

# Tema : Algoritmos Divide y Vencerás

## Algorítmica

Luis M. de Campos  
Departamento de Ciencias de la Computación e I. A.  
Grado en Informática  
E.T.S.I. Informática y de Telecomunicaciones  
Universidad de Granada  
Email: lci@decsai.ugr.es



- La técnica Divide y Vencerás (DV) consiste en:
  - Descomponer (dividir) el caso a resolver de un problema en un cierto número de subcasos más pequeños del mismo problema.
  - Resolver (vencer) sucesiva e independientemente todos estos subcasos.
  - Combinar las soluciones obtenidas para obtener la solución del caso original.
- Este enfoque, sobre todo cuando se utiliza recursivamente, a menudo proporciona soluciones eficientes de los problemas.
- Las ecuaciones recurrentes serán naturales en este método.

- ¿Cómo descomponer el problema en subproblemas?
- ¿Cómo resolver los subproblemas?
- ¿Cómo combinar las soluciones?
- ¿Merece la pena hacer esto?

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

$$t_A(n) \leq cn^2$$

- Dividimos P en 3 subproblemas de tamaños  $n/2$ , siendo cada uno de ellos del mismo tipo que A, y consumiendo un tiempo lineal la combinación de sus soluciones:  $t(n) \leq dn$ .
- Tenemos un nuevo algoritmo B, Divide y Vencerás, que consumirá un tiempo:

$$t_B(n) = 3t_A(n/2) + t(n) \leq 3t_A(n/2) + dn \leq (3c/4)n^2 + dn$$

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

- Pero si cada subproblema se resuelve de nuevo con Divide y Vencerás, podemos hacer un tercer algoritmo C, recursivo, que tendría un tiempo:

$$t_C(n) = \begin{cases} t_A(n) & \text{si } n \leq n_0 \\ 3t_C(n/2) + t(n) & \text{si } n > n_0 \end{cases}$$

- C es mejor en eficiencia que los algoritmos A y B ( $t_C(n) \leq bn^{1,58}$ )
- Al valor  $n_0$  se le denomina umbral y es fundamental para que funcione bien la técnica.

# Algoritmo Divide y Vencerás

Función  $DV(x)$

si  $x$  es suficientemente pequeño entonces

devolver  $ad\ hoc(x)$

descomponer  $x$  en casos más pequeños  $x_1, \dots, x_l$

para  $i=1$  hasta  $l$

$y_i = DV(x_i)$

combinar los  $y_i$  para obtener una solución  $y$  de  $x$

devolver  $y$

- $l$  es el número de subcasos.
- Si  $l=1$  hablamos de reducción o simplificación.
- $ad\ hoc(x)$  es un algoritmo básico.

# Ejemplo: Las Torres de Hanoi



- Problema: Mover los  $n$  discos de la aguja A a la B usando la aguja C como aguja intermedia temporal
- Enfoque Divide y Vencerás:
  - Dos subproblemas de tamaño  $n-1$ :
    - (1) Mover los  $n-1$  discos más pequeños de A a C usando B como intermedia
    - (\*) Mover el disco que queda de A a B
    - (2) Mover los  $n-1$  discos más pequeños de C a B usando A como intermedia
  - El movimiento de los  $n-1$  discos más pequeños se hace con la aplicación recursiva del metodo



# Requisitos para usar DyV

- El problema tiene que poder descomponerse en subproblemas, más sencillos de resolver que el original.
- Los subproblemas deben ser del mismo tipo entre ellos y con el original.
- Los subproblemas se resuelven independientemente (casi siempre).
- No existe solapamiento entre subproblemas.
- Ha de tener sentido combinar las soluciones de los subproblemas para obtener la solución final.

# Condiciones para que DyV sea ventajoso

- Selección cuidadosa de cuándo utilizar el algoritmo ad hoc (calcular el umbral de recursividad).
- Poder descomponer el problema en subproblemas y recombinar de forma bastante eficiente a partir de las soluciones parciales.
- el número / de subproblemas debe ser razonablemente pequeño.
- Los subproblemas deben tener aproximadamente el mismo tamaño.
- Los subproblemas deben de ser del menor tamaño posible.

- 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 inputs de menor tamaño.
- En el caso de DyV nos encontraremos recurrencias como:

$$T(n) = \begin{cases} t(n) & \text{si } n \leq n_0 \\ lT(n/b) + G(n) & \text{si } n > n_0 \end{cases}$$

- donde  $l$  es el numero de subproblemas y  $n/b$  el tamaño de estos.
- $G(n) = D(n) + C(n)$ , y  $D(n)$  es el tiempo de dividir el problema en los subproblemas y  $C(n)$  el tiempo de combinación de las soluciones de los subproblemas.

$$T(n) = IT(n/b) + G(n)$$

Si  $G(n) \in \Theta(n^k)$ , entonces  $T(n)$  es de orden:

- $\Theta(n^k)$  si  $l < b^k$
- $\Theta(n^k \log n)$  si  $l = b^k$
- $\Theta(n^{\log_b l})$  si  $l > b^k$

El orden de eficiencia depende de la relación entre el número de subproblemas ( $l$ ), el tamaño de los subproblemas ( $b$ ) y la dificultad de dividir y combinar ( $k$ ).

# La importancia de las condiciones

- El número de subproblemas importa mucho

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

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

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

- La eficiencia en combinar las soluciones importa mucho

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

$$t(n) = 2t(n/2) + n \implies t(n) \in O(n \log n)$$

$$t(n) = 2t(n/2) + n^2 \implies t(n) \in O(n^2)$$

# La importancia de las condiciones 2

- Que los problemas sean aprox. del mismo tamaño importa mucho

$$t(n) = 2t(n/2) + n \implies t(n) \in O(n \log n)$$

$$t(n) = t(1) + t(n-1) + n \implies t(n) \in O(n^2)$$

- El tamaño de los subproblemas importa mucho

$$t(n) = 2t(n/4) + c \implies t(n) \in O(n^{\log_4 2}) = O(n^{0,5}) = O(\sqrt{n})$$

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

$$t(n) = 2t(n-1) + c \implies t(n) \in O(2^n)$$

# Ejemplo: Valor máximo

- Problema: Dado un vector de elementos, determinar la posición que ocupa el valor máximo del mismo.
- ¿Se puede aplicar DyV para resolver este problema?
- ¿Resuelve nuestro algoritmo el problema?
- Lo hace suficientemente bien?

# Ejemplo: Picos y valles

- Dada una secuencia de números, se define pico al valor  $i$  tal que  $x[i - 1] < x[i] > x[i + 1]$  y valle al valor  $j$  tal que  $x[j - 1] > x[j] < x[j + 1]$ .
- Un pico  $i$  es consecutivo a un valle  $j$  si ningún valor entre  $i$  y  $j$  es pico o valle.

Problema: Diseñar un algoritmo que permita encontrar la mayor diferencia entre un pico y un valle consecutivos.



# Ejemplo: Selección de puntos de parada

- Un camión va desde Granada a Moscú siguiendo una ruta predeterminada.
- La capacidad del tanque de combustible es  $C$ , y conocemos el consumo por Km. del camión
- Conocemos las gasolineras que se encuentran en la ruta (y la distancia entre ellas).

Problema: Minimizar el número de paradas que hace el conductor.

# Ejemplo: Multiplicación de Enteros Grandes

- Chequear si un número es primo requiere muchas multiplicaciones de enteros grandes (desde dos a millones de dígitos). Útil en criptografía.
- Para resolver este problema debemos implementar algoritmos eficientes capaces de trabajar con estos valores.
  - Método clásico (escuela)
  - Método basado en Divide y Vencerás

Tamaño:  $n$  = número dígitos

- Algoritmo clásico:  $1234 * 5678 =$   
 $1234 * [5 * 1000 + 6 * 100 + 7 * 10 + 8] =$
- Operaciones básicas:
  - Multiplicaciones de dígitos  $O(1)$
  - Sumas de dígitos  $O(1)$
  - Desplazamientos  $O(1)$
- Eficiencia algoritmo:  $O(n^2)$

- Para aplicar DyV debemos de poder obtener la solución en base a problemas de tamaño menor
- Truco:
  - $5632 = 56 \cdot 100 + 32$  y  $3427 = 34 \cdot 100 + 27$
  - $(56 \cdot 100 + 32) \cdot (34 \cdot 100 + 27) =$   
Se reduce la multiplicación de 4 cifras a cuatro multiplicaciones de 2 cifras, más tres sumas y varios desplazamientos  
 $56 \cdot 34 \cdot 10000 + (56 \cdot 27 + 32 \cdot 34) \cdot 100 + (32 \cdot 27)$

- Dividir

$X = 12345678$

- $xi = 1234$        $xd = 5678$
- $X = xi * 10^4 + xd$

$Y = 24680135$

- $yi = 2468$        $yd = 0135$
- $Y = yi * 10^4 + yd$

- Combinar

$$X \times Y = (xi * 10^4 + xd)(yi * 10^4 + yd) =$$
$$xi * yi * 10^8 + (xi * yd + xd * yi) * 10^4 + xd * yd$$

En general

- $X = xi * 10^{n/2} + xd$
- $Y = yi * 10^{n/2} + yd$
- $X * Y = (xi * yi) * 10^n + (xi * yd + xd * yi) * 10^{n/2} + xd * yd$

```

Función DVbásico (X,Y,n) {
  if P es pequeño return X*Y;
  else {
    Obtener xi, xd, yi, yd;                                //DIVIDIR

    z1 = DVbásico (xi, yi, n/2);
    z2 = DVbásico (xi, yd, n/2);
    z3 = DVbásico (xd, yi, n/2);
    z4 = DVbásico (xd, yd, n/2);

    aux= Sumar(z2,z3);                                     //COMBINAR
    z1 = DesplazarDcha(z1,n);
    aux = DesplazarDcha(aux,n/2);
    z = Sumar(z1,aux,z4);
    return z;
  }
}

```

- Eficiencia:
  - $T(n) = 4T(n/2) + cn$
  - Como  $l = 4 > b^k = 2^1 = 2$ ,  $T(n)$  está en el orden  $O(n^{\log_2 4}) = O(n^2)$
- El cuello de botella está en el número de multiplicaciones de tamaño  $n/2$ : 4
- Para mejorar la eficiencia necesitamos reducir el número de multiplicaciones que hacemos.



# Algoritmo DyV mejorado

- Sean

- $r = (x_i + x_d) * (y_i + y_d) = (x_i * y_i) + (x_i * y_d + x_d * y_i) + x_d * y_d$
- $p = x_i * y_i$
- $q = x_d * y_d$
- Luego  $x_i * y_d + x_d * y_i = r - p - q$

- Podemos calcular

$$X * Y = p * 10^n + (r - p - q) * 10^{n/2} + q$$

- 1 multiplicación tamaño  $n \implies 3$  multiplicaciones de tamaño  $n/2$

```

Función DV (X,Y,n) {
  if P es pequeño return X*Y;
  else {
    Obtener xi, xd, yi, yd;           //DIVIDIR
    s1 = Sumar(xi,xd);
    s2 = Sumar(yi,yd);

    p = DV (xi,yi,n/2);
    q = DV (xd,yd,n/2);
    r = DV (s1,s2,n/2);

    aux = Sumar(r,-p,-q);           //COMBINAR
    p = DesplazarDcha(p,n);
    aux = DesplazarDcha(aux,n/2);
    z = Sumar(p,aux,q);
    return z;
  }
}

```

- eficiencia

- $T(n) = 3T(n/2) + O(n)$
- $T(n) \in O(n^{\log_2 3}) = O(n^{1,585})$

- Ejemplo de diferencias de tiempo:

$n$	$n^2$	$n^{1,585}$
10	100	38.46
100	10000	1479.11
1000	1000000	56885.29
10000	100000000	2187751.62

# Determinación del umbral

- Es difícil hablar del umbral  $n_0$  si no tratamos con implementaciones, ya que gracias a ellas conocemos las constantes ocultas que nos permitirán afinar el cálculo de dicho valor.
- El umbral no es único, pero si lo es en cada implementación.
- En principio no hay restricciones sobre el valor que puede tomar  $n_0$ , por tanto variará entre uno e infinito.
  - Un umbral de valor infinito supone no aplicar nunca DyV de forma efectiva, porque estaríamos resolviendo con el algoritmo básico siempre.
  - Si  $n_0 = 1$ , entonces estaríamos en el caso opuesto, ya que el algoritmo básico sólo actúa una vez, y se aplica la recursividad continuamente.

# Umbral: ejemplo para enteros grandes

$$t(n) = \begin{cases} cn^2 & \text{si } n \leq n_0 \\ 3t(n/2) + dn & \text{si } n > n_0 \end{cases}$$

- Para una implementación hipotética en la que  $c = 1$  y  $d = 16$  (ms), y un caso de tamaño  $n = 1024$
- Las dos posibilidades extremas nos llevan a
  - Si  $n_0 = 1$ ,  $t(1024) = 32$  m y 34sg
  - Si  $n_0 = +\infty$ ,  $t(1024) = 17$  m y 40 sg
- Si puede haber tan grandes diferencias, ¿cómo podremos determinar el valor óptimo del umbral?

# Umbral: ejemplo para enteros grandes

Con una implementación real:

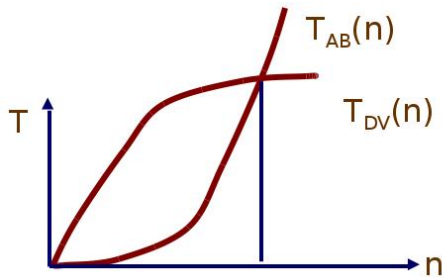
- Si umbral es igual a 1, entonces
  - DyV (5.000 cifras)  $\Rightarrow$  41 seg.
  - Clásico (5.000 cifras)  $\Rightarrow$  25 seg
  - A partir de 32.789 cifras es mejor DyV (15 minutos !!!)
- Si umbral es igual a 64
  - DyV (5.000 cifras)  $\Rightarrow$  6 seg.
  - DyV(32.789 cifras)  $\Rightarrow$  2 minutos !!

La selección del umbral es problemática:

- No afecta al orden del algoritmo DyV pero sí a las constantes ocultas.
- Depende del algoritmo y de la implementación.
- Se estima empíricamente.

# Umbral: Método experimental

- Implementamos el algoritmo básico (AB) y el algoritmo DyV
- Resolvemos para distintos valores de  $n$  con ambos algoritmos
- Hay que esperar que conforme  $n$  aumente, el tiempo del algoritmo básico vaya aumentando más deprisa que el del DyV.



# Umbral: Método teórico

- Comparar el tiempo del algoritmo básico con el del DyV usando sólo un nivel de recursividad

$$t(n) = \begin{cases} h(n) & \text{si } n \leq n_0 \\ 3t(n/2) + g(n) & \text{si } n > n_0 \end{cases}$$

- Determinar el valor de  $n$  para el que los tiempos coinciden

$$h(n) = t(n) = 3h(n/2) + g(n)$$

- Para una implementación concreta (por ejemplo, la anterior,  $h(n) = n^2$  y  $g(n) = 16n$  (ms))

$$n^2 = 3(n/2)^2 + 16n = 3/4n^2 + 16n \implies n = 3/4n + 16$$



$$n_0 = 64$$



# Umbral: Método híbrido

- Calculamos las constantes ocultas utilizando un enfoque empírico.
- Calculamos el umbral, igualando los tiempos del algoritmo básico y el DyV.
- Probamos valores alrededor del umbral teórico (umbrales de tanteo) para determinar el umbral óptimo.

# Búsqueda binaria

- En esencia es el algoritmo que se emplea para buscar una palabra en un diccionario o un nombre en un directorio telefónico.
- Es tal vez la aplicación más sencilla de DyV. Realmente es un caso de reducción o simplificación: la solución de todo caso se reduce a un único caso más pequeño (concretamente de tamaño mitad).
- Sea  $V[1..n]$  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  $V[i - 1] < x \leq V[i]$  (con la convención lógica de que  $V[0] = -\infty$  y  $V[n + 1] = +\infty$ ).

# Búsqueda. Ejemplo

0	$-\infty$
1	3
2	7
3	25
4	41
5	53
6	$\infty$

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 hacer una búsqueda secuencial hasta que lleguemos al final o encontremos un elemento que no sea menor que  $x$

```
funcion secuencial(V[1..n], x) {  
    for i=1 to n  
        if V[i] >= x return i;  
    return n+1;  
}
```

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

# Búsqueda binaria: 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  $V[1..k]$ ; en otro caso buscamos en  $V[k + 1..n]$ .
- Eficiencia:
  - Si  $n$  es el tamaño del vector
  - $T(n) = T(n/2) + c$
  - Como  $1 = 1 = b^k = 2^0$ , entonces  $T(n) = O(\log n)$

# Búsqueda binaria: algoritmo

```
Funcion BuscaBin(V[1..n])  
  if n = 0 or x > V[n]  
  then return n+1;  
  return Binrec (V[1..n],x);
```

```
Funcion Binrec(V[i..j],x)  
  if i = j then return i;  
  k = (i + j) div 2;  
  if x <= V[k]  
  then return Binrec (V[i..k],x);  
  else return Binrec (V[k+1,j],x);
```

Se puede transformar fácilmente en un método iterativo en vez de recursivo.

# Multiplicación de Matrices

- Si tenemos dos matrices A y B cuadradas del mismo tamaño ( $n \times n$ ), se trata de multiplicar A y B para obtener una nueva matriz C.
- La multiplicación de matrices se realiza mediante

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$$

- Esta fórmula corresponde a la multiplicación normal de matrices, que consiste en tres bucles anidados, por lo que es  $O(n^3)$ .
- Para aplicar la técnica DyV, vamos a proceder como con la multiplicación de enteros, con la intención de obtener un algoritmo más eficiente para multiplicar matrices.

- La multiplicación puede hacerse como sigue:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae+bf & ag+bh \\ ce+df & cg+dh \end{pmatrix}$$

$\Uparrow$   
 $C$

$\Uparrow$   
 $A$

$\Uparrow$   
 $B$

- Esta formulación divide una matriz  $n \times n$  en cuatro matrices de tamaños  $n/2 \times n/2$ , con lo que divide el problema en 8 subproblemas de tamaños  $n/2$ .
- $n$  se usa como tamaño del caso aunque la dimensión de la matriz es  $n^2$



- Este enfoque da lugar a la siguiente recurrencia:

$$T(n) = 8T(n/2) + cn^2$$

- Como  $l = 8 > b^k = 2^2 = 4$ , entonces  $T(n)$  es de orden  $O(n^{\log_2 8}) = O(n^3)$ .
- Pero, basándonos en el enfoque DyV que empleamos para multiplicar enteros, la multiplicación de matrices también puede calcularse de forma más eficiente.

# Algoritmo de Strassen

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae+bf & ag+bh \\ ce+df & cg+dh \end{pmatrix}$$

$\begin{matrix} \Uparrow & \Uparrow & \Uparrow \\ C & A & B \end{matrix}$

$$P = (a+d)(e+h)$$

$$Q = (c+d)e$$

$$R = a(g-h)$$

$$S = d(f-e)$$

$$T = (a+b)h$$

$$U = (c-a)(e+g)$$

$$V = (b-d)(f+h)$$

$$r = P+S-T+V$$

$$s = R+T$$

$$t = Q+S$$

$$u = P+R-Q+U$$

Es evidente que sólo se necesitan 7 multiplicaciones y 18 adiciones/substracciones, en lugar de las anteriores 8.

# Algoritmo de Strassen: eficiencia

- Eficiencia:

$$T(n) = 7T(n/2) + cn^2$$

Luego  $T(n)$  es de orden  $O(n^{\log_2 7}) = O(n^{2,81})$ .

- Se conocen mejoras del algoritmo, pero las rebajas que consiguen son a costa de grandes aumentos en los valores de las correspondientes constantes ocultas, y no mejoran en la práctica el algoritmo de Strassen.
- Las matrices cuadradas cuyo tamaño no sea potencia de 2 se pueden tratar añadiéndoles filas y columnas de ceros, doblando como mucho su tamaño.

# Algoritmos de Ordenación

- La ordenación es una de las tareas más frecuentemente realizadas.
- Los algoritmos de ordenación recibirán una colección de registros a ordenar. Cada registro contendrá un campo clave por el que se ordenarán los registros.
- La clave puede ser de cualquier tipo (numérica, alfanumérica, ...) para el que exista una función de comparación.
- El problema de la ordenación es fijar un conjunto de registros de forma que los valores de sus claves estén en orden no decreciente.
- Esta definición permite la existencia de valores clave **repetidos**. Cuando existen valores clave repetidos puede ser interesante mantener el orden relativo en que ocurren en la colección de entrada.

# Diferentes algoritmos de ordenación

- Lentos  $\Theta(n^2)$  (ordenación por cambio)
  - Ordenación de la burbuja
  - Ordenación por inserción
  - Ordenación por selección
  - son algoritmos sencillos
  - se comportan mal cuando la entrada es muy grande
- Rápidos  $\Theta(n \log n)$ 
  - Ordenación por montículo (Heapsort)
  - Ordenación por fusión o mezcla (Mergesort)
  - Ordenación de Shell (Shellsort)
  - Ordenación rápida (Quicksort)
  - son algoritmos más complejos
  - se comportan muy bien cuando la entrada es muy grande.

# Ordenación por mezcla

- Si  $n = 1$  terminar (toda lista de 1 elemento está ordenada)
- Si  $n > 1$ ,
  - partir la lista de elementos en dos o más sublistas;
  - ordenar cada una de ellas;
  - combinar en una sola lista.

Pero,

- ¿Cómo hacer la partición?
- ¿Cómo combinar las sublistas?

# Mezcla: Cómo hacer la partición (mal)

- Primeros  $n - 1$  elementos en el conjunto A, último elemento en B.
- Ordenar A utilizando este esquema de división recursivamente (B está ordenado)
- Combinar A y B utilizando el método Inserta() (= insertar en un array ordenado )
- Llegamos a una version recursiva del algoritmo de Insercion()
- Número de comparaciones:  $t(n) = t(n - 1) + n \implies O(n^2)$

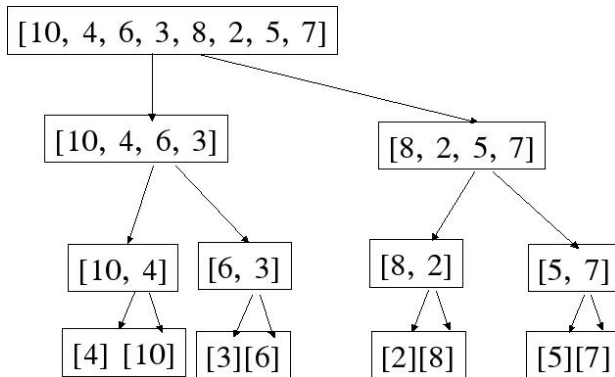
# Mezcla: Cómo hacer la partición (bien)

- Intentemos repartir los elementos de forma equitativa entre los dos conjuntos.
- A toma  $n/k$ , B el resto (habitualmente  $k = 2$ ).
- Ordenar A y B recursivamente.
- Combinar A y B utilizando el proceso de mezcla, que combina las dos listas ordenadas en una.



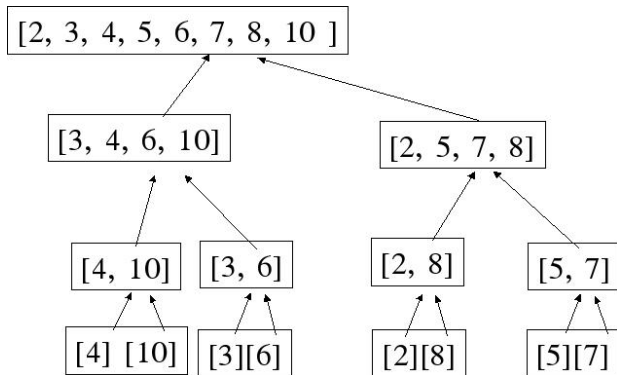
# Mezcla: ejemplo

Se va dividiendo la lista en otras dos de tamaño  $n/2$ :



# Mezcla: ejemplo

La operación de mezcla produce:



# Mezcla: Código

```
void mergeSort(vector<tipo> a, int left, int right)
{
    // sort a[left:right]
    if (left < right)
    {
        // al menos dos elementos
        int mid = (left+right)/2; //punto medio
        copy(u, a, left, mid); //copia a en u
        copy(v, a, mid+1, right); //copia a en v
        mergeSort(u, left, mid);
        mergeSort(v, mid + 1, right);
        merge(a, u, v, left, mid, right); //mezcla en a
    }
}
```

REQUIERE  $O(n)$  espacio adicional !!

# Mezcla: Código de la Combinación

```
void merge(a, u, v, left, mid, right)
{
    j = left;
    k = mid+1;
    for (i = left; i < right; i++) {
        if (u[j] < v[k]) {
            a[i] = u[j];
            j++;
        }
        else{
            a[i] = v[k];
            k++;
        }
    }
}
```

Necesita un pequeño ajuste al llegar al final de un vector.

Suponemos que  $n$  es potencia de 2

$$T(n) = \begin{cases} c_1 & \text{si } n = 1 \\ 2T(n/2) + c_2n & \text{si } n > 1, n = 2^k \end{cases}$$

Podemos intentar la solución por expansión:

$$T(n) = 2T(n/2) + c_2n; \quad T(n/2) = 2T(n/4) + c_2n/2$$

$$T(n) = 4T(n/4) + 2c_2n; \quad T(n) = 8T(n/8) + 3c_2n$$

En general,

$$T(n) = 2^i T(n/2^i) + ic_2n$$

$$T(n) = 2^i T(n/2^i) + ic_2n$$

Tomando  $n = 2^k$ , la expansión termina cuando llegamos a  $T(1)$  en el lado de la derecha, lo que ocurre cuando  $i = k$

$$T(n) = 2^k T(1) + kc_2n$$

Como  $2^k = n$ , entonces  $k = \log n$ ;  
Como además  $T(1) = c_1$ , tenemos

$$T(n) = c_1n + c_2n \log n$$

Por tanto el tiempo para el algoritmo de ordenación por mezcla es  $O(n \log n)$

# Ordenación: Quicksort

- Propuesto por C.A.R. Hoare en 1962.
- Es el algoritmo de ordenación general más eficiente.  
Aprox. el doble de rápido que mergesort.
- Ordena “en el vector” (como inserción o heapsort, pero no como mergesort).
- Muy práctico (con ajustes)
  - Ordena en  $O(n \lg n)$  en caso promedio
  - Ordena en  $O(n^2)$  en el peor caso

# Quicksort: planteamiento

- Ordena el vector eligiendo un valor clave  $p$  entre sus elementos, que actúa como pivote.
- Organiza tres secciones: izquierda, pivote, derecha.
- Todos los elementos en la izquierda son menores o iguales que el pivote, todos los elementos en la derecha son mayores que el pivote.
- Ordena los elementos en la izquierda y en la derecha, sin requerir ninguna mezcla para combinarlos (a diferencia de mergesort, que divide fácilmente pero luego gasta esfuerzo en combinar).
- Lo ideal sería que el pivote se colocara en la mediana para que la parte izquierda y la derecha tuvieran el mismo tamaño.



# Quicksort: pseudocódigo

Algoritmo QUICKSORT( $S$ )

IF TAMAÑO( $S$ )  $\leq$  umbral THEN Insercion( $S$ )

ELSE

    Elegir un elemento  $p$  del array como pivote

    Partir  $S$  en  $(S_i, p, S_d)$  de modo que

        1.  $\forall x \in S_i, z \in S_d$  se verifique  $x \leq p < z$

        2.  $\text{size}(S_i) < \text{size}(S)$  y  $\text{size}(S_d) < \text{size}(S)$

    QUICKSORT( $S_i$ ) // ordena recursivamente parte izda

    QUICKSORT( $S_d$ ) // ordena recursivamente parte dcha

    Combinacion:  $T = S_i + p + S_d$

End Algoritmo

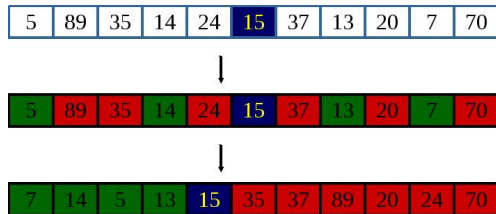
# Quicksort: elección del pivote

- Cada uno puede diseñar su propio algoritmo Quicksort (otra cosa es que funcione mejor que los que ya hay...): La elección del pivote condiciona el tiempo de ejecución.
- El pivote puede ser cualquier elemento en el dominio, pero no necesariamente tiene que estar en S
  - Podría ser la media de los elementos seleccionados en S.
  - Podría elegirse aleatoriamente (pero la función `RAND()` consume tiempo, que habría que añadirse al tiempo total del algoritmo).
- Pivotes usuales son la mediana de un mínimo de tres elementos, o el elemento medio de S.

# Quicksort: 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 array porque lo divide en dos sub-arrays de igual tamaño
  - mediana =  $(n/2)^{\circ}$  mayor elemento
  - elegir tres elementos al azar y escoger su mediana; esto suele reducir el tiempo de ejecución aproximadamente en un 5 %
- La elección más rápida es escoger como pivote, entre los dos primeros elementos del array, el mayor de ellos.

# Quicksort: ejemplo de partición



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

# Quicksort: 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.
- Podemos explorar el vector una sola vez, pero empezando por los dos extremos.

# Quicksort: pivoteo lineal

- Sea  $p = T[i]$  el pivote (el primer elemento del subvector).
- Una buena forma de pivotear consiste en explorar el subvector  $T[i..j]$  solo una vez, pero comenzando desde ambos extremos:
- Los punteros  $k$  y  $l$  se inicializan en  $i$  y  $j + 1$  respectivamente.
- El puntero  $k$  se incrementa entonces hasta que  $T[k] > p$ , y el puntero  $l$  se disminuye hasta que  $T[l] \leq p$ .
- Ahora se intercambian  $T[k]$  y  $T[l]$ . Este proceso continua mientras que  $k < l$ .
- Finalmente,  $T[i]$  y  $T[l]$  se intercambian para poner el pivote en su posición correcta.

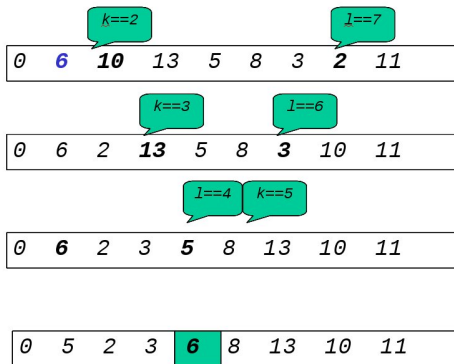
# Quicksort: Algoritmo de pivoteo

Permuta los elementos en el vector  $T[i..j]$  de tal forma que al final  $i \leq l \leq j$ , los elementos de  $T[i..l-1]$  no son mayores que  $p$ ,  $T[l] = p$ , y los elementos de  $T[l+1..j]$  son mayores que  $p$ , donde  $p$  es el valor inicial de  $T[i]$ .

Procedimiento pivote ( $T[i..j], l$ )

```
p=T[i]
k=i; l=j+1;
repetir k=k+1 hasta T[k]>p o k>=j
repetir l=l-1 hasta T[l]<=p
mientras k<l hacer
    intercambiar T[k] y T[l]
    repetir k=k+1 hasta T[k]>p
    repetir l=l-1 hasta T[l]<=p
intercambiar T[i] y T[l]
```

# Ejemplo de pivoteo





# Otro ejemplo de pivoteo

Matriz que hay que ordenar

3	1	4	1	5	9	2	6	5	3	5	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---

La matriz se particiona tomando como pivote su primer elemento,  $p = 3$

<u>3</u>	1	4	1	5	9	2	6	5	3	5	8	9
----------	---	---	---	---	---	---	---	---	---	---	---	---

Se busca el primer elemento mayor que el pivote (subrayado) y el último elemento no mayor que el pivote (superrayado)

3	1	<u>4</u>	1	5	9	2	6	5	<u>3</u>	5	8	9
---	---	----------	---	---	---	---	---	---	----------	---	---	---

Se intercambian esos elementos

3	1	3	1	5	9	2	6	5	4	5	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---

Se vuelve a explorar en ambas direcciones

3	1	3	1	<u>5</u>	9	<u>2</u>	6	5	4	5	8	9
---	---	---	---	----------	---	----------	---	---	---	---	---	---

Se intercambian

3	1	3	1	2	9	5	6	5	4	5	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---

Se explora

3	1	3	1	<u>2</u>	<u>9</u>	5	6	5	4	5	8	9
---	---	---	---	----------	----------	---	---	---	---	---	---	---

Los punteros se han cruzado (el elemento superrayado está a la izquierda del subrayado): se intercambia el pivote con el elemento superrayado.

2	1	3	1	3	9	5	6	5	4	5	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---

La partición ya está completada

Se ordenan recursivamente las submatrices a cada lado del pivote

1	1	2	3	3	4	5	5	5	6	8	9	9
---	---	---	---	---	---	---	---	---	---	---	---	---

# Algoritmo Quicksort

```
Procedimiento quicksort (T[i..j])  
    // ordena un array T[i..j] en orden creciente  
    Si j-i es pequeño entonces Insercion(T[i..j])  
    en caso contrario  
        pivote (T[i..j], l)  
        // tras el pivoteo, si  $i \leq k < l$ ,  $T[k] \leq T[l]$   
        // y si  $l < k \leq j$ ,  $T[k] > T[l]$   
        quicksort (T[i..l-1])  
        quicksort (T[l+1..j])
```

# Quicksort: eficiencia, peor caso

- Si admitimos que
  - El procedimiento de pivoteo es lineal,
  - Quicksort lo llamamos para  $T[0..n-1]$ , y
  - Elegimos como peor caso que el pivote sea el primer elemento del vector,
- Entonces el tiempo del algoritmo es

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

- Que evidentemente proporciona un tiempo cuadrático.
- En el peor caso quicksort es tan malo como el peor caso del metodo de inserción (y también de selección).
- Sin embargo, en la práctica quicksort es el mejor algoritmo de ordenación que se conoce...
- ¿Qué pasará con el tiempo del caso promedio?

# Quicksort: eficiencia, caso promedio

- Suponemos que el vector está dado en orden aleatorio.
- Suponemos que todos los posibles órdenes del vector son igualmente probables (esto puede ser erróneo en algunas aplicaciones, p.e. para ordenar vectores que ya están casi ordenados).
- El pivote puede ser cualquier elemento.
- Puede demostrarse que en el caso promedio quicksort tiene un tiempo  $T(n) = 2n \ln n + O(n)$ , que se debe al número de comparaciones que hace en promedio en un vector de  $n$  elementos.
- Quicksort tiene un tiempo promedio  $O(n \log n)$