Análisis y Diseño de Algoritmos

Tema 3: Divide y Vencerás

UNIVERSIDAD DE MÁLAGA
E.T.S. INGENIERÍA INFORMÁTICA

Contenido

- Introducción
- Estrategia
  - Esquema
  - Requisitos de eficiencia
- Ejemplos
  - Búsqueda Binaria
  - Ordenación por mezcla
  - Ordenación rápida
  - Multiplicación de números grandes
- Referencias

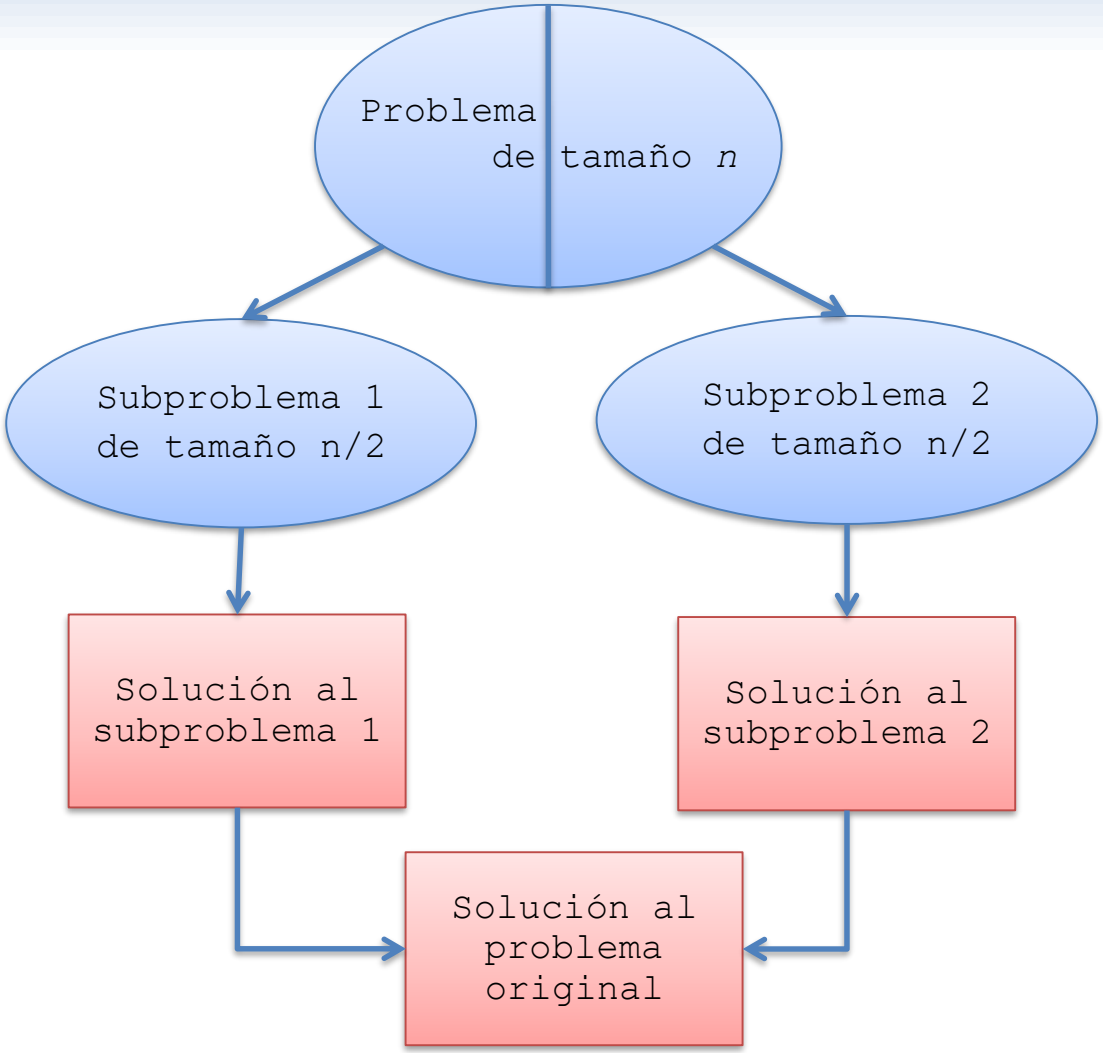
Introducción

- Divide y Vencerás es una técnica intuitiva y que aplicamos a diario en la vida cotidiana.
- 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:
  - Algoritmos de ordenación MergeSort y QuickSort.
  - Algoritmo de búsqueda binaria.

Estrategia

- La estrategia Divide y Vencerás consiste en:
  - Para resolver una instancia de un problema, se divide en instancias más pequeñas del **mismo problema** (que tengan más o menos el mismo tamaño).
  - Se resuelven esas instancias más pequeñas (normalmente utilizando recursividad) de la misma forma.
  - Si es necesario se combinan las soluciones obtenidas para obtener la solución del problema original.

Esquema Típico




```
Esquema DyV(problema  x)
    si pequeño(x) entonces
      metodoDirecto(x)

    si no
      descomponer(x, k_subproblemas)

      para cada subproblema
            DyV(subproblema_i)

      combinar(k_subsoluciones)
    finsi
```

Caso base(x)

Consideraciones

- En general, una instancia de tamaño n se divide en $a \geq 1$ instancias de tamaño n/b.
  - En caso de $a = 1$, habremos reducido el problema original a uno de menor tamaño cuya solución es equivalente.

- Muchas veces $a = b$, i.e., la instancia se divide en $b > 1$ pedazos de tamaño n/b, cada uno de los cuales se resuelve por separado.

- En algunos casos, puede que no haga falta resolver todos los subproblemas (i.e., a < b), o puede que los subproblemas se solapen en cierta medida (i.e., a > b).

- Si el coste de dividir y combinar es $f(n)$, el coste total del algoritmo de Divide y Vencerás es $T(n) = aT(n/b) + f(n)$.

Requisitos de Eficiencia

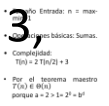
- Las operaciones descomponer y combinar han de ser eficientes.
- El número de subproblemas pequeño.
- El tamaño de los subproblemas ha de ser aproximadamente igual.
- Los subproblemas no deben solaparse entre sí.
- No existe un método sistemático que nos indique siempre cuál es la mejor forma de dividir el problema en subproblemas, ni cuál debe ser el tamaño de los subproblemas.
- La consecuencia es que el modelo no siempre ofrece la mejor resolución, sino la más sencilla.

Ejemplo de eficiencia

```
/**
 * @param l lista de números a sumar
 * @param min y max, índices máximo y mínimo
 * @return Suma de los elementos de la lista
*/
public static int suma(List<Integer> l,
                       int min,int max){
      if (max < min) return 0;
      if (max == min) return l.get(min);
      else {
         int m = (min + max)/2;
         int s1 = suma(l, min, m);
         int s2 = suma(l, m+1,max);
         return s1+s2;
      }
}
```

- Tamaño Entrada: n = max-min+1
- Operaciones básicas: Sumas.
- Complejidad:
  T(n) = 2 T(n/2) + 3
- Por el teorema maestro $T(n) \in \Theta(n)$
  porque a = 2 > 1= 2^0 = b^d

- Esta aplicación de la técnica DyV no mejora el enfoque por fuerza bruta, que consiste en recorrer la lista y sumar sus elementos y tendría un coste $T_{FB}(n) = n - 1$.
- Incluso lo empeora, porque la expresión exacta de la complejidad es $T(n) = 3n - 3$, que se consigue con T(1) = 0 y T(2) = 3.



Búsqueda Binaria (I)

- El algoritmo de Búsqueda Binaria es un ejemplo donde la técnica Divide y Vencerás da como resultado un algoritmo más eficiente.

- Se puede aplicar cuando deseamos encontrar un elemento *x* en una colección ordenada de datos *V[1..N].*
  - Es decir, V satisface $\forall k \in \mathbb{N}, 1 \leq k < N : V[k-1] \leq V[k]$

- Idea Clave: Inspeccionar el elemento de índice **m**
  - Si V[m] = x Fin Búsqueda
  - Si V[m] < x $\forall k \in \mathbb{N}, 1 \leq k < m : V[k] < x \Rightarrow$ eliminados
  - Si V[m] > x $\forall k \in \mathbb{N}, m < k \leq N : V[k] > x \Rightarrow$ eliminados

- Elección Óptima: m = (primero + último)/2

Búsqueda Binaria(II)

```
/** Busqueda binaria recursiva  */
public static int busqBinaria(int[] a,int inf,int sup, int x){
   int pos = -1;
   if (inf <= sup){
       int medio = (inf + sup)/2;
       if (x == a[medio]) {pos = medio;}
       else if (a[medio] > x) {pos = busqBinaria(a,inf,medio-1,x);}
            else {pos =  busqBinaria(a,medio+1,sup,x);}
    }
    return pos;
}
```

```
/**  Búsqueda binaria iterativa */
public static int busqBinaria(int[] a,int
x){
    int izda = 0,dcha = a.length-1;
    int medio = (izda + dcha)/2;
    while (izda <= dcha && a[medio]!=x){
      if (a[medio]>x) dcha = medio - 1;
      else izda = medio + 1;
      medio = (izda + dcha)/2;
    }
    if (izda <= dcha) return medio;
    else return -1;
}
```

Ordenación por Mezcla (I)

- El algoritmo de 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:

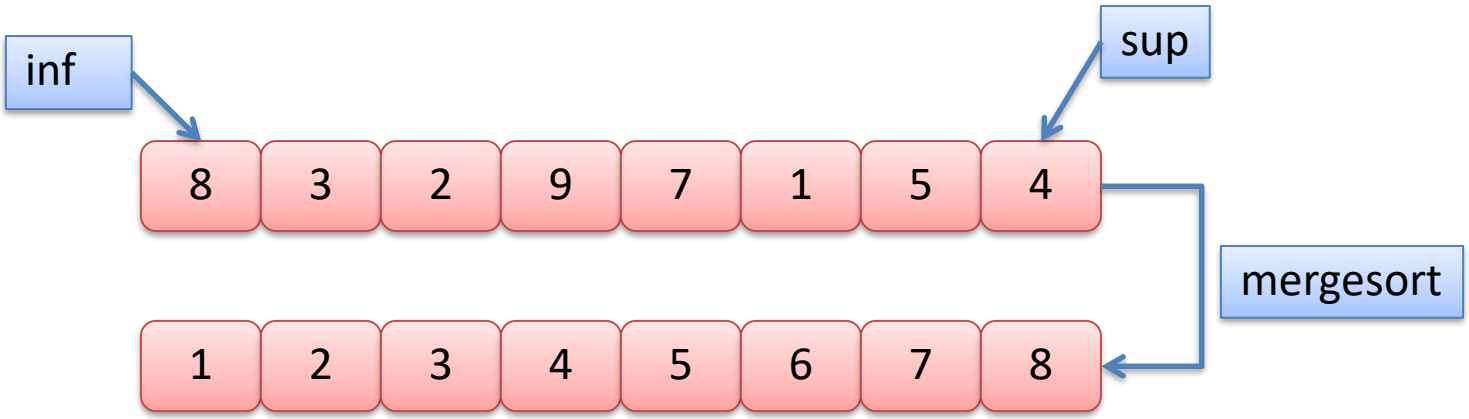
  1. Se divide el array en dos mitades $A[1..\lfloor n/2 \rfloor]$ y $A[\lfloor n/2 \rfloor+1..n]$.

  2. Se ordenan recursivamente estas mitades. El caso base es la ordenación de una mitad con un único elemento.

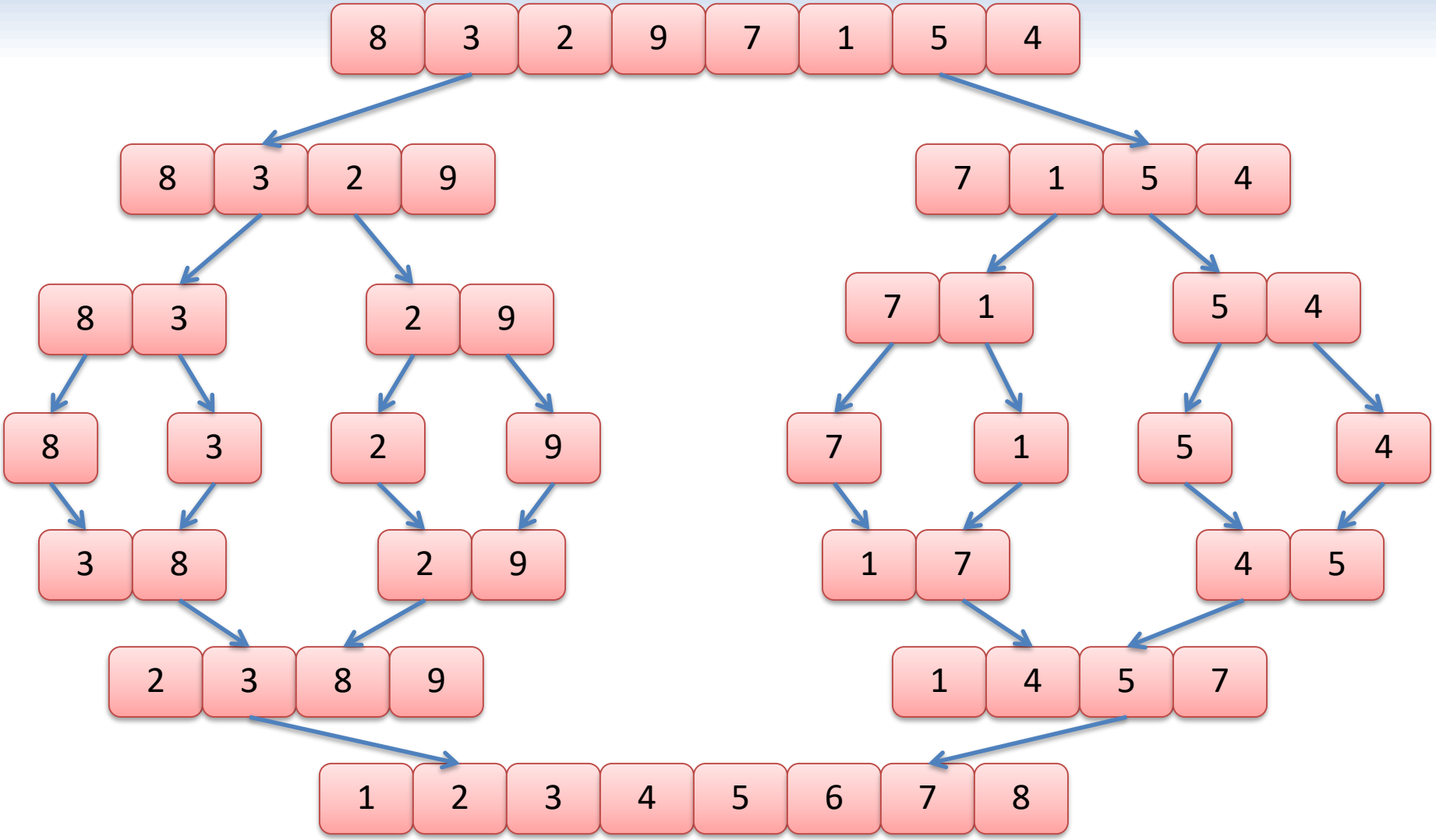
  3. Se mezclan las dos mitades ordenadas para tener un único array ordenado.

Ordenación por Mezcla (II)

```
/**
 *
 * @param a array
 * @param inf primera posición a considerar
 * @param sup última posición a considerar
 */
public static void mergeSort(int[] a,int inf, int sup){
    if (inf < sup){
            mergeSort(a,inf,(inf+sup)/2);
            mergeSort(a,(inf+sup)/2+1,sup);
            mezclar(a,inf,(inf+sup)/2,sup);
     }
}
```



Ordenación por Mezcla (III)



Ordenación por Mezcla (IV)

```
/* mezcla las dos mitades de a[inf..sup] de forma ordenada.
   Utiliza un array intermedio  */
public  static  void  mezclar(
          int[] a,int  inf,int medio,int sup){
   int i = inf; int j = medio+1;
   int[] b = new int[sup-inf+1];
   int k = 0;
   while (i<=medio && j <=sup){
     if (a[i]<=a[j]){
        b[k] = a[i];i++;
     }else{
        b[k] = a[j];j++;
     } k++;
   }
   while (i<=medio){
     b[k] = a[i];
     i++; k++;
   }
   while (j<=sup){
     b[k] = a[j];
     j++; k++;
   }
   k=0;
   for (int f=inf; f<= sup; f++){
     a[f] = b[k];k++;
    }
}
```

Estudiar la complejidad del algoritmo mezclar

Ordenación por Mezcla (IV)

```
/* mezcla las dos mitades de a[inf..sup] de forma ordenada.
   Utiliza un array intermedio */
public static void mezclar(
          int[] a,int inf,int medio,int sup){
   int i = inf; int j = medio+1;
   int[] b = new int[sup-inf+1];
   int k = 0;
   while (i<=medio && j <=sup){
     if (a[i]<=a[j]){
       b[k] = a[i];i++;
     }else{
       b[k] = a[j];j++;
     } k++;
   }
   while (i<=medio){
     b[k] = a[i];
     i++; k++;
   }
   while (j<=sup){
     b[k] = a[j];
     j++; k++;
   }
   k=0;
   for (int f=inf; f<= sup; f++){
     a[f] = b[k];k++;
   }
}
```

- Tamaño entrada: n = sup-inf+1
- Operaciones básicas: cada comparación y cada asignación de componentes del array tienen un coste de 1.
- Complejidad temporal:
  - Caso mejor: el primer bucle hace n/2 iteraciones

  $T(n) = 2 \cdot n/2 + n/2 + n = 2.5\ n$
  - Caso peor: el primer bucle hace n-1 iteraciones.

  $T(n) = 2\ (n-1) + 1 + n = 3n - 1$
- Complejidad espacial:

  Array auxiliar de n componentes,

  $E(n) = n$

Ordenación por Mezcla (V)

```
/**
 *
 * @param a array con elementos desordenados
 * @param inf ordena mediante la ordenación por mezcla
 * @param sup el array a[inf..sup]
 */
public static void mergeSort(int[] a,int inf, int sup){
    if (inf < sup){
            mergeSort(a,inf,(inf+sup)/2);
            mergeSort(a,(inf+sup)/2+1,sup);
            mezclar(a,inf,(inf+sup)/2,sup);
     }
}
```

$$T(n) = 2T(\tfrac{n}{2}) + T_{mezclar}(n) = 2T(\tfrac{n}{2}) + 3n - 1$$

$$T(1) = 0$$

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

Ordenacion Rápida (I)

- La Ordenación Rápida (Quicksort) es un algoritmo de ordenación extremadamente eficiente descubierto por Charles A.R. Hoare.

- Se desea ordenar un array *A[1..n]*. Para ello:

  - Se divide el array en dos mitades *A[1..p − 1]* y *A[p+1..n]*, previa reorganización de los valores del array de manera que
    1. $\forall i \in \mathbb{N}, 0 < i < p : A[i] \leq A[p]$
    2. $\forall i \in \mathbb{N}, p < i < n : A[i] \geq A[p]$

  - 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 (II)

```
/**
 * @param a array con elementos desordenados
 * ordena mediante la ordenación rápida
 * el array a[inf..sup]
 */
public static void quickSort(int[] a,int inf,int sup){
        if (inf < sup){
           int p = partir(a,inf,sup);
           quickSort(a,inf,p-1);
           quickSort(a,p+1,sup);
         }
}
```

método partir:

parte el array **a[inf..sup]** en dos subarrays (que pueden ser vacíos) **a[inf..p-1], a[p+1..sup]** tal que

-todos los elementos de **a[inf..p-1] son menores o iguales que a[p]**

-todos los elementos de **a[p+1..sup] son mayores o iguales que a[p]**

y devuelve el índice **p**

método quickSort:

-parte el array **a[inf..sup]** en dos subarrays (que pueden ser vacíos) **a[inf..p-1], a[p+1..sup]** con el método **partir.**

-Ordena **a[inf..p-1] y a[p+1..sup]** llamando de forma recursiva a **quickSort**

Ordenación Rápida (III)

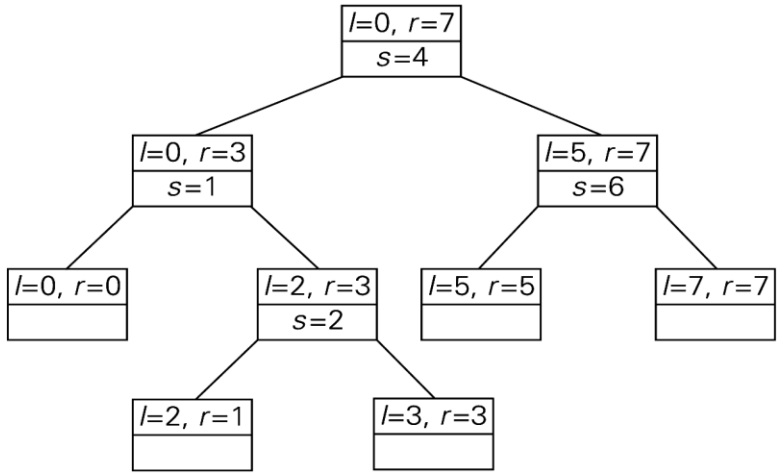
```
/**
 * Parte el array[inf..sup] en dos subarrays a[inf..j]
 * y a[j+1..sup], de forma que todos los elementos de
 * a[inf..j] son menores o iguales que un pivote y todos los elementos
 * de a[j+1..sup] son mayores o iguales que el pivote
 * @return  la posición del pivote (j)
 */
public static  int partir(int[] a, int inf, int sup){
  int pivote = a[inf]; int i = inf+1;  int j = sup;
  do {
        while((i<=j) && (a[i] <= pivote)){ i++; }
        while((i<=j) && (a[j] > pivote)){  j--; }
        if (i<j){ intercambia(i,j); }

  }while (i < j)

  intercambia(inf,j);
  return j;}
```

```
void intercambia(int []a, int i, int j){
   int aux = a[i];
   a[i] = a[j];
   a[j] = aux;
}
```

Al acabar el bucle do_while, j es la posición del elemento con valor menor o igual que el pivote que está más a la derecha. Por ello, intercambiamos a[inf], que tiene el pivote, con a[j].

Ordenación Rápida (IV)

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

Ordenación Rápida (V)

- Determinemos la complejidad temporal del algoritmo
  - Tamaño de la entrada, n: número de elementos del array.
  - Operaciones básicas: comparaciones y asignaciones de elementos del array. Éstas se realizan en exclusiva dentro del procedimiento Partir.

- Cada invocación del procedimiento *Partir* conlleva *Θ(n)* operaciones básicas.

- En el mejor caso, cada llamada recursiva divide el array por la mitad, luego:

$$\mathrm{T}(n) = 2T\left(\frac{n}{2}\right) + n, n > 1$$

- De acuerdo con el Teorema Maestro, $f(n) \in \Theta\left(n^{\log_2(2)} \cdot \log^0 n\right) = \Theta(n)$ por lo que *T(n) ∈ Θ(n log n)*.

Ordenación Rápida (VI)

- 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*:

$$T(n) = T(n-1) + T(0) + n = T(n-1) + n, \qquad n > 1$$

Si se resuelve la recurrencia obtenemos que $T(n) \in \Theta(n^2)$.

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

- En la práctica, la complejidad media de Quicksort es del mismo orden que la del mejor caso. Concretamente, puede demostrarse que $T_{avg}(n) = 2n \cdot \ln(n) \approx 1.38n \log_2(n)$

Ejercicio

- Implementar mediante
  - Fuerza bruta
  - Divide y Vencerás

  el algoritmo

$\{a.length \geq 3 \wedge (N_{i \in \{1..a.length-2\}} esPico(a,i)) = 1\}$

int pico (int [] a) //valor

$\{\exists i: 0 \leq i < a.length: (valor = a[i] \wedge esPico(a,i)\}$

donde $esPico(a,p) \equiv a[p] > a[p-1] \wedge a[p] > a[p+1]$

Multiplicación de Enteros (I)

- 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 $A = 23$ y $B = 14$. Para calcular el producto $A \cdot B$ hacemos:

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

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

Multiplicación de Enteros (II)

- 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

$$\begin{aligned} c_1 &= a_1 b_0 + a_0 b_1 = \\ &= a_1 b_0 + a_0 b_1 + (a_1 b_1 + a_0 b_0) - (a_1 b_1 + a_0 b_0) = \\ &= (a_1 + a_0)(b_1 + b_0) - (a_1 b_1 + a_0 b_0) = \\ &= (a_1 + a_0)(b_1 + b_0) - (c_2 + c_0) \end{aligned}$$

- Podemos emplear el hecho de que se realizan sólo 3 multiplicaciones de números de un dígito para optimizar el proceso con números más grandes.

Algoritmo de Karatsuba (I)

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

A_1 A_0

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

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

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

- La misma relación anterior sigue cumpliéndose:

$$\begin{aligned} C &= A \cdot B = (A_1 10^{n/2} + A_0) \cdot (B_1 10^{n/2} + B_0) = \\ &= (A_1 B_1)\, 10^n + (A_1 B_0 + A_0 B_1)\, 10^{n/2} + (A_0 B_0) = \\ &= C_2 10^n + C_1 10^{n/2} + C_0 \end{aligned}$$

Algoritmo de Karatsuba (II)

- Una vez calculados 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)

Algoritmo de Karatsuba (III)

- 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 este problema:

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

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.

Algoritmo de Karatsuba (IV)

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

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

- Aplicando versión reducida del Teorema Maestro tenemos que a = 3, b = 2 y $f(n) = 0 \in \Theta(n^0) = \Theta(1)$ (por lo que d = 0).

- Por tanto, dado que $a = 3 > 2^0 = b^d$, podemos afirmar que $T(n) \in \Theta\left(n^{\log_2 3}\right) = \Theta(n^{1.585})$, lo que supone una notable ganancia con relación al algoritmo de fuerza bruta que tiene complejidad $\Theta(n^2)$

Referencias

- *Introduction to The Design & Analysis of Algorithms*. A. Levitin. Ed. Adison-Wesley

- *Introduction to Algorithms*. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Ed. The MIT Press