de 4)

7. (2.3-4) Describa `Insertion-Sort` de forma recursiva: para ordenar $A[p \dots r]$, ordenamos $A[p \dots r-1]$ y luego insertamos $A[r]$ en el arreglo ordenado $A[p \dots r-1]$. Escriba y solucione una recurrencia para `Insertion-Sort-Recursive`
8. (2.3-7) Diseñe un algoritmo que corra en tiempo $\Theta(n \log n)$ que dado un un arreglo $A[p \dots r]$ de $n = r - p + 1$ enteros y un entero x, determine si existen índices i y j tales que $p \leq i < j \leq r$ y $A[i] + A[j] = x$
9. Calcule el tiempo de ejecución del algoritmo ingenuo de multiplicación de matrices en función del tamaño de la entrada. Haga lo mismo para el Algoritmo de Strassen
10. (4.2-1) Utilice el algoritmo de Strassen para calcular

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \times \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

Ejercicios (3 de 4)

11. Verifique la correctitud del algoritmo de Strassen (i.e. verifique las expresiones para las matrices C_{ij} en función de las matrices P_k)
12. (4.2-2) Implemente `Square-Matrix-Multiply` y el algoritmo de Strassen y compare sus tiempos de corrida sobre casos de prueba de diferente dimensión
13. (4.2-3) Modifique el algoritmo de Strassen para que multiplique matrices de dimensión $n \times n$ cuando n no sea una potencia de 2. El algoritmo resultante debe correr en tiempo $\Theta(n^{\log_2 7})$
14. (4.2-6) ¿Qué tan rápido puede calcular el producto de dos matrices A y B de dimensiones $kn \times n$ y $n \times kn$ usando el algoritmo de Strassen como subrutina?

Ejercicios (4 de 4)

15. (4.2-7) Muestre como multiplicar dos números complejos $a + bi$ y $c + di$ con tan solo 3 multiplicaciones. El algoritmo recibe como entrada los números a, b, c y d, y retorna la parte real y compleja, $ac - db$ y $ad + bc$, de forma separada
16. Implemente el algoritmo clásico para multiplicación de enteros y el algoritmo de Karatsuba y compare sus tiempos de corrida sobre casos de prueba de diferente tamaño
17. Modifique el algoritmo de Karatsuba para que pueda multiplicar enteros de n dígitos cuando n no es potencia de 2