Tema 2. Divide y Vencerás

Tabla de Contenidos

.:: Miguel Ángel Redondo ::.

Aplicación

- Diseño de algoritmos eficaces
- Algoritmos artificiosos
- Problemas con tamaño mediano

Planteamiento

- Algoritmo **A** con $O(n^2)$ $\Rightarrow$ $\exists c / T_A(n) \leq c * n^2$
- Si *n* es grande dividimos el problema en dos subproblemas de tamaño *n/2* cada uno
- Resolvemos cada subproblema aplicando **A**
- Obtenemos la solución combinando las soluciones de cada subproblema

Introducción al método

Consideraciones

- Tiempos para dividir y combinar pequeños comparados con el tiempo de **A**
  - Lo ideal es que fuesen tiempos lineales

$$T_D(n) \leq c'*n$$

Un primer diseño del algoritmo (*B*)

1. Dividir el problema *P* en dos subproblemas P_1 y P_2
2. Resolver P_1 mediante ***A*** obteniendo S_1
3. Resolver P_2 mediante ***A*** obteniendo S_2
4. Combinar S_1 y S_2 para obtener *S*

Estudio de eficiencia

$$T_A(n) \leq c * n^2$$

$$T_B(n) = 2 * c * T_A(n) + T_D(n)$$

$$T_B(n) \leq c * \frac{n^2}{4} + c'*n$$

$$T_B(n) \leq \frac{c}{2} * n^2 + c'*n$$

Aplicación
Situaciones de *n* grande para que *c'*n* sea despreciable

Reiteración en la descomposición

- Algoritmo **C**

1. Si el problema es pequeño resolver directamente, retomar la solución y parar
2. Dividir el problema P en dos subproblemas P_1 y P_2
3. Resolver P_1 mediante **C** obteniendo S_1
4. Resolver P_2 mediante **C** obteniendo S_2
5. Combinar S_1 y S_2 para obtener S

Reiteración en la descomposición

- Algoritmo **C** – Estudio de la eficiencia

$$T_C(n) = \begin{cases} T_A(n) & \text{si n es pequeño} \Rightarrow \mathrm{O}(1) \\ \\ c'*n*T_C(n/2) & \text{en otro caso} \end{cases}$$

$$T_C(n) = O(n*\log n) < O(n^2) = T_A(n)$$

Búsqueda dicotómica

- Array ordenado ***A*** con *n* números enteros
- Determinar si un valor *x* está en ***A***

?

	1	2	3	4	5	6	7	8	9	10		*n*
A							*x*				....	

```
i = 0;
encontrado = 0;
while ( (i<n) && (encontrado==0) ) {
        i++;
        if (x == A[i]) encontrado = 1;
}
```

O(n)

Ejemplos clásicos

Búsqueda dicotómica

- Aplicación del esquema *Divide y Vencerás (DAC)*

Sentido descendente

```
i = 0;   j=n;
encontrado = 0;
while ( (i<j) && (encontrado==0) ) {
        k = (i+j) / 2
        if (x == A[k])  encontrado = 1;
        else {
                if (x < A[k])    j = k-1;
                else             i = k+1;
        }
}
```

O(log n)

Mínimo elemento de un array

```
minimo = A[1];
for (i=2; i<=n; i++) {
        if (minimo > A[i]
                minimo = A[i];
}
```

O(n)

Mínimo elemento de un array

- Aplicación del esquema *Divide y Vencerás (DAC)*

Sentido ascendente

1. Obtener el mínimo de cada una de las dos mitades del array
2. Elegir el menor
3. Reducir el problema a dos subproblemas de tamaño la mitad
4. Repetir la operación
5. Seguir hasta que sólo queden dos pares de elementos

Sentidos

- **Ascendente** *(DAC1)*
  - Primero se desciende para plantear todos los subproblemas
  - Después se asciende para ir construyendo la solución
  - Es necesario hacer algún tipo de operación con las soluciones de los subproblemas que se van originando

  ---

  - Partir el problema en subproblemas más pequeños
  - Combinar sus soluciones
  - Obtener la solución al problema original
- **Descendente** *(DAC2)*
  - Se parte del problema original
  - Se plantea la solución como la de uno o más subproblemas
  - No requiere combinación

Forma general de un DAC

Versión recursiva

```
DAC (int tamanyo) {
    if ( facil(tamanyo )
        return (resolver(tamanyo));
    else {
        dividir(tamanyo);
        /* suponemos k divisiones */
        return (combinar (DAC(n1), ...,DAC(nk));
    }
}
```

Forma general de un DAC

Funciones utilizadas

- ***facil*** (n)
  - Es una función booleana que es cierta cuando el problema es realizable
- ***resolver*** (n)
  - Procedimiento que da la solución si *facil*(n)
- ***dividir*** (n)
  - Obtiene tamaños de problema menores que *n* en el caso de *no facil(n)*
  - Los nuevos tamaños son n1, n2, ..., nk
- ***combinar*** (s1, s2, ..., sk)
  - Combina las subsoluciones para obtener otra subsolución (o solución si estamos en el primer nivel)

Determinan si será DAC1 o DAC2

Forma general de un DAC

Versión iterativa

```
DAC (int tamanyo) {
    while (tamanyo >= 1) {
        for (i=1; i<=k; i++)
            s[i] = resolver(n_i);
        combinar (s);
        modificar(tamanyo);
    }
}
```

Funciones utilizadas

- g(n)
  - Para *n* pequeño: *facil (n)* + *resolver (n)*
- f(n)
  - Para *n* grande: *dividir (n)* + *combinar (s_i)* + *facil (n)*

$$T(n) = \begin{cases} g(n) & \text{si } facil(n) \\ f(n) + \sum_{i=1}^{k} T(n_i) & \text{si no } facil(n) \end{cases}$$

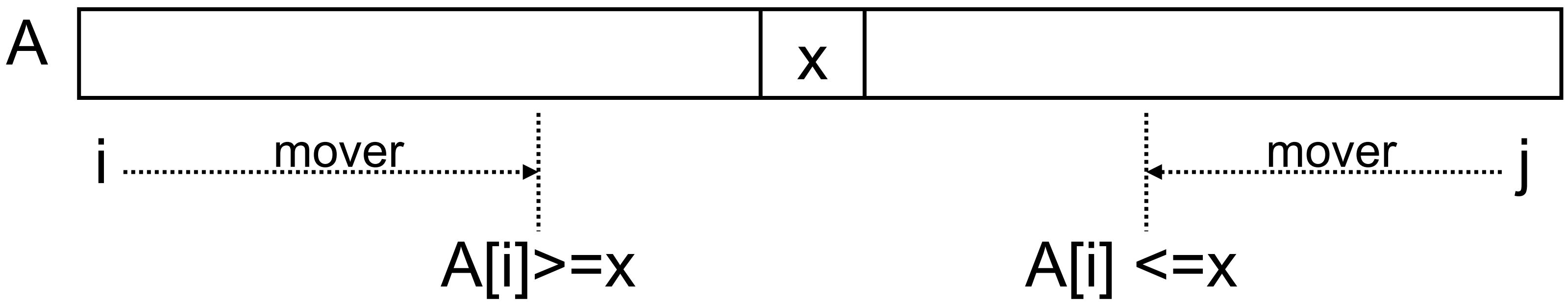
Ejemplos representativos

Métodos de ordenación

- Interna
  - Inserción
    - Inserción binaria, SHELL
  - Selección
    - HEAPSORT
  - Intercambio
    - **QUICKSORT**
- Externa
  - **MERGESORT**

QUICKSORT

- C.A.R. Hoare (1962)
- Técnica de intercambio
- DAC2
- **Planteamiento**
  - Tomar un elemento y definitivamente ponerlo en la posición correcta
    - Todos los menores que él se sitúan a un lado
    - Todos los mayores que él se sitúan a otro lado



Esquema general

1. Elegir “x”
2. Mover los menores que “x” a un lado (izquierda)
3. Mover los mayores que “x” a otro lado (derecha)
4. Tenemos a “x” colocado
5. Colocar los de la izquierda
6. Colocar los de la derecha

QUICKSORT


```
void QuickSort (Type v[], int li, int ls) {
 // Suponemos que v[ls+1]>=v[k], li<=k<=ls
   if (li < ls) {
   // si tenemos más de un elemento en el vector
   // lo dividimos en dos subproblemas
   int pos = Divide (v,li,ls+1)
   // pos es la posición del pivote
   // Resolvemos los subproblemas
   QuickSort (v,li,pos-1);
   QuickSort (v,pos+1,ls);
   // No es necesario combinar las soluciones
   }
}
```

QUICKSORT


```
int Divide (Type v[], int li, int ls) {
// Colocamos los elementos del segmento v[li..ls-1]
// asumimos que v[ls] >= v[k] para todo k, li <= k <= ls-1
// Elegimos un elemento pivote y colocamos los otros elementos
// con respecto a él, de tal forma que devolvemos una posición p
// del vector que verifica pivote = v[p]
// v[k] <= pivote, li <= k <= p, pivote <= v[k], p <= k <= ls
   Type pivote = v[li];
   int izq = li, der = ls;
   do {
      while (v[++izq] < pivote);
      while (v[--der] > pivote);
      if (izq < der) Intercambia (&v[izq],&v[der]);
   } while (izq < der);
   Intercambia (&v[li],&v[der]);
   // Hasta “der” son <= que pivote, y a partir de “der” son >=
   return der;
}
```

Estudio de la complejidad

- Complejidad *facil (n)* $\Rightarrow$ O(1)
- Complejidad *dividir (n)* $\Rightarrow$ O(n)
- Complejidad *combinar (n)* $\Rightarrow$ O(1)
- Llamadas recursivas
  - Suponemos que el vector se divide en *k* (1≤k ≤(n-1))
  - $T_a(k)+T_a(n-k)$ siendo $T_a(n)$ la complejidad media

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

Estudio de la complejidad

- **Caso medio** (todos los posibles valores de *k*)

$$T_a(n) = T(n) = \begin{cases} 1 & \text{si n} = 1 \\ n + \dfrac{1}{n-1}\displaystyle\sum_{k=1}^{n-1}\left(T_a(k) + T_a(n-k)\right) & \text{si n} > 1 \end{cases}$$

- Como T(k) aparece dos veces

$$T(n) = n + \frac{1}{n-1}\sum_{k=1}^{n-1}\left(2T(k)\right)$$

- Multiplicamos por (n-1) con n>1

$$(n-1)T(n) = n(n-1) + \sum_{k=1}^{n-1}\left(2T(k)\right) \qquad \textbf{[A]}$$

Estudio de la complejidad (*cont.*)

- Consideramos n>2 y sustituimos n por (n-1)

$$(n-2)T(n-1) = (n-1)(n-2) + \sum_{k=1}^{n-2}\left(2T(k)\right) \qquad \textbf{[B]}$$

- Hacemos [A]-[B] y obtenemos

$$(n-1)T(n) = 2(n-1) + nT(n-1)$$

- Dividimos por n(n-1) y queda

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

- Esto es una expresión recurrente (n>1) que podemos desarrollar

Estudio de la complejidad (*cont.*)

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

$$= 2\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) + T(1) = T(1) + 2\sum_{i=2}^{n}\frac{1}{i} = O(\log n)$$

- Luego $T(n) = O(n \log n)$ para el caso medio

Estudio de la complejidad

- **Caso mejor** (pivote=mediana)

$$T_m(n) = n + 2T_m(n/2)$$

$$T_m(n) = O(n \log n)$$

  - Prácticamente en todos los casos se sigue el mismo comportamiento de **O(n log n)**

Esquema general

1. Dividir en n/2
2. Ordenar (1....n/2) y obtener A
3. Ordenar (n/2....n) y ordenar B
4. Mezclar (A,B) y obtener V

MERGESORT


```
void MergeSort (Type v[], int li, int ls) {
   if (li < ls) {
      // si más de un elemento en el vector
      // lo dividimos en dos subproblemas
      int med = (li+ls)/2;
      // Resolvemos los subproblemas
      MergeSort (v,li,med);
      MergeSort (v,med+1,ls);
      // Combinamos las soluciones:
      // mezclamos los subvect. ordenados
      Mezcla (v,li,med,ls)
   }
}
```

MERGESORT

```
void Mezcla (Type v[], int li, int med, int ls) {
   // w es un vector del mismo tipo que v
   int rec = li, rec1 = li, rec2 = med+1, k;
   while ((rec1 <= med) && (rec2 <= ls)) {
      if (v[rec1] < v[rec2]) {
            w[rec] = v[rec1];       rec1++;
      } else {
            w[rec] = v[rec2];       rec2++;
      }
      rec++;
   }
   for (k=rec1; k<=med; k++) { w[rec] = v[k]; rec++; }
   for (k=rec2; k<=ls; k++) { w[rec] = v[k]; rec++; }
   // Ahora volcamos w en v
   for (k=li; k<=ls; k++) v[k] = w[k];
}
```

Estudio de la complejidad

- Complejidad *facil (n)* $\Rightarrow$ O(1)
- Complejidad *dividir (n)* $\Rightarrow$ O(n)
- Complejidad *combinar (n)*
  - Dos llamada recursivas con tamaño (n/2) $\Rightarrow$ 2T(n/2)
  - Mezcla: tamaño de los tramos a mezclar $\Rightarrow$ O(n log n)

$$T(n)=\left\{\begin{array}{cc} 1 & n=1 \\ n+2T(n/2) & n>1 \end{array}\right| \Rightarrow O(n\log n)$$