

# Algorítmica

## Tema 2. Algoritmos «divide y vencerás»

---

Francisco Javier Cabrerizo Lorige

Curso académico 2023-2024

ETS de Ingenierías Informática y de Telecomunicación. Universidad de Granada

1. La técnica «divide y vencerás»
2. El problema del umbral
3. Análisis de algoritmos «divide y vencerás»
4. Búsqueda binaria
5. Ordenación rápida
6. Problema de selección
7. Multiplicación de enteros grandes

# Objetivos

- Ser capaz de proponer diferentes soluciones para un determinado problema y evaluar la calidad de estas.
- Ser consciente de la importancia del análisis de la eficiencia de un algoritmo como paso previo a su implementación.
- Entender la técnica de resolución de un problema por división en problemas de menor tamaño.
- Conocer y saber aplicar los esquemas básicos de la técnica divide y vencerás.
- Conocer los criterios de aplicación de cada una de las diferentes técnicas de diseño de algoritmos.

**La técnica «divide y vencerás»**

---

## Características de los problemas

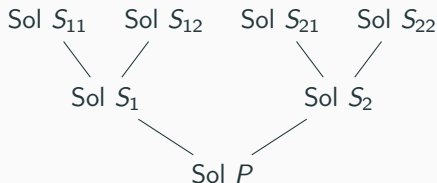
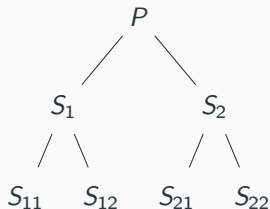
- La instancia del problema se puede descomponer en subinstancias más pequeñas del mismo problema.
- Las subinstancias se pueden resolver independientemente unas de otras (casi siempre).
- Las subinstancias son disjuntas, sin solapamiento.
- La solución de la instancia original se obtienen mediante una combinación de las subsoluciones.

# La técnica «divide y vencerás»

## Pasos

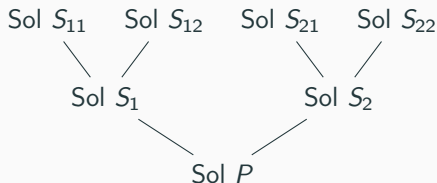
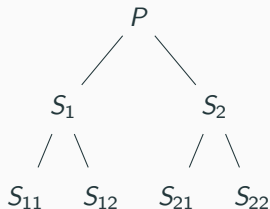
- Descomponer (dividir) la instancia del problema de tamaño  $n$  en  $k$  subinstancias ( $1 \leq k \leq n$ ) del mismo tipo y menor tamaño (cada una con una entrada de tamaño  $n_k$ , donde  $0 < n_k < n$ ).
- Resolver (vencer) sucesiva e independientemente las subinstancias, bien directamente si son elementales (casos «base») o bien de forma recursiva.
- Combinar las subsoluciones obtenidas para construir la solución de la instancia original.

## La técnica «divide y vencerás»



- Si las subinstancias son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subinstancias lo suficientemente pequeñas para ser resueltas directamente.

# La técnica «divide y vencerás»



- Si las subinstancias son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar subinstancias lo suficientemente pequeñas para ser resueltas directamente.
- Esto sugiere el uso de la recursividad en las implementaciones de estos algoritmos (soluciones eficientes).
- Las ecuaciones recurrentes son naturales en esta técnica.



# La técnica «divide y vencerás»

## Esquema general

```
function DyV( $P, n$ )  
  if  $P$  es suficientemente pequeño then  
     $Solucion \leftarrow \text{adhoc}(P, n)$   
  else  
    Dividir  $P$  en  $k$  subinstancias más pequeñas  $S_1, S_2, \dots, S_k$  con  
    tamaños  $n_1, n_2, \dots, n_k$ .  
    for all  $i = 1, \dots, k$  do  
       $Subsolucion_i \leftarrow \text{DyV}(S_i, n_i)$   
    end for  
     $Solucion \leftarrow \text{Combinar}(Subsolucion_1, \dots, Subsolucion_k)$   
  end if  
  return  $Solucion$   
end function
```

## Algunas consideraciones

- El número de subinstancias,  $k$ , suele ser pequeño e independiente de la instancia concreta que se vaya a resolver.
- Si  $k = 1$ , hablamos de reducción o simplificación.
- $adhoc(P, n)$  es un algoritmo básico capaz de resolver el problema:
  - Debe ser eficiente en instancias pequeñas.
  - Su rendimiento en instancias grandes nos da igual.

## Ejemplo: elevar un número real a un entero

- Sean  $r$  y  $n$  un número real y un número entero, respectivamente.
- Se requiere un algoritmo que realice la operación  $r^n$ .
- Existe un método básico que resuelve el problema

# La técnica «divide y vencerás»

## Ejemplo: elevar un número real a un entero

- Sean  $r$  y  $n$  un número real y un número entero, respectivamente.
- Se requiere un algoritmo que realice la operación  $r^n$ .
- Existe un método básico que resuelve el problema (basta con multiplicar  $r$  un total de  $n$  veces:

$$r^n = r \cdot r \cdots r \quad (n \text{ veces})$$

# La técnica «divide y vencerás»

## Ejemplo: elevar un número real a un entero

- Sean  $r$  y  $n$  un número real y un número entero, respectivamente.
- Se requiere un algoritmo que realice la operación  $r^n$ .
- Existe un método básico que resuelve el problema (basta con multiplicar  $r$  un total de  $n$  veces:

$$r^n = r \cdot r \cdots r \quad (n \text{ veces})$$

## Algoritmo

```
1 double Elevar_basico(double r, int n) {  
2     double resultado = 1.0;  
3     for (int i = 0; i < n; i++) {  
4         resultado *= r;  
5     }  
6     return resultado;  
7 }
```

## Ejemplo: elevar un número real a un entero

- ¿Se puede dividir el problema?

## Ejemplo: elevar un número real a un entero

- ¿Se puede dividir el problema?
  - Sí:  $r^n = r \cdot r^{n-1}$

## Ejemplo: elevar un número real a un entero

- ¿Se puede dividir el problema?
  - Sí:  $r^n = r \cdot r^{n-1}$
  - Esta división no es muy interesante.
- ¿Alguna otra división?



## Ejemplo: elevar un número real a un entero

- ¿Se puede dividir el problema?
  - Sí:  $r^n = r \cdot r^{n-1}$
  - Esta división no es muy interesante.
- ¿Alguna otra división?
  - $r^n = r^{n/2} \cdot r^{n/2}$
  - Esta división es más eficiente.

## Ejemplo: elevar un número real a un entero

- ¿Se puede dividir el problema?
  - Sí:  $r^n = r \cdot r^{n-1}$
  - Esta división no es muy interesante.
- ¿Alguna otra división?
  - $r^n = r^{n/2} \cdot r^{n/2}$
  - Esta división es más eficiente.
- Expresión:

$$r^n = \begin{cases} r^{n/2} \cdot r^{n/2} & n \text{ par} \\ r \cdot r^{n/2} \cdot r^{n/2} & n \text{ impar} \end{cases}$$

## Ejemplo: elevar un número real a un entero

- Expresión:

$$r^n = \begin{cases} r^{n/2} \cdot r^{n/2} & n \text{ par} \\ r \cdot r^{n/2} \cdot r^{n/2} & n \text{ impar} \end{cases}$$

- «Divide y vencerás»: la instancia del problema se tiene que poder dividir en una o más subinstancias equivalentes de menor tamaño, independientes entre sí, que se puedan resolver por separado:

## Ejemplo: elevar un número real a un entero

- Expresión:

$$r^n = \begin{cases} r^{n/2} \cdot r^{n/2} & n \text{ par} \\ r \cdot r^{n/2} \cdot r^{n/2} & n \text{ impar} \end{cases}$$

- «Divide y vencerás»: la instancia del problema se tiene que poder dividir en una o más subinstancias equivalentes de menor tamaño, independientes entre sí, que se puedan resolver por separado:
  - Se puede dividir en dos subinstancias de menor tamaño ( $n/2$ ), independientes y que se pueden resolver por separado (sacando partido de las propiedades de las potencias).

## Ejemplo: elevar un número real a un entero

- Expresión:

$$r^n = \begin{cases} r^{n/2} \cdot r^{n/2} & n \text{ par} \\ r \cdot r^{n/2} \cdot r^{n/2} & n \text{ impar} \end{cases}$$

- «Divide y vencerás»: las soluciones de las subinstancias de menor tamaño se deben poder combinar entre sí para obtener la solución de la instancia inicial:

## Ejemplo: elevar un número real a un entero

- Expresión:

$$r^n = \begin{cases} r^{n/2} \cdot r^{n/2} & n \text{ par} \\ r \cdot r^{n/2} \cdot r^{n/2} & n \text{ impar} \end{cases}$$

- «Divide y vencerás»: las soluciones de las subinstancias de menor tamaño se deben poder combinar entre sí para obtener la solución de la instancia inicial:
  - La combinación es la multiplicación de la subsolución de calcular  $r^{n/2}$  por la subsolución de calcular  $r^{n/2}$ .

## Ejemplo: elevar un número real a un entero

- Expresión:

$$r^n = \begin{cases} r^{n/2} \cdot r^{n/2} & n \text{ par} \\ r \cdot r^{n/2} \cdot r^{n/2} & n \text{ impar} \end{cases}$$

- «Divide y vencerás»: debe existir un método básico que resuelva el problema o un caso base indivisible donde el problema esté resuelto:

## Ejemplo: elevar un número real a un entero

- Expresión:

$$r^n = \begin{cases} r^{n/2} \cdot r^{n/2} & n \text{ par} \\ r \cdot r^{n/2} \cdot r^{n/2} & n \text{ impar} \end{cases}$$

- «Divide y vencerás»: debe existir un método básico que resuelva el problema o un caso base indivisible donde el problema esté resuelto:
  - Existe el método básico mostrado anteriormente y los casos base  $r^0 = 1$  y  $r^1 = r$ :

$$r^n = \begin{cases} 1 & n = 0 \\ r & n = 1 \\ r^{n/2} \cdot r^{n/2} & n > 1 \text{ y } n \text{ par} \\ r \cdot r^{n/2} \cdot r^{n/2} & n > 1 \text{ y } n \text{ impar} \end{cases}$$



## Ejemplo: elevar un número real a un entero

- Diseño del algoritmo:
  - Caso base: Si  $n \leq 1 \Rightarrow r^n$ .
  - División: la instancia original del problema  $r^n$  se divide en subinstancias de tamaño  $n/2$  (en realidad, en una).
  - Combinación: sea  $S_1$  la subsolución de la subinstancia, la solución  $S$  de la instancia original se calcula como:

$$S = \begin{cases} S_1 \cdot S_1 & n \text{ es par} \\ r \cdot S_1 \cdot S_1 & n \text{ es impar} \end{cases}$$

# La técnica «divide y vencerás»

## Ejemplo: elevar un número real a un entero

```
function DyV( $r, n$ )  
  if  $n = 0$  then  
     $Solucion \leftarrow 1$   
  else if  $n = 1$  then  
     $Solucion \leftarrow r$   
  else  
     $parcial \leftarrow DyV(r, \text{floor}(n/2))$   
    if  $n$  es par then  
       $Solucion \leftarrow parcial \cdot parcial$   
    else  
       $Solucion \leftarrow r \cdot parcial \cdot parcial$   
    end if  
  end if  
  return  $Solucion$   
end function
```

# La técnica «divide y vencerás»

## Ejemplo: elevar un número real a un entero

```
1 double Elevar_DyV(double r, int n) {
2     if (n == 0)
3         return 1;
4     else if (n == 1)
5         return r;
6     else {
7         double subsolucion = Elevar_DyV(r, n/2);
8         if (n % 2 == 0)
9             return subsolucion * subsolucion;
10        else
11            return r * subsolucion * subsolucion;
12    }
13 }
```

# La técnica «divide y vencerás»

## Ejemplo: elevar un número real a un entero

```
1 double Elevar_DyV(double r, int n) {
2     if (n == 0)
3         return 1;
4     else if (n == 1)
5         return r;
6     else {
7         double subsolucion = Elevar_DyV(r, n/2);
8         if (n % 2 == 0)
9             return subsolucion * subsolucion;
10        else
11            return r * subsolucion * subsolucion;
12    }
13 }
```

Ecuación de recurrencias  $T(n) = T(n/2) + 1$

Eficiencia del algoritmo  $O(\log_2(n))$

# El problema del umbral

---

## Cuestiones clave de la técnica «divide y vencerás»

- ¿Cómo descomponer la instancia del problema en subinstancias?
- ¿Cómo resolver las subinstancias?
- ¿Cómo combinar las soluciones?
- ¿Merece la pena hacer esto?

# El problema del umbral

- Supongamos una instancia de un problema  $P$ , de tamaño  $n$ , que sabemos puede resolverse con un algoritmo (básico)  $A$ :

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

- Dividimos  $P$  en 3 subinstancias de tamaños  $n/2$ , siendo cada una de ellas del mismo tipo que  $P$ , y consumiendo un tiempo lineal la combinación de sus soluciones:  $T_{\text{com}}(n) \leq d \cdot n$ .
- Tenemos un nuevo algoritmo  $B$ , «divide y vencerás», que consumirá un tiempo:

# El problema del umbral

- Supongamos una instancia de un problema  $P$ , de tamaño  $n$ , que sabemos puede resolverse con un algoritmo (básico)  $A$ :

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

- Dividimos  $P$  en 3 subinstancias de tamaños  $n/2$ , siendo cada una de ellas del mismo tipo que  $P$ , y consumiendo un tiempo lineal la combinación de sus soluciones:  $T_{\text{com}}(n) \leq d \cdot n$ .
- Tenemos un nuevo algoritmo  $B$ , «divide y vencerás», que consumirá un tiempo:

$$T_B(n) = 3T_A(n/2) + T_{\text{com}}(n) \leq 3T_A(n/2) + d \cdot n \leq \frac{3}{4} \cdot c \cdot n^2 + d \cdot n$$

- $B$  tiene un tiempo de ejecución mejor que el algoritmo  $A$ , ya que disminuye la constante oculta.



# El problema del umbral

- Pero si cada subproblema se resuelve de nuevo con «divide y vencerás», podemos hacer un tercer algoritmo  $C$ , recursivo, que tendría un tiempo:

$$T_C(n) = \begin{cases} T_A(n) & n \leq n_0 \\ 3T_C(n/2) + T_{\text{com}}(n) & n > n_0 \end{cases}$$

- $C$  es mejor en eficiencia que los algoritmos  $A$  y  $B$ :

$$T_C(n) \leq b \cdot n^{1.58}$$

- Al valor  $n_0$  se le denomina umbral y es fundamental para que funcione bien la técnica.

## Tamaño del caso muy pequeño

- Muchas llamadas recursivas.
- Muy costoso de tantas llamadas.

## Alternativa

- ¿Se posee un método menos costoso?
- Ejecutarlo cuando  $P$  es *suficientemente pequeño*.

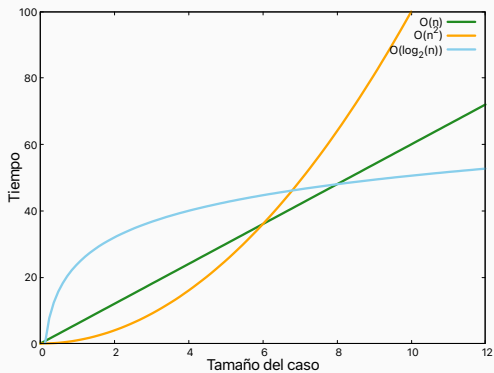
## Tamaño del caso muy pequeño

- Muchas llamadas recursivas.
- Muy costoso de tantas llamadas.

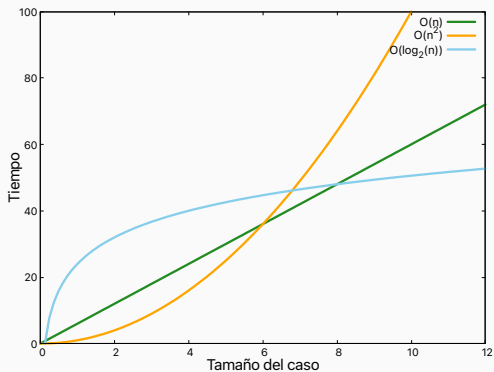
## Alternativa

- ¿Se posee un método menos costoso?
- Ejecutarlo cuando  $P$  es *suficientemente pequeño*.
- ¿Cuándo es *suficientemente pequeño*? Comparando los tiempos de ejecución del algoritmo «divide y vencerás» diseñado y el algoritmo básico.

# El problema del umbral



# El problema del umbral



$$\text{Mejor} = \begin{cases} O(n^2) & n \leq 6 \\ O(n) & 6 < n \leq 8 \\ O(\log_2(n)) & n > 8 \end{cases}$$

## Mergesort y algoritmo base inserción

- El algoritmo base para el problema de ordenación será el algoritmo de ordenación por inserción ( $O(n^2)$ ).
- Para determinar cuando el tamaño es suficientemente pequeño como para dejar de aplicar «divide y vencerás» calculamos empíricamente el tiempo de ejecución de los dos algoritmos:
  - Se mide el tiempo de ejecución de ambos algoritmos para distintos tamaños del problema.
  - Se calcula la constante oculta:  $T_{\text{inserción}}(n) \leq c_1 \cdot n^2$
  - Se calcula la constante oculta:  $T_{\text{mergesort}}(n) \leq c_2 \cdot n \cdot \log_2(n)$
  - Se calcula  $n$  de la igualdad:  $c_1 \cdot n^2 = c_2 \cdot n \cdot \log_2(n)$

# La técnica «divide y vencerás»

## Inserción

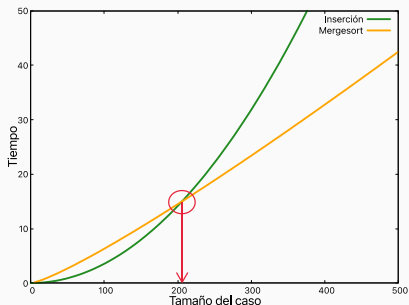
$n$	$T(n)$
10000	45425
20000	140155
30000	303081
40000	542304
50000	856232
60000	1228797
70000	1673289
80000	2198850
90000	2787441
100000	3449736

$$c_1 = 3.54 \cdot 10^{-4}$$

## Mergesort

$n$	$T(n)$
10000	2267
20000	2599
30000	3992
40000	5461
50000	6968
60000	8413
70000	9932
80000	11629
90000	11835
100000	12038

$$c_2 = 9.48 \cdot 10^{-3}$$



¿En qué punto comienza a ser mergesort mejor que inserción?

$$c_1 \cdot n^2 = c_2 \cdot n \cdot \log_2(n)$$

$n$  se puede calcular gráficamente o de forma aproximada si no se dispone de métodos matemáticos para ello

## Mergesort y algoritmo base inserción

- Implementación del algoritmo MergeSort tras resolver el problema del umbral:

```
1 void MergeSort(int *v, int ini, int fin) {  
2     if (fin - ini <= 208) {  
3         Insercion(v, ini, fin);  
4     } else {  
5         int med = (ini + fin) / 2;  
6         MergeSort(v, ini, med);  
7         MergeSort(v, med + 1, fin);  
8         Combina(v, ini, med, fin);  
9     }  
10 }
```



## Condiciones para que «divide y vencerás» sea ventajoso

- Selección cuidadosa de cuándo usar el algoritmo *ad hoc* (calcular el umbral de recursividad).
- Poder descomponer la instancia del problema en subinstancias y combinar de forma eficiente a partir de las soluciones parciales.
- El número  $k$  de subinstancias debe ser razonablemente pequeño.
- Las subinstancias deben tener aproximadamente el mismo tamaño.
- Las subinstancias deben ser del menor tamaño posible.

# Análisis de algoritmos «divide y vencerás»

---

# Análisis de algoritmos «divide y vencerás»

- Cuando un algoritmo contiene una llamada recursiva a sí mismo, generalmente su tiempo de ejecución puede describirse por una recurrencia que da el tiempo de ejecución para un caso de tamaño  $n$  en función de entradas de menor tamaño.
- En «divide y vencerás» tenemos recurrencias del tipo:

$$T(n) = \begin{cases} d(n) & n \leq n_0 \\ k \cdot T(n/b) + g(n) & n > n_0 \end{cases}$$

- $k$  representa el número de subinstancias,  $n/b$  el tamaño de estas,  $g(n)$  el coste de descomponer la instancia del problema inicial en las  $k$  subinstancias y el de combinar las soluciones para obtener la solución de la instancia original, y  $d(n)$  el coste de resolver un problema elemental.

## Fórmula maestra

$$T(n) = k \cdot T(n/b) + g(n)$$

- Si  $g(n) \in \Theta(n^a)$ ,  $a \in \mathbb{R}^{\geq 0}$ , el orden de complejidad de la solución a esta ecuación es:
  - $\Theta(n^a)$  si  $k < b^a$
  - $\Theta(n^a \cdot \log(n))$  si  $k = b^a$
  - $\Theta(n^{\log_b(k)})$  si  $k > b^a$
- El orden de eficiencia depende de la relación entre el número de subinstancias ( $k$ ), el tamaño de estas ( $b$ ) y la dificultad de dividir y combinar ( $a$ ).

## La importancia de las condiciones

- El número de subinstancias tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow$$

$$T(n) = 4T(n/2) + c \longrightarrow$$

$$T(n) = 8T(n/2) + c \longrightarrow$$

## La importancia de las condiciones

- El número de subinstancias tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 4T(n/2) + c \longrightarrow$$

$$T(n) = 8T(n/2) + c \longrightarrow$$

## La importancia de las condiciones

- El número de subinstancias tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 4T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(4)}) = O(n^2)$$

$$T(n) = 8T(n/2) + c \longrightarrow$$

## La importancia de las condiciones

- El número de subinstancias tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 4T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(4)}) = O(n^2)$$

$$T(n) = 8T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(8)}) = O(n^3)$$



## La importancia de las condiciones

- El número de subinstancias tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 4T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(4)}) = O(n^2)$$

$$T(n) = 8T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(8)}) = O(n^3)$$

- La eficiencia de la combinación de subsoluciones tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow$$

$$T(n) = 2T(n/2) + n \longrightarrow$$

$$T(n) = 2T(n/2) + n^2 \longrightarrow$$

## La importancia de las condiciones

- El número de subinstancias tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 4T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(4)}) = O(n^2)$$

$$T(n) = 8T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(8)}) = O(n^3)$$

- La eficiencia de la combinación de subsoluciones tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 2T(n/2) + n \longrightarrow$$

$$T(n) = 2T(n/2) + n^2 \longrightarrow$$

## La importancia de las condiciones

- El número de subinstancias tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 4T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(4)}) = O(n^2)$$

$$T(n) = 8T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(8)}) = O(n^3)$$

- La eficiencia de la combinación de subsoluciones tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 2T(n/2) + n \longrightarrow T(n) \in O(n \cdot \log_2(n))$$

$$T(n) = 2T(n/2) + n^2 \longrightarrow$$

## La importancia de las condiciones

- El número de subinstancias tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 4T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(4)}) = O(n^2)$$

$$T(n) = 8T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(8)}) = O(n^3)$$

- La eficiencia de la combinación de subsoluciones tiene importancia:

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 2T(n/2) + n \longrightarrow T(n) \in O(n \cdot \log_2(n))$$

$$T(n) = 2T(n/2) + n^2 \longrightarrow T(n) \in O(n^2)$$

## La importancia de las condiciones

- Que las subinstancias sean aproximadamente del mismo tamaño tiene importancia:

$$T(n) = 2T(n/2) + n \longrightarrow$$

$$T(n) = T(1) + T(n-1) + n \longrightarrow$$

## La importancia de las condiciones

- Que las subinstancias sean aproximadamente del mismo tamaño tiene importancia:

$$T(n) = 2T(n/2) + n \longrightarrow T(n) \in O(n \cdot \log n)$$

$$T(n) = T(1) + T(n-1) + n \longrightarrow$$

## La importancia de las condiciones

- Que las subinstancias sean aproximadamente del mismo tamaño tiene importancia:

$$T(n) = 2T(n/2) + n \longrightarrow T(n) \in O(n \cdot \log n)$$

$$T(n) = T(1) + T(n-1) + n \longrightarrow T(n) \in O(n^2)$$

## La importancia de las condiciones

- Que las subinstancias sean aproximadamente del mismo tamaño tiene importancia:

$$T(n) = 2T(n/2) + n \longrightarrow T(n) \in O(n \cdot \log n)$$

$$T(n) = T(1) + T(n-1) + n \longrightarrow T(n) \in O(n^2)$$

- El tamaño de las subinstancias tiene importancia:

$$T(n) = 2T(n/4) + c \longrightarrow$$

$$T(n) = 2T(n/2) + c \longrightarrow$$

$$T(n) = 2T(n-1) + c \longrightarrow$$



## La importancia de las condiciones

- Que las subinstancias sean aproximadamente del mismo tamaño tiene importancia:

$$T(n) = 2T(n/2) + n \longrightarrow T(n) \in O(n \cdot \log n)$$

$$T(n) = T(1) + T(n-1) + n \longrightarrow T(n) \in O(n^2)$$

- El tamaño de las subinstancias tiene importancia:

$$T(n) = 2T(n/4) + c \longrightarrow T(n) \in O(n^{\log_4(2)}) = O(n^{0.5}) = O(\sqrt{n})$$

$$T(n) = 2T(n/2) + c \longrightarrow$$

$$T(n) = 2T(n-1) + c \longrightarrow$$

## La importancia de las condiciones

- Que las subinstancias sean aproximadamente del mismo tamaño tiene importancia:

$$T(n) = 2T(n/2) + n \longrightarrow T(n) \in O(n \cdot \log n)$$

$$T(n) = T(1) + T(n-1) + n \longrightarrow T(n) \in O(n^2)$$

- El tamaño de las subinstancias tiene importancia:

$$T(n) = 2T(n/4) + c \longrightarrow T(n) \in O(n^{\log_4(2)}) = O(n^{0.5}) = O(\sqrt{n})$$

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 2T(n-1) + c \longrightarrow$$

## La importancia de las condiciones

- Que las subinstancias sean aproximadamente del mismo tamaño tiene importancia:

$$T(n) = 2T(n/2) + n \longrightarrow T(n) \in O(n \cdot \log n)$$

$$T(n) = T(1) + T(n-1) + n \longrightarrow T(n) \in O(n^2)$$

- El tamaño de las subinstancias tiene importancia:

$$T(n) = 2T(n/4) + c \longrightarrow T(n) \in O(n^{\log_4(2)}) = O(n^{0.5}) = O(\sqrt{n})$$

$$T(n) = 2T(n/2) + c \longrightarrow T(n) \in O(n^{\log_2(2)}) = O(n)$$

$$T(n) = 2T(n-1) + c \longrightarrow T(n) \in O(2^n)$$

# Búsqueda binaria

---

## Enunciado

- Sea  $V[1 \dots n]$  un vector ordenado en orden no decreciente,  $V[i] \leq V[j]$  para  $1 \leq i \leq j \leq n$ , y sea  $x$  un elemento a buscar.
- Formalmente, se quiere encontrar el índice  $i$  tal que  $1 \leq i \leq n + 1$  y  $V[i - 1] < x \leq V[i]$ .

1	2	3	4	5
3	7	25	41	53

Si  $x = 25$ , entonces  $i = 3$

Si  $x = 15$ , entonces  $i = 3$

Si  $x = 67$ , entonces  $i = 6$

Si  $x = 2$ , entonces  $i = 1$ .

## Búsqueda secuencial

- La forma simple de resolver el problema es buscar hasta que llegar al final o encontrar un elemento que no sea menor que  $x$ .

```
1 int Busqueda ( int *v, int n, int x) {  
2     int i = 0;  
3     while (i < n && x > v[i]){  
4         i++;  
5     }  
6     return i;  
7 }
```

- El orden de eficiencia es  $O(n)$ .

## Fundamento

- Para acelerar la búsqueda, podemos buscar  $x$  bien en la primera mitad del vector o bien en la segunda.
- Para averiguar cuál de esas búsquedas es la correcta comparamos  $x$  con un elemento del vector,  $k = n/2$ .
- Si  $x \leq V[k]$  podemos restringir la búsqueda a  $V[1 \dots k]$ . En otro caso, buscamos en  $V[k + 1 \dots n]$ .
- Realmente es un caso de reducción o simplificación: la solución de toda instancia se reduce a una única instancia más pequeña (concretamente de tamaño mitad).

# Búsqueda binaria

## Algoritmo

```
1 int Binaria_recursiva (int *v, int ini, int fin, int x){
2     int centro = (ini + fin) / 2;
3     if (ini == fin) return ini;
4     if (x <= v[centro])
5         return Binaria_recursiva (v, ini , centro, x);
6     return Binaria_recursiva (v, centro+1, fin, x);
7 }
8 int Busqueda_binaria (int *v, int n, int x) {
9     if (x > v[n-1]) return n;
10    else {
11        return Binaria_recursiva(v, 0, n-1, x);
12    }
13 }
```

## Eficiencia



# Búsqueda binaria

## Algoritmo

```
1 int Binaria_recursiva (int *v, int ini, int fin, int x){
2     int centro = (ini + fin) / 2;
3     if (ini == fin) return ini;
4     if (x <= v[centro])
5         return Binaria_recursiva (v, ini , centro, x);
6     return Binaria_recursiva (v, centro+1, fin, x);
7 }
8 int Busqueda_binaria (int *v, int n, int x) {
9     if (x > v[n-1]) return n;
10    else {
11        return Binaria_recursiva(v, 0, n-1, x);
12    }
13 }
```

## Eficiencia

- $T(n) = T(n/2) + c$
- Como  $k = 1, b^a = 2^0$ , entonces  $T(n) = O(\log(n))$ .

# Ordenación rápida

---

## Quicksort

- Sea  $V[1 \dots n]$  un vector de  $n$  elementos. La idea es ordenarlo en orden ascendente, es decir,  $V[i] \leq V[j]$  para  $1 \leq i \leq j \leq n$ .
- Propuesto por C.A.R. Hoare en 1962.
- Es el algoritmo «divide y vencerás» de ordenación general más eficiente (en caso promedio).

## Idea básica

- Determinar un elemento, denominado pivote, que divida al vector en dos partes: una que contenga los elementos menores que él y otra que contiene los mayores o iguales.
- Ordenar los dos subvectores obtenidos.
- Combinar los dos subvectores ordenados para obtener el vector inicial ordenado.

## División

- Se divide el vector  $V[1 \dots n]$  en dos subvectores (mejor si son tamaño parecido), independientes, y que se pueden ordenar por separado.
- Para dividir el vector, seleccionaremos un elemento, denominado pivote, de modo que los elementos menores que este queden a la izquierda del vector, y los mayores o iguales a la derecha.

## Casos

## División

- Se divide el vector  $V[1 \dots n]$  en dos subvectores (mejor si son tamaño parecido), independientes, y que se pueden ordenar por separado.
- Para dividir el vector, seleccionaremos un elemento, denominado pivote, de modo que los elementos menores que este queden a la izquierda del vector, y los mayores o iguales a la derecha.

## Casos

- **Mejor caso:** el pivote divide al vector en dos mitades iguales.
- **Peor caso:** el pivote deja un subvector con 1 elemento y el otro con  $n - 1$  elementos.

## Resolución

- Al llegar al caso base de 1 elemento, el vector estará ordenado.
- La penúltima llamada recursiva (vector de 2 elementos) también hará que el vector resultante esté ordenado, ya que se ha pivotado y el elemento mayor estará en la segunda posición, mientras que el elemento menor estará en la primera.
- En las llamadas recursivas anteriores, pasará lo mismo debido a que se va pivotando cada vez que se ejecuta la función.

## Combinación

- No se requiere combinación adicional ya que al usar el pivote y gracias a las llamadas recursivas no es necesario.

## Algoritmo

```
1 void quicksort(int *v, int ini, int fin) {  
2     if (ini < fin) {  
3         int pos_pivote = pivotar(v, ini, fin);  
4         quicksort(v, ini, pos_pivote - 1);  
5         quicksort(v, pos_pivote + 1, fin);  
6     }  
7 }
```

## Elección del pivote

- Cualquiera puede diseñar su algoritmo quicksort: la elección del pivote condiciona el tiempo de ejecución.
- El pivote puede ser cualquier elemento en el dominio (no es necesario que esté en el vector):
  - Podría ser la media de los elementos seleccionados del vector.
  - Podría elegirse aleatoriamente.
- Pivotes usuales:
  - La mediana de un mínimo de tres elementos del vector.
  - El elemento medio del vector.



## Elección del pivote

- El empleo de la mediana de tres elementos no tiene justificación teórica.
- Si queremos usar el concepto de mediana, deberíamos escoger como pivote la mediana del vector porque lo divide en dos subvectores de igual tamaño:
  - Mediana:  $(n/2)^{\text{o}}$  mayor elemento.
  - Elegir tres elementos al azar y coger su mediana. Esto suele reducir el tiempo de ejecución en un 5%.
- Una elección más rápida es escoger el mayor de los dos primeros elementos del vector.

# Ordenación rápida

5	89	35	14	24	15	37	13	20	7	70
---	----	----	----	----	----	----	----	----	---	----

5	89	35	14	24	15	37	13	20	7	70
---	----	----	----	----	----	----	----	----	---	----

7	14	5	13	15	35	37	89	20	24	70
---	----	---	----	----	----	----	----	----	----	----

¿Cómo conseguir realizar eficientemente la partición, es decir colocar todos los menores que el pivote a su izquierda y todos los mayores o iguales a su derecha?

## Partición

- Es fácil crear un algoritmo de partición con tiempo lineal.
- Es importante que la constante oculta sea lo más pequeña posible para que quicksort sea competitivo.
- Se puede recorrer el vector una sola vez, pero empezando por los dos extremos.

## Partición

- Sea  $V[k \dots l]$  el vector.
- Sea  $p = V[k]$  el pivote (el primer elemento).
- Los punteros  $i$  y  $j$  se inician en  $k$  y  $l$ , respectivamente.
- El puntero  $i$  se incrementa hasta que  $V[i] \geq p$  y el puntero  $j$  se disminuye hasta que  $V[j] < p$ .
- Entonces, se intercambian  $V[i]$  y  $V[j]$ . Este proceso se repite mientras  $i < j$ .
- Finalmente,  $V[k]$  y  $V[j]$  se intercambian para poner el pivote en su posición correcta.

# Ordenación rápida

## Algoritmo

```
1 int pivotar(int *v, int ini, int fin) {
2     int pivote = v[ini], i = ini + 1, j = fin;
3
4     while (i <= j) {
5         while (i <= j && v[i] < pivote) {
6             i++;
7         }
8         while (i <= j && v[j] >= pivote) {
9             j--;
10        }
11        if (i < j) {
12            intercambiar(v[i], v[j]);
13        }
14    }
15    if (ini < j) {
16        intercambiar(v[ini], v[j]);
17    }
18    return j;
19 }
```

## Eficiencia en el peor caso

- El pivote es el primer elemento del vector.
- $T(n) = T(1) + T(n - 1) + n$

## Eficiencia en el peor caso

- El pivote es el primer elemento del vector.
- $T(n) = T(1) + T(n - 1) + n$
- $O(n^2)$ .
- En el peor caso es tan malo como el peor caso del método de inserción (selección).

## Eficiencia en el peor caso

- El pivote es el primer elemento del vector.
- $T(n) = T(1) + T(n - 1) + n$
- $O(n^2)$ .
- En el peor caso es tan malo como el peor caso del método de inserción (selección).

## Eficiencia en el mejor caso

- El pivote es la mediana.
- $T(n) = 2T(n/2) + n$



## Eficiencia en el peor caso

- El pivote es el primer elemento del vector.
- $T(n) = T(1) + T(n-1) + n$
- $O(n^2)$ .
- En el peor caso es tan malo como el peor caso del método de inserción (selección).

## Eficiencia en el mejor caso

- El pivote es la mediana.
- $T(n) = 2T(n/2) + n$
- $\Omega(n \cdot \log_2(n))$ .

## Eficiencia en el caso promedio

- La probabilidad del peor caso es muy baja.
- Se ha demostrado que es  $O(n \cdot \log_2(n))$ .
- En promedio se comporta mejor que los demás algoritmos de ordenación.

# Problema de selección

---

## Enunciado

- Sea  $V[1 \dots n]$  un vector de  $n$  elementos. Se desea conocer qué elemento se situaría en la  $i$ -ésima posición del vector en caso de que este estuviese ordenado.

## Enunciado

- Sea  $V[1 \dots n]$  un vector de  $n$  elementos. Se desea conocer qué elemento se situaría en la  $i$ -ésima posición del vector en caso de que este estuviese ordenado.

## Ejemplo de uso

- Cálculo de la mediana.

## Algoritmo básico

## Enunciado

- Sea  $V[1 \dots n]$  un vector de  $n$  elementos. Se desea conocer qué elemento se situaría en la  $i$ -ésima posición del vector en caso de que este estuviese ordenado.

## Ejemplo de uso

- Cálculo de la mediana.

## Algoritmo básico

- Ordenar el vector y devolver el elemento que hubiese en la  $i$ -ésima posición.
- Complejidad:  $O(n \cdot \log(n))$ . La ordenación tiene esa eficiencia y es la operación más costosa que se realiza.

## Idea general

- No es necesario que el vector esté totalmente ordenado.
- Lo único necesario es que el elemento  $i$ -ésimo esté ordenado.

**¿Podemos usar algo ya conocido?**

## Idea general

- No es necesario que el vector esté totalmente ordenado.
- Lo único necesario es que el elemento  $i$ -ésimo esté ordenado.

## ¿Podemos usar algo ya conocido?

- La función pivotar de quicksort.
- Al terminar su ejecución, en la  $j$ -ésima posición del vector se encuentra el pivote y todos los elementos a su izquierda son menores que él, por lo que se encuentra ordenado en su posición.



## División

- Para encontrar el elemento en la  $i$ -ésima posición del vector, se colocan los elementos menores que un pivote a la izquierda del vector, y los mayores o iguales en la parte derecha. La posición del pivote,  $posPivote$ , estará por tanto ordenada.
- Si la posición  $i$  que buscamos es la posición del pivote,  $posPivote$ , hemos terminado.
- Si la posición  $i$  que buscamos es inferior a  $posPivote$ , dividimos la instancia del problema en 1 subinstancia (vector  $V$  desde la primera posición hasta  $posPivote$ ).
- Si la posición  $i$  que buscamos es superior a  $posPivote$ , dividimos la instancia del problema en 1 subinstancia (vector  $V$  desde  $posPivote + 1$  hasta el último elemento).

## Resolución

- Se vuelve a calcular el pivote para el subvector generado, aplicando recursivamente este proceso hasta que se encuentre la  $i$ -ésima posición.

## Combinación

- Como la función de pivotar deja en *posPivot* el elemento que estamos buscando, y solo dividimos la instancia del problema en 1 subinstancia, no es necesario realizar combinación ni operaciones adicionales.

# Problema de selección

## Pseudocódigo

```
function Seleccion(v, ini, fin, i)  
  if ini = fin then  
    return v[ini]  
  else  
    posPivote, v  $\leftarrow$  pivotar(v, ini, fin)  
    if posPivote = i then  
      return v[posPivote]  
    else if posPivote < i then  
      return Seleccion(v, ini, posPivote - 1, i)  
    else if posPivote > i then  
      return Seleccion(v, posPivote + 1, fin, i)  
    end if  
  end if  
end function
```

# Problema de selección

## Algoritmo

```
1 int seleccion(int *v, int ini, int fin, int posicion) {
2     if (ini == fin) {
3         return v[ini];
4     }
5     else {
6         int p = pivotar(v, ini, fin);
7
8         if (p == posicion) {
9             return v[p];
10        }
11        else if (posicion < p) {
12            return seleccion(v, inicio, p-1, posicion);
13        }
14        else {
15            return seleccion(v, p+1, fin, posicion);
16        }
17    }
18 }
```

## Eficiencia en el peor caso

- Divide al vector en dos partes de tamaño 1 y  $n - 1$ , respectivamente.
- $T(n) = T(n - 1) + n$
- $O(n^2)$ .

## Eficiencia en el mejor caso

- Divide al vector en dos partes de igual tamaño ( $n/2$ ).
- $T(n) = T(n/2) + n$
- $\Omega(n)$ .

## Eficiencia en el caso promedio

- Es muy improbable que se dé el peor caso.
- En promedio, se comporta como  $\Omega(n)$ .

## Multiplicación de enteros grandes

---

# Bibliografía

---

## Aclaración

- El contenido de las diapositivas es esquemático y representa un apoyo para las clases teóricas.
- Se recomienda completar los contenidos del tema 1 con apuntes propios tomados en clase y con la bibliografía principal de la asignatura.

## Por ejemplo



G. Brassard and P. Bratley.

***Fundamentals of Algorithmics.***

Prentice Hall, Englewood Cliffs, New Jersey, 1996.



J. L. Verdegay.

***Lecciones de Algorítmica.***

Editorial Técnica AVICAM, 2017.