

Unidad 2: Análisis de Algoritmos

Eficiencia de un Algoritmo

Máquina más rápida vs Algoritmo más eficiente

Suponga el problema de ordenar un arreglo de números, existen varios algoritmos, pero consideraremos InsertSort y MergeSort

Suponga que tiene dos máquinas, A y B, donde:

* A ejecuta 10^9 instrucciones por segundo

* B ejecuta 10^7 instrucciones por segundo (100 veces más lenta)

Cuando se analiza la eficiencia de un algoritmo, se expresa el tiempo de ejecución de la siguiente manera:

$$T(n) = c \cdot f(n)$$

Donde:

- * $T(n)$: Se llama tiempo de ejecución, pero en realidad es la cantidad de instrucciones en función del tamaño de la entrada
- * $f(n)$: función de complejidad del algoritmo
- * c : constante multiplicativa, representa cuantas operaciones básicas necesita el algoritmo en promedio por cada unidad de trabajo

Para nuestro ejemplo, consideramos:

* MergeSort : $T_M(n) = 50 \cdot (n \cdot \log_2 n)$

* InsertSort : $T_I(n) = 2 \cdot (n^2)$

Supongamos que queremos ordenar 1.000.000 de números ($n = 10^6$) ¿Cómo se comparan los algoritmos?

* InsertSort en máquina A (Algoritmo inefficiente en máquina rápida)

$$\frac{\text{Tiempo Total}}{\text{Total}} = \frac{T_I(10^6) \text{ instrucciones}}{10^9 \text{ instrucciones/segundo}} = \frac{2 \cdot (10^6)^2}{10^9} = 2000 \text{ segundos}$$

* MergeSort en máquina B (Algoritmo eficiente en máquina lenta)

$$\frac{\text{Tiempo Total}}{\text{Total}} = \frac{T_M(10^6) \text{ instrucciones}}{10^7 \text{ instrucciones/segundo}} = \frac{50 \cdot (10^6 \cdot \log_2 10^6)}{10^7} \approx 100 \text{ segundos}$$

Podemos observar que a pesar que A es 100 veces más rápida que B, el segundo caso se ejecuta mucho más rápido. Sin embargo, para valores de n pequeños (por ejemplo: $n = 1000$)

* InsertSort en máquina A (Algoritmo inefficiente en máquina rápida)

$$\frac{\text{Tiempo Total}}{\text{Total}} = \frac{T_I(10^3) \text{ instrucciones}}{10^9 \text{ instrucciones/segundo}} = \frac{2 \cdot (10^3)^2}{10^9} = 0,002 \text{ Segundos}$$

* MergeSort en máquina B (Algoritmo eficiente en máquina lenta)

$$\frac{\text{Tiempo Total}}{\text{Total}} = \frac{T_M(10^3) \text{ instrucciones}}{10^7 \text{ instrucciones/segundo}} = \frac{50 \cdot (10^3 \cdot \log_2 10^3)}{10^7} \approx 0,05 \text{ Segundos}$$

el orden de $n \log_2 n$ vs. n^2

En conclusión, para valores de n grandes, la eficiencia del algoritmo domina totalmente, incluso al comparar con un algoritmo más sencillo ($C_I < C_M$) en una máquina 100 veces más rápida. Sin embargo, mientras más pequeño sea el valor de n , más importa las constantes (C) y la rapidez de la máquina, pudiendo darse el caso que un algoritmo ineficiente sea más rápido que uno más eficiente.

Por lo general, para entradas grandes es mejor siempre usar el algoritmo más eficiente, aunque para entradas chicas, un algoritmo más sencillo (aunque ineficiente) puede tener resultados similares o incluso mejor.

Efecto del Algoritmo y la rapidez de la máquina

Tiempo de ejecución del algoritmo - $T(n)$ menor complejidad (cant. instrucciones)	Máximo tamaño solucionable en 1 segundo		
	Computador actual	100 veces más rápida	1000 veces más rápida
n	$N_0 = 100$ millones	$100N_0$	$1000N_0$
$100n$	$N_1 = 1$ millón	$100N_1$	$1000N_1$
n^2	$N_2 = 10000$	$10N_2$	$31.6N_2$
n^3	$N_3 = 464$	$4.64N_3$	$10N_3$
2^n	$N_4 = 26$	$N_4 + 6.64$	$N_4 + 9.97$

Complejidad del Algoritmo
↓
Menor complejidad
↑
Mayor complejidad

Caso Base Caso Máquina Rápida Caso Máquina Lenta

¿Cómo escoger el Mejor Algoritmo?

Una forma de elegir el mejor algoritmo podría ser simplemente medir el tiempo de respuesta de varios algoritmos, pero esto no es muy conveniente porque requiere: codificar el algoritmo, medir el tiempo, graficar los resultados y hallar la tendencia. Esto es mucho trabajo y además se presta a interferencia de factores externos (lenguaje de programación, calidad de código, rapidez de máquina, etc.)

Existe una forma de predecir el comportamiento de un algoritmo sin necesidad de implementarlo, esto se hace mediante el análisis de algoritmo y el cálculo asintótico.

Análisis de Algoritmos y Cálculo Asintótico

El análisis de Algoritmos, estudia la eficiencia teórica de un algoritmo. Es una técnica de estimación.

Útil para determinar si vale la pena implementar un algoritmo.

El cálculo asintótico es una técnica de caracterización de los algoritmos para poder comparlos.

Análisis de Algoritmos

Utilizando una descripción de alto nivel del algoritmo (por ejemplo pseudocódigo), caracterizamos el tiempo de ejecución o costo en función al tamaño de la entrada, como $T(n)$

Se toma en cuenta todas las posibles entradas, lo que permite evaluar la velocidad del algoritmo independientemente al entorno de hardware o software.

Modelo RAM (Random Access Memory)

- ④ Las instrucciones se ejecutan secuencialmente sin concurrencia
- ④ Banco de memoria potencialmente ilimitado
- ④ Celulas de memoria numeradas, el acceso toma un tiempo constante.
- ④ Cada operación elemental (OE) (+ ; - ; * ; = ; jump; etc) toma exactamente 1 unidad de tiempo

Observación : esto es una simplificación de la realidad

El tiempo depende de la entrada, y el tamaño de la entrada a su vez depende del problema.

El tiempo de ejecución ($T(n)$) para un valor de n específico, es la cantidad de OE ejecutadas

Análisis de Algoritmos por Casos

Peor Caso "como mucho va a costar x OE"

Función definida por el máximo número de pasos, para cualquier n
Valor máximo de $T(n)$

Caso Promedio "en promedio va a costar y OE"

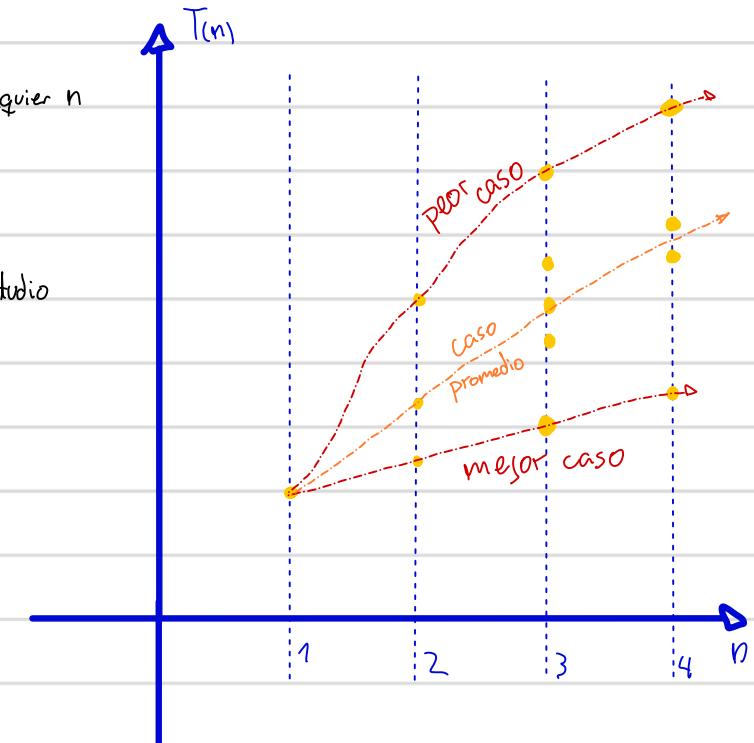
Es la función definida por el valor promedio de $T(n)$, implica un estudio estadístico de la distribución de los valores de $T(n)$

Es el mejor estudio, aunque su cálculo es más complejo

Mejor Caso "va a costar al menos z OE"

Es la función definida por el valor mínimo de $T(n)$.

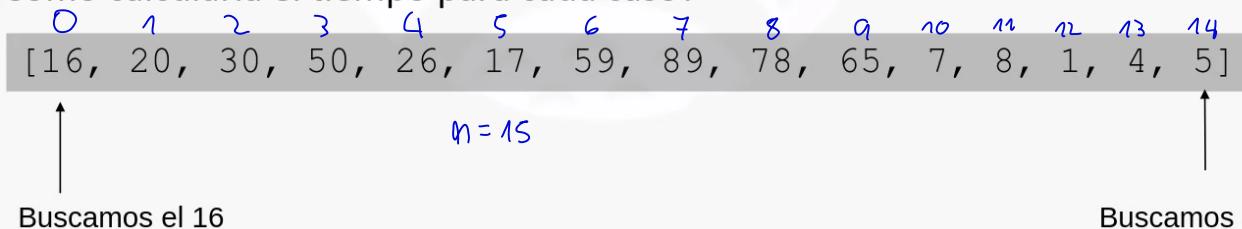
Es el caso más engañoso



Ejemplo 1 de análisis de casos

Considere el algoritmo de buscar un elemento en un arreglo desordenado:

- ¿Cuál es el peor caso? Cuando el elemento buscado está al final o no se encuentra
- ¿Cuál es el mejor caso? Cuando el elemento buscado es el primero
- ¿Cuál es el caso promedio? Cuando el elemento buscado está en el medio
- ¿Cómo calcularía el tiempo para cada caso?



En este ejemplo, el tiempo de ejecución depende de la cantidad de comparaciones

- ⊗ En el mejor caso, el elemento buscado es el primero, por lo que $T(n) = 1 \cdot C$ (tiempo constante)
- ⊗ En el peor caso, hay que recorrer todo el array, por lo que $T(n) = n \cdot C$ (tiempo lineal)
- ⊗ En el caso promedio, se recorre el array hasta la mitad, por lo que $T(n) = \frac{n}{2} \cdot C$ (tiempo lineal)

Estimación de Tiempo Consideraciones sobre las OE

- ⌚ Asignación
 - ⌚ Comparación
 - ⌚ Acceso a un arreglo
 - ⌚ Llamada a función
 - ⌚ Operación Simple
- } 1 OE cada una

Consideramos que todas las operaciones tienen el mismo costo, a pesar que esto no ocurre en la vida real.

Cálculo de tiempo de ejecución

Inspeccionando el código podemos determinar el número de operaciones elementales.

Analicemos el siguiente algoritmo, encontrar el máximo elemento de en un arreglo
(Recibimos el vector A y su tamaño n)

Algoritmo **findMax(A, n)**// Arreglo A[1..n]

Operaciones

<i>Maximo</i> $\leftarrow A[1]$	2	Asignación + Acceso Arreglo
for $i \leftarrow 2$ to n do	$1 + n$	Asignación + comparación for
if $A[i] > Maximo$ then	$2(n-1)$	(Comparación + Acceso) x iteraciones for
<i>Maximo</i> $\leftarrow A[i]$	$2(n-1)$	(peor caso)
{incremento del contado i}	$2(n-1)$	
return <i>Maximo</i>	<u>1</u>	return
Total	$7n - 2$	

27

En el peor caso, findMax se ejecuta $7n-2$ OE. Definimos entonces

$a =$ tiempo que toma la OE más rápida

$b =$ tiempo que toma la OE más lenta

Entonces :

$$a(7n-2) \leq T(n) \leq b(7n-2)$$

De esto concluimos que $T(n)$ esta limitada por dos funciones lineales.

Un cambio de entorno de HW o SW afecta a $T(n)$ en un factor constante, pero no afecta la tasa de crecimiento de $T(n)$

Por lo tanto, la tasa de crecimiento lineal de $T(n)$ es una propiedad intrínseca del algoritmo

Ejemplo de análisis de InsertSort

Algoritmo **InsertSort(A, n)** // Arreglo A[1..n]

Operaciones

for $j \leftarrow 2$ to $\text{length}(A)$ do	$1 + n$
$\text{key} \leftarrow A[j]$	$2(n - 1)$
$i \leftarrow j - 1$	$2(n - 1)$
while $i > 0$ and $A[i] > \text{key}$ do	$4 \cdot \sum_{j=2}^n t_j$
$A[i + 1] \leftarrow A[i]$	$6 \cdot \sum_{j=2}^n (t_j - 1)$
$i \leftarrow i - 1$	
$\text{key} \leftarrow A[j]$	$2(n - 1)$
{ <i>incremento del contador j</i> }	$2(n - 1)$
TOTAL	$4 \cdot \sum_{j=2}^n t_j + 6 \cdot \sum_{j=2}^n (t_j - 1) + 9n - 5$

30

④ Mejor Caso \rightarrow Lista ordenada, $t_j = 1$ para $j = 2, 3, 4, \dots, n$

$$T(n) = 4 \cdot (n-1) \cdot 1 + 6 \cdot (n-1) \cdot 0 + 9n - 5$$

$$T(n) = 4n - 4 + 9n - 5$$

$$T(n) = 13n - 9$$

⑤ Peor Caso \rightarrow Lista en orden inverso, $t_j = j$ para $j = 2, 3, \dots, n$

$$T(n) = 4 \left(\frac{n(n+1)}{2} - 1 \right) + 6 \left(\frac{n(n-1)}{2} \right) + 9n - 5$$

$$T(n) = 4 \left(\frac{n^2+n}{2} - 1 \right) + 3(n^2-n) + 9n - 5$$

$$T(n) = 2n^2 + 2n - 4 + 3n^2 - 3n + 9n - 5$$

$$T(n) = 5n^2 + 8n - 9$$

Cálculo Asintótico

Permite clasificar un algoritmo según su tasa de crecimiento.

Se ignoran las constantes y nos concentraremos en el grado dominante de la función.

Tasas de crecimiento usuales

Clase	Nombre de la función	
c	Constante	
$\lg n$	Logarítmica	
$\lg^2 n$	Logarítmica al cuadrado	
n	Lineal	
$n \lg n$	" $n \lg n$ " o <i>polilogarítmica</i>	
n^2	Cuadrática	
n^3	Cúbica	
$n^m, m=0,1,2,3,..$	Polinomial	
$c^n, c > 1$	Exponencial	Crecimiento
$n!$	Factorial	

Análisis Asintótico

Requisitos :

- ④ Encontrar $T(n)$ del peor caso
- ④ Expresar esta función en notación asintótica, por ejemplo \mathcal{O} (o grande)

Ejemplo :

En el peor caso, `findMax` tiene un $T(n) = 7n - 2$

Por lo que decimos que `findMax` está en $\mathcal{O}(n)$

Es decir, tiene tasa de crecimiento lineal.

Notación O (Cota Superior) (Peor Caso)

Definición Formal: Dada las funciones $f(n)$ y $g(n)$, se dice que " $f(n)$ es $O(g(n))$ " \Leftrightarrow existen las constantes positivas c y n_0 tales que $f(n) \leq c \cdot g(n)$ para $n \geq n_0$.
 $f(n) \in O(g(n)) \Leftrightarrow \exists c, n_0 > 0 / f(n) \leq c \cdot g(n) \text{ para } n \geq n_0$.

¿Qué significa esto?

$f(n) \rightarrow$ cantidad de OE en función al tamaño de la entrada, es decir $T(n)$

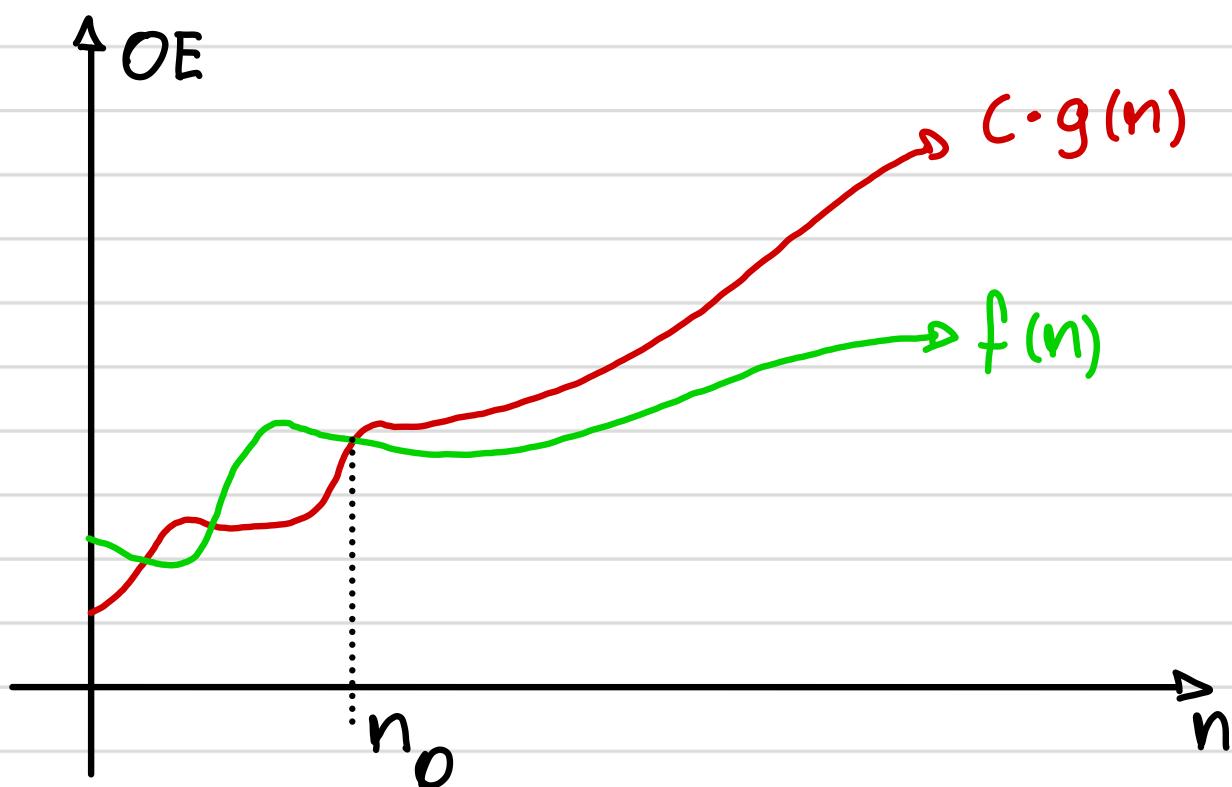
$g(n) \rightarrow$ función de referencia para comparar el crecimiento ($n, n^2, \log n$, etc)

$c \rightarrow$ constante multiplicativa para ajustar la escala. Da igual que $f(n)$ sea 2 o 10 veces más grande.
lo importante es que crezca de la misma forma

$n_0 \rightarrow$ el punto a partir del cual la comparación siempre se cumple.
no nos importan los valores de n pequeños

A partir de cierto punto (n_0), la función $f(n)$ no crece más rápido que $c \cdot g(n)$

" $f(n)$ crece como mucho tan rápido como $g(n)$ "



Reglas de la Notación O

⊗ Si $f(n)$ es un polinomio de grado $d \rightarrow f(n)$ está en $O(n^d)$
Se ignoran los términos de menor grado y las constantes

⊗ Siempre se utiliza la clase más ajustada
Se dice que $2n$ está en $O(n)$ y no $O(n^2)$

⊗ Siempre se utiliza la expresión simple de la clase
Se dice que $3n$ está en $O(n)$ y no $O(3n)$

Notación Ω (Omega) (Cota Inferior)

Definición Formal:

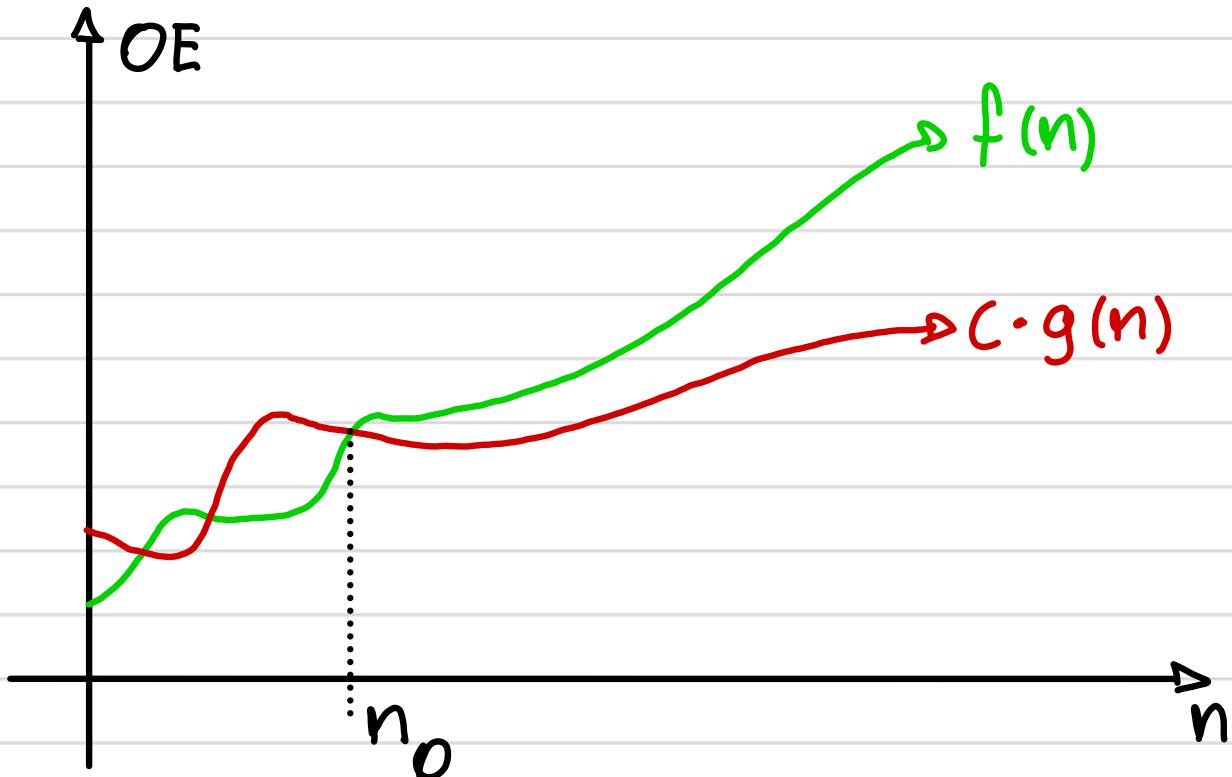
$f(n)$ está en $\Omega(g(n))$ si existe una constante c y un entero $n_0 \geq 1$ tal que $f(n) \geq c \cdot g(n)$ para $n \geq n_0$

¿Qué significa esto?

Al igual que en notación Θ , $f(n)$ representa $T(n)$ y $g(n)$ es una función de referencia. Decir que $f(n) \in \Omega(g(n))$ significa que $f(n)$ crece al menos tan rápido como $g(n)$. Es una cota asintótica inferior.

A partir de un punto n_0 , siempre se cumple que $f(n) \geq c \cdot g(n)$

" $f(n)$ crece como mínimo tan rápido como $g(n)$ "



Notación Θ (theta) (Cota Ajustada)

Definición Formal: $f(n)$ está en $\Theta(g(n))$ si existen dos constantes c_1 y c_2 y un número entero $n_0 \geq 1$ tales que:

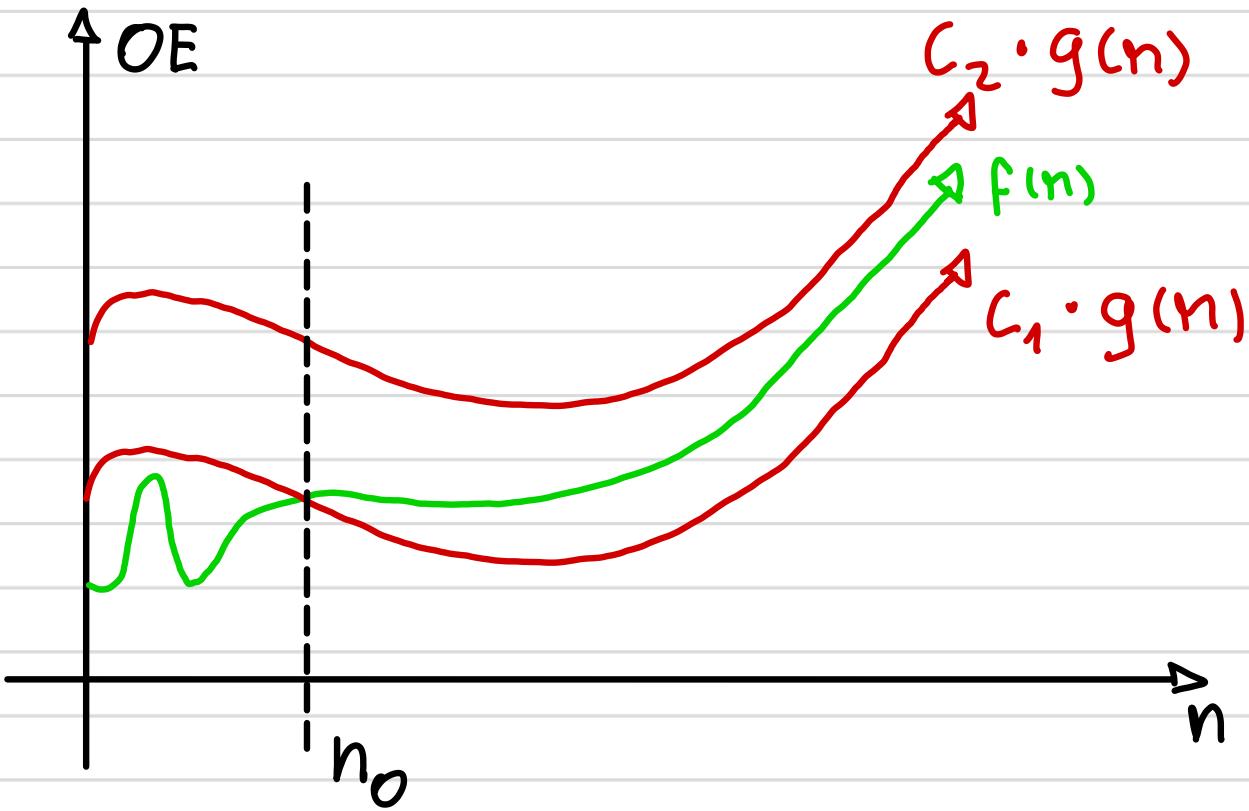
$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) ; \text{ para } n \geq n_0$$

¿Qué significa esto?

Igual que antes, $f(n)$ representa a $T(n)$ y $g(n)$ es una función de referencia.

Lo que la definición quiere decir es que a partir de cierto punto (n_0), $f(n)$ no crece más lento que $c_1 \cdot g(n)$ ni más rápido que $c_2 \cdot g(n)$.

En otras palabras, a partir de n_0 , $f(n)$ crece al mismo ritmo que $g(n)$, excepto por las constantes multiplicativas.



Notaciones Asintóticas - Resumen

⊗ Notación \mathcal{O} grande (Cota Superior)

$f(n)$ está en $\mathcal{O}(g(n))$ si $f(n)$ es asintóticamente igual o menor a $g(n)$

$$f(n) \leq C \cdot g(n) ; \text{ para } n \geq n_0$$

En el mejor caso, $f(n)$ crece tan rápido como $g(n)$

⊗ Notación Ω (omega) (Cota Inferior)

$f(n)$ está en $\Omega(g(n))$ si $f(n)$ es asintóticamente igual o mayor a $g(n)$

$$C \cdot g(n) \leq f(n) ; \text{ para } n \geq n_0$$

En el peor caso, $f(n)$ crece tan rápido como $g(n)$

⊗ Notación Θ (theta) (Cota Ajustada)

$f(n)$ está en $\Theta(g(n))$ si $f(n)$ es asintóticamente igual a $g(n)$

$$C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$$

$f(n)$ crece exactamente al mismo orden que $g(n)$

Otras Notaciones Asintóticas

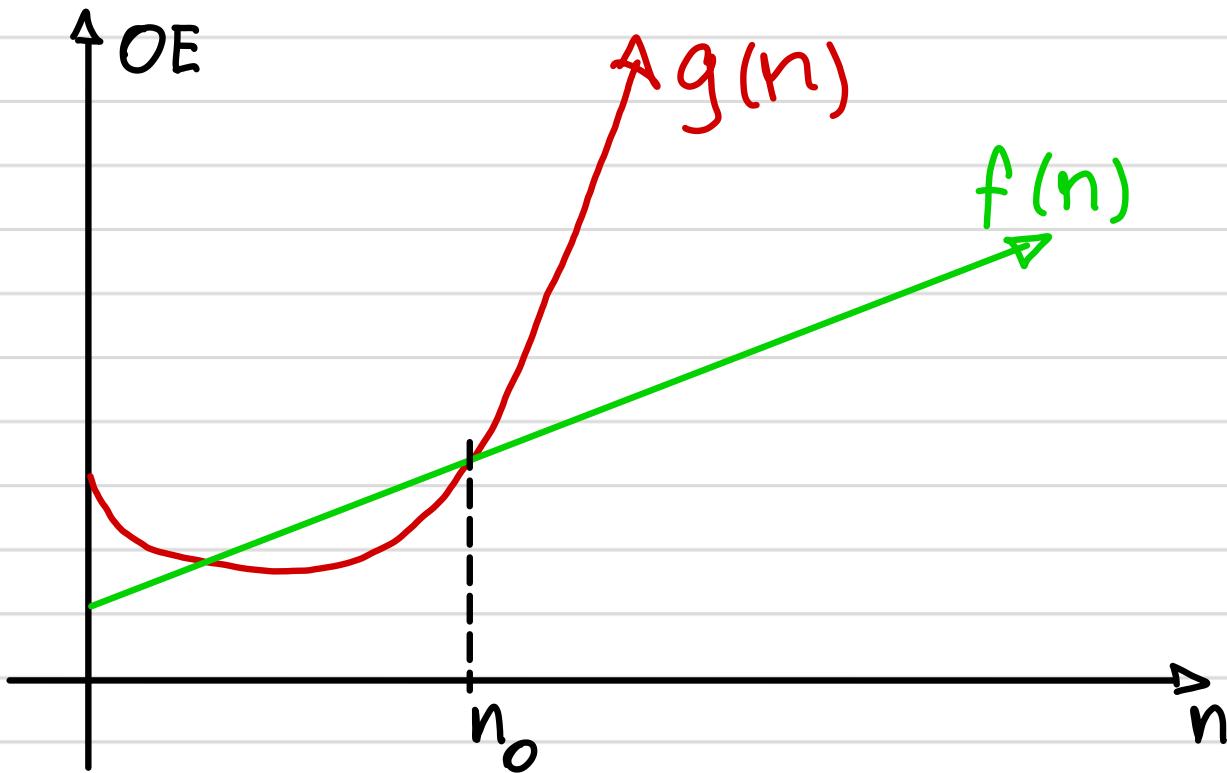
Notación \mathcal{O} (\mathcal{o} chica) (estrictamente más lento)

Definición Formal: Se dice que $f(n)$ está en $\mathcal{o}(g(n))$ si existen constantes positivas C y n_0 tales que: $f(n) < C \cdot g(n)$ para $n \geq n_0$

¿Qué significa esto?

Si $f(n)$ está en $\mathcal{o}(g(n))$, significa que $f(n)$ crece mucho más lento que $g(n)$, al punto que $\lim_{n \rightarrow +\infty} \left(\frac{f(n)}{g(n)} \right) = 0$

$f(n)$ es insignificante comparado con $g(n)$ para valores de n grandes



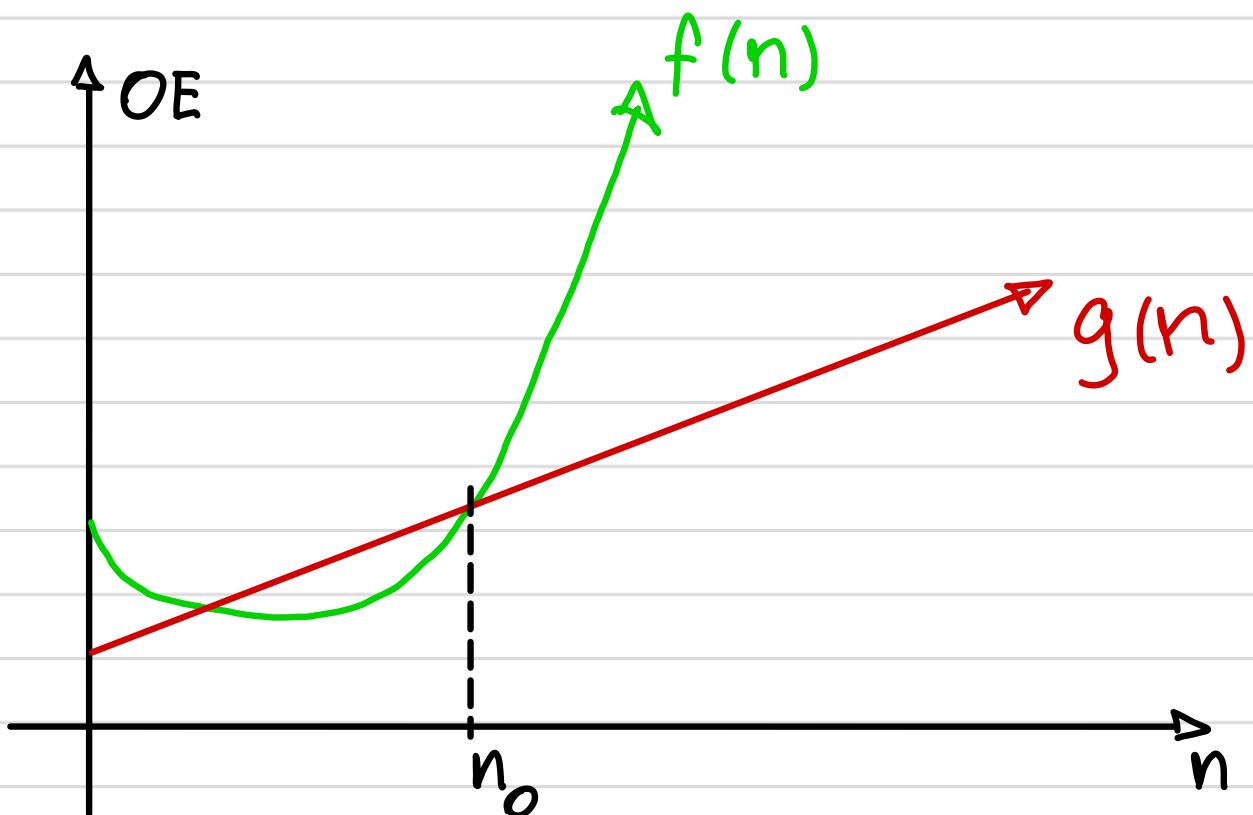
Notación ω (Omega chica) (estrictamente más rápido)

Definición Formal: $f(n)$ está en $\omega(g(n))$ si existen las constantes positivas c y n_0 tales que:

$$f(n) > c \cdot g(n) \text{ para } n \geq n_0$$

¿Qué significa esto?

Significa que $f(n)$ crece mucho más rápido que $g(n)$, de tal forma que $\lim_{n \rightarrow +\infty} \left(\frac{f(n)}{g(n)} \right) = +\infty$
Es decir, para valores de n grandes $f(n)$ es mucho mayor a $g(n)$



Notaciones Asintóticas - Resúmes fr fr

Notación	Que Significa	Ejemplo	Explicación
$f(n) \in o(g(n))$	$f(n)$ crece estrictamente más lento que $g(n)$	$f(n) = n ; g(n) = n^2$ $n \in o(n^2)$	$\frac{n}{n^2} = \frac{1}{n} = 0$ cuando $n \rightarrow \infty$ $f(n)$ es insignificante comparado a $g(n)$
$f(n) \in O(g(n))$	$f(n)$ crece como mucho tan rápido como $g(n)$	$f(n) = 3n^2 + 2n - 1$ $g(n) = n^2$ $f(n) \in O(n^2)$	$f(n) \leq C \cdot g(n)$ (para valores grandes de n)
$f(n) \in \Theta(g(n))$	$f(n)$ crece exactamente al mismo ritmo que $g(n)$	$f(n) = 5n^2 + 3n$ $g(n) = n^2$ $f(n) \in \Theta(n^2)$	$C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$ (para valores de n grandes)
$f(n) \in \Omega(g(n))$	$f(n)$ crece como mínimo tan rápido como $g(n)$	$f(n) = 3n^2$ $g(n) = n$ $f(n) \in \Omega(n)$	$f(n) \geq C \cdot g(n)$ para n grandes
$f(n) \in \omega(g(n))$	$f(n)$ crece estrictamente más rápido que $g(n)$	$f(n) = 7n^2$ $g(n) = n$ $f(n) \in \omega(n)$	$\frac{n^2}{n} = n = \infty$ para $n \rightarrow \infty$ $f(n)$ aplasta a $g(n)$ para valores de n grandes

Ayuda - Memoria

O chica \longrightarrow $f(n)$ crece más lento que $g(n)$

O grande \longrightarrow $f(n)$ crece igual o más lento que $g(n)$

Θ theta \longrightarrow $f(n)$ crece al mismo ritmo que $g(n)$

Ω omega grande \longrightarrow $f(n)$ crece igual o más rápido que $g(n)$

ω omega chica \longrightarrow $f(n)$ crece más rápido que $g(n)$

Límites Aplicados al Cálculo Asintótico

Un método simple para comparar las tasas de crecimiento de dos funciones es calcular el límite cuando $n \rightarrow +\infty$ de la razón entre $f(n)$ y $g(n)$. Hay 3 casos :

$$\lim_{n \rightarrow +\infty} \left(\frac{f(n)}{g(n)} \right) = \begin{cases} 0 & \rightarrow T(n) < g(n) \\ K > 0 & \rightarrow T(n) = g(n) \\ +\infty & \rightarrow T(n) > g(n) \end{cases}$$

⊗ Los primeros dos casos implican que : $T(n) \in O(g(n))$

⊗ Los dos últimos casos implican que : $T(n) \in \Omega(g(n))$

⊗ El segundo caso implica que : $T(n) \in \Theta(g(n))$

Aplicación de Límites

Recordatorio que existen la Regla de L'Hôpital y la Fórmula de Stirling para factoriales

L'Hôpital

$$\lim_{n \rightarrow +\infty} \left(\frac{f(n)}{g(n)} \right) = \lim_{n \rightarrow +\infty} \left(\frac{f'(n)}{g'(n)} \right)$$

Fórmula Stirling

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e} \right)^n$$

Definiciones Alternativas de Cotas

⊕ $f(n) \in O(g(n))$ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = C$; $0 \leq C < +\infty$

⊕ $f(n) \in \Theta(g(n))$ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = C$; $0 < C < +\infty$

⊕ $f(n) \in \Omega(g(n))$ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = C$; $0 < C \leq +\infty$

⊕ $f(n) \in o(g(n))$ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0$

⊕ $f(n) \in w(g(n))$ si $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = +\infty$

Propiedades de las Cotas

① Reflexividad (Aplica a O, Θ, Ω)

Toda función está acotada por sí misma

$$Ej: f(n) \in O(f(n))$$

② Simetría (Para Θ)

Si dos funciones crecen en el mismo orden, la relación es simétrica

$$Ej: f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$$

③ Simetría Transpuesta (Para O y Ω)

Si f está acotada superiormente por $g \rightarrow g$ está acotada inferiormente por f

$$Ej: f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$

④ Transitividad (Aplica a O, Θ, Ω)

Si f está acotada por g , y g está acotada por h , entonces f también está acotada por h .

$$Ej: [f(n) \in O(g(n)) \wedge g(n) \in O(h(n))] \rightarrow f(n) \in O(h(n))$$

⑤ Suma de Funciones (Aplica a O, Θ, Ω)

La suma de funciones está dominada por la función que más crece

$$Ej: f_1(n) \in O(g_1(n)), f_2(n) \in O(g_2(n)) \rightarrow f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$$

⑥ Definición de Θ

Una función pertenece a $\Theta(g(n))$, si está acotada superior e inferiormente por $g(n)$

$$Ej: f(n) \in \Theta(g(n)) \Leftrightarrow [f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))]$$

O, Θ, Ω como funciones anónimas

Podemos usar notaciones asintóticas para representar funciones asintóticas.

$$Ej: f(n) = 10n^2 + O(n)$$

Significa que $f(n)$ es igual a $10n^2$ más una función que no conocemos o no nos interesa y que es asintóticamente al menos lineal en n .

Recordar Que :

Notación Asintótica	Se comporta similar a
O chica	$>$
O grande	\geq
Θ	$=$
Ω grande	\leq
ω chica	$<$

Series Útiles para el análisis

① Suma de Constantes

Sumar n veces el valor 1 da como resultado n. Crecimiento Lineal

$$\sum_{i=1}^n 1 = n \in \Theta(n)$$

② Suma de enteros consecutivos

La suma de los primeros n enteros crece cuadráticamente

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

③ Suma de Cuadrados

La suma de los cuadrados de los primeros n enteros tiene crecimiento cúbico

$$\sum_{i=1}^n i^2 = \frac{n(n-1)(2n+1)}{6} \in \Theta(n^3)$$

④ Suma de Potencias (Progresión Geométrica)

Fórmula de suma de una progresión geométrica. Crecer como a^n

$$\sum_{i=1}^n a^i = \frac{a^{n-1} - 1}{a - 1} \in \Theta(a^n)$$

⑤ Caso Particular - Suma de Potencias de 2

Suma de Potencias de 2. Crecimiento exponencial

$$\sum_{i=1}^n 2^i = 2^{n-1} - 1 \in \Theta(2^n)$$

⑥ Serie Armónica

Crecer lentamente. Aproximadamente como el $\ln(n)$. No es lineal ni polinómica

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

Propiedades de las Sumatorias

① Suma de sumas

Se puede separar la suma (o resta) en dos sumatorias

$$\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i$$

② Factor Constante

Una constante se puede sacar afuera de la sumatoria

$$\sum K a_i = K \sum a_i$$

③ Partición del Rango

Se puede dividir una sumatoria en intervalos más pequeños

$$\sum_{l \leq i \leq u} a_i = \sum_{l \leq i \leq m} a_i + \sum_{m+1 \leq i \leq u} a_i$$

Estimar una suma discreta

Se puede estimar el resultado de una sumatoria reemplazándola por una integral definida.

Ejemplos : ① $\sum_{i=1}^n i \sim \int_1^n x dx = \frac{x^2}{2} \Big|_1^n = \frac{n^2}{2} - \frac{1}{2} =$

② $\sum_{i=1}^n \frac{1}{i} \sim \int_1^n \frac{dx}{x} = \ln(x) \Big|_1^n$

Algoritmos de Estructura Recurrente

Condiciones a tener en cuenta para una solución recurrente o recursiva:

1. Definir un caso base.
2. Avanzar en cada llamada recursiva hacia el caso base
3. Asumir que la recursión funciona
4. No realizar cálculos repetidos

Divide and Conquer

Es una estrategia para resolver problemas grandes dividiéndolos en problemas más pequeños pero del mismo tipo. Subdividir tantas veces como sea necesario. Resolver los subproblemas y combinar las soluciones para resolver el problema.

Estructura Básica

```
DivideAndConquer (Problema){  
    if Problema es chico {  
        resolver (Problema)  
    } else {  
        Dividir Problema en P1, P2, P3, ..., Pn  
        Aplicar DivideAndConquer (P1, P2, ..., Pn)  
        Combinar (Divide And Conquer (P1, P2, ..., Pn))  
    }  
}
```

Ejemplos de DAC

- ④ Binary Search
- ④ Max y Min
- ④ Merge Sort

Relaciones Recurrentes $T(n) = T(n-1) + 1 ; n > 0$

Para algoritmos de la forma $T(n) = \begin{cases} 1 & \text{para } n=0 \\ T(n-1)+1 & \text{para } n>0 \end{cases}$

Se resuelve hallando $T(n-1)$ y reemplazando.

Por ejemplo:

$$T(n) = \begin{cases} 1 & \text{para } n=0 \\ T(n-1) + 1 & \text{para } n>0 \end{cases}$$

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

$$T(n) = [T(n-2) + 1] + 1$$

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

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

: Seguir reemplazando
: hasta K veces

$$T(n) = T(n-K) + K$$

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

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

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

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

Asumimos que $n-K = 0 \therefore n = K$

$$\therefore T(n) = T(0) + n$$

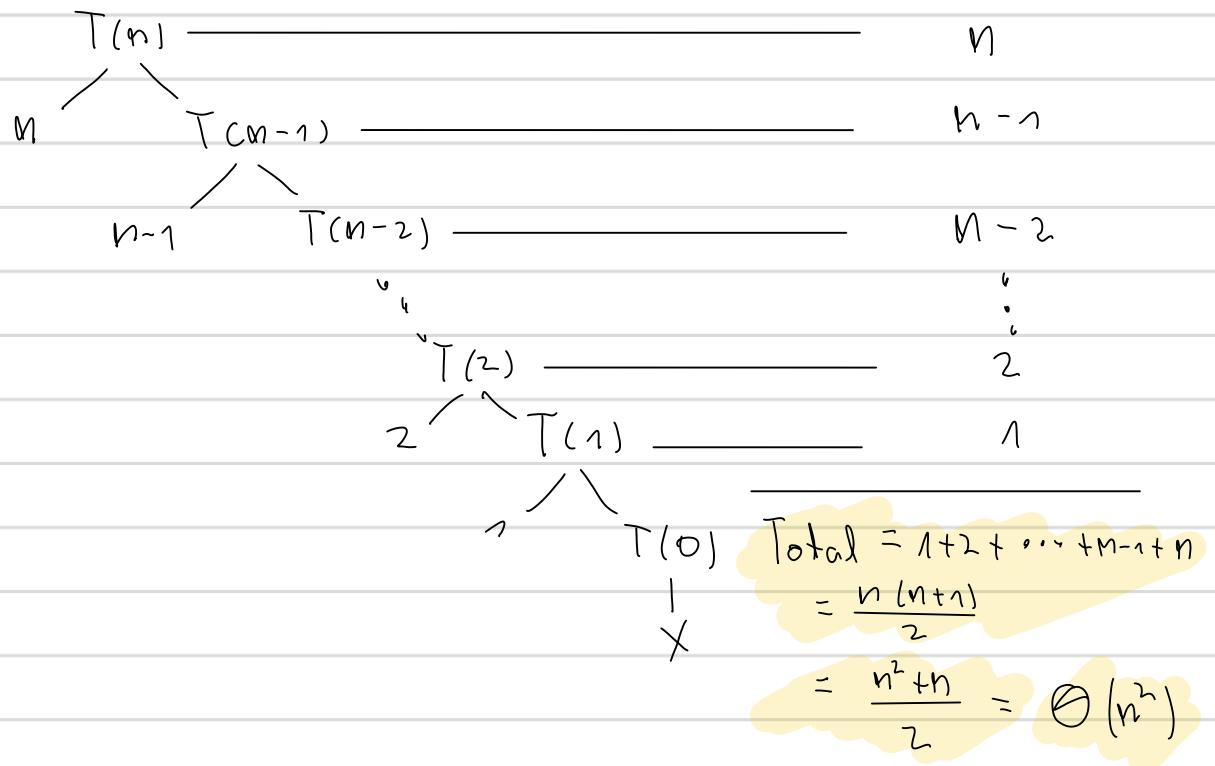
$$T(n) = 1 + n = \Theta(n)$$

Relaciones Recurrentes - Función Decreciente

Funciones decrecientes de la forma $T(n) = T(n-1) + \Theta(n)$; $n > 0$
Se puede resolver de muchas formas

① Árbol Recursivo

$$T(n) = \begin{cases} 1 & \text{para } n=0 \\ T(n-1) + n & \text{para } n > 0 \end{cases}$$



② Por Sustitución

$$T(n) = \begin{cases} 1 & \text{para } n = 0 \\ T(n-1) + n & \text{para } n > 0 \end{cases}$$

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

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

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

$$T(n) = T(n-1) + n - ①$$

$$T(n) = \lceil f(n-2) + (n-1) \rceil + n$$

$$T(n) = T(n-2) + (n-1) + n \quad \text{--- (2)}$$

$$T(n) = T(n-3) + (n-2) + (n-1) + n - 3$$

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$$

$$T(n) = T(0) + 1 + 2 + \dots + n$$

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

Asumimos que $T(m-K) = T(0)$

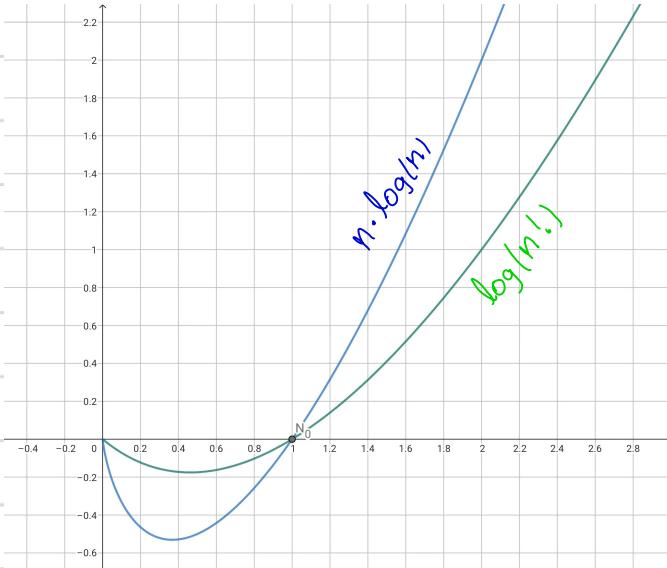
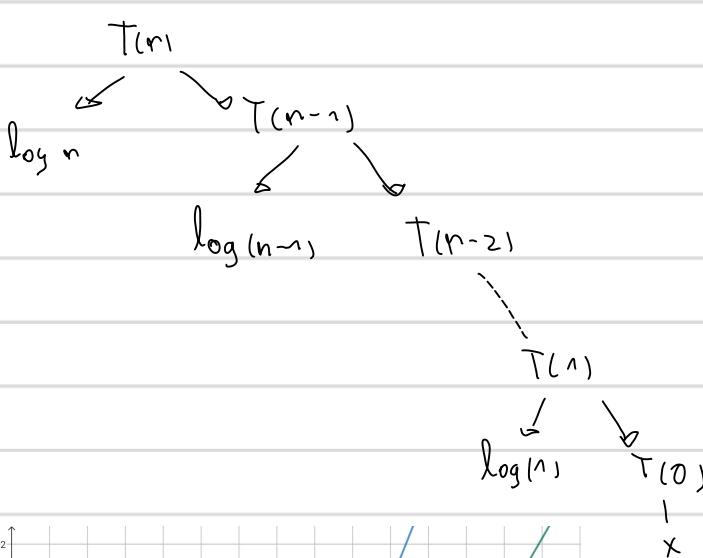
$$S - K = 0$$

$$r=k$$

Ejemplos de Algoritmos

① $T(n) = \begin{cases} 1 & \text{para } n=0 \\ T(n-1) + \log(n) & \text{para } n>0 \end{cases}$

Por Árbol de recursividad



$$T(n) = \log n + \log(n-1) + \dots + \log(1)$$

$$T(n) = \log[n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1]$$

$$T(n) = \log(n!)$$

No existe una tasa de crecimiento usual que sea exactamente igual a $\log(n!)$, pero graficando ó aplicando límites nos damos cuenta que $n \cdot \log(n) \geq \log(n!)$ para $n \geq 1$.

$$\therefore T(n) \in O(n \log n)$$

$$T(n) = \begin{cases} 1 & \text{para } n=0 \\ T(n-1) + \log(n) & \text{para } n>0 \end{cases}$$

Por Sustitución

$$T(n) = T(n-1) + \log(n) \quad \text{--- ①}$$

$$T(n) = T(n-2) + \log(n-1) + \log(n) \quad \text{--- ②}$$

$$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log(n)$$

⋮
: K veces

$$T(n) = T(n-k) + \log(1) + \log(2) + \dots + \log(n-2) + \log(n-1) + \log(n)$$

$$n - k = 0$$

$$n = k$$

$$\therefore T(n) = T(0) + \log(1) + \log(2) + \dots + \log(n-1) + \log(n)$$

$$T(n) = 1 + \log[1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n]$$

$$T(n) = 1 + \log(n!)$$

Según demostración de arriba, $\log(n!) \leq n \cdot \log(n) \quad \forall n \geq 1$

$$\therefore T(n) \in O(n \cdot \log(n))$$

Regla General - Algoritmos Recursivos Decrecientes

Sea $T(n) = T(n-K) + f(x)$; donde K es una constante cualquiera y $f(x)$ es una función polinómica cualquiera

Se cumple que la notación asintótica de $T(n)$ será:

$$\mathcal{O}(n \cdot f(x))$$

Ejemplos:

$$\textcircled{*} \quad T(n) = T(n-1) + 1 \rightarrow \mathcal{O}(n)$$

$$\textcircled{*} \quad T(n) = T(n-1) + n \rightarrow \mathcal{O}(n^2)$$

$$\textcircled{*} \quad T(n) = T(n-1) + \log(n) \rightarrow \mathcal{O}(n \cdot \log(n))$$

$$\textcircled{*} \quad T(n) = T(n-2) + n^2 \rightarrow \mathcal{O}\left(\frac{n}{2} \cdot n^2\right) = \mathcal{O}(n^3)$$

$$\textcircled{*} \quad T(n) = T(n-100) + n^5 \rightarrow \mathcal{O}\left(\frac{n}{100} \cdot n^5\right) = \mathcal{O}(n^6)$$

Obs: En el caso que $T(n-K)$ tenga un coeficiente $\neq 1$, esto NO se cumple

Algoritmo - T(n) recu con coef $\neq 1$

$$T(n) = \begin{cases} 1 & \text{para } n=0 \\ 2T(n-1) + 1 & \text{para } n>0 \end{cases}$$

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

$$T(n) = 2[2T(n-2) + 1] + 1$$

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

$$T(n) = 2^2 [2T(n-3) + 1] + 2 + 1$$

$$T(n) = 2^3 T(n-3) + 2^2 + 2 + 1$$

⋮
⋮
⋮ K veces

$$T(n) = 2^K T(n-K) + 2^{K-1} + 2^{K-2} + \dots + 2 + 1$$

$$n-K=0$$

$$n=K$$

$$\therefore T(n) = 2^n T(0) + 2^{n-1} + 2^{n-2} + \dots + 2 + 1$$

$$T(n) = 2^n + 2^{n-1} + 2^{n-2} + \dots + 2 + 1$$

eso es la suma de los términos de una progresión geométrica
cuya fórmula es: $S = \frac{a(r^{k+1}-1)}{r-1}$, donde $a=1$; $r=2$

$$\therefore T(n) = 2^{n-1} - 1 \rightarrow O(2^n)$$

Teorema Maestro para Funciones Decrecientes

Forma general de una función decreciente:

$$T(n) = \underbrace{a \cdot T(n-b)}_{\text{Coeficiente}} + \underbrace{f(n)}_{\text{constante}}$$

Donde :

$$\begin{cases} a > 0 \\ b > 0 \\ f(n) \in O(n^k) \end{cases}$$

No interesa el valor exacto,
solo su valor en notación asintótica,
es decir, su valor en Θ, O, Ω , etc.

Posibles Casos :

① $a < 1 \rightarrow O(f(n))$

② $a = 1 \rightarrow O(n \cdot f(n))$

③ $a > 1 \rightarrow O(a^{n/b} \cdot f(n))$

Dividing Functions

Algoritmos recurrentes de la forma:

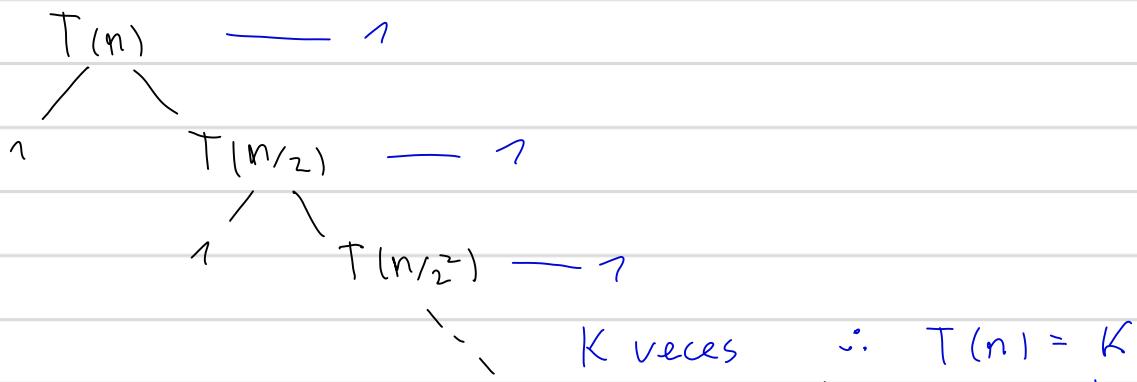
```

Algoritmo Test(n) {
    if (n < 1) {
        AlgoritmoTest(n/2)
    }
}
  
```

Ejemplo

$$T(n) = \begin{cases} 1 & \text{para } n = 1 \\ T(n/2) + 1 & \text{para } n > 1 \end{cases}$$

Por árbol de recursión



Por sustitución

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

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

$$T(n) = T(n/2^3) + 3$$

\vdots K veces

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

$$T(n) = T(1) + \log(n)$$

$$T(n) = 1 + \log(n)$$

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

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

$$\frac{n}{2^K} = 1$$

$$n = 2^K$$

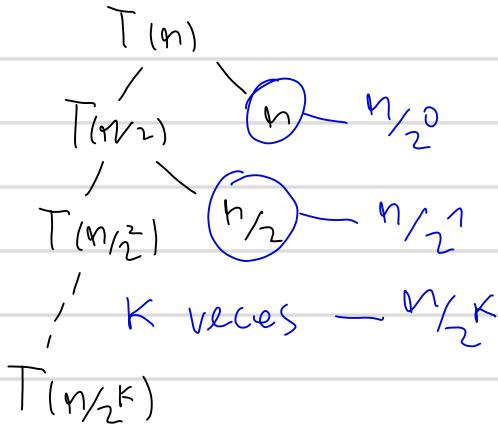
$$\log(n) = K$$

$$\left| \begin{array}{l} T(n) = \log(n) \\ T(n) \in O(\log n) \end{array} \right.$$

Ejemplo 2

$$T(n) = \begin{cases} 1 & \text{para } n=1 \\ T(n/2) + n & \text{para } n > 1 \end{cases}$$

Por Árbol Recursivo



$$T(n) = \frac{n}{2^0} + \frac{n}{2^1} + \dots + \frac{n}{2^K}$$

$$T(n) = n \left(\frac{1}{2^0} + \frac{1}{2^1} + \dots + \frac{1}{2^K} \right)$$

$$T(n) = n \cdot \left(\sum_{i=0}^K \frac{1}{2^i} \right) \rightarrow 1$$

$$\therefore T(n) = n \cdot 1$$

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

Por Sustitución

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

$$T(n) = T(1) + n \underbrace{\left(1 + \frac{1}{2} + \dots + \frac{1}{2^{K-1}} \right)}_{\sum_{i=1}^{K-1} \frac{1}{2^i} \approx 1}$$

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

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

$\because K$ veces

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

$$T(n) = T(n/2^K) + n/2^{K-1} + \dots + n/2 + n$$

$$n/2^K = 1$$

$$n = 2^K$$

$$\log(n) = K$$

Ejemplo 3

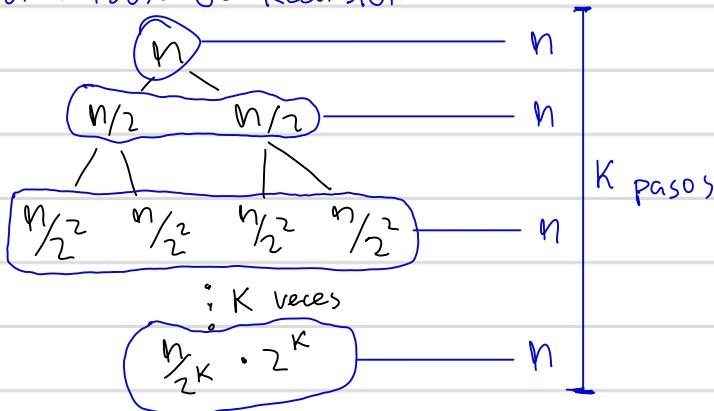
```

Test (n) {
    if (n > 1) {
        for (i=0; i < n; i++) {
            print ("Abdul Bari la fokin cabra")
        }
    }
    Test (n/2) → T(n/2)
    Test (n/2) → T(n/2)
}

```

$$T(n) = \begin{cases} 1 & \text{para } n=1 \\ 2T(n/2) + n & \text{para } n>1 \end{cases}$$

Por Árbol de Recursión



$$n = 2^K$$

$$\therefore T(n) = n \cdot K$$

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

Por Sustitución

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

$$T(n) = 2 \left[2 T(n/2) + n/2 \right] + n$$

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

$$T(n) = 2^2 \left[T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} \right] + 2n$$

$$T(n) = 2^3 T(n/2^3) + 3n$$

: k veces

$$T(n) = 2^K T(n/2^K) + K \cdot n$$

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

$$n = 2^k$$

$$\log(n) = K$$

$$\therefore T(n) = n \cdot T(1) + \log(n) \cdot n$$

$$T(n) = n + n \cdot \log(n)$$

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

Teorema Maestro para Funciones Divisorias

Funciona para $T(n)$ de la forma:

$$T(n) = \underbrace{a \cdot T\left(\frac{n}{b}\right)}_{a \geq 1} + f(n)$$

obs: si no hay log $\rightarrow P=0$

$$f(n) = \Theta\left(n^k \cdot \log^P(n)\right)$$

1er Paso: Hallar los valores: $\log_b(a)$ K
Para identificar el caso

Posibles Casos:

① $\log_b(a) > K \longrightarrow \Theta\left(n^{\log_b(a)}\right)$

② $\log_b(a) = K$

2.1) $P > -1 \longrightarrow \Theta\left(n^k \log^{P+1}(n)\right)$

2.2) $P = -1 \longrightarrow \Theta\left(n^k \log\log(n)\right)$

2.3) $P < -1 \longrightarrow \Theta\left(n^k\right)$

③ $\log_b(a) < K$

3.1) $P \geq 0 \longrightarrow \Theta\left(n^k \log^P(n)\right)$

3.2) $P < 0 \longrightarrow \Theta\left(n^k\right)$

Ejemplo TM 1

$$T(n) = \underbrace{2}_{a} T\left(\frac{n}{2}\right) + \underbrace{1}_{b} \rightarrow f(n) = n^k + \log^p(n) = n$$
$$\therefore k = 0 ; p = 0$$

$$\log_b(a) = \log_2(2) = 1$$

$$\log_b(a) > k \quad \therefore \text{es caso 1}$$

$$\therefore T(n) \in \Theta\left(n^{\log_b(a)}\right) = \Theta(n^1) = \Theta(n)$$

Ejemplo 2

$$T(n) = \underbrace{4}_{a} T\left(\frac{n}{2}\right) + \underbrace{n}_{b} \rightarrow f(n) = n^k \log^p(n) = n$$
$$\therefore k = 1 ; p = 0$$

$$\log_b(a) = \log_2(4) = 2$$

$$\log_b(a) > k \quad \therefore \text{es primer caso}$$

$$T(n) \in \Theta\left(n^{\log_b(a)}\right) = \Theta(n^2)$$

Ejemplo 3

$$T(n) = \underbrace{8}_{a} T\left(\frac{n}{2}\right) + \underbrace{n}_{b} \rightarrow f(n) = n^k \cdot \log^p(n) = n$$
$$\therefore k = 1 ; p = 0$$

$$\log_b(a) = \log_2(8) = 3$$

$$\log_b(a) > k \quad \therefore \text{es 1er caso}$$

$$T(n) \in \Theta\left(n^{\log_b(a)}\right) = \Theta(n^3)$$