Teoría de Algoritmos

Tema 1. Planteamiento General

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos "Divide y Vencerás"

Tema 4. Algoritmos Voraces ("Greedy")

Tema 5. Algoritmos para la Exploración de Grafos ("Backtraking", "Branch and Bound")

Tema 6. Algoritmos basados en Programación Dinámica

Tema 7. Otras Técnicas Algorítmicas de Resolución de Problemas

Tema 3: Algoritmos Divide y Venceras

Bibibliografía:

G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997).
J.L. VERDEGAY. Curso de Teoría de Algoritmos. Universidad de Granada (2004).

Objetivos

- Comprender el principio de Divide y Vencerás
- Conocer las características de un problema resoluble con DV
- Saber calcular el umbral
- Conocer los principales algoritmos de ordenación
- Resolución de diversos subproblemas

Indice

EL ENFOQUE DIVIDE Y VENCERÁS

APLICACIONES DE LA TÉCNICA DIVIDE Y VENCERÁS

1. El Enfoque Divide y Venceras

Introducción

La técnica Divide y Vencerás (DV) consiste en:

- Descomponer el caso a resolver en un cierto número de subcasos más pequeños del mismo problema.
- Resolver sucesiva e independientemente todos estos subcasos.
- Combinar las soluciones obtenidas para obtener la solución del caso original.

Cuestiones:

¿Por qué hacer esto?

¿Cómo se resuelven los subcasos?

2. Método General DV

Procedimiento General

Función DV(x)

si x es suficientemente pequeño entonces

devolver ad hoc(x)

descomponer x en casos más pequeños $x_1, x_2, \ldots, x_l$

para i=1 hasta l

$y_i = DV(x_i)$

recombinar los y_i para obtener una solución y de x

devolver y

- l es el número de subcasos
- si l=1 hablamos de reducción
- ad hoc(x) es un algoritmo básico

2. Método General DV

Características:

- Subproblemas del mismo tipo que el original.
- Los subproblemas se resuelven independientemente
- No existe solapamiento entre subproblemas.

2. Método General DV

Condiciones para que DV sea ventajoso:

- Selección de cuando utilizar el algoritmo ad hoc, calcular el umbral de recursividad.
- Poder descomponer problema en subproblemas y recombinar de forma eficiente a partir de las soluciones parciales.
- Los subcasos deben tener aproximadamente el mismo tamaño.

2. Método General DV

Análisis del Orden de los Algoritmos DV: Fórmula Maestra

Para l subcasos con tamaño n/b

$$t(n) = l\, t(n/b) + g(n)$$

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 $l > b^k$

3. La 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 ==> Depende de la implementación

- De partida no hay restricciones sobre el valor que puede tomar n_0, por tanto variará entre cero e infinito.

  - Un umbral de valor infinito supone no aplicar nunca DV de forma efectiva, porque siempre estaríamos resolviendo con el algoritmo básico siempre.
  - Si n0 = 1, entonces estaríamos en el caso opuesto, ya que el algoritmo básico sólo actúa una vez, y se aplica la recursividad continuamente.

Indice

EL ENFOQUE DIVIDE Y VENCERÁS

1. Enfoque Divide y Vencerás para el Diseño deAlgoritmos
2. Método General DV
3. La Determinación del Umbral

APLICACIONES DE LA TÉCNICA DIVIDE Y VENCERÁS

- Algoritmos de Ordenación
- Multiplicación de Matrices
- Viajante de Comercio

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.
- La clave debe ser de un tipo lo suficientemente grande como para que haya una relación de orden lineal entre las claves.

Problema de Ordenación

- Dados un conjunto de registros $r_1, r_2, \ldots, r_n$ con valores clave $k_1, k_2, \ldots, k_n$ respectivamente, fijar los registros con algún orden s tal que los registros $r_{s1}, r_{s2}, \ldots, r_{sn}$ tengan claves que obedezcan la propiedad $k_{s1} <= k_{s2} <= \ldots <= k_{sn}$.

  En otras palabras, 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.

Algoritmos de Ordenación

- Lentos *Θ (n²) (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 *Θ (n log n)*
  - Ordenación por montículo (Heapsort)
  - Ordenación por fusión (Mergesort)
  - Ordenación de Shell (Shellsort)
  - Ordenación rápida (Quicksort)
  - ✗ son algoritmos más complejos
  - ✓ se comportan muy bien cuando la entrada es muy grande.

Aplicación del DyV a Ordenación

Ordenación por mezcla

- Divide y Venceras:
  - Si n=1 terminar (toda lista de 1 elemento esta ordenada)
  - Si n>1, partir la lista de elementos en dos o mas subcolecciones; ordenar cada una de ellas; combinar en una sola lista.

Pero, ¿Como hacer la partición?

Método 1

- 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 la version recursiva del algoritmo de Insercion()
  - Numbero de comparaciones: $O(n^2)$

Método 2

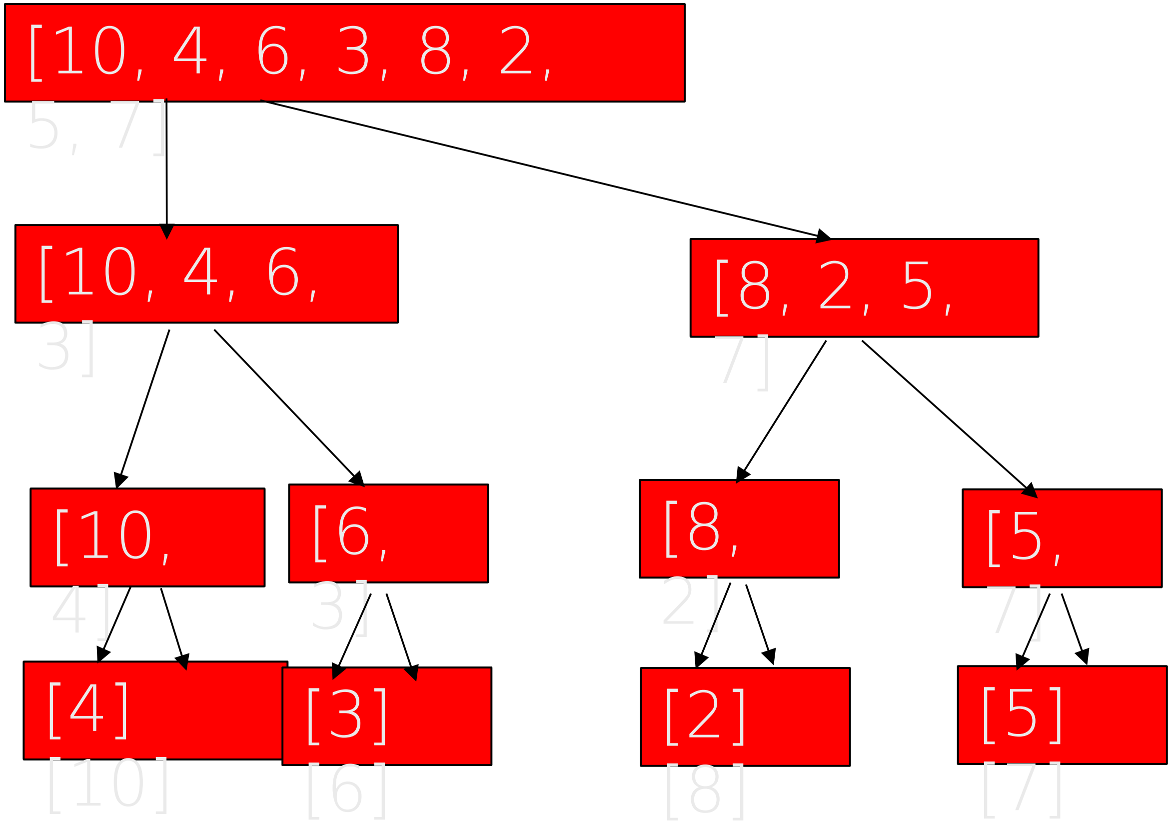
- Intentemos repartir los elementos de forma equitativa entre los dos conjuntos
- A toma n/k, B el resto
- Ordenar A y B recursivamente
- Combinar A y B utilizando el proceso de *mezcla*, que combina las dos listas en una
- .....(consideremos k=2)

Algoritmos de Ordenación

```
Begin  Ordenar(L)
   Si L tiene longitud mayor de 1
   Entonces
      Begin
         Partir la lista en dos listas, izquierda y derecha
         Ordenar(izquierda)
         Ordenar(derecha)
         Combinar izquierda y derecha
      End
End
```

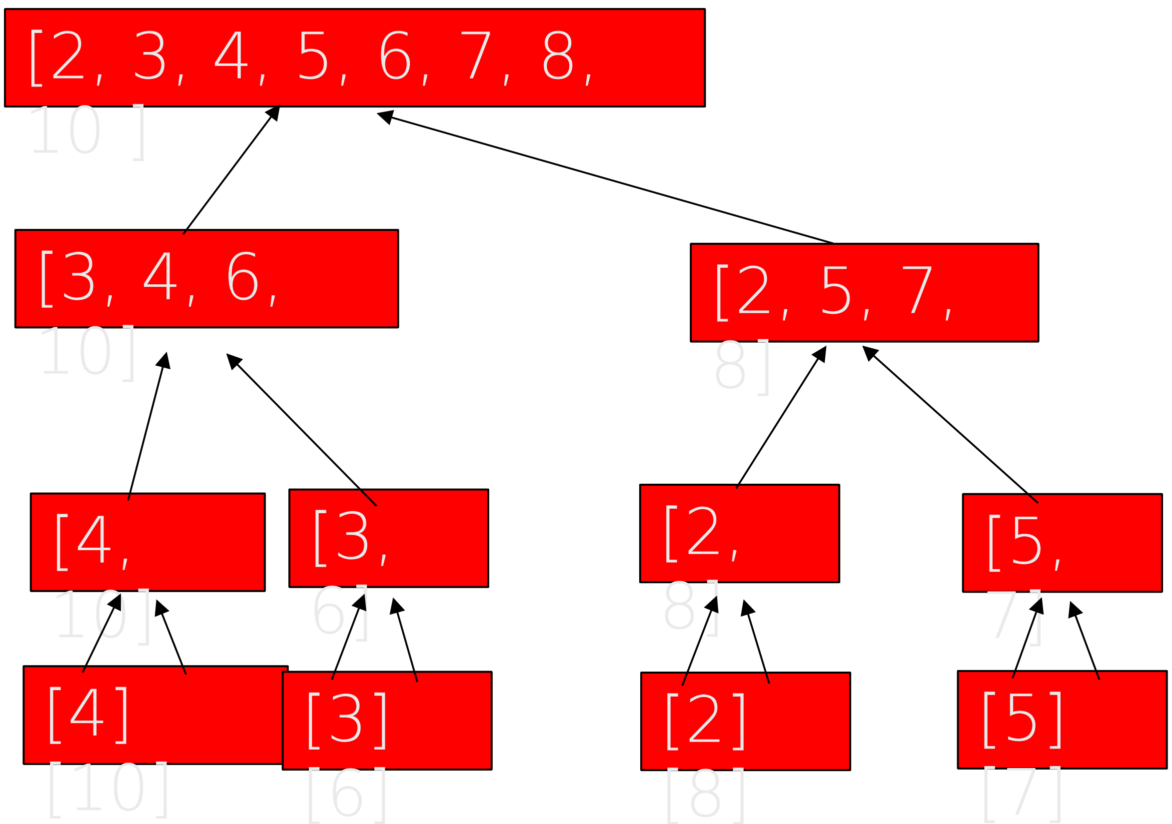
Algoritmos de Ordenación

Ejemplo: k=2 (Partimos la lista en otras dos de tamaño n/2)



Algoritmos de Ordenación

Ejemplo: La operación de mezcla para k=2



Algoritmos de Ordenación

Código de ordenación por mezcla

```
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
      mergeSort(a, left, mid);
      mergeSort(a, mid + 1, right);
      merge(a, b, left, mid, right); // mezclar en vector auxiliar  “b”
      copy(b, a, left, right);  //copia el resultado en a
    }
}
```

REQUIERE O(n) espacio adicional !!!!

Algoritmos de Ordenación

```
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
       mergeSort(a, left, mid);
       mergeSort(a, mid + 1, right);
# merge utilizando un array auxiliar de  n/2
b = copy of a[left..mid]
i = 0, j = mid+1, k = left
while i <= mid and j <= right,
    a[k++] = (a[j] < b[i]) ? a[j++] : b[i++]
    → invariante: a[0..k] se encuentran en posicion correcta
while i <= mid,
    a[k++] = b[i++]
    → invariante: a[0..k] se encuentran en posicion correcta
     }
}
```

Algoritmos de Ordenación

Cálculo de la eficiencia

Ecuación recurrente

- Suponemos que n es potencia de 2

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

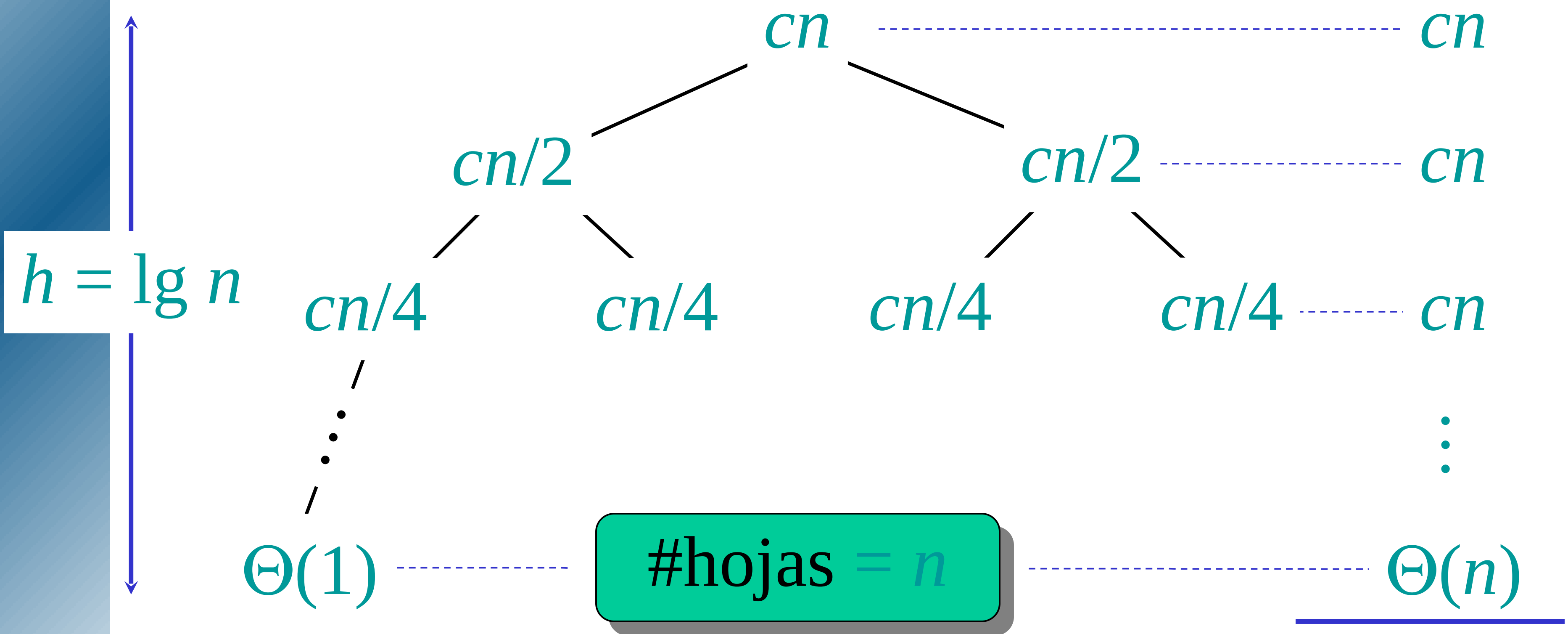
- Tenemos

$$T(n) = c_1 n + c_2 n\log n$$

- Por tanto el tiempo para el algoritmo de ordenacion por mezcla es O(nlogn)

Arbol de Recursion

$T(n) = 2T(n/2) + cn$, con $c > 0$ constante.



Total $= \Theta(n \lg n)$

Ordenación: Quicksort

- Propuesto por C.A.R. Hoare en 1962.
- Algoritmo Divide y Vencerás
- 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 $O(n^2)$ en el peor caso

- Es el algoritmo de ordenación general más eficiente. Aprox. el doble de rápido que mergesort.

Ordenación: Quicksort

- Ordena el array A eligiendo un valor clave v entre sus elementos, que actua como pivote
- organiza tres secciones: izquierda, pivote, derecha
  - todos los elementos en la izquierda son menores que el pivote, todos los elementos en la derecha son mayores o iguales que el pivote
- ordena los elementos en la izquierda y en la derecha, sin requerir ninguna mezcla para combinarlos.
  - lo ideal sería que el pivote se colocara en la mediana para que la parte izquierda y la derecha tuvieran el mismo tamaño

Ordenación: Quicksort

Pseudo Codigo para quicksort

Algoritmo QUICKSORT(S)

IF TAMAÑO(S) ≤ 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 $x<p<z$
2. size(S_i) < size(S) y size(S_d) < size(S)

QUICKSORT(S_i) // ordena recursivamente parte izquierda

QUICKSORT(S_d) // ordena recursivamente parte derecha

Combinacion: T = S_i + p + S_d

End Algoritmo

Ordenación: Quicksort

Operación Clave: La elección del pivote

- **La eleccion 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
  - Podria elegirse aleatoriamente, pero la funcion RAND() consume tiempo, que habria que añadírselo al tiempo total del algoritmo
- Pivotes usuales son la mediana de un mínimo de tres elementos, o el elemento medio de S.

Ordenación: Quicksort

La elección del pivote

- El empleo de la mediana de tres elementos no tiene justificacion teórica.
- Si queremos usar el concepto de mediana, deberiamos escoger como pivote la mediana del array porque lo divide en dos sub-arrays de igual tamaño
  - mediana = (n/2)º mayor elemento
  - elegir tres elementos al azar y escoger su mediana; esto suele reducir el tiempo de ejecucion aproximadamente en un 5%
- La elección más rápida es escoger como pivote, entre los dos primeros elementos del array, el mayor de ellos

Ordenación: Ejemplo Quicksort

array:

5	89	35	10	24	15	37	13	20	17	70

tamaño: 11

Con este ejemplo vamos a ilustrar su funcionamiento

Ordenación: Ejemplo Quicksort

Quicksort: Ejemplo

array:

5	89	35	14	24	15	37	13	20	7	70

"elemento pivote"

Ordenación: Ejemplo Quicksort

Quicksort: Ejemplo

array:

5	89	35	14	24	15	37	13	20	7	70

partición:

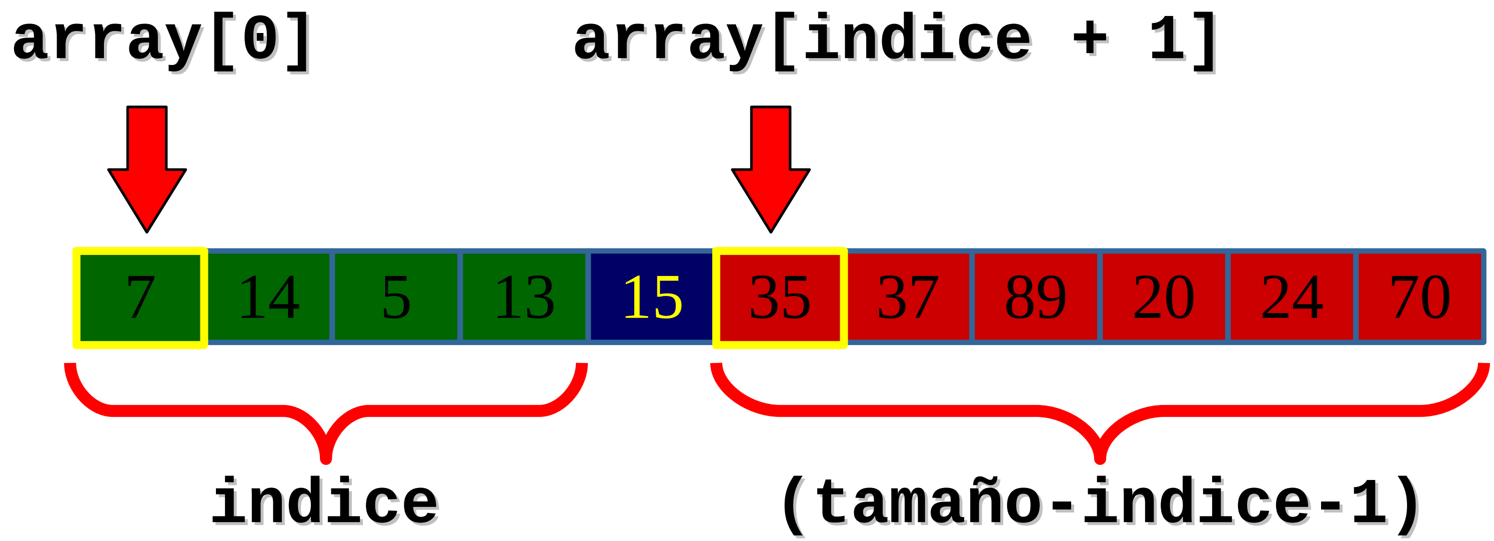
5	89	35	14	24	15	37	13	20	7	70

7	14	5	13	15	35	37	89	20	24	70

indice: 4

Ordenación: Ejemplo Quicksort

Quicksort: Ejemplo



Ordenación: Ejemplo Quicksort

medio: 4

0

array:

5	89	35	14	24	15	37	13	20	7	70

15	89	35	14	24	5	37	13	20	7	70

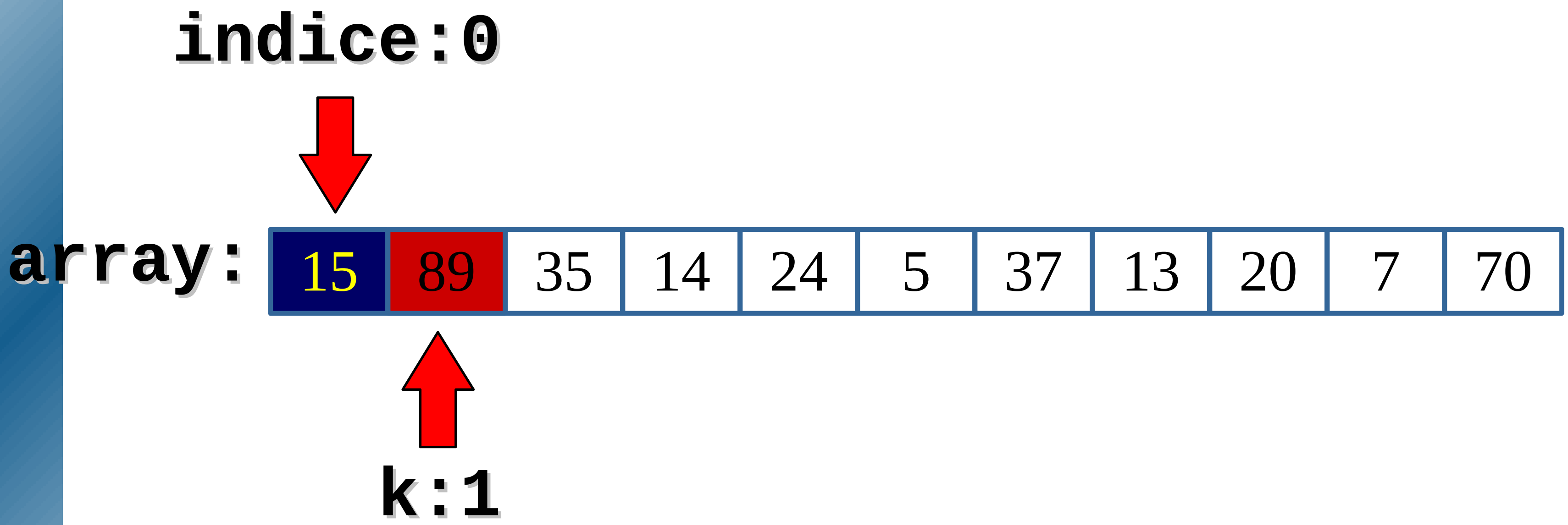
Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort

	0										
array:	15	89	35	14	24	5	37	13	20	7	70
		k=1									

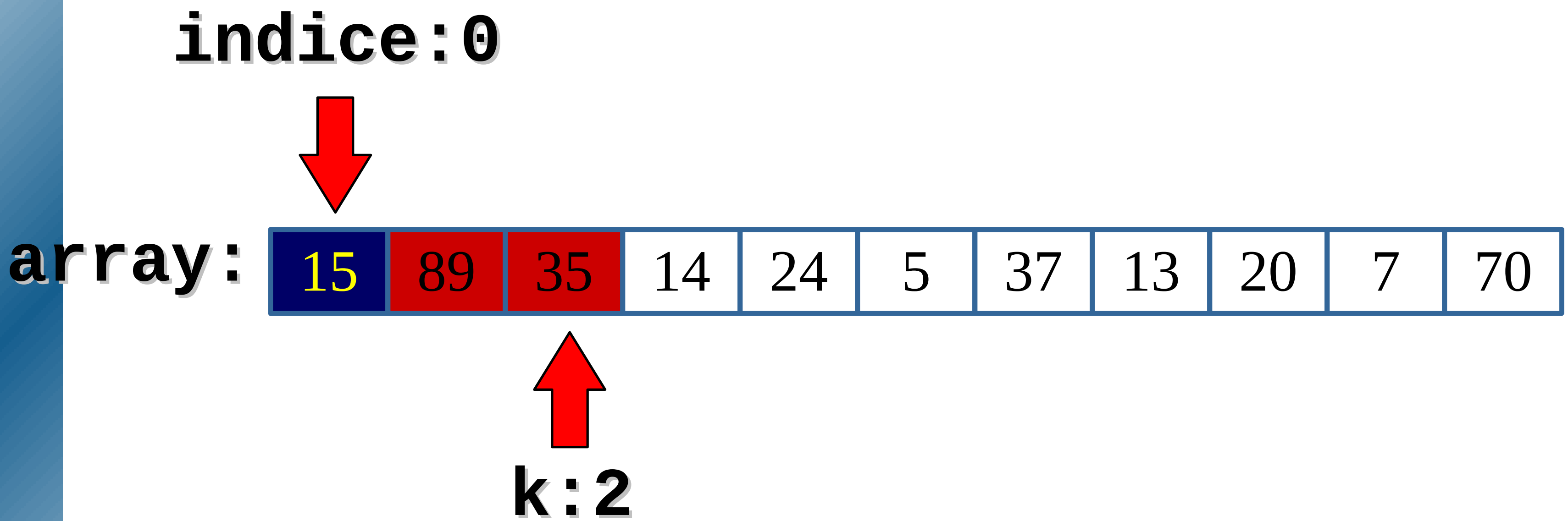
Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort

indice:0

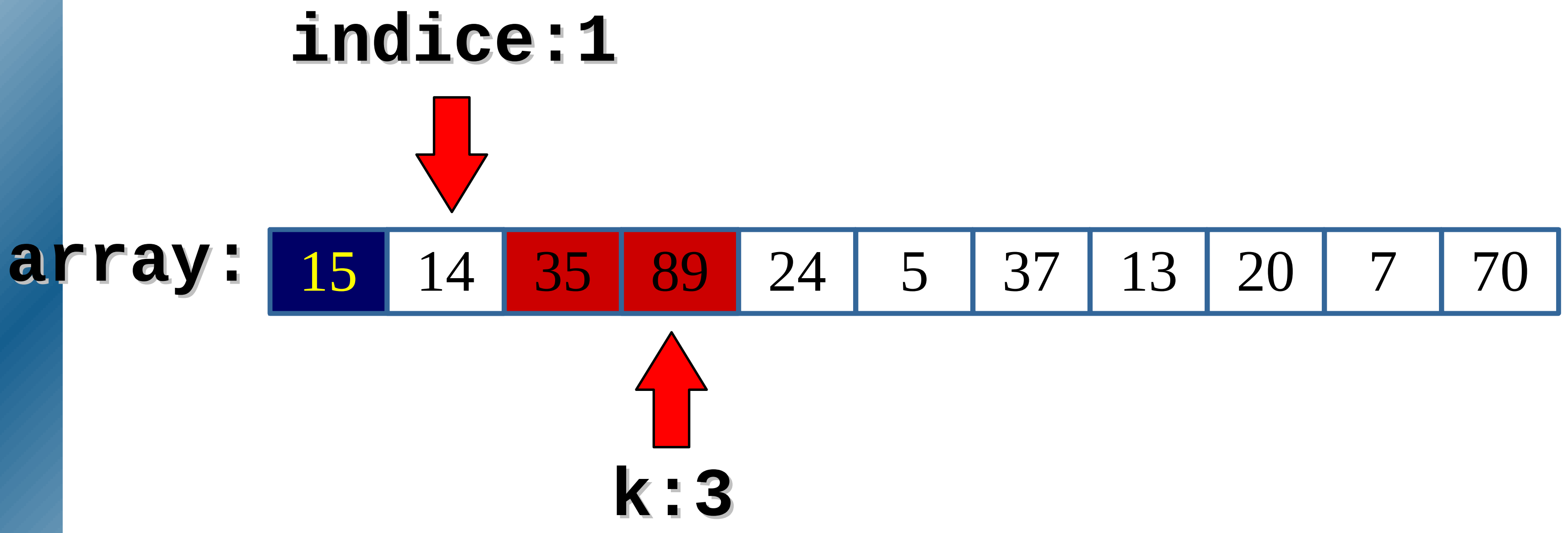
array:

15	89	35	14	24	5	37	13	20	7	70

k:3

Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort

indice:1

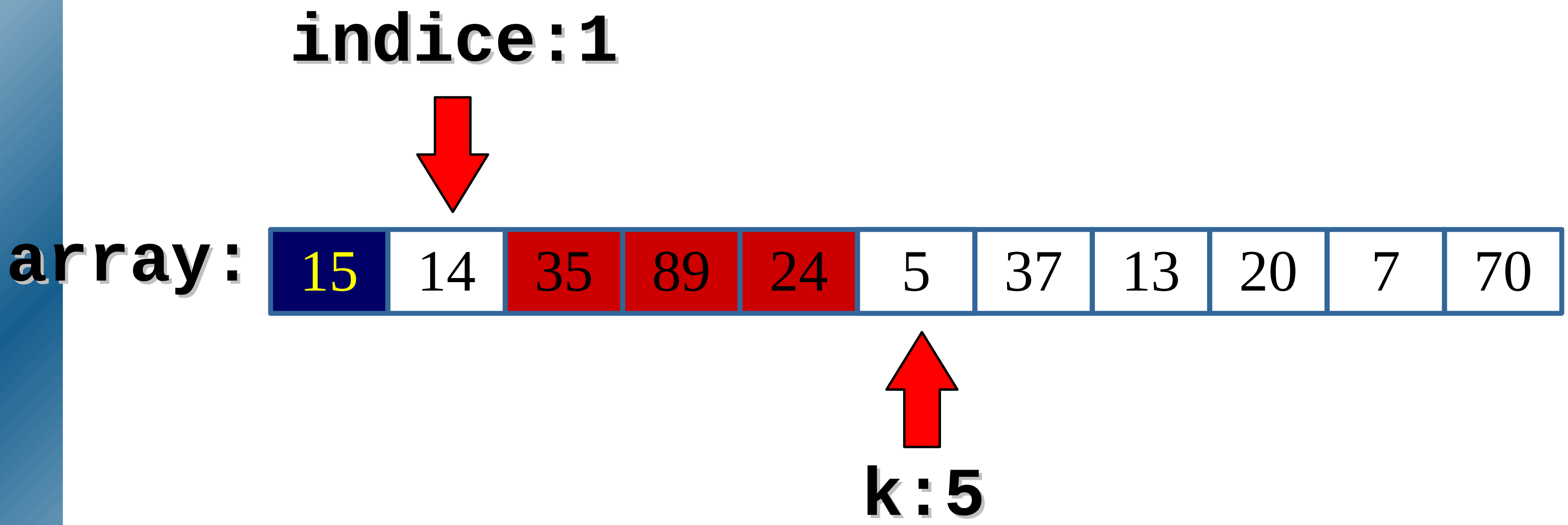
array:

15	14	35	89	24	5	37	13	20	7	70

k:4

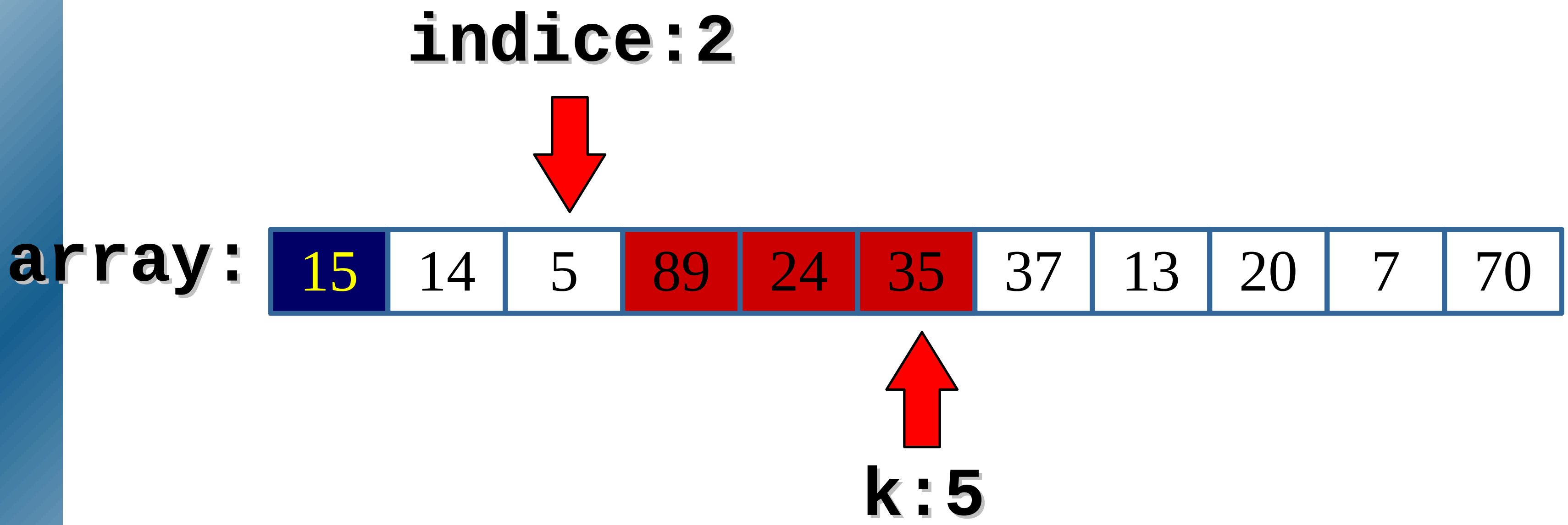
Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



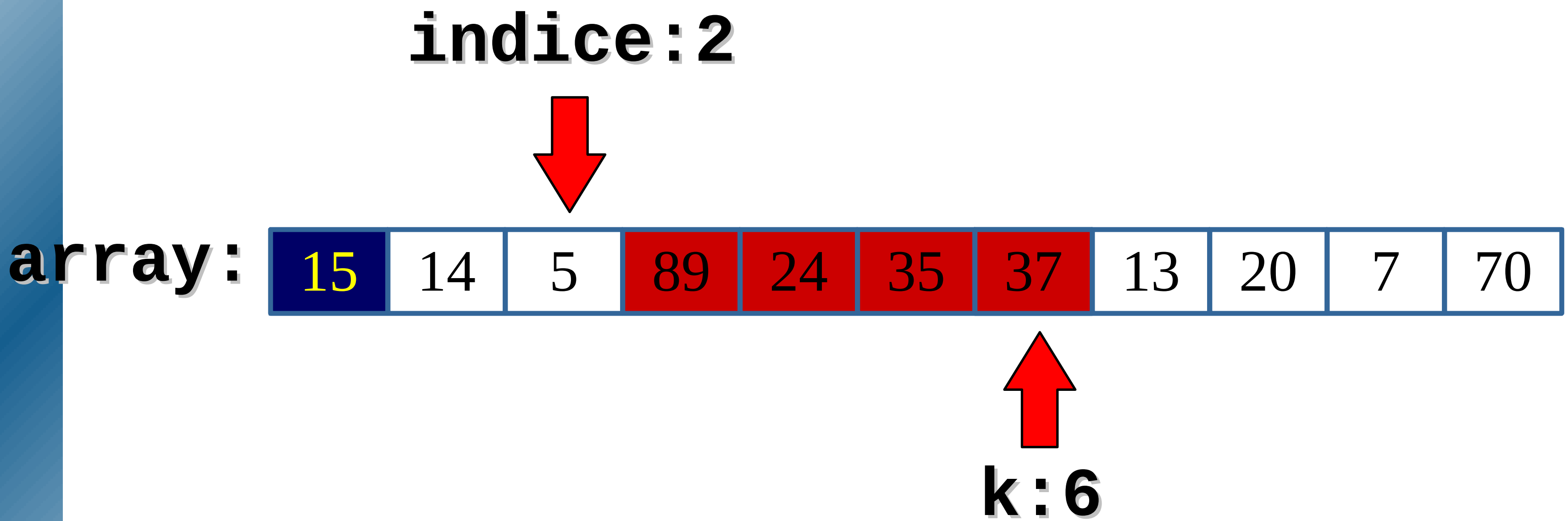
Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



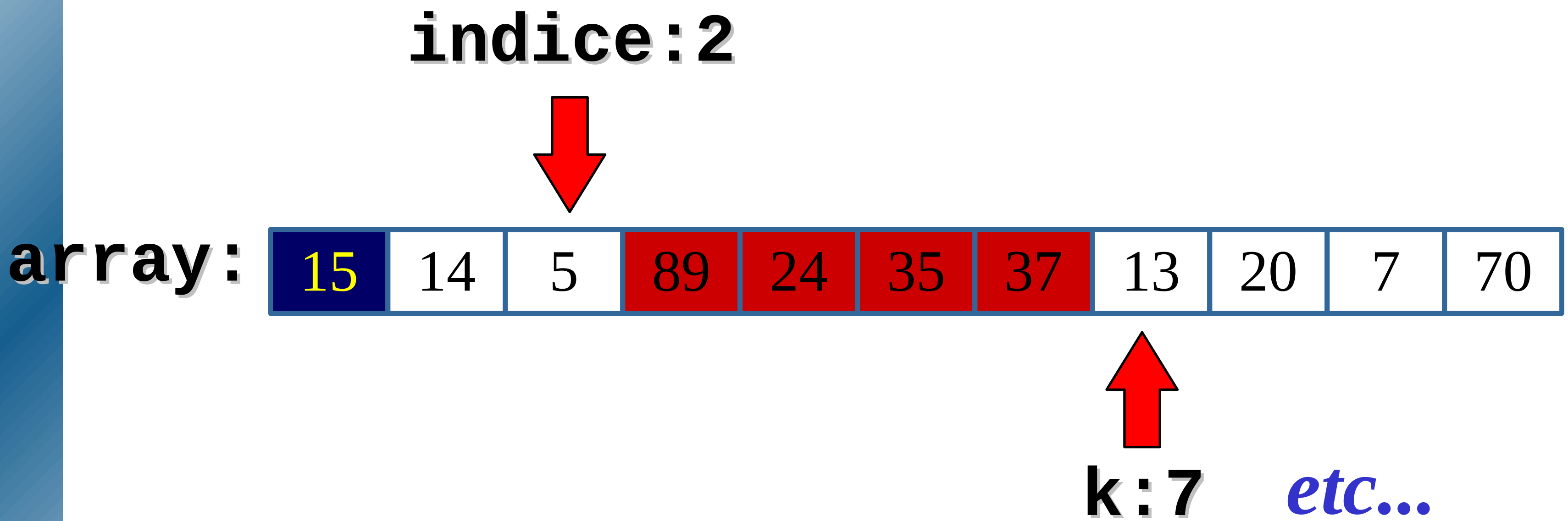
Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



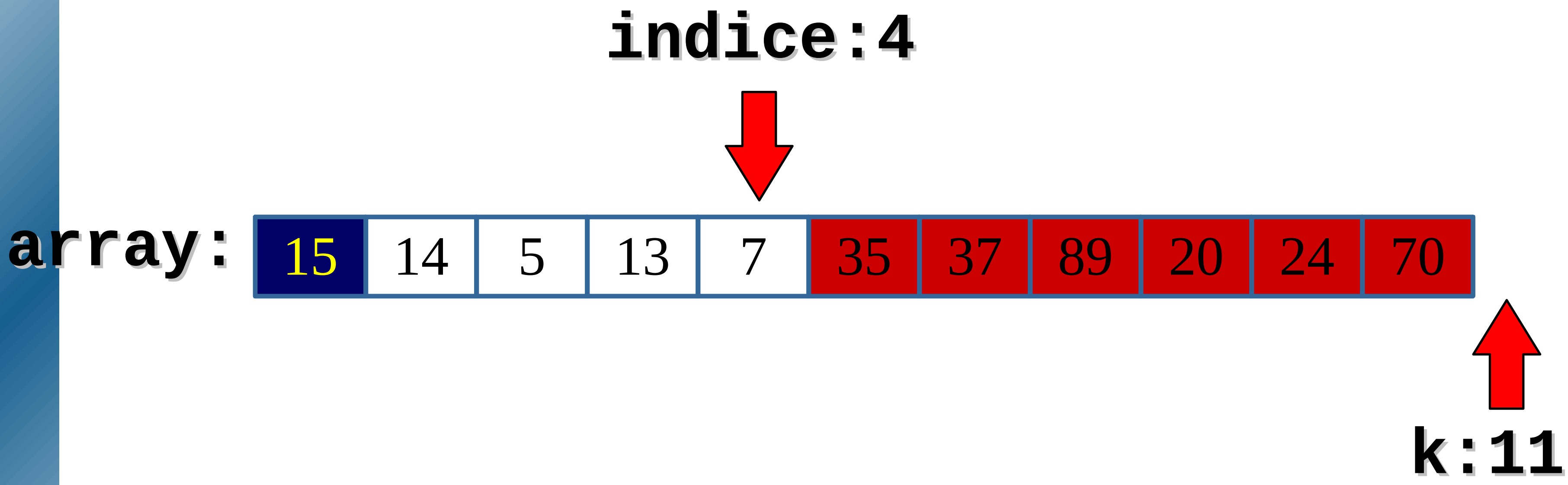
Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort

indice:4

array:

7	14	5	13	15	35	37	89	20	24	70

$x < 15$

$15 \leq x$

Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort

El pivote ahora está en la posicion correcta

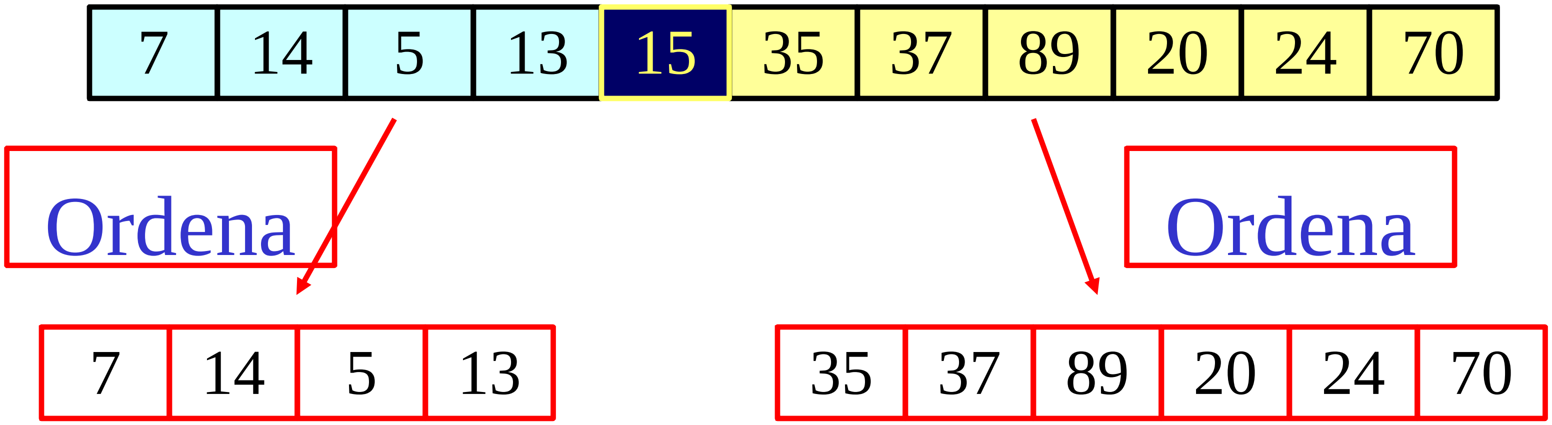
array:

7	14	5	13	15	35	37	89	20	24	70

$x < 15$ **$15 \leq x$**

Quicksort: Ejemplo

Ordenación: Ejemplo Quicksort



Quicksort: Ejemplo

Ordenación: Código QuickSort

Procedimiento quicksort (vector<tipo> &T, int i, int j)

{ordena un array T[i..j] en orden creciente}

Si j-i es pequeño **entonces**

Insercion (T, i, j)

en caso contrario

l = pivote (T, i, j)

{tras pivoteo, $i \leq k < l \Rightarrow T[k] \leq T[l]$ y, $l < k \leq j \Rightarrow T[k] > T[l]$}

quicksort (T, i, l-1)

quicksort (T, l+1, j)

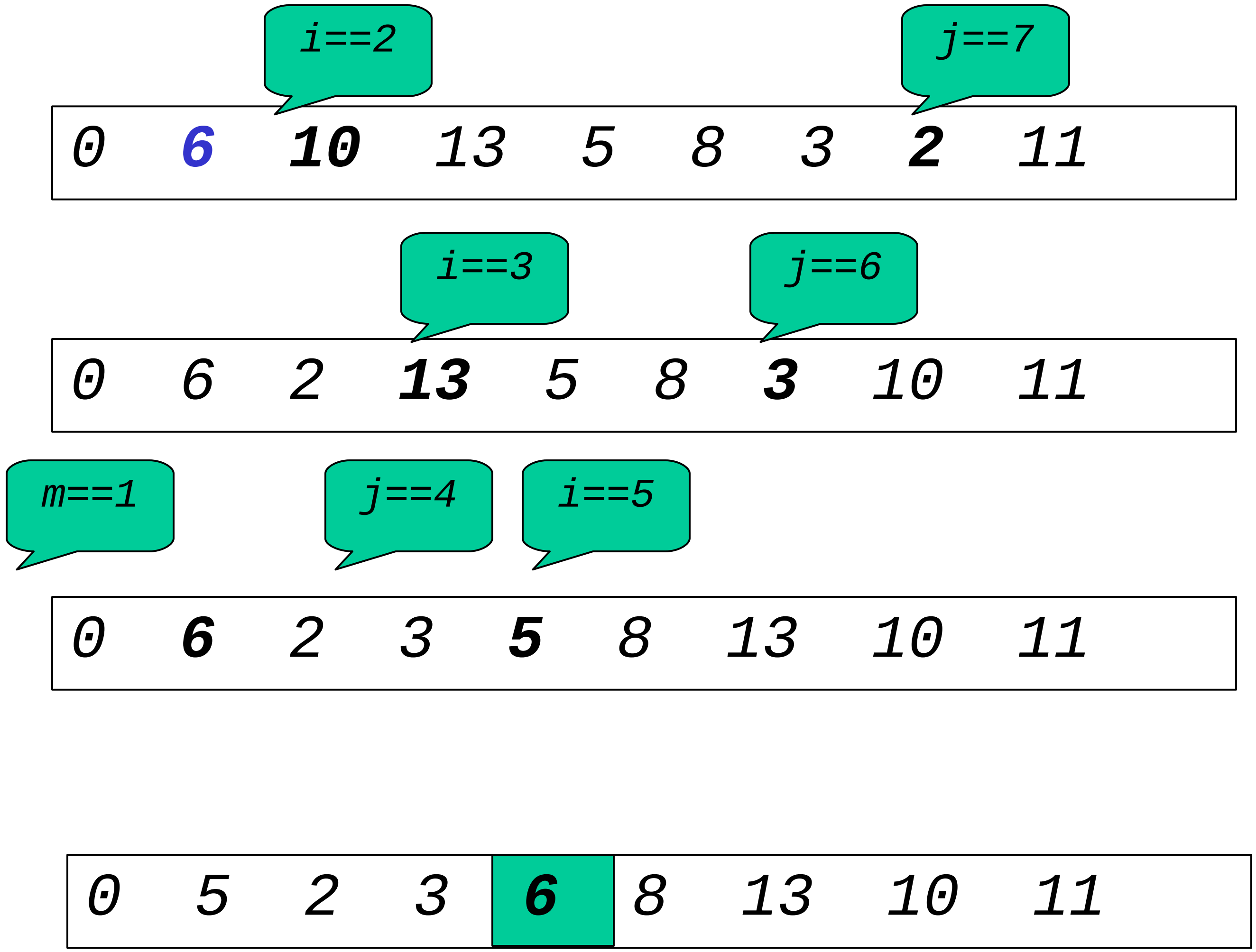
fin si

Ordenación: Quicksort

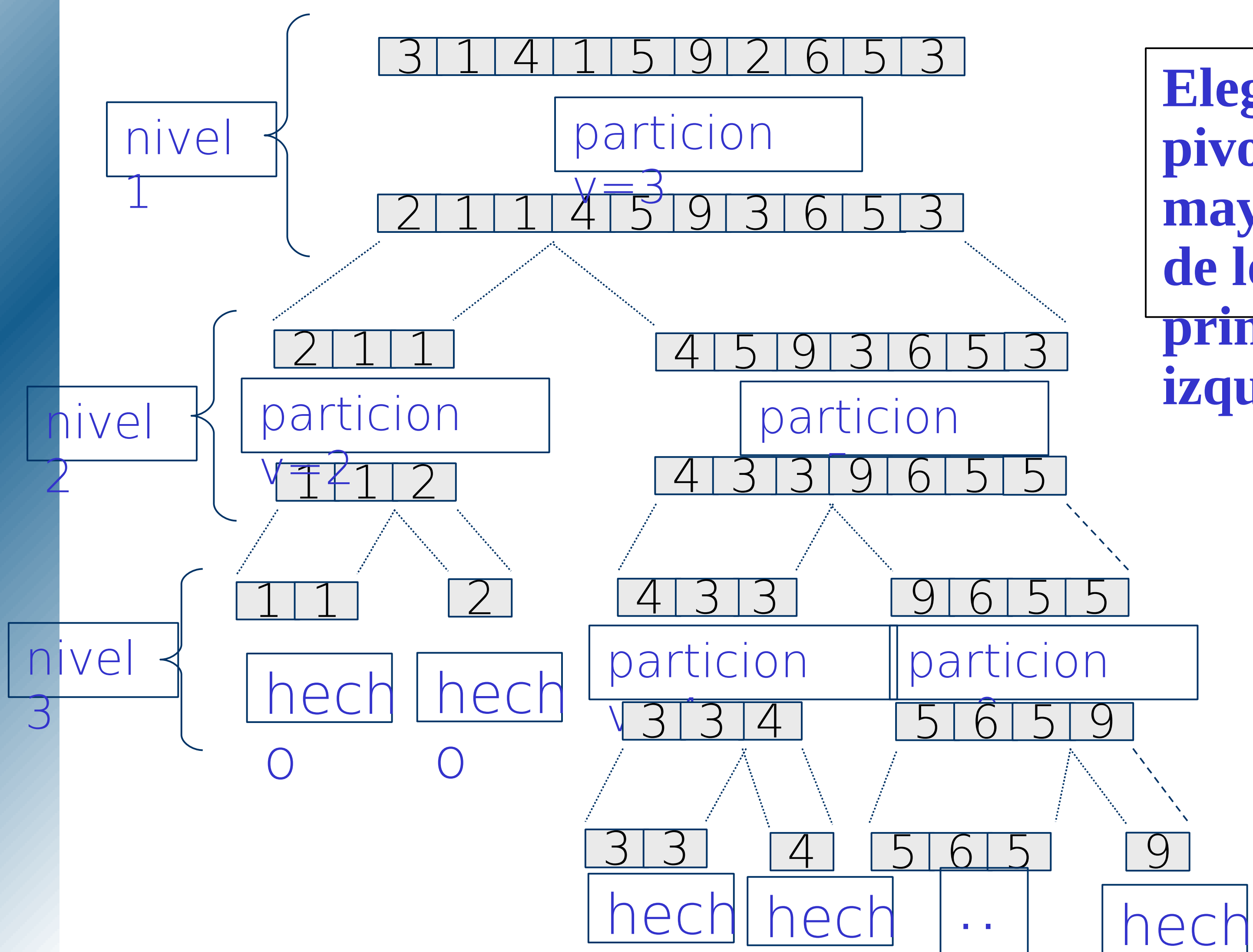
Quicksort: Pivoteo lineal

- Sea p = T[inf] el pivote.
- Una buena forma de pivotear consiste en explorar el array T[inf..sup] solo una vez, pero comenzando desde ambos extremos.
- Los punteros i y j se inicializan en inf y sup respectivamente.
- El puntero i se incrementa entonces hasta que T[i] > p, y el puntero j se disminuye hasta que T[j] ≤ p. Ahora T[i] y T[j] estan intercambiados. Este proceso continua mientras que inf < sup.
- Finalmente, T[inf] y T[j] se intercambian para poner el pivote en su posicion correcta.

```
int Partition(vector<T> &a[], int inf, int sup)
  {  T pivote=a[inf]; int i=inf, j=sup;
     do {
           do i++;
           while (a[i] <= pivote) && (i<j);
           do j--;
           while (a[j] > pivote) && (i<j);
           if (i < j) intercambia(a, i, j);
        } while (i < j);
        a[inf] = a[j]; a[j] = pivote;
        return(j);
  }
```


i==2
j==7
0 6 10 13 5 8 3 2 11
i==3
j==6
0 6 2 13 5 8 3 10 11
m==1
j==4
i==5
0 6 2 3 5 8 13 10 11
0 5 2 3 6 8 13 10 11

Algoritmos de Ordenación



Elegimos el pivote como el mayor elemento de los dos primeros por la izquierda

Ordenación: Quicksort, Eficiencia

- 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 array,
- Entonces el tiempo del anterior algoritmo es

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

- Que evidentemente proporciona un tiempo cuadrático

Ordenación: Quicksort, Eficiencia

Analisis de Quicksort

- Recordemos que el algoritmo de ordenacion por Inserción hacia aproximadamente 1/2 $n^{\mathbf{2}}$ - 1/n comparaciones, es decir es O($n^{\mathbf{2}}$) en el peor caso.
- En el peor caso quicksort es tan malo como el peor caso del método de inserción (y también de selección).
- Es que el numero de intercambios que hace quicksort es unas 3 veces el numero de intercambios que hace el de inserción.
- Sin embargo, en la practica quicksort es el mejor algoritmo de ordención que se conoce...
- ¿Que pasará con el tiempo del caso promedio?

Algoritmos de Ordenación

Análisis del caso promedio

- Suponemos que la lista esta dada en orden aleatorio
- Suponemos que todos los posibles ordenes del array son igualmente probables
- 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 una lista de n elementos
- Quicksort, tiene un tiempo promedio O(n log n)

Aun más

Suponer que alternamos suerte (L), no suerte (U), L, U, L, U,.....

$$L(n) = 2U(n/2) + \Theta(n) \quad \textbf{\textit{Suerte}}$$

$$U(n) = L(n-1) + \Theta(n) \quad \textbf{\textit{No suerte}}$$

Resolvemos:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Como podemos asegurar que tendremos suerte?

Análisis del mejor-caso

Si tenemos suerte, se divide el array por la mitad

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n)$$

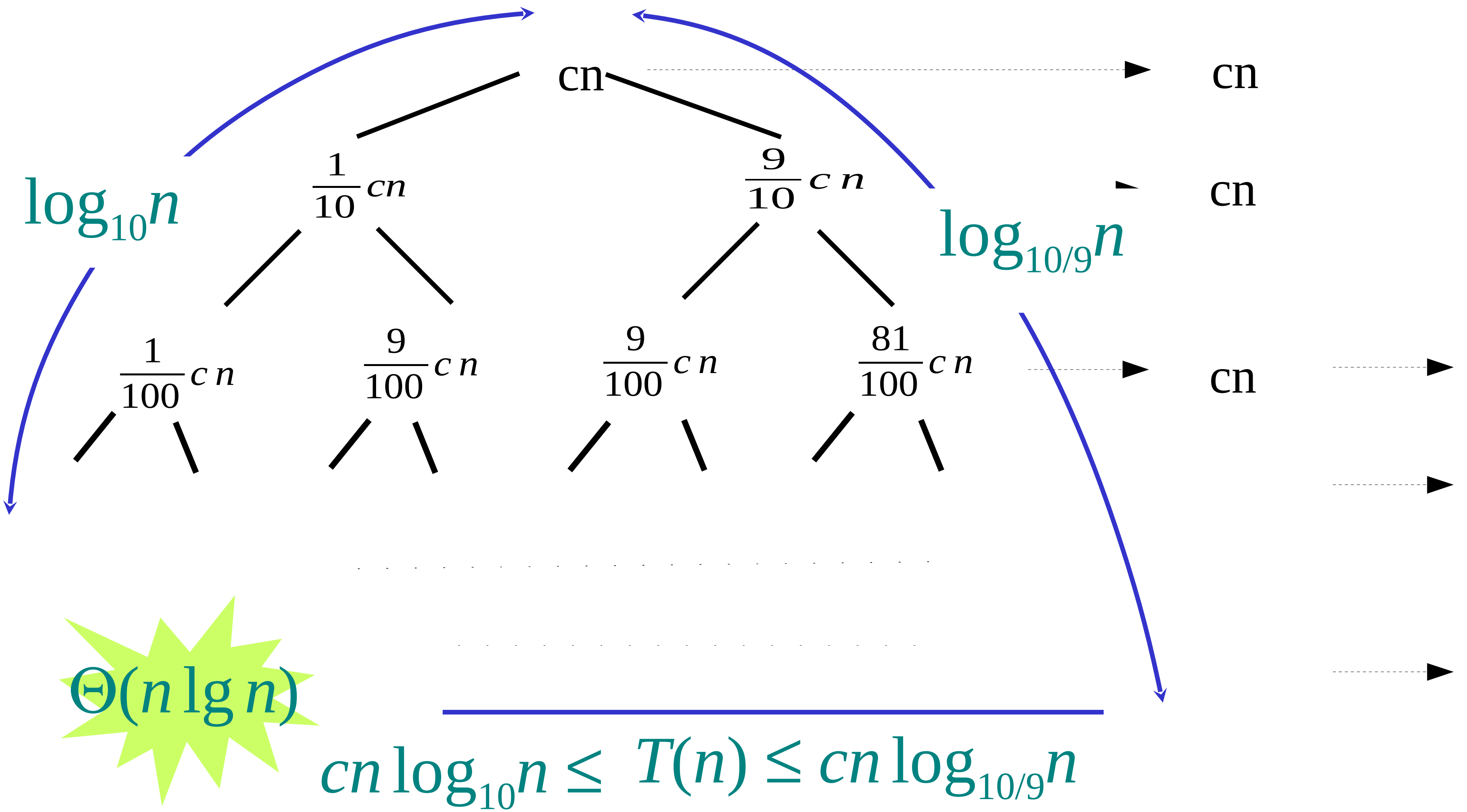
(igual que merge sort)

Que pasa si dividimos en $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T(\tfrac{1}{10}) + T(\tfrac{9}{10}) + \Theta(n)$$

Cual es la solucion de la recurrencia?

Analisis de “casi-mejor” caso



$$cn \log_{10} n \leq T(n) \leq cn \log_{10/9} n$$

Multiplicación de Matrices

- Si tenemos dos matrices A y B cuadradas en donde A tiene el mismo número de filas que columnas de B, se trata de multiplicar A y B para obtener una nueva matriz C.

- La multiplicación de matrices se realiza conforme a

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

- 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 DV, vamos a proceder como con la multiplicació de enteros, con la intención de obtener un algoritmo más eficiente para multiplicar matrices.

Multiplicación de Matrices

La multiplicacion puede hacerse como sigue:

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

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

Multiplicación de Matrices

- Este enfoque da la siguiente recurrencia

$$T(n) = \begin{cases} b & \text{si } n = 1 \\ 8T(n/2) + bn^2 & \text{si } n > 1 \end{cases}$$

- A partir de la cual, es evidente que T(n) sigue siendo O(n^3).
- Pero, basándonos en el enfoque DV que empleamos para multiplicar enteros, la multiplicación de las matrices también puede calcularse como sigue.

Multiplicación de Matrices

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

$$\underset{C}{\Uparrow} \qquad \underset{A}{\Uparrow} \qquad \underset{B}{\Uparrow}$$

El método de Strassen

$$\begin{aligned} P &= (a+d)(e+h) \\ Q &= (c+d)e \\ R &= a(g-h) \\ S &= d(f-e) \\ T &= (a+b)h \\ U &= (c-a)(e+g) \\ V &= (b-d)(f+h) \end{aligned}$$

$$\begin{aligned} r &= P+S-T+V \\ s &= R+T \\ t &= Q+S \\ u &= P+R-Q+U \end{aligned}$$

Multiplicación de Matrices

El método de Strassen

$$
\begin{aligned}
T(n) &= 7T(n/2) + bn^2 \\
&= 7\,(7T(n/4) + b(n/2)^2) + bn^2 \\
&= 7^2\,(7T(n/8) + b(n/4)^2) + (7/4 + 1)bn^2 \\
&= 7^m T(1) + ((7/4)^{m-1} + \ldots + (7/4) + 1)bn^2 \qquad \text{as } n = 2^m \\
&= ((7/4)^m + \ldots + (7/4) + 1)bn^2 \\
&= ((7/4)^m - 1)bn^2 \,/\, ((7/4) - 1) \\
&= O(7^m) \;=\; O(n^{\log 7}) \;=\; O(n^{2.81})
\end{aligned}
$$

¿Y si las matrices no fueran cuadradas ?

Selección de puntos de Parada

- Un camión va desde Granada a Moscú siguiendo una ruta predeterminada. Se asume que conocemos las gasolineras que se pueden encontrar en la ruta.
- La capacidad del tanque es = C. 

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

¿Cómo se aplicaría DyV para resolver este problema?

Prácticas: Multiplicación de Enteros Largos

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

Algoritmo clásico

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)

Mult. Enteros Largos D&V

- Para aplicar D&V debemos de poder obtener la solución en base a problemas de tamaño menor
- Truco:
  - 5632 = 56*100 + 32 y 3427 = 34*100 + 27
  - (56*100 + 32) * (34*100 + 27) =

Se reducen las dos multiplicaciones de 4 cifras a cuatro multiplicaciones de 2 cifras, mas tres sumas y varios desplazamientos

56*32*10000 + (56*27 + 32*34)*100 +(32*27)

Divide y Vencerás básico

Dividir

X=12345678

xi = 1234 xd = 5678

X = xi*10^4 + xd

Y = 24680135

yi = 2468 yd=0135

Y=yi*10^4 + yd

Combinar

$$
\begin{aligned}
X \times Y = & \left[x_i 10^4 + x_d\right] \times \left[y_i 10^4 + y_d\right] \\
= & x_i y_i 10^8 + (x_i y_d + x_d y_i) 10^4 + x_d y_d
\end{aligned}
$$

Mult. Enteros Largos D&V

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

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

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

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

Eficiencia

```
Función DV_basico (X,Y,n) {
    if P es pequeño return X*Y;                         O(1)
    else {
      Obtener xi, xd, yi, yd;                           O(n)
  z1 = DV_basico  (xi,yi,n/2);                          T(n/2)
  z2 = DV_basico  (xi,yd,n/2);                          T(n/2)
  z3 = DV_basico  (xd,yi,n/2);                          T(n/2)
  z4 = DV_basico  (xd,yd,n/2);                          T(n/2)

      aux= Sumar(z2,z3);                                O(n)
      z1   = Desplazar_Dcha(z1,n);                      O(n)
      aux = Desplazar_Dcha(aux,n/2);                    O(n)
      z = Sumar(z1,aux,z4);                             O(n)
      return z;
    }
}
```

Eficiencia del algoritmo DV_bas

- $T(n) = 4T(n/2) + n$

T(n) está en el orden $O(n^2)$

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.

Mult. Enteros Largos D&V

- Considerar

$$r = \underbrace{(xi + xd) * (yi + yd)}_{\textit{sumas de } n/2} = \underbrace{(xi*yi)}_{p} + (xi*yd+xd*yi) + \underbrace{xd*yd}_{q}$$

Luego, podemos calcular

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

1 multiplicación tamaño n --> 3 mult. tamaño n/2

Ahorramos tiempo al operar ?

- Supongamos $X*Y = p*10^{n} + (r-p-q)*10^{n/2} + q$
  - Algoritmo Clásico (AC): h(n) = c n2
  - Sea g(n) operaciones en el algoritmo DV excepto las 3 mutiplicaciones de tamaño n/2.
- Ecuación DV con el AC para tamaño n/2:
  - 3h(n/2) + g(n) = 3c(n/2)2 + g(n) = ¾ cn2 + g(n) = ¾ h(n) + g(n)
- Como h(n) es O(n2) y g(n) es□O(n) → ==> ahorro 25%.
- Ganancia de tiempo – No dismunición de orden
- ¿Cómo resolver los subcasos?

```
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   = Desplazar_Dcha(p,n);
      aux = Desplazar_Dcha(aux,n/2);
      z = Sumar(p,aux,q);
      return z;
    }
}
```

Eficiencia Divide y Vencerás

T(n) = 3T(n/2) + 8n = 3T(n/2)+O(n)

$$T(n) \in O(n^{\log_2 3}) = O(n^{1.585})$$

n	N^2	N^1.585
10	100	38.46
100	10000	1479.11
1000	1000000	56885.29
1000 0	100000000	2187751.62

D&V: Umbrales

Mult. Enteros Largos D&V

Si umbral es igual a 1, entonces

D&V (5.000 cifras) => 41 seg.

Clásico (5.000 cifras) => 25 seg

A partir de 32.789 cifras es mejor D&V (15 minutos !!!)

Si umbral es igual a 64

D&V (5.000 cifras) => 6 seg.

D&V(32.789 cifras) =>2 minutos !!

Selección umbral es problemática:

Depende del algoritmo y de la implementación

Se estima empíricamente.

El Viajante de Comercio

Formulación: **Dado un grafo conexo G, con pesos en las aristas, encontrar el ciclo hamiltoniano (que pasa por todos los vértices, una sóla vez por cada uno) de costo mínimo.**

Ejemplo de aplicación real: taladros de placas de circuitos impresos. Soldar componentes de una placa de circuito impreso, etc.

3. La Determinación del Umbral

Ejemplo: Multiplicación de grandes números.

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

con $h(n) \in \Theta(n^2)$, $g(n) \in \Theta(n)$

¿Cual es el valor óptimo para n_0?

3. La Determinación del Umbral

Ejemplo: Multiplicación de grandes números.

Una implementación concreta $h(n) = n^2$ y $g(n) = 16n$ (ms), y un caso de tamaño $n = 1024$.

Las dos posibilidades extremas nos llevan a

Si $n_0 = 1$, $t(n) = 32$ m y 34sg

Si $n_0 = \infty$, $t(n) = h(n) = 17$ m y 40 sg

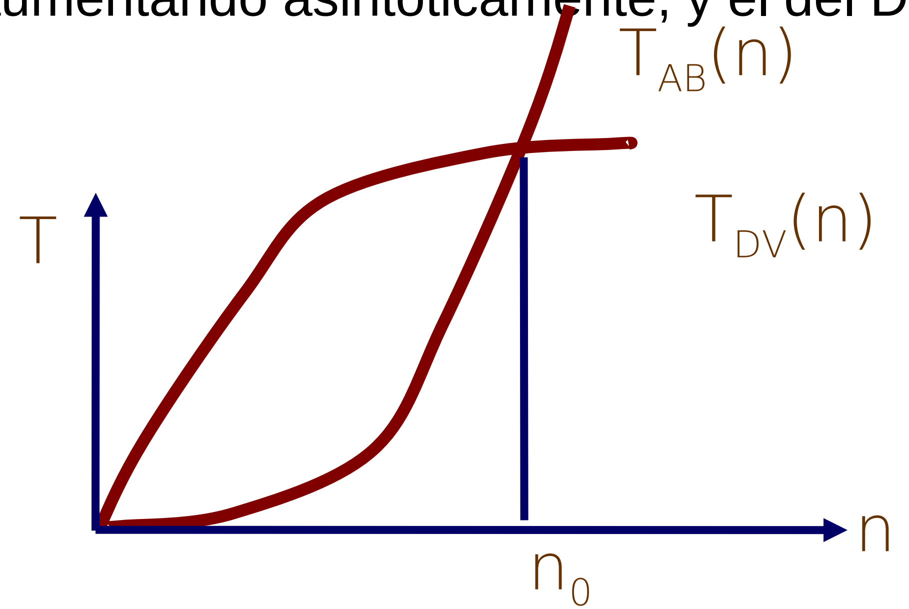
Si puede haber tan grandes diferencias, ¿como podremos determinar el valor óptimo del umbral?

Dos métodos: Experimental y Teórico

3. La Determinación del Umbral

Método experimental

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



3. La Determinación del Umbral

Método teórico

- La idea del enfoque experimental se traduce teoricamente a lo siguiente

  $t(n) \quad = h(n) \quad \text{si } n \leq n_0$

  $= 3\, t\,(n/2) + g(n) \text{ si } n \geq n_0$

- Cuando coinciden los tiempos de los dos algoritmos

$$h(n) = t(n) = 3\, h(n/2) + g(n);\ n = n_0$$

- Para una implementación concreta (por ejemplo, la anterior, $h(n) = n^2$ y $g(n) = 16n$ (ms) y $n = 1024$)

$$n^2 = \tfrac{3}{4}\, n^2 + 16\, n \rightarrow n = \tfrac{3}{4}\, n + 16$$

$$\mathbf{n_0 = 64}$$

3. La Determinación del Umbral

Método híbrido

- Calculamos las constantes ocultas utilizando un enfoque empírico.
- Calculamos el umbral, utilizando el criterio seguido para el umbral teórico.
- Probamos valores alrededor del umbral teórico (umbrales de tanteo) para determinar el umbral óptimo.
- Inconveniente: las constantes ocultas (son poco importantes con n grandes)

El Viajante de Comercio

(Descomponer)

Dividir el rectángulo que contiene todos los nodos horizontal o verticalmente de forma que:

i) la línea divisoria es paralela al lado más corto,

ii) la línea divisora pasa por un punto, incluido en ambos subcasos, elegida de forma tal que ambos contienen un mismo número de puntos.

Resolver los casos

(Combinar)

Formar un ciclo a partir de dos subciclos que se encuentra exactamente en un punto

Teoría de Algoritmos

Tema 1. Planteamiento General

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos "Divide y Vencerás"

Tema 4. Algoritmos Voraces ("Greedy")

Tema 5. Algoritmos para la Exploración de Grafos ("Backtraking", "Branch and Bound")

Tema 6. Algoritmos basados en Programación Dinámica

Tema 7. Otras Técnicas Algorítmicas de Resolución de Problemas