

Fundamentos de análisis y diseño de algoritmos

Divide y vencerás

Divide y vencerás

Introducción

Ejemplos

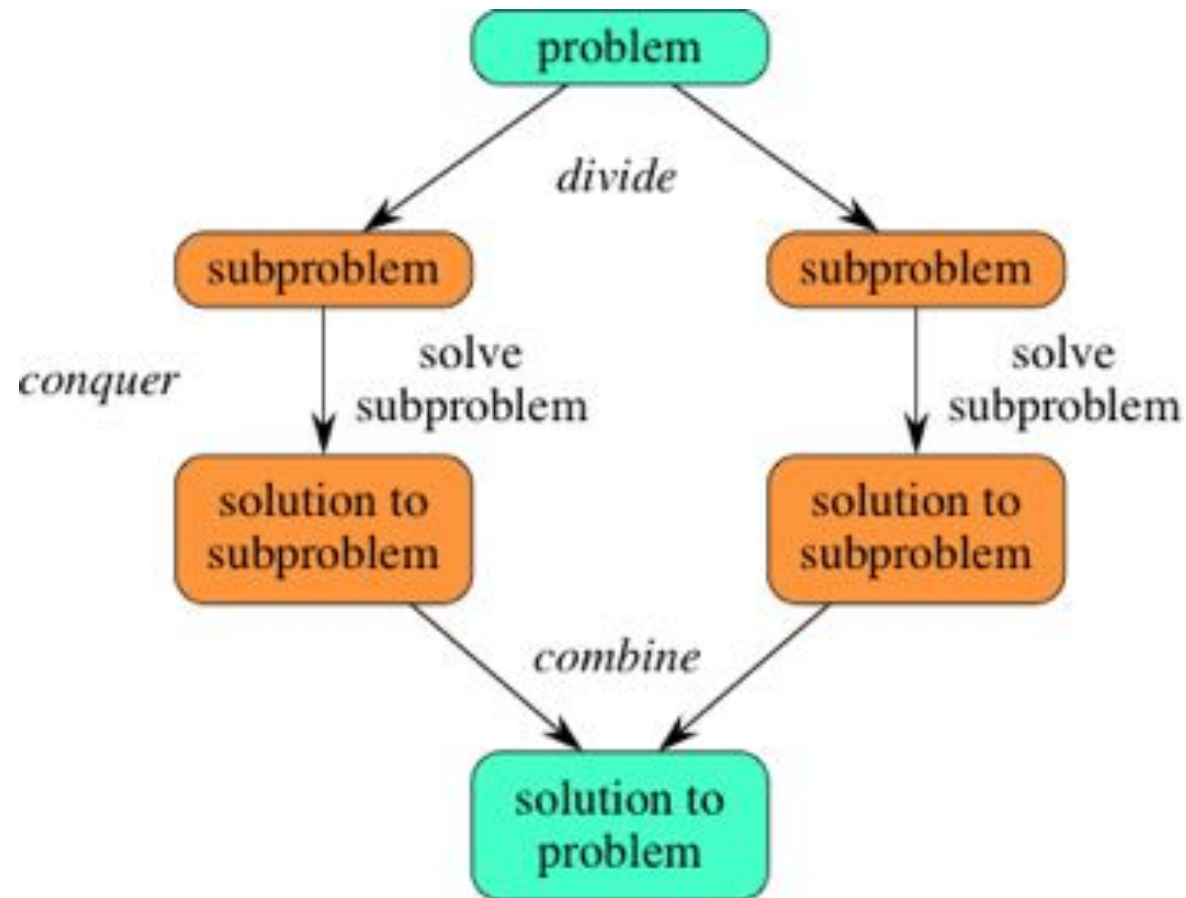
Cálculo de complejidad de algoritmos recursivos

Divide y vencerás

Considera recursividad

- **Dividir** el problema en subproblemas. Dividir hasta problema trivial, es aquel que tiene solución inmediata
- **Conquistar** los subproblemas (solucionarlos recursivamente). El enfoque es que al solucionar los subproblemas, solucionamos el general
- **Combinar** las soluciones de los subproblemas para crear la solución al problema original

Divide y vencerás



Divide y vencerás

Algoritmo MergeSort

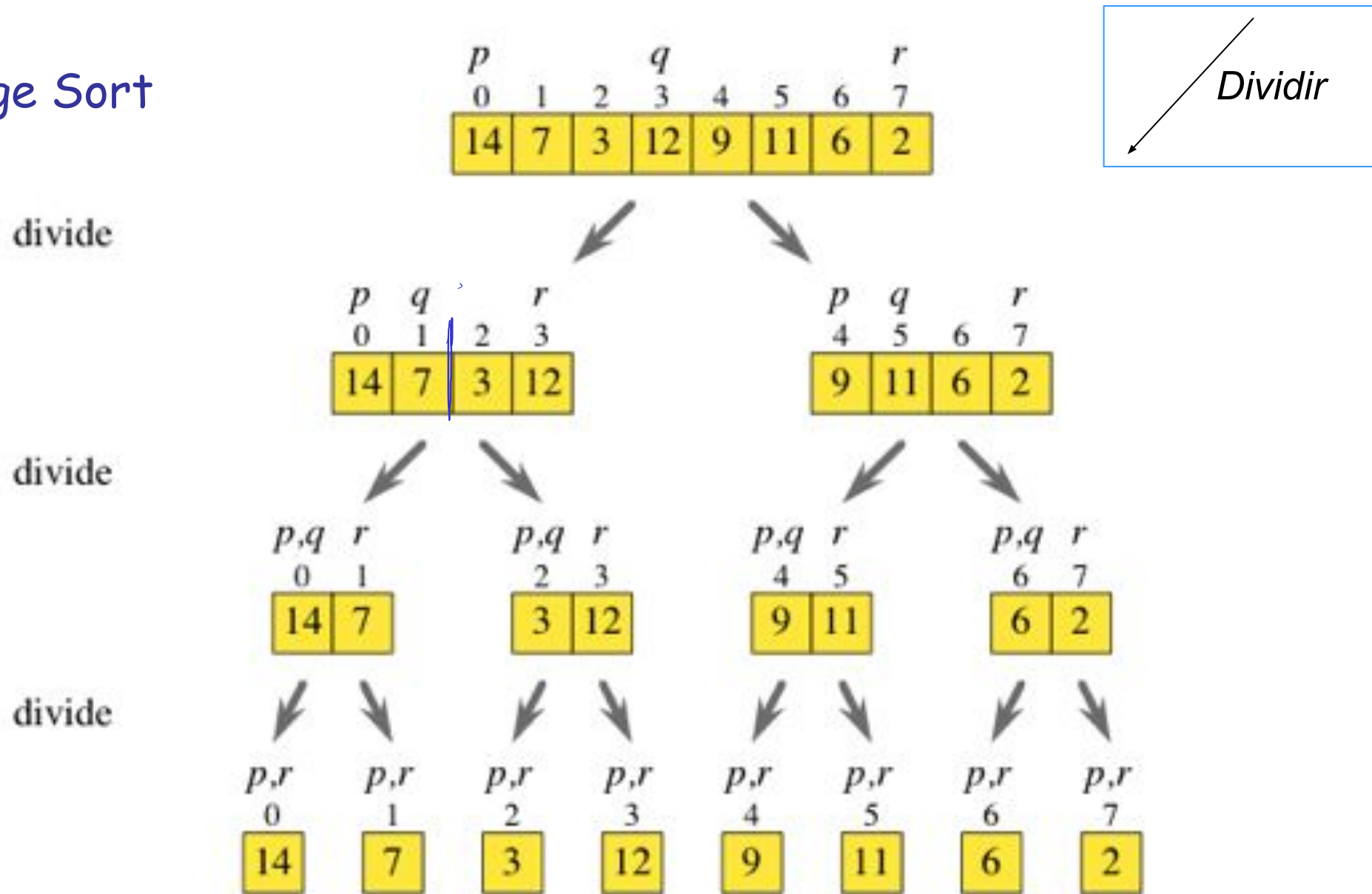
El caso trivial del algoritmo es ordenar una lista ~~vacía~~ (está ordenado por defecto)
con un elemento

- **Dividir** Divida una lista de n elementos, en dos listas de $n/2$ elementos cada una
- **Conquistar** Ordene dos subsecuencias recursivamente
- **Combinar** Mezcle dos lista ordenadas para producir una lista ordenada

valor

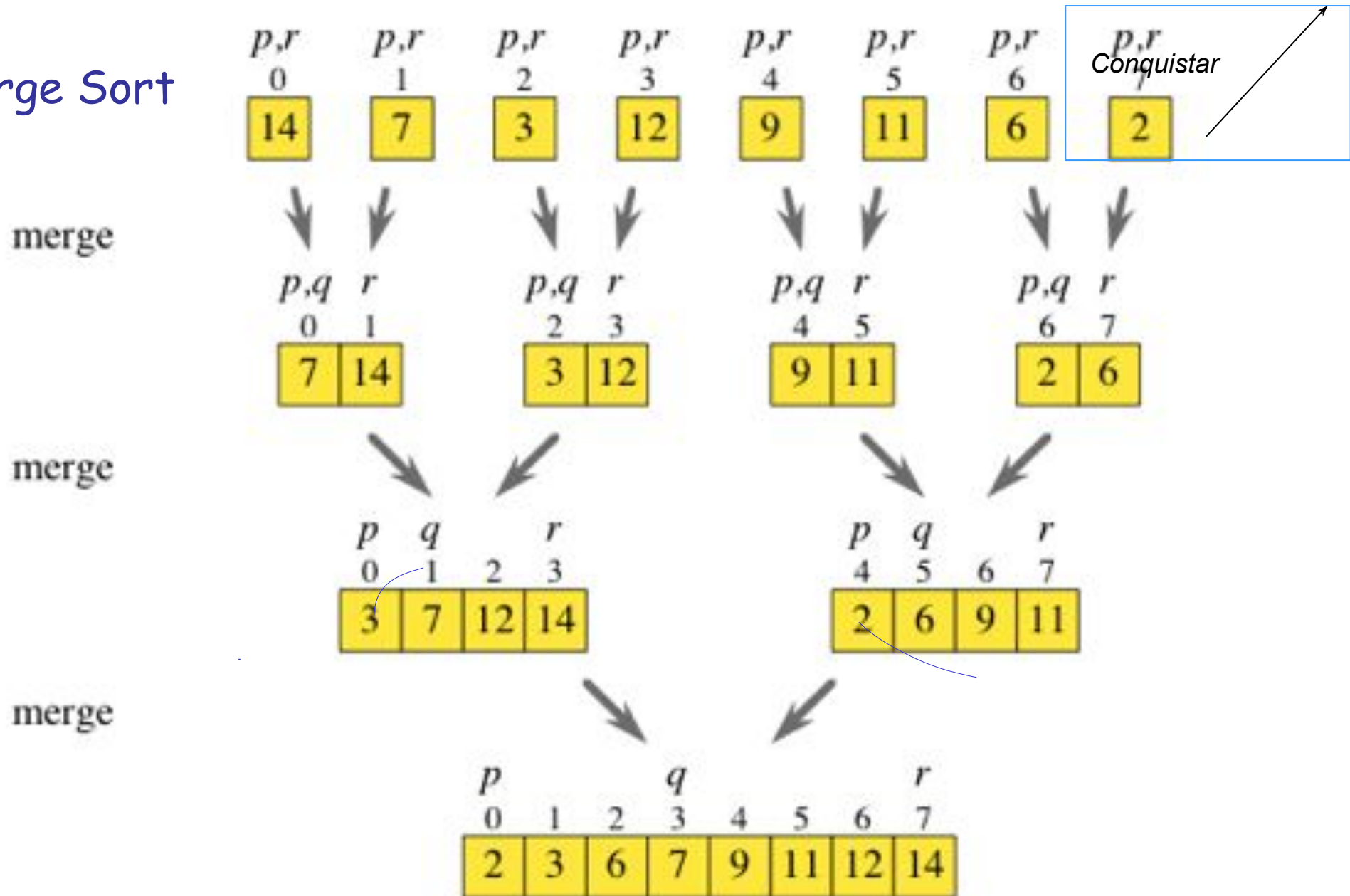
Divide y vencerás

Merge Sort



Divide y vencerás

Merge Sort



p	q		r	
0	1	2	3	
3	7	12	14	∞
			\uparrow	
			i	

p	q		r	
4	5	6	7	
2	6	9	11	∞
			\uparrow	
			j	

2	3	6	7	9	11	12	14
---	---	---	---	---	----	----	----

Divide y vencerás

Algoritmo MergeSort

MERGE(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

Divide y vencerás

Algoritmo MergeSort

MERGE-SORT(A, p, r)

1 **if** $p < r$

2 $q = \lfloor (p + r)/2 \rfloor$

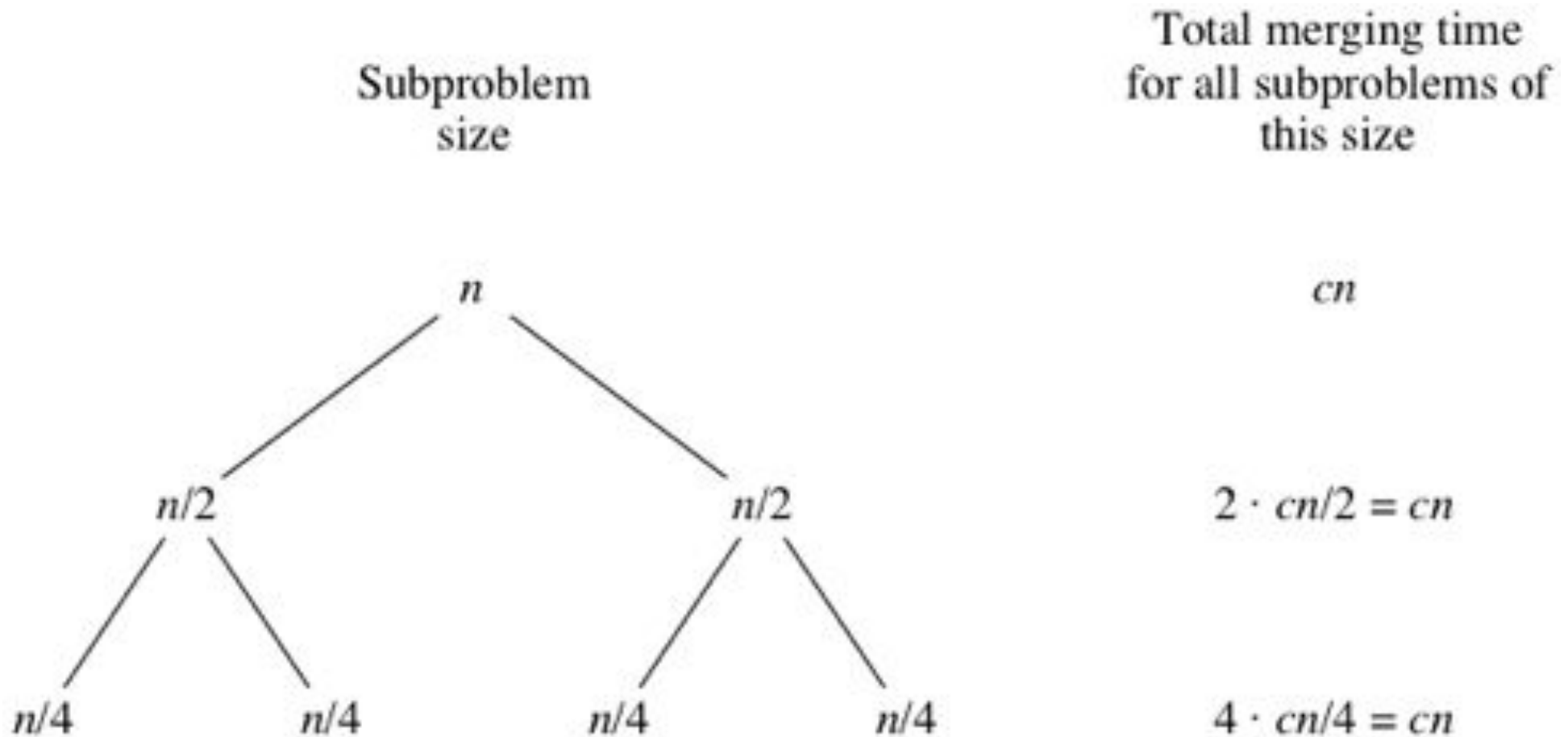
3 MERGE-SORT(A, p, q)

4 MERGE-SORT($A, q + 1, r$)

5 MERGE(A, p, q, r)

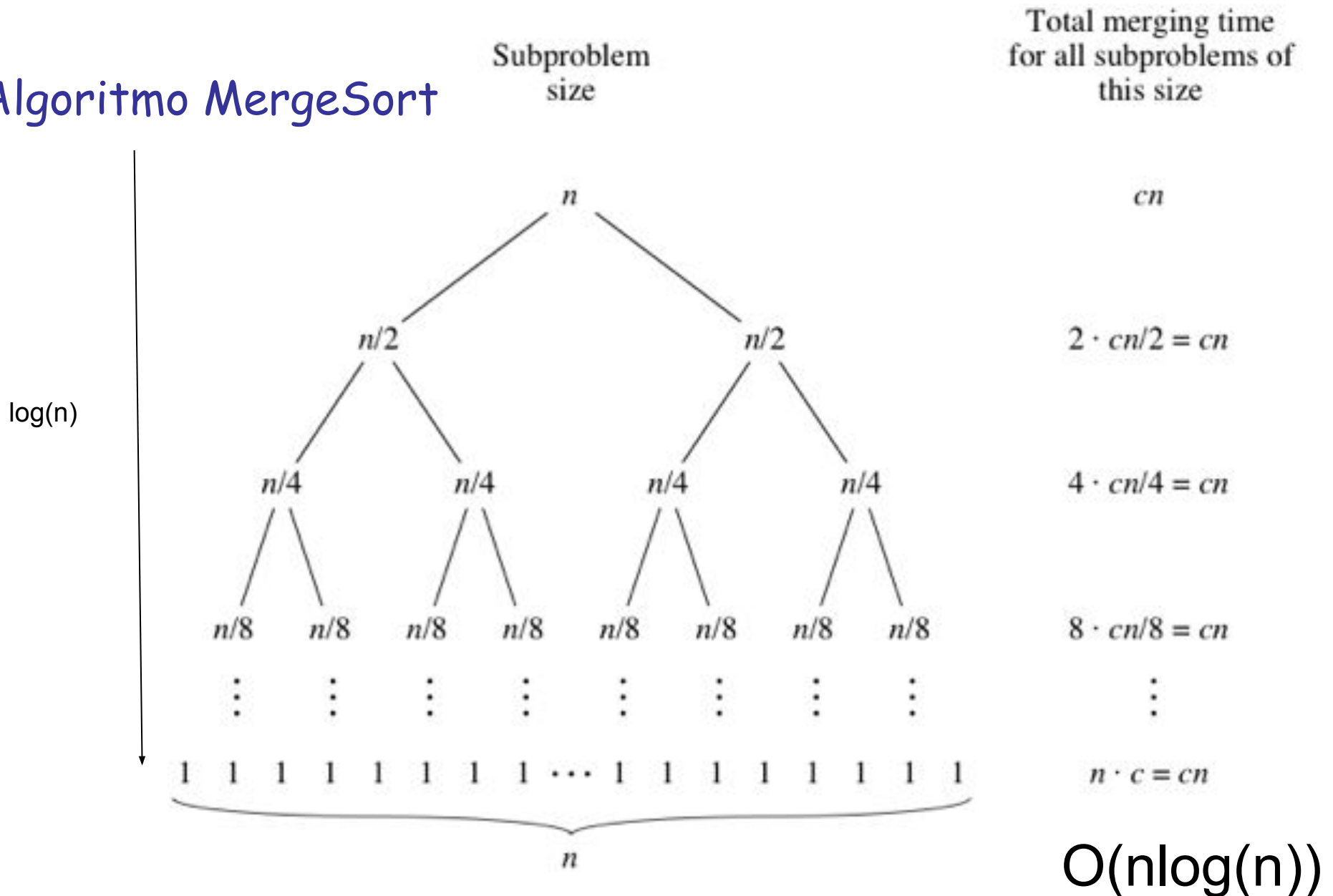
Divide y vencerás

Algoritmo MergeSort



Divide y vencerás

Algoritmo MergeSort



Divide y vencerás

Buscar el máximo de una lista $\sim O(n)$

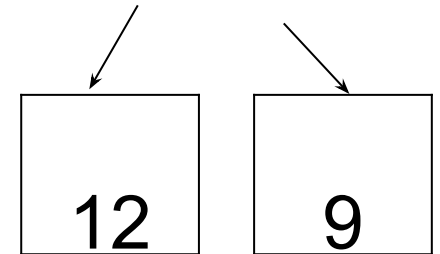
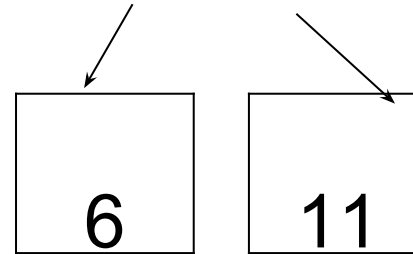
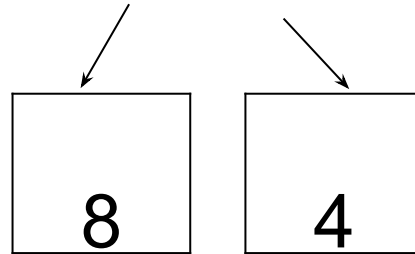
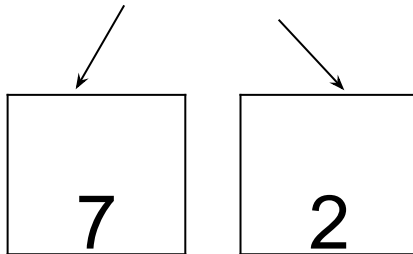
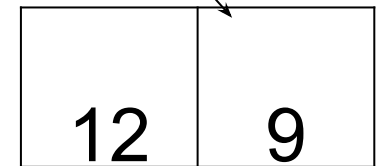
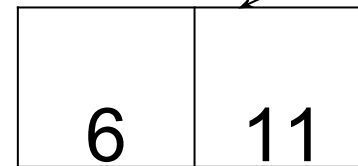
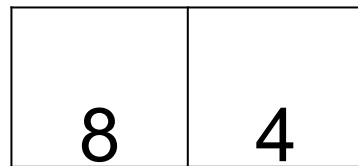
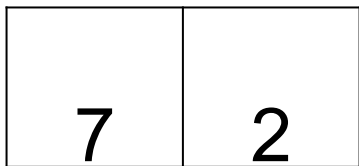
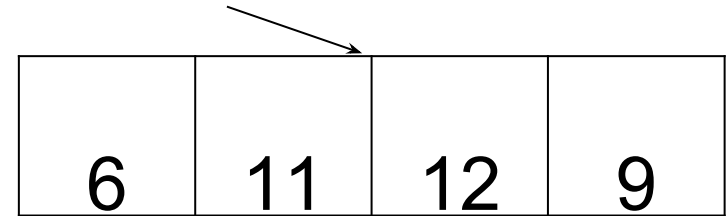
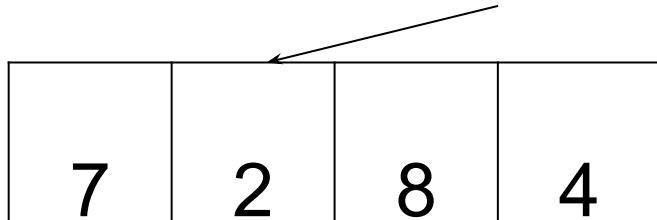
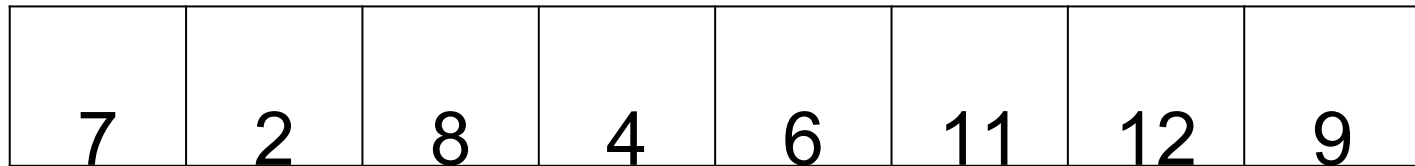
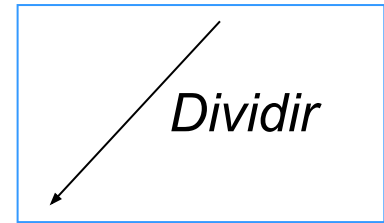
Dividir divide la lista a la mitad sucesivamente,

Conquistar Llegar al caso trivial de tener un elemento. Este será el mayor de la lista.

Combinar Combinar sucesivamente las listas, dejando como primer elemento el mayor. Así, al llegar a la lista completa el primer elemento será el mayor

Divide y vencerás

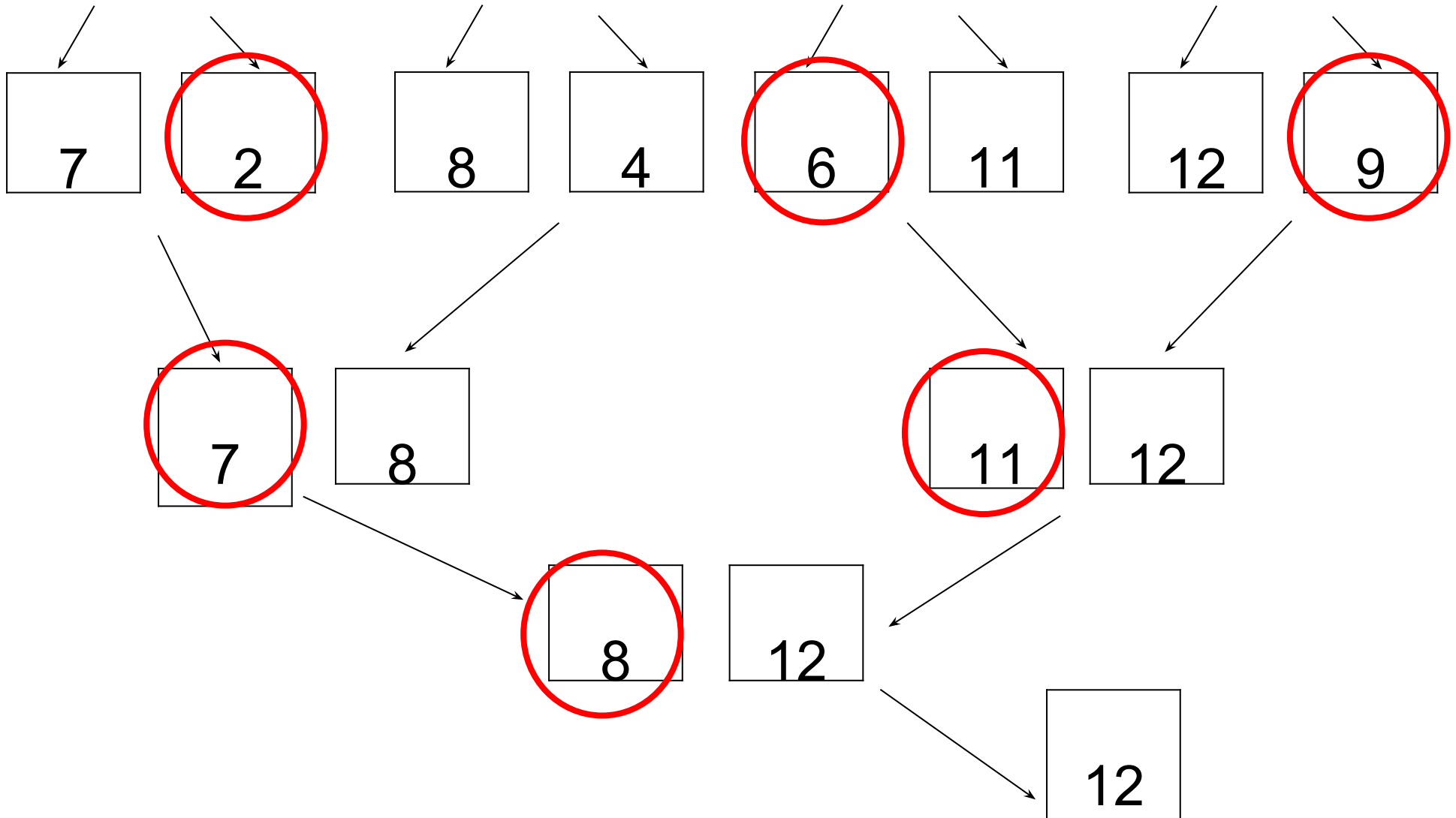
Buscar el máximo de una lista



Divide y vencerás

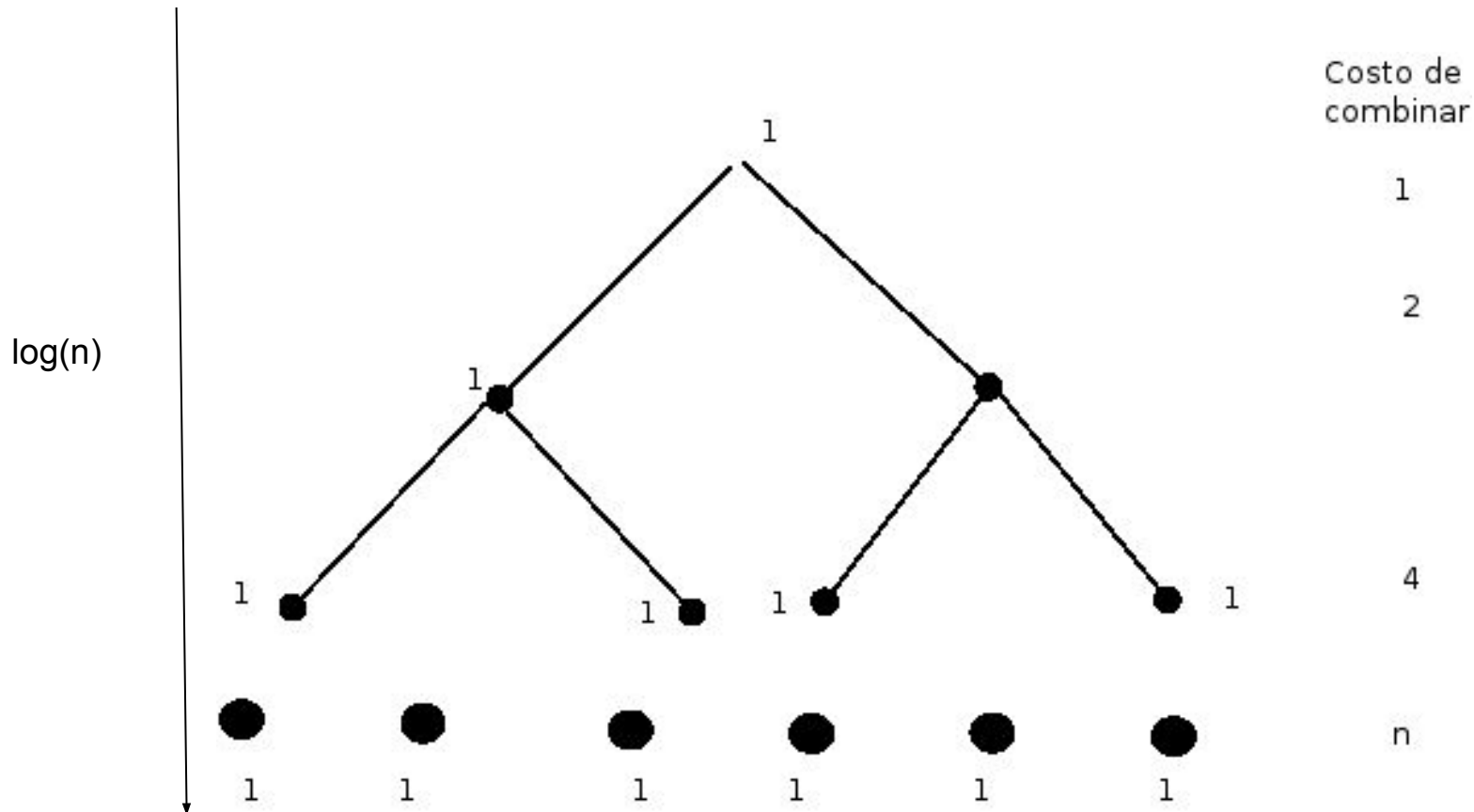
Buscar el máximo de una lista

Conquistar



Divide y vencerás

Buscar el máximo de una lista



$$\sum_{i=0}^{\log(n)} 2^i = \frac{2^{\log(n)+1} - 1}{2 - 1} = O(n)$$

Divide y vencerás

Buscar el máximo de una lista

```
DevolverMaximo(a, b)
```

```
    Si a > b
```

```
        Retornar a
```

```
    Sino
```

```
        Retornar b
```

```
FinProc
```

```
BuscarMaximo(A[], r, q)
```

```
    Si A.size == 1
```

```
        Return A[1]
```

```
    Sino
```

```
        index = Piso(A.size/2);
```

```
        a = BuscarMaximo(A, r, index)
```

```
        b = BuscarMaximo(A, index+1, q)
```

```
        Retornar DevolverMaximo(a, b)
```

```
FinProc
```

Divide y vencerás

Busqueda binaria

Suponga que la lista está ordenada y que busca un elemento x

Dividir divide la lista a la mitad,

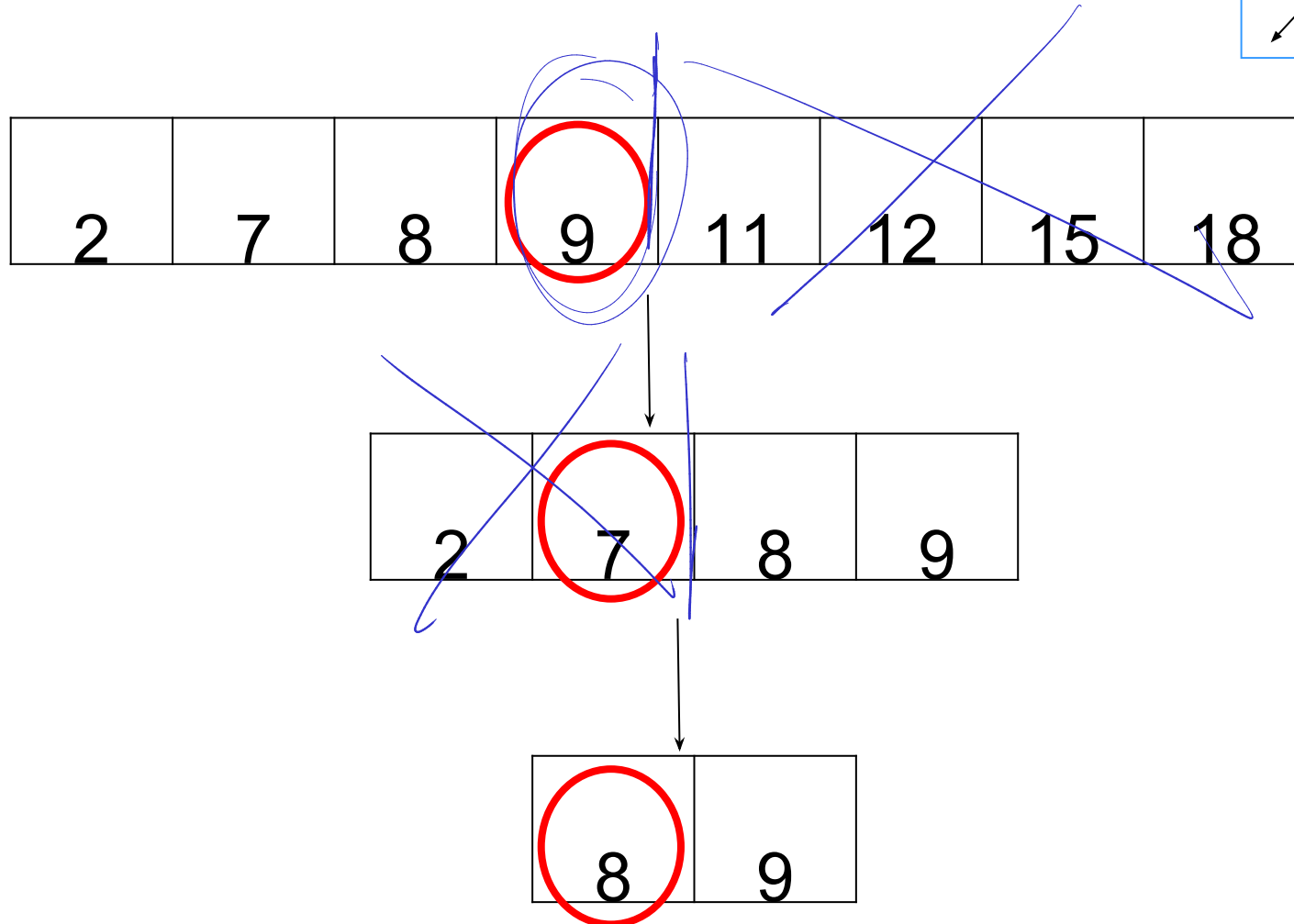
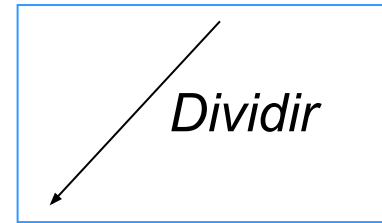
Conquistar Examine el último elemento de la primera lista y el primero de la segunda. Si el primero es menor o igual que x , repita dividir sobre la primera lista. En caso contrario hágalo sobre la segunda.

Combinar El espacio de búsqueda irá reduciéndose hasta encontrar el elemento.

¿Cual es el costo computacional?. ¿Si la lista está desordenada, vale la pena ordenar y aplicar este algoritmo?

Divide y vencerás

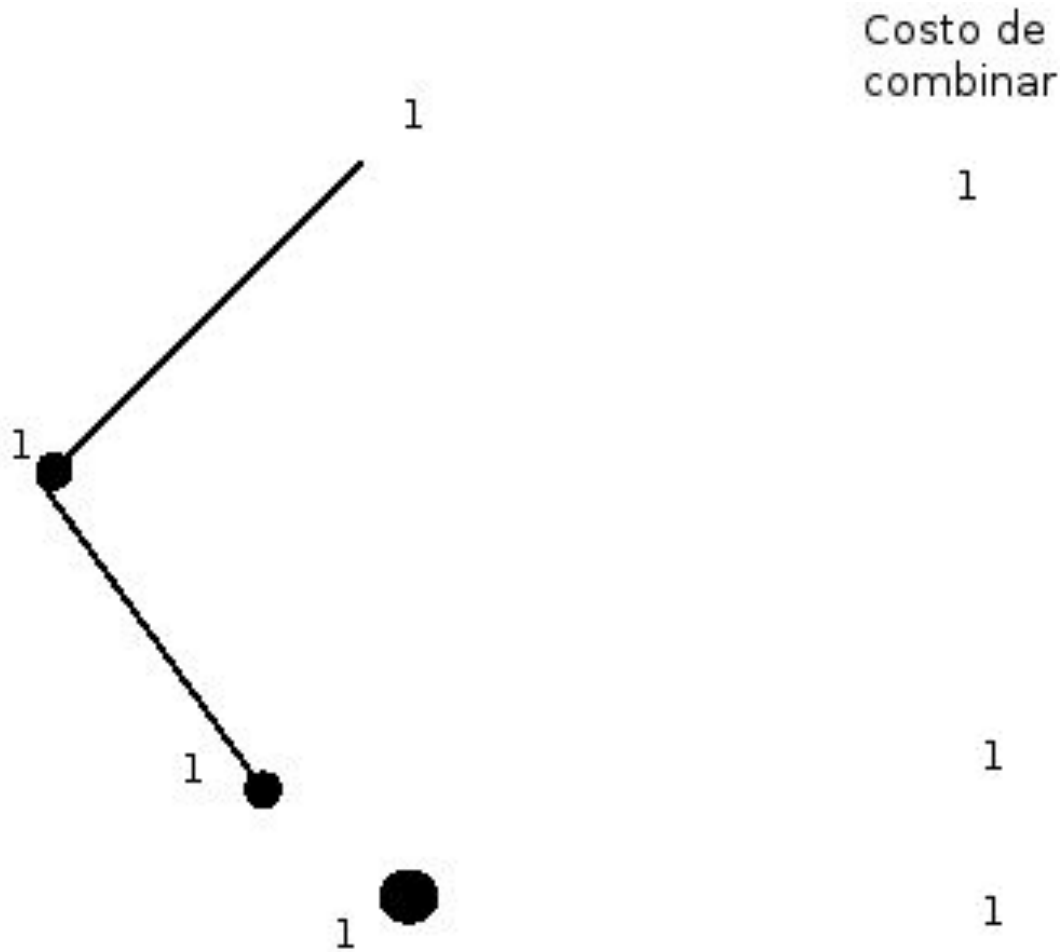
Busqueda binaria Buscamos a 8.



Combinar no es necesario, ya que al buscar de esta forma nos queda un elemento si tenemos éxito o ninguno si no lo tenemos

Divide y vencerás

Busqueda binaria



$$\sum_{i=0}^{\log(n)} 1 = \log(n) = O(\log(n))$$

Divide y vencerás

Busqueda binaria

```
BusquedaBinaria(A[], r, q, x)
    Si A.size == 0
        imprimir "No se encuentra el número"
    Si A[r] == x
        Retorne x
    Sino
        index = Piso(A.size/2);
        Si A[index] <= x
            Retorne BusquedaBinaria(A, r, index, x)
        Sino
            Retorne BusquedaBinaria(A, index+1, q, x)

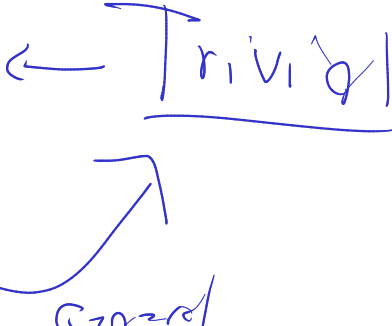
endProc
```

Divide y vencerás

Cálculo de complejidad de algoritmos recursivos

Análisis de algoritmos recursivos

Un algoritmo recursivo tiene las siguientes partes:

- 1) Una condición de parada 
- 2) Un llamado recursivo

El análisis de estos algoritmos lo realizaremos analizando

- 1) Su complejidad en un llamado
- 2) Cómo es el llamado recursivo y cómo cambia la entrada a medida que se realizan los llamados
- 3) Cómo es la forma de la entrada para llegar a la condición de parada

Análisis de algoritmos recursivos

Para el caso de la estrategia Divide y Vencerás, se debe considerar

- $T(n)$ Complejidad de solucionar el problema para entrada tamaño n
- $D(n)$ Es el costo de dividir un problema de tamaño n
- $C(n)$ Es el costo de combinar los subproblemas
- a es el número de subproblemas que generamos
- b es la razón a la cual dividimos el problema original $n \leq c$

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{si } n > c \end{cases}$$

Análisis de algoritmos recursivos

Para el caso de merge sort

- $a = 2$
- $b = 2$
- $C(n) = n$ // Requiere ordenar sublistas
- $D(n) = 1$ // Es pegar dos listas ordenadas

$$T(n) = 2T\left(\frac{n}{2}\right) + n + 1$$

Por método del maestro.

$$f(n) = n + 1, \log_b(a) = \log_2(2) = 1$$

Entra en segundo caso. $\Theta(n) = n + 1$, entonces

$$T(n) = \Theta(n^{\log_b(a)} \log(n)) = \Theta(n \log(n))$$

Análisis de algoritmos recursivos

Ejemplo, pensemos en este algoritmo para calcular la serie de Fibunnaci para un número (n) dado

Recuerda:

$f(n) = f(n-1) + f(n-2), f(0) = 1, f(1) = 1$
fibunnaci(n)

Si $n = 0$ retorne 1

Sino si $n = 1$ retorne 2

Sino fibunnaci(n - 1) + fibunnaci(n - 2)

Análisis de algoritmos recursivos

Si

Recuerda:

$$f(n) = f(n-1) + f(n-2), f(0) = 1, f(1) = 1$$

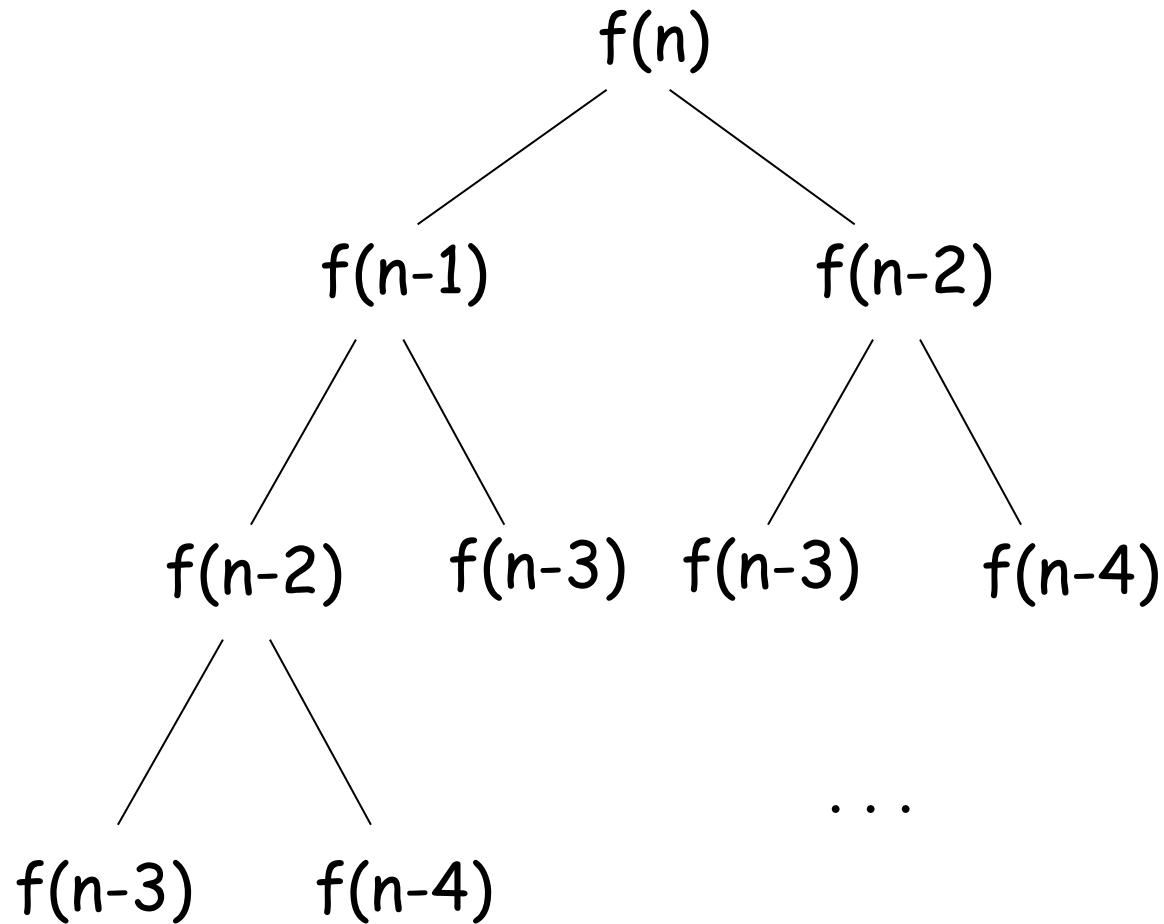
fibunnaci(n)

Si $n = 0$ retorne 1

Sino si $n = 1$ retorne 2

Sino fibunnaci($n - 1$) + fibunnaci($n - 2$)

Análisis de algoritmos recursivos



Análisis de algoritmos recursivos

Si observamos para cada llamado de n se realiza el llamado para $n-1$ y $n-2$, la parada se encuentra cuando $n = 0$ y $n = 1$ entonces, la complejidad de este algoritmo está dada por la relación

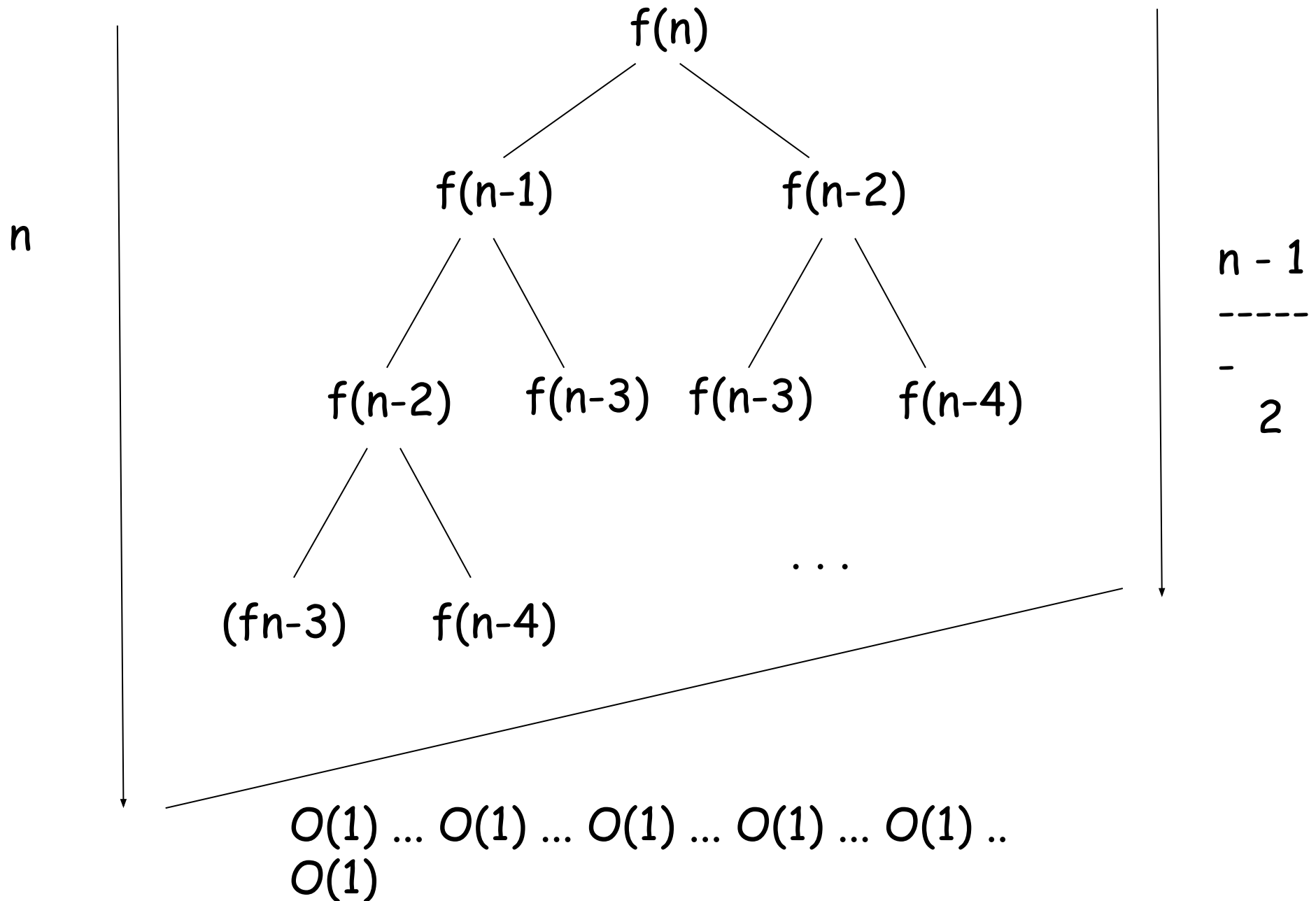
$$T(n) = T(n-1) + T(n-2) + O(1)$$

Se cuenta $O(1)$ en cada llamado debido a la verificaciones que debe realizar, las cuales son ejecutadas en **tiempo constante**

Si observa en las condiciones de parada el tiempo también es constante, entonces:

$$T(0) = O(1), T(1) = 1$$

Análisis de algoritmos recursivos



Análisis de algoritmos recursivos

Aplicamos método de sustitución obtenemos ~~$O(2^n)$~~
 $O(2^n)$

¿Se puede implementar el cálculo de la serie de Fibonacci, de tal forma obtenegamos una cota menor de complejidad?

Análisis de algoritmos recursivos

Tarea

Analizar algoritmos que tienen el siguiente comportamiento.

Algoritmo 1

Particiones: $n - 2$ y $n - 4$, complejidad cada paso $O(n)$

Algoritmo 2

Particiones: $n/2$ y $n/3$, complejidad cada paso $O(\log(n))$

Análisis de algoritmos recursivos

Tarea

Analizar el siguiente algoritmo:

Algoritmo(n)

Si $n = 0$ retorne 1

Si $n = 1$ retorne 2

Sino Si n es par retorne $n + f(\lfloor n/2 \rfloor)$

Referencias

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd ed.). The MIT Press. Chapter 4

Gracias

Próximo tema:

Estructuras de datos