Metodología y Tecnología de la Programación

Ingeniería en Informática

Curso 2008-2009

Esquemas algorítmicos. Divide y Vencerás

Yolanda García Ruiz	**D228**	**ygarciar@fdi.ucm.es**
Jesús Correas	**D228**	**jcorreas@fdi.ucm.es**

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(elaborado a partir de [NN98], [BB97], [GV00] y notas de S. Estévez, A. Verdejo y R. González del Campo)

Bibliografía

- **Importante**: Estas transparencias son un material de apoyo a las clases presenciales y no sustituyen a la bibliografía básica ni a las propias clases presenciales para el estudio de la asignatura
- Bibliografía básica:
  - [NN98]: capítulo 2
  - [GV00]: capítulo 3
- Bibliografía complementaria:
  - [BB97]: capítulo 7
- Ejercicios resueltos:
  - [MOV04]: capítulo 11

Esquemas algorítmicos. Divide y Vencerás

1. Características generales de la técnica Divide y Vencerás
2. Esquema general y estudio de complejidad
3. *Mergesort*
4. *Quicksort*
5. Búsqueda de la mediana de dos *arrays*
6. El elemento en su posición
7. Búsqueda del elemento mayoritario
8. Multiplicación de matrices

Características generales de la técnica Divide y Vencerás

- Esta técnica consiste en dividir un problema original en subproblemas que sean:
  - de la misma naturaleza que el problema original
  - de menor tamaño
- Cuando se genera un único subproblema de menor tamaño, esta técnica se denomina *reducción* o *simplificación*
- Lo veremos con un ejemplo sencillo

Un ejemplo: Búsqueda binaria

- Este es uno de los problemas más sencillos de esta técnica
- Problema: Desarrollar un algoritmo para encontrar un elemento x en una lista ordenada S, y devolver la posición donde está x en S, o bien la posición donde debería estar si x no se encuentra en S
- Los pasos de este esquema son los siguientes:
  - Si s es igual al elemento en la posición central de S, terminar. En caso contrario:
  - *Dividir* la lista en dos sublistas de aproximadamente el mismo tamaño. Si x es menor al elemento central de la lista, elegir la sublista izquierda; en caso contrario, elegir la sublista derecha
  - Resolver (*vencer*) la sublista elegida determinando si x está en esta sublista (posiblemente de forma recursiva)
  - *Obtener* la solución para la lista a partir de la solución de la sublista

Un ejemplo: Búsqueda binaria (cont.)

- Si buscamos $x = 18$ y tenemos la siguiente lista:

10	12	13	14	18	20	25	27	30	35	40	45	47

- El elemento central es $S[7] = 25$. Como $x < S[7]$, determinamos si x está en la sublista izquierda:

10	12	13	14	18	20

  - En esta sublista aplicamos el procedimiento de forma recursiva: el elemento central es $S[3] = 13$. Como $x > S[3]$, buscamos x en la sublista derecha:

14	18	20

  - Seguimos aplicando el mismo procedimiento hasta encontrar el elemento buscado o hasta llegar a una lista vacía

- Obtenemos la solución para la lista a partir de la solución de la sublista

Un ejemplo: Búsqueda binaria (cont.)

- El algoritmo resultante es:

fun busqueda_bin_rec(S[1..n], x, inf, sup)
  si inf $>$ sup **entonces**
    devolver inf
  si no
    mitad $\leftarrow \lfloor$ (inf+sup) / 2 $\rfloor$
    si x $=$ S[mitad] **entonces**
      devolver mitad
    si no si x $<$ S[mitad] **entonces**
      devolver busqueda_bin_rec(S, x, inf, mitad-1)
    si no
      devolver busqueda_bin_rec(S, x, mitad+1, sup)
    fin si
  fin si
fin fun

Un ejemplo: Búsqueda binaria (cont.)

- Análisis de la complejidad de este algoritmo en el caso peor
- El caso peor puede ocurrir cuando x es mayor a cualquier elemento de la lista, o bien cuando no está en la lista
- En cada llamada recursiva, el tamaño de la lista se divide por dos
- Suponiendo que n es una potencia de 2, la complejidad del algoritmo es

$$W(n) = \begin{cases} c_1 & si \;\; n = 1 \\ W(n/2) + c_2 & si \;\; n > 1 \end{cases}$$

- Por tanto, $W(n) \in \Theta(lg \;\; n)$

Esquemas algorítmicos. Divide y Vencerás

1. Características generales de la técnica Divide y Vencerás
2. Esquema general y estudio de complejidad
3. *Mergesort*
4. *Quicksort*
5. Búsqueda de la mediana de dos *arrays*
6. El elemento en su posición
7. Búsqueda del elemento mayoritario
8. Multiplicación de matrices

Esquema general de la técnica Divide y Vencerás

fun DyV(x)
  si pequeño(x) **entonces**
    devolver solucion_directa(x)
  si no
    $\langle x_1, ..., x_k \rangle \leftarrow$ descomponer(x)
    desde i$\leftarrow$ 1 **hasta** k **hacer**
      $y_i \leftarrow$ DyV(x_i)
    fin desde
    devolver combinar(y_1, ..., y_k)
  fin si
fin fun

Esquema general de la técnica Divide y Vencerás (cont.)

- Para poder aplicar esta técnica es necesario que se cumplan algunas condiciones:
- El problema original debe poder dividirse en un conjunto de subproblemas más sencillos del mismo tipo
- Los subproblemas deben ser disjuntos: la solución de un subproblema debe poder obtenerse independientemente de los otros
- Es necesario un método (más o menos directo) para resolver problemas de tamaño pequeño
- La combinación de las soluciones de los subproblemas puede ser trivial (caso de búsqueda binaria) o no
- La complejidad del esquema Divide y Vencerás es en muchos casos del tipo:

$$T(n) = \begin{cases} c & si\ 1 \leq n < b \\ aT(n/b) + f(n) & si\ n \geq b \end{cases}$$

Esquemas algorítmicos. Divide y Vencerás

1. Características generales de la técnica Divide y Vencerás
2. Esquema general y estudio de complejidad
3. *Mergesort*
4. *Quicksort*
5. Búsqueda de la mediana de dos *arrays*
6. El elemento en su posición
7. Búsqueda del elemento mayoritario
8. Multiplicación de matrices

Mergesort

- *Mergesort* es otro algoritmo de ordenación de *arrays*, como los algoritmos de inserción y selección que hemos visto antes
- La idea básica de este algoritmo de ordenación es la siguiente:
  - Dividir el *array* a ordenar en fragmentos más pequeños
  - Ordenar los *subarrays*
  - Combinar los *subarrays* ordenados
- Como *arrays* pequeños podemos considerar los de tamaño 1, que además están trivialmente ordenados
- Veamos un ejemplo de aplicación

Mergesort (cont.)

- Dada la siguiente lista inicial:

27	10	12	20	25	13	15	22

- Divide:

27	10	12	20

25	13	15	22

- Divide:

27	10

12	20

25	13

15	22

- Divide:

27

10

12

20

25

13

15

22

- Combina los resultados:

10	27

12	20

13	25

15	22

- Combina los resultados:

10	12	20	27

13	15	22	25

- Combina los resultados:

10	12	13	15	20	22	25	27

- ¿Cómo podemos especificar este algoritmo?

Mergesort (cont.)

- El algoritmo de *Mergesort* puede especificarse de la siguiente forma:

```
proc mergesort(S[1..n])
   h ← ⌊ n/2 ⌋ ; m ← n-h
   crear U[1..h], V[1..m]
   si n>1 entonces
      U[1..h] ← S[1..h]
      V[1..m] ← S[h+1..n]
      mergesort(U)
      mergesort(V)
      combinar(U,V,S)
   fin si
fin proc
```

- Observaciones:
  - La instrucción **crear** nos permite crear *arrays* locales
  - Las asignaciones de múltiples elementos de un *array* tienen complejidad lineal respecto al número de elementos asignados (como se puede hacer con un bucle **desde**)

Mergesort (cont.)

```
proc combinar(U[1..h],V[1..m],S[1..h+m])
    i ← 1 ; j ← 1 ; k ← 1
    mientras i ≤ h Y j ≤ m hacer
        si U[i] < V[j] entonces
            S[k] ← U[i]
            i ← i+1
        si no
            S[k] ← V[j]
            j ← j+1
        fin si
        k ← k+1
    fin mientras
    si i>h entonces
        S[k..h+m] ← V[j..m]
    si no
        S[k..h+m] ← U[i..h]
    fin si
fin proc
```

10	12	20	27

13	15	22	25

10	12	13	15	20	22	25	27

Mergesort: Análisis de complejidad de caso peor

- Primero lo hacemos sobre el procedimiento `combinar`
- Podemos observar que el bucle `mientras` de las líneas 3-12 se va a ejecutar p veces, donde $p < h + m$
- Podemos obtener el número de operaciones elementales en función del número de vueltas del bucle `mientras`:

línea	instrucciones	op. elem.
2	**Asignaciones iniciales**:	3
3	Condición del bucle:	3
4-10	instrucción **si-entonces-si no**:	3+max(5,5)
11	Incremento de k:	2
3-12	**Total bucle**:	$3 + p \cdot (3 + 3 + 5 + 2) = 3 + p \cdot 13$
13	Condición **si-entonces-si no**:	1
14	asignación múltiple (*):	$2 + (h + m - p) \cdot (2 + 3 + 4) + 2$
16	asignación múltiple (*):	$2 + (h + m - p) \cdot (2 + 3 + 4) + 2$
13-17	**Total si-entonces-si no**:	$1 + 2 + (h + m - p) \cdot (2 + 3 + 4) + 2$
		$= 5 + (h + m - p) \cdot 9$

Mergesort: Análisis de complejidad de caso peor (cont.)

- (*) Las asignaciones a múltiples elementos de los *arrays* se han calculado como un bucle:

  $aux_1 \leftarrow k$; $aux_2 \leftarrow j$
  mientras $aux_1 \leq h+m$ **hacer**
     $S[aux_1] \leftarrow V[aux_2]$
     $aux_1 \leftarrow aux_1 + 1$
     $aux_2 \leftarrow aux_2 + 1$
  fin mientras

- El caso peor ocurre cuando el bucle **mientras** da el número máximo de vueltas: cuando $p = h + m - 1$
- En este caso,
  $T_{combinar}(h, m) = 3 + 3 + 13 \cdot (h + m - 1) + 5 + 14 = 13 \cdot (h + m) + 12$

Mergesort: Análisis de complejidad de caso peor (cont.)

- Pasamos a analizar el caso peor del procedimiento `mergesort`
- El número de operaciones elementales de `mergesort` es:

línea	**instrucciones**	**op. elem.**
2	Asignaciones iniciales:	4
4	Cond. **si-entonces**:	1
5	Asignación *subarray* (*):	$2 + h \cdot (1 + 3 + 4) + 1$
6	Asignación *subarray* (*):	$2 + m \cdot (1 + 3 + 4) + 1$
7	Llamada a `mergesort`:	$1 + T(h)$
8	Llamada a `mergesort`:	$1 + T(m)$
9	Llamada a `combinar`:	$1 + T_{combinar}(h, m)$

- Por tanto, la función de complejidad en el caso peor es:

$$T(n) = \underbrace{4}_{\text{línea 2}} + \underbrace{1}_{\text{línea 4}} + \underbrace{3 + 8h}_{\text{línea 5}} + \underbrace{3 + 8m}_{\text{línea 6}} + \underbrace{1 + T(h)}_{\text{línea 7}}$$
$$+ \underbrace{1 + T(m)}_{\text{línea 8}} + \underbrace{1 + 13 \cdot (h + m) + 12}_{\text{línea 9}}$$

Mergesort: Análisis de complejidad de caso peor (cont.)

- Si n es una potencia de 2, entonces $h = n/2$ y $m = n/2$, y por tanto:
$$T(n) = \begin{cases} c & si \;\; n = 1 \\ 2T(n/2) + 21n + 26 & si \;\; n \geq 2 \end{cases}$$
- Por tanto, como $2 = 2^1$, según el teorema de reducción por división,
$$T(n) \in \Theta(n \;\, lg \;\, n)$$
- Como $n \;\, lg \;\, n$ es una función suave, y $T(n)$ es no decreciente, esto se puede afirmar para todo n
- En [NN98] se estudia la complejidad de *mergesort* pero utilizando el método de la instrucción característica
- ¿Cuál es el espacio ocupado por este algoritmo? ¿Se podría mejorar?

Esquemas algorítmicos. Divide y Vencerás

1. Características generales de la técnica Divide y Vencerás
2. Esquema general y estudio de complejidad
3. *Mergesort*
4. *Quicksort*
5. Búsqueda de la mediana de dos *arrays*
6. El elemento en su posición
7. Búsqueda del elemento mayoritario
8. Multiplicación de matrices

Quicksort

- *Quicksort* es otro algoritmo de ordenación desarrollado por C.A.R. Hoare en 1962
- En él se utiliza la técnica divide y vencerás, de forma parecida a *mergesort*: el *array* se divide en dos *subarrays*, y se ordena cada *subarray* de forma recursiva
- Sin embargo, el *array* se divide en dos de forma distinta:
  - Se elige un elemento del *array* como *pivote*
  - Todos los elementos del *array* menores al pivote se sitúan en el primer *subarray*
  - Todos los elementos del *array* mayores al pivote se sitúan en el segundo *subarray*
- El pivote puede ser cualquier elemento del *array*, pero por conveniencia se elige el primero
- Una vez obtenidas las dos particiones, se realiza la misma operación con cada una de ellas

Quicksort (cont.)

- El resultado de la llamada a *quicksort* es la concatenación de la primera partición ordenada, el pivote y la segunda partición ordenada. Por tanto, no es necesaria una fase posterior de combinación de los resultados como en *mergesort*
- De esta forma, el coste que en *mergesort* estaba dedicado a la combinación de los resultados, ahora se dedica a la partición del *array* en dos
- Veamos un ejemplo de aplicación con la siguiente lista inicial:

15	22	13	27	12	10	20	25

Quicksort (cont.)

- El algoritmo de *quicksort* se puede definir de la siguiente forma:

```
proc quicksort(S[i..j])
   si i<j entonces
      particion(S[i..j],p)
      quicksort(S[i..p-1])
      quicksort(S[p+1..j])
   fin si
fin proc
```

- Inicialmente, `quicksort` debe llamarse con $S[1..n]$
- Los elementos se ordenan en el mismo *array* (*in-place sorting*)
- `particion` debe cambiar de posición los elementos del *array* de forma que el pivote (situado en $S[p]$) separe el *subarray* de los elementos menores a él ($S[i..p-1]$) de los mayores ($S[p+1..j]$)
- ¿Cómo sería una especificación eficiente de `particion`?

Quicksort: `partition`

```
proc particion(S[inf..sup],p)
   pivote ← S[inf]
   j ← inf
   desde i=inf+1 hasta sup hacer
      si S[i] < pivote entonces
         j ← j + 1
         aux ← S[i] ; S[i] ← S[j] ; S[j] ← aux
      fin si
   fin desde
   p ← j
   aux ← S[inf] ; S[inf] ← S[p] ; S[p] ← aux
fin proc
```

- ¿Cómo funciona `particion`?

Quicksort: `partcion` (cont.)

Situación inicial:	**15**	22	13	27	12	10	20	25
i = 2, j = 1	**15**	**22**	13	27	12	10	20	25
i = 3, j = 1*	**15**	22	**13**	27	12	10	20	25
i = 4, j = 2	**15**	13	22	**27**	12	10	20	25
i = 5, j = 2*	**15**	13	22	27	**12**	10	20	25
i = 6, j = 3*	**15**	13	12	27	22	**10**	20	25
i = 7, j = 4	**15**	13	12	10	22	27	**20**	25
i = 8, j = 4	**15**	13	12	10	22	27	20	**25**

- Después del bucle, se intercambia el pivote con S[j]

10	13	12	15	22	27	20	25

(* Diferente respecto a [NN98])

Quicksort: Análisis de complejidad de `partition`

- Vamos a utilizar la técnica de la instrucción característica para estudiar la complejidad de `partition`
- ¿Qué instrucción se puede utilizar como instrucción característica?
- Suponemos que n es el número de elementos del *subarray* que estamos considerando ($n = sup - inf + 1$)

a) En el caso mejor: Como se recorren todos los elementos del *array*, podemos obtener una medida de la complejidad:

$$T_{partition}(n) = n - 1 \in \Theta(n)$$

b) En el caso peor ocurre lo mismo. Por tanto, en cualquier caso la complejidad es lineal

Quicksort: Análisis de complejidad de `quicksort`

- Primero estudiamos el caso peor. Este caso se produce cuando la división en *subarrays* está muy desequilibrada: cuando el *array* ya está ordenado (en orden creciente o decreciente)
- En este caso, la complejidad se puede expresar como la siguiente recurrencia:

$$T(n) = T(0) + T(n-1) + n - 1$$

  Cuando el *array* está vacío, $T(0)$ es simplemente el coste de la propia llamada más el de comprobar que *inf* $\not<$ *sup* (un coste constante)
- Podemos resolver esta recurrencia fácilmente, o bien clasificarla utilizando la **reducción por sustracción**:

$$T(n) \in \Theta(n^2)$$

Quicksort: Análisis de complejidad de `quicksort` (cont.)

- Vamos a estudiar la complejidad de `quicksort` en el caso medio
- Suponemos que todas las permutaciones de S son igualmente probables. Es importante que esta suposición sea correcta: el comportamiento puede ser muy diferente en otro caso
- La recurrencia en este caso se puede definir como:

$$A(n) = \sum_{p=1}^{n} \frac{1}{n}[A(p-1) + A(n-p)] + n - 1$$

- Se puede comprobar que $\sum_{p=1}^{n}[A(p-1) + A(n-p)] = 2\sum_{p=1}^{n} A(p-1)$
- Por tanto,

$$A(n) = \frac{2}{n}\sum_{p=1}^{n} A(p-1) + n - 1$$

Quicksort: Análisis de complejidad de `quicksort` (cont.)

- Podemos multiplicar esta ecuación por n

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1) \tag{1}$$

- Si aplicamos la ecuación (1) a n-1 tenemos:

$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2) \tag{2}$$

- Restando (2) de (1), se obtiene

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

- Se puede despejar $A(n)$:

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

Quicksort: Análisis de complejidad de `quicksort` (cont.)

- Si se hace la sustitución $a_n = \frac{A(n)}{n+1}$ se obtiene:
  $a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)}, n > 0$
- Esta recurrencia no se puede resolver por los métodos que conocemos, pero podemos aproximarla de alguna forma. Como $\frac{n-1}{n+1} < 1$, la recurrencia anterior se puede aproximar por $a'_n = a'_{n-1} + \frac{2}{n}, n > 0$
- Desplegando esta recurrencia,

$$a'_n = c + \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n} = c + 2\sum_{i=2}^{n} \frac{1}{i} \approx c + 2\ ln\ n$$

- De este modo,

$$A(n) \approx (n+1)(c + 2\ ln\ n) = (n+1)(c + 2\ ln\ 2\ lg\ n) \approx$$
$$1{,}38(n+1)lg\ n + (n+1)c \in \Theta(n\ lg\ n)$$

- ¿Qué se puede decir acerca del espacio ocupado por este algoritmo?

Esquemas algorítmicos. Divide y Vencerás

Búsqueda de la mediana de dos *arrays*

([GV00], p. 117)

- Sean dos *arrays* X e Y de tamaño n, ordenados de forma no decreciente. Debe diseñarse un algoritmo para calcular la mediana de los $2n$ elementos de ambos *arrays*
- La mediana de un *array* ordenado de n elementos es el elemento que ocupa la posición $\lfloor \frac{n+1}{2} \rfloor$: el elemento del *array* que tiene tantos elementos a su derecha como a su izquierda (salvo una diferencia de 1)
- Debemos encontrar una forma de aplicar el esquema divide y vencerás: reducir el problema original a subproblemas más pequeños de la misma naturaleza
- ¿Cómo debe ser el subproblema para que se pueda aplicar la misma técnica?

Búsqueda de la mediana de dos *arrays* (cont.)

- Sea Z el *array* resultante de combinar X e Y, y m_z la mediana de Z, que es lo que queremos calcular
- Sean m_x y m_y las medianas de X e Y.
- Si $m_x = m_y$, entonces quiere decir que tanto X como Y tienen el mismo elemento en el centro $\implies$ al combinar ambos *arrays*, m_x y m_y continuarán estando en el centro $\implies$ $m_z = m_x = m_y$
- Si no son iguales, supongamos que $m_x < m_y$. En este caso, $m_x \leq m_z \leq m_y$.
- La clave para dividir el problema en subproblemas está en descubrir la siguiente propiedad: **La mediana de un array $S[1..n]$ es la misma que la del array resultante de quitar $2k$ elementos, la mitad a cada lado de la mediana, para valores de $k < \lfloor \frac{n}{2} \rfloor$**
- Por tanto, como sabemos que $m_x \leq m_z \leq m_y$, en el *array* Z podemos descartar k elementos que estén situados a la izquierda de m_x y otros k elementos a la derecha de m_y

Búsqueda de la mediana de dos *arrays* (cont.)

- Para que el algoritmo sea lo más eficiente posible, el subproblema debe ser lo más pequeño posible, eliminando el mayor número de elementos de los arrays: $k = \lfloor \frac{n-1}{2} \rfloor$.
- El segundo aspecto que debe estudiarse es: cuándo se considera un problema suficientemente pequeño, y cómo se resuelve
- Hay dos posibilidades:
  a) los *arrays* tienen un solo elemento: se toma el menor de los dos
  b) los *arrays* tienen dos elementos: se calcula la mediana *ad hoc*
- El algoritmo queda definido de la siguiente forma:

Búsqueda de la mediana de dos *arrays* (cont.)

```
fun mediana(X,infX,supX, Y,infY,supY) // X e Y son arrays de tamaño n
  si infX ≥ supX Y infY ≥ supY entonces devolver min(X[supX],Y[supY])
  si no
    nitems ← supX - infX + 1
    si nitems = 2 entonces
      si X[supX] < Y[infY] entonces devolver X[supX]
      si no si Y[supY] < X[infX] entonces devolver Y[supY]
      si no devolver max(X[infX],Y[infY])
    fin si
    nitems ← ⌊ (nitems - 1) / 2 ⌋
    posX ← infx + nitems ; posY ← infY + nitems
    si X[posX] = y[posY] entonces
      devolver X[posX]
    si no si X[posX]<Y[posY] entonces
      devolver mediana(X,supX-nitems,supX,Y,infY,infY+nitems)
    si no
      devolver mediana(X,infX,infX+nitems,Y,supY-nitems,supY)
    fin si
  fin si
fin fun
```

mediana de dos *arrays*: análisis de complejidad

- Se puede observar en la definición del algoritmo que este problema es un *problema de simplificación*: se genera un único subproblema de menor tamaño
- Estudiamos la complejidad en el caso peor
- En cada ejecución de la función `mediana` para un tamaño n de los *arrays* mayor a 2, se realiza una única llamada recursiva, con *arrays* de tamaño $n/2$
- Por tanto, podemos expresar su tiempo de ejecución como

$$T(2n) = T(n) + A$$

  donde A es una constante.
- Por el teorema de la reducción por división ($a = 1$, $b = 2$, $k = 0$, $1 = 2^0$),

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

Esquemas algorítmicos. Divide y Vencerás

1. Características generales de la técnica Divide y Vencerás
2. Esquema general y estudio de complejidad
3. *Mergesort*
4. *Quicksort*
5. Búsqueda de la mediana de dos *arrays*
6. El elemento en su posición
7. Búsqueda del elemento mayoritario
8. Multiplicación de matrices

El elemento en su posición

([GV00], p. 119)

- Dado un *array* ordenado A de enteros **todos distintos**, el problema consiste en diseñar un algoritmo de complejidad $\mathcal{O}(log\ n)$ en el peor caso que sea capaz de encontrar un índice $1 \leq i \leq n$ tal que $A[i] = i$, si existe tal índice.

El elemento en su posición (cont.)

- Podemos aprovechar que el array está ordenado para aplicar un algoritmo parecido al de la búsqueda binaria
- Si tomamos el elemento del centro del *array*, $A[i], i = \lfloor \frac{n+1}{2} \rfloor$, se pueden dar tres casos distintos:

a) Si $A[i] = i$, ya hemos encontrado un elemento en su posición

b) Si $A[i] < i$, no hace falta que busquemos entre los elementos de las posiciones anteriores a i, pues los elementos del *array* deben ser distintos y A está ordenado

c) Si $A[i] > i$, no hace falta buscar entre los elementos de las posiciones posteriores a i

- Como se puede comprobar, es un problema de *simplificación*

El elemento en su posición (cont.)

El algoritmo queda de la siguiente forma:

fun elem_en_pos(A[inf..sup])
  si inf $>$ sup **entonces**
    devolver 0
  si no
    i $\leftarrow \lfloor (inf + sup + 1)/2 \rfloor$
    si A[i] $=$ i **entonces**
      devolver i
    si no si A[i] $>$ i **entonces**
      devolver elem_en_pos(A[inf..i-1])
    si no
      devolver elem_en_pos(A[i+1..sup])
    fin si
  fin si
fin fun

El elemento en su posición: análisis de complejidad

- Podemos comprobar fácilmente que la complejidad de este algoritmo en el peor caso está en $\Theta(lg\ n)$
- Cuando el tamaño del *array* es 0, el valor de la función de complejidad es una constante c_1
- Cuando el tamaño de A es n, se ejecuta un número constante de instrucciones (acotado por una constante c_2) más una llamada recursiva con un *array* de tamaño mitad
- Por tanto, la función de complejidad temporal es:

$$T(n) = \begin{cases} c_1 & si\ \ n = 1 \\ T(n/2) + c_2 & si\ \ n > 1 \end{cases}$$

- Por el teorema de reducción por división, $T(n) \in \Theta(lg\ n)$

Esquemas algorítmicos. Divide y Vencerás

Búsqueda del elemento mayoritario

([GV00], p. 121)

- Dado un *array* A de tamaño n, se denomina *elemento mayoritario* el elemento que aparece repetido en A más de $n/2$ veces
- Se debe implementar un algoritmo que permita determinar de forma eficiente el elemento mayoritario de A, si éste existe

Búsqueda del elemento mayoritario (cont.)

- Forma sencilla pero costosa de hacerlo:
  - Recorrer el vector hasta la mitad menos uno, y por cada elemento, recorrer lo que queda del vector para comprobar si es mayoritario
  - La complejidad es cuadrática
- Otra forma más eficiente:
  - Ordenar el vector
  - Si el vector tiene elemento mayoritario, éste estará en la mediana
  - por tanto, hay que elegir el elemento de la mediana y comprobar que efectivamente es el elemento mayoritario
  - La complejidad está en el orden de $n \lg n$
- Otra forma más, en el orden de $n \lg n$, consiste en aplicar la técnica divide y vencerás, dividiendo el *array* en dos y obteniendo el elemento mayoritario de cada *subarray*.
  - En este caso también hay que comprobar que el elemento mayoritario de un *subarray* lo es también del *array*.
  - Esto hace que en cada llamada recursiva se realice la comprobación, con coste lineal

Búsqueda del elemento mayoritario (cont.)

- Sin embargo, podemos hacerlo algo mejor (con coste lineal)
- Se puede buscar un candidato utilizando la técnica divide y vencerás, y después comprobar que efectivamente es el elemento mayoritario
- Podemos comparar los elementos del *array* dos a dos, y determinar si estos *arrays* tienen elemento mayoritario
- En un *array* de 2 elementos, existe elemento mayoritario si ambos elementos son iguales
- Si hacemos esto con todos los pares, obtenemos un *array* de tamaño $\leq n/2$ (sólo consideramos los elementos mayoritarios de los pares que tienen elemento mayoritario)
- Podemos aplicar recursivamente esta técnica al *array* de los elementos mayoritarios
- ¿Qué ocurre si el tamaño n del *array* A es impar? se considera el array de tamaño $n-1$, y si éste no tiene candidato, se utiliza $A[n]$ como candidato

Búsqueda del elemento mayoritario (cont.)

- El algoritmo queda como sigue:

  //Devuelve *cierto* si `candidato` es el elemento mayoritario
  fun mayoritario(A[inf..sup],candidato)
    suma $\leftarrow$ 0
    crear B[inf..sup]
    B[inf..sup] $\leftarrow$ A[inf..sup] //Errata en [GV00]
    si buscar_candidato(B[inf..sup],candidato) **entonces**
      //comprobación del candidato
      desde i $\leftarrow$ inf **hasta** sup **hacer**
        si A[i]=candidato **entonces** suma $\leftarrow$ suma + 1
      fin desde
    fin si
    devolver suma $> \lfloor (\text{sup-inf}+1)/2 \rfloor$
  fin fun

- El algoritmo para `buscar_candidato` es el siguiente:

Búsqueda del elemento mayoritario (cont.)

```
fun buscar_candidato(A[inf..sup],candidato) // Devuelve cierto si existe
   candidato ← A[inf]
   si inf>sup entonces devolver falso
   si inf=sup entonces devolver cierto
   j ← inf
   si (sup-inf+1) mod 2 = 0 entonces
      desde i ← inf+1 hasta sup sumando 2 hacer
         si A[i-1]=A[i] entonces A[j] ← A[i] ; j ← j+1
      fin desde
      devolver buscar_candidato(A[inf..j-1],candidato)
   si no
      desde i ← inf+1 hasta sup-1 sumando 2 hacer
         si A[i-1]=A[i] entonces A[j] ← A[i] ; j ← j+1
      fin desde
      si ¬ buscar_candidato(A[inf..j-1],candidato) entonces candidato ← A[sup]
      devolver cierto
   fin si
fin fun
```

Búsqueda del elemento mayoritario: Análisis de complejidad

- Primero estudiamos la complejidad de `buscar_candidato`
- En el caso peor, el *array* generado en cada llamada recursiva es de tamaño $n/2$
- Además, en cada llamada recursiva se realiza un recorrido del *array*
- En el caso base se hace un número constante de operaciones
- Por tanto, la función de complejidad es de la forma:

$$T_{buscar\_candidato}(n) = \begin{cases} c_1 & si \;\; n \leq 1 \\ T_{buscar\_candidato}(n/2) + p(n) & si \;\; n > 1 \end{cases}$$

  donde $p(n)$ es un polinomio de grado 1. Aplicando el teorema de reducción por división, $1 < 2^1 \implies T_{buscar\_candidato}(n) \in \Theta(n)$
- Ahora estudiamos `mayoritario`: la función de complejidad es $T_{mayoritario}(n) = T_{buscar\_candidato}(n) + c_1 n + c_2$, donde c_1 es el número de operaciones elementales en cada iteración del bucle, y c_2 es el resto de operaciones del procedimiento. Por tanto:

$$T_{mayoritario}(n) \in \Theta(n)$$

Búsqueda del elemento mayoritario (cont.)

- En el desarrollo que hemos hecho del algoritmo no es fácil ver que funciona en todos los casos. Vamos a comprobar que lo que hacemos es correcto
- En cada llamada recursiva del algoritmo anterior se aplican las siguientes reglas que generan un *array* B de menor tamaño con el mismo elemento mayoritario. Para cada $i = 1, ..., \lfloor n/2 \rfloor$,
  1. Si $A[2i-1] \neq A[2i]$, se pueden eliminar estos dos elementos en B
  2. Si $A[2i-1] = A[2i]$, solo uno de estos dos elementos aparece en B
  3. Si el número de elementos es impar, se busca el elemento mayoritario del *array* $A[1..n-1]$ utilizando las reglas anteriores. Si existe, éste es el elemento mayoritario de $A[1..n]$; si no existe, se elige como candidato a $A[n]$
- Debemos demostrar lo siguiente:

(1) Si el *array* $A[1..n]$ tiene un elemento mayoritario M, entonces el algoritmo anterior proporciona M como elemento candidato

Búsqueda del elemento mayoritario (cont.)

- Primero vemos la corrección de un paso de `buscar_candidato`:

(2) Si $A[1..n]$ tiene elemento mayoritario M, entonces el *array* B, resultado de aplicar a A las reglas anteriores, tiene a M como elemento mayoritario

- Tratamos cada regla por separado. Primero vemos el caso en el que n es par:
  1) Si se elimina de A un par de elementos distintos, a lo sumo uno de ellos es igual a M. El *array* pasa a tener $n-2$ elementos, de los cuales como mínimo $\frac{n}{2}+1-1$ tienen el valor M. Como $\frac{n}{2}=\frac{n-2}{2}+1$, M es mayoritario después de aplicar esta regla
  2) Una vez aplicada la regla 1 para todos los pares de elementos distintos, queda un *array* de tamaño n' con $n'/2$ pares de elementos iguales y en el que M es mayoritario: M aparece en este *array* al menos $n'/2+1$ veces. Podemos obtener un *array* (de tamaño $n'/2$) en el que haya un solo representante de cada par, y en el que M aparezca al menos $\lceil (n'/2+1)/2 \rceil > n'/4$ veces, y por tanto sea mayoritario en este nuevo *array*. ($\lceil \cdot \rceil$ porque M aparece un número par de veces)

Búsqueda del elemento mayoritario (cont.)

- Si n es impar:
  3) Si el número de elementos del *array* es impar y tiene elemento mayoritario M, éste puede estar en la última posición del *array* o no.
     - ⋆ Si $A[1..n-1]$ tiene elemento mayoritario M, éste es el mismo que en $A[1..n]$, pues $n-1$ es par y M debe aparecer al menos $\frac{n-1}{2}+1 = \lfloor \frac{n}{2} \rfloor + 1$ veces
     - ⋆ Si $A[1..n-1]$ no tiene elemento mayoritario y $A[1..n]$ sí lo tiene, entonces el elemento mayoritario de $A[1..n]$ está exactamente $(n-1)/2$ veces en $A[1..n-1]$ y además está en $A[n]$. Por tanto, podemos tomar el valor de $A[n]$ como candidato
- Por tanto, **podemos afirmar que es cierta la implicación (2)**
- Aplicando sucesivas veces esta implicación, obtenemos finalmente un *array* de tamaño 1 o 2, para el que se puede obtener trivialmente el elemento mayoritario, por lo que también **hemos demostrado que la implicación (1) es cierta**
- Pudiera ocurrir que `buscar_candidato` proporcione un candidato que no es elemento mayoritario. Este caso solamente ocurrirá si A no tiene elemento mayoritario

Esquemas algorítmicos. Divide y Vencerás

1. Características generales de la técnica Divide y Vencerás
2. Esquema general y estudio de complejidad
3. *Mergesort*
4. *Quicksort*
5. Búsqueda de la mediana de dos *arrays*
6. El elemento en su posición
7. Búsqueda del elemento mayoritario
8. Multiplicación de matrices

Multiplicación de matrices

- El algoritmo tradicional de multiplicación de matrices $n \times n$ realiza n multiplicaciones por cada elemento de la matriz producto. Por tanto, está en el orden de $\mathcal{O}(n^3)$
- La complejidad de las multiplicaciones es bastante mayor que la de las sumas o restas, por lo que los algoritmos de multiplicación de matrices se centran en reducir el número de multiplicaciones
- A finales de los años 60, Strassen desarrolló un algoritmo con complejidad mejor que cúbica
- En el caso del producto de matrices 2×2

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Multiplicación de matrices (cont.)

- El producto de estas matrices es:

$$\begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

  donde

$$\begin{aligned}
m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_2 &= (a_{21} + a_{22})b_{11} \\
m_3 &= a_{11}(b_{12} - b_{22}) \\
m_4 &= a_{22}(b_{21} - b_{11}) \\
m_5 &= (a_{11} + a_{12})b_{22} \\
m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
m_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}$$

Multiplicación de matrices (cont.)

- En el caso de matrices 2×2 la mejora es muy pequeña: sólo se ahorra una multiplicación, y se incrementa el número de sumas y restas
- La ventaja de este método es que no utiliza la propiedad conmutativa del producto, por lo que se puede aplicar a submatrices $\Longrightarrow$ se puede aplicar la técnica divide y vencerás
- De esta forma, el producto de dos matrices $n \times n$ (suponemos n potencia de 2) se puede ver como el producto de dos matrices 2×2 en las que cada "elemento" es una matriz $n/2 \times n/2$:

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

donde por ejemplo A_{11} es de la forma

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n/2} \\ a_{21} & a_{22} & \cdots & a_{2,n/2} \\ & & \vdots & \\ a_{n/2,1} & a_{n/2,2} & \cdots & a_{n/2,n/2} \end{bmatrix}$$

Multiplicación de matrices (cont.)

- Una especificación informal del algoritmo ([NN98], p. 69) puede ser la siguiente:

proc strassen(A[1..n][1..n],B[1..n][1..n],C[1..n][1..n])
  si n $\leq$ umbral **entonces**
    calcular $C = A \times B$ utilizando el algoritmo estándar
  si no
    //calcular $C = A \times B$ utilizando el algoritmo de Strassen
    // particionando A y B en submatrices de tamaño $n/2 \times n/2$.
    // Por ejemplo, para calcular M_1:
    $S_1 \leftarrow$ A[1..n/2][1..n/2] + A[n/2+1..n][n/2+1..n]
    $S_2 \leftarrow$ B[1..n/2][1..n/2] + B[n/2+1..n][n/2+1..n]
    strassen(S_1,S_2,M_1)
    // ...
  fin si
fin proc

Multiplicación de matrices: Análisis de complejidad

- Se puede estudiar la complejidad respecto al número de multiplicaciones de este algoritmo
- En cada ejecución de `strassen` se realizan:
  - 7 llamadas recursivas, cada una con un problema de tamaño $n/2$
  - un número constante de sumas de matrices de tamaño $n/2 \times n/2$, cada una de ellas de complejidad $\Theta(n^2)$
  - En el caso base, un número constante de operaciones
- Por tanto, la función de complejidad será de la forma:

$$T(n) = \begin{cases} c_1 & si\ \ n \leq umbral \\ 7T(n/2) + p(n) & si\ \ n > umbral \end{cases}$$

  donde $p(n)$ es un polinomio de grado 2. Aplicando el teorema de reducción por división, $7 > 2^2$, por lo que $T(n) \in \Theta(n^{lg\ 7}) \approx \Theta(n^{2,81})$