

# Análisis y diseño de algoritmos

## 3. El diseño de algoritmos. Divide y vencerás

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos  
Universidad de Alicante

30-01-2018 (389)



1 El diseño de algoritmos

2 Divide y vencerás



# El diseño de algoritmos: objetivos

- Dar a conocer las familias más importantes de problemas algorítmicos y estudiar diferentes esquemas o paradigmas de diseño aplicables para resolverlos.
- Aprender a instanciar (particularizar) un esquema genérico para un problema concreto, identificando los datos y operaciones del esquema con las del problema, previa comprobación de que se satisfacen los requisitos necesarios para su aplicación.
- Justificar la elección de un determinado esquema cuando varios de ellos pueden ser aplicables a un mismo problema.



# El diseño de algoritmos: definición

- El diseño de algoritmos estudia la aplicación de métodos para resolver problemas en programación.
- La resolución de problemas:
  - Diseño **ad hoc** (frecuentemente “fuerza bruta”)
    - Algoritmos dependientes del problema y no generalizables
    - Dificultad de adecuar cambios en la especificación
  - Esquemas:
    - Cada esquema representa un grupo de algoritmos con características comunes (análogos)
    - Permiten la generalización y reutilización de algoritmos
    - Cada instanciación de un esquema da lugar a un algoritmo diferente



## Esquemas algorítmicos más comunes

- Divide y vencerás (*divide and conquer*)
- Programación dinámica (*dynamic programming*)
- Algoritmos voraces (*greedy method*)
- Algoritmos de búsqueda y enumeración
  - Vuelta atrás (*backtracking*)
  - Ramificación y poda (*branch and bound*)
- Algoritmos probabilísticos y heurísticos<sup>1</sup>
  - Algoritmos probabilísticos
  - Algoritmos heurísticos
  - Algoritmos genéticos

---

<sup>1</sup>No se tratarán en la asignatura

1 El diseño de algoritmos

2 Divide y vencerás



# Ordenación por selección directa

- Ordenar de forma ascendente un vector  $v$  de  $n$  elementos.

```
1 void selection_sort( Elem v[], int n) {  
2     for( int i=0; i<n-1; i++) {  
3         int m = i;  
4         for( int j = i+1; j<n; j++)  
5             if (v[j] < v[m])  
6                 m = j;  
7         swap(v[i],v[m]);  
8     }  
9 }
```

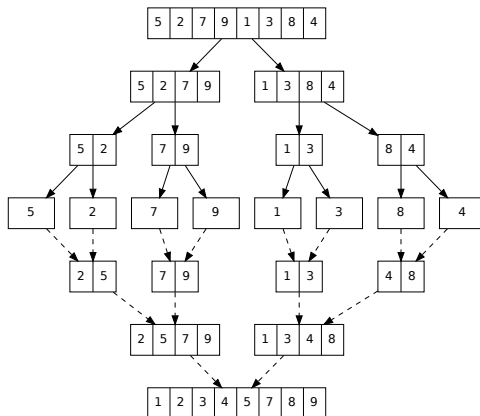
- El bucle de la línea 3 se ejecuta  $n - 1$  veces y el bucle de la línea 4 se ejecuta  $n - i - 1$  veces con  $i \in [0, n - 2]$ .
- La línea 5 se ejecuta  $\sum_{i=0}^{n-2} (n - i - 1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n - 1)$  veces.
- La línea 7 se ejecuta  $n$  veces.

Complejidad:  $f(n) \in \Theta(n^2)$



# Algoritmo Mergesort: idea

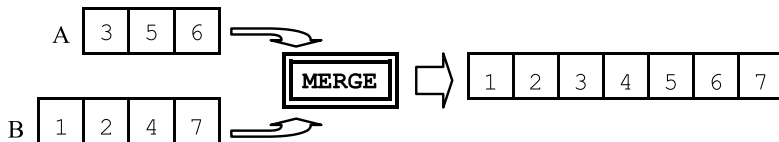
- Ordenar de forma ascendente un vector  $V$  de  $n$  elementos.
- Solución usando el esquema “divide y vencerás”:





# Algoritmo Mergesort: función merge

- El algoritmo mergeSort utiliza la función merge que obtiene un vector ordenado como fusión de dos vectores también ordenados



## Mergesort

```
1 void mergeSort(Elem v[], int pi, int pf) {  
2     if ( pi < pf ) { // pi==pf quiere decir 1 elemento  
3         int m = (pi+pf)/2;  
4         mergeSort(v, pi, m);  
5         mergeSort(v, m+1, pf);  
6         merge(v, pi, m, pf);  
7     }  
8 }
```

$\text{merge}(v, pi, m, pf) \in \Theta(n)$  donde  $n = pf - pi + 1$ .

# Algoritmo *Mergesort*: Complejidad

- Talla:  $n$  ( $n = p_f - p_i + 1$  : número de elementos del vector)
- Ecuación de recurrencia (coste exacto):

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

- Complejidad temporal:  $f(n) \in \Theta(n \log n)$



# Algoritmo *Mergesort*: Complejidad

- Talla:  $n$  ( $n = p_f - p_i + 1$  : número de elementos del vector)
- Ecuación de recurrencia (coste exacto):

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

- Complejidad temporal:  $f(n) \in \Theta(n \log n)$

¿Cuál es la complejidad espacial?



# Algoritmo *Mergesort*: Complejidad

- Talla:  $n$  ( $n = p_f - p_i + 1$  : número de elementos del vector)
- Ecuación de recurrencia (coste exacto):

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

- Complejidad temporal:  $f(n) \in \Theta(n \log n)$

## ¿Cuál es la complejidad espacial?

- ¿Cuál es la complejidad espacial de merge?



# Algoritmo *Mergesort*: Complejidad

- Talla:  $n$  ( $n = p_f - p_i + 1$  : número de elementos del vector)
- Ecuación de recurrencia (coste exacto):

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

- Complejidad temporal:  $f(n) \in \Theta(n \log n)$

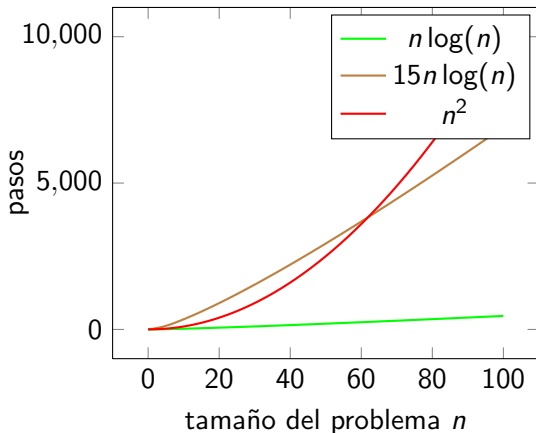
## ¿Cuál es la complejidad espacial?

- ¿Cuál es la complejidad espacial de merge?
- ¡Ojo!: se puede reutilizar la memoria



# Mergesort vs. selección directa

Comparación de algoritmos:



Sólo si el coeficiente de  $n \log n$  es mayor que e puede pasar que para ciertos valores pequeños de  $n$ ,  $n \log n$  sea menos ventajoso que  $n^2$  (asumiendo logaritmos naturales).



# Técnica de divide y vencerás

- Técnica de diseño de algoritmos que consiste en:
  - descomponer el problema en subproblemas de menor tamaño que el original
  - resolver cada subproblema de forma individual e independiente
  - combinar las soluciones de los subproblemas para obtener la solución del problema original
- Consideraciones:
  - No siempre un problema de talla menor es más fácil de resolver
  - La solución de los subproblemas no implica necesariamente que la solución del problema original se pueda obtener fácilmente
- Aplicable si encontramos:
  - Forma de descomponer un problema en subproblemas de talla menor
  - Forma directa de resolver problemas menores a un tamaño determinado
  - Forma de combinar las soluciones de los subproblemas que permita obtener la solución del problema original





# Esquema de Divide y vencerás

## Esquema divide y vencerás

```
1 Solucion DyV( Problema x ) {  
2  
3     if (pequeno(x))  
4         return trivial(x);  
5  
6     list<Solucion> s;  
7     for( Problema q : descomponer(x) )  
8         s.push_back( DyV(q) );  
9     return combinar(s);  
10 }
```



# Mergesort como divide y vencerás

Particularización (**instanciación**) del esquema general para el caso de Mergesort:

- **descomponer**:  $m = (p_i + p_f)/2$
- **trivial**: retorno sin hacer nada si **pequeño** ( $p_i = p_f$ )
- **combinar**: `merge(...)`



# Quicksort

```
1 void quicksort( Elem v[], int pri, int ult ) {  
2  
3     if( ult <= pri )  
4         return;  
5  
6     int p = pri;    // posicion del pivote  
7     int j = ult;  
8     while(p < j) {  
9         if (v[p+1] < v[p]) {  
10             swap( v[p+1], v[p] );  
11             p++;  
12         } else {  
13             swap( v[p+1], v[j] );  
14             j--;  
15         }  
16     }  
17  
18     quicksort(v, pri, p-1);  
19     quicksort(v, p+1, ult);  
20 }
```

# Quicksort como divide y vencerás

Particularización (**instanciación**) del esquema general para el caso de quickSort:

- **descomponer**: cálculo de la posición del elemento pivote
- **trivial**: retorno sin hacer nada si **pequeño** ( $ult \leq pri$ )
- **combinar**: no es necesario



# Análisis de eficiencia (1)

- Eficiencia: costes de logarítmicos a exponenciales.

Depende de:

- N° de subproblemas ( $h$ )
  - Tamaño de los subproblemas
  - Grado de intersección entre los subproblemas
- Ecuación de recurrencia:
    - $g(n)$  = tiempo del esquema para tamaño  $n$ . (sin llamadas recursivas)
    - $b$  = Cte. de división de tamaño de problema

$$T(n) = hT\left(\frac{n}{b}\right) + g(n)$$

- Solución general: suponiendo la existencia de un entero  $k$  tal que:  
 $g(n) \in \Theta(n^k)$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } h < b^k \\ \Theta(n^k \log_b n) & \text{si } h = b^k \\ \Theta(n^{\log_b h}) & \text{si } h > b^k \end{cases}$$



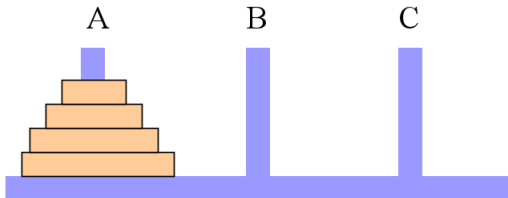
- **Teorema de reducción:** los mejores resultados en cuanto a coste se consiguen cuando los subproblemas son aproximadamente del mismo tamaño (y no contienen subproblemas comunes).
- Si se cumple la condición del teorema de reducción ( $b = h = a$ )

$$T(n) = aT\left(\frac{n}{a}\right) + g(n) \qquad g(n) \in \Theta(n^k)$$

$$T(n) = \begin{cases} \Theta(n^k) & k > 1 \\ \Theta(n \log n) & k = 1 \\ \Theta(n) & k < 1 \end{cases}$$

# Las torres de Hanoi

- Colocar los discos de la torre A en la C empleando como ayuda la torre B
- Los discos han de moverse uno a uno y sin colocar nunca un disco sobre otro más pequeño



- ¿Cómo se afrontaría el problema?
- ¿Cuál sería la complejidad del algoritmo resultante?

# Las torres de Hanoi: solución

- $\text{hanoi}(n, A \xrightarrow{B} C)$  es la solución del problema: mover los  $n$  discos superiores del pivote  $A$  al pivote  $C$ .
- Supongamos que sabemos mover  $n - 1$  discos: sabemos cómo resolver  $\text{hanoi}(n - 1, X \xrightarrow{Y} Z)$ .
- También sabemos como mover 1 disco del pivote  $X$  al  $Y$ :  $\text{hanoi}(1, X \xrightarrow{Y} Z)$ , que es el caso trivial. Lo llamaremos  $\text{mover}(X \rightarrow Z)$ .
- Resolver  $\text{hanoi}(n, A \xrightarrow{B} C)$  equivale a ejecutar:
  - $\text{hanoi}(n - 1, A \xrightarrow{C} B)$
  - $\text{mover}(A \rightarrow C)$
  - $\text{hanoi}(n - 1, B \xrightarrow{A} C)$





# Las torres de Hanoi: Complejidad (1)

Nótese que aquí la talla de los subproblemas no es  $\frac{n}{a}$  sino  $n - 1$ :

- No se pueden aplicar las fórmulas generales de las transparencias anteriores
- Divide y vencerás es aquí más una estrategia de solución (un esquema algorítmico) que una manera de conseguir una solución con menor complejidad
  - El problema tiene una complejidad intrínseca peor que las descritas en las transparencias anteriores



## Las torres de Hanoi: Complejidad (2)

- Ecuación de recurrencia para el coste exacto (asumiendo coste 1 para todas las operaciones de 1 disco):

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

- Solución:

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

$$\stackrel{2}{=} 1 + 2 + 4T(n-2)$$

$$\stackrel{3}{=} 1 + 2 + 4 + 8T(n-3) = \dots$$

$$\stackrel{k}{=} \sum_{i=1}^k 2^{i-1} + 2^k T(n-k) = 2^k - 1 + 2^k T(n-k)$$

- Paramos cuando  $n - k = 1$ , osea, en el paso  $k = n - 1$ ,

$$T(n) = 2^n - 1 \in \Theta(2^n)$$

- ① Selección del  $k$ -ésimo mínimo
  - Dado un vector  $A$  de  $n$  números enteros diferentes, diseñar un algoritmo que encuentre el  $k$ -ésimo valor mínimo.
- ② Búsqueda binaria o **dicotómica**
  - Dado un vector  $X$  de  $n$  elementos ordenado de forma ascendente y un elemento  $e$ , diseñar un algoritmo que devuelva la posición del elemento  $e$  en el vector  $X$ .
- ③ Calculo recursivo de la potencia enésima.

# Selección del $k$ -ésimo mínimo

Dado un vector  $A$  de  $n$  enteros, encontrar el  $k$ -ésimo valor mínimo.

```
1 Elem quickselect( Elem v[],int pri,int ult,int k ) {
2
3     if( ult == pri )
4         return v[k];
5
6     int p = pri; // pivote
7     int j = ult;
8     while(p < j) {
9         if (v[p+1] < v[p]) {
10             swap( v[p+1], v[p] );
11             p++;
12         } else {
13             swap( v[p+1], v[j] );
14             j--;
15         }
16     }
17
18     if( k == p ) return v[k];
19     if( k < p ) return quickselect(v,pri,p-1,k);
20     return quickselect(v,p+1,ult,k);
21 }
```

# Búsqueda binaria o dicotómica

Dado un vector  $v$  ordenado de forma ascendente y un elemento  $e$ , diseñad un algoritmo que devuelva la posición del elemento en el vector.

## Búsqueda binaria

```
1 int bb( Elem v[], int p, int u, Elem e){
2     if( p > u ) return -1 // No hay elementos
3
4     int m = ( p + u ) / 2;
5     if ( e == v[m] ) return m;
6     if ( e < v[m] ) return bb(v,p,m-1,e);
7     return bb(v,m+1,u,e);
8 }
```

- ¿Reconocéis en el algoritmo los componentes del esquema *divide y vencerás*?



- Esta solución se puede ver como un *divide y vencerás* en el que
  - La operación **descomponer** viene representada por  $m = (p + u)/2$
  - El problema **pequeno** corresponde a cuando  $p > u$
  - Sólo se resuelve uno de los dos subproblemas
  - No es necesaria la combinación



# Búsqueda binaria: complejidad

- Ecuación de recurrencia **para el caso peor**:

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

(agrupamos  $n = 0$  en  $n = 1$  porque no se produce división).

- Solución:

$$\begin{aligned} T(n) &\stackrel{1}{=} 1 + T(\frac{n}{2}) \\ &\stackrel{2}{=} 2 + T(\frac{n}{2^2}) \\ &\dots \\ &\stackrel{k}{=} k + T(\frac{n}{2^k}) \end{aligned}$$

- Paramos cuando  $\frac{n}{2^k} = 1$ , o sea en el paso  $k = \log(n)$ ,

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

# Cálculo de la potencia enésima

Si asumimos que multiplicar dos elementos de un determinado tipo tiene un coste constante, es posible calcular la enésima potencia  $x^n$  de un elemento  $x$  de ese tipo en tiempo sublineal usando la siguiente recursión:

$$x^n = \begin{cases} x & n = 1 \\ x^{\frac{n}{2}} x^{\frac{n}{2}} & n \text{ es par} \\ x^{\frac{n-1}{2}} x^{\frac{n-1}{2}} x & n \text{ es impar} \end{cases}$$

Escribid un algoritmo para calcular eficientemente  $x^n$ .

- ¿Se puede evitar repetir operaciones?
- ¿Cuál es el coste asintótico del algoritmo resultante?

