Algorítmica

Tema 3: Algoritmos Divide y Venceras

Bibibliografía:

G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997).

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

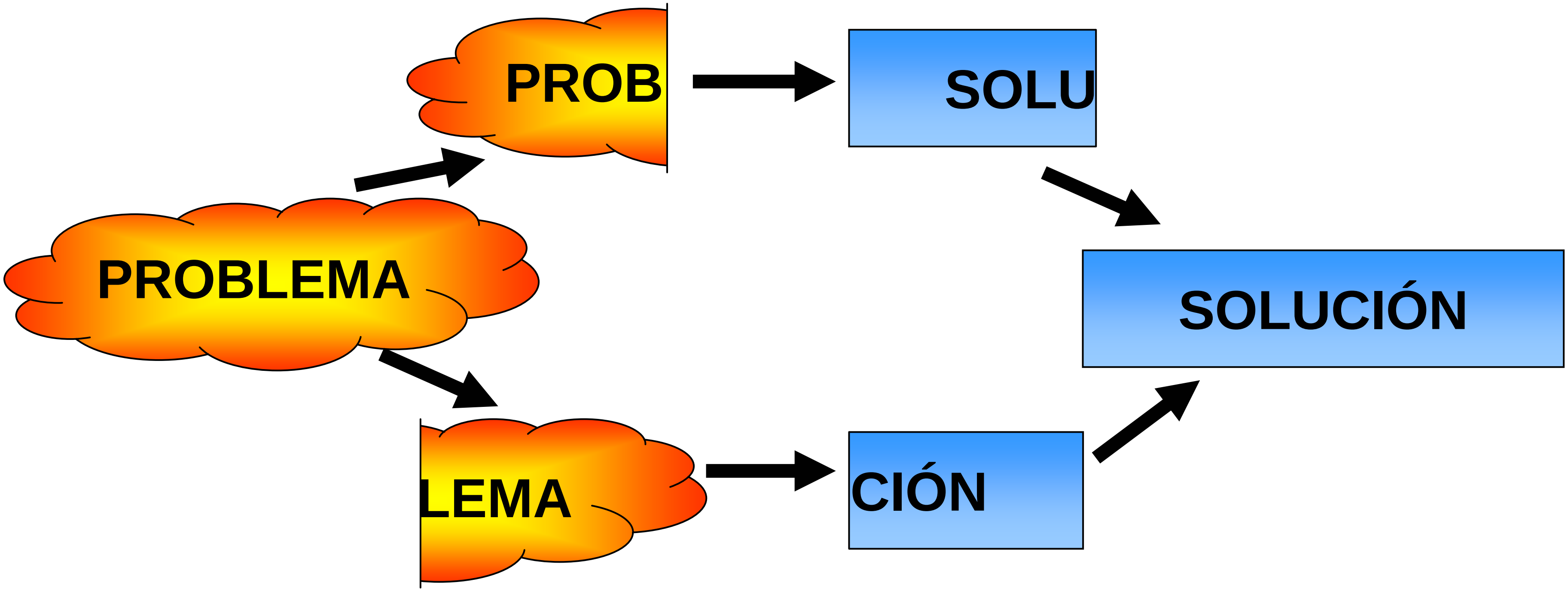
Indice

EL ENFOQUE DIVIDE Y VENCERÁS

APLICACIONES DE LA TÉCNICA DIVIDE Y VENCERÁS

Intuitivamente...

- La técnica **divide y vencerás** consiste en:
  - Descomponer un problema en un conjunto de subproblemas más pequeños.
  - Se resuelven estos subproblemas.
  - Se combinan las soluciones para obtener la solución para el problema original.



Introducción

- **Esquema general:**

  DivideVencerás (p: problema)

  Dividir (p, p_1, p_2, ..., p_k)

  para i:= 1, 2, ..., k

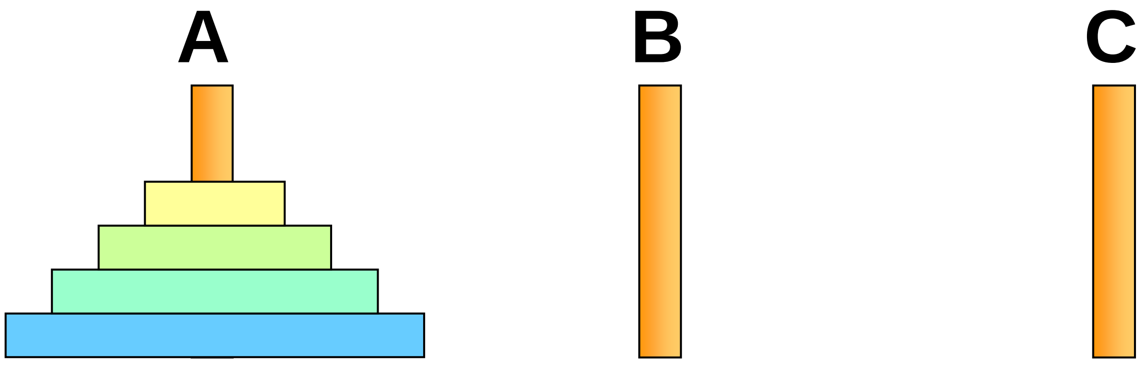
  s_i:= *Resolver* (p_i)

  solución:= *Combinar* (s_1, s_2, ..., s_k)

- Normalmente para resolver los subproblemas se utilizan llamadas recursivas al mismo algoritmo (aunque no necesariamente).

- **Ejemplo.** Problema de las Torres de Hanoi.

Introducción



- **Ejemplo.** Problema de las torres de Hanoi. Mover **n** discos del poste A al C:
  - Mover **n-1** discos de A a B
  - Mover 1 disco de A a C
  - Mover **n-1** discos de B a C

Introducción

Hanoi (n, A, B, C: entero)

```
si  n==1 entonces
     mover (A, C)
sino
     Hanoi (n-1, A, C, B)
     mover (A, C)
     Hanoi (n-1, B, A, C)
finsi
```

- Si el problema es “pequeño”, entonces se puede resolver de forma directa.
- **Otro ejemplo.** Cálculo de los números de Fibonacci: $F(n) = F(n-1) + F(n-2)$
- $F(0) = F(1) = 1$

Introducción

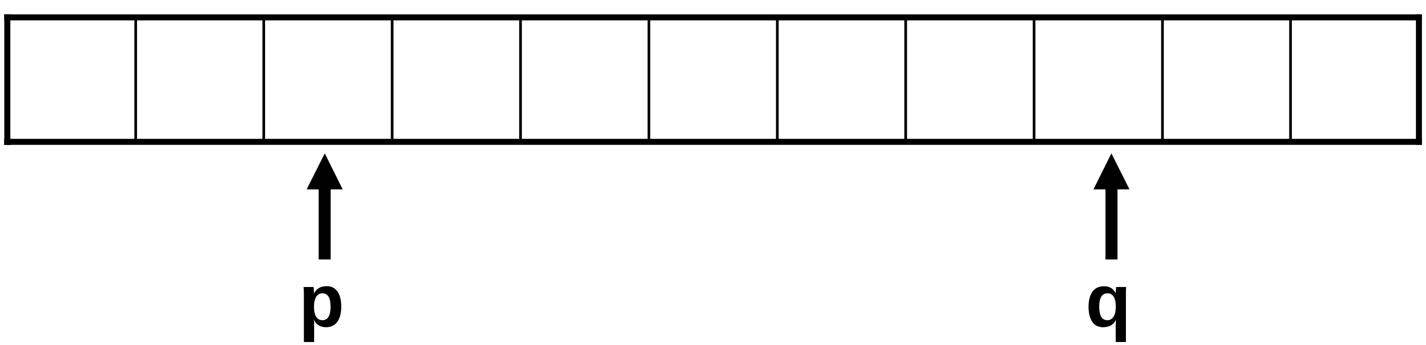
- **Ejemplo.** Cálculo de los números de Fibonacci.
  - El cálculo del n-ésimo número de Fibonacci se descompone en calcular los números de Fibonacci n-1 y n-2.
  - *Combinar*: sumar los resultados de los subproblemas.

- La idea de la técnica divide y vencerás es aplicada en muchos campos:
  - Estrategias militares.
  - Demostraciones lógicas y matemáticas.
  - Diseño modular de programas.
  - Diseño de circuitos.
  - Etc.

Introducción

- **Esquema recursivo.** Con división en 2 subproblemas y datos almacenados en una tabla entre las posiciones p y q:

```
DivideVencerás (p, q: indice)
var m: indice
    si Pequeño (p, q) entonces
        solucion:= SoluciónDirecta (p, q)
    sino
        m:= Dividir (p, q)
        solucion:= Combinar (DivideVencerás (p, m),
                             DivideVencerás (m+1, q))
    finsi
```



Introducción

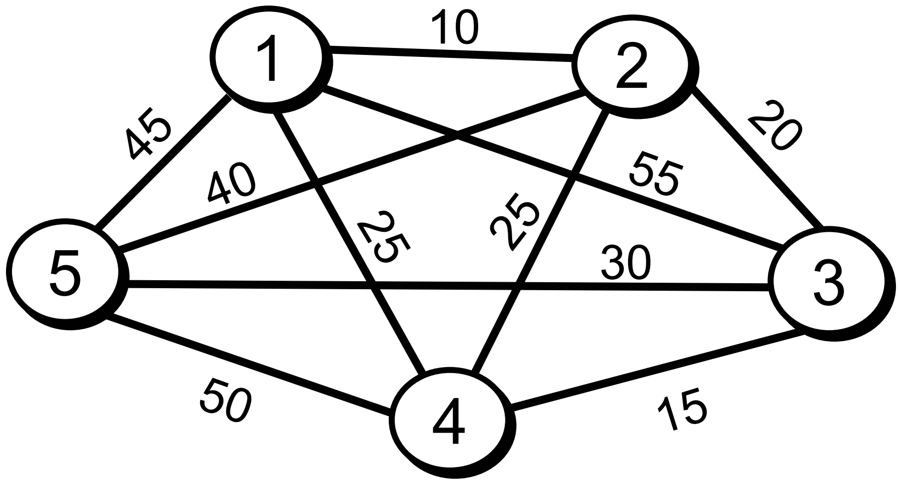
- Aplicación de divide y vencerás: encontrar la forma de definir las **funciones genéricas**:
  - *Pequeño*: Determina cuándo el problema es pequeño para aplicar la resolución directa.
  - *SoluciónDirecta*: Método alternativo de resolución para tamaños pequeños.
  - *Dividir*: Función para descomponer un problema grande en subproblemas.
  - *Combinar*: Método para obtener la solución al problema original, a partir de las soluciones de los subproblemas.
- Para que pueda aplicarse la técnica divide y vencerás debe existir una forma de definirlas → Aplicar un razonamiento inductivo...

Introducción

- Requisitos para aplicar divide y vencerás:
  - Necesitamos un **método** (más o menos **directo**) de resolver los problemas de tamaño pequeño.
  - El problema original debe poder dividirse fácilmente en un conjunto de subproblemas, del **mismo tipo** que el problema original pero con una resolución **más sencilla** (menos costosa).
  - Los subproblemas deben ser **disjuntos**: la solución de un subproblema debe obtenerse independientemente de los otros.
  - Es necesario tener un método de **combinar** los resultados de los subproblemas.

Introducción

- **Ejemplo.** Problema del viajante.



- Método directo de resolver el problema:
  Trivial con 3 nodos.
- Descomponer el problema en subproblemas más pequeños:
  ¿Por dónde?
- Los subproblemas deben ser disjuntos:
  ...parece que no
- Combinar los resultados de los subproblemas:
  ¡¡Imposible aplicar divide y vencerás!!

Introducción

- Normalmente los subproblemas deben ser de tamaños parecidos.
- Como mínimo necesitamos que hayan dos subproblemas.
- Si sólo tenemos un subproblema entonces hablamos de técnicas de **reducción** (o **simplificación**).

- **Ejemplo sencillo:** Cálculo del factorial.
  Fact(n):= n*Fact(n-1)

Introducción

- Para el esquema recursivo, con división en dos subproblemas con la mitad de tamaño:

$$t(n) = \begin{cases} g(n) & \text{Si } n \leq n_0 \text{ (caso base)} \\ 2*t(n/2) + f(n) & \text{En otro caso} \end{cases}$$

- **t(n)**: tiempo de ejecución del algoritmo DV.
- **g(n)**: tiempo de calcular la solución para el caso base, algoritmo directo.
- **f(n)**: tiempo de dividir el problema y combinar los resultados.

Introducción

- **Ejemplo 1.** La resolución directa se puede hacer en un tiempo constante y la división y combinación de resultados también.

  $g(n) = c; \qquad f(n) = d$

  $\Rightarrow \; t(n) \in \Theta(n)$

- **Ejemplo 2.** La solución directa se calcula en $O(n^2)$ y la combinación en $O(n)$.

  $g(n) = c \cdot n^2; \quad f(n) = d \cdot n$

  $\Rightarrow \; t(n) \in \Theta(n \log n)$

Introducción

- En general, si se realizan **a** llamadas recursivas de tamaño **n/b**, y la división y combinación requieren $f(n) = d \cdot n^p \in O(n^p)$, entonces:

  $$t(n) = a \cdot t(n/b) + d \cdot n^p$$

Suponiendo $n = b^k \Rightarrow k = \log_b n$

$$t(b^k) = a \cdot t(b^{k-1}) + d \cdot b^{pk}$$

Podemos deducir que:

$$t(n) \in \left\{ \begin{array}{ll} O(n^{\log_b a}) & \text{Si } a > b^p \\ O(n^p \cdot \log n) & \text{Si } a = b^p \\ O(n^p) & \text{Si } a < b^p \end{array} \right\}$$

Fórmula maestra

±

Introducción

- **Ejemplo 3.** Dividimos en 2 trozos de tamaño n/2, con $f(n) \in O(n)$:

  $a = b = 2$

  $t(n) \in O(n \cdot \log n)$

- **Ejemplo 4.** Realizamos 4 llamadas recursivas con trozos de tamaño n/2, con $f(n) \in O(n)$:

  $a = 4;\ b = 2$

  $t(n) \in O(n^{\log_2 4}) = O(n^2)$

1. El Enfoque Divide y Venceras

Resumen de la Introducción:

Por tanto...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?

Ejemplo: Multiplicación de enteros muy grandes

Algoritmo clásico de multiplicación de enteros de n cifras: $\Theta(n^2)$

Algoritmo basado en la división de enteros de tamaño n:

981 1234

0981 1234 (n = 4)

w = 09, x = 81 y = 12, z = 34 (n = 2)

$981 = 10^2 w + x$ $1234 = 10^2 y + z$

Ejemplo: Multiplicación de enteros muy grandes

$981 = 10^2w + x$ $\qquad$ $1234 = 10^2y + z$

$981 \times 1234 = (10^2w + x) \times (10^2y + z)$

$= 10^4wy + 10^2(wz + xy) + xz$

$= 1.080.000 + 127.800 + 2.754$

$= 1.210.554$

Se precisan 4 operaciones productos de tamaño n/2: wy,wz, xy, xz

Ejemplo: Multiplicación de enteros muy grandes

¿Es posible reducir el número multiplicaciones de tamaño n/2?

Consideremos:

$$r = (w + x) \times (y + z) = wy + (wz + xy) + xz$$
$$= (09 + 81) \times (12 + 34) = 90 \times 46 = 4140$$
$$p = wy = 09 \times 12 = 108$$
$$q = xz = 81 \times 34 = 2754$$

Finalmente:

$$981 \times 1234 = 10^4 p + 10^2 (r - p - q) + q =$$
$$= 1.080.000 + 127.800 + 2.754 = 1.210.554$$

1 multiplicación de tamaño n $\rightarrow$ 3 de tamaño n/2 más operaciones sumas y restas.

Reducción de 4 multiplicacones de tamaño n/2 a 3 mutiplicaciones de tamaño n/2: 25%

Ejemplo: Multiplicación de enteros muy grandes

Ecuaciones de tiempo:

Algoritmo básico (AB): $h(n) = cn^2$

Consideremos $g(n)$ operaciones en el algoritmo DV excepto las 3 mutiplicaciones de tamaño $n/2$.

Ecuación DV con el Algoritmo básico (AB) para tamaño $n/2$:

$$3h(n/2) + g(n) = 3c(n/2)^2 + g(n) = \tfrac{3}{4} cn^2 + g(n) = \tfrac{3}{4} h(n) + g(n)$$

$h(n) \in \Theta(n^2)$, $g(n) \in \Theta(n)$ → ganancia aproximada 25%.

- Ganancia de tiempo
- ¿Cómo resolver los subcasos?

Ejemplo: Multiplicación de enteros muy grandes

¿Qué ocurre si resolvemos los subcasos de forma recursiva?

$t(n) = 3t(n/2) + g(n),\ g(n) \in \Theta(n)$

$t(n) \in \Theta(n^{\log 3}),\ t(n) \in \Theta(n^{1.585}),$

Algunos estudios pendientes:

- ¿Cómo se tratan los números de longitud impar?
- ¿Cómo multiplicamos dos números de diferente tamaño?

Brassard-Bratley, 1997.

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, ..., 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 el 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

Para l subcasos con tamaño n/b

$$t(n) = a\ t(n/b) + g(n)$$

si $g(n) \in \Theta(n^k)$, entonces $t(n)$ es de orden:

$\Theta(n^k)$ si $a < b^k$

$\Theta(n^k \log n)$ si $a = b^k$

$\Theta(n^{\log_b a})$ si $a > b^k$

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.
- El umbral no es único, pero si lo es en cada 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.
  - 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.

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 $g(n) = 16n$ (μs), y un caso de tamaño $n = 5000$.

Las dos posibilidades extremas nos llevan a

Si $n_0 = 1$, $t(n) = 41$ sg

Si $n_0 = \infty$, $t(n) = h(n) = 25$ sg

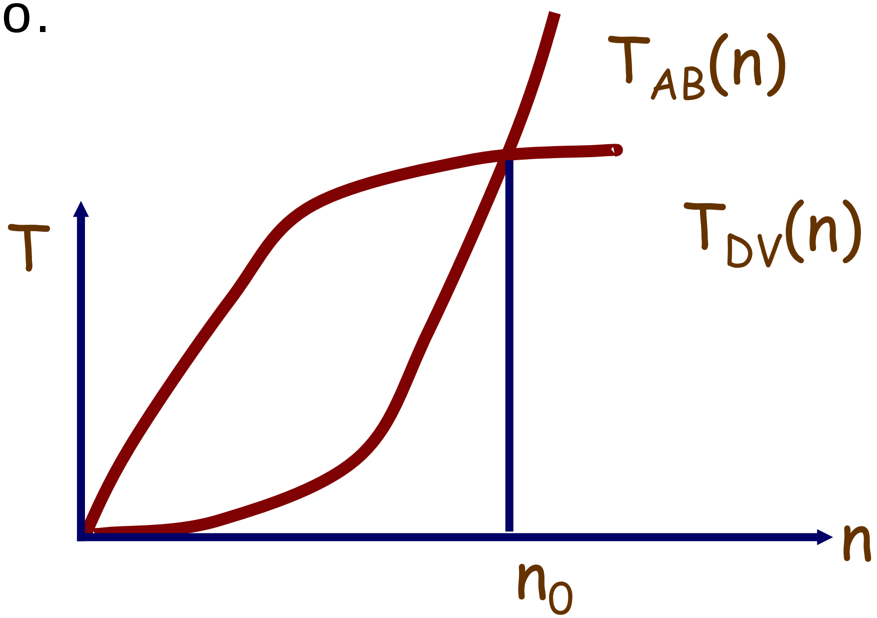
Si puede haber tan grandes diferencias, **¿como podremos determinar el valor óptimo del umbral?**

Tres métodos: **Experimental, Teórico e Híbrido**

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.



¡MUY COSTOSO EN TIEMPO Y RECURSOS!

3. La Determinación del Umbral

Método teórico

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

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

- Cuando coinciden los tiempos de los dos algoritmos

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

Método híbrido

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

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

$$\mathbf{n_0 = 64}$$

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)

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.
- Supondremos que todos los registros tienen una función ***clave ()*** que devuelve la clave.
- También supondremos que está definida una función ***swap ()*** que intercambia la posición de dos registros cualesquiera.

Algoritmos de Ordenación

- **El problema de la 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 $k_{s1} \leq k_{s2} \leq \ldots \leq k_{sn}$.

  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.

- Se denomina **estable** el algoritmo de ordenación que *mantiene el orden relativo en que ocurren los registros con clave repetida en la 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 de Shell (Shellsort)
  - Ordenación rápida (Quicksort)
  - Ordenación por fusión (Mergesort)
  - Ordenación por montículo (Heapsort)
  - × son algoritmos más complejos
  - ✓ se comportan muy bien cuando la entrada es muy grande.

Algoritmos de Ordenación

Ordenación por inserción I

- La ordenación por inserción procesa secuencialmente la lista de registros.
- El algoritmo mantiene una lista ordenada con aquellos registros ya procesados.
- Cada registro se inserta en su posición correcta dentro de la lista ordenada cuando le toca.

Algoritmos de Ordenación

Ordenación por inserción II

```
public void insercion (Elemento [] vector) {
  // método de ordenación por inserción
  int i, j;
  for (i = 1; i < vector.length; i++)
   for (j = i; (j > 0) && (vector[j].clave () < vector[j –1].clave ()); j--)
    swap (vector, j, j – 1);
}

public void swap (Elemento [] vector, int i, int j) {
  // método que intercambia dos posiciones del vector
  Elemento aux;

  aux = vector [i];
  vector [i] = vector [j];
  vector [j] = aux;
}
```

Algoritmos de Ordenación

Ordenación por inserción III

pasada 0: 42 20 17 13 28 14 23 15

pasada 1: 20 42 17 13 28 14 23 15

pasada 2: 17 20 42 13 28 14 23 15

pasada 3: 13 17 20 42 28 14 23 15

pasada 4: 13 17 20 28 42 14 23 15

pasada 5: 13 14 17 20 28 42 23 15

pasada 6: 13 14 17 20 23 28 42 15

pasada 7: 13 14 15 17 20 23 28 42

Algoritmos de Ordenación

Ordenación por inserción IV

Análisis del algoritmo

- El cuerpo del algoritmo está formado por dos bucles *for* anidados.
- El bucle *for* externo se ejecuta *n – 1* veces.
- El bucle interno depende del número de claves en la parte ordenada que son menores (o mayores) que la clave a la que buscamos acomodo.

Algoritmos de Ordenación

Ordenación por inserción IV

- En el peor caso, cada registro se debe mover hasta el principio del vector (*i comparaciones* por pasada).

  En el peor caso, el coste será:

$$\sum_{i=1}^{n} i = \Theta(n^2)$$

- En el mejor caso, las claves estarán ordenadas de menor a mayor y sólo habrá que realizar 1 comparación por pasada.

$$\sum_{i=1}^{n} 1 = \Theta(n)$$

Algoritmos de Ordenación

Ordenación por inserción IV

- El *mejor caso* es significativamente más rápido que el *peor caso*.
- El **peor caso** suele ser una **indicación de mayor confianza** del tiempo "típico" que tarda el proceso en realizarse.
- Hay situaciones en la que es muy posible que se comporte como en el mejor caso:
  - e.g.: una lista ordenada se desordena ligeramente.
- Algunos algoritmos *aprovechan este mejor caso* del algoritmo de ordenación por inserción para mejorar su rendimiento:
  - Shellsort
  - Quicksort

  cuando van a ordenar listas pequeñas (9 elementos o menos) utilizan inserción.

Algoritmos de Ordenación

Ordenación por inserción IV

- ¿Cuál es el **coste del caso medio**?
- Cuando se ordena el registro *i-ésimo*, el número de iteraciones en el bucle interno depende de lo "desordenado" que estuviera este registro.
- Se darán tantas pasadas al bucle interno como registros haya con clave mayor que la del registro *i-ésimo*.
- Cada vez que se intercambian dos posiciones se denomina **inversión**.
- Contar el número de inversiones determina el número de comparaciones e intercambios a realizar.
- Supongamos que, en media, la mitad de las primeras *i-1* claves tienen clave mayor que la del registro *i-ésimo*.
- El caso medio debería tener cerca de la mitad del coste del peor caso, $\Theta(n^2)$.

Algoritmos de Ordenación

Ordenación por burbuja

Es de los métodos más simples.

Idea: Los elementos más ligeros ascienden.

```
void burbuja(int T[], int tam)
    {
    int i, j;
    int aux;
    for (i = 0; i < tam - 1; i++)
    for (j = tam - 1; j > i; j--)
    if (T[j] < T[j-1]) {
    aux = T[i];
    T[i] = T[j];
    T[j] = aux;
    }}
```

Algoritmos de Ordenación

Algoritmo por montículos (heapsort)

Se basa en la simulación de la inserción y borrado en un árbol parcialmente ordenado, simulado sobre un vector:

```
template <typename tipo>
void heapsort(vector<tipo> &T, int inicial, int final)
{
  APO<tipo> a;
  for (int i= inicial; i < final; i++)
    a.insertar(T[i]);
  for (i = inicial; i< final; i++)
    T[i] = a.BorrarMinimo();
}
```

Algoritmos de Ordenación

```
template <typename tipo>
void heapsort(vector<tipo> &T)
{
  int N = T.size()-1;
  for (int i= N/2 -1; i>= 0; i--)
    reajustar(T, N, i);
  for (i= N-1; i>=1; i--) {
    swap(T[0], T[i]);
    reajustar(T, i, 0);
  }
}
```

donde **reajustar(T, n, j)** coloca el elemento **j** en la posición que le corresponde viendo el vector **T** como el recorrido por niveles de un APO con **n** elementos

Algoritmos de Ordenación

El esquema general de ordenación

Divide y Vencerás es el siguiente

Algoritmo de Ordenacion con Divide y Vencerás

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

Algoritmos de 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.

J_li&!t=¬g i b[]_l f[!j [lnd c¬h9

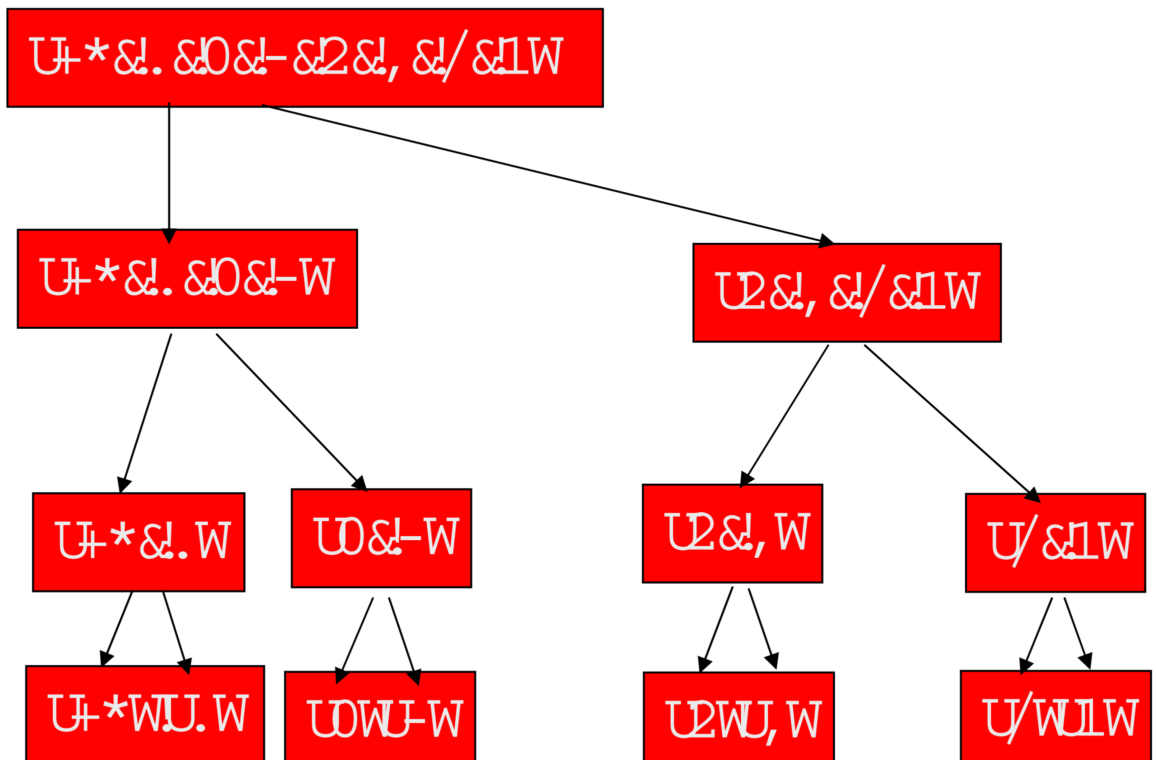
Algoritmos de Ordenación

Ordenación por mezcla

- Buscamos hacer una partición equilibrada de la lista en dos partes A y B
- En A habrá n/k elementos, y en B el resto
- Ordenamos entonces A y B recursivamente
- Combinamos las listas ordenadas A y B usando un procedimiento llamado ***mezcla***, que combina las dos listas en una sola
- Las diferentes posibilidades nos las va a dar el valor k que escojamos

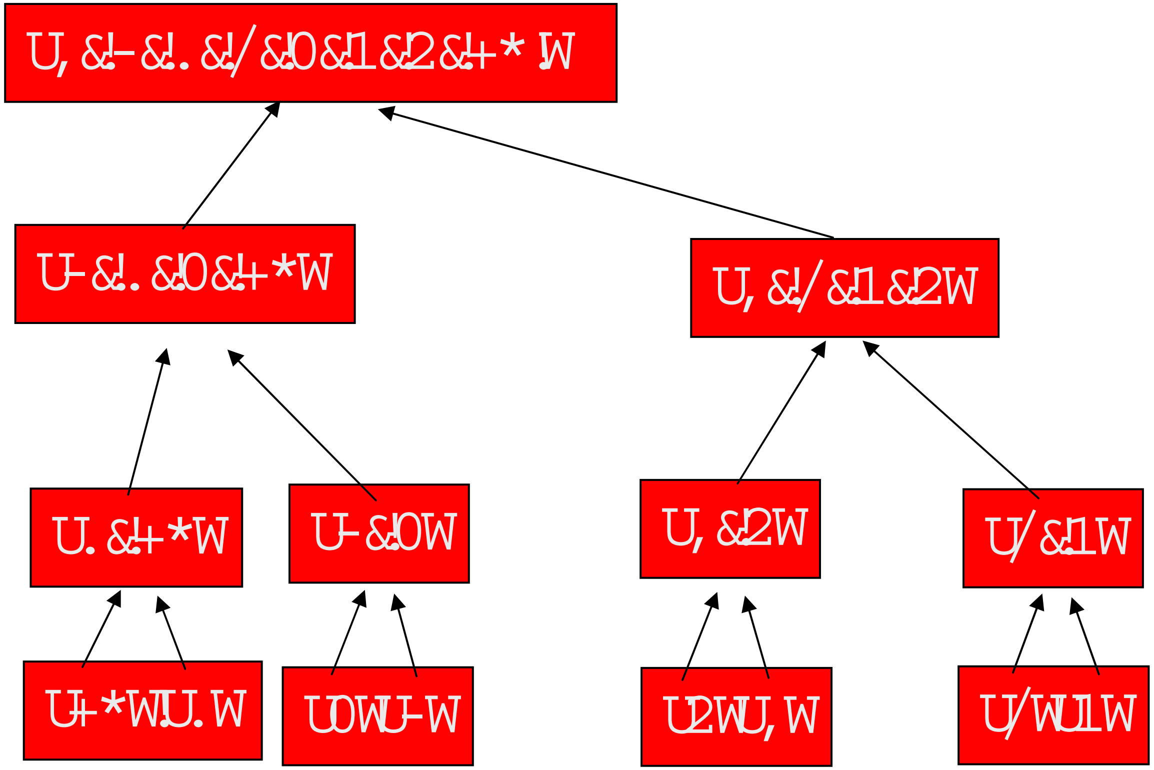
Algoritmos de Ordenación

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(Comparable []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); // combina de “a” a
   “b”
        copy(b, a, left, right);  //copy el resultado en a
   }
}
```

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 $O(n\log n)$

Algoritmos de Ordenación

Quicksort

- Es el algoritmo (general) de ordenación mas eficiente
  - 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

Algoritmos de Ordenación

Pseudo Codigo para quicksort

Algoritmo QUICKSORT(S,T)

IF TAMAÑO(S) $\leq$ q (umbral) THEN INSERCION(S,T)
ELSE

Elegir cualquier elemento p del array como pivote

Partir S en (S1,S2,S3) de modo que
1. $\forall$ x$\in$S1, y$\in$S2, z$\in$S3 se verifique x<p<z e y=p
2. TAMAÑO(S1) < TAMAÑO(S) y TAMAÑO(S3) < TAMAÑO(S)
QUICKSORT(S1,T1) // ordena recursivamente particion izquierda
QUICKSORT(S3,T3) // ordena recursivamente particion derecha
Combinacion: T = T1 || S2 || T3 //S2 es el elemento intermedio entre cada mitad ordenada

End Algoritmo

Algoritmos de Ordenación

Quicksort: 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
  - Podria ser la media de los elementos seleccionados en S
  - Podria elegirse aleatoriamente, pero la funcion RAND() consume tiempo, que habria que añadirselo al tiempo total del algoritmo
- Pivotes usuales son la mediana de un minimo de tres elementos, o el elemento medio de S.

Algoritmos de 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

Algoritmos de Ordenación

Quicksort: Ejemplo escogiendo el elemento medio

array:

5	89	35	10	24	15	37	13	20	17	70

tamaño: 11

Con este ejemplo vamos a ilustrar su funcionamiento

Algoritmos de Ordenación

Quicksort: Ejemplo

array:

5	89	35	14	24	15	37	13	20	7	70

“elemento pivote”

Algoritmos de Ordenación

Quicksort: Ejemplo

array:

5	89	35	14	24	15	37	13	20	7	70

partición:

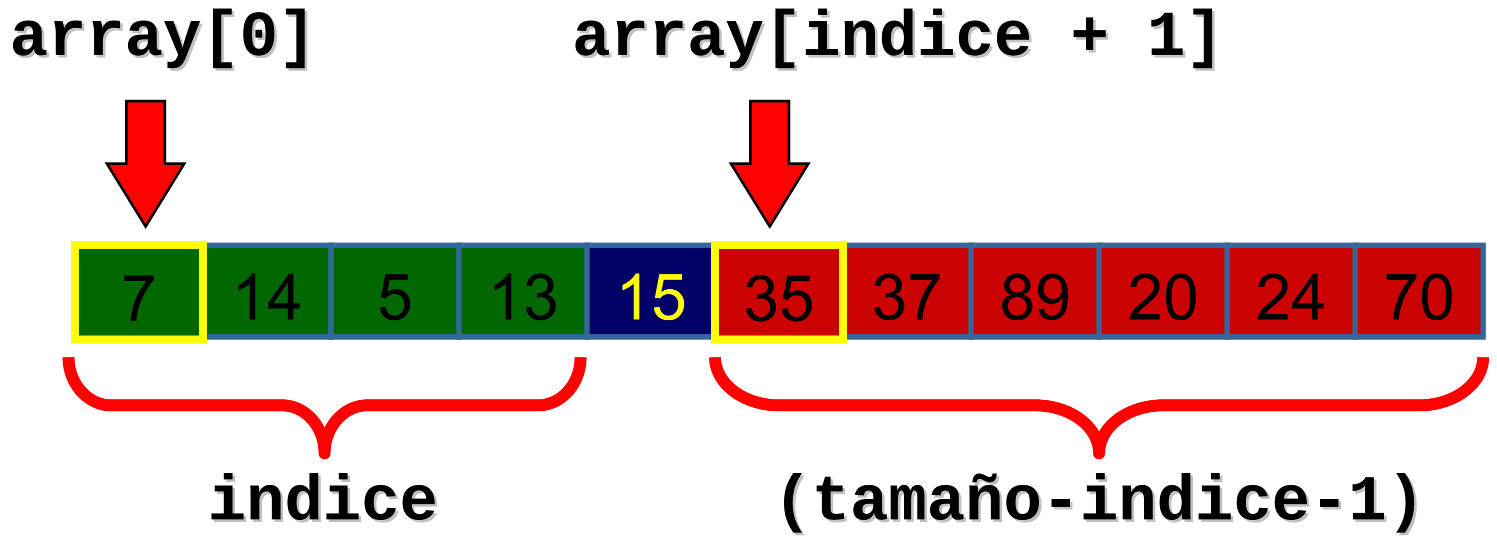
5	89	35	14	24	15	37	13	20	7	70

7	14	5	13	15	35	37	89	20	24	70

indice: 4

Algoritmos de Ordenación

Quicksort: Ejemplo



Algoritmos de Ordenación

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

Algoritmos de Ordenación

0

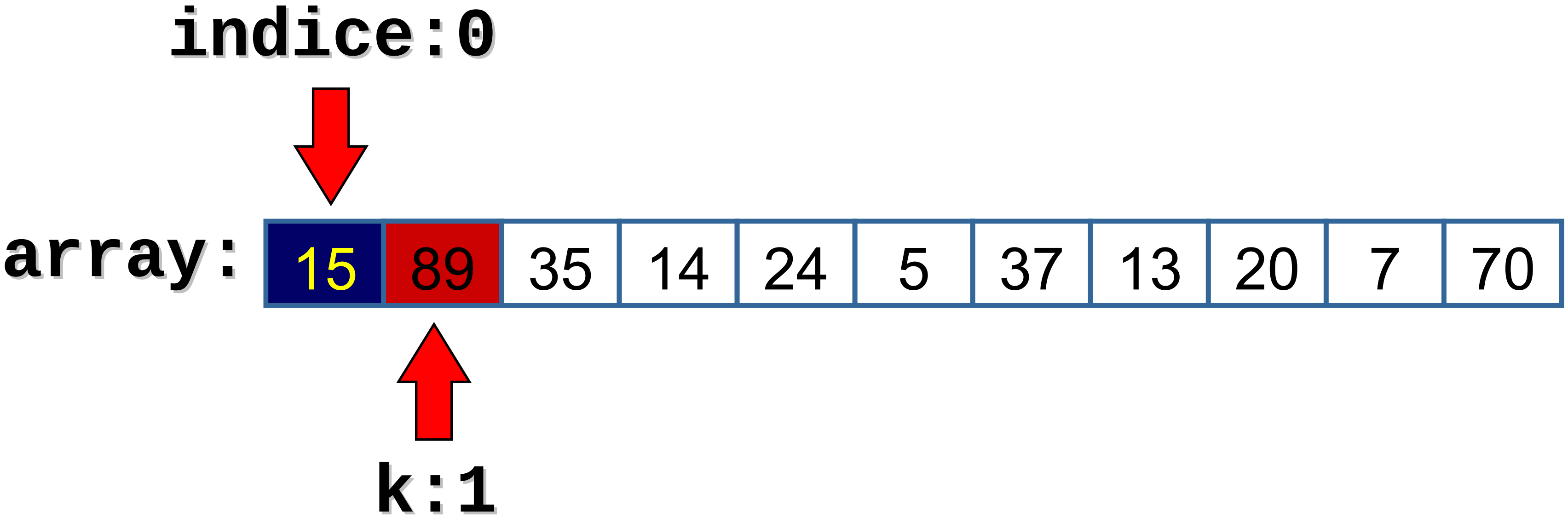
array:

15	89	35	14	24	5	37	13	20	7	70

k=1

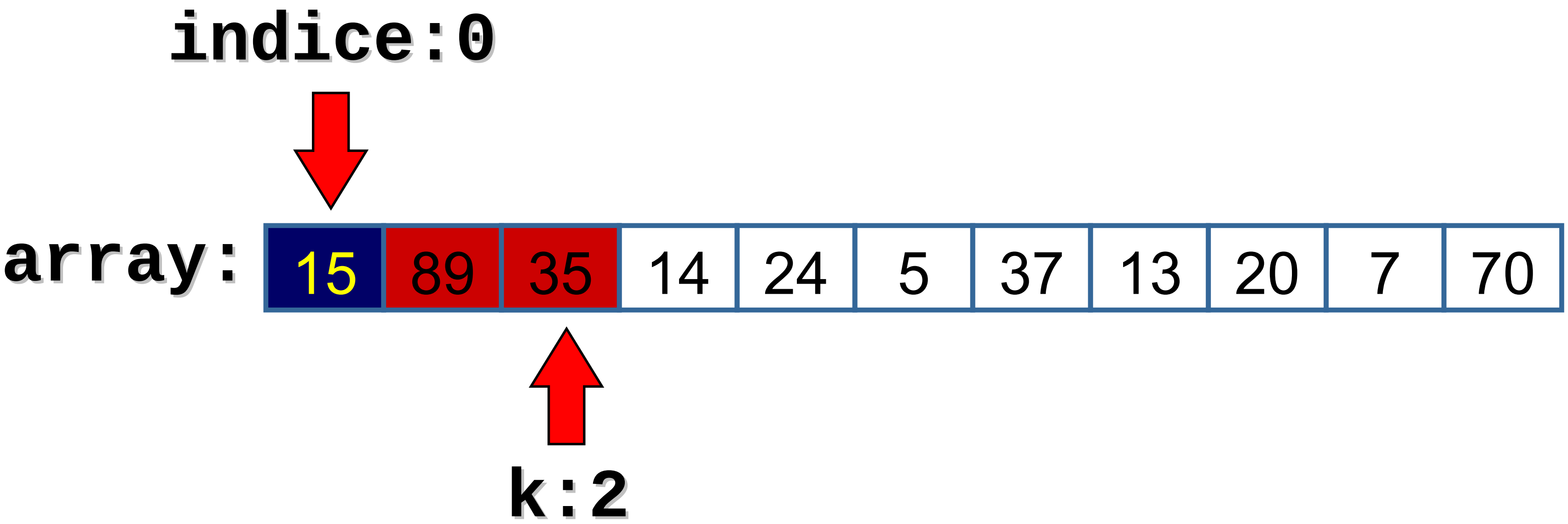
Quicksort: Ejemplo

Algoritmos de Ordenación



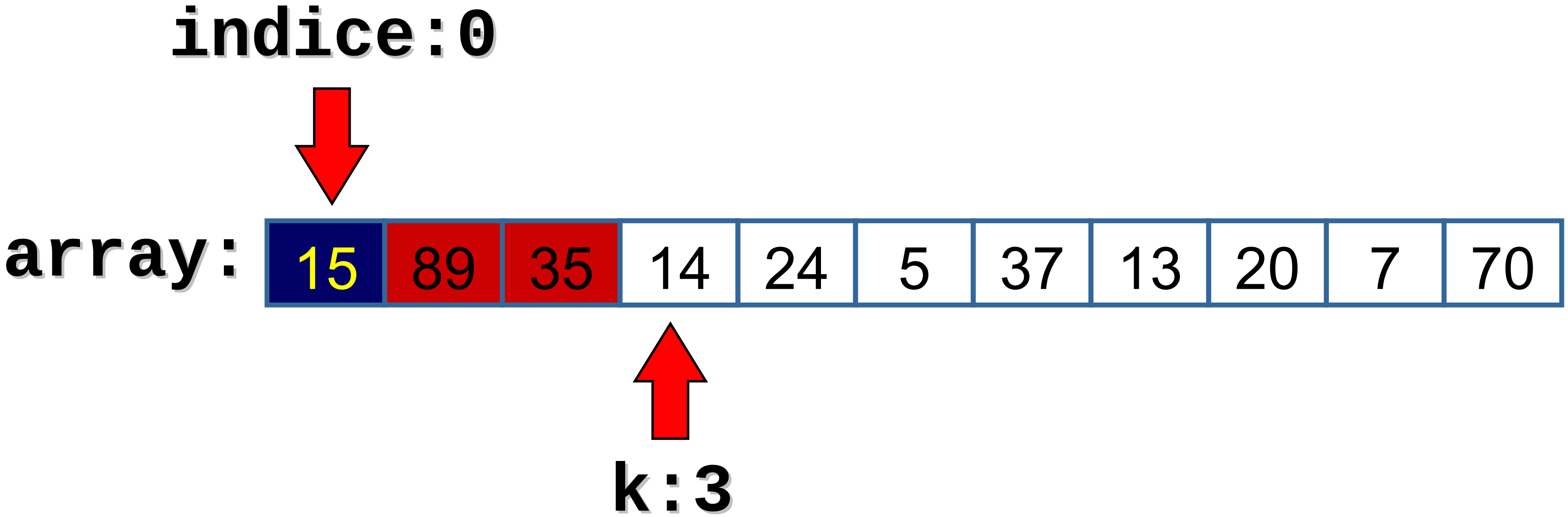
Quicksort: Ejemplo

Algoritmos de Ordenación



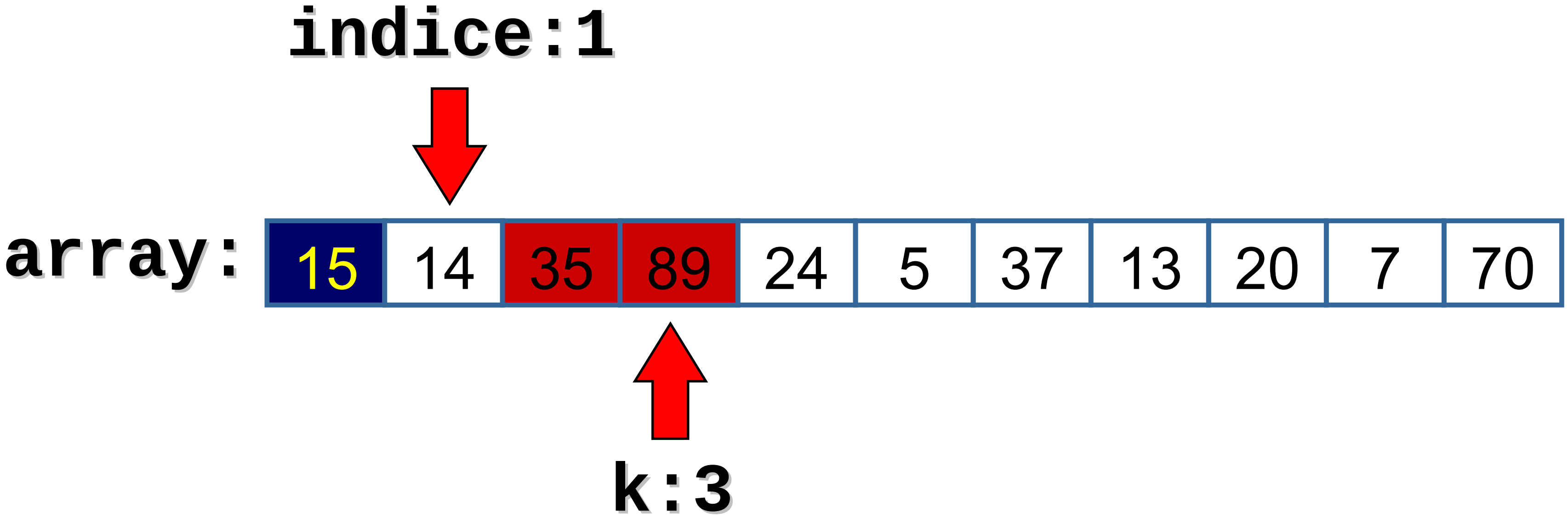
Quicksort: Ejemplo

Algoritmos de Ordenación



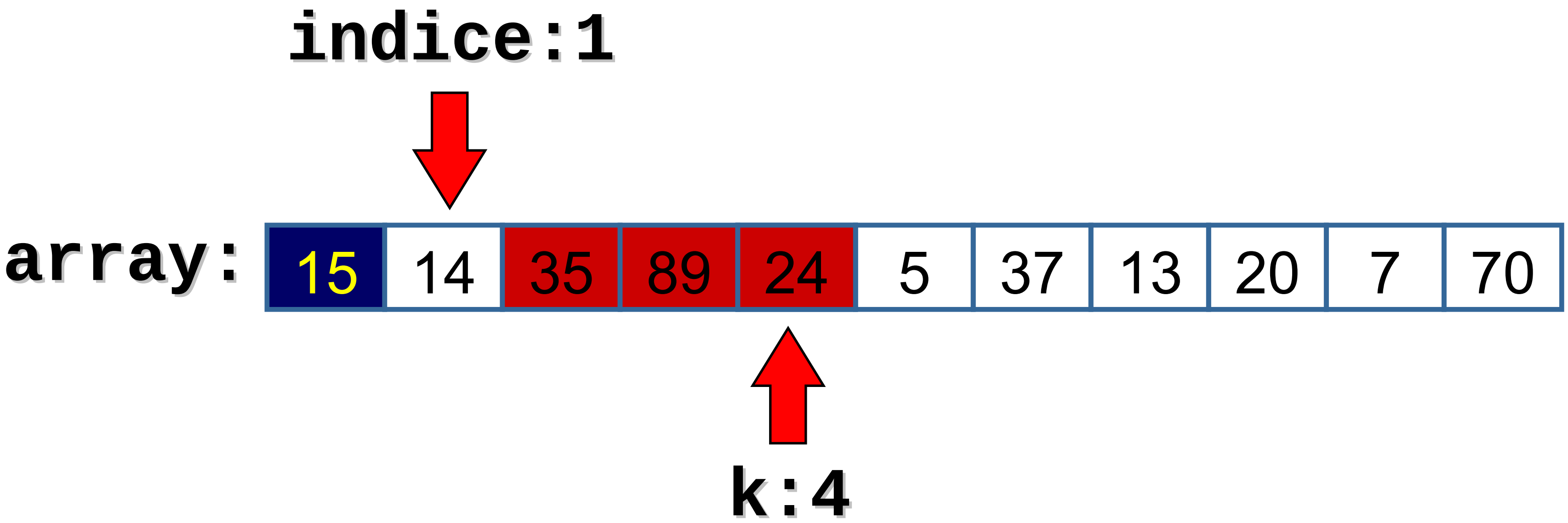
Quicksort: Ejemplo

Algoritmos de Ordenación



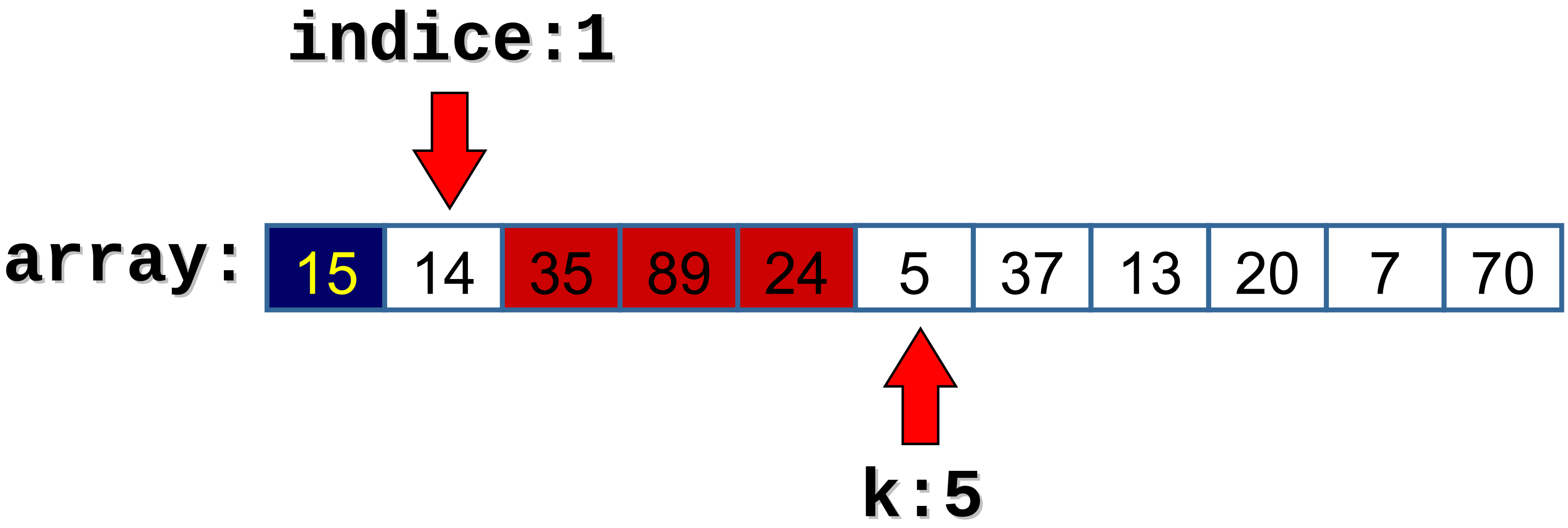
Quicksort: Ejemplo

Algoritmos de Ordenación



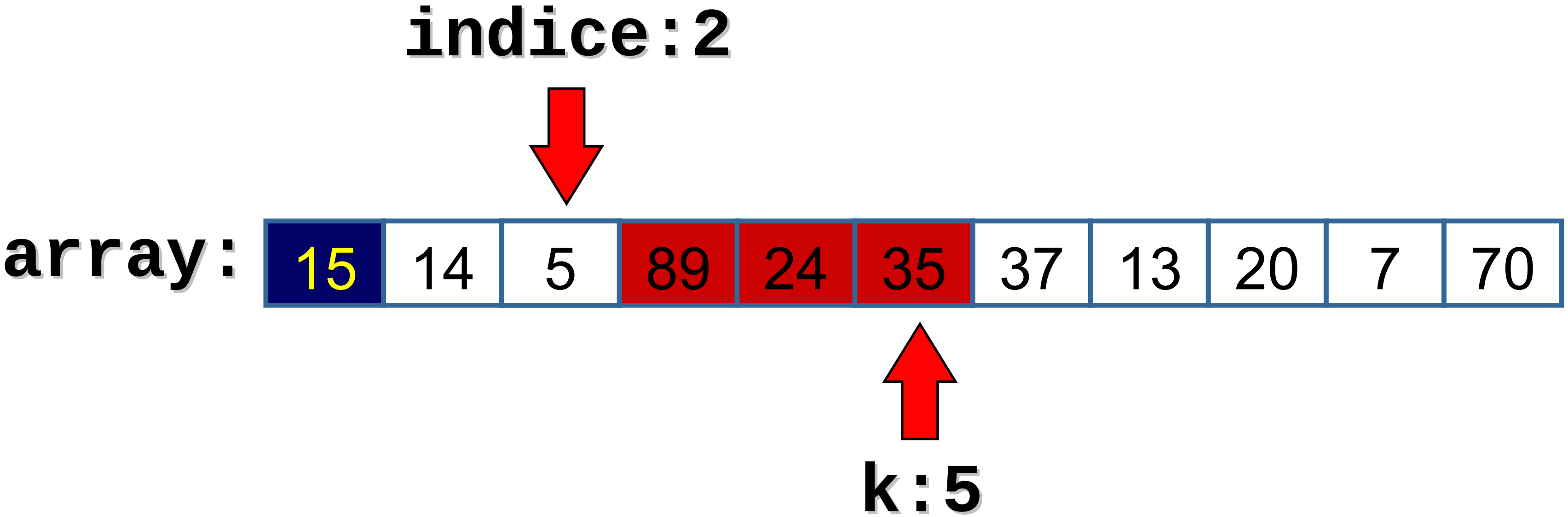
Quicksort: Ejemplo

Algoritmos de Ordenación



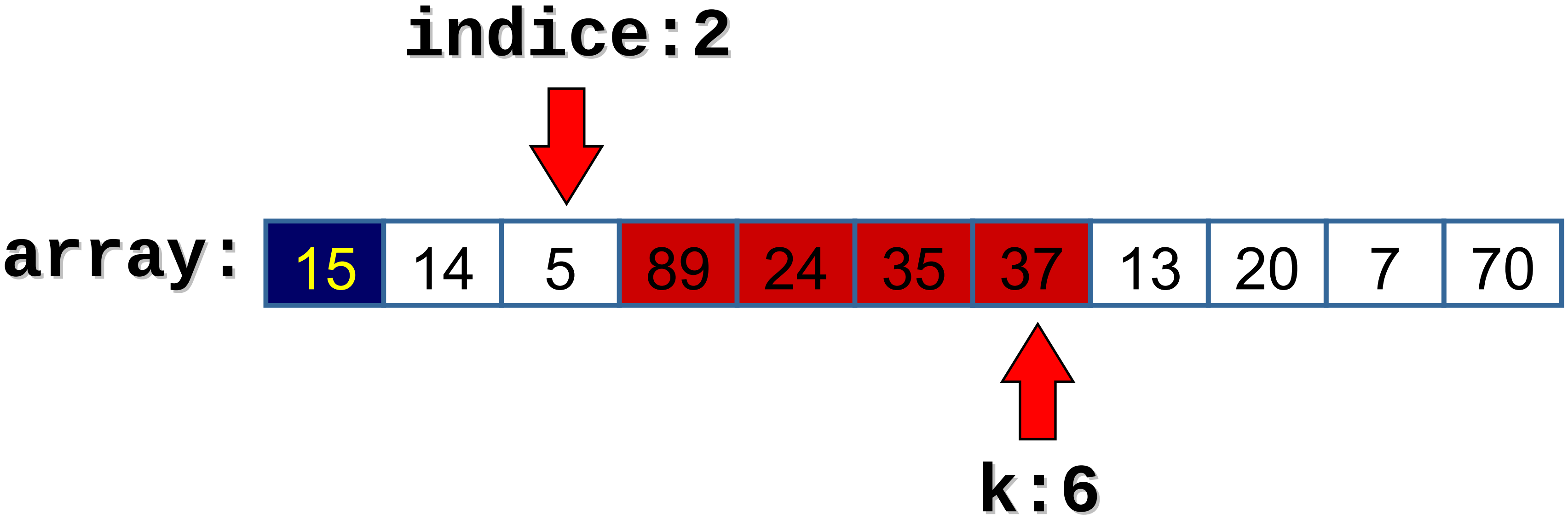
Quicksort: Ejemplo

Algoritmos de Ordenación



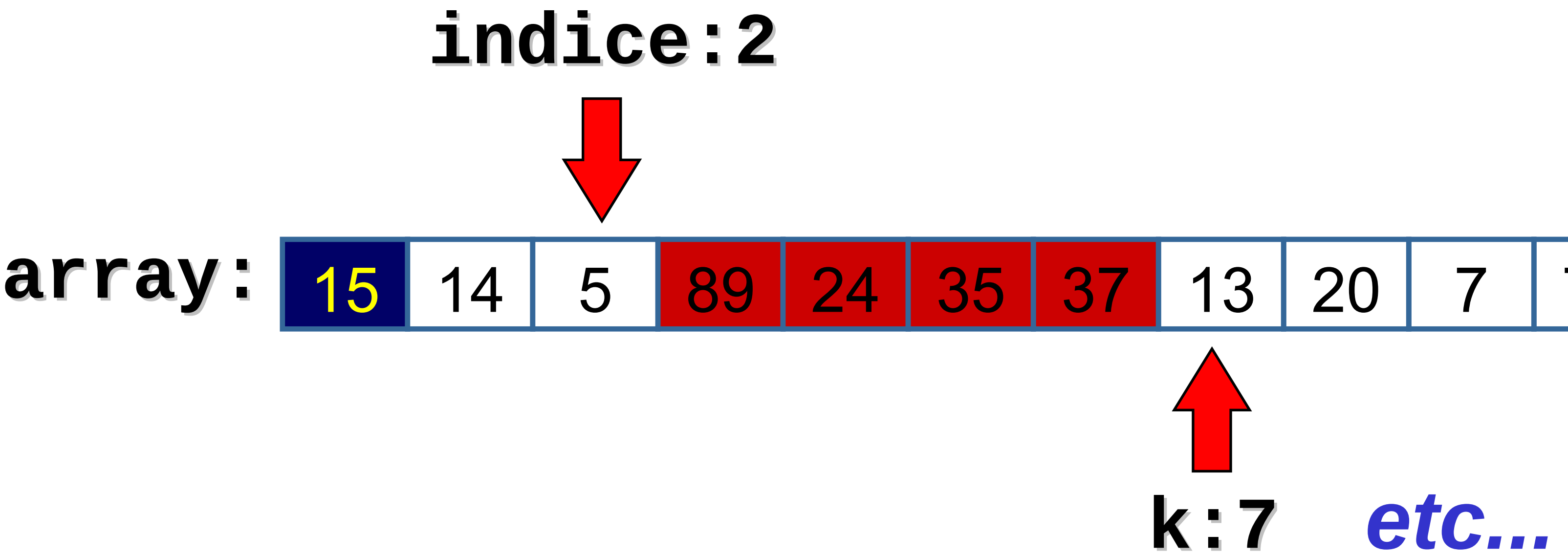
Quicksort: Ejemplo

Algoritmos de Ordenación



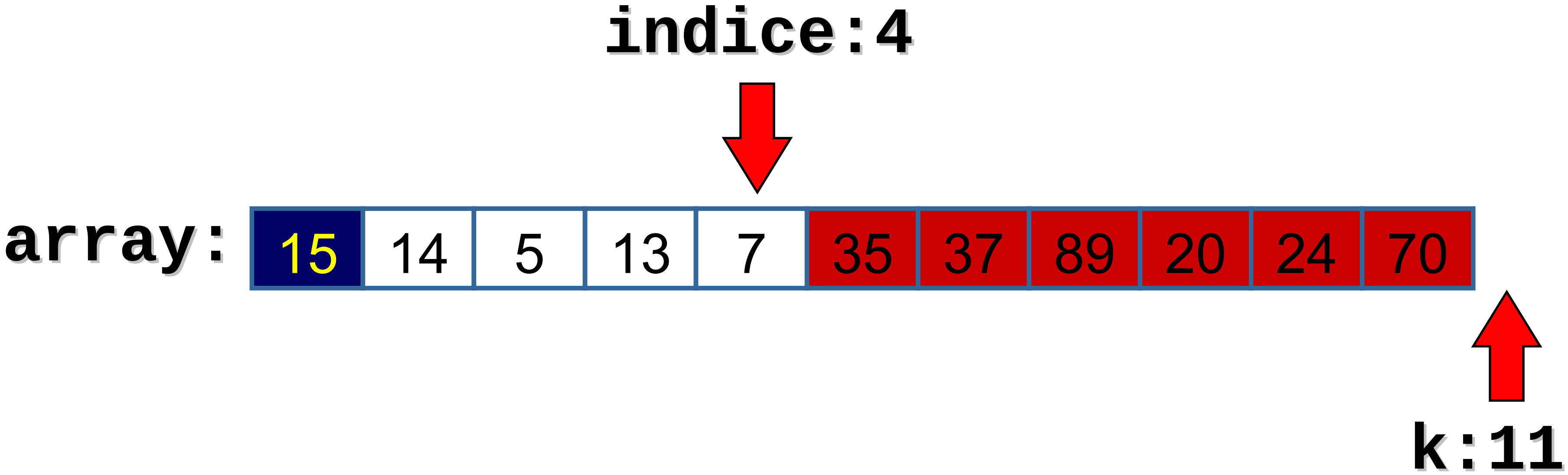
Quicksort: Ejemplo

Algoritmos de Ordenación



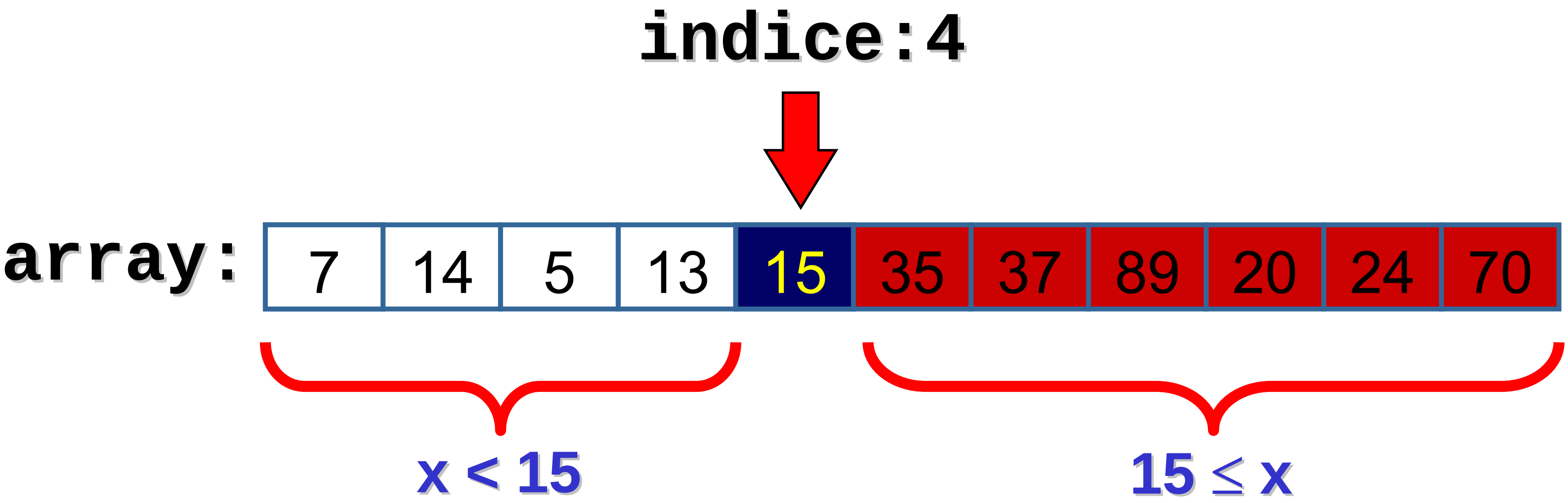
Quicksort: Ejemplo

Algoritmos de Ordenación



Quicksort: Ejemplo

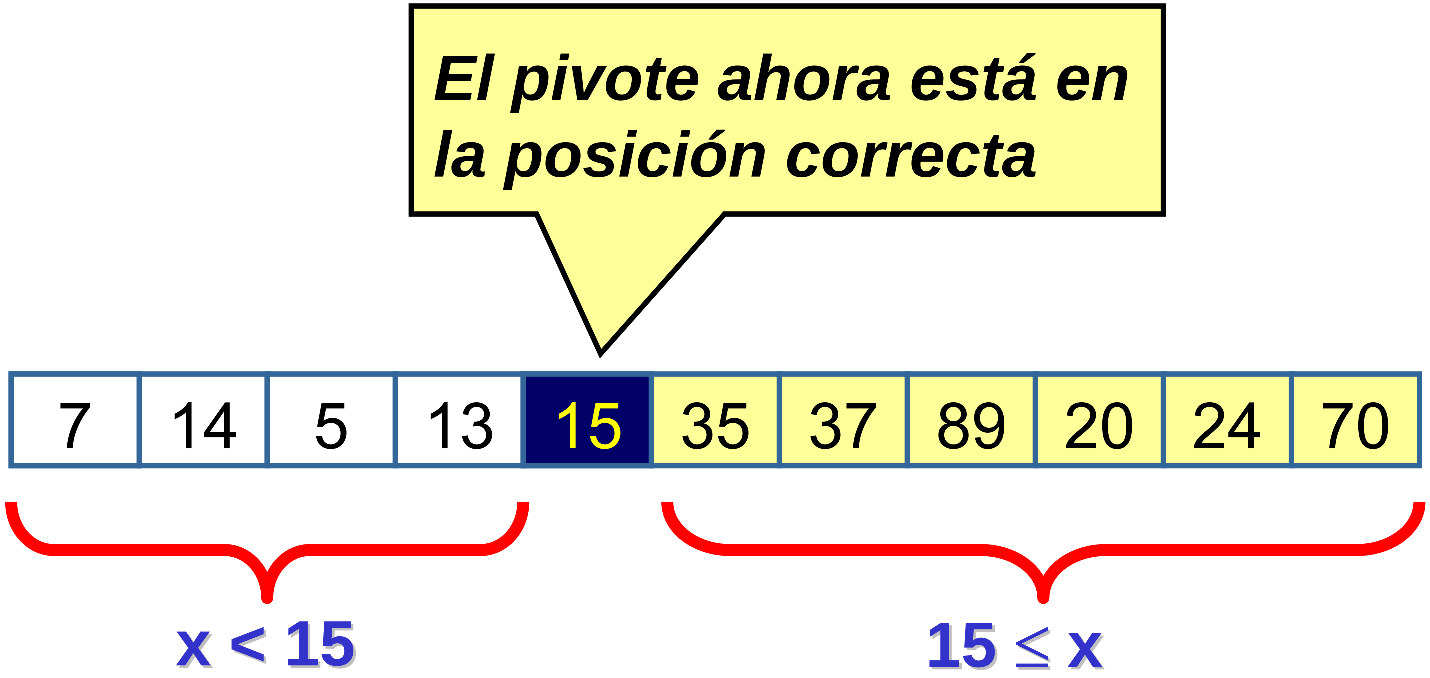
Algoritmos de Ordenación



Quicksort: Ejemplo

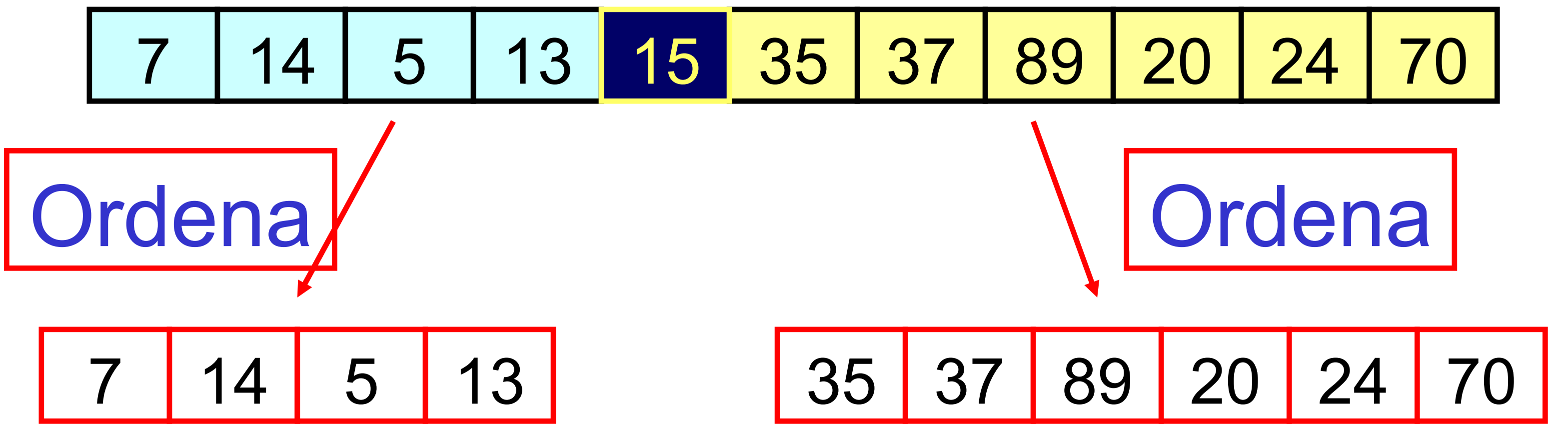
Algoritmos de Ordenación

array:



Quicksort: Ejemplo

Algoritmos de Ordenación



Quicksort: Ejemplo

Algoritmos de Ordenación

Procedimiento Quicksort

El pivote sigue quedándose fuera de ambas partes

Procedimiento quicksort (T[i..j])

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

Si j-i es pequeño Entonces Insercion (T[i..j])

Caso contrario

Escoger un pivote adecuado (sea l la posición del mismo)

Pivoteo_lineal (T[i..j], l)

{tras el 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])

Algoritmos de Ordenación

Quicksort: Pivoteo_lineal (T[i..j], var pos)

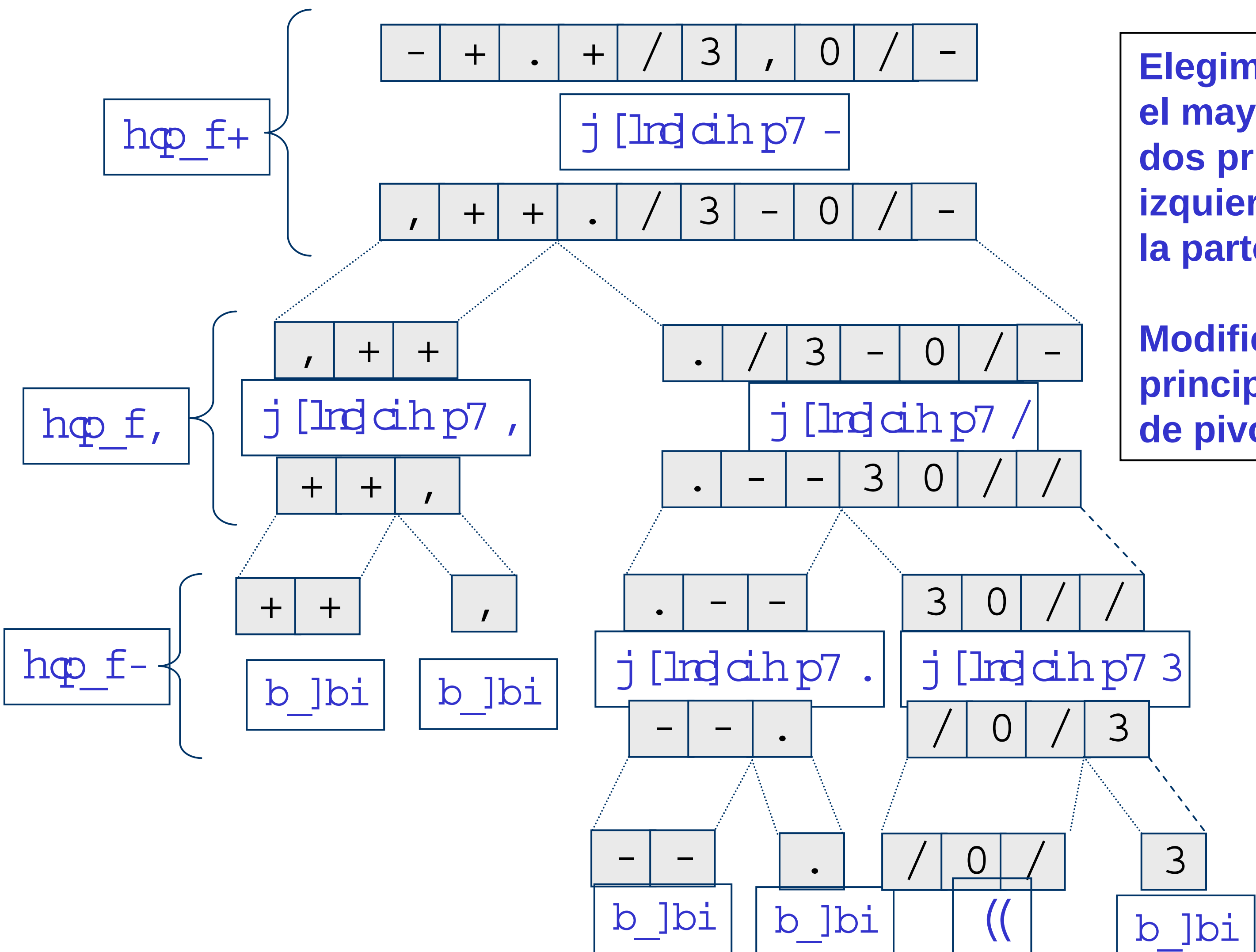
- Intercambiar T[pos] y T[i]
- Sea p = T[i] el pivote.
- Una buena forma de pivotear consiste en explorar el array T[i..j] solo una vez, pero comenzando desde ambos extremos.
- Los punteros k y l se inicializan en i y j+1 respectivamente.
- El puntero k se incrementa entonces hasta que T[k] > p, y el puntero l se disminuye hasta que T[l] ≤ p. Ahora T[k] y T[l] estan intercambiados. Este proceso continua mientras que k < l.
- Finalmente, T[i] y T[l] se intercambian para poner el pivote en su posicion correcta, y se devuelve pos = l

Algoritmos de Ordenación

Quicksort: Algoritmo de Pivoteo

```
Procedimiento Pivoteo_lineal (T[i..j], var pos)
   {permuta los elementos en el array T[i..j] de tal forma que
    al final i ≤ l ≤ 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]}
   intercambiar T[pos] y T[i];
   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]
   pos = l;
```

Algoritmos de Ordenación



Elegimos el pivote como el mayor elemento de los dos primeros por la izquierda y se incluye en la parte de la derecha

⇓

Modificación del cuerpo principal y de la función de pivoteo lineal

Quicksort:
Otro
Ejemplo

Algoritmos de Ordenación

Eficiencia de quicksort

- Si admitimos que
  - El procedimiento de pivoteo es lineal,
  - Quicksort lo llamamos para T[1..n], y
  - Elegimos como peor caso que el pivote es 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

Algoritmos de Ordenación

Analisis de Quicksort

- Recordemos que el algoritmo de ordenación por Inserción hacia aproximadamente $n^2/2 - n/2$ comparaciones, es decir es $O(n^2)$ en el peor caso.
- En el peor caso quicksort es tan malo como el peor caso del método de inserción (y tambien de selección).
- Es que el número de intercambios que hace quicksort es unas 3 veces el número de intercambios que hace el de inserción en el peor de los casos.
- Sin embargo, en la práctica quicksort es el mejor algoritmo de ordencion que se conoce...
- ¿Qué 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 numero de comparaciones que hace en promedio en una lista de n elementos
- Quicksort, tiene un tiempo promedio O(n log 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 multiplicación de enteros, con la intención de obtener un algoritmo más eficiente para multiplicar matrices.

Multiplicación de matrices

- Supongamos el problema de multiplicar dos matrices cuadradas A, B de tamaños **n**x**n**. $C = AxB$

  $$C(i, j) = \sum_{k=1..n} A(i, k) \cdot B(k, j); \text{ Para todo } i, j = 1..n$$

- Método clásico de multiplicación:

```
for i:= 1 to N do
   for j:= 1 to N do
      suma:= 0
      for k:= 1 to N do
         suma:= suma + a[i,k]*b[k,j]
      end
      c[i, j]:= suma
   end
end
```

- El método clásico de multiplicación requiere $\Theta(n^3)$.

Multiplicación de matrices

- Aplicamos **divide y vencerás**:
  Cada matriz de **n**x**n** es dividida en cuatro submatrices de tamaño (**n**/2)x(**n**/2): A_{ij}, B_{ij} y C_{ij}.

$$\begin{array}{|c|c|}\hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \times \begin{array}{|c|c|}\hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|}\hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- Es necesario resolver 8 problemas de tamaño **n**/2.
- La combinación de los resultados requiere un O(**n**^{2}).

$$t(n) = 8 \cdot t(n/2) + a \cdot n^2$$

- Resolviéndolo obtenemos que t(n) es $O(n^3)$.
- Podríamos obtener una mejora si hiciéramos 7 multiplicaciones (o menos)...

Multiplicación de matrices

- **Multiplicación rápida de matrices (Strassen):**

  $P = (A_{11}+A_{22})(B_{11}+B_{22})$

  $Q = (A_{12}+A_{22})\ B_{11}$

  $R = A_{11}\ (B_{12}-B_{22})$

  $S = A_{22}(B_{21}-B_{11})$

  $T = (A_{11}+A_{12})B_{22}$

  $U = (A_{21}-A_{11})(B_{11}+B_{12})$

  $V = (A_{12}-A_{22})(B_{21}+B_{22})$

  $C_{11} = P + S - T + U$

  $C_{12} = R + T$

  $C_{21} = Q + S$

  $C_{22} = P + R - Q + U$

- Tenemos 7 subproblemas de la mitad de tamaño.
- ¿Cuánto es el tiempo de ejecución?

Multiplicación de Matrices

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

El método de Strassen

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

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

- **Es evidente que solo se necesitan 7 multiplicaciones en lugar de las anteriores 8, más adiciones/substracciones $O(n^2)$.**

Multiplicación de matrices

- El tiempo de ejecución será:

  $$t(n) = 7 \cdot t(n/2) + a \cdot n^2$$

- Resolviéndolo, tenemos que:
  $t(n) \in O(n^{\log_2 7}) \approx O(n^{2.807})$.

- Las constantes que multiplican al polinomio son mucho mayores (tenemos muchas sumas y restas), por lo que sólo es mejor cuando la entrada es muy grande (empíricamente, para valores en torno a **n**>120).

Multiplicación de matrices

- Aunque el algoritmo es más complejo e inadecuado para tamaños pequeños, se demuestra que la **cota de complejidad del problema** es menor que $O(n^3)$.

- **Cota de complejidad de un problema:** tiempo del algoritmo más rápido posible que resuelve el problema.

- Algoritmo clásico → $O(n^3)$
- V. Strassen (1969) → $O(n^{2.807})$
- V. Pan (1984) → $O(n^{2.795})$
- D. Coppersmith y S. Winograd (1990) → $O(n^{2.376})$
- ...

Divide y vencerás.

Conclusiones:

- **Idea básica Divide y Vencerás:** dado un problema, descomponerlo en partes, resolver las partes y juntar las soluciones.
- Idea muy sencilla e intuitiva, pero...
- ¿Qué pasa con los problemas reales de interés?
  - Pueden existir muchas formas de descomponer el problema en subproblemas → Quedarse con la mejor.
  - Puede que no exista ninguna forma viable, los subproblemas no son independientes → Descatar la técnica.
- Divide y vencerás requiere la existencia de un **método directo** de resolución:
  - Tamaños pequeños: solución directa.
  - Tamaños grandes: descomposición y combinación.
  - ¿Dónde establecer el límite pequeño/grande?

Algorítmica

<u>Tema 1. Planteamiento General</u>

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos "Divide y vencerás"

Tema 4. Algoritmos Voraces ("Greedy")

Tema 5. Algoritmos basados en Programación Dinámica

Tema 6. Algoritmos para la Exploración de Grafos ("Backtracking", "Branch and Bound")

Tema 7. Otras metodologías algorítmicas