


Programa de teoría

Parte I. Estructuras de Datos.

1. Abstracciones y especificaciones.
2. Conjuntos y diccionarios.
3. Representación de conjuntos mediante árboles.
4. Grafos.

Parte II. Algorítmica.

1. Análisis de algoritmos.
-  **2. Divide y vencerás.**
3. Algoritmos voraces.
4. Programación dinámica.
5. Backtracking.
6. Ramificación y poda.

PARTE II: ALGORÍTMICA

Tema 2. Divide y vencerás.

2.1. Método general.

2.2. Análisis de tiempos de ejecución.

2.3. Ejemplos de aplicación.

2.3.1. Multiplicación rápida de enteros largos.

2.3.2. Multiplicación rápida de matrices.

2.3.3. Ordenación por mezcla y ordenación rápida.

2.3.4. El problema de selección.

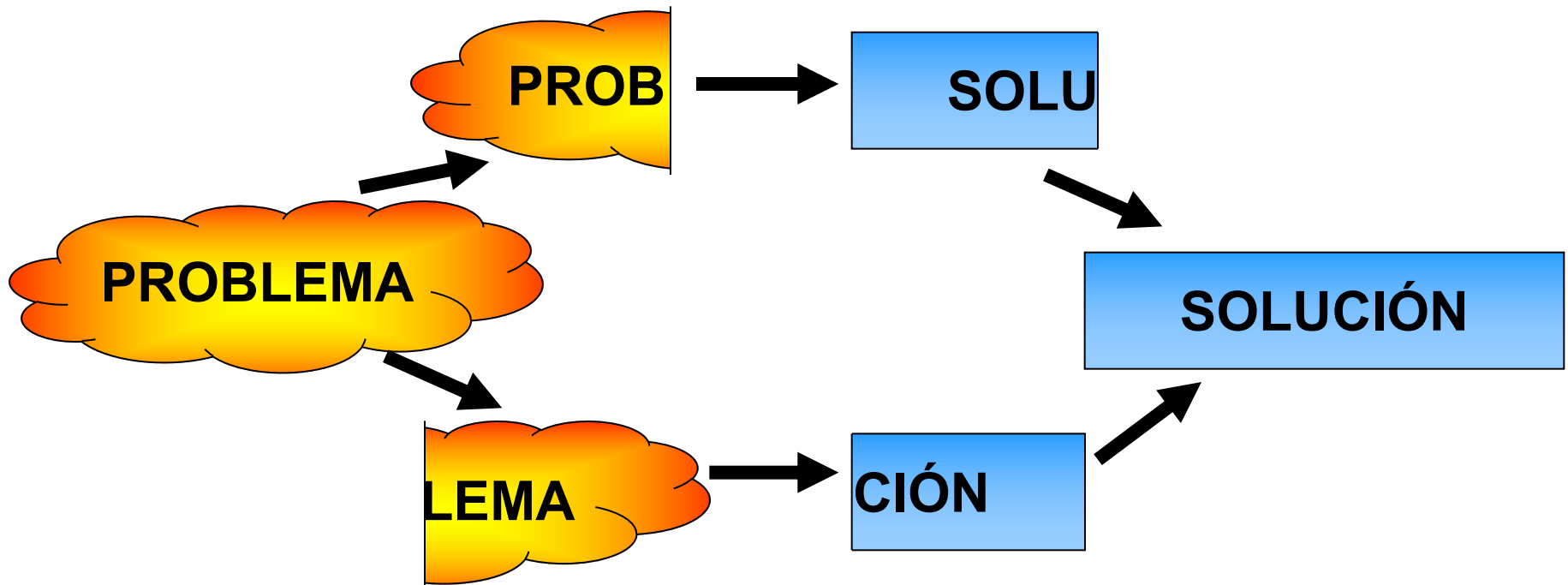
2.3.5. Ejemplo de DyV 1D – Secuencia más larga

2.3.6. Ejemplo de DyV 2D – El derbi

2.3.7. DyV en búsqueda/optimización

2.1. Método general.

- 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.



2.1. Método general.

- **Ventajas:**

- Simplificar problemas “difíciles” - Ej.: Torres de Hanoi
- Eficiencia: ayuda a descubrir algoritmos eficientes. Ejs.:
 - Multiplicación rápida de enteros largos/matrices.
 - Ordenación rápida (mergesort, quicksort)
- Paralelismo: esquema ideal para paralelizar.
- Jerarquía de memoria: disminuye fallos de caché.

- **Desventajas:**

- Sobrecarga:
 - Dividir y combinar.
 - Llamadas recursivas.
- Cálculos repetidos (a veces). Ejemplo Fibonacci.

2.1. Método general.

- **Esquema general:**

DivideVencerás (p: problema)

Dividir (p, p_1, p_2, \dots, p_k)

para $i := 1, 2, \dots, k$

$s_i := \text{Resolver}(p_i)$

solución := *Combinar* (s_1, s_2, \dots, s_k)

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

2.1. Método general (**bis**).

- Esquema general:

DivideVencerás (p: problema)

Dividir (p, p_1, p_2, \dots, p_k)

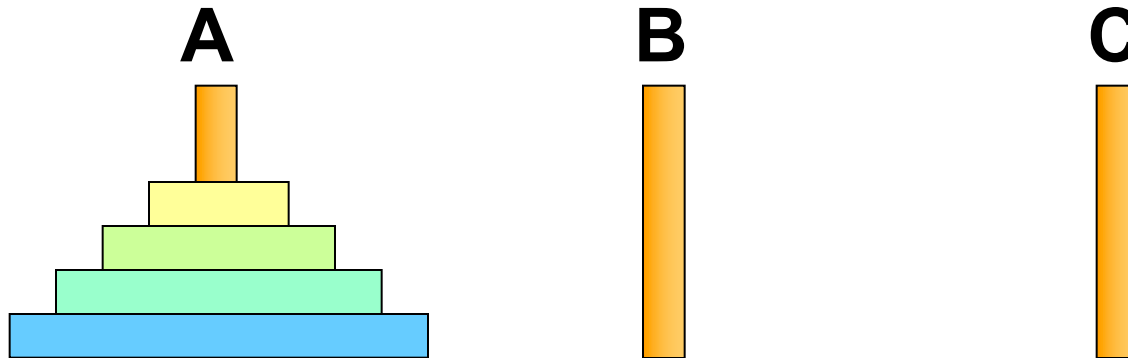
para $i := 1, 2, \dots, k$

$s_i := \text{Resolver}(p_i)$

solución := *Combinar* ($s_1, s_2, \dots, s_k, [p_1, p_2, \dots, p_k]$)

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

2.1. Método general.



- **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

2.1. Método general.

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:
 $F(n) = F(n-1) + F(n-2)$
- $F(0) = F(1) = 1$

2.1. Método general.

- **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.

2.1. Método general.

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

DivideVencerás (p, q: índice)

var m: índice

si *Pequeño* (p, q) **entonces**

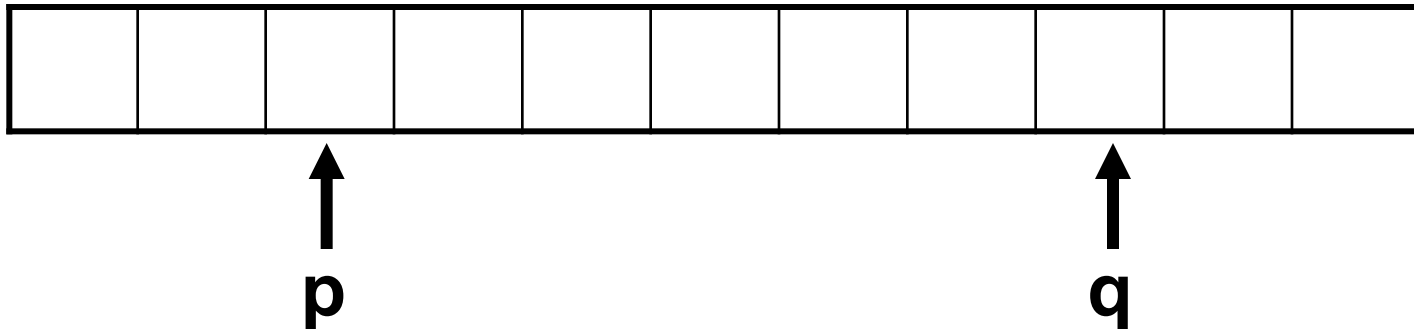
 solucion:= *SoluciónDirecta* (p, q)

sino

 m:= *Dividir* (p, q)

 solucion:= *Combinar* (DivideVencerás (p, m),
 DivideVencerás (m+1, q))

finsi



2.1. Método general (**bis**).

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

DivideVencerás (p, q: índice)

var m: índice

si *Pequeño* (p, q) **entonces**

 solucion:= *SoluciónDirecta* (p, q)

sino

 m:= *Dividir* (p, q)

 solucion , [**cálculos auxiliares**]:=

Combinar (DivideVencerás (p, m),

 DivideVencerás (m+1, q),

 [**+ datos, típicamente p, q, m**])

finsi



p

A.E.D.

Tema 2. Divide y vencerás.

q

2.1. Método general.

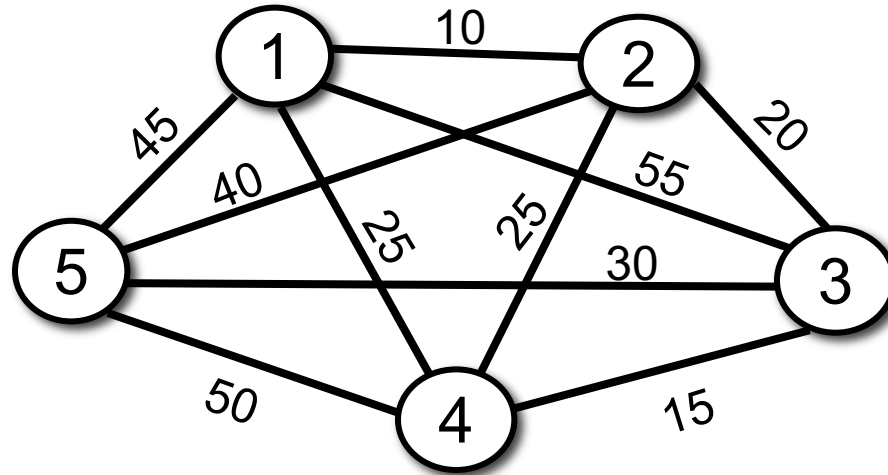
- Aplicación de divide y vencerás: encontrar la forma de definir las **funciones genéricas**:
 - *Pequeño*: Determina cuándo el problema es pequeño para aplicar la resolución directa.
 - *SoluciónDirecta*: Método alternativo de resolución para tamaños pequeños.
 - *Dividir*: Función para descomponer 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...

2.1. Método general.

- 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.

2.1. Método general.

- **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!!

2.1. Método general.

- 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.
 $\text{Fact}(n) := n * \text{Fact}(n-1)$

2.2. Análisis de tiempos de ejecución.

(REPASO del tema 1)

- 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:

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

2.2. Análisis de tiempos de ejecución.

- **Ejemplo 1.** 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 2.** Realizamos 4 llamadas recursivas con trozos de tamaño $n/2$, con $f(n) \in O(n)$:
 $a = 4; b = 2$
 $t(n) \in O(n^{\log_2 4}) = O(n^2)$

(...)

2.3. Ejemplos de aplicación.

2.3.1. Multiplicación rápida de enteros largos.

>> Concepto: **TAMAÑO ÓPTIMO DEL CASO BASE**

2.3.2. Multiplicación rápida de matrices.

>> Concepto: **COTA DE COMPLEJIDAD de un PROBLEMA**

2.3.3. Ordenación por mezcla y ordenación rápida.

2.3.4. El problema de selección.

2.3.5. Ejemplo de DyV 2D – El derbi

2.3.6. DyV en búsqueda/optimización

2.3. Ejemplos de aplicación.

2.3.1. Multiplicación rápida de enteros largos.

- Queremos representar enteros de tamaño arbitrariamente grande: mediante listas de cifras.

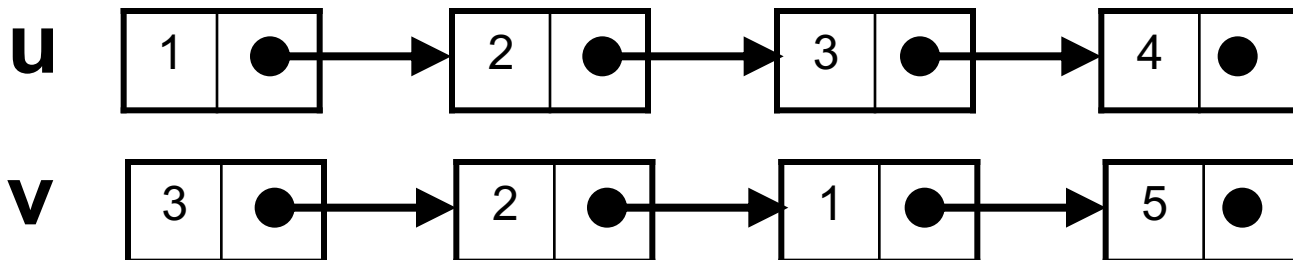
tipo EnteroLargo = Puntero[Nodo]

Nodo = **registro**

valor : 0..9

sig : EnteroLargo

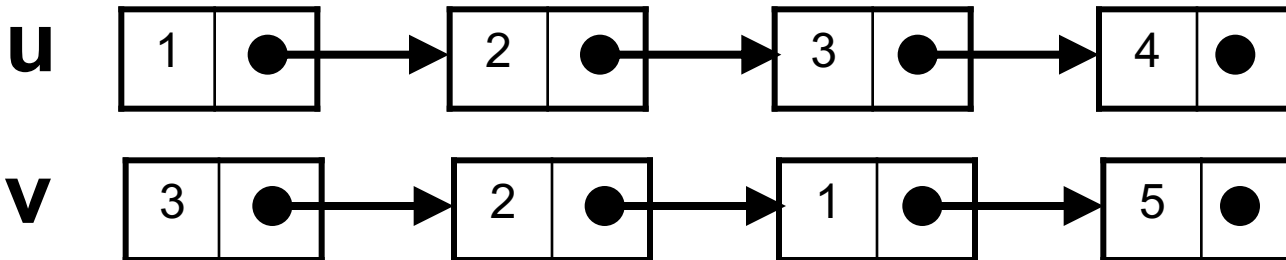
finregistro



- Implementar operaciones de suma, resta, **multiplicación**, etc.

2.3.1. Multiplicación rápida de enteros largos.

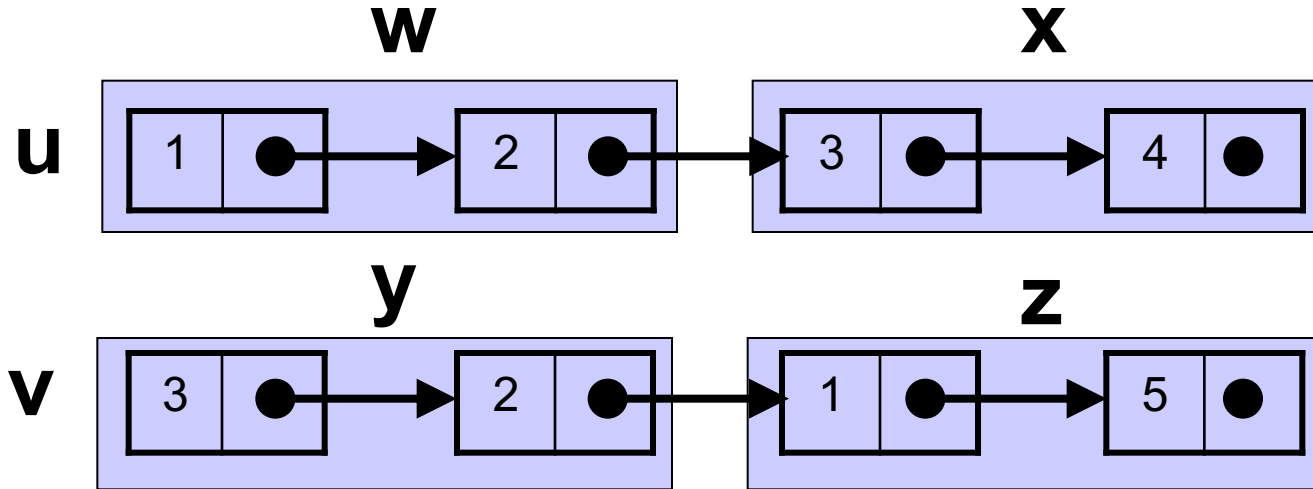
- Algoritmo clásico de multiplicación:
 - Inicialmente $r = 0$
 - Para cada cifra de v hacer
 - Multiplicar todas las cifras de u por v
 - Sumar a r con el desplazamiento correspondiente



- Suponiendo que u es de tamaño n , y v de tamaño m , ¿cuánto es el tiempo de ejecución?
- ¿Y si los dos son de tamaño n ?

2.3.1. Multiplicación rápida de enteros largos.

- Algoritmo de multiplicación con divide y vencerás:



- *Solución Directa*: si el tamaño es 1, usar la multiplicación escalar
- *Dividir*: descomponer los enteros de tamaño n en dos trozos de tamaño $n/2$
- Resolver los subproblemas correspondientes
- *Combinar*: sumar los resultados de los subproblemas con los desplazamientos correspondientes

2.3.1. Multiplicación rápida de enteros largos.

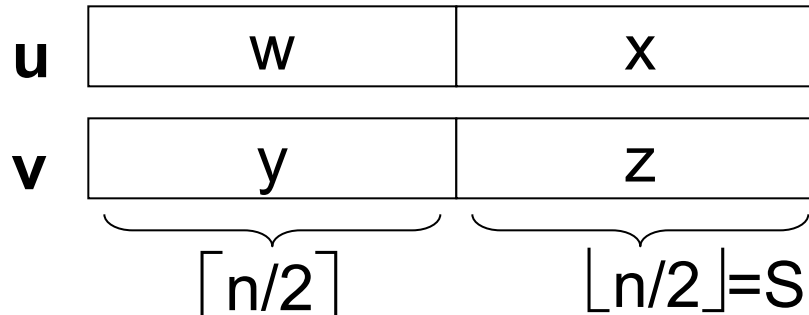
u	w	x
v	y	z

$u = w \cdot 10^S + x$
 $v = y \cdot 10^S + z$

$\underbrace{\hspace{100px}}_{\lceil n/2 \rceil} \quad \underbrace{\hspace{100px}}_{\lfloor n/2 \rfloor = S}$

- Cálculo de la multiplicación con **divide y vencerás**:
$$r = u \cdot v = 10^{2S} \cdot w \cdot y + 10^S \cdot (w \cdot z + x \cdot y) + x \cdot z$$
- El problema de tamaño **n** es descompuesto en 4 problemas de tamaño **n/2**.
- La suma se puede realizar en un tiempo lineal $O(n)$.
- ¿Cuánto es el tiempo de ejecución?
- $t(n) = 4 \cdot t(n/2) + d \cdot n \in O(n^2) \rightarrow$ No mejora el método clásico.

2.3.1. Multiplicación rápida de enteros largos.



$$u = w \cdot 10^S + x$$

$$v = y \cdot 10^S + z$$

- ¿Y si en vez de 4 tuviéramos 3 subproblemas...?
- **Multiplicación rápida de enteros largos (Karatsuba y Ofman):**

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

- Subproblemas:
 - $m1 = w \cdot y$
 - $m2 = (w-x) \cdot (z-y)$
 - $m3 = x \cdot z$

2.3.1. Multiplicación rápida de enteros largos.

operación **Mult**(u, v : EnteroLargo; $n, base$: entero) : EnteroLargo

si $n == base$ **entonces**

devolver MultBasica(u, v)

sino

asignar(w , primeros($n/2$, u))

asignar(x , ultimos($n/2$, u))

asignar(y , primeros($n/2$, v))

asignar(z , ultimos($n/2$, v))

asignar($m1$, Mult($w, y, n/2, base$))

asignar($m2$, Mult($x, z, n/2, base$))

asignar($m3$, Mult(restar(w, x), restar(z, y), $n/2, base$))

devolver sumar(sumar(Mult10($m1, n$),
Mult10(sumar(sumar($m1, m2$), $m3$), $n/2$)), $m2$)

finsi

- ¿Cuánto es el tiempo de ejecución?

2.3.1. Multiplicación rápida de enteros largos.

- En este caso se requieren 3 multiplicaciones de tamaño $n/2$:
$$t(n) = 3 \cdot t(n/2) + d' \cdot n \in O(n^{\log_2 3}) \approx O(n^{1.59})$$
- El método es asintóticamente mejor que el método clásico.
- Sin embargo, las constantes y los términos de menor orden son mucho mayores (la combinación es muy costosa).
- En la práctica se obtiene beneficio con números de más de 500 bits...

2.3.1. Multiplicación rápida de enteros largos.

- **Conclusión:**
 - Para tamaños pequeños usar el método directo.
 - Para tamaños grandes usar el método de Karatsuba y Ofman.
- El caso base no necesariamente debe ser $n = 1...$
- Cuestión importante en Divide y Vencerás:
 - >> ¿Cómo elegir del caso base?

Concepto de **TAMAÑO ÓPTIMO DEL CASO BASE**

>> Ver fichero “tema2a - estudio caso base óptimo texto guía”

>> Ver vídeo en “tema2b - Integer multiplication in time $O(n \log n)$ ”

2.3.2. Multiplicación rápida de matrices.

- Supongamos el problema de multiplicar dos matrices cuadradas A, B de tamaños $n \times n$. $C = A \times B$

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

- 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 $\Theta(n^3)$.

2.3.2. Multiplicación rápida de matrices.

- Aplicamos **divide y vencerás**:

Cada matriz de $n \times n$ es dividida en cuatro submatrices de tamaño $(n/2) \times (n/2)$: A_{ij} , B_{ij} y C_{ij} .

A_{11}	A_{12}
A_{21}	A_{22}

 \times

B_{11}	B_{12}
B_{21}	B_{22}

 $=$

C_{11}	C_{12}
C_{21}	C_{22}

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

- 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)...

2.3.2. Multiplicación rápida de matrices.

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

$$\left. \begin{aligned} P &= (A_{11}+A_{22})(B_{11}+B_{22}) \\ Q &= (A_{12}+A_{22}) B_{11} \\ R &= A_{11} (B_{12}-B_{22}) \\ S &= A_{22}(B_{21}-B_{11}) \\ T &= (A_{11}+A_{12})B_{22} \\ U &= (A_{21}-A_{11})(B_{11}+B_{12}) \\ V &= (A_{12}-A_{22})(B_{21}+B_{22}) \end{aligned} \right\} \begin{aligned} C_{11} &= P + S - T + U \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

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

2.3.2. Multiplicación rápida 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 $n > 120$).
- ¿Cuál es el tamaño óptimo del caso base?

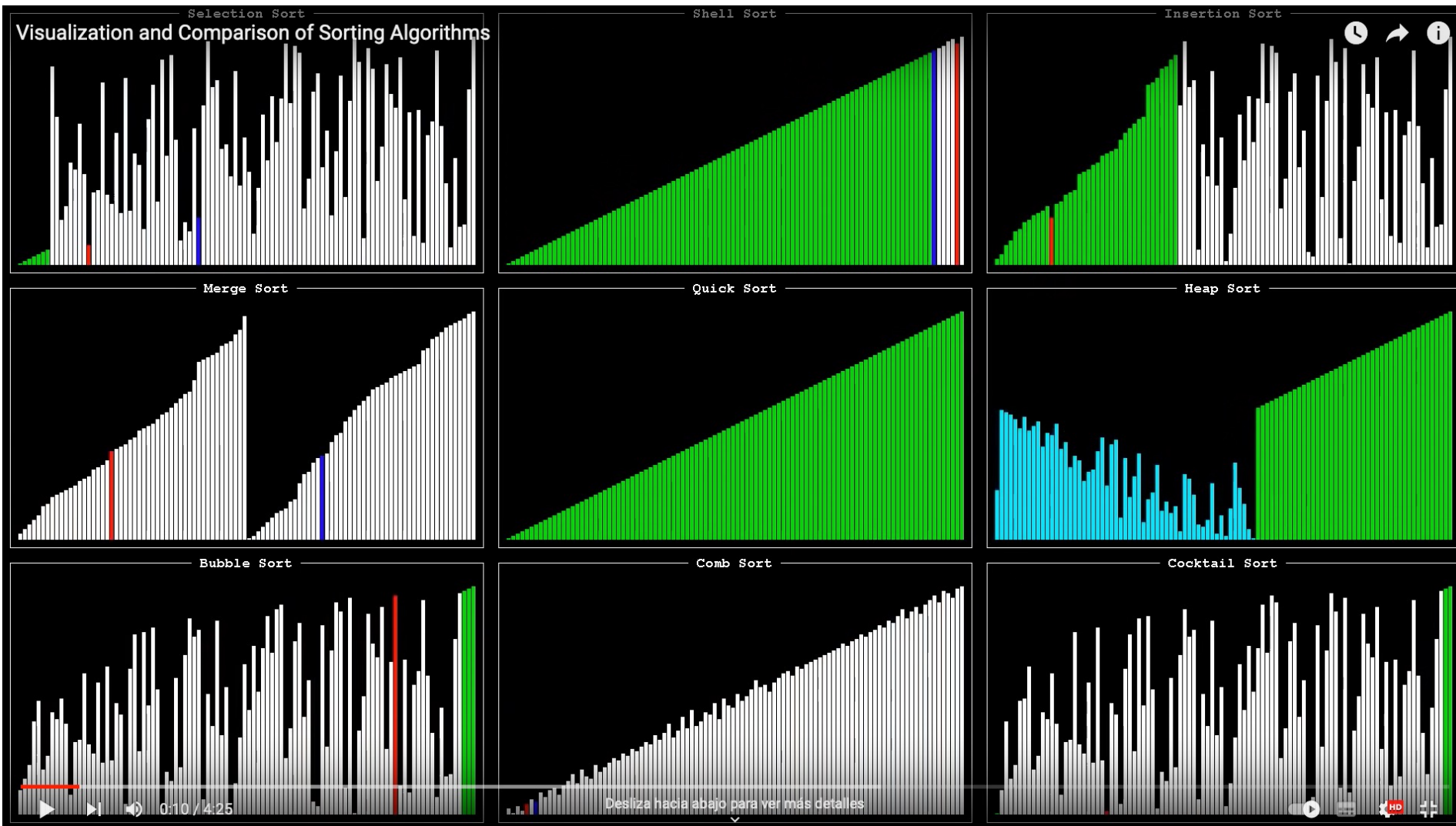
2.3.2. Multiplicación rápida de matrices.

- 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 $\rightarrow O(n^3)$
- V. Strassen (1969) $\rightarrow O(n^{2.807})$
- V. Pan (1984) $\rightarrow O(n^{2.795})$
- D. Coppersmith y S. Winograd (1990) $\rightarrow O(n^{2.376})$
- ...
- >> V. Vassilevska (2014) $\rightarrow O(n^{2.376})$

2.3.3. Métodos de ordenación rápidos ($n \log(n)$)

- Cuando pensamos en ordenar una serie de elementos, se nos ocurren seguramente algoritmos intuitivos como:
 - Selección (ir eligiendo de los restantes el siguiente mayor)
 - Inserción (coger el siguiente e insertarlo ordenado)
- Ambos tardan un t de orden n^2 .
- Existen otros algoritmos cuadráticos como:
 - Burbuja (bubble sort)
 - Sacudida (shaker sort o cocktail sort)
 - (...)
- Afortunadamente se han conseguido algoritmos de ordenación más rápidos, con orden $n \log(n)$, como los que veremos en este apartado:
 - Mergesort
 - Quicksort

2.3.3. Métodos de ordenación rápidos ($n \log(n)$)



<https://www.youtube.com/watch?v=ZZuD6iUe3Pc>

2.3.3. Métodos de ordenación rápidos ($n \log(n)$)

- La **ordenación por mezcla (mergesort)** es un buen ejemplo de divide y vencerás.
- Para ordenar los elementos de un array de tamaño n :
 - *Dividir*: el array original es dividido en dos trozos de tamaño igual (o lo más parecidos posible), es decir $\lceil n/2 \rceil$ y $\lfloor n/2 \rfloor$.
 - Resolver recursivamente los subproblemas de ordenar los dos trozos.
 - *Pequeño*: si tenemos un solo elemento, ya está ordenado.
 - *Combinar*: combinar dos secuencias ordenadas. Se puede conseguir en $O(n)$.

A

1	2	3	4	5	6	7	8
5	2	7	9	3	2	1	6

2.3.3. Métodos de ordenación rápidos ($n \log(n)$)

operación MergeSort (i, j: entero)

si *Pequeño* (i, j) entonces

OrdenaciónDirecta (i, j)

sino

$s := (i + j) \text{ div } 2$

 MergeSort (i, s)

 MergeSort (s+1, j)

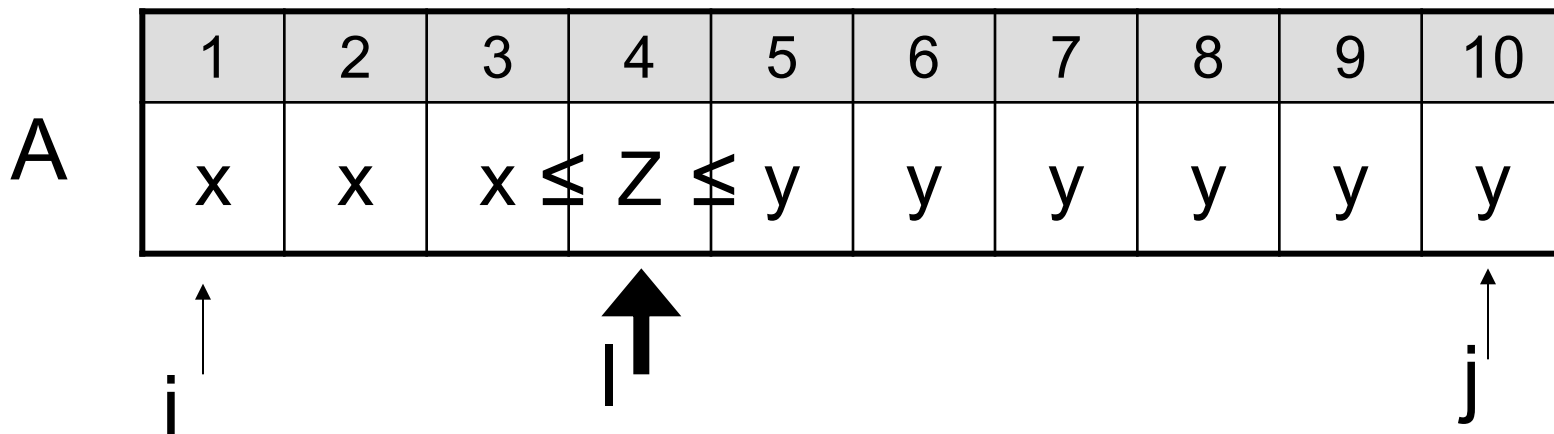
Combinar (i, s, j)

finsi

- ¿Cómo es la operación *Combinar*?
- ¿Cuánto es el tiempo de ejecución?
- **Ojo:** para tamaños pequeños es mejor un método de ordenación directa → Usar un caso base mayor que 1...

2.3.3. Métodos de ordenación rápidos ($n \log(n)$)

- La **ordenación rápida (quicksort)** también se basa en la técnica divide y vencerás.
 - *Dividir*: el array (i..j) es dividido usando un procedimiento **Pivote**, que devuelve un entero l entre (i, j), tal que:
$$A[i_a] \leq A[l] \leq A[j_a], \text{ para } i_a = i..l-1; j_a = l+1..j.$$
 - Ordenar recursivamente los trozos (i..l-1) y (l+1..j).
 - *Pequeño*: si tenemos un solo elemento, entonces ya está ordenado.
 - *Combinar*: no es necesario realizar ninguna operación.



2.3.3. Métodos de ordenación rápidos ($n \log(n)$)

operación QuickSort (i, j: entero)

si $i \geq j$ entonces

{ Ya está ordenado, no hacer nada }

sino

Pivote (i, j, l)

QuickSort (i, l-1)

QuickSort (l+1, j)

finsi

- Aunque no hay coste de combinar los resultados, la llamada a *Pivote* tiene un coste $O(n)$.
- Las particiones no tienen por qué ser de tamaño $n/2...$

2.3.3. Métodos de ordenación rápidos ($n \log(n)$)

operación Pivote (*i, j: entero; var l: entero*)

var *p: tipo* *{ p es el pivote, del tipo del array }*
 k: entero

p := A[i] *{ se toma como pivote el primer elemento }*

k := i

l := j+1

repetir *k := k+1 hasta (A[k] > p) or (k ≥ j)*

repetir *l := l-1 hasta (A[l] ≤ p)*

mientras *k < l hacer*

intercambiar (k, l)

repetir *k := k+1 hasta (A[k] > p)*

repetir *l := l-1 hasta (A[l] ≤ p)*

finmientras

intercambiar (i, l)

2.3.3. Métodos de ordenación rápidos ($n \log(n)$)

Tiempo de ejecución de la ordenación rápida:

- **Mejor caso.**

Todas las particiones son de tamaño similar, $n/2$.

$$t(n) = 2 \cdot t(n/2) + b \cdot n + c \in \Omega(n \cdot \log n)$$

- **Peor caso.**

Se da cuando el array está ya ordenado. En ese caso una partición tiene tamaño 0 y la otra $n-1$.

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

- **Caso promedio.**

Se puede comprobar que $t(n) \in O(n \cdot \log n)$

2.3.4. El problema de selección.

- 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.


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?

2.3.4. El problema de selección.

- **Forma sencilla** de resolver el problema de selección:
 1. Ordenar T y devolver el valor T[s]
 2. Esto requeriría $\Theta(n \log n)$
- ¡Igual de complejo que una ordenación completa!
- Pero sólo necesitamos “ordenar” uno...
- **Idea:** usar el procedimiento **pivote** de QuickSort.

1	2	3	4	5	6	7	8	9	10
x	x	$x \leq$	$Z \leq$	y	y	y	y	y	y



2.3.4. El problema de selección.

- Utilizando el procedimiento **Pivote** podemos resolverlo en $O(n)$...

Selección (T: array [1..n] de entero; s: entero)

var i, j, l: entero

i := 1

j := n

repetir

 Pivote (i, j, l)

si s < l **entonces**

 j := l-1

sino si s > l **entonces**

 i := l+1

hasta l == s

devolver T[l]

2.3.4. El problema de selección.

- Se trata de un **algoritmo de reducción**: el problema es descompuesto en un solo subproblema de tamaño menor.
- El procedimiento es no recursivo.
- En el **mejor caso**, el subproblema es de tamaño $n/2$:
$$t(n) = t(n/2) + a \cdot n; \quad t(n) \in \Omega(n)$$
- En el **peor caso** (array ordenado) el subproblema es de tamaño $n-1$:
$$t(n) = t(n-1) + a \cdot n; \quad t(n) \in O(n^2)$$
- En el **caso promedio** el tiempo del algoritmo es un $O(n)$.

2.3.5. Ejemplo con secuencias

- Dada una secuencia S de n números, encontrar:
 - El mayor/menor...
 - La subsecuencia más larga que cumple:
 - Ascendente
 - Simétrica
 - Pares
 - Potencias del primer elemento
 - Números en posiciones pares son pares
 - ...

2.3.6. Ejemplo de DyV 2D – El derbi

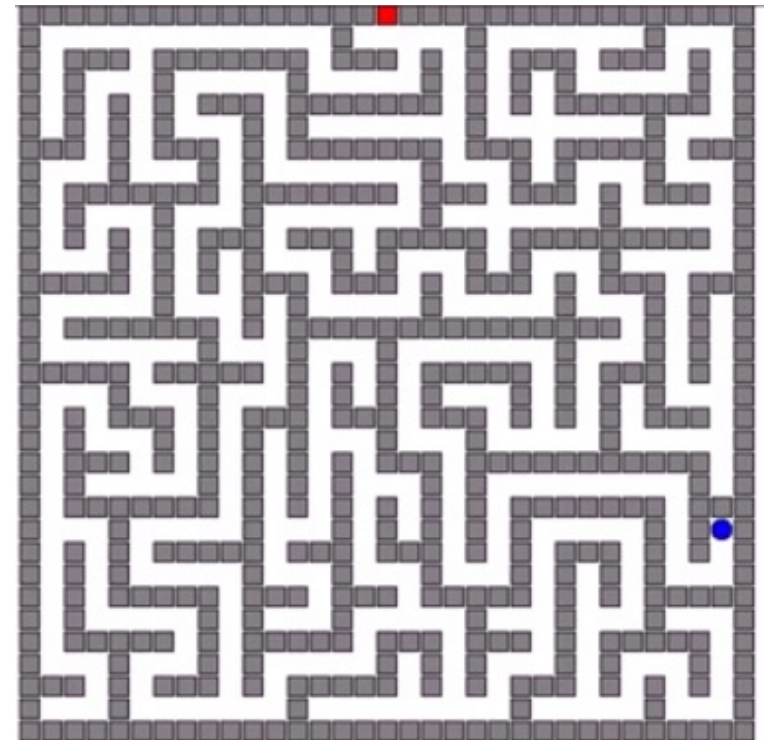
- Barrio en forma de matriz de caracteres M .
- Cada celda (i, j) es una casa
- $M[i, j]$ = casa del equipo 'A', 'B', o neutra ('N')
- Adyacencia conflictiva: $M[i, j] = 'A'$ y $M[k, l] = 'B'$ con:
 - $i=k$ y $j=l-1$ ó $j=l+1$
 - $j=l$ y $i=k-1$ ó $i=k+1$
- Objetivo: contar adyacencias conflictivas por DyV
- (...)

2.3.7. DyV en búsqueda/optimización

- ¿Cómo encontramos el tesoro?



Versión sin colorines...



A.E.D.

Tema 2. Divide y vencerás.

- El **backtracking** (retroceso o vuelta atrás) es una técnica general de resolución de problemas, aplicable en problemas de **optimización**, juegos y otros tipos.
- Por cierto, ¿qué es eso de *problema de optimización*?
- En todos los problemas hay **restricciones**, condiciones que determinan si tenemos o no una *solución*.
- Si además elegimos la **mejor solución: optimización**
- ¿Mejor? Se define **función objetivo**: $FO(\text{solución})$, y se busca solución que genera su menor (mayor) valor.

- El backtracking realiza una búsqueda **exhaustiva** y **sistemática** en el *espacio de soluciones*
>> Suele resultar muy **ineficiente**.
- Se puede entender como opuesto a avance rápido:
 - **Backtracking**: probar todas las combinaciones (con constantes vueltas hacia atrás).
 - **Avance rápido**:
Laberinto con oso hambriento...
tomo decisiones y no miro atrás
(no las reconsidero).
- Se puede ver como un **BT sin vuelta atrás**, las decisiones tomadas no se deshacen, así que la **elección tomada en cada paso es clave**.



PROFE, TENGO UNA PREGUNTA



¿Y qué tiene que ver esto con **Divide y Vencerás?**

- Podríamos plantear el mejor camino como una recursión:
 - >> mejor camino = el más corto entre:
 - 1) ir por la izquierda
 - 2) ir por la derecha
- ¿Y esto es eficiente? ¿No se repiten cálculos?
 - >> Programación dinámica evitará repetir cálculos
 - >> BT iterativo evitará recursión y algunos subproblemas

2. 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 → Descartar 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?