



AED II : TEORÍA

# Tema 1

## ANÁLISIS DE ALGORITMOS

22 - 1 - 25

### 1.1 INTRODUCCIÓN

Maximizar la eficiencia tiene relación con los recursos consumidos y productos conseguidos.

- Recursos consumidos:

#### Búsqueda con centinela

```
i := 0
A[n+1] := x
repetir
    i := i + 1
hasta A[i] == x
```

Influye: lo que vale  $n$ , lo que hay en  $A$  y  $x$ , tipos de datos...

→ Evitamos hacer 2 comprobaciones

Factores externos: ordenador, lenguaje, compilador, estructuras de datos...

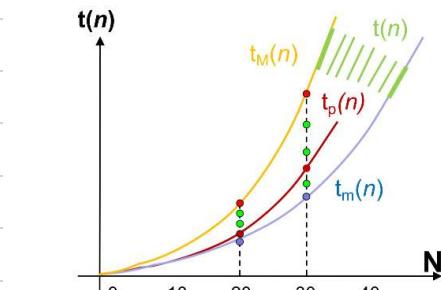
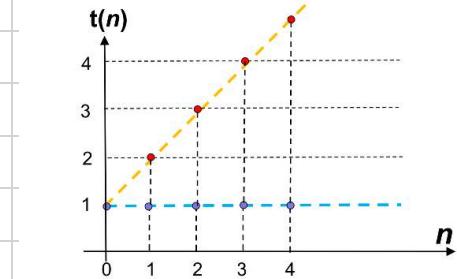
Factores internos: tamaño entrada, contenido (mejor, peor y caso promedio).

$t_m \rightarrow$  caso mejor (comprobar 1º pos)

$t_M \rightarrow$  caso peor (comprobar  $n+1$  pos)

EL MEJOR CASO NO ES QUE

N SEA PEQUEÑO



Siempre se cumple....  
 $t_m(n) \leq t(n) \leq t_M(n)$   
 $t_m(n) \leq t_p(n) \leq t_M(n)$

¿Qué es  $t(n)$ ?

#### BC (peor caso)

```
i := 0
A[n+1] := x
repetir
    i := i + 1
hasta A[i] == x
```

a  
b  
c  
d

- Tiempo ejecución:  $t(n) = (n+1)(c+d) + (a+b)$
  - Instrucciones ejecutadas  $t(n) = (n+1)(1+1) + (1+1)$
  - Ejecuciones del bucle principal  $t(n) = n+1$
- Usaremos conteo de instrucciones. Si no se puede predecir flujo de ejecución, predecimos trabajo total realizado.

### Reglas básicas conteo:

- Cada instrucción + 1
- Los bucles FOR son sumatorios

Si el nº de iteraciones es fijo

$$t_p(n) = t_m(n) = t_m(n)$$

$$\left\{ \begin{array}{l} \sum_{i=1}^n k = k \cdot n \\ 1 \dots n \\ \sum_{i=a}^b k = k \cdot (b-a) + 1 \\ a \dots b \\ \sum_{i=a}^b r^i = \frac{r^{b+1} - r^a}{r-1} \\ \sum_{i=1}^n i^2 \approx \int_0^n i^2 di = \frac{i^3}{3} \Big|_0^n \\ \text{puede ser de } 0 \text{ a } n \text{ o de } 1 \text{ a } n+1 \end{array} \right.$$

recuerda ejemplo  $1+100\dots$

- Llamadas a procedimientos: contar primero los que no llaman a otros.
- If y case: mejor y peor caso según la condición.
- Bucles WHILE y REPEAT: algunos se pueden convertir en FOR.

$$\sum_{i=1}^n \left( \sum_{j=1}^n \left( 1 + \sum_{k=1}^j (1) + 1 \right) \right) =$$

$$\sum_{i=1}^n \left( \sum_{j=1}^n (2+n) \right) =$$

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

$$((2+n) \cdot n) \cdot n =$$

$$n^3 + 2n^2$$

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

Función Fibonacci ( $n: \text{int}$ ):  $\text{int}$   
if  $n < 0$  then  
 error('No válido')  
case  $n$  of  
 0, 1: return N  
else  
 fnm2:= 0  
 fnm1:= 1  
 for i:= 2 to  $n$   
 fn:= fnm1 + fnm2  
 fnm2:= fnm1  
 fnm1:= fn  
 end  
 return fn  
end

$$s: < 0 \rightarrow 2$$

$$\text{si es } 1 \rightarrow 3$$

$$\text{si es } 2 \text{ o } +$$

$$5 + 3(n-1)$$

función por trozos

$$t(n) = \begin{cases} 2, & n < 0 \\ 3, & n = 0; 1 \\ 5 + 3(n-1), & n \geq 2 \end{cases}$$

$$tm(n) < tM(n)$$

```
A[0, (n-1) div 2]:= 1
key:= 2
i:= 0
j:= (n-1) div 2
cuadrado:= n*n
while key<=cuadrado do
  k:=(i-1) mod n
  l:=(j-1) mod n
  if A[k, l] ≠ 0 then
    j:=(i+1) mod n
  else
    j:= k
    i:= l
  end
  key:= key+1
end
```

$$t(n) = 5 + \sum_{key=2}^{n^2} (4 + t_{IF})$$

$$tm(n) = 5 + \sum_{key=2}^{n^2} (4 + 1) = 5 + 5(n^2 - 2) + 1$$

$$tM(n) = 5 + \sum_{key=2}^{n^2} (4 + 2) = 5 + 6(n^2 - 2) + 1$$

Para calcular el tiempo medio  $t_p(n)$ :

- Para IF/CASE, agrupar los casos en conjuntos que tarden lo mismo y hacer media ponderada.

$$tp\_if(n) = t_{true} \cdot p_{true} + t_{false} \cdot p_{false}$$

- Para los while constante si cada iteración es constante.

- Es útil parametrizar la probabilidad con índice.

# REPASO INTEGRALES

(inmediatas)

$$1. \int dx = x$$

$$10. \int u' \cdot u^u dx = \frac{u^u}{\ln u}$$

$$19. \int \frac{1}{\sqrt{1-x^2}} dx = \arcsen x$$

$$2. \int k dx = kx$$

$$11. \int \operatorname{sen} x dx = -\cos x$$

$$20. \int \frac{u'}{\sqrt{1-u^2}} dx = \arcsen u$$

$$3. \int x^n dx = \frac{x^{n+1}}{n+1}$$

$$12. \int u' \cdot \operatorname{sen} u dx = -\cos u$$

$$21. \int \frac{1}{1+x^2} dx = \operatorname{arctg} x$$

$$4. \int u' \cdot u^n dx = \frac{u^{n+1}}{n+1}$$

$$13. \int u' \cos u dx = \operatorname{sen} u$$

$$22. \int \frac{u'}{1+u^2} dx = \operatorname{arctg} u$$

$$5. \int \frac{1}{x} dx = \ln x$$

$$14. \int \frac{1}{\cos^2 x} dx = \operatorname{tg} x$$

$$6. \int \frac{u'}{u} dx = \ln u$$

$$15. \int (1+\operatorname{tg}^2 x) dx = \operatorname{tg} x$$

$$7. \int e^x dx = e^x$$

$$16. \int \frac{u'}{\cos^2 u} dx = \operatorname{tg} u$$

$$8. \int u' \cdot e^u dx = e^u$$

$$17. \int \frac{1}{\operatorname{sen}^2 x} dx = -\operatorname{cotg} x$$

$$9. \int k^x dx = \frac{k^x}{\ln k}$$

$$18. \int \frac{u'}{\operatorname{sen}^2 u} dx = -\operatorname{cotg} u$$

$k \rightarrow$  constantes

$u \rightarrow$  función

$u' \rightarrow$  función derivada de  $u$

$x^n \rightarrow n$ : exponente (constante)

$e^u \rightarrow u$ : exponente función

### Ejercicio (tiempo promedio):

Estudiar  $t_p(n)$  para las instrucciones de asignación...

$$t_p(n) = 1 + \sum_{i=1}^n \sum_{j=1}^{i-1} (1 \cdot 0.5 + 0 \cdot 0.5) =$$

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

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

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

$$\boxed{1 + \frac{n(n+1)}{4} - \frac{n}{2}}$$

```

cont:=0
para i:= 1,...,n hacer
    para j:= 1,...,i-1 hacer
        si a[i] < a[j] entonces
            cont:= cont + 1
        finsi
    finpara
finpara

```

```

i:= 1
mientras i ≤ n hacer
    si a[i] ≥ a[n] entonces
        a[n]:=a[i]
    finsi
    i:= i + 1
finmientras

```

$$t_p(n) = 1 + \sum_{i=1}^n \left( 1 + \frac{1}{i+1} \right) =$$

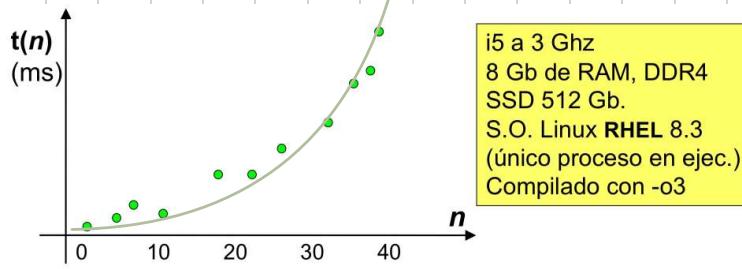
$$1 + \sum_{i=1}^n 1 + \sum_{i=1}^n \frac{1}{i+1}$$

$$\int_0^n \frac{1}{i+1} = (n(n+1)) - (1(1))$$

$$\boxed{1 + n + \ln(n+1)}$$

El análisis de los algoritmos puede ser también a posteriori:

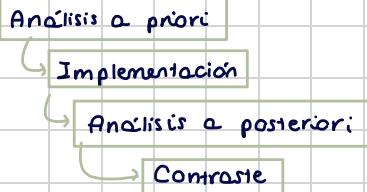
Se implementa el algoritmo y se calcula el tiempo para diferentes "n".



← Tiene pinta de ser cuadrática  
Podemos comprobarlo haciendo una  
regresión y comprobando que se ajusta  
bien.

Cuando hemos hecho ambos análisis hacemos un contraste teórico/experimental. Permite detectar errores.

Con el análisis teórico podemos extraer información relevante sobre el tiempo que tal vez con el experimental sería más complejo.



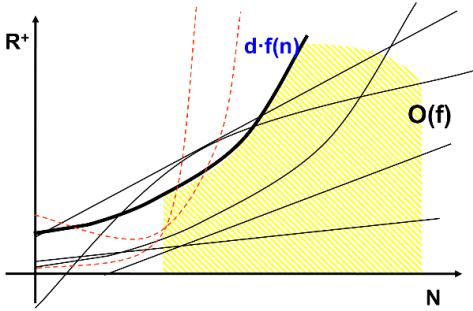
Mie 29 Enero 2025

## 1.2 NOTACIONES ASINTÓTICAS

Indican como crece t sin considerar factores externos y para valores grandes.

- $O(t)$ : orden complejidad
- $\Omega(t)$ : orden inferior
- $\Theta(t)$ : orden exacto
- $o(t)$ : o pequeña de t

### a) orden de complejidad $O(f)$



Conjunto de funciones acotadas bajo f a partir de cierta n.

multiplo positivo de f

$$O(f) = \{t: N \rightarrow R^+ / \exists d \in R^+, \exists n_0 \in N, \forall n \geq n_0; t(n) \leq d \cdot f(n)\}$$

- $O(f)$  es un conjunto de funciones, NO una función.
- no nos importa para valores pequeños.

El uso de los órdenes sirve para:

- 1) simplificar un tiempo  $t(n)$  para comparar.
- 2) una función puede ser difícil de acotar.
- 3) acotar una función la cual  $t_m(n) \neq t_M(n)$ .  $O(t_m(n))$

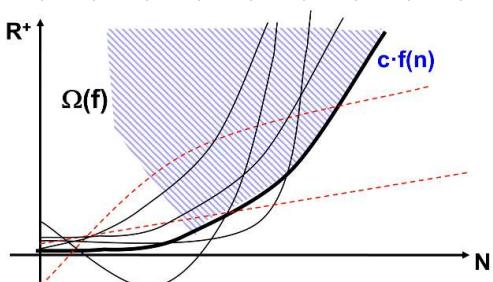
Si el orden de una función f está contenido en el orden de una función g :

$$O(f) \subseteq O(g) \Leftrightarrow O(f) \subseteq O(g), \text{ toda } t \in O(f), t \in O(g)$$

Se cumple :

- $O(c) = O(d)$ , c y d son constantes positivas
- $O(c) \subseteq O(n)$
- $O(cn+b) = O(dn+e)$
- $O(p) = O(q)$  si p y q tienen el mismo orden

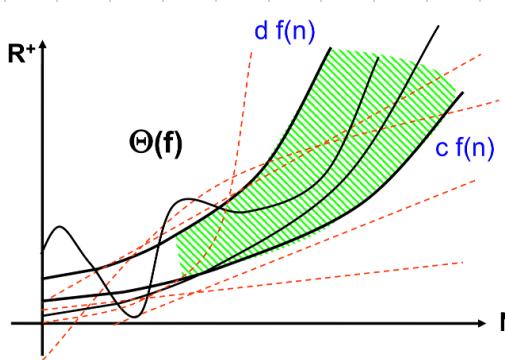
### b) Orden inferior $\Omega(f)$



Acota inferiormente a un conjunto de funciones

$$\Omega(f) = \{t: M \rightarrow R^+ / \exists c \in R^+, \exists n_0 \in N, \forall n \geq n_0; c \cdot f(n) \leq t(n)\}$$

c) Orden exacto de  $f(n)$ :  $\Theta(f)$



La intersección de los conjuntos  $\Omega(f)$  y  $O(f)$ .

$\Theta(f) : \Omega(f) \cap O(f) =$

$$\{ t : N \rightarrow R^+ / \exists c, d \in R^+, \exists n_0 \in N, \forall n \geq n_0; c \cdot f(n) \leq t(n) \leq d \cdot f(n) \}$$

si  $t \in O(f)$ ,  $t \in \Omega(f)$  entonces  $t \in \Theta(f)$

Ejemplos. ¿Cuáles son ciertas y cuáles no?

$$3n^2 \in O(n^2) \quad V \quad n^2 \in O(n^3) \quad V \quad n^3 \in O(n^2) \quad F$$

$$3n^2 \in \Omega(n^2) \quad V \quad n^2 \in \Omega(n^3) \quad F \quad n^3 \in \Omega(n^2) \quad V$$

$$3n^2 \in \Theta(n^2) \quad V \quad n^2 \in \Theta(n^3) \quad F \quad n^3 \in \Theta(n^2) \quad F$$

$$2^{n+1} \in O(2^n) \quad V \quad (2+1)^n \in O(2^n) \quad F \quad (2+1)^n \in \Omega(2^n) \quad V$$

$$O(n) \in O(n^2) \quad F \quad (n+1)! \in O(n!) \quad F \quad n^2 \in O(n!!) \quad V$$

d) o pequeña de  $f(n)$ :  $o(f)$

Llamamos o pequeña al conjunto de todas las funciones de  $N$  en  $R^+$  que crecen igual que  $f$  asintóticamente.

Es como descomponer el orden exacto preservando la constante del término más grande.

$$o(f) : \{ t : N \rightarrow R^+ / \lim_{n \rightarrow \infty} t(n)/f(n) = 1 \}$$

$n^2$  y  $n^2+1$  están en  $o(n^2)$ , pero  $2n^2$  y  $n^2+1$  están en  $o(2n^2)$  y  $o(n^2)$

Martes 4 Febrero 2025

Ahora vamos a ver algunas propiedades de las notaciones asintóticas.

P1) Transitividad: si  $f \in O(g)$  y  $g \in O(h)$  entonces  $f \in O(h)$ .

Ejemplo:  $2n+1 \in O(n)$ ,  $n \in O(n^2) \rightarrow 2n+1 \in O(n^2)$

P2 Si  $f \in O(g)$  entonces  $O(f) \subseteq O(g)$ . ¿para los  $\Omega$ ?

P3 Relación pertenencia / contenido:

i)  $O(f) = O(g) \Leftrightarrow f \in O(g) \text{ y } g \in O(f) \rightarrow \text{crecen igual}$

ii)  $O(f) \subseteq O(g) \Leftrightarrow f \in O(g)$

P4 Propiedad del máximo: dadas  $f$  y  $g$   $O(f+g) = O(\max(f,g))$

Con  $\Omega$  es  $\Omega(f+g) = \Omega(\max(f,g))$ , para  $\Theta$  exactamente igual.

Ejemplo  $\Omega(f+g)$  siendo  $f = n^2$  y  $g = n^3 \rightarrow \Omega(n^3)$

P5 Equivalencia entre notaciones

$O(f) = O(g) \Leftrightarrow \Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g) \Leftrightarrow \Omega(f) = \Omega(g)$

P6 Relación límites / órdenes : dadas  $f$  y  $g$ :

i)  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow O(f) = O(g) \rightarrow$  si da 1  $\rightarrow o(f) = o(g)$

ii)  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subset O(g)$

iii)  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow O(f) \supset O(g)$

La notación puede tener más de un parámetro:  $n, m, p, \dots$  etc. Los conceptos vistos aplican también a varios parámetros. Las propiedades se siguen cumpliendo. Pero no podemos asumir cosas como " $O(n) = O(m)$ ". Hay que ver que términos acotan a otros.

$$t(n, m, r) = nmr + mr^2 + nr + n^3 + n^2r$$

$$\downarrow \\ nr \in nmr$$

$$\downarrow$$

$$t(n, m, r) \in O(nmr + mr^2 + n^3 + n^2r) \text{ se deja así de feo :)}$$

¿Qué relación entre  $O(n+m)$  y  $O(nm)$ ,  $O(n^2)$  y  $O(n+2^m)$ ?

$O(n+m) \subset O(nm)$ , pero en el otro par no se puede afirmar nada.

En algunos casos interesa estudiar el tiempo sólo para ciertos tamaños de entrada.

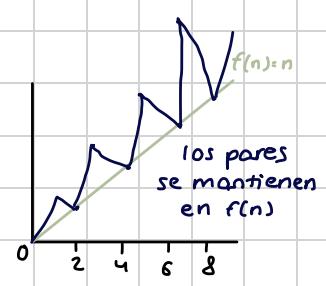
Ejemplo:  $n = 2^k$ , se usa el orden condicionado de  $f(n)$ :  $O(f|P)$

$\downarrow$   
predicado

El conjunto de funciones debe cumplir el predicado.

•  $O(f) = O(f|\text{true}) \rightarrow \forall n$

• Para cualquier  $f$  y  $g$ ,  $f \in O(g|\text{false}) \quad ? O(f) \leftrightarrow O(f|P)?$



Relaciones entre órdenes de complejidad.

$$\begin{aligned} O(1) &\subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset \\ O(n \cdot (\log n)^2) &\subset O(n^{1.001\dots}) \subset O(n^2) \subset O(n^3) \subset \dots \\ &\subset O(2^n) \subset O(n!) \subset O(n^n) \end{aligned}$$

En los  $\Omega$  funciona al revés.

En  $\Theta$  no habrá relación de inclusión, no se pueden ordenar.

- Si hacemos una operación para  $n$ , otra para  $n_2$ , otra para  $n_4 \dots$  aparecerá un orden logarítmico  $O(\log_2 n)$ .

$$\begin{aligned} \frac{n}{1}, \frac{n}{2}, \frac{n}{4}, \dots, \frac{n}{n} & \xrightarrow{n=2^k} \\ \frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2}, \dots, \frac{n}{2^k} & \xrightarrow{\quad} 1 + \log_2 n \end{aligned}$$

$$\sum_{i=1}^n i \in O(n) \quad \sum_{i=1}^n i \in O(n^2) \quad \sum_{i=1}^n i^m \in O(n^{m+1})$$

### 1.3 ECUACIONES DE RECURRENCIA

Miércoles 5 Febrero 2025

¿Cuántas ecuaciones "mover" se ejecutan?

Hanoi (n, i, j, k)

if  $n > 0$  then

Hanoi (n-1, i, k, j)

mover (i, j)

Hanoi (n-1, k, j, i)

else

mover (i, j)

$$t(n) = \begin{cases} 1, & n=0 \\ 2 + 2 \cdot t(n-1), & n > 0 \end{cases}$$

en la ecuación se refiere a sí mismo

Tipos de ecuaciones de recurrencia: estas ecuaciones tendrán "casos base" y luego "en otro caso" (recursividad).

- Lineales y homogéneas:

$$t(n) - 3t(n-1) + 2t(n-2) \dots = 0$$

↑  
si no tuviera

no sería ni considerado

algoritmo

- Lineales y no homogéneas:

$$t(n) - 3t(n-1) + 2t(n-2) \dots = 1$$

- No lineales:

$$t(n) = n \cdot t(n-1) \dots$$

**algoritmoRec1(int a[], int n)**

```

    si n<=1
        return 1
    si no
        return 1+algoritmoRec1(a,n-1)
    
```

**algoritmoRec2(int a[], int pos)**

```

    si pos>1
        si par(a[pos])
            algoritmoRec2(a,pos/2)
        en otro caso
            algoritmoRec2(a,pos/4)
    
```

Ejemplos

$$t(n) : \begin{cases} 2, & n \leq 1 \\ 2 + t(n-1), & n > 1 \end{cases}$$

lineal y no homogénea

$$t(n) = \begin{cases} 1, & n \leq 1 \\ 1 + 1 + t_2, & n > 1 \end{cases}$$

$$t_m(n) = \begin{cases} 1, & n \leq 1 \\ 2 + t_m(n/4), & n > 1 \end{cases}$$

$$t_M(n) = \begin{cases} 1, & n \leq 1 \\ 2 + t_M(n/2), & n > 1 \end{cases}$$

¿Cómo resolver una ecuación lineal homogénea?

Las soluciones tendrán la forma  $t(n) = x^n$ .

$$t(n) : \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ 3 + (n-1) - 4t(n-2), & n > 1 \end{cases}$$

el valor de las constantes  
depende de los casos bases

Calcular para  $t(2)$ ,  $t(3)$ ,  $t(4)$

usando la ecuación solución  
específica y la de  $t(n)$  !!

$$t(2) = 3t(n-1) - 4t(n-2) = 3 \cdot 1 - 4 \cdot 0 = 3$$

$$t(2) = \frac{1}{5} \cdot 4^n - \frac{1}{5} \cdot (-1)^n = \frac{1}{5} \cdot 16 - \frac{1}{5} \cdot 1 =$$

$$\frac{16}{5} - \frac{1}{5} = \frac{15}{5} = 3 \quad \text{coinciden!!}$$

$$\textcircled{1} \quad t(n) - 3t(n-1) - 4t(n-2) = 0 \quad (\text{ecuación original})$$

$$x^n - 3x^{n-1} - 4x^{n-2} = 0 / x^{n-2}$$

$$\textcircled{2} \quad x^2 - 3x - 4 = 0 \quad (\text{ecuación característica})$$

$$\textcircled{3} \quad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{3 \pm \sqrt{9 + 4 \cdot 4}}{2} \rightarrow x = 4 \quad x = -1$$

$$\textcircled{4} \quad t(n) = c_1 \cdot 4^n + c_2 \cdot (-1)^n \quad \text{siendo } c_k \text{ constantes}$$

(ecuación solución genérica)

$$\textcircled{5} \quad \begin{cases} t(0) = 0 \\ t(1) = 1 \end{cases} = \begin{cases} c_1 + c_2 \\ 4c_1 - c_2 \end{cases} \quad t(n) = \frac{1}{5} 4^n - \frac{1}{5} (-1)^n$$

(ecuación solución específica)

$$1 = 5c_1 / \quad c_1 = \frac{1}{5} \quad c_2 = -\frac{1}{5}$$

¿Qué pasa si no todas las soluciones son distintas?

$$t(n) = c_1 \cdot s_1^n + c_2 \cdot s_2^n + \dots + c_k \cdot s_k^n + \\ + c_{k+1} \cdot n \cdot s_k^n + c_{k+2} \cdot n^2 \cdot s_k^n + \dots + c_{k+1+m} \cdot n^{m-1} \cdot s_k^n$$

Ejemplo 1:

$$t(n) : \begin{cases} 1, & n = 0; 1 \\ 2t(n-1) - t(n-2), & n > 1 \end{cases}$$

$$t(n) - 2t(n-1) + t(n-2) = 0$$

$$x^n - 2x^{n-1} + x^{n-2} = 0 / x^{n-2}$$

$$x^2 - 2x + 1 = 0 \rightarrow \frac{x^2 - 2x + 1}{2} = \frac{2 \pm \sqrt{4-4(1)(1)}}{2} = \frac{2 \pm 0}{2} \quad \begin{matrix} x=1 \\ x=1 \end{matrix}$$

multiplicidad 2

$$t(n) = c_1 \cdot 1^n + c_2 \cdot n \cdot 1^n = c_1 + c_2 n$$

$$\begin{cases} t(0) = 1 = c_1 \\ t(1) = 1 = c_1 + c_2 \end{cases} \quad \begin{matrix} c_1 = 0 \\ c_2 = 1 \end{matrix}$$

hace falta llegar al paso 5

si piden la o pequeña o

el tiempo de ejecución

$$t(n) = 1$$

Ejemplo 2:

Martes 11 Febrero 2025

$$t(n) \begin{cases} 0, & n=0 \\ 3, & n=1 \\ 10, & n=2 \\ 5 \cdot t(n-1) - 8 \cdot t(n-2) + 4 \cdot t(n-3), & n>2 \end{cases}$$

$$\begin{aligned} t(n) - 5t(n-1) + 8t(n-2) - 4t(n-3) &= 0 / x^{n-3} \\ x^3 - 5x^2 + 8x - 4 &= 0 \end{aligned}$$

"teorema de las raíces racionales"

las raíces de un polinomio son p/q, donde p es divisor de  $a_k$  y q divisor de  $a_0$ .

$$\begin{array}{|r|rrrr} \hline 1 & 1 & -5 & 8 & -4 \\ \hline & 1 & -4 & 4 & 0 \\ \hline & 1 & -4 & 4 & 0 \\ \hline \end{array}$$

$$(x-1)(x^2-4x+4)=0$$

soluciones = 1; 2; 2

$$x = \frac{4 \pm \sqrt{16-4(1)(4)}}{2} \leftarrow 2$$

$$t(n) = c_1 \cdot 1^n + c_2 \cdot 2^n + c_3 \cdot n \cdot 2^n$$

resolver paso 5!!

### Recurrencias no homogéneas

$$\frac{t(n) - t(n-1)}{(x-1)(x-b)^{g(p(n))+1}} = b^n \cdot p(n) \rightarrow \text{hay que reconocer estructuras de este tipo}$$

por cada término verde que encontramos añadir esto

$$t(n) - t(n-1) = 2^n(n^2+1)$$
$$(x-1)(x-2)^{2+1}$$

$$t(n) \begin{cases} 1+n, & n=0,1 \\ 4t(n-2) + (n+5)3^n + n^2, & n>1 \end{cases}$$

$$\begin{aligned} \textcircled{1} \quad t(n) - 4t(n-2) &= 3^n(n+5) + n^2 \\ \textcircled{2} \quad (x^2-4)(x-3)^2(x-1)^3 &= 0 \\ \textcircled{3} \quad x = 2, -2, 3, 3, 1, 1, 1 \\ \textcircled{4} \quad t(n) = c_1 \cdot 2^n + c_2(-2)^n + c_3 3^n + c_4 n \cdot 3^n + c_5 \cdot 1^n + c_6 \cdot n \cdot 1^n + c_7 \cdot n^2 \cdot 1^n \end{aligned}$$

puede ser  $1^n(n^2)$

orden de  $t(n)$

• OJO! El  $1^n$  puede estar implícito

• Agrupar términos con misma base b  $3^n(n+5) + 3^n \cdot n^2 = 3^n(n+5+n^2)$

$$\text{Ejemplo: } t(n) - t(n-3) = 2 + n^3 + n^2 \cdot 3^n + 2^{(n+1)} + 8n^2$$

¿Cuál es la ecuación característica?

$$t(n) - t(n-3) = 1^n \cdot 2 + 1^n \cdot n^3 + 3^n \cdot n^2 + 2^n \cdot 2 + 1^n \cdot 8n^2$$

$$t(n) - t(n-3) = 1^n(2 + n^3 + 8n^2) + 3^n \cdot n^2 + 2^n \cdot 2$$

$$(x^3 - 1)(x-1)^4(x-3)^3(x-2) = 0 \quad x = 1, 1, 1, 1, 1, 3, 3, 3, 2$$

$$t(n) = c_1 \cdot 1^n + c_2 \cdot n \cdot 1^n + c_3 \cdot n^2 \cdot 1^n + c_4 \cdot n^3 \cdot 1^n + c_5 \cdot n^4 \cdot 1^n + c_6 \cdot 3^n + c_7 \cdot n \cdot 3^n + c_8 \cdot n^2 \cdot 3^n + c_9 \cdot 2^n$$

$$0(t(n)) = 3^n \cdot n^2$$

### Cambio de variable

$$t(n) = \begin{cases} a, & n=1 \\ 2t\left(\frac{n}{2}\right) + b \cdot n, & n>1, b>0 \end{cases}$$

cambiar de "n" a "k"

$$\frac{n}{2} = \frac{2^k}{2} = 2^{k-1}$$

chuletillo

$$n = 2^k \quad k = \log n$$

cambio anticambio

$$t'(k) := t(n) = t(2^k)$$

$$\begin{aligned} ① \quad t(n) - 2t\left(\frac{n}{2}\right) &= b \cdot n \\ ①' \quad t'(k) - 2t'(k-1) &= b \cdot 2^k \end{aligned} \quad \text{reescribir con } t'(k)$$

$$② \quad (x-2)(x-2) = 0$$

situación 1

$$③ \quad x=2, 2$$

$$④ \quad t'(k) = c_1 \cdot 2^k + c_2 \cdot k \cdot 2^k \quad \text{vuelta atrás}$$

$$④ \quad t(n) = c_1 \cdot n + c_2 \cdot \log(n) \cdot n$$

$$t(n) = \begin{cases} n, & n < b \\ 3t(n-b) + n^2 + 1, & n \geq b \end{cases}$$

situación 2

$$① \quad t(n) - 3t(n-b) = n^2 + 1$$

$$①' \quad t'(k) - 3t'(k-1) = (bk)^2 + 1$$

$$② \quad (x-3)(x-1)^2 = 0$$

$$③ \quad x=3, 1, 1, 1.$$

$$④ \quad t'(k) = c_1 \cdot 3^k + c_2 \cdot 1^k + c_3 \cdot k \cdot 1^k + c_4 \cdot k^2 \cdot 1^k$$

$$④ \quad t(n) = c_1 \cdot 3^{\frac{n}{b}} + c_2 \cdot 1^{\frac{n}{b}} + c_3 \cdot \frac{n}{b} \cdot 1^{\frac{n}{b}} + c_4 \cdot \left(\frac{n}{b}\right)^2 \cdot 1^{\frac{n}{b}}$$

orden

chuleta

$$n = b \cdot k \quad k = \frac{n}{b}$$

$$t'(k) := t(n) = t(b \cdot k)$$

Cuando hacemos  $n=2^k$ , estamos restringiendo el estudio del orden a los  $n$ 's que sean potencia de 2.  $t(n) \in \Theta(f \mid P(n))$  condición

Se puede quitar la condición si:

$$\begin{cases} \text{Eventualmente no decreciente } (+ y f) \rightarrow \text{a partir de un } n \text{ ya no decrece} \\ f \text{ es } b\text{-armónica} \quad f(bn) \in \Theta(f(n)) \end{cases}$$

$$\text{Ejemplo: } t(n) = n^2 + n + 1 \in \Theta(n^2 \mid n=2^k)$$

$$\begin{cases} \text{END} \xrightarrow{+} \checkmark \\ f \text{ es } b\text{-armónica} \quad f(bn) = b \cdot n^2 \in \Theta(n^2) \checkmark \end{cases}$$

Ejemplo no armónica:

$$t(n) = 2^n + 1 \in \Theta(2^n \mid n=2^k)$$

$$f(x) = 2^x$$

$$f(b \cdot n) = (2^b)^n \in ? \Theta(2^n)$$

no porque  $2^b$  es una base MAYOR A 2



Ahora vamos a ver otras técnicas para recurrencias. En concreto, vamos a ver las no lineales.

No son métodos que siempre funcionan, son sólo técnicas:

- Expansión de recurrencia.
- Inducción constructiva.
- Transformación de imagen (no lo vamos a ver).

Vamos a ver expansión de recurrencia:

$$t(n) = \begin{cases} 1, & n=0 \\ 2t(n-1)+1, & n>0 \end{cases} \quad t(n) = 2t(n-1)+1 = 2[2t(n-2)+1] + 1 = \\ 2^2[2t(n-3)+1] + 2 + 1 = 2^3t(n-3) + 2^2 + 2 + 1$$

expandimos en busca de un patrón

$$= 2^i t(n-i) + 2^{i-1} + \dots + 2 + 1 \quad \stackrel{i=n}{\curvearrowright} \quad 2^n t(n-n) + 2^{n-1} + \dots + 2^0$$

$$\boxed{\sum_{j=0}^n 2^j = \frac{2^{n+1} - 2^0}{2-1} = 2^{n+1} - 1}$$

NO APLICAR SI HAY

MÁS DE UN TÉRMINO T

Ahora vemos la inducción constructiva:

no irá para examen!!

- sólo usar en última instancia (si no hay otra opción)
- te da el orden pero no la t

Hay que suponer el orden y entonces demostrarlo por inducción.

$$O(f) = \{ t : \mathbb{N} \rightarrow \mathbb{R}^+ / \exists d \in \mathbb{R}^+, \exists c_0 \in \mathbb{N} : t(n) \leq d \cdot f(n) \}$$

$$t(n) = \begin{cases} a, & n=1 \\ bn^2 + nt(n-1), & n>1 \end{cases} \quad \text{tiene pinta de factorial}$$

$t(n) \in \Theta(n!)$   $\rightarrow \epsilon O \rightarrow$  más compleja (en el texto guía)

$\epsilon \Omega$

CASO BASE:  $t(n) \geq c_1 \cdot n!$

$$t(1) = a \geq c_1 \cdot 1! \quad c_1 = a, \text{ se cumple}$$

CASO GENERAL:  $t(n-1) \geq c_2 \cdot (n-1)!$

$$\xrightarrow{??} nt(n-1) \geq c_2 \cdot n \cdot (n-1)!$$

$$nt(n-1) \geq c_2 \cdot n!$$

$$\underbrace{bn^2 + nt(n-1)}_{\text{t}(n)} \geq c_2 \cdot n!$$

$$\boxed{t(n) \geq c_2 \cdot n!}$$

demonstración  $\Omega$  completada!

Condición inicial: casos base de una ecuación recurrente.

Hay que elegir tantas condiciones como constantes salen en el paso 4.

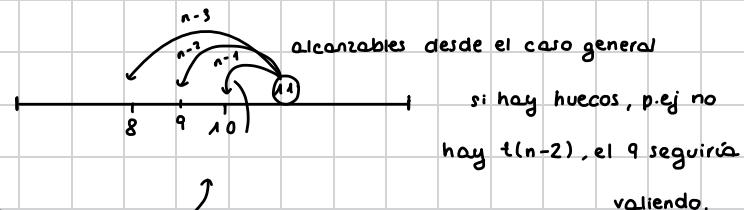
- Si hay cambio de variable, se ha de respetar.
  - Los casos deben ser alcanzados por la recurrencia general.

$$t(n) = \begin{cases} 0, & n=0 \\ 1, & n=1 \\ c_1 \cdot 2^n + c_2 \cdot n \cdot 2^n + c_3 \cdot 3^n, & n \geq 2 \end{cases}$$

A partir de ahora, deberemos elegir las condiciones iniciales correctas.

$$\begin{cases} t(1) : \boxed{\phantom{000}} = \boxed{\phantom{000}} \\ t(2) : \boxed{\phantom{000}} = \boxed{\phantom{000}} \\ t(3) : \boxed{\phantom{000}} = \boxed{\phantom{000}} \end{cases}$$

$\downarrow$  ecuación  
 $\downarrow$  ecuación paso 4  
enunciado



## Ejemplo:

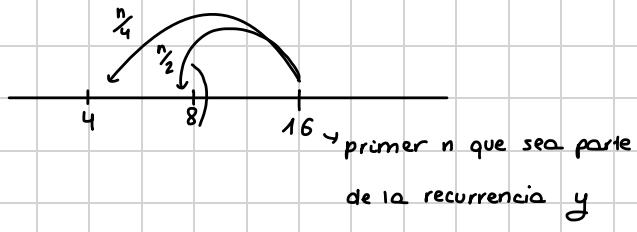
$$t(n) = \begin{cases} n, & n \leq 10 \\ 5t(n-1) - 8t(n-2) + 4t(n-3), & n > 10 \end{cases}$$

$$\textcircled{4} \quad t(n) = c_1 + c_2 \cdot 2^n + c_3 \cdot n \cdot 2^n$$

$$\begin{cases} t(8) = 8 &= c_1 + c_2 \cdot 256 + c_3 \cdot 2048 \\ t(9) = 9 &= c_1 + c_2 \cdot 512 + c_3 \cdot 4608 \\ t(10) = 10 &= c_1 + c_2 \cdot 1024 + c_3 \cdot 10240 \end{cases}$$

Ejemplo con cambio de variable:

$$t(n) = \begin{cases} n, & n \leq 8 \\ 2t\left(\frac{n}{2}\right) + t\left(\frac{n}{4}\right) + n + 1, & n > 9 \end{cases}$$



El cálculo de constantes también se puede aplicar en el estudio experimental.

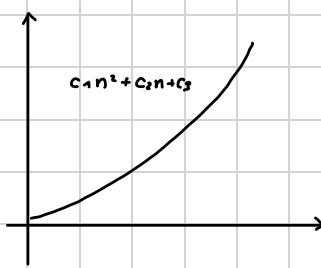
cumpla la condición

## Proceso:

1. Estimación teórica del tiempo de ejecución.
  2. Expresar el tiempo en función de constantes indefinidas.
  3. Tomar medidas del tiempo de ejecución para distintos tamaños de entrada.
  4. Resolver constantes.

$$t(n) = a(n+1)^2 + (b+c)n + d$$

$$t(n) = c_1 n^2 + c_2 \cdot n + c_3$$



Para sacar las constantes sacamos muchas medidas, grandes y separadas.

# Tema 2

## DIVIDE Y VENCERÁS

Miércoles 19 Febrero 2025

### 2.1. MÉTODO GENERAL

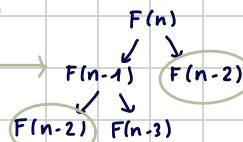
Dividir un problema en problemas más pequeños, para luego combinar las soluciones.

- Ventajas:

- Simplificar problemas "difíciles".
- Eficiencia: ayuda a descubrir algoritmos eficientes.
- Paralelismo
- Jerarquía de memoria: disminuye fallos de caché.

- Desventajas:

- Sobrecarga (dividir, combinar, recursividad)
- Cálculos repetidos.



#### Esquema general:

DivideVencerás ( $p$ : problema)

Dividir ( $p_1, p_2, \dots, p_n$ )

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

$s_i :=$  Resolver ( $p_i$ )

solución := Combinar ( $s_1, s_2, \dots, s_n$ )

a normalmente llamada recursiva

se puede meter cosas adicionales

#### Esquema recursivo:

DivideVencerás ( $p, q$ : índice)

var  $m$ : índice

si Pequeño ( $p, q$ ) entonces

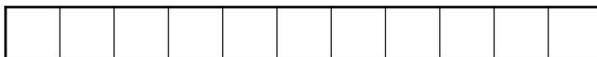
solución := SoluciónDirecta ( $p, q$ ) → no recursiva

sino

$m :=$  Dividir ( $p, q$ )

solución := Combinar (DivideVencerás ( $p, m$ ), DivideVencerás ( $m+1, q$ )) → dividimos en 2 subproblemas

finsi



$p$

$q$

Las funciones en verde son la clave del esquema.

Los requisitos para aplicar DyV son:

- 1 Método directo para tamaños pequeños
- 2 El problema debe dividirse fácilmente en subproblemas del mismo tipo con resolución más sencilla.
- 3 Los subproblemas deben ser DISJUNTOS.
- 4 Es necesario tener un método Combinar.

Normalmente, es bueno que los subproblemas tengan tamaños parecidos.

## 2.2 ANÁLISIS DE TIEMPOS DE EJECUCIÓN

$$a^{\log_b n} = n^{\log_b a}$$

$$t(n) : \begin{cases} 0, & n=1 \\ 2t(n/2) + n^p, & n>1 \end{cases}$$

ejemplo de sumar los valores de un vector  
tiempo de dividir y combinar

Si  $t(n) = a \cdot t(n/b) + d \cdot n^p$ , podemos deducir que:  $t(n) = \begin{cases} O(n^{\log_b a}) & \text{Si } a > b^p \quad \text{fórmula} \\ O(n^p \cdot \log n) & \text{Si } a = b^p \quad \text{maestra} \\ O(n^p) & \text{Si } a < b^p \end{cases}$

Ejemplo 1:

Dividimos en trozos  $n/2$ ,  $f(n) \in O(n)$

$$a = b = 2$$

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

Ejemplo 2:

4 llamadas recursivas, trozos  $n/2$ ,  $f(n) \in O(n)$

$$a = 4, b = 2$$

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

## 2.3 EJEMPLOS DE APLICACIÓN

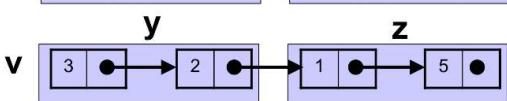
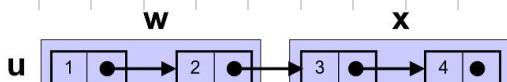
### Multiplicación rápida de enteros largos

Representamos enteros arbitrariamente largo con listas de cifras.

Una suma o resta tendría un orden de  $n$ .

Pero, ¿y la multiplicación?  $n \cdot n = n^2$ , si son los 2 de tamaño  $n$ .

¿Cómo aplicamos divide y vencerás?

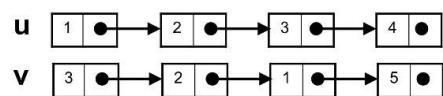


$$u = 43 \cdot 10^2 + 21$$

$$s = \lfloor \frac{n}{2} \rfloor$$

$$v = 51 \cdot 10^2 + 32$$

```
tipo EnteroLargo = Puntero[Nodo]
Nodo = registro
valor : 0..9
sig : EnteroLargo
finregistro
```



Dividimos los enteros de tamaño  $n$  en 2 trozos  $n/2$ .

Solución directa: multiplicación escalar cuando tamaño  $l$ .

$$u \cdot v = (w \cdot 10^3 + x)(y \cdot 10^3 + z) =$$

$$u \cdot v = w \cdot y \cdot 10^{2s} + w \cdot z \cdot 10^3 + x \cdot y \cdot 10^3 + x \cdot z$$

↓  
productos  
enteros largos  
de tamaño largo

añade ceros (tiempo  $n$ )

Nos sale que el orden será  $n^{\log_2 4} = n^2$

$a=4, b=2, p=1$  NO MEJORA !!

### Multiplicación rápida de enteros largos (Karatsuba y Ofman):

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

operación Mult( $u, v: \text{EnteroLargo}; n, \text{base}: \text{entero}$ ) :  $\text{EnteroLargo}$

si  $n == \text{base}$  entonces

devolver MultBasica( $u, v$ )

sino

asignar( $w, \text{primeros}(n/2, u)$ )

asignar( $x, \text{ultimos}(n/2, u)$ )

asignar( $y, \text{primeros}(n/2, v)$ )

asignar( $z, \text{ultimos}(n/2, v)$ )

asignar( $m1, \text{Mult}(w, y, n/2, \text{base})$ )

asignar( $m2, \text{Mult}(x, z, n/2, \text{base})$ )

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

devolver sumar(sumar(Mult10( $m1, n$ ),

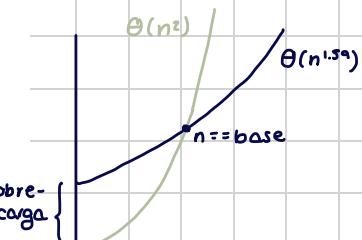
Mult10(sumar(sumar( $m1, m2$ ),  $m3$ ),  $n/2$ )),  $m2$ )

finsi

se parametriza el caso base porque para  $n$ 's pequeños es mejor el método directo

Ahora habrá 3 subproblemas en vez de 4.

$$\Theta(n^{\log_2 3}) = \Theta(n^{1.59})$$



Es interesante hacer un estudio sobre cuál sería el caso base óptimo para la práctica.

Multiplicación rápida de matrices

La típica tendría orden de  $n^3$  (multiplicación escalar  $\times$  filas  $\times$  columnas), asumiendo que la matriz es cuadrada ( $n \times n$ ).

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

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

$$a = 8 \quad b = 2 \quad n = 1 \longrightarrow O(n^3), \text{ no mejora}$$

si conseguimos reducir  $a$  (el nº de subproblemas), el orden mejorará.

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

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

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

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

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

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

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

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

Aplicando divide y vencerás:

8 problemas de  $n/2$

$$C_{11} = \boxed{A_{11}B_{11}} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + \boxed{A_{12}B_{22}}$$

$$C_{21} = A_{21}B_{11} + \boxed{A_{22}B_{21}}$$

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

$$t(n) = 8t(n/2) + f(n)$$

$$\downarrow \quad \downarrow \quad O(n^2)$$

$$\begin{matrix} n \log_2 \\ \downarrow \\ n^3 \end{matrix}$$

aquí reducimos a 7 subproblemas

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

$$n^{2.80}$$

**MEJORA YEI!!**

Aquí, igual que antes, el algoritmo normal funcionará mejor para  $n$ 's pequeños.

**COTA DE COMPLEJIDAD DEL PROBLEMA :** tiempo del algoritmo más rápido posible que resuelve el problema.

Métodos de ordenación rápido

El algoritmo clásico tiene un orden de  $n^2$  (recorrer el array tamaño  $n$ ,  $n$  veces).

$$t(n) \in \Theta(n^2)$$

1 MERGE SORT

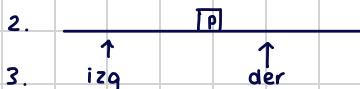
Dividimos el array a la mitad y combinamos. Combinar 2 arrays de tamaño  $n$ , tiene un orden exacto de  $n$ .

$$t(n) = 2t(n/2) + f(n) \quad t(n) \in \Theta(n \cdot \log n)$$

## 2) QUICK SORT

0. Si pequeño  $\rightarrow$  caso base

1. Escoger  $p$  aleatorio



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

La ordenación del  
pivot (solo el pivot)  
es  $O(n)$ .

operación Pivote ( $i, j$ : entero; var  $l$ : entero)  
var  $p$ : tipo {  $p$  es el pivot, del tipo del array }  
 $k$ : entero

```

 $p := A[i]$  { se toma como pivot el primer elemento }
 $k := i$ 
 $l := j + 1$ 
repetir  $k := k + 1$  hasta ( $A[k] > p$ ) or ( $k \geq j$ )
repetir  $l := l - 1$  hasta ( $A[l] \leq p$ )
mientras  $k < l$  hacer
    intercambiar ( $k, l$ )
    repetir  $k := k + 1$  hasta ( $A[k] > p$ )
    repetir  $l := l - 1$  hasta ( $A[l] \leq p$ )
finmientras
intercambiar ( $i, l$ )

```

ver detenidamente  
en casa

$$t(n) = t(p_1(n)) + t(p_2(n)) + f(n)$$

$\uparrow$   
 $n$

se usa más quick sort!  
porque en mergesort se  
usa mucha memoria

$$t(n) = 2t(h/2) + f(n) \leftarrow \text{mejor caso}$$

$\Theta(n \cdot \log n)$   $\uparrow$  caso promedio

$$t_{\text{w}}(n) = t(n-1) + t(n) \leftarrow \text{peor caso}$$

$\Theta(n^2)$

Martes 4 Marzo 2025

### Problema de selección

Queremos seleccionar un elemento que estaría en esa posición si el array estuviera ordenado.

Ejemplo:

ENTRADA
{1, 7, 5, 10}
$s = 2$
SALIDA
7

Se podría hacer ordenando el array primero y luego sacando el elemento.  $(O(n \log n))$

Otra manera es usando un pivot. Se escoge un elemento al azar y se ordena en el array. Si el  $s$  que buscamos es igual que el pivot, se devuelve el elemento, si es menor, nos vamos a poner un pivot al lado izquierdo.

- Mejor caso  $\rightarrow \Theta(n)$  supera al método ingenuo
- Caso promedio  $\rightarrow \Theta(n)$
- Peor caso  $\rightarrow \Theta(n^2)$  MUY IMPROBABLE

## Problema del derbi

Encontrar el nº de conflictos en celdas.

Usando un algoritmo simple tenemos un orden de  $n^2$  (recorrer la matriz).

	B	
B	A	B
	B	

Aplicamos divide y vencerás dividir la matriz en 4

$$\begin{pmatrix} & A_{11} & A_{12} \\ & \hline A_{21} & A_{22} \end{pmatrix}$$

Para combinar sumaremos los conflictos de las 4 submatrices y luego comprobamos que haya en las diagonales principales (orden de  $n$ ):

$$t(n) = 4t(n/2) + f(n), \quad f(n) \in \Theta(n)$$

$$a = 4, b = 2$$

$$\log_2 4 = n \quad t(n) \in \Theta(n^2) \quad \text{MALO!!}$$

## Subsecuencia más larga

La subsecuencia más larga que cumple:

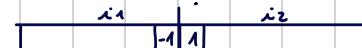
- Ascendente
  - Simétrica
  - Pares

•

Caso base  $\rightarrow$   $DyV(\text{arr}, i, j)$   $i=j$   $\text{return}(i, j)$

Caso recursivo  $\rightarrow i_1, i_2$

## Cómo combinamos?



hasta que se rompa

devolver el máximo entre la subsecuencia de la izq, de la derecha y la de en medio.  
( $i_1$ ) ( $i_2$ )

El mejor caso es que los punteros  $i$  y  $j$  no se muevan, es decir, que el array esté ordenado al revés.  $t(n) \in O(n)$

El peor caso → que este ordenado ascendente:  $t(n) \in O(n \cdot \log n)$



# Tema 3

## ALGORITMOS VORACES

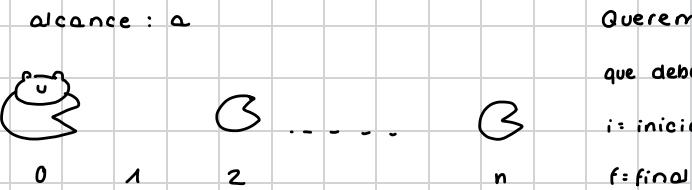
(greedy)

Tomar una decisión y no volver atrás.

Son algoritmos que avanzan rápido y toman decisiones conforme avanzan. Se suelen utilizar en problemas de optimización. Se pueden implementar con recursividad.

Se da un problema  $P$  y un conjunto  $S$  solución vacía. Se toma una decisión y se añade al conjunto  $S$ , obteniendo  $P'$  (un problema más pequeño pero de la misma naturaleza de  $P$ ). Cuando se llega a una solución final se termina.

### PROBLEMA DE LA RANA



Queremos dar el mínimo nº de saltos que debe dar la rana hasta llegar a  $n$ .

Vamos tomando decisiones: el nenúfar más alejado alcanzable por la rana (alcance).

¿Es la solución óptima?

Es óptimo si se cumple:

- Existe una solución óptima de  $P$  vamos bien encaminadas (propiedad de decisión voraz)
- Sea  $l$  la decisión voraz para  $P$  y  $S'$  la solución óptima al problema  $P'$ , entonces  $l$  unida a  $S'$  es una solución óptima de  $P$  (subestructura óptima).

Técnicas para demostrarlo:

- Propiedad de la decisión voraz:

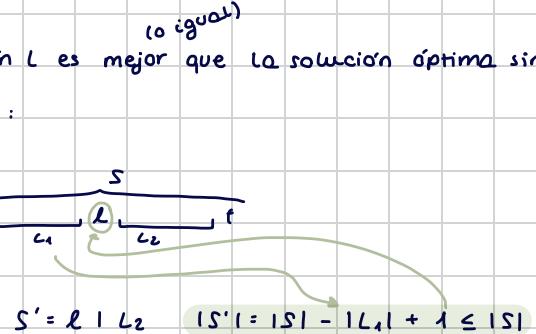
demostrar que la solución tomando la decisión  $L$  es mejor que la solución óptima sin  $L$ .

Usando el caso de la rana tenemos 3 casos:

1)  $L$  está en  $S$  y es el primero

2)  $L$  está en  $S$  y no es el primero

3)  $L$  no está en  $S$ :  $S = L_1 | L_2$



• Subestructura óptima: se demuestra por reducción al absurdo (negando la conclusión y llegando a una contradicción en 3 casos)

$$|W| \leq |S'| \cup \{l\} = |S'| + 1$$

(ver en el github)

## MOCHILA NO 0/1

Tenemos un conjunto de objetos  $O = \{1, \dots, n\}$ ,  $n$  objetos con peso  $p_i > 0$  beneficios  $b_i > 0$ .

Podemos fraccionar objetos.

No podemos pasarnos del peso máximo ( $M$ ).

El problema P está parametrizado por  $(M, O)$ .

$n=3, M=20$

$p = (18, 15, 10)$

$b = (25, 24, 15)$

$S_1 = (1, 2/3, 0)$ , beneficio = 28.2

$S_2 = (0, 2/3, 1)$ , beneficio = 31

- 1) Tomamos la decisión y añadimos S. La decisión voraz es el objeto  $l$  que metemos y su proporción  $x_l$ .
- 2) Construimos  $P(M - x_l p_l, O - \{l\})$
- 3) Volver a ejecutar 1 y 2

Criterio para la selección de objetos:

- El de mayor beneficio  $X$
  - El de menor peso  $x$
  - El de mejor proporción  $b_l/p_l$   
↓ demostración en el power point
- } podemos demostrar que no son óptimos con un contraejempl

Martes 11 Marzo 2025

Sin embargo, el PROBLEMA DE LA MOCHILA 0/1, es decir, en el que no se pueden fraccionar objetos no se resuelve con algoritmo voraz. Es un problema NP-Completo.

## CAMBIO DE MONEDAS

Tenemos  $n$  monedas con  $V = v_1, v_2, \dots, v_n$ . El objetivo es devolver la cantidad  $C$  minimizando el nº de monedas.

→ no hay límite de cada moneda

El problema está parametrizado por  $C$  (cantidad) y  $O$  (monedas disponibles).

Decisión voraz: seleccionar la moneda de más valor y en su máxima cantidad sin pasarse de  $C$ .

No en todos los sistemas monetarios da la solución óptima ( $\epsilon$  y  $\$$  sí). Los sistemas monetarios a los que sí se llega a la solución óptima se llaman CANÓNICOS.

### PLANIFICACIÓN DE TAREAS:

Tareas requieren 1 unidad de tiempo, beneficio  $b_i$  y plazo  $p_i$ ; si se pasa del plazo no hay beneficio.

Objetivo: secuencia que maximice beneficio. Ejemplo:

$$b = (20, 15, 10, 7, 5, 3)$$

$$p = (3, 1, 1, 3, 1, 3)$$

Instante	1	2	3	4	5	6
s	1	3	0	2	4	5

$$B(s) = 15 + 7 + 20 + 0 + 0 + 0 = 42$$

Está parametrizado por el instante actual y el nº tareas.

Decisión voraz: seleccionas el mayor beneficio pero colocando inteligentemente la tarea en el tiempo.

### ESQUEMA VORAZ

voraz (var s: Cjto Solución)

$S := \emptyset$

C = generar Candidatos (S, ...)

mientras ( $C \neq \emptyset$ ) Y NO solución(S) hacer

$x :=$  seleccionar ( $C$ )

$C := C - \{x\}$

si factible ( $S, x$ ) entonces

insertar ( $S, x$ )

C = generar Candidatos (S, ...)

finsi

fin mientras

si NO solución (S) entonces

devolver "No se puede resolver"

finsi

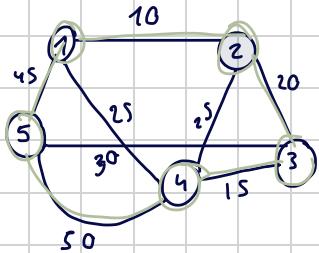
Hay problemas en los que los candidatos siempre son los mismos (mochila, monedas...) y otros en los que no (ranita).

Usaremos este esquema para las prácticas.

## EL PROBLEMA DEL VIAJANTE

Dado un grafo completo no dirigido con pesos. Queremos encontrar el ciclo Hamiltoniano.

Es NP-completo.



Decisión voraz: ir eligiendo el camino mínimo entre 2 nodos no conectados.

## RECORRIDO POR MANHATTAN:

Tenemos un grafo  $G = (V, E)$  en forma de Grid. Aristas con pesos. Objetivo: encontrar el camino más largo. Solo se puede bajar y ir a la derecha.

Parametrizado por posición inicial y final.

Decisión voraz: elegir la arista con más peso.

NO ES ÓPTIMO!!  $\therefore$  Pero no es NP-Completo

# Tema 4

## PROGRAMACIÓN DINÁMICA

### PROGRAMACIÓN DINÁMICA CON MEMORIZACIÓN

La resolución típica del problema del término  $n$ -ésimo de la sucesión Fibonacci da un algoritmo muy lento, de orden exponencial ( $\sqrt{2}^n$ ).

```
def fibo(n, memo):
    if n in memo:
        return memo[n]
    if n ≤ 1:
        f=1
    else:
        f=fibo(n-1, memo) + fibo(n-2, memo)
        memo[n]=f
    return f;
```

Este algoritmo alternativo que usa una caché (memo) tiene un orden de  $n$ .

El uso de la caché hace que si se cumple el primer if sea constante.

En general, se consigue que:

$$t(n) = t(n-1) + \Theta(1)$$

(asumiendo caché infinita)

"Una vez resuelto un subproblema, lo guardamos en memoria".

El orden de estos algoritmos:

#subproblemas diferentes x tiempo trabajo no recursivo / subproblema

### PROGRAMACIÓN DINÁMICA ASCENDENTE

En vez de usar caché o recursión, podemos resolverlo apuntando las soluciones de los subproblemas en un array (es iterativo). Suele ser más rápida que el de memorización.

#### 1. Estudiamos las dimensiones de la tabla

Cuando hay  $n$  subproblemas, la tabla será de dimensión  $n$  y cada  $i$  contendrá la solución  $i$ -ésima.

#### 2. Estudiar el DAG de los subproblemas

DAG → grafo de dependencias, para Fibonacci

Definimos el orden topológico (donde empezamos a rellenar la tabla).



3. Se rellena la tabla siguiendo el orden seleccionado

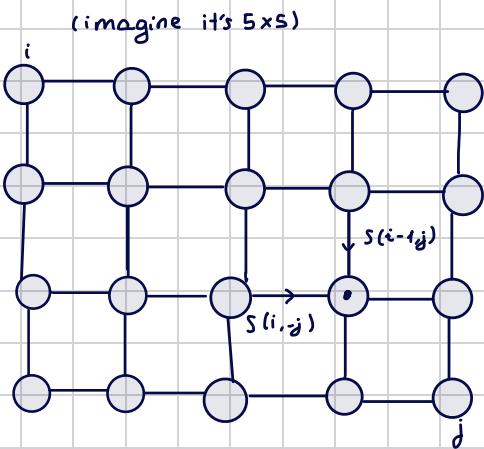
Para aplicar programación dinámica:

- 1 Sacar la ecuación de recurrencia.
- 2 Escoger la técnica PD.
- 3 Recomponer la solución a partir de los valores de la tabla.

¿Cuándo funciona PD?

- Subestructura óptima
- Intersección de subproblemas
- Subproblemas polinomiales (número de subproblemas).

### RECORRIDO POR MANHATTAN



Primeramente definimos la ecuación de recurrencia.

$S(i,j) \rightarrow$  camino que más pesa de  $i$  a  $j$

$$S(i,j) = \max(S(i-1,j) + \text{arriba}, S(i,j-1) + \text{izq})$$

$$S(i,j) = \begin{cases} 0 & i=j=0 \\ S(i-1,j) + \text{arriba} & j=0 \\ S(i,j-1) + \text{izq} & i=0 \\ \max(S(i-1,j) + \text{arriba}, S(i,j-1) + \text{izq}) & \text{resto} \end{cases}$$

Recordemos que este problema trata de encontrar el camino con más peso que va desde  $i$  hasta  $j$ , solo pudiendo movernos hacia abajo y hacia la derecha. Asumimos que el tablero es cuadrado ( $n \times n$ ).

Si escribimos con código la recursión definida anteriormente nos sale una complejidad exponencial.

El segundo paso es escoger la técnica PD y aplicarla.

Vamos a usar memorización:

```

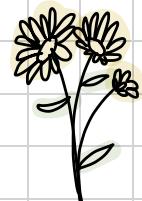
def Manhattan_memo(i, j, memo):
    if (i, j) in memo:
        return memo[i, j]
    if i == j == 0:
        m = 0
    else:
        if j > 0:
            e_arriba = down[i - 1][j]
            m1 = Manhattan_memo(i - 1, j, memo) + e_arriba
        else:
            m1 = float("-inf")
        if j > 0:
            e_izq = right[i][j - 1]
            m2 = Manhattan_memo(i, j - 1, memo) + e_izq
        else:
            m2 = float("-inf")
        m = max(m1, m2)
        memo[i, j] = m
    return m

```

¿Orden del algoritmo?  
 nº subprobdiff x tiempo / subprob =  
 $n^2 \cdot \Theta(1) = \Theta(n^2)$

Hay  $n^2$  subproblemas  
 (recordemos que  $i=0\dots n$  y  
 $j=0\dots n$ ), pero se  
 resuelven en orden constan-  
 te cada uno.

OJO, no siempre será  
 constante el tiempo por  
 subproblema.



Viernes 21 Marzo 2025

paso 3

Vale, tenemos el peso, pero no el camino en sí, ¿cómo recomponemos la resolución?

En memo se ha ido guardando el camino. Para recuperarlo

$$\begin{pmatrix} & \hline 9 & 14 & 22 & 22 & 25 \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 14 & 20 & 30 & 32 & 34 \end{pmatrix} \quad \begin{array}{l} 34 = \text{max } \underline{\text{izq}} = 32+2, \text{ arr} = 25+3 \\ 32 = \text{max } \underline{\text{izq}} = 30+2, \text{ arr} = 22+5 \\ \dots \text{ hasta } (0,0) \end{array}$$

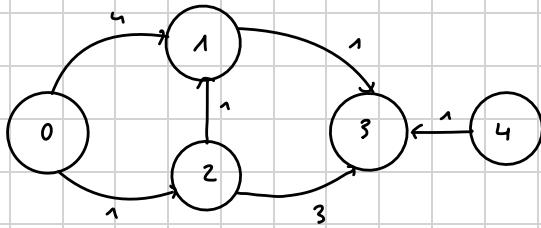
se crea una función para reconstruir

Hemos resuelto el problema por memorización, ahora lo resolveremos por DP ascendente.

DAG:  $S_{00} \leftarrow S_{01} \leftarrow S_{02}$  Luego se implementa usando un array, en el orden en el  
 $\uparrow \quad \uparrow \quad \uparrow$  que hayamos elegido recorrerlo (horizontal o vertical).  
 $S_{10} \leftarrow S_{11} \leftarrow S_{12}$   
 $\uparrow \quad \uparrow \quad \uparrow$   
 $S_{20} \leftarrow S_{21} \leftarrow S_{22}$

## BELLMAN - FORD EN UN DAG

Objetivo: calcular el camino de  $s$  a todos los nodos.



$G = \{$

$0: []$  aristas INCIDENTES

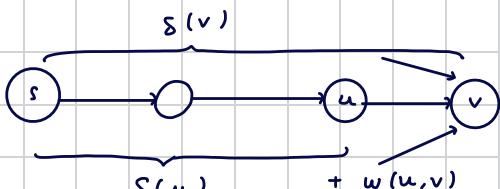
$1: [(0,4), (2,1)]$  ↓ peso  
↓ nodo

$2: [(0,1)]$

$3: [(1,1), (2,3), (4,1)]$

$4: []$

$\}$



CASO BASE:

$\delta(v) = 0$  si  $s = v$

$\delta(v) = +\infty$  si no hay aristas incidentes

RECURRENCIA:

$$\delta(s, v) = \min \{ \delta(s, u) + w(u, v) \}$$

¿Qué orden tiene con memorización?

$|V|$  subproblemas (nº nodos)

$C \text{ indegree}(v) + C'$

$$\Theta(|V| + |E|)$$

¿Por qué pedimos que no haya ciclos?

Bucle infinito  $\infty$ .

Martes 25 Marzo 2025

## MOCHILA 0/1

Meter objetos en una mochila buscando que no se pase del peso y obtener el beneficio máximo.

Los objetos no se pueden fraccionar.

$P(j, m)$  = beneficio de considerar objetos de  $1 \dots j$  con capacidad  $m$

$\begin{cases} b_j + P(j-1, m-p_j) & \rightarrow \text{cojer o no cojer el objeto } j \\ P(j-1, m) & \end{cases}$

$P(j, m) = \max \{ b_j + P(j-1, m-p_j), P(j-1, m) \}$  RECURRENCIA

CASOS BASE

$\rightarrow m < 0 \text{ o } j < 0 \Rightarrow -\infty$

$\rightarrow j = 0 \text{ y } m \geq 0 \Rightarrow 0$

$\rightarrow m = 0, j \geq 0 \Rightarrow 0$

con memorización

```

def mochila_memo (P, B, m, j, memo):
    if (j, m) in memo:
        return memo[(j, m)]
    if m < 0 || j < 0:
        r = -∞
    elif j == 0 and m ≥ 0:
        r = 0
    elif m == 0 and j ≥ 0:
        r = 0
    else:
        r1 = B[j-1] + mochila_memo (P, B, m-P[j-1], j-1, memo)
        r2 = mochila_memo (P, B, m, j-1, memo)
        r = max(r1, r2)
        memo[(j, m)] = r
    return r

```

Ahora vamos a verlo de manera ascendente.

Dimensiones de la tabla:

$$(n+1) \times (M+1)$$

nº objetos      capacidad mochila

$$n=3 \quad M=6 \quad p=(2,3,4) \quad b=(1,2,5)$$

	0	1	2	3	4	5	6	M
0	P <sub>00</sub>	P <sub>01</sub>	P <sub>02</sub>	P <sub>03</sub>	P <sub>04</sub>	P <sub>05</sub>	P <sub>06</sub>	
1	P <sub>10</sub>	P <sub>11</sub>	P <sub>12</sub>	P <sub>13</sub>	P <sub>14</sub>	P <sub>15</sub>	P <sub>16</sub>	
2	P <sub>20</sub>	P <sub>21</sub>	P <sub>22</sub>	P <sub>23</sub>	P <sub>24</sub>	P <sub>25</sub>	P <sub>26</sub>	
3	P <sub>30</sub>	P <sub>31</sub>	P <sub>32</sub>	P <sub>33</sub>	P <sub>34</sub>	P <sub>35</sub>	P <sub>36</sub>	

es aplicar la fórmula

P<sub>36</sub> depende del P<sub>26</sub> y del P<sub>22</sub>

porque: P(2,6)

P(3,6) ↗ no cogerlo

cogerlo ↗ P(2,2)

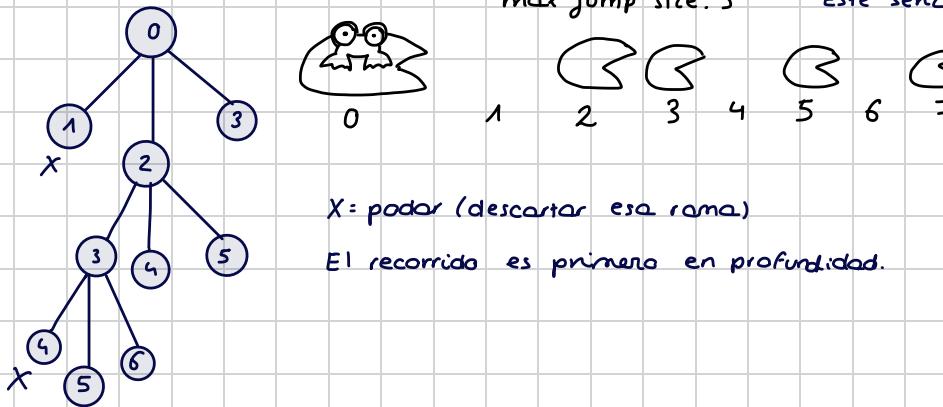
Para rellenar la tabla lo hacemos por filas.

# Tema 5

Martes 1 Abril 2025

Backtracking: búsqueda exhaustiva del espacio de soluciones.

- Problemas de decisión → 1 solución
- Problemas de enumeración → todas las soluciones
- Problemas de optimización → la mejor solución



Antes de resolver un problema vemos que tipo de árbol va implícito.

## TIPOS DE ÁRBOLES

- Binarios: cada nodo tiene 2 hijos.  
Sirven para cuando hay que escoger o no un objeto
- $k$ -arios:  $k$  hijos  
Por ejemplo para el cambio de monedas
- Permutacionales:  $n!$  hojas : sirven para asignaciones → fijarse en las aristas
- Arboles combinatorios : sirven para los mismos problemas que los binarios (rana por ejemplo)

Los combinatorios se pueden mapear a binarios y viceversa.

## ESQUEMA BACKTRACKING GENERAL (OPTIMIZACION):

```

1 def backtracking_optimizacion_minimizar():
2     nivel = 1
3     s = s_0
4     voa = float("inf") # valor óptimo actual
5     soa = None # solución óptima actual
6
7     while nivel != 0:
8         # genero un hermano
9         generar(nivel, s)
10
11        # si he encontrado una solución que mejora a voa
12        # lo guardo
13        if solucion(nivel, s) and valor(s) < voa:
14            voa = valor(s)
15            soa = s
16
17        # si podríamos llegar a la solución final, seguimos
18        if criterio(nivel, s):
19            nivel += 1
20        else:
21            # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
22            # por explorar
23            while nivel > 0 and (not masHermanos(nivel, s)):
24                retroceder(nivel, s)

```

*s* almacena la solución parcial.

*s<sub>0</sub>* almacena el valor de iniciación.

*nivel*: indica el nivel en el que está el algoritmo.

*generar(nivel, s)*: genera el siguiente hermano para el nivel actual.

*solucion(nivel, s)*: comprueba si la tupla actual es solución.

*criterio(nivel, s)*: comprueba si a partir de la solución actual es posible llegar a la solución final.

*masHermanos(nivel, s)*: devuelve True si hay más hermanos del nodo actual.

*retroceder(nivel, s)*: retrocede un nivel del árbol actualizando *s* si fuese necesario.

Miércoles 2 Abril 2025

## SUMA DEL SUBCONJUNTO:

Conseguir el subconjunto que sume un entero *K*, escogiendo el subconjunto de un array de enteros.

Se resuelve con árbol binario (se coge o no el número).

Se inicializa todo a -1 (meaning no se ha explorado).

```

1 def backtracking_decision():
2     nivel = 1
3     s = s_0
4
5     while nivel != 0:
6         # genero un hermano
7         generar(nivel, s)
8
9         # si he encontrado la solución termino
10        if solucion(nivel, s):
11            return s
12
13        # si podemos llegar a la solución final en niveles inferiores
14        if criterio(nivel, s):
15            nivel += 1
16        else:
17            # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
18            # por explorar
19            while nivel > 0 and (not masHermanos(nivel, s)):
20                retroceder(nivel, s)
21
22    print("No hay solución")

```

Lo que tarda:

nºnodos x tiempo/nodo

Podemos optimizar llevando una variable "suma":

```

def subconjunto_backtracking_optimizado(A, M):
    nivel = 1
    s = [-1]*len(A)
    suma = 0
    while nivel != 0:
        # genero un hermano
        generar(nivel, s)
        if s[nivel-1] == 1:
            suma += A[nivel - 1]
        # si he encontrado la solución termino
        if solucion_optimizada(nivel, A, M, suma):
            return s
        # si renta seguir explorando
        if criterio_optimizado(nivel, A, M, suma):
            nivel += 1
        else:
            # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
            # por explorar
            while nivel > 0 and (not masHermanos(nivel, s)):
                suma -= s[nivel-1]*A[nivel - 1]
                s[nivel-1] = -1
                nivel -= 1
    print("No hay solución")

```

```

1 def generar(nivel, s):
2     s[nivel-1] += 1
3
4 def solucion(nivel, s, A, M):
5     suma = 0
6     for i in range(0, nivel):
7         suma += A[i]*s[i]
8     return suma == M and len(A) == nivel
9
10 def criterio(nivel, s, A, M):
11     suma = 0
12     for i in range(0, nivel):
13         suma += A[i]*s[i]
14     return suma <= M and nivel < len(A)
15
16 def masHermanos(nivel, s):
17     return s[nivel-1] == 0

```

```

1 def solucion_optimizada(nivel, A, M, suma):
2     return suma == M and len(A) == nivel
3
4 def criterio_optimizado(nivel, A, M, suma):
5     return suma <= M and nivel < len(A)

```

Ahora queremos devolver TODAS LAS SOLUCIONES (enumeracin):

```

def subconjunto_backtracking_enumeracion(A, M):
    nivel = 1
    s = [-1]*len(A)
    suma = 0
    S = []
    while nivel != 0:
        # genero un hermano
        generar(nivel, s)
        if s[nivel-1] == 1:
            suma += A[nivel - 1]

        # si he encontrado la solucion termino
        if solucion_optimizada(nivel, A, M, suma):
            S.append(s.copy())

        # si renta seguir explorando
        if criterio_optimizado(nivel, A, M, suma):
            nivel += 1
        else:
            # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
            # por explorar
            while nivel > 0 and (not masHermanos(nivel, s)):
                suma -= s[nivel-1]*A[nivel - 1]
                s[nivel-1] = -1
                nivel -= 1

    return S

```

## CREAR UM ALGORITMO BACKTRACKING

1. Representación de la tupla solución
  2. Elección del árbol
  3. Definir las funciones
  4. Optimizar: o bien reduciendo el tiempo por nodo, o teniendo una buena función de poda (para no recorrer todo el árbol).

## BACKTRACKING VS PROGRAMACIÓN DINÁMICA

La programación dinámica es como un backtracking pero con una gran poda.

## MOCHILA 0/1

(Con backtracking)

$B: [b_1, \dots, b_n]$   $P: [p_1, \dots, p_n]$   $M$  hay que llevar el beneficio actual y la suma actual

Es un problema de optimización (maximización).

Vamos a representarlo con un árbol binario (0, no se coge el objeto; 1, si se coge).

- $s_0 = (-1, \dots, -1)$
  - generar(nivel, s): genera el siguiente hermano  $\sim ++s[nivel]$ .
  - solución(nivel, s): ¿hemos analizado todos los objetos? (es decir, ¿estamos en el último nivel?) y ¿nos hemos pasado  $M$ ?
  - criterio(nivel, s): ¿nos hemos pasado  $M$ ? ¿quedan objetos por analizar?
  - masHermanos(nivel, s): ¿hemos analizado ambas posibilidades del objeto?
  - retroceder(nivel, s): pasamos al objeto anterior.

## Implementación de la solución:

voa → valor óptimo actual    soa → solución óptima actual

```

1 def backtracking_optimizacion_maximizar():
2     nivel = 1
3     s = s_0
4     voa = -float("inf")
5     soa = None
6
7     while nivel != 0:
8         # genero un hermano
9         generar(nivel, s)
10
11        # si he encontrado una solución que mejora a voa
12        # lo guardo
13        if solucion(nivel, s) and valor(s) > voa:
14            voa = valor(s)
15            soa = s
16
17        # si podríamos llegar a la solución final, seguimos
18        if criterio(nivel, s):
19            nivel += 1
20        else:
21            # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
22            # por explorar
23            while nivel > 0 and (not masHermanos(nivel, s)):
24                retroceder(nivel, s)
25
26    return soa

```

```

1 def mochila(B, P, M):
2     nivel = 1
3     s = [-1]*len(B)
4     voa = -float("inf")
5     soa = None
6     peso_actual = 0
7     beneficio_actual = 0
8
9     while nivel != 0:
10        # generar
11        s[nivel - 1] += 1
12        peso_actual += P[nivel - 1]*s[nivel - 1]
13        beneficio_actual += B[nivel - 1]*s[nivel - 1]
14        # fin generar
15
16        # solución
17        if peso_actual <= M and len(B) == nivel and beneficio_actual > voa:
18            voa = beneficio_actual
19            soa = s.copy()
20
21        # criterio
22        if peso_actual <= M and nivel < len(B):
23            nivel += 1
24        else:
25            # masHermanos y retroceder
26            while nivel > 0 and (not s[nivel-1] <= 0):
27                beneficio_actual -= s[nivel-1]*B[nivel-1]
28                peso_actual -= s[nivel-1]*P[nivel-1]
29                s[nivel - 1] = -1
30            nivel -= 1
31
32    return soa, voa

```

Sin embargo, hay muchos casos donde la función criterio no podo nada . por ejemplo, si la suma de los  $n-1$  primeros pesos de los objetos no excede a  $M$ .

Debemos mejorar la función criterio:

### COTA SUPERIOR FIABLE

Antes de meternos a explorar una rama, estimar el máximo beneficio que se podría llegar a obtener.

### Inicialización y variables auxiliares

```

1 s=[-1]*len(B)
2 beneficio_actual = 0
3 peso_actual = 0

```

### Generar

```

1 s[nivel - 1] += 1
2 peso_actual += P[nivel - 1]*s[nivel - 1]
3 beneficio_actual += B[nivel - 1]*s[nivel - 1]

```

### Solución

```
1 peso_actual <= M and len(B) == nivel
```

### Criterio

```
1 peso_actual <= M and nivel < len(B)
```

### Más hermanos

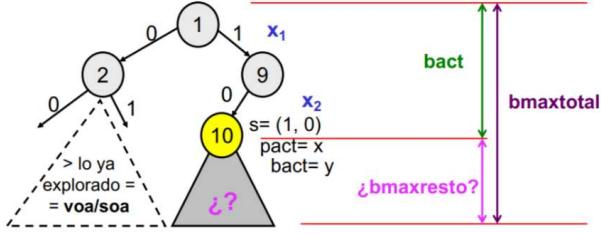
```
1 s[nivel-1] <= 0
```

### Retroceder

```

1 beneficio_actual -= s[nivel-1]*B[nivel-1]
2 peso_actual -= s[nivel-1]*P[nivel-1]
3 s[nivel - 1] = -1
4 nivel -= 1

```



Si  $voa \geq bmaxtotal = bac1 + bmaxresto$ , entonces no exploramos.

¿Cómo calculamos  $bmaxresto$ ?

Opción 1: beneficio de meter todos los objetos restantes.

mejor porque sale una cota más tight

Opción 2: usar algoritmo voraz de mochila no 0/1 considerando  $peso_{max} = M - peso_{actual}$

¿Por qué?

↓ el beneficio de usar mochila no 0/1 será mayor o igual a usar mochila 0/1



demostración: si es siempre una solución válida para mochila no 0/1.

## Criterio

```
1 beneficio_estimado = int(mochila_voraz(set([i for i in range(nivel, len(B))]),
2 B, P, M - peso_actual))
3 peso_actual <= M and nivel < len(B) and (beneficio_estimado + beneficio_actual > voa)
```

Otra optimización es ordenar el árbol de manera que  $voa$  se encuentre en las primeras hojas y que el criterio voraz pade más. ¿Cómo?

① Explorando primero la opción de meter un objeto.

② Ordenando los objetos usando  $b_i/p_i$  (la solución de la mochila no 0/1 se parece a la de la mochila 0/1). Si ordenamos así, el algoritmo voraz será de  $O(n)$ .

Sin embargo, con esta mejora, en el peor caso estamos ante un orden  $O(n \cdot 2^n)$ , pero, en la práctica, al poder más:  $t(n) \sim n^{\alpha}$  con  $\alpha < 2$  (que es mucho mejor que  $2^n$ ).

Vamos a probar ahora a hacerlo con árbol combinatorio:

Generar

```
1 if s[nivel-1] == 0: # primer hermano?
2     if nivel == 1: # primer nivel?
3         s[nivel - 1] = 1
4         peso_actual += P[0]
5         beneficio_actual += B[0]
6     else:
7         # se genera el siguiente al padre
8         # y se actualiza el peso y beneficio actual
9         s[nivel - 1] = s[nivel - 2] + 1
10        peso_actual += P[s[nivel - 1] - 1]
11        beneficio_actual += B[s[nivel - 1] - 1]
12    else:
13        # generar el siguiente hermano
14        # se genera el siguiente al que había
15        # se actualizan las var aux restando y sumando
16        peso_actual -= P[s[nivel - 1] - 1]
17        beneficio_actual -= B[s[nivel - 1] - 1]
18        s[nivel - 1] += 1
19        peso_actual += P[s[nivel - 1] - 1]
20        beneficio_actual += B[s[nivel - 1] - 1]
```

## Inicialización y variables auxiliares

```
1 s=[0]*len(B)
2 beneficio_actual = 0
3 peso_actual = 0
```

## Solución

```
1 peso_actual <= M # ya que todos los nodos son solución
```

## Criterio

```
1 beneficio_estimado = int(mochila_voraz(set([i for i in range(s[nivel - 1], len(B))]), B, P,  
2 M - peso_actual))  
2 peso_actual <= M and s[nivel-1] < len(B) and (beneficio_estimado + beneficio_actual > voa)
```

## Más hermanos

```
1 s[nivel-1] < len(B)
```

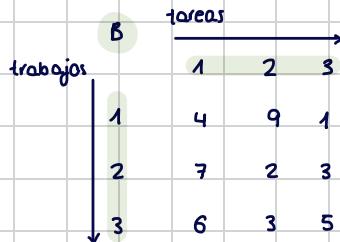
## Retroceder

```
1 beneficio_actual -= B[s[nivel - 1] - 1]  
2 peso_actual -= P[s[nivel - 1] - 1]  
3 s[nivel - 1] = 0  
4 nivel -= 1
```

normalmente, los árboles combinatorios exploran una menor cantidad de nodos que los binarios.

Martes 29 Abril 2025

## ASIGNACIÓN DE TRABAJOS



Buscar la combinación que dé el máximo beneficio. Si nos damos cuenta, esto es una permutación.

Podemos usar un árbol permutacional, ya que no se pueden repetir tareas.

En generar(): 2 casos

si es el primer hermano

si no

$s[nivel-1] += 1$

$\{ \text{if } s[nivel-1] == 1 \text{ //primer hermano}$

$bact += B[nivel-1][s[nivel-1]-1]$

$\{ bact += B[nivel-1][s[nivel-1]-1]$

$\{ bact -= B[nivel-1][s[nivel-1]-2]$

se le resta el beneficio del anterior

Solución()

```
{ if nivel < n  
    return False  
    for i in range(0, nivel - 1):  
        if s[i] == s[nivel - 1]  
            return False  
    return True
```

criterio()

```
{ for i in range(0, nivel - 1)  
    if s[i] == s[nivel - 1]  
        return False  
return nivel < n
```

Para optimizar llevar array con tareas asignadas y sin asignar (true/false). !