Fundamentos de análisis y diseño de algoritmos

Divide y vencerás

Divide y vencerás

Introducción

Ejemplos

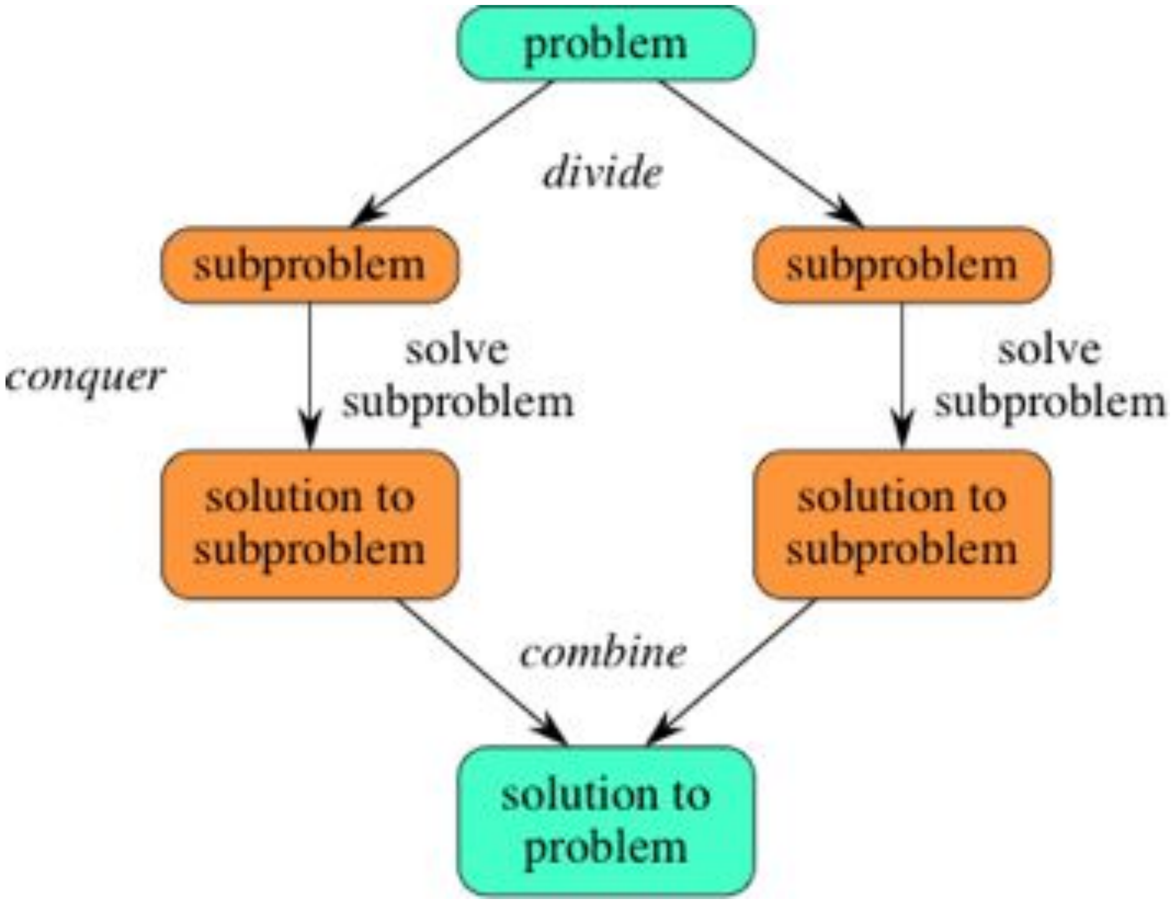
Cálculo de complejidad de algoritmos recursivos

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



Tomado de https://www.khanacademy.org, Marzo 2018

Divide y vencerás

1 $\boxed{2 \quad 3 \quad 4}$ 8 10 $\boxed{12 \quad 14}$ 16

Algoritmo MergeSort

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

[ ] [1]

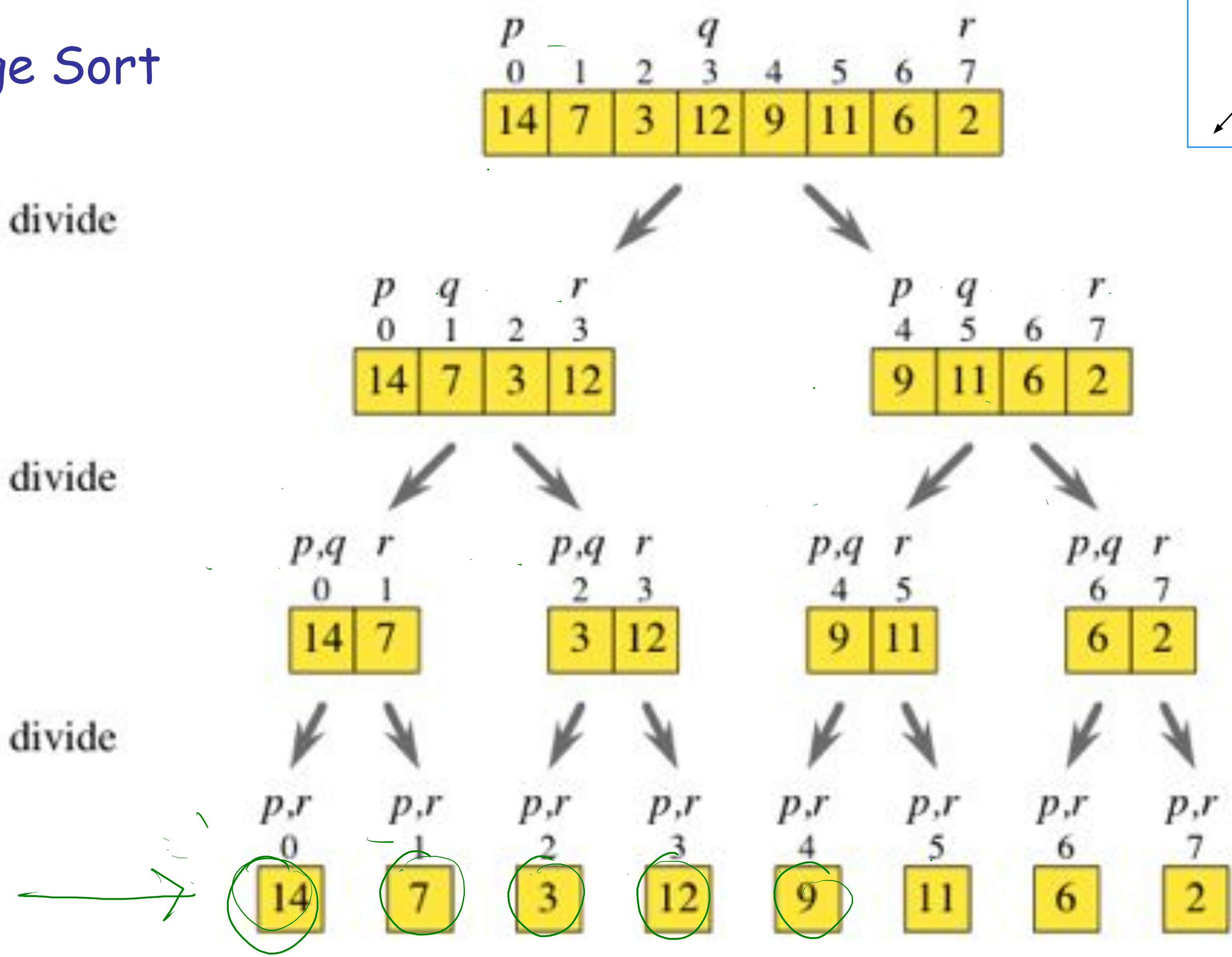
•Dividir Divida una lista de n elementos, en dos listas de n/2 elementos cada una

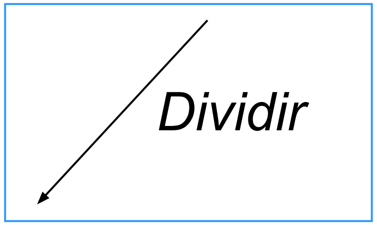
•Conquistar Ordene dos subsecuencias recursivamente

•Combinar Mezcle dos lista ordenadas para producir una lista ordenada

Divide y vencerás

Merge Sort

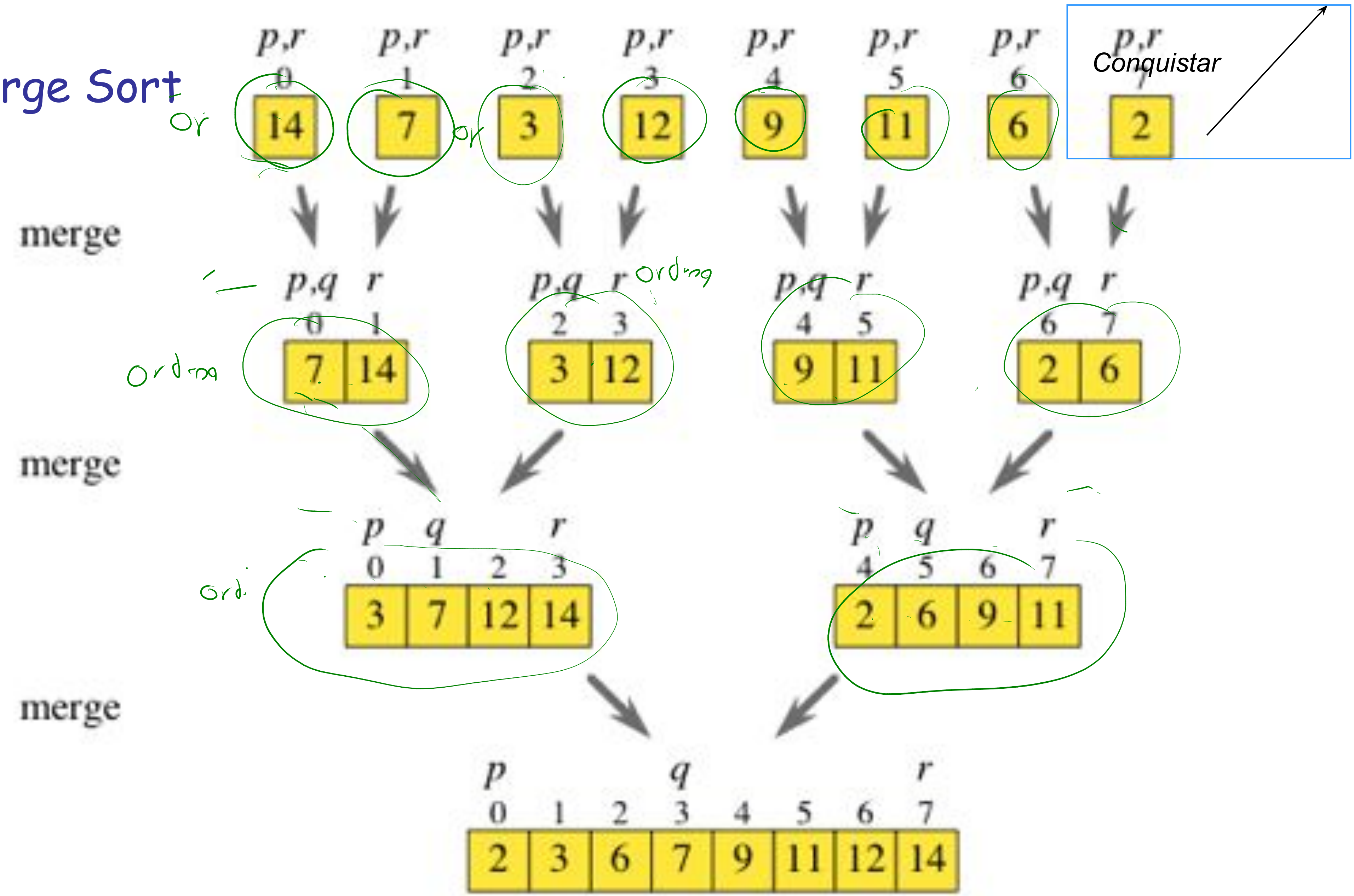




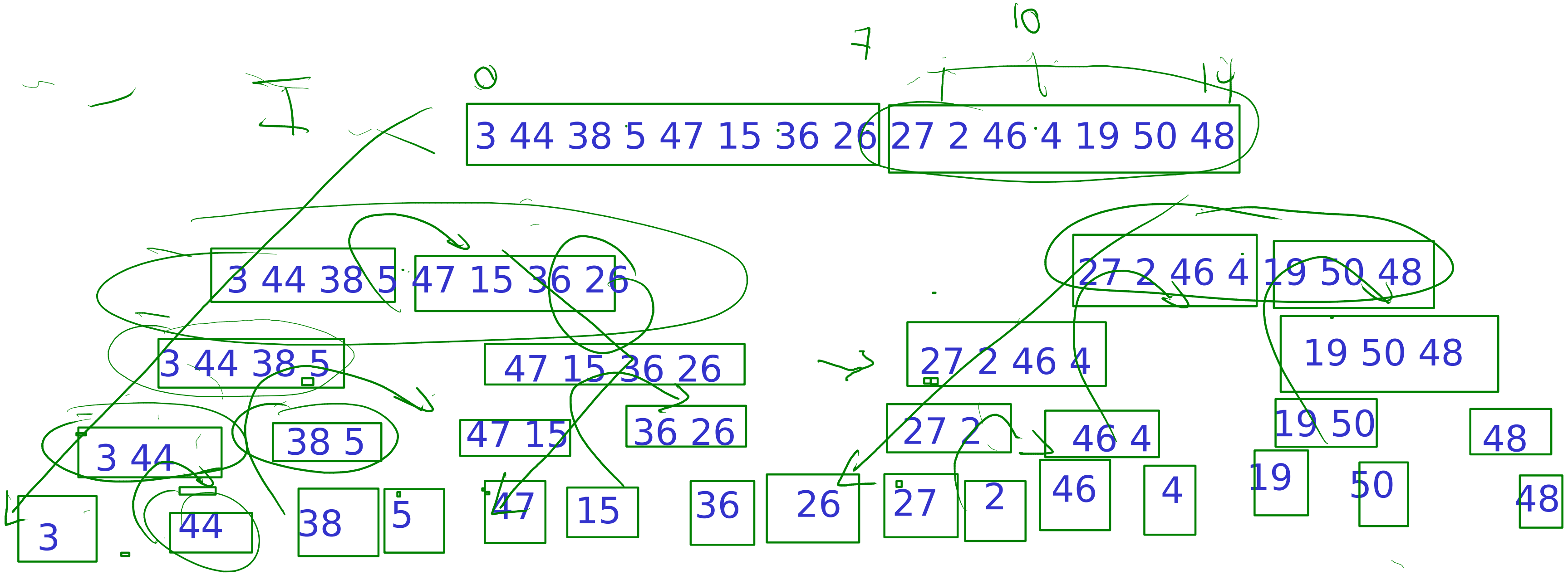
Tomado de https://www.khanacademy.org, Marzo 2018

Divide y vencerás

Merge Sort



Tomado de https://www.khanacademy.org, Marzo 2018



✱ mergesort([3.. 48])

✱ I mergeSort([3..26])

x mergsort([3..5])

x mergeSort([3 44])

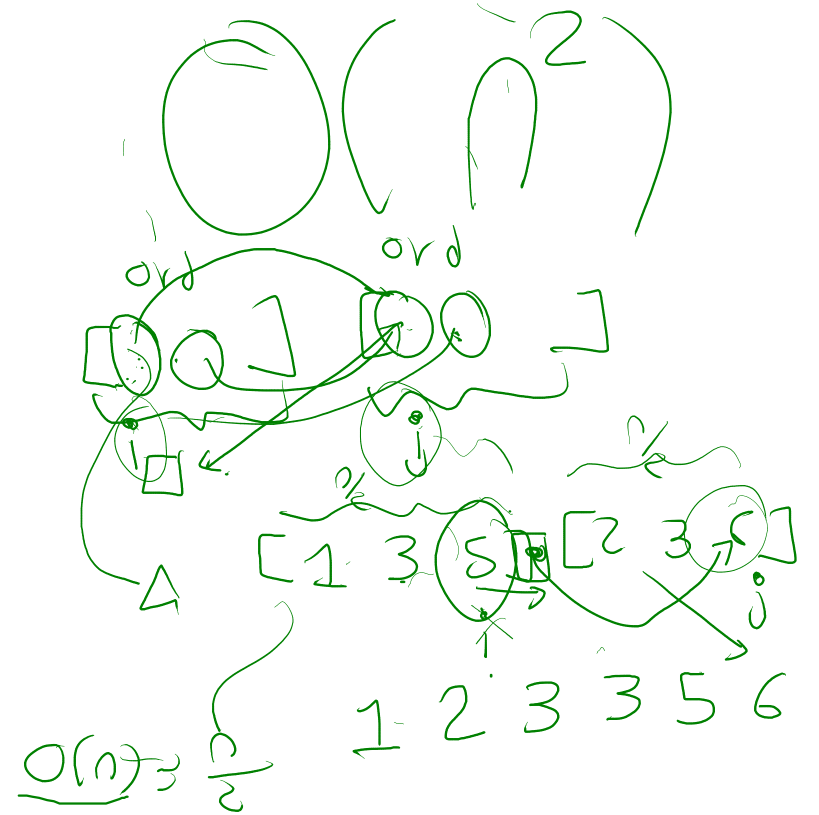
x mergeSort([3])

x mergeSor([44])

mergeSor([38 5])

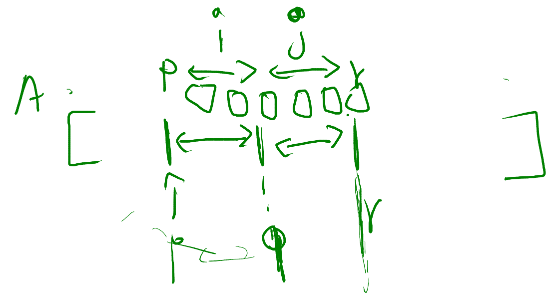
mergesort([47..26])

D mergeSort(27..48)

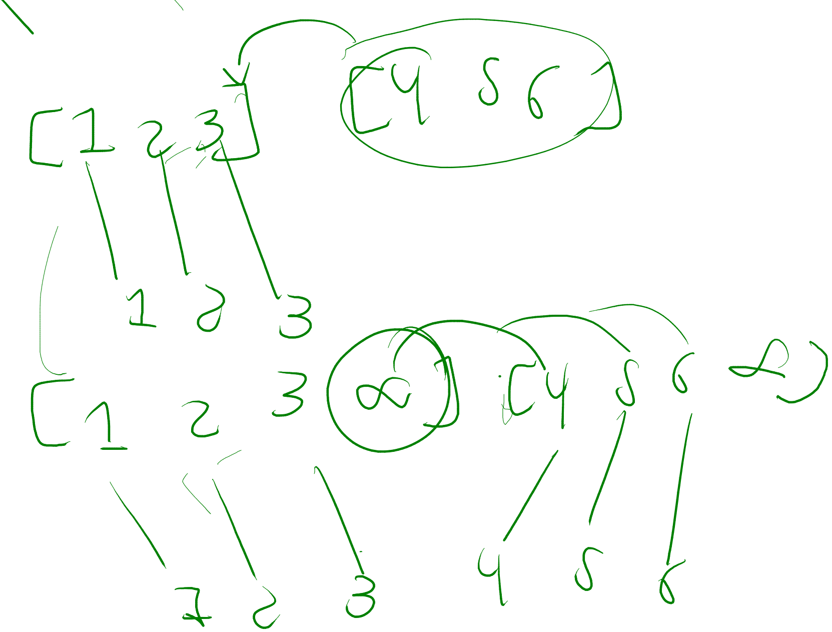


Divide y vencerás

Algoritmo MergeSort



```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

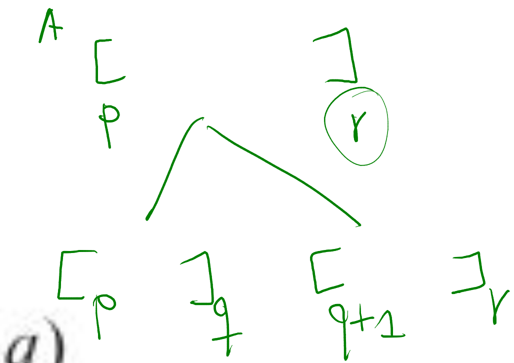


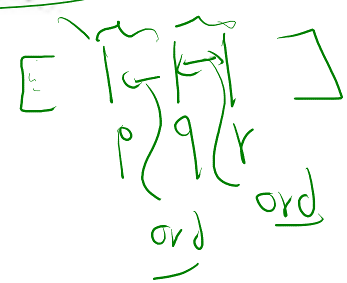
Divide y vencerás

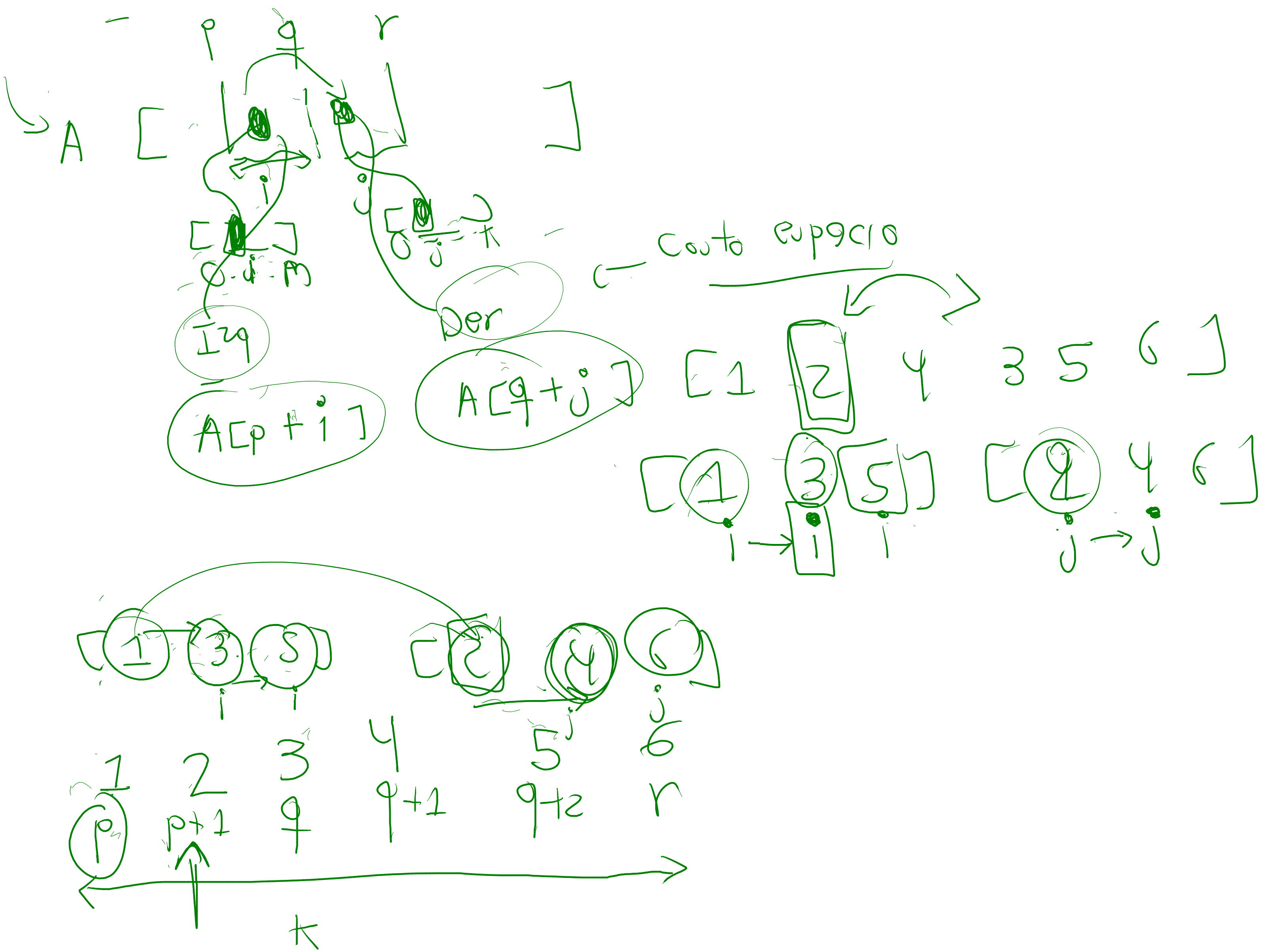
Algoritmo MergeSort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

combinar






p q r
A [ ]
i
j
0 .. i .. m
0 .. j .. k
Izq
Der
A[p+i]
A[q+j]
Costo espacio
[1 2 4 3 5 6]
[1 3 5] [2 4 6]
i → i
j → j
1 2 3 4 5 6
p p+1 q q+1 q+2 r
k

Divide y vencerás

Algoritmo MergeSort

Tamaño subproblema

$$T(n) = A\,T\left(\frac{n}{B}\right) + F(n) = 2T\left(\frac{n}{2}\right) + n$$

$\log_b a = \log_2 2 = 1$

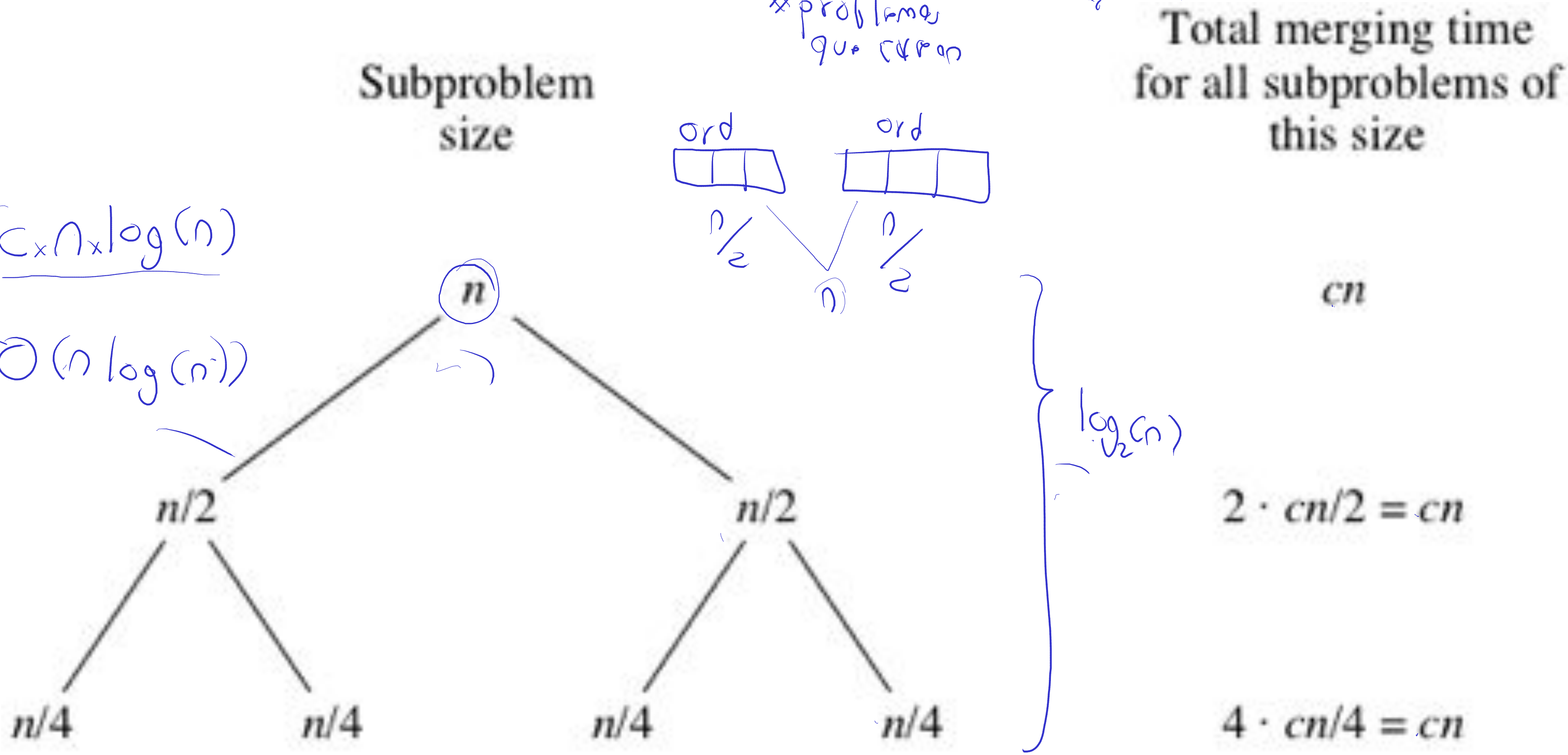
problemas que crean

combinar

2) n es $\Theta(n)$

$C \times n \times \log(n)$

$O(n \log(n))$



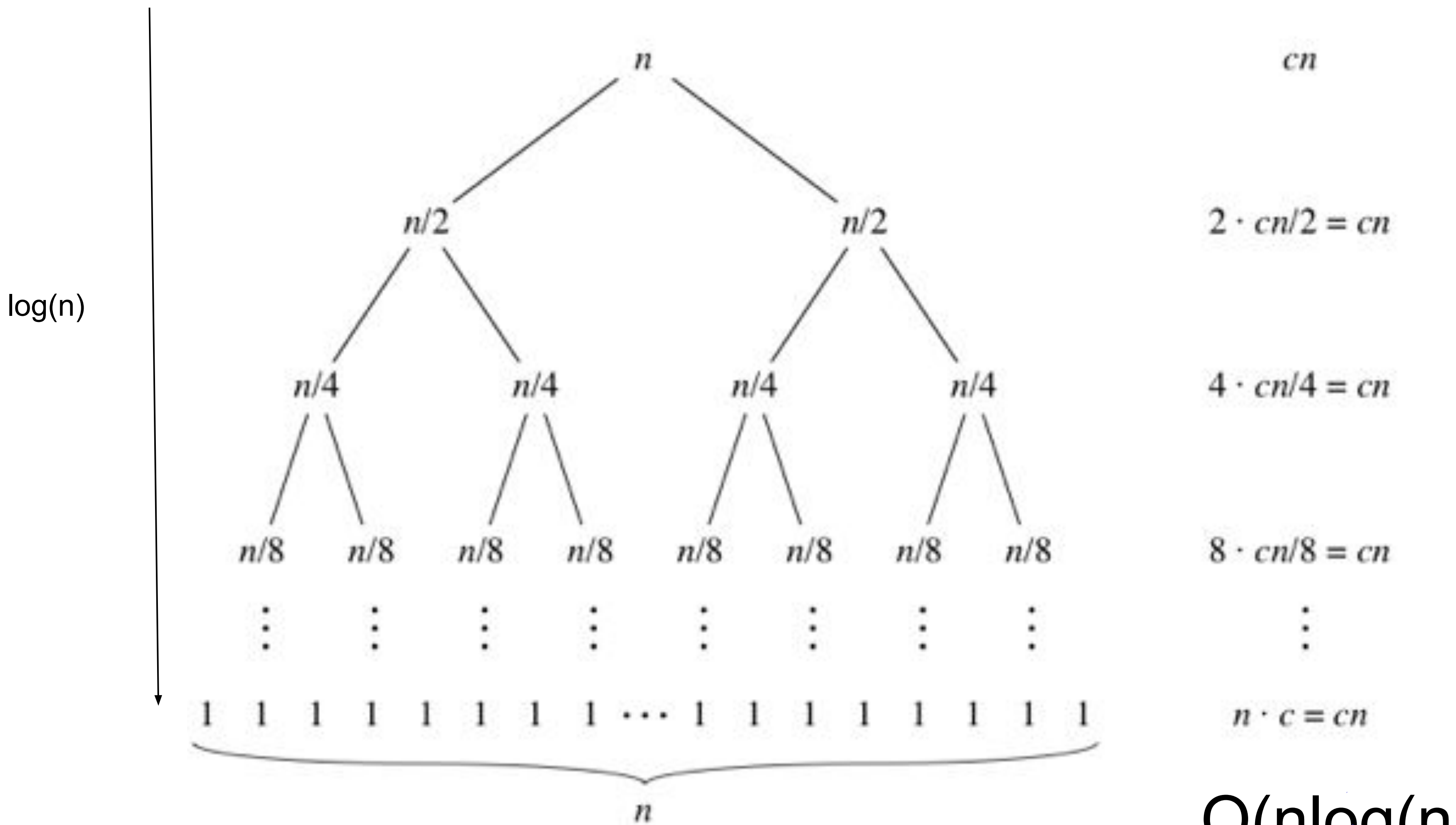
Tomado de https://www.khanacademy.org, Marzo 2018

Divide y vencerás

Algoritmo MergeSort

Subproblem size

Total merging time for all subproblems of this size



O(nlog(n))

Tomado de https://www.khanacademy.org, Marzo 2018

Divide y vencerás

Buscar el máximo de una lista

3	5	1	8	1	7

$O(n^2)$

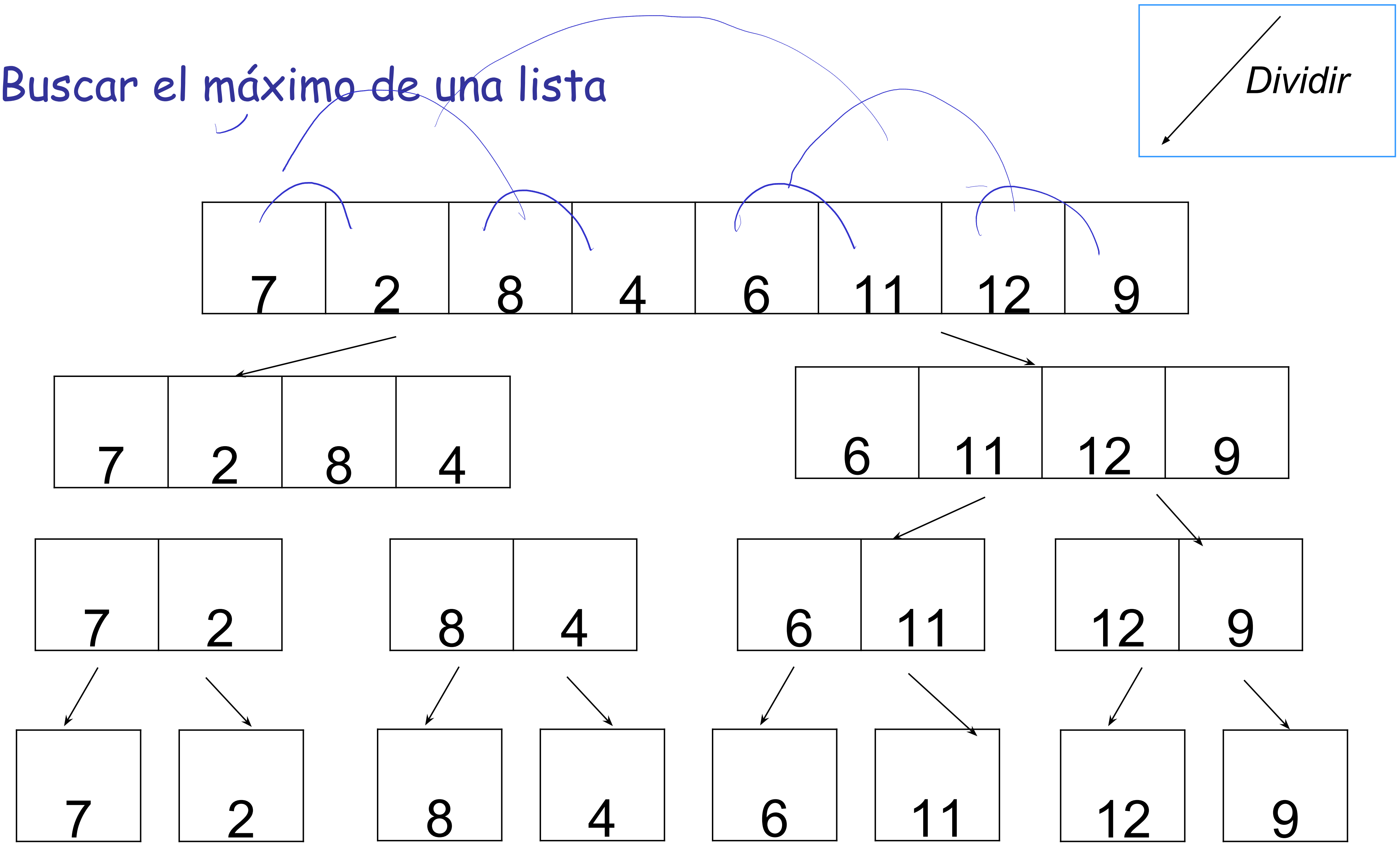
Dividir divide la lista a la mitad sucesivamente,

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 al la lista completa el primer elemento será el mayor

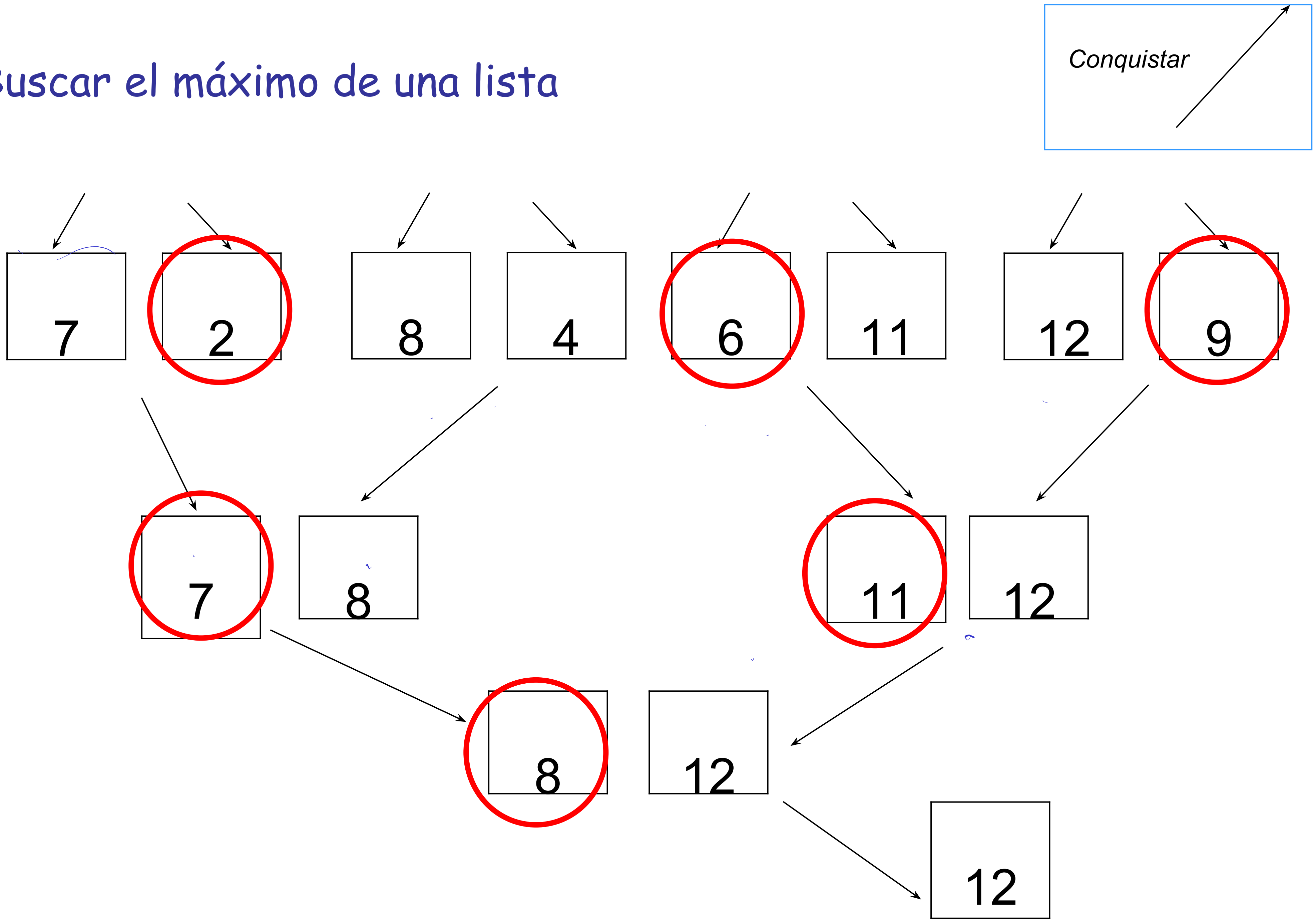
Divide y vencerás

Buscar el máximo de una lista



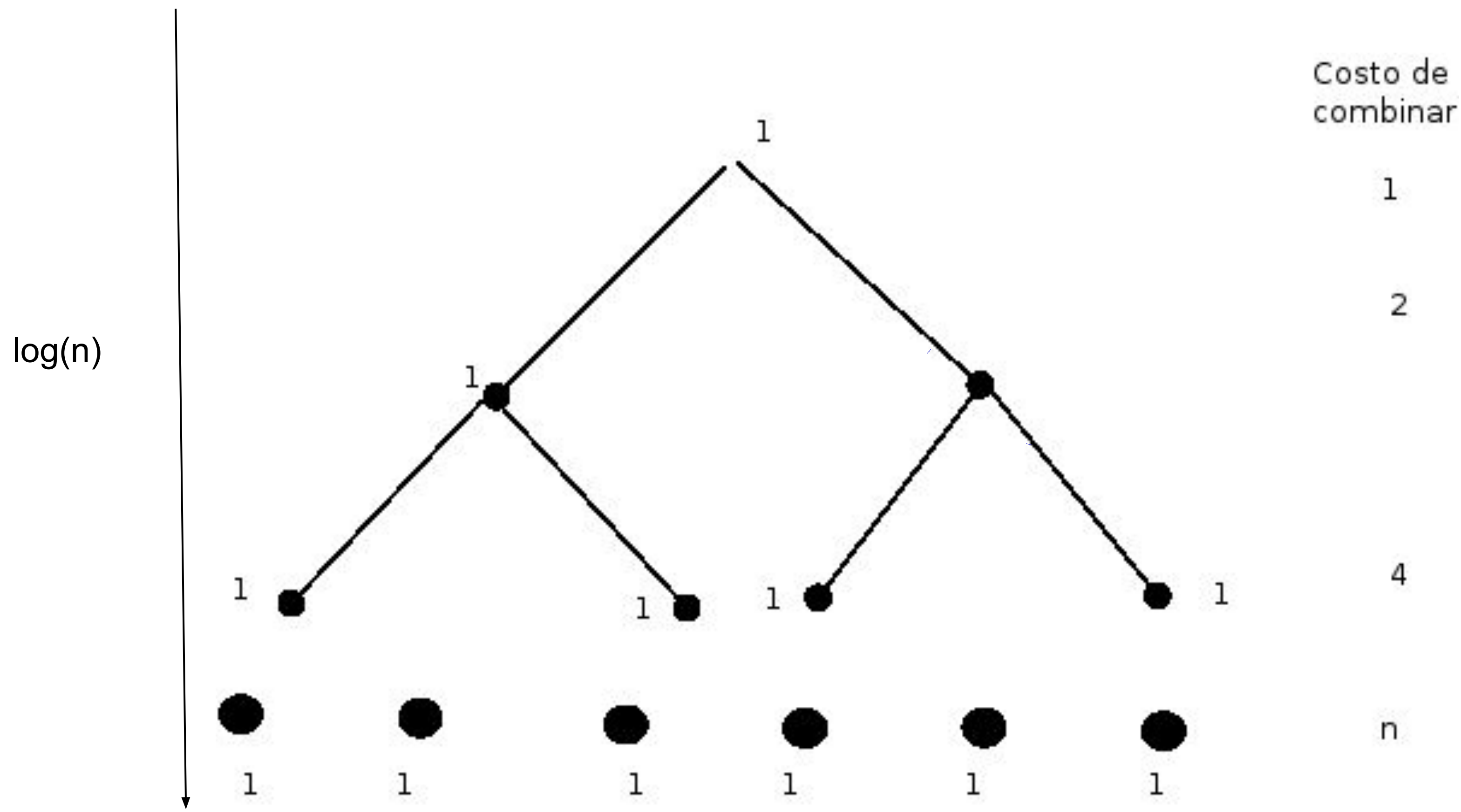
Divide y vencerás

Buscar el máximo de una lista



Divide y vencerás

Buscar el máximo de una lista



$\log_2(n)$

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

Tomado de https://www.khanacademy.org, Marzo 2018

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
```

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

$$\frac{n(n+1)}{2} - Sum(A) = ?$$

Divide y vencerás

Busqueda binaria

1	2	4	6	8	10	k

$O(n \log(n))$

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

Dividir divide la lista a la mitad,

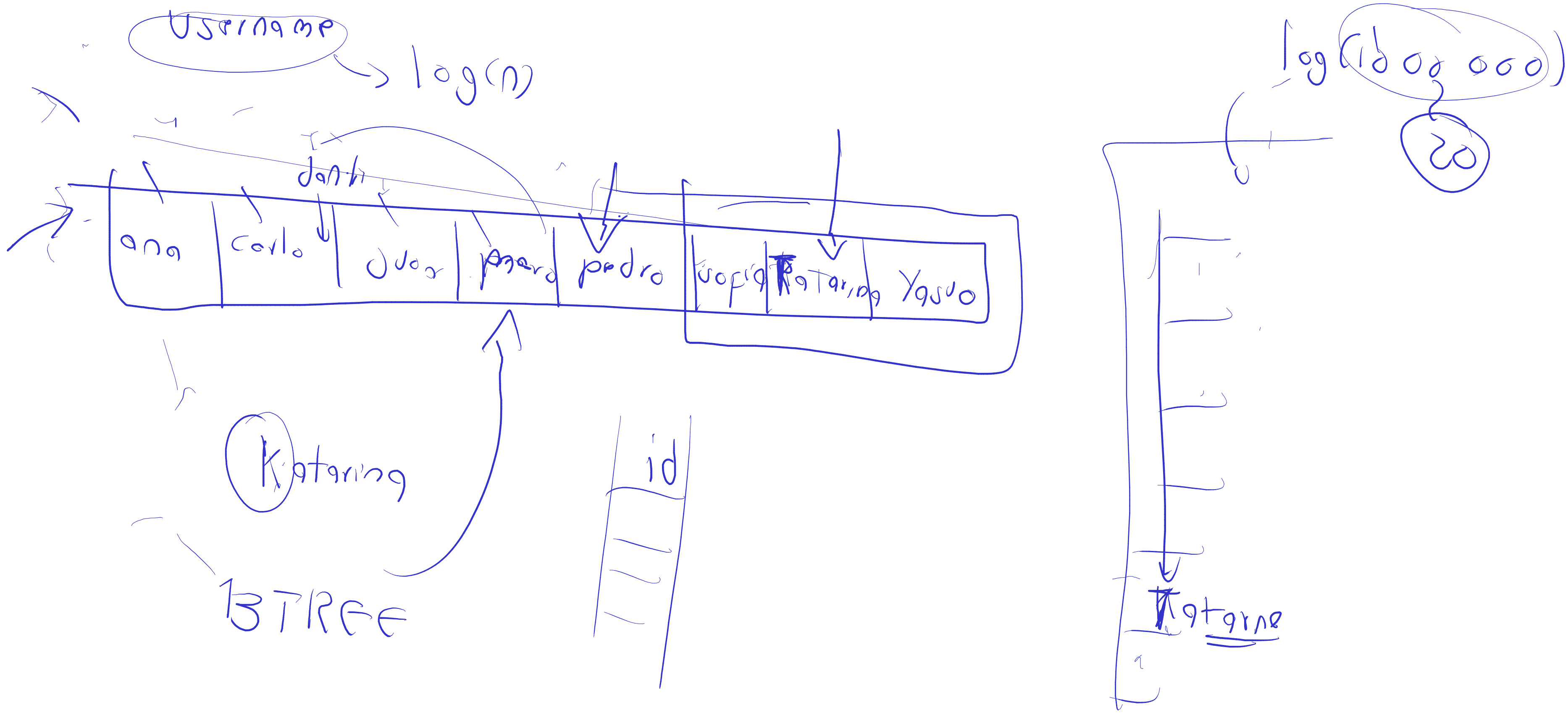
BD?

In direc

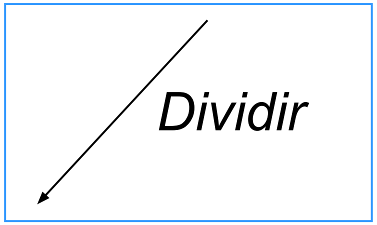
Conquistar Examine el último elemento de la primera lista y el primero de la segunda. Si el primero es menor o igual que x, repita dividir sobre la primera lista. En caso contrario hágalo sobre la segunda.

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

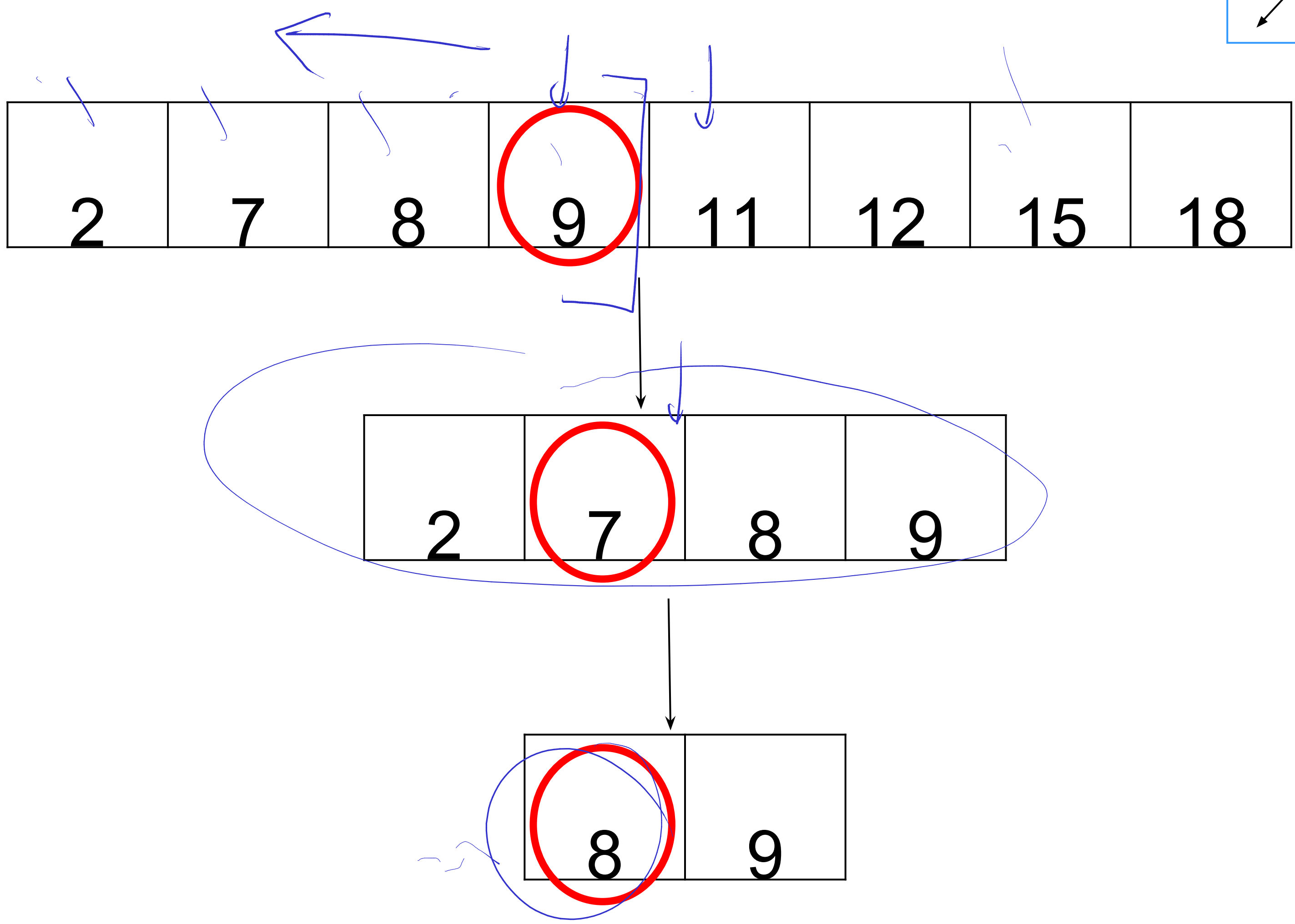
¿Cual es el costo computacional?. ¿Si la lista está desordenada, vale la pena ordenar y aplicar este algoritmo?


Username
log(n)
log(10.000.000)
20
ana
carlo
juan
pedro
sofia
Katarina
Yasuo
Katarina
BTREE
id

Divide y vencerás



Busqueda binaria Buscamos a 8.



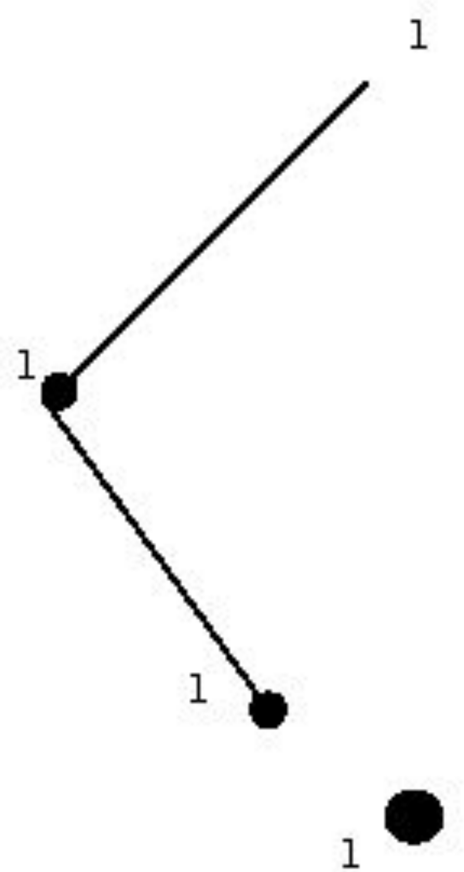
Combinar no es necesario, ya que al buscar de esta forma nos queda un elemento si tenemos éxito o ninguno si no lo tenemos

Preguntar

$T(n) = T(n/2) + 1$

$O(\log(n))$

Busqueda binaria



Costo de combinar

1

1

1

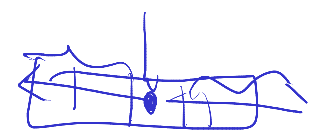
$$\sum_{i=0}^{log(n)} 1 = log(n) = O(log(n))$$

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
         Returne 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

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

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

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

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:

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

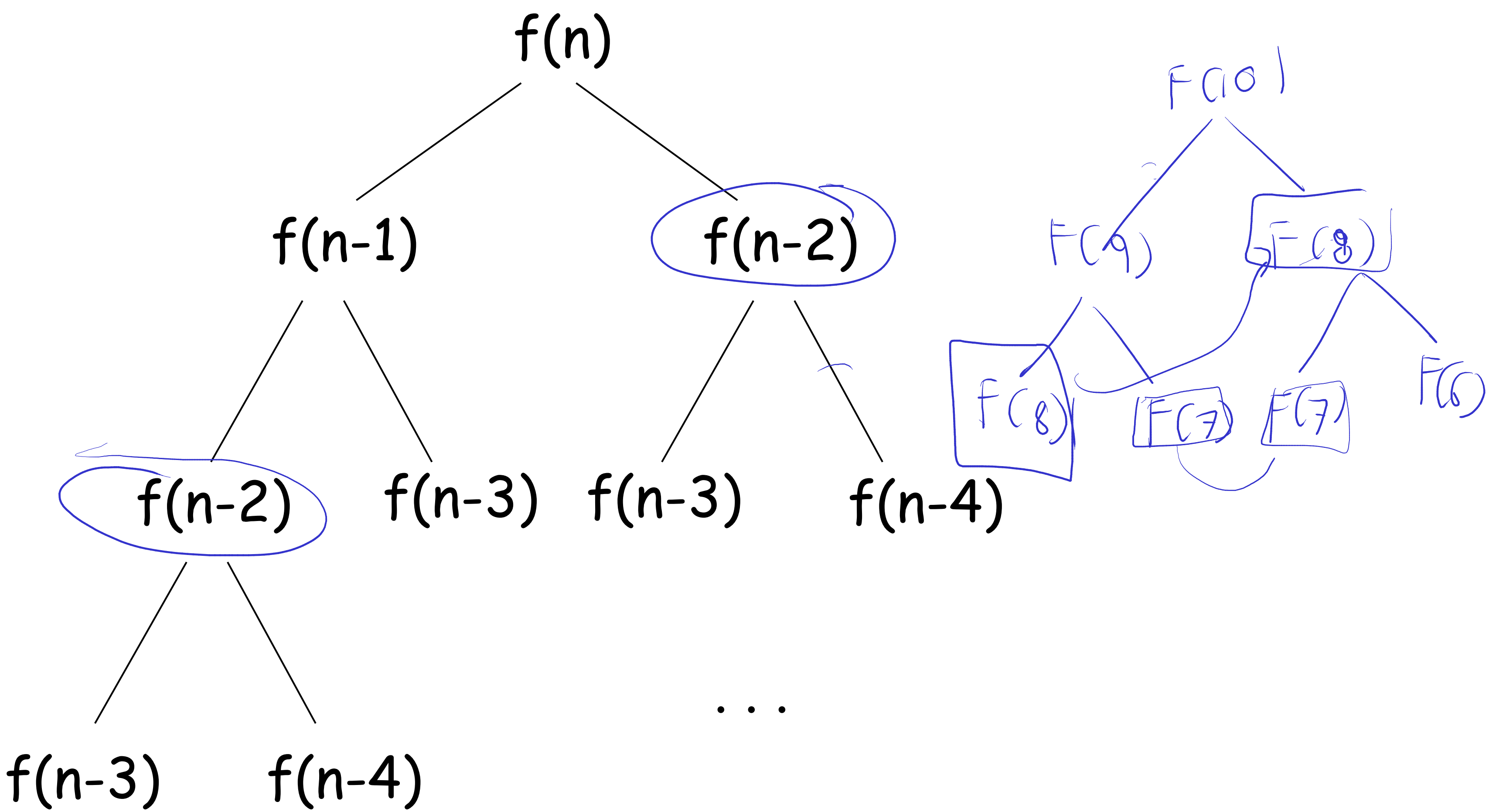
fibunnaci(n)

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

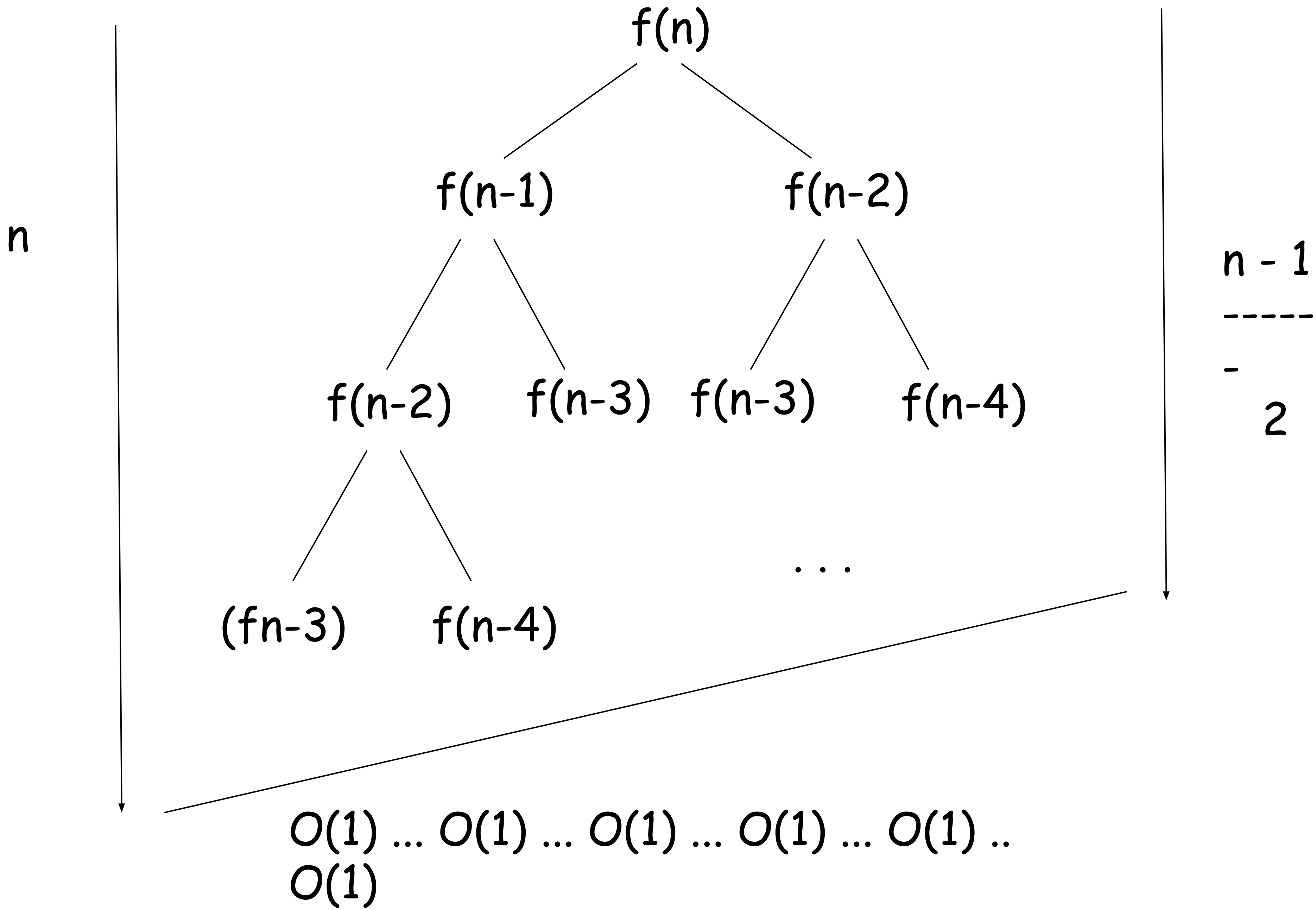
$$\boxed{T(n) = T(n-1) + T(n-2) + O(1)}$$

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 $O(2^{n})$

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

```
def algoritmo(n):
    if n==1:
        return 4
    else
        return 3*algoritmo(n/3)
```

$\Theta(n)$

$\log_b a = 1$

$T(n) = 3T(n/3) + 1$

$1 \in O(n^{1-\epsilon})$

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

```
def algoritmo2(n):
    for(i = 0; i<n; i++)
        ....
    if n == 1
        return 6
    else
        return 2*algoritmo2(n/4)
```

$\rightarrow O(n)$

$\Theta(n)$

$T(n) = 2 \times T(n/4) + cn$

$\log_4 2 = 0.5$

1) $cn \in O(n^{0.5-\epsilon})$

2) $cn \in \Theta(n^{0.5})$

3) $cn \in \Omega(n^{0.5+\epsilon})$

```
def algoritmoX(n):
   for (i = 0; i<n; i++)
       algoritmoY(n)
```

$O(n)$

$\Theta(n^3)$

```
def algoritmo(Y):
    for(i =0; i<n*n; i++)
        ..
   if n==1
      return 5
   else
       return 5*algoritmo(n/5)
```

$\Theta(n^2)$

$O(n^2)$

$T(n) = 5T(n/5) + n^2$

$\Theta(n^2)$

1) n^2 es $O(n^{1-\epsilon})$

2) n^2 es $\Theta(n)$

3) n^2 es $\Omega(n^{1+\epsilon})$

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))

$T(n) = T(n/3) + T(n/2) + \log(n)$ ← árbol

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~~ + $f(\lfloor n/2 \rfloor)$

$$T(n) = T(n/2) + 1$$

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