



Algoritmos Estructuras de Datos I

Facultad de Ciencias Exactas y Tecnología
Universidad Nacional de Tucumán

2025

Complejidad

Notación O grande(3)

Complejidad Práctica

La complejidad de un Programa es generalmente función del tamaño del problema a resolver y de las características de la instancia.

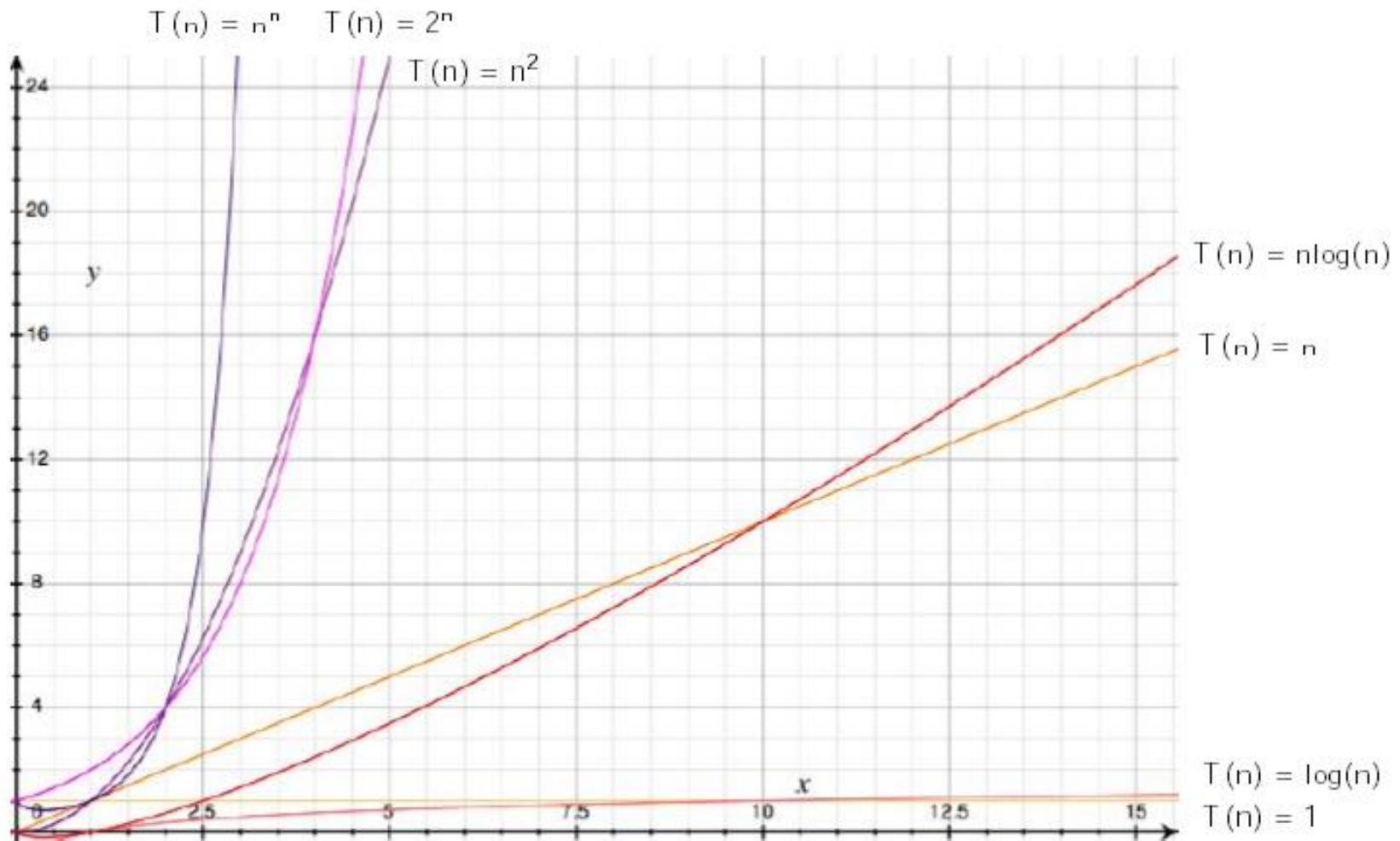
La función de complejidad se puede usar para comparar dos programas P y Q que hagan la misma tarea.

Se presentan tabla y gráfico con los valores aproximados para distintas funciones de crecimiento.

Crecimiento de distintas funciones de costo

n	$\log n$	$n \log n$	n^2	n^3	2^n
1	0	0	1	1	2
5	0.7	3	25	125	32
10	1.0	10	100	1000	1024
20	1.3	26	400	8000	1048576
50	1.7	85	2500	25000	1125900000×10^6
100	2.0	200	10000	1000000	$1267651000 \times 10^{21}$
200	2.3	460	40000	8000000	-
500	2.7	1349	250000	125000000	-
1000	3.0	3000	1000000	1000000000	-

Gráfica de los valores de las distintas funciones $T(n)$



Tiempo $T(n)$

(si se ejecutan en promedio 10^6 instrucciones/seg)

$n \setminus T(n)$	n	$n \text{ LOG}_2 n$	n^2	n^3	1.5^n	2^n	$n!$
10	<1s	<1s	<1s	<1s	<1s	<1s	10^{25} a
30	<1s	<1s	<1s	<1s	<1s	18 min	-
50	<1s	<1s	<1s	<1s	11 min	36 a	-
100	<1s	<1s	<1s	1 s	12892 a	$3 \cdot 10^7$ a	-
1.000	<1s	<1s	1s	17 min	-	-	-
10.000	<1s	<1s	2 min	12 d	-	-	-
100.000	<1s	2s	3 h	32 a	-	-	-
1.000.000	1s	20s	12 d	31710 a	-	-	-

s=segundo min=minuto h=hora d=día a=año

Tiempo T(n)

(si se ejecutan en promedio **10⁹** instrucciones/seg)

n \ T(n)	LOG ₂ n	n	n LOG ₂ n	n ²	n ³	n ⁴	n ¹⁰	2 ⁿ	n!
10	.0003 μs	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs	3.63 ma
20	.0004 μs	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms	77.1 a
30	.0005 μs	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s	8.4 10 ¹⁵ a
40	.0005 μs	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121.36 d	18.3 min	-
50	.0006 μs	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 a	13 d	-
100	.0007 μs	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 a	4 10 ¹³ a	-
1.000	.0010 μs	1. μs	9.96 μs	1 ms	1 s	16.67 min	3.17 10 ⁷ a	32 10 ²⁸³ a	-
10.000	.0013 μs	10. μs	130.03 μs	100 ms	16.67 min	115.7 d	3.17 10 ²³ a	-	-
100.000	.0017 μs	100 μs	1.66 ms	10 s	11.57 d	3171 a	3.17 10 ³³ a	-	-
1.000.000	.0020 μs	1.ms	19.92 ms	16.67 min	31.71 a	3.17 10 ⁷ a	3.17 10 ⁴³ a	-	-

μs= microsegundo = 10⁻⁶ segundo ms= milisegundo = 10⁻³ segundo

s=segundo

min=minuto

h=hora

d=día

a=año

Tiempo T(n)

(si se ejecutan en promedio **10⁹** instrucciones/seg)

n \ T(n)	LOG ₂ n	n	n LOG ₂ n	n ²	2 ⁿ	n!
100	.007 μs	.10 μs	.65 μs	10 μs	4 10 ¹³ a	-
1.000	.010 μs	1. μs	9.96 μs	1 ms	32 10 ²⁸³ a	-
10.000	.013 μs	10. μs	130 μs	100 ms	-	-
100.000	.017 μs	100 μs	1.67 ms	10 s	-	-
1.000.000	.020 μs	1.ms	19.93 ms	16.67 min	-	-
10.000.000	.023 μs	.01 s	.23 s	1.16 d	-	
100.000.000	.027 μs	.10 s	2.66 s	115.7 d	-	
1000.000.000	.030 μs	1 s	29.9 s	3.7 a	-	

μs= microsegundo = 10⁻⁶ segundo ms= milisegundo = 10⁻³ segundo

s=segundo

min=minuto

h=hora

d=día

a=año

Tamaño del problema más grande que se puede resolver en 1 hora

complejidad del algoritmo	con una computadora actual	con una computadora 100 veces más rápida	con una computadora 1000 veces más rápida.
n	N_1	$100 N_1$	$1000 N_1$
n^2	N_2	$10 N_2$	$312.6 N_2$
n^3	N_3	$4.64 N_3$	$10 N_3$
n^5	N_4	$2.5 N_4$	$3.98 N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

Ejemplo: Ordenar n datos

Algoritmo1: $T_1(n) \in O(n^2)$ Vs Algoritmo2: $T_2(n) \in O(n \log_2 n)$

Máquina A: ejecuta 10^7 instrucciones/seg

Máquina B: ejecuta 10^{10} instrucciones/seg

Considerar $n=10^7$ datos

Algoritmo 1: $T_1(n) = 2n^2$

$TA = (2 \cdot 10^{14}) / 10^7 = 200000000 \text{ seg} \approx 5555 \text{ horas} \approx 231 \text{ dias}$

$TB = (2 \cdot 10^{14}) / 10^{10} = 20000 \text{ seg} \approx 5.5 \text{ horas}$

Algoritmo 2: $T_2(n) = 50n \log_2 n$

$TA \approx 1163 \text{ seg} \approx 20 \text{ min}$

$TB \approx 1,163 \text{ seg}$

Ejemplo: Ordenar n datos

Algoritmo1: $T_1(n) \in O(n^2)$ Vs Algoritmo2: $T_2(n) \in O(n \log_2 n)$

Máquina A: ejecuta 10^7 instrucciones/seg

Máquina B: ejecuta 10^{10} instrucciones/seg

Considerar $n=10^8$ datos

Algoritmo 1: $T_1(n) = 2n^2$

$TA = (2 \cdot 10^{16}) / 10^7 = 2000000000$ seg ≈ 23148 días ≈ 63 años

$TB = (2 \cdot 10^{16}) / 10^{10} = 2000000$ seg ≈ 23 días

Algoritmo 2: $T_2(n) = 50n \log_2 n$

$TA \approx 3,69$ horas

$TB \approx 13$ seg

Algoritmos eficientes

Un **algoritmo es eficiente** si existe un polinomio $P(n)$ tal que el algoritmo puede resolver cualquier caso de tamaño de entrada n en un tiempo:

$$T(n) \in O(P(n))$$

Algoritmo eficiente: Algoritmo de complejidad polinomial

- Un algoritmo es eficiente si tiene un *tiempo polinómico* de ejecución.
- Si un algoritmo **no** es eficiente se lo denomina algoritmo de *tiempo exponencial*.

Algoritmos no eficientes

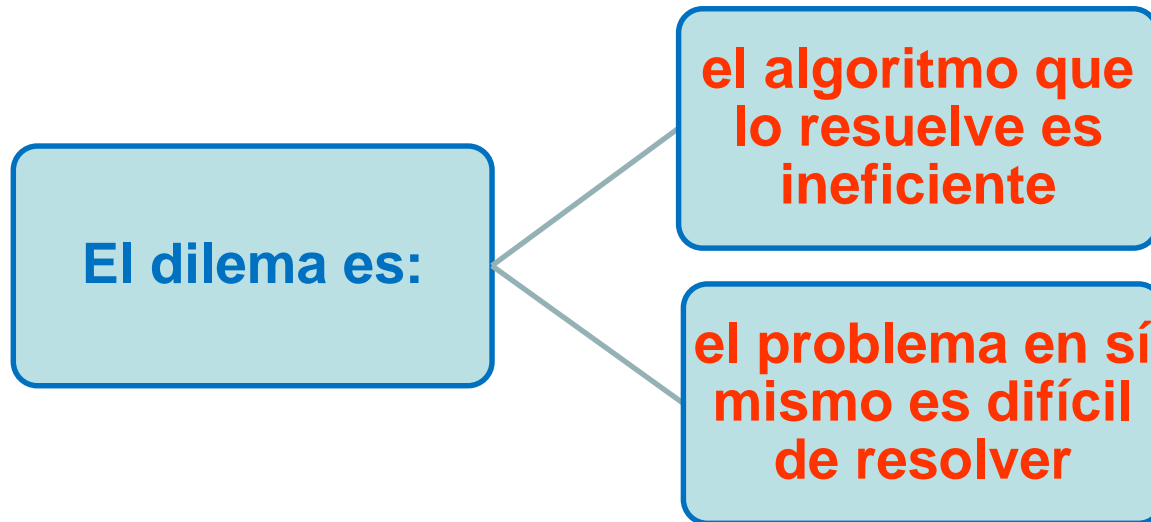
Los ejemplos de problemas más populares para los que *no se conocen algoritmos eficientes* son:

- viajante de comercio
- ciclo hamiltoniano
- satisfactibilidad de una fórmula booleana
- coloreado óptimo de grafos

Todos los algoritmos conocidos para estos problemas son *no eficientes*, de modo que son algoritmos inútiles cuando el tamaño de la entrada n crece mucho.

Problemas-Algoritmos

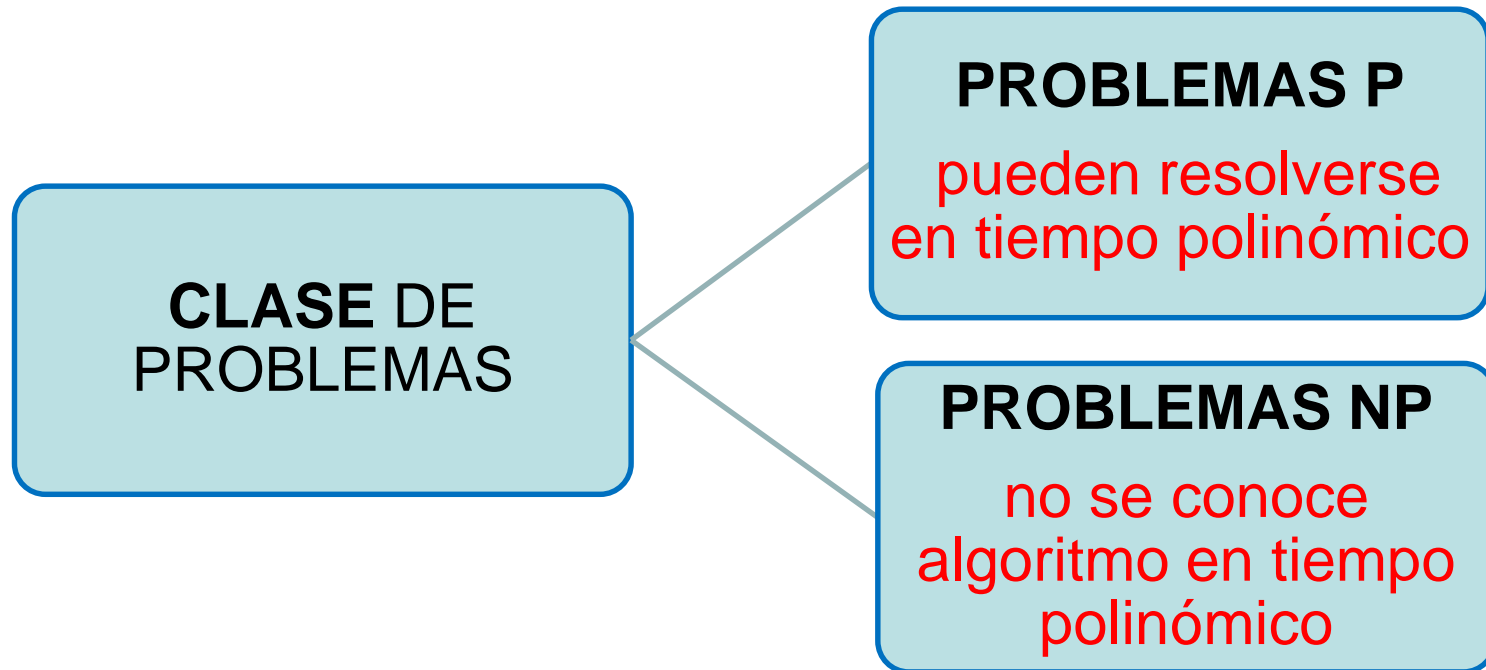
Entre los problemas que se resuelven con una computadora existen algunos para los cuales no se conocen ningún algoritmo eficiente que los resuelva, tampoco se ha demostrado que tienen una dificultad intrínseca.



- Podría ser que todavía no se ha encontrado un algoritmo eficiente para resolverlo...
- Podría ser que el problema sea intrínsecamente difícil, pero todavía no se lo ha podido demostrar...

Problemas

Los problemas se pueden dividir en 2 grandes grupos:



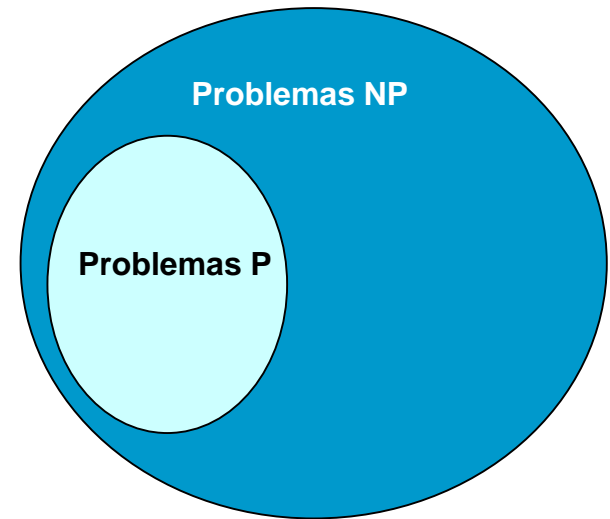
Problemas P y NP

- Es fácil de ver que:

$$P \subset NP$$

- Problema abierto:

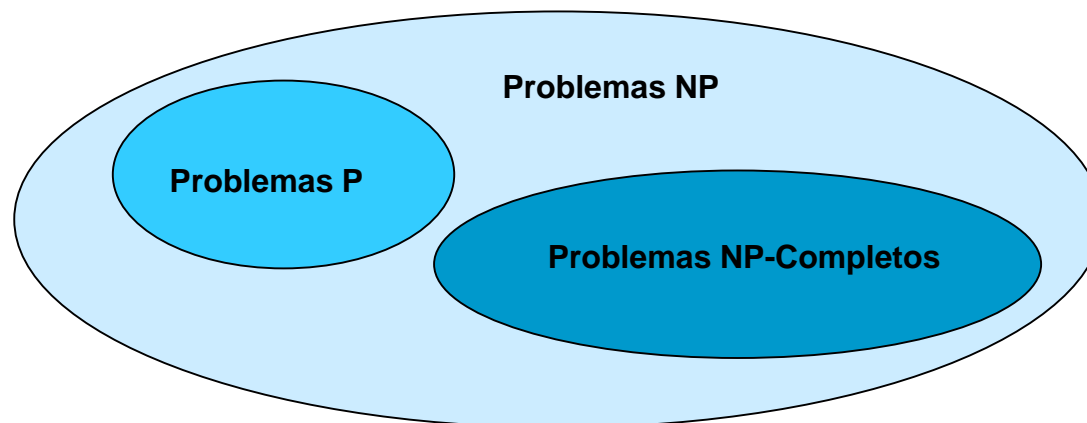
$$\text{es } P \neq NP?$$



*Este es **El Problema** abierto
más importante de teoría de la computación.*

Problemas NP completos

- Fueron caracterizados en 1971 por Stephen Cook. Todos los *problemas NP completos son equivalentes desde el punto de vista computacional*. Esto significa que se puede pasar de un problema a otro problema de NP con por una transformación polinómica..
- Ejemplo del problemas de este tipo son: SAT, agente viajero, ciclo hamiltoniano en un grafo, coloreado de un mapa con 3 colores.



Técnicas algorítmicas(1)

Técnicas de diseño de Algoritmos

- Fuerza bruta
- Recursión
- Dividir para Conquistar (Divide & Conquer)
- Programación dinámica
- Técnica ambiciosa (Greedy)
- Vuelta atrás (Backtracking)
- Ramificación y poda (Branch and Bound)
- Algoritmos Probabilistas.

Recursión

Un objeto es **recursivo** cuando se define en función de si mismo.

El concepto de recursividad se usa mucho en la matemática pero también en la computación.

Una **función** se dice **recursiva** si **se invoca a si misma**.

Ej. La definición recursiva de la función factorial:

$n! : \text{entero} \geq 0 \rightarrow \text{entero} \geq 1$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n - 1)! & \text{si } n > 0 \end{cases}$$

Recursión

El uso de la recursión permite escribir algoritmos **sencillos y concisos**.

FUNCION Fac(n) : entero $\geq 0 \rightarrow$ entero ≥ 1

SI $n=0$ ENTONCES

retorna (1)

SINO

retorna ($n * \text{Fac}(n-1)$)

FIN

Recursión

Principios generales del análisis recurrente

Dado un problema que trata información de un cierto tipo y de un tamaño, se quiere reducir el problema, expresando el tratamiento de la información de **igual tipo** pero de **menor tamaño**.

Es importante hacer un análisis del problema para encontrar un medio de *seccionar* esta información en *subinformaciones* de igual tipo.

Recursión

Problema **P** con información:

Tipo: **T**

Tamaño: **n**



Principio de seccionamiento

Problema **P'** con información:

Tipo: **$T' \equiv T$**

Tamaño: **$n' < n$**

Recursión

Este tipo de análisis requiere gran precisión y se puede guiar con las siguientes etapas:

- definición de las informaciones tratadas
- principio de seccionamiento
- expresión de los cálculos
- comprobación de las soluciones obtenidas

La búsqueda de las relaciones de recurrencia es la parte central del trabajo de análisis. Puede hacerse de manera deductiva o de manera directa inductiva.

Recursión-Ejemplo x^n

Calcular x^n , cuando n es un entero positivo o nulo y x es un numero real.

pot(x,n) : real x entero $\geq 0 \rightarrow$ real

1) Algoritmo1 usando *iteración*:

Se define como el producto de n copias de x .

$x * x * x \dots x$ (n veces)

2) Algoritmo2 usando *recursión*:

Se define x^n en función de x^{n-1}

$$x^n = x * x^{n-1} \quad \text{si } n \geq 1$$

$$x^0 = 1$$

Ejemplo x^n - Algoritmo 1

Algoritmo de potencia iterativo:

```
FUNCION pot1(x,n) : real x entero  $\geq 0 \rightarrow$  real  
  SI n=0 ENTONCES  
    p  $\leftarrow$  1  
  SINO  
    p  $\leftarrow$  x  
    PARA i=1,n-1 HACER  
      p  $\leftarrow$  p*x  
  retorna (p)  
FIN
```

- La complejidad de pot1 $\in O(n)$

Ejemplo x^n - Algoritmo 2

Algoritmo de potencia usando recursión:

```
FUNCION pot2(x,n) : real x entero  $\geq 0 \rightarrow$  real  
    SI n=0 ENTONCES  
        retorna (1)  
    SINO  
        retorna ( x * pot2(x,n-1) )  
FIN
```

- La complejidad de pot2 $\in O(n)$
- **Como se calcula la complejidad en las funciones recursivas?**

Recursión

Si se tienen funciones recursivas, se debe asociar una función de tiempo desconocida $T(n)$.

- Se plantea una **recurrencia para $T(n)$** , donde n mide el tamaño de la entrada al proceso.
- Se **generaliza** la ecuación para $T(n)$ en términos de $T(k)$ para distintos valores de k .
- Se **particulariza** para algún valor apropiado de k llegando al caso base.

Recursión-Ejemplo pot2

FUNCION pot2(x,n) : real x entero $\geq 0 \rightarrow$ real

SI n=0 ENTONCES

retorna (1)

SINO // n ≥ 1

retorna (x * pot2(x,n-1))

FIN

Sea $T(n)$ el tiempo para calcular pot2, su forma recurrente:

$$T(n) = \begin{cases} d & \text{si } n = 0 \\ c + T(n-1) & \text{si } n \geq 1 \end{cases} \quad c, d \text{ son constantes}$$

Recursión-Ejemplo pot2

Sea $T(n)$ el tiempo para calcular pot2 definido por:

$$\text{si } n = 0 \quad T(n) = d$$

$$\text{si } n \geq 1 \quad T(n) = c + T(n-1) \quad c, d \text{ son constantes}$$

desarrollando la recurrencia:

$$\text{si } n \geq 1 \quad T(n) = c + T(n-1)$$

$$\text{si } n-1 \geq 1, n \geq 2 \quad T(n-1) = c + T(n-2), \quad T(n) = 2*c + T(n-2)$$

$$\text{si } n-2 \geq 1, n \geq 3 \quad T(n-2) = c + T(n-3), \quad T(n) = 3*c + T(n-3)$$

...generalizando:

$$\forall n \geq k \quad T(n) = k*c + T(n-k)$$

en particular, vale para $n=k$ entonces:

$$\text{si } n=k \quad T(n) = n*c + T(n-n) = n*c + T(0) = n*c + d$$

Entonces $T(n) \in O(n)$

Recursión-Ejemplo Factorial

```
FUNCION Factorial (n) : entero  $\geq 0 \rightarrow$  entero  $\geq 1$   
    SI  $n=0$  OR  $n=1$  ENTONCES retorna (1)  
    SINO retorna ( n * Factorial (n-1) )  
FIN
```

Sea $T(n)$ el tiempo para calcular $\text{Factorial}(n)$:

$$T(n) = \begin{cases} d & \text{si } 0 \leq n \leq 1 \\ c + T(n-1) & \text{si } n > 1 \end{cases} \quad \text{c,d son constantes}$$

Recursión-Ejemplo Factorial

Sea $T(n)$ el tiempo para calcular Factorial(n)

$$\text{si } n \leq 1 \quad T(n) = d$$

$$\text{si } n > 1 \quad T(n) = c + T(n-1)$$

c, d son constantes

desarrollando la recurrencia:

$$\text{si } n > 1$$

$$T(n) = c + T(n-1)$$

$$\text{si } n-1 > 1, n > 2$$

$$T(n-1) = c + T(n-2),$$

$$T(n) = 2*c + T(n-2)$$

$$\text{si } n-2 > 1, n > 3$$

$$T(n-2) = c + T(n-3),$$

$$T(n) = 3*c + T(n-3)$$

...generalizando:

$$\forall n > k$$

$$T(n) = k*c + T(n-k)$$

en particular, vale para $n = k+1$ entonces:

$$\text{si } n = k+1$$

$$T(n) = (n-1)*c + T(1) = n*c - c + d$$

entonces $T(n) \in O(n)$

Recursión-Ejemplo Binario

Escribir una función recursiva que muestre el código binario de un entero dado.

Ej. $(25)_{10} \rightarrow (11001)_2$

Cociente $(25 / 2) = 12$

Cociente $(12 / 2) = 6$

Cociente $(6 / 2) = 3$

Cociente $(3 / 2) = 1$

Resto $(25 / 2) = 1$

Resto $(12 / 2) = 0$

Resto $(6 / 2) = 0$

Resto $(3 / 2) = 1$



Recursión-Ejemplo Binario

Función **Binario** (n) :Entero ≥ 1 en base 10

→ escribe secuencia de dígitos binarios

Si $n = 1$ ENTONCES

Escribir (n)

SINO // es $n > 1$

Binario(n/2)

Escribir (n mod 2)

FIN

Relación de recurrencia:

$$T(n) = \begin{cases} d & \text{si } n = 1 \\ c + T(n/2) & \text{si } n \geq 2 \end{cases} \quad c, d \text{ son constantes}$$

Recursión-Ejemplo Binario

Sea $T(n)$ el tiempo del algoritmo Binario

$$\text{si } n = 1 \quad T(n) = d$$

$$\text{si } n \geq 2 \quad T(n) = c + T(n/2)$$

c, d son constantes

Si $n \geq 2$

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

Si $n/2 \geq 2$, $n \geq 4$

$$T(n/2) = c + T(n/4)$$

$$T(n) = 2*c + T(n/4)$$

Si $n/4 \geq 2$, $n \geq 8$

$$T(n/4) = c + T(n/8)$$

$$T(n) = 3*c + T(n/8)$$

...generalizando:

$$\forall n \geq 2^k : T(n) = k*c + T(n/2^k)$$

En particular vale para: $n=2^k$, $k = \log_2 n$

$$T(n) = c * \log_2 n + T(1) = c * \log_2 n + d$$

Entonces para algoritmo Binario: $T(n) \in O(\log_2 n)$

RECORDATORIO

- Horarios de consulta:
 - Presencial Jueves de 10 - 12 hs - Lab B
 - Virtual Viernes de 14:30 a 15:30 hs-
- Práctico N° 2: jueves 3 de Abril
(Respetar distribución de comisiones)
- Evaluativo TP1: viernes 4 de Abril a las 15:30 hs - Virtual