UNIVERSIDAD NACIONAL
GUILLERMO BROWN

CLASE 10 - Unidad 6

Análisis de Algoritmos.

ESTRUCTURAS DE DATOS (271)

Clase N. 10. Unidad 6.

Clase 10: AGENDA

AGENDA

- **Temario:**
  - Análisis asintótico, comportamiento en el mejor caso, caso promedio y peor caso. Modelo computacional.
  - Concepto de tiempo de ejecución. Notación O(), Ω, Θ. Reglas generales para el calculo del tiempo de ejecución.
  - Calculo de tiempo y orden de ejecución en algoritmos iterativos y recursivos. Comparación de distintas estrategias de diseño de algoritmos..
- **Ejemplos** en Lenguajes **Python**
- **Temas relacionados y links de interés**
- **Práctica**
- **Cierre de la clase**

Análisis de algoritmos

- Nos permite comparar algoritmos en forma independiente de una plataforma en particular.
- Mide la **eficiencia** de un algoritmo, dependiendo del tamaño de la entrada de los datos

¿Cuando un algoritmo es eficiente?

¿Como elegir entre varios algoritmos para el mismo problema?

- Si el problema es sencillo → podemos elegir un algoritmo que sea más rápido programar o uno que ya este desarrollado.
- Si el problema es complejo → el proceso de selección del algoritmo deberá ser más cuidadoso y con criterio.

Análisis asintótico de algoritmos

El análisis asintótico es un concepto fundamental en la ciencia de la computación que describe cómo el tiempo y el espacio requeridos por un algoritmo aumentan en relación con el tamaño de la entrada.

Comprender esta complejidad es crucial para evaluar el rendimiento y la eficiencia de los algoritmos en diferentes situaciones.

Análisis asintótico de algoritmos

La notación asintótica es de suma importancia para determinar el tiempo de ejecución de los algoritmos y/o hacer comparaciones entre ellos. Sirve de parámetro de referencia estándar. Ya que:

- ❑ Diferentes maquinas, diferentes tiempos
- ❑ La misma máquina puede dar diferentes medidas dependiendo factores (S,O,, Interrupciones, etc)
- ❑ Máquinas con capacidad distinta o con arquitecturas diferentes.

.

Análisis asintótico de algoritmos

Definición: Sea $g : \mathbb{N} \longrightarrow \mathbb{R}^*$ una función arbitraria de los números naturales en los números reales no negativos. $O(g)$ representa el conjunto de todas las funciones $t : \mathbb{N} \longrightarrow \mathbb{R}^*$ tales que existe una constante real positiva M y un número natural n_0 de manera tal que para todo número natural $n \geq n_0$ se tiene que $t(n) \leq M * g(n)$. Simbólicamente:

$$O(g) = \{t : \mathbb{N} \rightarrow \mathbb{R}^* | \exists M \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : [n \geq n_0 \Rightarrow f(n) \leq M * g(n)]$$

Análisis asintótico de algoritmos

Definición: Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}^*$ dos funciones arbitrarias de los números naturales en los números reales no negativos. Se dice entonces que f está en O-grande de g, y se escribe $f \in O(g)$, si y sólo si existe una constante real positiva M y un número natural n_0 tales que para todo número natural $n \geq n_0$ se tiene que $f(n) \leq M * g(n)$. Simbólicamente:

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

$$\exists M \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : [n \geq n_0 \Rightarrow f(n) \leq M * g(n)]$$

Análisis asintótico de algoritmos

Ejemplo: Considere la función $f(n) = 8n + 128$, y suponga que se quiere mostrar que $f(n) \in O(n^2)$. Según la definición, se debe encontrar una constante real positiva M y un número natural n_0 tales que para todo número natural $n \geq n_0$ se verifique que $f(n) \leq M * n^2$. Suponga que se selecciona $M = 1$. Se tiene entonces que:

$$f(n) \leq M * n^2 \Leftrightarrow 8n + 128 \leq n^2$$

M = 1

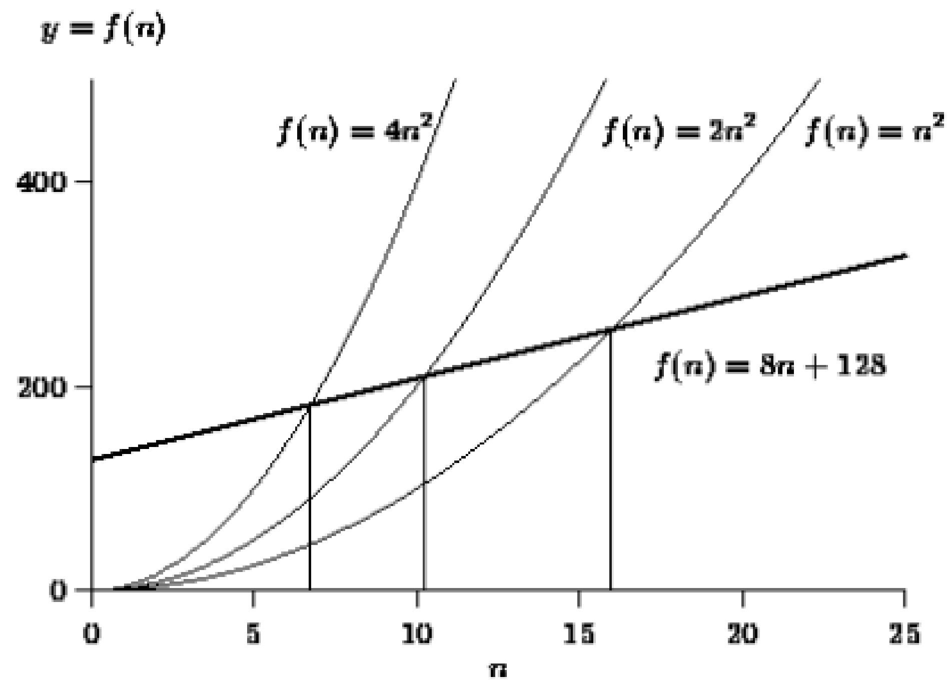
$$\Leftrightarrow 0 \leq n^2 - 8n - 128$$

Pasamos todos los términos del lado derecho

$$\Leftrightarrow 0 \leq (n - 16) * (n + 8)$$

Como $\forall n \in \mathbb{N} : n + 8 > 0$, se concluye que $n - 16 \geq 0$. Por lo tanto, $n_0 = 16$. Así que, para $M = 1$ y $n_0 = 16$, $f(n) \leq M * n^2$. Luego, $f(n) \in O(n^2)$.

Análisis asintótico de algoritmos



f(16)=8*16+128=256
f(17)=8*17+128=264
f(18)=8*18+128=272
,
,
,
f(25)=8*25+128=328
f(26)=8*26+128=336

Orden de crecimiento de $f(n) = 8n + 128$

Notaciones asintóticas

Las anotaciones asintóticas como **Big-O, Ω(Omega)** y **Θ(Theta)** son herramientas clave para el análisis de algoritmos. Estas notaciones nos permiten describir la complejidad temporal y espacial de un algoritmo en términos de su comportamiento a medida que aumenta el tamaño de la entrada. Por ejemplo, **Big O representa el límite superior del tiempo o espacio** que un algoritmo puede usar en el peor de los casos. En cambio, **Ω representa el límite inferior y Θ proporciona límites estrictos** que representan los límites superior e inferior de la complejidad. Estas características son importantes para comparar y clasificar algoritmos, lo que permite a los desarrolladores comprender el rendimiento de los algoritmos a medida que crece el problema y seleccionar más fácilmente el enfoque más eficiente para una tarea determinada.

Tiempo de ejecución

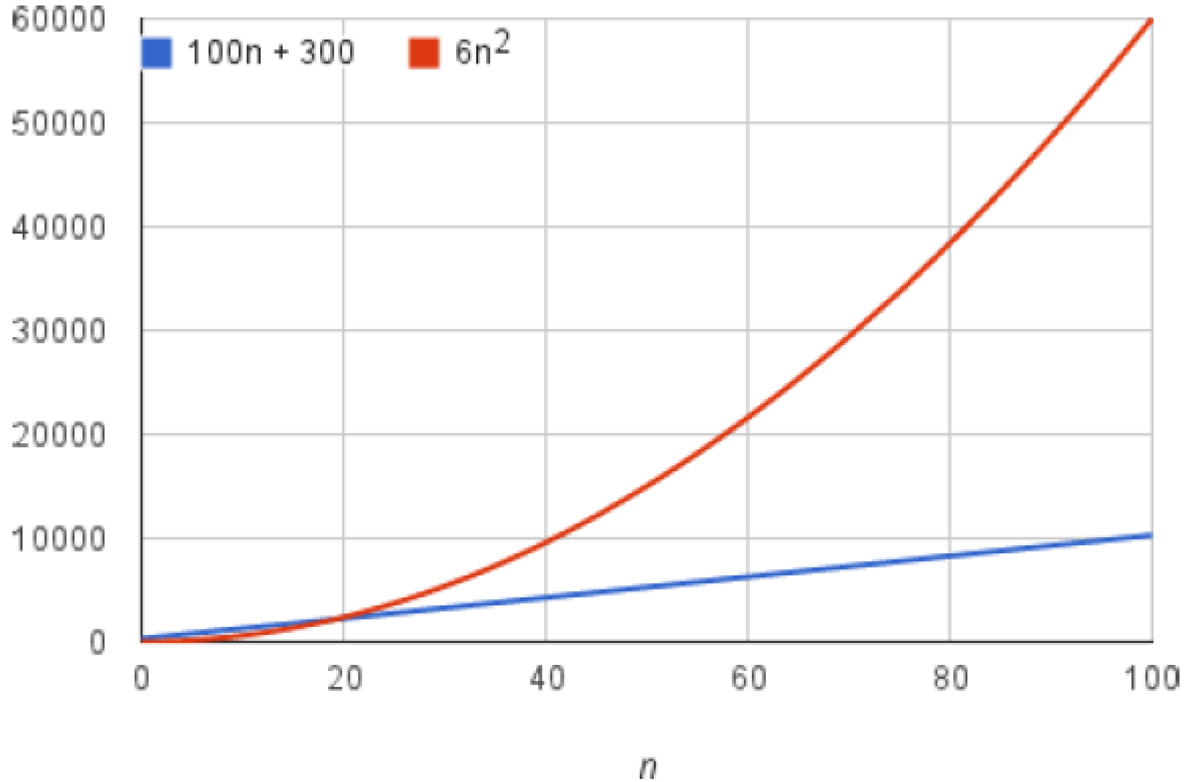
El tiempo de ejecución es la determinación del tiempo asintótico de ejecución, se recurre al redondeo en base a las secciones del código o programa que mas ciclos de CPU consumen. Debemos enfocarnos en cuán rápido crece una función T(n) respecto al tamaño de la entrada. A esto lo llamamos la **tasa o velocidad de crecimiento** del tiempo de ejecución.

Supongamos que un algoritmo, que se ejecuta con una entrada de tamaño n, tarda $6n^2+100n+300$ instrucciones de máquina.

Tiempo de ejecución

Gráfica que muestra los valores de $6n^2$ y de $100n+300$ para valores de n de 0 a 100:

El término $6n^2$ se vuelve más grande que el resto de los términos, $100n+300$ una vez que n se hace suficientemente grande.



Clase 10: Notación Big-Oh

Big-Oh definición:

$$T(n) = O(f(n))$$

si existen constantes **c > 0** y $\mathbf{n_0}$ tales que:

$$T(n) <= c\ f(n) \text{ para todo } n >= n_0$$

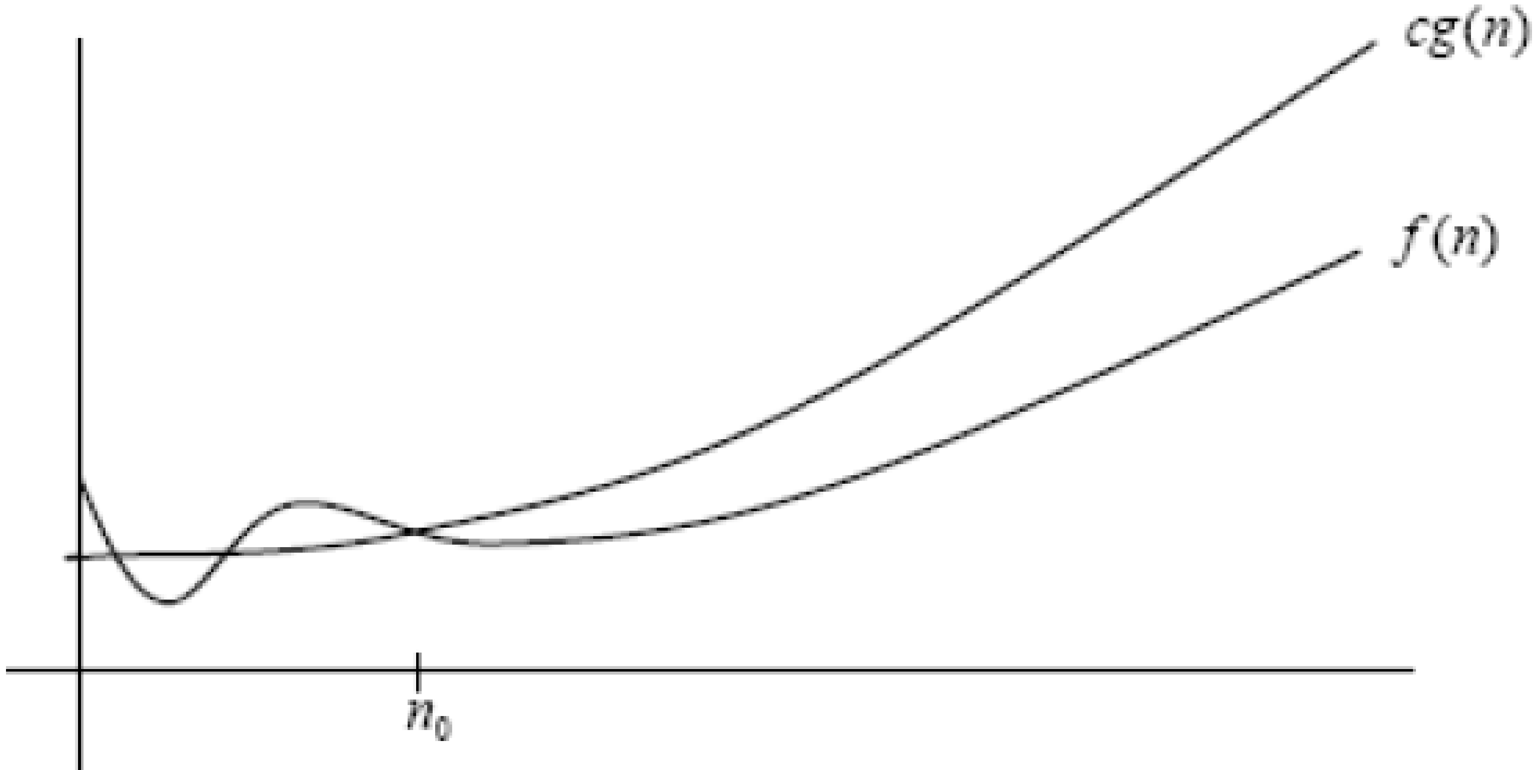
Se lee: T(n) es de orden de f(n)

f(n) representa una cota superior de T(n)

La tasa de crecimiento de T(n) es menor o igual que la de f(n)

Big-Oh representación:

$$f(n) = O(g(n))$$



Clase 10: Notación Big-Oh

Notación Big-Oh:

$$f(n) = O(g(n))$$

La notación O grande solamente da una cota asintótica superior, y no una cota asintóticamente ajustada, podemos hacer declaraciones que en primera instancia parecen incorrectas, pero que son técnicamente correctas.

Propiedades Notación Big-*Oh:*

- Reflexividad: $f \in O(f)$
- Transitividad: $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
- $\lambda * O(f) = O(f)$
- Si $\lambda > 0$ entonces $O(\lambda * f) = O(f)$
- $O(f) + O(g) = O(max(f,g))$ (Regla de la suma)
- $O(f) * O(g) = O(f * g)$ (Regla del producto)

Propiedades Notación Big-*Oh:*

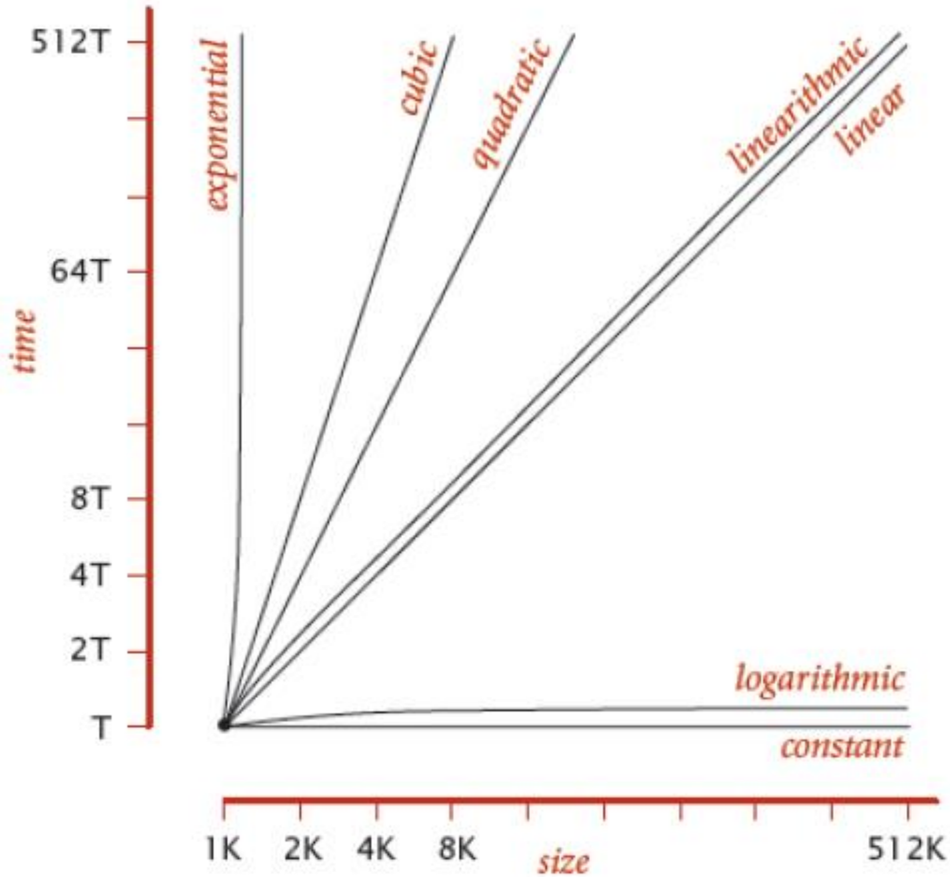
Otras reglas:

- $T(n)$ es un polinomio de grado $\boldsymbol{k} \Rightarrow T(n) = O(n^k)$
- $T(n) = \log^k(n) \Rightarrow O(n)$ para cualquier k

  n siempre crece más rápido que cualquier potencia de $\log(n)$
- $T(n) = cte \Rightarrow O(1)$
- $T(n) = cte * f(n) \Rightarrow T(n) = O(f(n))$

Ordenes de crecimiento:

Orden	Nombre
$O(1)$	Constante
$O(log(n))$	Logarítmico
$O(n)$	Lineal
$O(nlog(n))$	nlog(n)
$O(n^2)$	Cuadrático
$O(n^3)$	Cúbico
$O(n^k)$, $k > 3$	Polinomial
$O(2^n)$	Exponencial



Ejemplos:

Cuales son correctos?

1.- $T(n) = 3n^3 + 2n^2$ es $O(n^3)$?

2.- $T(n) = 3n^3 + 2n^2$ es $O(n^4)$?

3.- $T(n) = 1000$ es $O(1)$?

4.- $T(n) = 3^n$ es $O(2^n)$?

Ejemplos:

1.- $T(n) = 3n^3 + 2n^2$ es $O(n^3)$? True

2.- $T(n) = 3n^3 + 2n^2$ es $O(n^4)$? True

3.- $T(n) = 1000$ es $O(1)$? True

4.- $T(n) = 3^n$ es $O(2^n)$? False

Ejemplos:

```python
def constant_example(arr):
    return arr[0]
```

O(1) independientemente de los datos de entrada

```python
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

O(log n) En una lista con datos Ordenados

Ejemplos:

```python
def linear_search(arr, target):
    for i in arr:
        if element == target:
            return i
    return -1
```

O(*n*)
El tiempo de ejecución es directamente proporcional a los datos de entrada

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

$O(n^2)$
El tiempo es proporcional al cuadrado del tamaño de entrada

Clase 10: Notación-Big-Oh

unab

Ejemplos:

```
def generate_combinations(s):
    if len(s) == 0:
        return ['']
    prev_combinations = generate_combinations(s[1:])
    new_combinations = []
    for combo in prev_combinations:
        new_combinations.append(combo)
        new_combinations.append(s[0] + combo)
    return new_combinations

```

$O(e^n)$

Problema es Armar todas las combinaciones posibles de una cadena. El tiempo de ejecución crece exponencialmente con la longitud de la cadena.

Ejemplos:

```
print ("Programa que calcula el factorial")
numero = int(input("Introduzca el número: "))

factorial = 1

i = 1
while (i <= numero):
    factorial = factorial * i
    i = i + 1

print ("El factorial de " + str(numero) + " es " + str(factorial))
```

$$T(n) = cte_1 + \sum_{i=\,.}^{n} \left(cte_2 + cte_3 \right)$$

T(n)= cte_1+(n*(cte_2 + cte_3)) si cte_2 + cte_3 = cte_4 Reemplazando en T(n)

T(n) = cte_1+(n * cte_4)=>por prop. de *O*, *O*(1)+cte_4***O*(n)=***O*(n)**

Ejemplos:

```
def factorial(n):
  if n==0 or n==1:
        resultado=1
  elif n>1:
        resultado=n*factorial(n-1)
  return resultado
```

$$T(n) = \begin{cases} cte_1 & n = 1 \\ cte_2 + n_* T(n-1) & n > 1 \end{cases}$$

Clase 10: Notación-Big-Oh

Ejemplos:

$$T(n) = \begin{cases} cte_1 & n = 1 \\ cte_2 + n * T(n-1) & n > 1 \end{cases}$$

$T(n) = cte_2 + i * T(n-i)$ para cualquier valor $i >= 1$

$n-i = 1 => -i = 1-n => i = -1+n => i = n-1$

Reemplazando en T(n)

$T(n) = cte_2+(n-1)*T(n-(n-1)) = cte_2+(n-1)*T(n-n+1)=cte_2+(n-1)*T(1)$

$cte_2+(n-1)*cte_3 = T(cte_2)+T((n-1)*cte_3)$=por prop. de O =

$O(1)+cte_3*O(n-1)=O(1)+O(n-1) = O(n-1) = O(n)$

Clase 10: Notación-Big-Oh

Ejemplos:

Máximo en un arreglo

$$T(n) = \begin{cases} cte_1 & n = 1 \\ 2 * T(n/2) + cte_2 & n > 1 \end{cases}$$

$$T(n) = \underbrace{2 * T(n/2)}\ + cte_2 \qquad n > 1$$

$$\underbrace{2 * T(n/4)}\ + cte_2$$

$$\underbrace{2 * T(n/8)}\ + cte_2$$

$$2 * T(n/16) + cte_2$$

Desarrollamos la recurrencia

Clase 10: Notación-Big-Oh

Ejemplos:

$T(n) = 2 * T(n/2) + cte_2 = 2 * [2 * T(n/4) + cte_2] + cte_2 =$

$= 4 * T(n/4) + 3\ cte_2 = 4 * [2 * T(n/8) + cte_2] + 3\ cte_2 =$

$= 8 * T(n/8) + 7\ cte_2 = 8 * [2 * T(n/16) + cte_2] + 7\ cte_2 =$

$= 16 * T(n/16) + 15\ cte_2 = \ldots\ldots\ldots$

Paso i:

$T(n) = 2^i * T(n/2^i) + (2^i - 1) * cte_2$

El desarrollo termina cuando $T(n/2^i)=1$

$T(n) = 2^i * T(n/2^i) + (2^i - 1) * cte_2$

Cuando $\mathbf{n/2^i= 1}$ $\mathbf{n = 2^i}$ x prop de log $\mathbf{i= log_2 n}$,

Reemplazamos **i** en la expresión de T(n)

$T(n) = n * T(n/n) + (n - 1) * cte_2 = n * cte_1 + (n - 1) * cte_2 = \boldsymbol{O}\mathbf{(n)}$

Clase 10: Notación-Big-Oh

Ejemplos:

```
def sumatoria_it(n):
    x = 0
    for i in range(n):
        x = x + i
    return x
```

$$T(n) = cte_1 + \sum_{i=1}^{n} cte_2$$

$$T(n) = O(n)$$

```
def sumatoria(n):
    if(n>0):
        return  n+sumatoria(n-1)
    else:
        return 0
```

$$T(n) = \begin{cases} cte_1 & n = 0 \\ cte_2 + T(n-1) & n >= 1 \end{cases}$$

$$T(n) = O(n)$$

Clase 10: Notación-Big-Oh

Ejemplos:

$$T(n) = cte_1 + \sum_{i=1}^{n} cte_2$$

$$T(n) = O(n)$$

$T(n) = cte_1 + n * cte_2$

$T(n) = cte_1 + O(n * cte_2)$

$T(n) = cte_1 + cte_2 * O(n)$

$T(n) = \boldsymbol{O}\mathbf{(n)}$

$$T(n) = \begin{cases} cte_1 & n = 0 \\ cte_2 + T(n-1) & n >= 1 \end{cases}$$

$$T(n) = O(n)$$

$T(n) = cte_1 + T(n-1)$

$= cte_1 + (T(n-2) + cte_1)$

$= cte_1 + (cte_1 + (T(n-3) + cte_1))$

$= i * cte_1 + T(n-i)$ **Cuando n-i = 0**

entonces n = i, Reemplazamos **i en** T(n)

$T(n) = n * cte_1 + T(n-n) = cte_1 * O(n)$ **= *O*(n)**

Consultas

Temas a desarrollar la próxima clase

- Grafos orientados y no orientados.
- Grafos pesados. Distintas representaciones: Listas de Adyacencia y Matriz de Adyacencia.
- Definiciones básicas y conceptos fundamentales. Grafos acíclicos. Grafos conexos y dígrafos
- fuertemente conexos.