# Programación y Estructuras de Datos Avanzadas

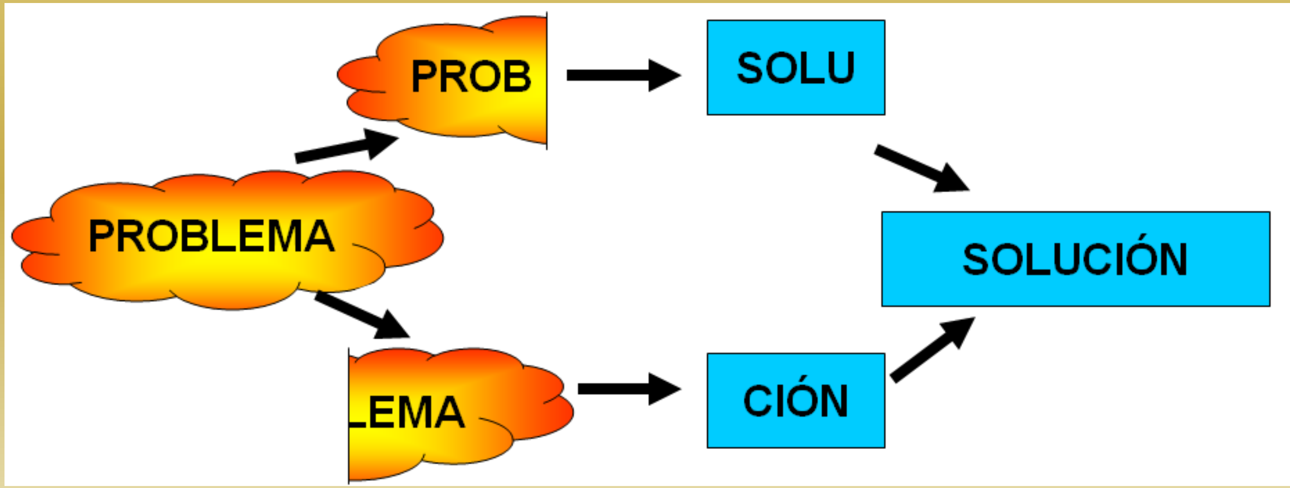
## Capítulo 4: Divide y vencerás

# Capítulo 4: Divide y vencerás (DyV)

## 4.1 Planteamiento y esquema general

- Se basa en la descomposición del problema en subproblemas del mismo tipo, reduciendo la complejidad y, en algunos casos, paralelizar su resolución.

- **Estrategia del algoritmo**
  - Descomposición del problema en subproblemas de su mismo tipo o naturaleza.
  - Se resuelven recursivamente estos subproblemas.
  - Si procede, se combinan las soluciones para obtener la solución para el problema original.



- **Esquema general:**

```
fun DyV(problema)
    si trivial(problema) entonces
        dev solución-trivial
    sino hacer
        {p1, p2, ...pk} ← descomponer(problema)
        para i ∈ (1..k) hacer
            si ← DyV(pi)
        fpara
    fsi
    dev combinar(s1, s2, ...sk)
ffun
```

  - **trivial:** comprueba si el problema es lo suficientemente pequeño para resolverlo mediante la solución-trivial → *tamaño umbral:* $n_0$
  - **solución-trivial:** método alternativo para resolver problemas pequeños (sin necesidad de descomponer el problema).
  - **descomponer**: divide el problema en *k* subproblemas de menor tamaño que el original. Si ***k*=1** (un solo subproblema) → problema de **reducción**, p. ej. factorial
  - **combinar:** obtiene la solución del problema combinando las soluciones de los subproblemas.

## • Ejemplo: Búsqueda binaria/dicotómica (en un vector ordenado)

```
fun Bbinaria(i,j:entero;v: vector [1..N] de entero; x:entero): booleano
    var
        m:entero
    var
    si i = j entonces          ← trivial: tamaño(v)=1
        si v[i] = x entonces   ┐
            dev verdadero      │
        sino                   ├ solución trivial
            dev falso          │
        fsi                    ┘
    sino
        m ← (i+j) div 2
        si v[m] ≥ x entonces
            bbinaria(i,m,v,x)      ← DyV(p1)  ┐
        sino                                  ├ Sólo un subproblema
            bbinaria(m+1,j,v,x)    ← DyV(p1') ┘
    fsi
ffun
```

No necesita función combinar $s=s_1$ (función recursiva final)

**Llamada inicial (vector completo): Bbinaria(1,n,v,x)**

**fun** DyV(problema)
  **si** trivial(problema) **entonces**
    **dev** solución-trivial
  **sino hacer**
    $\{p_1, p_2, \ldots p_k\} \leftarrow$ descomponer(problema)
    **para** $i \in (1..k)$ **hacer**
      $s_i \leftarrow \mathrm{DyV}(p_i)$
    **fpara**
  **fsi**
  **dev** combinar$(s_1, s_2, \ldots s_k)$
**ffun**

**Ejemplo: v=(1,3,8,12,13,32,56); x=32**

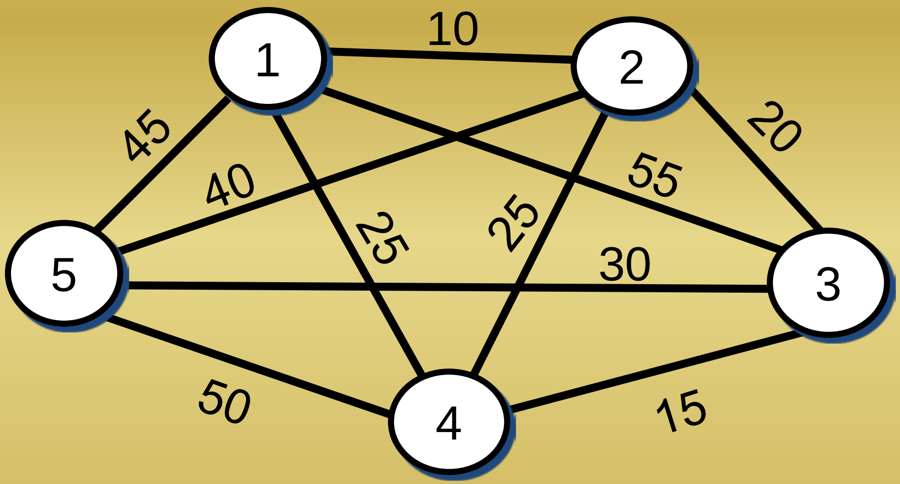
Cadena de llamadas

- Bbinaria(1,7,(1,3,8,12,13,32,56),32) → m=4
- Bbinaria(5,7,(-,-,-,-,13,32,56),32) → m=6
- Bbinaria(5,6,(-,-,-,-,13,32,-),32) → m=5
- Bbinaria(6,6,(-,-,-,-,-,32,-),32) → caso trivial

- Aplica el principio de inducción sobre los diversos ejemplares del problema.
  - Caso base: solución trivial recursiva.
  - Paso de inducción: función combinar.

- **Requisitos para aplicar divide y vencerás:**
  - Necesitamos un **método** (más o menos **directo**) de resolver los problemas de tamaño pequeño (solución trivial).
  - 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 para **combinar** los resultados de los subproblemas.

- Ejemplo: **problema del viajante**
  - Saliendo de un nodo origen encontrar el camino de forma que se recorran todos los nodos 1 vez minimizando el coste total.




- Método directo para resolver el problema $\rightarrow$ trivial con 3 nodos.
- Descomponer el problema en subproblemas más pequeños $\rightarrow$ ¿Por dónde?
- Los subproblemas deben ser disjuntos $\rightarrow$ ...parece que no
- ¿Combinar los resultados de los subproblemas? $\rightarrow$ ¡ni siquiera hay subproblemas disjuntos!

**¡Imposible aplicar divide y vencerás!**

- Cálculo del coste:
  - En todos los casos:
    - *a* $\rightarrow$ número de subproblemas en los que se descompone el problema inicial (=número de llamadas recursivas).
    - *b* $\rightarrow$ factor de reducción del tamaño del problema.
    - *k* $\rightarrow$ complejidad de las operaciones que no son llamada recursiva (p.ej. la operación de combinación o la generación de los subproblemas).
  - **Reducción por división**

$$T(n)=\begin{cases} cn^k & ,\text{ si } 1\le n<b \\ aT(n/b)+cn^k & ,\text{ si } n\ge b \end{cases} \Rightarrow T(n)\in\begin{cases} \Theta(n^k) & ,\text{ si } a<b^k \\ \Theta(n^k\log n) & ,\text{ si } a=b^k \\ \Theta(n^{\log_b a}) & ,\text{ si } a>b^k \end{cases}$$

  - **Reducción por substracción**

$$T(n)=\begin{cases} cn^k & ,\text{ si } 1\le n<b \\ aT(n-b)+cn^k & ,\text{ si } n\ge b \end{cases} \Rightarrow T(n)\in\begin{cases} \Theta(n^k) & ,\text{ si } a<1 \\ \Theta(n^{k+1}) & ,\text{ si } a=1 \\ \Theta(a^{n/b}) & ,\text{ si } a>1 \end{cases}$$

**Ejemplo:** búsqueda binaria, T(*n*)=1T(*n*/2)+*c* $\rightarrow$ *a*=1, *b*=2, *k*=0 (reducción por div.)

$$\Downarrow$$

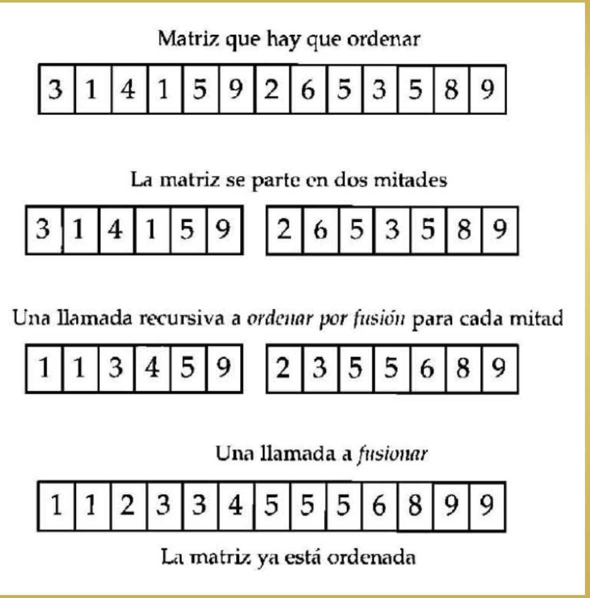
$$\Theta(n^k\log n)=\Theta(\log n)$$

# 4.2 Ordenación por fusión (*Mergesort*)

- Dado un vector de enteros T[1..*n*], se divide por la mitad y se invoca al algoritmo para ordenar cada mitad por separado. Tras ésta operación, se fusionan estos dos subvectores ordenados en uno solo.

- ***Algoritmo Mergesort***

**fun** Mergesort (T: vector [1..n] de entero): vector [1..n] de entero
    **var**
        U: vector $[1..n]$ de entero, V: vector $[1..n]$ de entero
    **fvar**
    **si** trivial(n) **entonces** Insertar(T[1..n])
    **sino**
        $U[1..\lfloor n \div 2 \rfloor] \leftarrow T[1..\lfloor n \div 2 \rfloor]$
        $V[1..\lceil n \div 2 \rceil] \leftarrow T[1+\lfloor n \div 2 \rfloor..n]$
        Mergesort(U)
        Mergesort(V)
        Fusionar(U,V,T)
    **fsi**
**ffun**




  - **Trivial/solución-trivial:** comprueba si *n* es pequeño, en tal caso lo resuelve mediante la ordenación por inserción (insertar)
  - **descomponer**: divide el problema de tamaño *n* en 2 subproblemas de tamaño n/2 (el vector *T* se divide en *U* y *V*)
  - **combinar:** fusiona los 2 vectores ordenados en uno (fusionar).

- ***Procedimiento fusionar***
  - **Objetivo:** fusiona los vectores ordenados *U* y *V* para obtener el vector ordenado *T*.
  - **Clave:** al fusionar 2 vectores que ya están ordenados utiliza 3 índices y consigue realizar la operación en un tiempo lineal O(n)

```
fun Fusionar (U:vector [1..n+1] de entero,V:vector [1..m+1] de entero,T:vector [1..m+n]
de entero)
    var
        i,j: natural
    fvar
    i,j ← 1
    U[m+1],V[n+1] ← ∞
    para k ← 1 hasta m+n hacer
    si U[i] < V[j]
    entonces T[k] ← U[i]
        i ← i+1
    sino T[k] ← V[j]
        j ← j+1
    fsi
    fpara
ffun
```

- ***Coste algoritmo:*** $T(n)=2T(n/2)+cn \rightarrow a=2,\ b=2,\ k=1$ (reducción por div.)

Nota: $k=1$ porque la complejidad de las operaciones que no son llamada recursiva (generación de subcasos, función combinar, etc.) es lineal (culpa de fusionar).

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

$$\Downarrow$$

$$\Theta(n^k \log n) = \Theta(n \log n)$$

# 4.4 Ordenación rápida (*Quicksort*)

- Ordenación por fusión → cálculo de subproblemas trivial, función combinar compleja.
- Ordenación rápida → cálculo de subproblemas más complejo, combinar trivial.
  - Se lleva a cabo una descomposición para que todos los elementos del primer subproblema sean menores o iguales que los del segundo.
  - Para esta división se utiliza un elemento del vector (pivote).

- ***Algoritmo Quicksort***

```
fun Quicksort (T[i..j])
    si trivial(j-i) entonces Insertar(T[i..j])
    sino
        Pivotar(T[i..j], l);
        Quicksort(T[i..l-1]);
        Quicksort(T[l+1..j])
    fsi
ffun
```

***Procedimiento Pivotar***

```
fun Pivotar (T:vector [i..j] de entero, var l:entero)
    var
        p,k:entero
    fvar
    p ← T[i]
    k ← i; l ← j+1
    repetir k ← k+1 hasta T[k] > p ∨ k ≥ j
    repetir l ← l-1 hasta T[l] ≤ p
    mientras k < l hacer
        intercambiar(T,k,l)
        repetir k ← k+1 hasta T[k] > p
        repetir l ← l-1 hasta T[l] ≤ p
    fmientras
    intercambiar(T,i,l)
ffun
```

- **Trivial/solución-trivial:** comprueba si *j-i* es pequeño, en tal caso lo resuelve mediante la ordenación por inserción (insertar)
- **descomponer**: divide el problema de tamaño *n* en 2 subproblemas (donde los elementos del primer vector son ≤ que los del segundo) de tamaño ≅n/2 *si el pivote es la mediana.*
- **combinar:** no es necesaria.

- ***Procedimiento Pivotar***

– **Entrada:** vector T[i..j], variable $l$ por referencia.

– **Salida:** índice $l$ y vector T[i..j] "semiordenado" t.q. i≤$l$≤j; T[k]≤p para todo i≤$k$<$l$; T[$l$]=$p$ y T[k]>p para todo $l$<$k$≤j, donde p es el valor inicial de T[i]

```
fun Pivotar (T:vector [i..j] de entero, var l:entero)
    var
        p,k:entero
    fvar
    p ← T[i]
    k ← i; l ← j+1
    repetir k ← k+1 hasta T[k] > p ∨ k ≥ j
    repetir l ← l−1 hasta T[l] ≤ p
    mientras k < l hacer
        intercambiar(T,k,l)
        repetir k ← k+1 hasta T[k] > p
        repetir l ← l−1 hasta T[l] ≤ p
    fmientras
    intercambiar(T,i,l)
ffun
```

Ejemplo Pivotar

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

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

| 3 | 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 | $\underline{4}$ | 1 | 5 | 9 | 2 | 6 | 5 | $\overline{3}$ | 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 | $\underline{5}$ | 9 | $\overline{2}$ | 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 | $\overline{2}$ | $\underline{9}$ | 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

≤ pivote | $l$=5 | > pivote

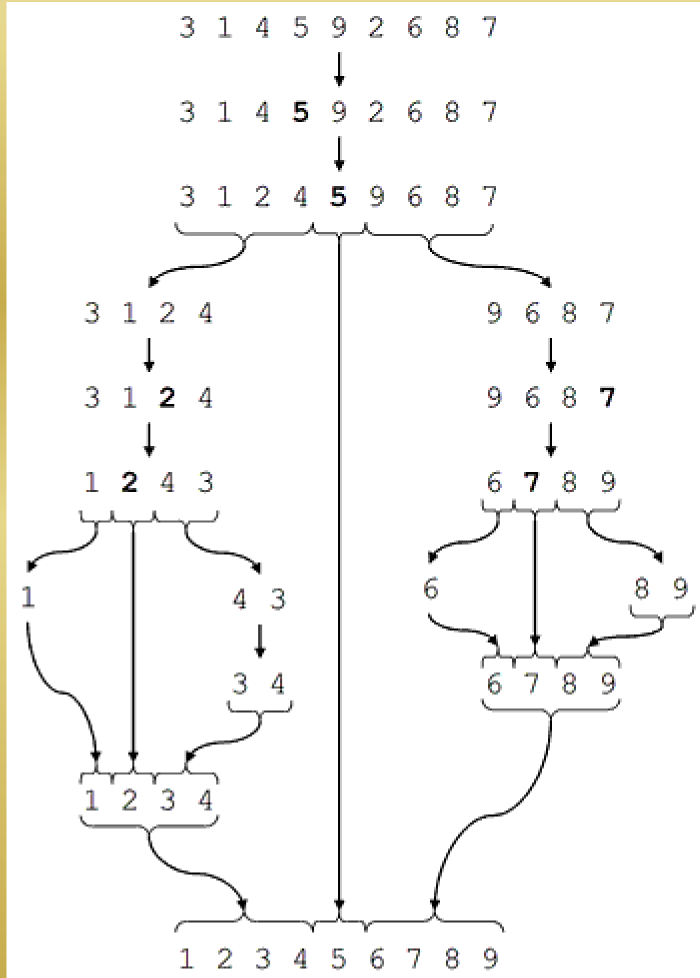
- ***Coste algoritmo (caso promedio=subproblemas equilibrados):***

T($n$)=2T($n$/2)+c$n$ $\rightarrow$ $a$=2, $b$=2, $k$=1 $\Rightarrow \Theta(n^k \log n) = \Theta(n \log n)$ en promedio

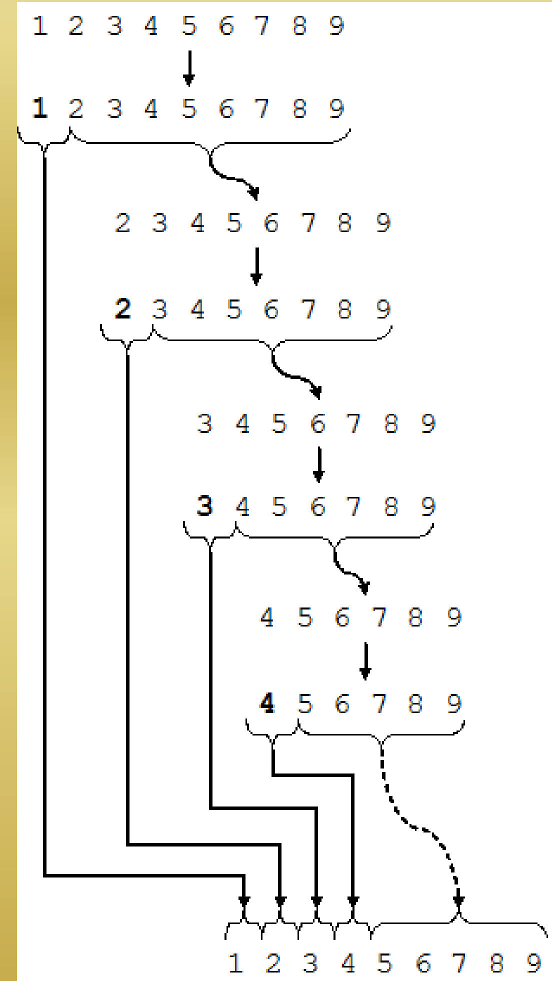
Nota: k=1 por la complejidad lineal de pivotar y, en el caso equilibrado, reducción por división con $b$=2.

*Ejemplo con vector aleatorio*

*log n* llamadas recursivas (y a Pivotar)




*Ejemplo con vector ya ordenado*




*n* llamadas recursivas (y a Pivotar)

- ***Coste algoritmo (caso peor=subproblemas desequilibrados):***

T($n$)=1T($n$-1)+c$n$ $\rightarrow$ $a$=1, $b$=1 (substracción), $k$=1 $\Rightarrow \Theta(n^{k+1}) = \Theta(n^2)$ caso peor

Nota: b=1 porque realmente se genera un solo subproblema de tamaño $n$-1 (el otro tiene tamaño 0)

## Algunos comentarios sobre Quicksort

- La eficiencia en el caso peor puede mejorarse utilizando un procedimiento Pivotar alternativo que devuelva 2 índices así como elegir como pivote la mediana del vector → *eficiencia en el caso peor: $\Theta(n \log n)$ aunque con una constante oculta tan grande que no sería más rápido que Mergesort of Heapsort (también de ese coste en caso peor).*

- Conviene elegir el tamaño umbral (caso trivial) de forma apropiada (aunque esta elección no influye en el coste asintótico). Un valor $n_0$=8 suele ser adecuado.

- Algoritmos de ordenación:
  - ✓ Clásicos: inserción, selección, burbuja: $\Theta(n^2)$, caso peor
  - ✓ No clásicos: HeapSort y MergeSort $\Theta(n \log n)$, caso peor; Quicksort $O(n \log n)$ en caso promedio

  A pesar del caso peor, este algoritmo de ordenación (Quicksort) es el más rápido en el caso promedio (las constantes ocultas son mejores que Mergesort y Heapsort).

# 4.3 El puzzle tromino

- **Tromino** $\rightarrow$ figura geométrica compuesta por 3 cuadros de tamaño 1x1 en forma de L.

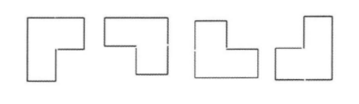
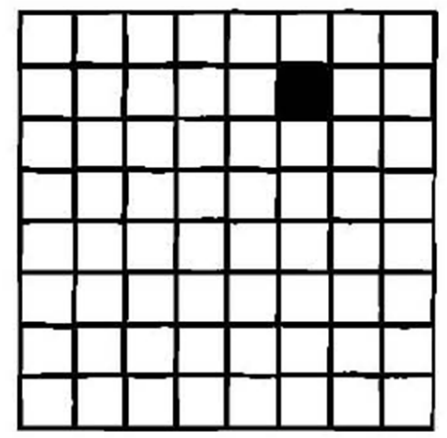


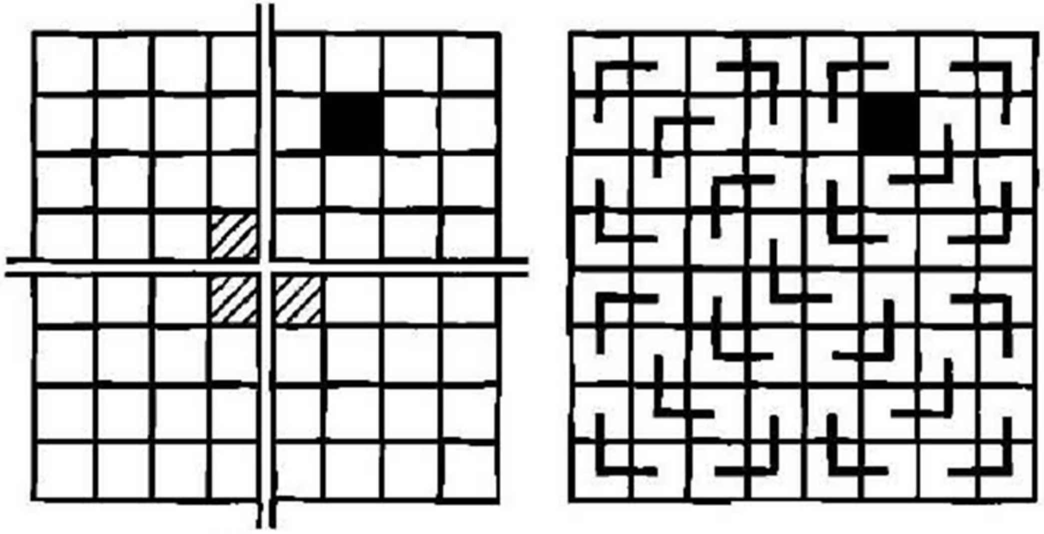
Figura 4.1: *Rotaciones de un tromino*

- **Problema** $\rightarrow$ se parte de una retícula de *nxn* con un *cuadrado especial* 1x1 marcado y el resto vacío. Se trata de rellenar completamente la retícula con trominos sin solaparlos.

- ***¡Resoluble mediante DyV cuando n es potencia de 2!***



Tablero con un cuadrado especial



Colocada la primera baldosa Solución

- ***Algoritmo Tromino***

```
fun Tromino(T:matriz [1..n,1..n] de natural,n,m:natural)
    var
        T1,T2,T3,T4 :matriz [1..n,1..n] de natural
    fvar

    si n = 2 entonces
        T ← colocaTromino(T,m)
    dev T
sino
    m' ← esquina cuadrante con casilla negra
    colocaTromino(T,m')
    T1,T2,T3,T4 ← dividir T en 4 cuadrantes
    tromino(T1,n/2,m1)
    tromino(T2,n/2,m2)
    tromino(T3,n/2,m3)
    tromino(T4,n/2,m4 )
fsi
T ← combinar(T1,T2,T3,T4)
dev T
ffun
```

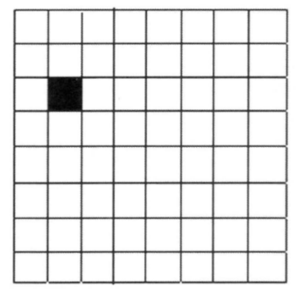


Figura 4.2: *Retículo con casilla en negro*

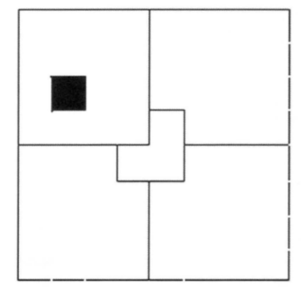


Figura 4.3: *Proceso de resolución en primera recursión*

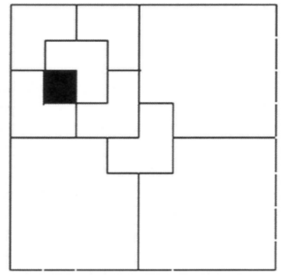


Figura 4.4: *Proceso de resolución en segunda recursión*

- **Trivial/solución-trivial:** cuando ***n*=2**, tenemos una retícula 2x2 con una casilla negra, por lo que la colocación del tromino es trivial.
- **descomponer**: el tablero de tamaño $n \times n = 2^k \times 2^k$ se descompone en 4 subtableros de tamaño $n/2 \times n/2 = 2^{k-1} \times 2^{k-1}$, cada unos con una casillas marcada (una sera la casilla especial del tablero y las otras 3 resultado de colocar un tromino en el centro) → 4 subproblemas de tamaño n/2
- **combinar:** se recogen en el tablero solución los triminos ubicados en cada cuadrante → si se utiliza un array único y las divisiones se marcan a través de índices descomponer+combinar∈O(1).

- ***Coste algoritmo:*** $T(n)=4T(n/2)+c \rightarrow a=4,\ b=2,\ k=0$ (reducción por div.) $\Rightarrow \Theta(n^{\log_b a}) = \Theta(n^2)$

# 4.5 Cálculo del elemento mayoritario en un vector

- **Problema** → dado un vector v[1..n] de naturales, se quiere averiguar *si existe un elemento mayoritario, es decir que aparezca al menos n/2+1 veces en el vector*. La forma más sencilla es dividir el vector en 2 mitades y combinar las soluciones adecuadamente.

- ***Algoritmo Mayoritario***

```
fun Mayoritario(i,j:natural;v:vector [1..n] de natural): entero
    var
        m:natural
        s1,s2:entero
    fvar
    si i = j entonces
        dev v[i]
    sino
        m ← (i + j) ÷ 2
        s1 ← Mayoritario(i,m,v)
        s2 ← Mayoritario(m+1,j,v)
    dev Combinar(s1,s2,v)
ffun
```

Devuelve el elemento mayoritario del vector o -1 si éste no existe

Si el vector tiene un solo elemento este es el elemento mayoritario

Los únicos candidatos para elementos mayoritarios son $s_1$ o $s_2$: combinar chequea ambas posibilidades

- **Trivial/solución-trivial:** cuando *n*=1 (i=j), el único elemento del vector es el mayoritario.
- **Descomponer**: el vector de tamaño n se descompone en 2 vectores de tamaño n/2 → 2 subproblemas de tamaño n/2.

- **Combinar**: en función de los valores $s_1$ y $s_2$ hay 4 posibilidades para el calculo de $s$:
  - $s_1=s_2=-1 \rightarrow$ No hay elemento mayoritario ($s$=-1)
  - $s_1\neq-1$ y $s_2=-1 \rightarrow$ Hay que comprobar si $s_1$ es el elemento mayoritario del vector
  - $s_1=-1$ y $s_2\neq-1 \rightarrow$ Hay que comprobar si $s_2$ es el elemento mayoritario del vector
  - $s_1=s_2\neq-1 \rightarrow$ El elemento mayoritario del vector es $s_1=s_2$
  - $-1\neq s_1\neq s_2\neq-1 \rightarrow$ Hay que comprobar si $s_1$ o $s_2$ es el elemento mayoritario.

***Función Combinar***

```
fun Combinar (a,b:entero;v:vector [1..n] de natural):entero
    si a = −1 ∧ b = −1 entonces dev -1 fsi
    si a = −1 ∧ b ≠ −1 entonces dev ComprobarMayoritario(b,v) fsi
    si a ≠ −1 ∧ b = −1 entonces dev ComprobarMayoritario(a,v) fsi
    si a ≠ −1 ∧ b ≠ −1 entonces
        si ComprobarMayoritario(a,v) = a entonces dev a
        sino si ComprobarMayoritario(b,v) = b entonces dev b fsi
        sino dev -1
        fsi
    fsi
fun
```

***Función Comprobar mayoritario (chequea un elemento en el vector completo)***

```
fun ComprobarMayoritario (x:natural;v:vector [1..n] de natural):entero
    var
        c:natural
    fvar
    c ← 0
    para k ← 1 hasta n hacer
        si v[k]=x entonces c ← c + 1 fsi
    fpara
    si c > n/2 entonces dev x sino dev -1 fsi
ffun
```

- Función combinar de coste lineal O(n), función descomponer O(1) $\rightarrow k=1$.

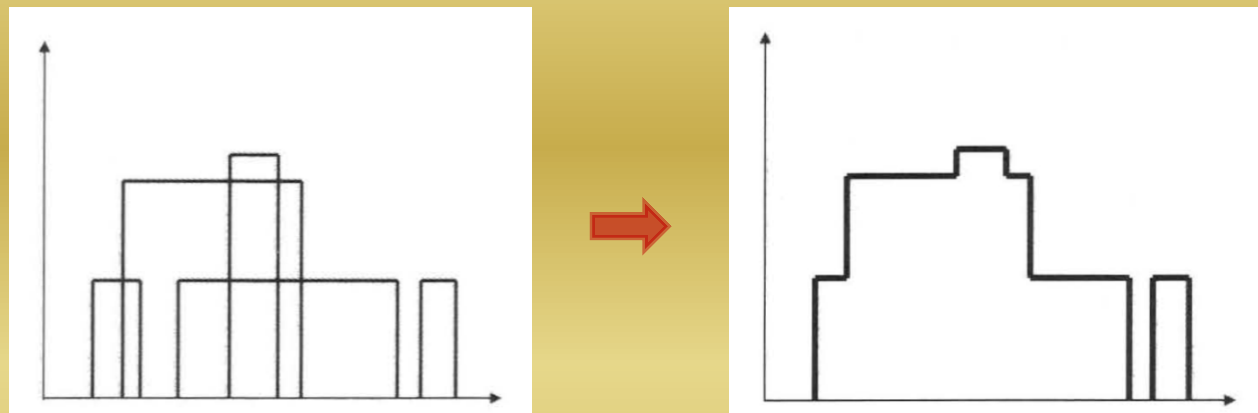
- ***Coste algoritmo:*** $T(n)=2T(n/2)+cn \rightarrow a=2, b=2, k=1$ (reducción por div.)

$$\Downarrow$$

$$\Theta(n^k \log n) = \Theta(n \log n)$$

# 4.7 *Skyline* de una ciudad

- **Problema** → dada una ciudad con edificios de distintas alturas hay que calcular la línea de horizonte de la ciudad.



  – Lo resolveremos dividiendo el problema (*n* edificios) en 2 subproblemas de tamaño mitad (n/2 edificios en cada uno).

***Esquema general***

```
fun DyV(problema)
    si trivial(problema) entonces
        dev solución-trivial
    sino hacer
        {p1, p2, ...pk} ← descomponer(problema)
        para i ∈ (1..k) hacer
            si ← DyV(pi)
        fpara
    dev combinar(s1, s2, ...sk)
ffun
```

***Refinamiento del esquema general***

- **Trivial/solución-trivial:** el caso trivial es realizar el skyline de un edificio **(*n*=1)**.
- **descomponer**: el conjunto de edificios se divide en dos mitades iguales. → 2 subproblemas de tamaño n/2
- **combinar:** suponemos, por inducción resuelto el problema, es decir, que la entrada a este algoritmo son dos soluciones consistentes en sendas líneas de horizonte. La combinación consiste en fusionar estas líneas de horizonte eligiendo la mayor ordenada para cada abscisa donde haya edificios → O(n)

- **Estructuras de datos:**
  - *Entrada* $\rightarrow$ ciudad *n* edificios: $C=\{e_1,e_2,..,e_n\}$, siendo $e_1=(x_1,x_2,h)$
  - *Salida* $\rightarrow$ TipoSkyline: $S=(x_1,h_1,x_2,h_2,..,x_k,h_k)$

$C = \{(1,3,3)(1,6,2)(2,4,4)(6,7,3)\}$

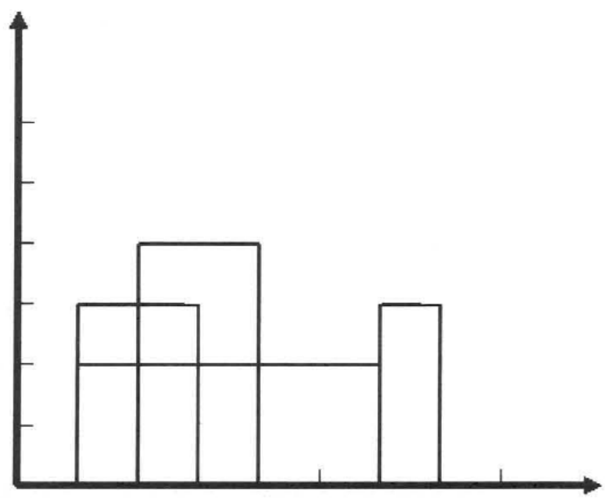


Figura 4.7: *Una ciudad, con 4 edificios*

$S = (1,3,2,4,4,2,6,3,7,0)$

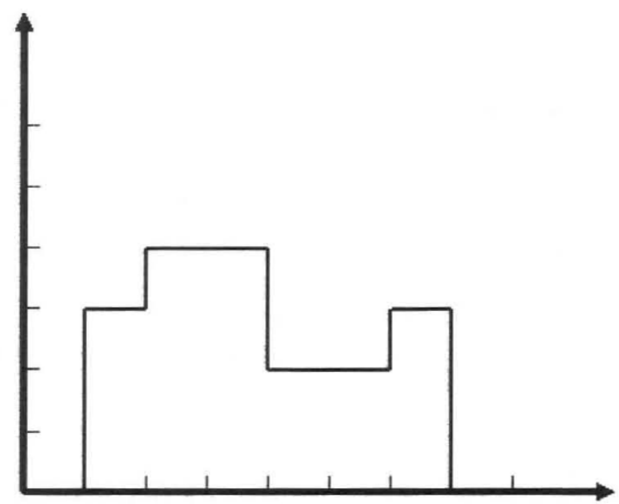


Figura 4.8: *Línea de horizonte resultante*

## Algoritmo edificios

```
fun Edificios(C: vector [1..n] de edificio; i,j:natural): TipoSkyline
    var
        m,n: natural
    fvar
    m ←(i+j-1)/2
    n ← j-i+1
    si n = 1 entonces
        dev convierte_edificio_en_skyline(C[i])      ← Trivial
    sino
        s1 ← Edificios(C,i,m)
        s2 ← Edificios(C,m+1,j)
        s ← Combinar(s1,s2)
        dev s
    fsi
ffun
```

– Función combinar de coste lineal O(n), función descomponer O(1) → $k$=1.

- ***Coste algoritmo:*** $T(n)=2T(n/2)+cn$ → $a$=2, $b$=2, $k$=1 (reducción por div.)

$$\Downarrow$$

$$\Theta(n^k \log n) = \Theta(n \log n)$$

## Función combinar

```
fun Combinar(s1,s2: TipoSkyline)
    var
        i,j,k: natural
    fvar
    n ← s1.longitud()
    m ← s2.longitud()
    s1_x ← ExtraeOrdenadas(s1)
    s1_h ← ExtraeAlturas(s1)
    s2_x ← ExtraeOrdenadas(s2)
    s2_h ← ExtraeAlturas(s2)
    i ← 1; j ← 1            ← Ojo: índices 1..n
    k ← 1; S ← [ ]
    mientras (i ≤ n) ∨ (j ≤ m) hacer      ← Se accede al índice 0 y al n+1 (centinelas?)
        x ← min(s1_x[i], s2_x[j])
        si s1_x[i] < s2_x[j] entonces
            max ← max(s1_h[i], s2_h[j − 1])
            i ← i+1
        sino
            si s1_x[i] > s2_x[j] entonces
                max ← max(s1_h[i − 1], s2_h[j])
                j ← j+1
            sino
                max ← max(s1_h[i], s2_h[j])
                i ← i+1
                j ← j+1
            fsi
        fsi
        S_x[k] ← x       ┐
        S_h[k] ← max     ├ ← Hacerlo sólo si max cambia con respecto al de la iteracción anterior
        k ← k+1          ┘
    fmientras
    S ← S_x ∪ S_h
    dev S
ffun
```

# 4.6 Liga de equipos

- **Problema** $\rightarrow$ tenemos $n$=$2^k$ equipos que desean realizar una liga. Cada equipo juega un partido al día y la liga se celebra en n-1 días. El objetivo es realizar un calendario el que por cada par de equipos se indique el día de juego.

*Solución:* matriz simétrica *T*, donde $T[e_i,e_j]=d_{ij}$ indica que los equipos $e_i$ y $e_j$ juegan el día $d_{ij}$

| | $e_1$ | $e_2$ | ... | $e_n$ |
|---|---|---|---|---|
| $e_1$ | - | 3 | ... | 1 |
| $e_2$ | 3 | - | ... | $\boxed{4}$ |
| ... | ... | ... | ... | ... |
| $e_n$ | 1 | 4 | | - |

Lo resolveremos dividiendo el problema (*n* equipos que tienen que jugar en *n*-1 días, en 2 subproblemas de tamaño mitad (*n*/2 equipos que tienen que jugar en *n*/2-1 días cada uno).

***Refinamiento del esquema general: Algoritmo Torneo***

```
fun Torneo(i,j,d:natural)
    var
        m,n: natural;
    fvar
    m ←(i+j-1)/2;
    n ← j-i+1;
    si n = 2 entonces
        T[i,j] ← d;
    sino
        Torneo(i,m,d)
        Torneo(m+1,j,d)
        Combinar(i,j,d)
```

- **Trivial/solución-trivial:** la liga consta de sólo 2 equipos **(*n*=2)** $e_i$ y $e_j$ que juegan en *n*-1=1 día. Si empiezan jugando el día *d*, jugarán ese mismo día $\rightarrow$ $T[i,j]=d$.
- **descomponer**: si tenemos *n* equipos en el rango *i*..*j*, lo dividiremos en 2 subproblemas con los equipos de *i*..*m* y de *m*+1..*j* (*m* indica el índice central, *m*=[*i*+*j*-1]/2), ambos de n/2 equipos $\rightarrow$ 2 subproblemas de tamaño n/2

- **combinar:** suponemos, por inducción resuelto el problema para tamaño $n/2$, los equipos de cada uno de los conjuntos $c_A=\{e_1,e_2,\ldots,e_{n/2}\}$ y $c_B=\{e_{m+1}..e_j\}$ ya tienen organizada la liga A y B jugando en $n/2$-1 días. El objetivo es que juegen entre ellos en n/2 días que quedan libres$\rightarrow$ rotaciones usando aritmética modular:
  - Equipos: $c_A=\{e_i..e_m\}$ y $c_B=\{e_{m+1}..e_j\}$
  - Días libres: $n/2$ hasta $n$-1

El día $n/2$:

$$\begin{array}{cccc} e_i & e_{i+1} & \cdots & e_m \\ | & | & & | \\ e_{m+1} & e_{m+2} & \cdots & e_j \end{array}$$

El día $n/2+1$:

$$\begin{array}{ccccc} e_i & e_{i+1} & \cdots & e_{m-1} & e_m \\ | & | & & | & | \\ e_{m+2} & e_{m+3} & \cdots & e_j & e_{m+1} \end{array}$$

$\cdots$

$\cdots$

Y así sucesivamente hasta el día $n-1$:

$$\begin{array}{cccc} e_i & e_{i+1} & \cdots & e_m \\ | & | & & | \\ e_j & e_{m+1} & \cdots & e_{j-1} \end{array}$$

***Algoritmo Combinar***

```
fun Combinar(i,j,d:natural)
    var
        m,n,s,t: natural
        a,b: natural
    fvar
    m ←(i+j-1)/2
    n ← j-i+1
    para s ← 0 hasta n/2-1 hacer        } O(n²)
        para t ← 0 hasta n/2-1 hacer    }
            a ← i+t
            b ← m+1+((t+s) mod (n/2))
            T[a,b] ← d+s+n/2-1
        fpara
    fpara
ffun
```

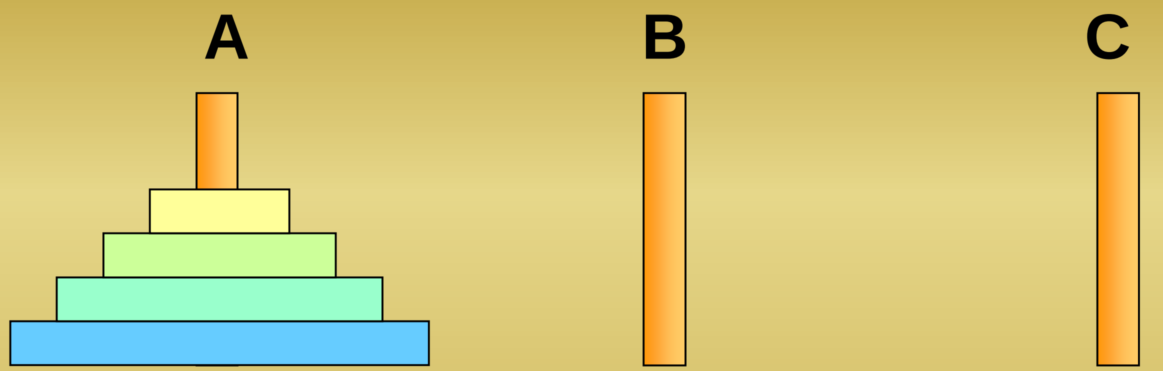
***Coste algoritmo:*** $T(n)=2T(n/2)+cn^2 \rightarrow a=2$, $b=2$, $k=2$ (reducción por división)

$$\Downarrow$$

$$\Theta(n^k)=\Theta(n^2)$$

# Ejemplos adicionales: Torres de Hanoi

- **Problema** $\rightarrow$ 3 postes A, B y C. Tengo n discos de distintos diámetros en el poste y el objetivo es pasarlos al C, pero con las siguientes condiciones:
  - Sólo se puede mover un disco de cada vez.
  - No se puede colocar un disco sobre otro de menor diámetro.




Solución:

- Mover ***n*-1** discos de A a B
- Mover 1 disco de A a C
- Mover ***n*-1** discos de B a C

**Hanoi (n, A, B, C: entero)**
    **si** n=1 **entonces** ⟵ trivial: 1 sólo disco
        mover (A, C)
    **sino**
        Hanoi (n-1, A, C, B)
        mover (A, C)
        Hanoi (n-1, B, A, C)
    **fin si**

***Coste algoritmo:*** $T(n)=2T(n-1)+c \rightarrow a=2$, $b=1$, $k=0$ (reducción por substracción)

$$\Downarrow$$

$$\Theta(a^{n/b}) = \Theta(2^n)$$

# Ejemplos adicionales: Multiplicación de enteros largos

- **Problema** $\rightarrow$ multiplicación de 2 enteros de *n* dígitos (los suponemos iguales)
- Algoritmo clásico: $n^2$ multiplicaciones de nº de 1 dígito (+ desplazamientos) $\rightarrow O(n^2)$
- Algoritmo divide y vencerás:
  - **Trivial/Caso trivial:** si $n$=1 (números de 1 dígito) utilizamos la multiplicación escalar.
  - **Descomponer:** los enteros de tamaño *n* son divididos en dos trozos de tamaño n/2. Subproblemas $\rightarrow$ multiplicación de enteros de tamaño n/2.

| | | | |
|---|---|---|---|
| **u** | w | x | $u = w \cdot 10^S + x$ |
| **v** | y | z | $v = y \cdot 10^S + z$ |
| | $\lceil n/2 \rceil$ | $\lfloor n/2 \rfloor$=S (número de dígitos de x y z) | |

  - **Combinar:** sumar los resultados de los anteriores (con los desplazamientos adecuados).

$$r = u \cdot v = 10^{2S} \cdot w \cdot y + 10^{S} \cdot (w \cdot z + x \cdot y) + x \cdot z$$

- Opción 1: 4 multiplicaciones de tamaño $n/2$ ($w \cdot y$; $w \cdot z$; $x \cdot y$ y $x \cdot z$)
  - Combinar: coste lineal $O(n)$
  - Coste: $t(n) = 4 \cdot t(n/2) + c \cdot n \in O(n^{\log_2 4}) = O(n^2) \rightarrow$ no mejora el algoritmo clásico.

- Opción 2: 3 multiplicaciones de tamaño $n/2$ (Karatsuba y Ofman):
  - $u \cdot v = 10^{2S} \cdot w \cdot y + 10^{S} \cdot [(w+x) \cdot (z+y) - w \cdot y - x \cdot z] + x \cdot z$
  - Coste: $t(n) = 3 \cdot t(n/2) + c' \cdot n \in O(n^{\log_2 3}) = O(n^{1.59}) \rightarrow$ mejor eficiencia asintótica pero ojo con las constantes ocultas (función combinar mucho más compleja), valor umbral $n_0 \approx 500$.

# Ejemplos adicionales: El problema de selección

- **Problema** → Sea **T**[1..n] un array (no ordenado) de enteros, y sea **s** un entero entre 1 y n. El **problema de selección** consiste en encontrar el elemento que se encontraría en la posición **s** si el array estuviera ordenado.

Vector T

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 9 | 2 | 5 | 4 | 3 | 4 | 10 | 1 | 6 |

- Si s = $\lceil n/2 \rceil$, entonces tenemos el problema de encontrar la **mediana** de T, es decir el valor que es mayor que la mitad de los elementos de T y menor que la otra mitad.
- **¿Cómo resolverlo?**

**a) Forma sencilla:** ordenar T y devolver el valor T[s] → Θ(n log n).

*¡Pero realmente no necesito ordenar todo el vector, sólo 1 elemento!*

**b) Idea:** Usar el procedimiento **pivote** de Quicksort

```
Selección (T: array [1..n] de entero; s: entero)
    var i, j, l: entero
    fvar
    i ← 1
    j ← n
    repetir
        Pivote (i, j, l)
        si s < l entonces
                j:= l-1
        sino si s > l entonces
                i:= l+1
        fsi
    hasta l=s
    dev T[l]
```

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| x | x | x ≤ | z | ≤ y | y | y | y | y | y |

↑

*l*

***Coste algoritmo:*** *No recursivo*

- *Caso promedio:* se reduce el tamaño en n/2 → log(n) iteraciones del bucle → se puede demostrar que es coste lineal
- Caso peor: el tamaño se reduce en 1 → n iteraciones del bucle → coste cuadrático.

# Ejemplos adicionales:

- Exponenciación rápida (función exponencial)

$$a^n = \begin{cases} a & \text{si } n = 1 \\ \left(a^{n/2}\right)^2 & \text{si } n \text{ es par} \\ a \times a^{n-1} & \text{en caso contrario} \end{cases}$$

Por ejemplo,

$$a^{29} = a\ a^{28} = a(a^{14})^2 = a((a^7)^2)^2 = \cdots = a\ ((a(a\ a^2)^2)^2)^2$$

- Generalización función de Fibonacci generalizada.
- Multiplicación de dos polinomios.
- Par de puntos más cercano.
- Multiplicación de matrices muy grandes (Algoritmo de Strassen).

…

→ Casi todos los ejercicios propuestos en “ESQUEMAS ALGORÍTMICOS. ENFOQUE METODOLÓGICO Y PROBLEMAS RESUELTOS”