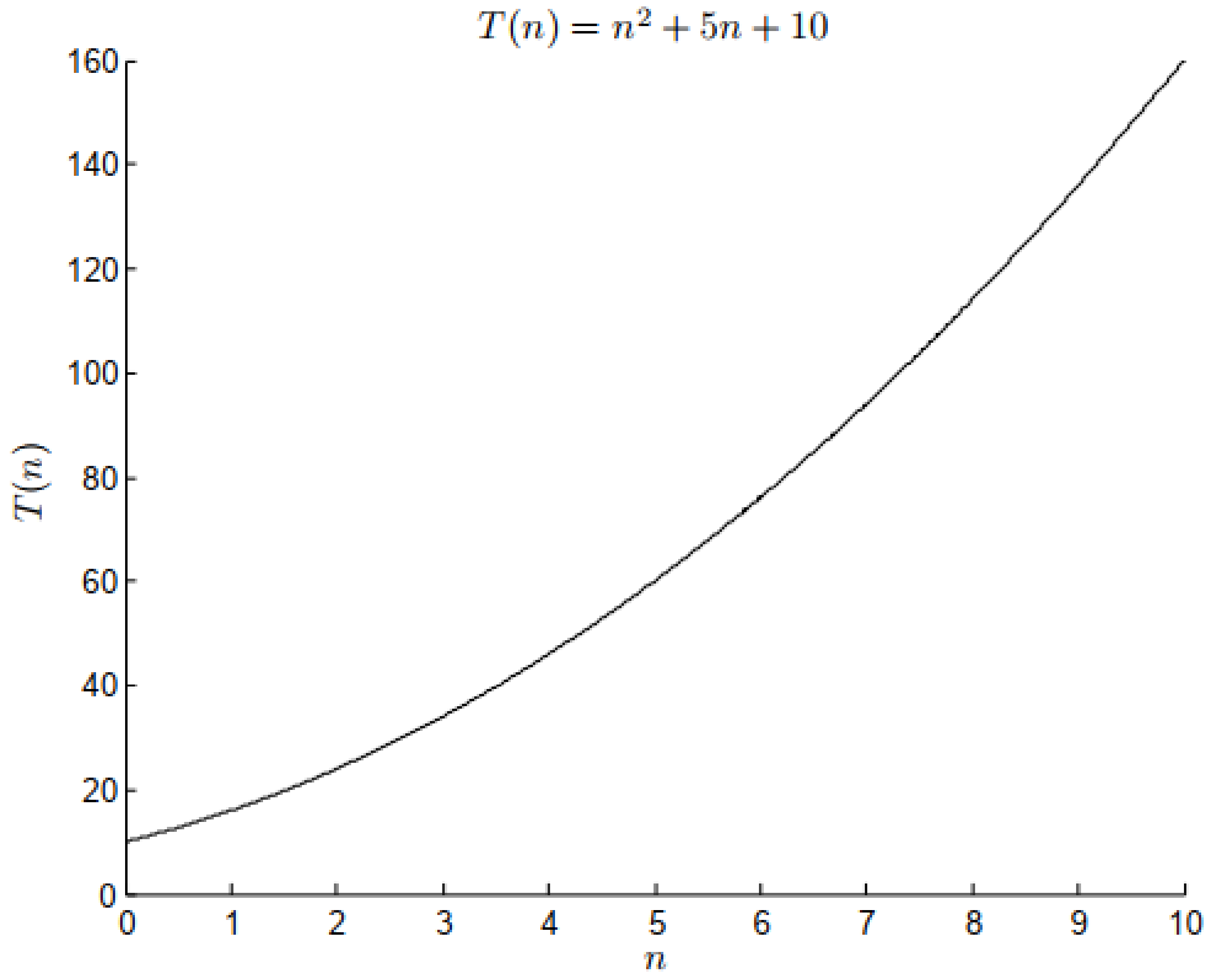
Análisis de Algoritmos

- Para **cuantificar la eficiencia** de los algoritmos, utilizamos funciones que miden, por ejemplo:
  - Cuánto **tiempo demora** un algoritmo en ejecutarse sobre una entrada dada,
  - Cuál es su **peor caso** sobre un conjunto de entradas posibles,
    - O cuánto demora en promedio, suponiendo una cierta distribución de probabilidad de las entradas.
  - También estudiaremos el uso de otro tipo de recursos, como por ejemplo la **cantidad de memoria utilizada**.

Tiempo Computacional $T(n)$

- Cuenta el número de operaciones fundamentales que el algoritmo realiza, lo cual puede variar según el tamaño de la entrada n.
- El tiempo de ejecución lo deberemos expresar mediante una fórmula (función) matemática.



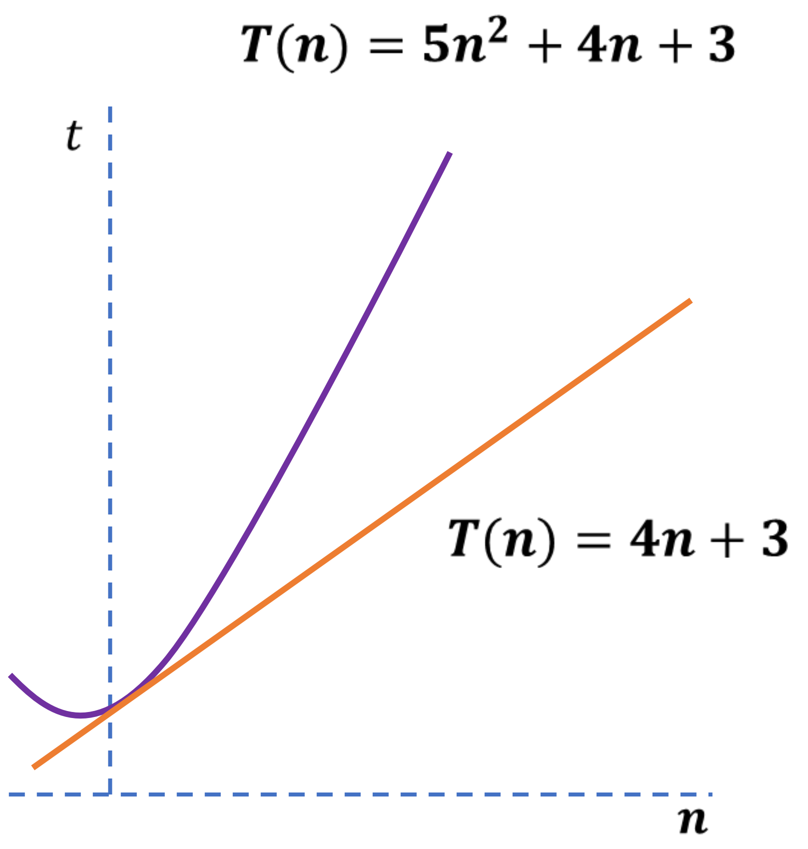
Tiempo Computacional $T(n)$

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

```
int suma = 0;
for (int i = 1; i <= n; i++)
    suma += suma;
```

`int suma = 0;`	1
`for (int i = 1; i <= n; i++)`	$2n+2$
`suma += suma;`	$2n$

$$\boldsymbol{T(n) = 4n + 3}$$



$$\sum_{i=1}^{n}\sum_{j=1}^{n} i * j$$

```
int suma = 0;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        suma = suma + i*j;
```

Tiempo Computacional $T(n)$

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

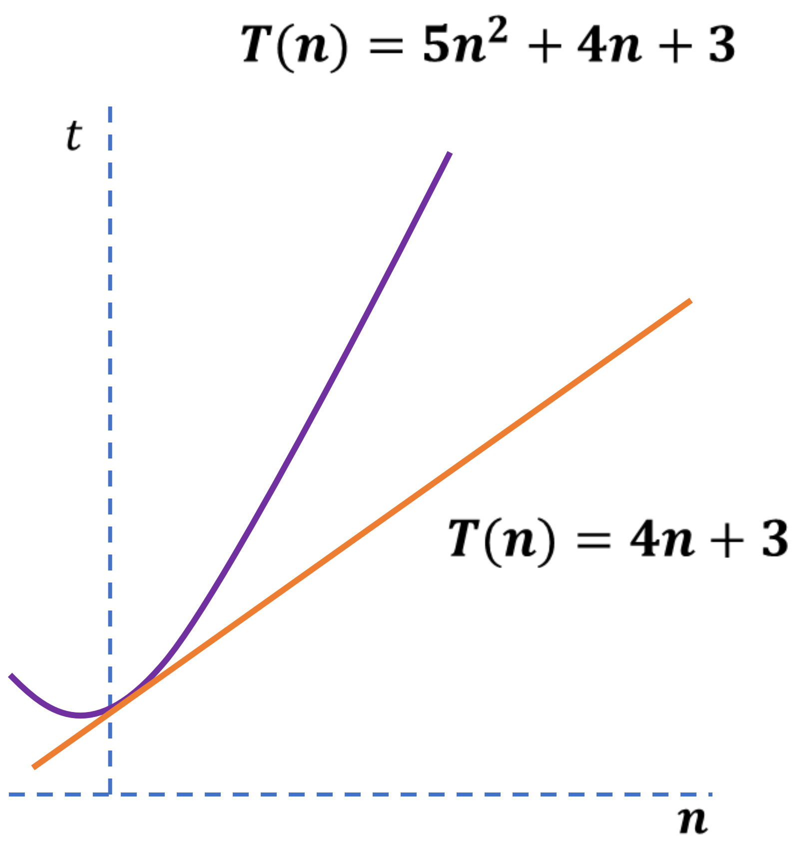
```
int suma = 0;                      1
for (int i = 1; i <= n; i++)       2n + 2
    suma += i;                     2n
```

$$\boldsymbol{T(n) = 4n + 3}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} i * j$$

```
int suma = 0;                          1
for (int i = 1; i <= n; i++)           2n + 2
    for (int j = 1; j <= n; j++)       (2n + 2) * n
        suma = suma + i*j;             3n^2
```

$$\boldsymbol{T(n) = 5n^2 + 4n + 3}$$



Tiempo Computacional $T(n)$

```
int i=1;
while(i<=n) {
    i = i * 2;
}
```

Valores de i en cada iteración x:

$2^1, 2^2, 2^3, 2^4, 2^5, \ldots, 2^x$

En algún momento:

$i > n$, es decir $2^x > n$

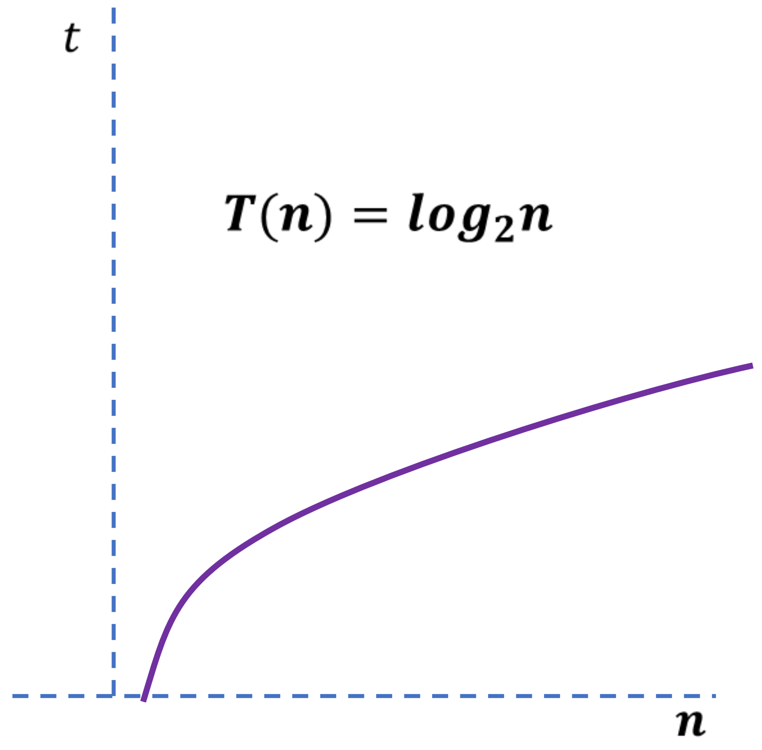
Entonces:

$2^x > n \rightarrow$

$log_2 2^x > log_2 n \rightarrow$

$x\ \cancel{log_2 2} > log_2 n \rightarrow$

$\boxed{\boldsymbol{x > log_2 n}}$ ← **Tiempo Computacional**



Tiempo Computacional $T(n)$

Análisis del Algoritmo Fibonacci Recursivo

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

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

::

$T(n-2)$ es casi $T(n-1)$, entonces:

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

$$T(n) = 2\{2T(n-2) + c\} + c = 4T(n-2) + 3c$$

$$T(n) = 4\{2T(n-3) + c\} + 3c = 8T(n-3) + 7c$$

$$T(n) = 16T(n-4) + 15c$$

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

Condición de parada $T(0), n - k = 0 \rightarrow k = n$

$$T(n) = 2^n T(0) + (2^n - 1)c$$

Tiempo Computacional $\longrightarrow$ $\boxed{\boldsymbol{T(n) = (1+c)2^n - c \approx 2^n}}$

Tiempo Computacional $T(n)$

Ejercicio: hallar el tiempo computacional del siguiente algoritmo (acotado al peor caso)

```
void insertion_sort(T *array, int n)
{
    int actual, j;
    for (int i = 1; i < n; ++i)
    {
        actual = array[i];
        j = i - 1;
        while (j >= 0 && array[j] > actual)
        {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = actual;
    }
}
```

Tiempo Computacional $T(n)$

Ejercicio: hallar el tiempo computacional del siguiente algoritmo (acotado al peor caso)

```
void insertion_sort(T *array, int n)
{
    int actual, j;                                      // 1
    for (int i = 1; i < n; ++i)                         // n
    {
        actual = array[i];                              // n
        j = i - 1;
        while (j >= 0 && array[j] > actual)             // n^2
        {
            array[j + 1] = array[j];                    // n^2
            --j;                                        // n^2
        }
        array[j + 1] = actual;                          // n
    }
}
```

$$\boldsymbol{T(n) = 3n^2 + 4n + 1}$$

Tiempo Computacional $T(n)$

Ejercicio: hallar el tiempo computacional de las siguientes sentencias

```
for (int i = 1; i <= n; i *= c ) {
    // cualquier sentencia constante
}
```

```
for (int i = n; i > 0; i /= c) {
    // cualquier sentencia constante
}
```

```
for (int i = 2; i <= n; i = pow(i, c)) {
    // cualquier sentencia constante
}
```

Tiempo Computacional $T(n)$

Ejercicio: hallar el tiempo computacional de las siguientes sentencias

```
for (int i = 1; i <= n; i *= c ) {
    // cualquier sentencia constante
}
```

$\longrightarrow$ $\boldsymbol{T(n) \approx log_c n}$

```
for (int i = n; i > 0; i /= c) {
    // cualquier sentencia constante
}
```

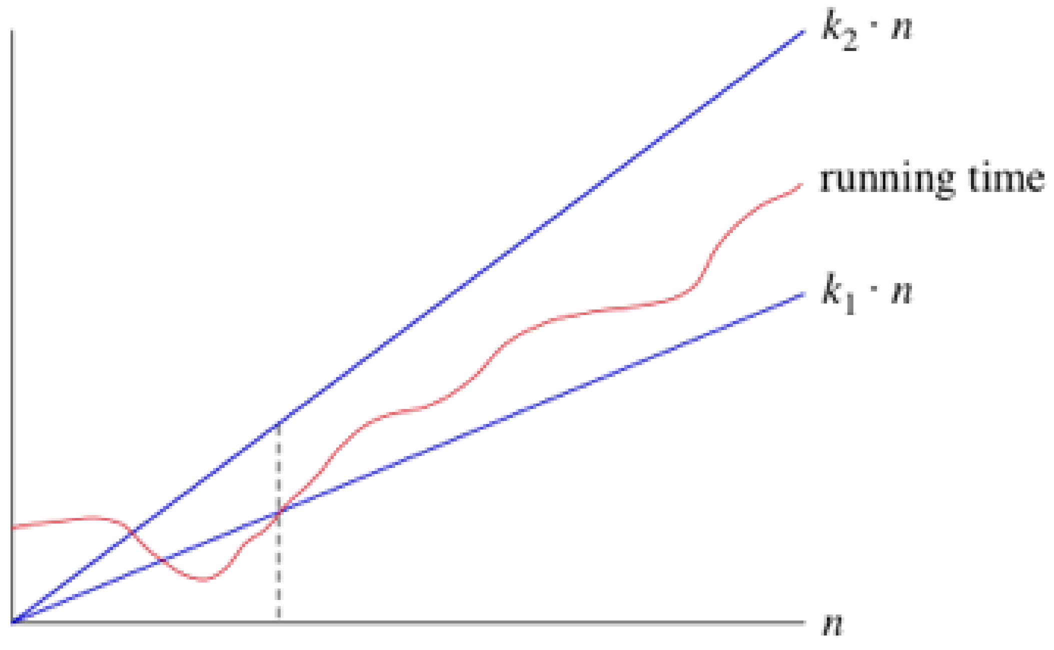
$\longrightarrow$ $\boldsymbol{T(n) \approx log_c n}$

```
for (int i = 2; i <= n; i = pow(i, c)) {
    // cualquier sentencia constante
}
```

$\longrightarrow$ $\boldsymbol{T(n) \approx log\ log_c n}$

Notación Asintótica

- Permite simplificar la taza de crecimiento de un algoritmo
- Eficiencia asintótica de algoritmos
  - Asumimos que las entradas son muy grande
  - Nos interesa el **“orden de crecimiento”**
  - Las constantes y términos de orden inferior no son relevantes, al ser dominados por un termino de orden superior.
- El algoritmo con mejor coste o eficiencia asintótica suele ser la mejor elección.
  - Salvo para entradas muy pequeñas



Órdenes que más aparecen

- Considerados generalmente como “tratables”

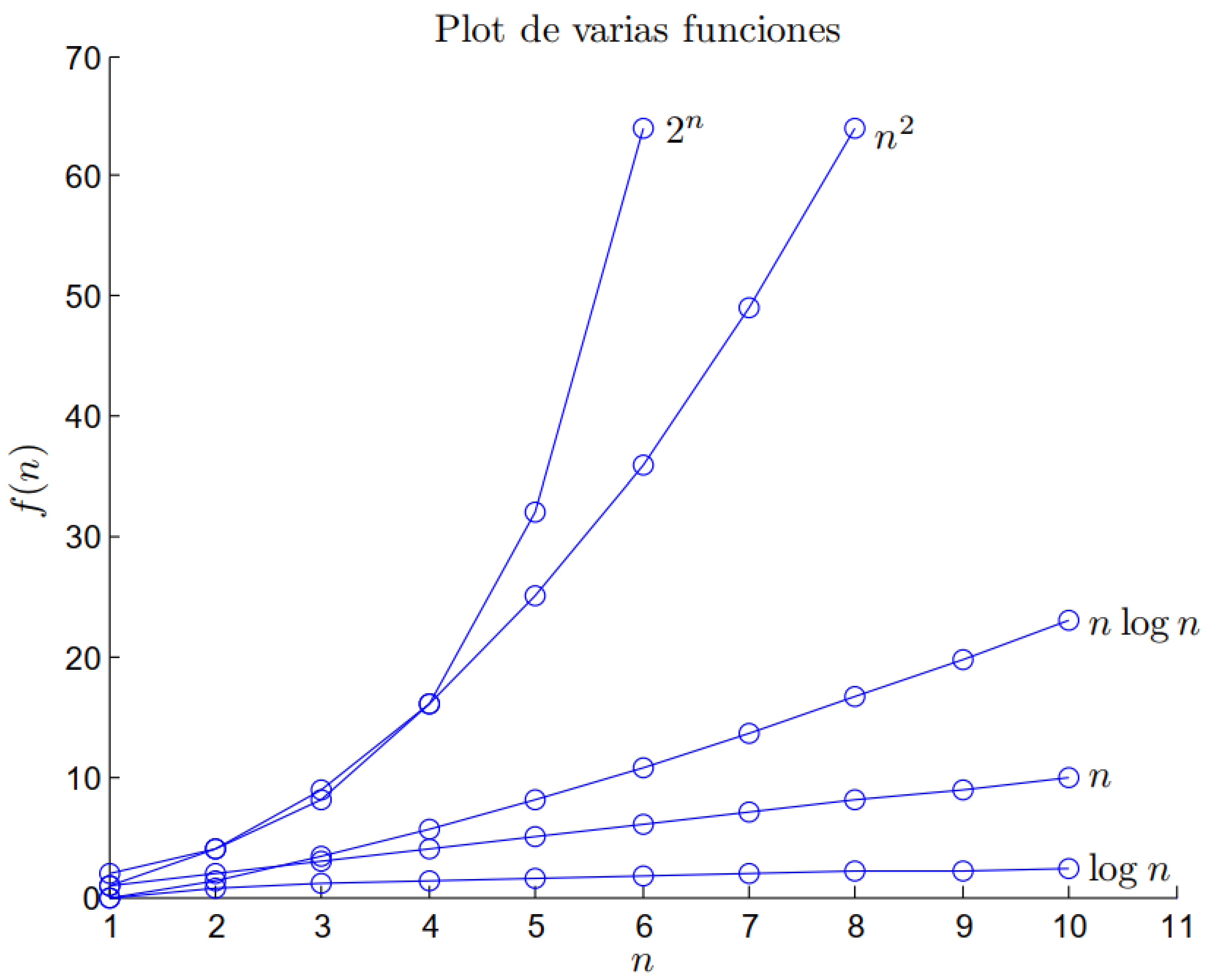
$$1 < \log n < n < n \log n < n^2$$

- Considerados generalmente como “intratables”

$$n^2 < n^3 < 2^n < n!$$

- n^2 se encuentra en el limite
- Siempre hay que tener en cuenta el tamaño de la entrada (n) para poder decir si un problema es tratable o intratable para cierto algoritmo

Órdenes que más aparecen



$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

- Un orden exponencial es extremadamente costoso, incluso frente a ordenes polinómicos.
- Un orden factorial es incluso más costoso que un orden exponencial.

¿Qué es la notación big O?

Provee un límite superior a la tasa de crecimiento de una función que representa el tiempo computacional de un algoritmo.

$$\boxed{f(n) \in \mathcal{O}(g(n)) \iff f(n) \leq g(n)}$$

- $f(n)$ es asintóticamente menor o igual que $g(n)$
- $g(n)$ es una cota superior de $f(n)$

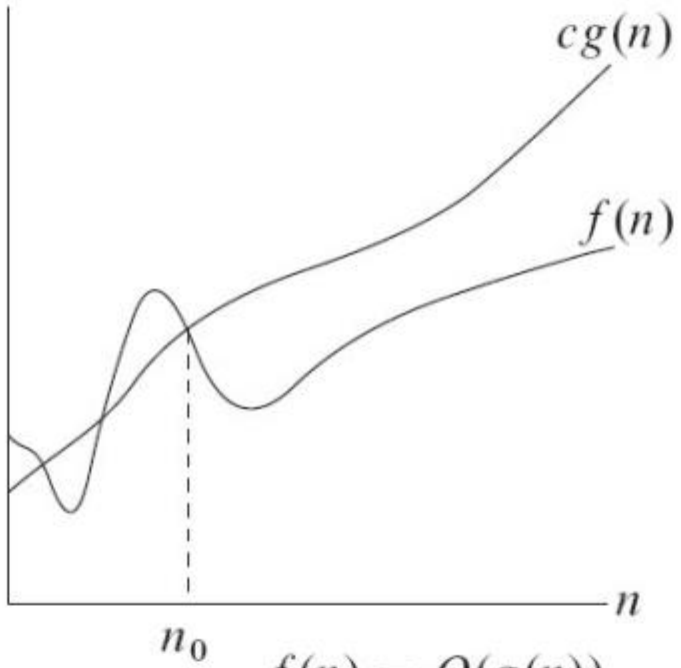
- $2n + 5 \in \mathcal{O}(3n^2 - 8n)$
- $2n + 5 \in \mathcal{O}(n + 10)$
- $2n + 5 \in \mathcal{O}(n!)$
- $2n + 5 \in \mathcal{O}(n)$ ← **Nos fijamos en la cota superior mas baja**

- $T = a = \boldsymbol{O}(\mathbf{1})$
- $T = an + b = \boldsymbol{O}(\boldsymbol{n})$
- $T = an^2 + bn + c = \boldsymbol{O}(\boldsymbol{n}^2)$

¿Qué es la notación big O?

- Definición formal

$$\mathcal{O}(g(n)) = \left\{ f(n) : \exists\ c > 0 \text{ y } n_0 > 0\ /\ 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0 \right\}$$



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

- La ide principal es que a partir de n_0, la función $c.\,g(n)$ siempre supera a $f(n)$

¿Qué es la notación big O?

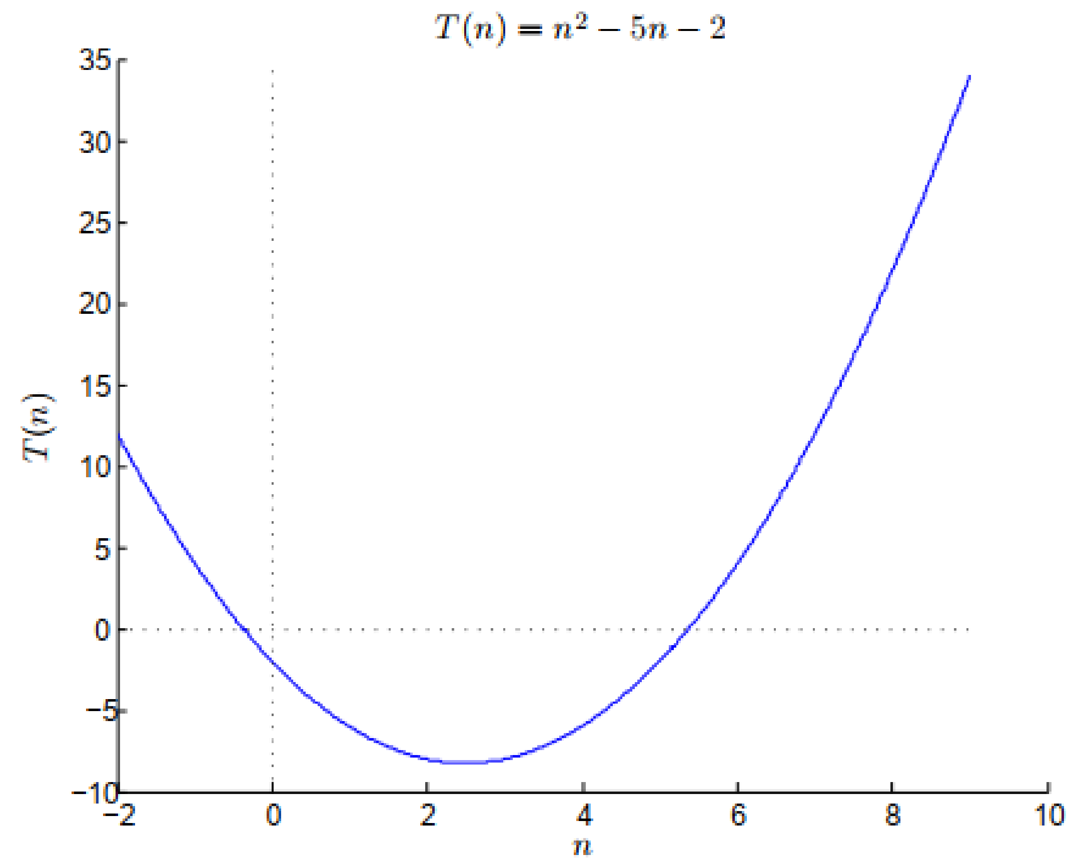
- Para demostrar que una función $f(n) \in O(g(n))$ será necesario encontrar una (cualquier) pareja de constantes $c > 0$ y $n_0 > 0$, de tal forma que se verifiquen las condiciones de la definición.
- Ejemplo: demostrar que $\boldsymbol{5n + 2 \in O(n)}$

¿Qué es la notación big O?

- Para demostrar que una función $f\ (n)\ \in O(g(n))$ será necesario encontrar una (cualquier) pareja de constantes $c\ >\ 0$ y $n_0 >\ 0$, de tal forma que se verifiquen las condiciones de la definición.
- Ejemplo: demostrar que $\mathbf{5}\boldsymbol{n}\ +\ \mathbf{2}\ \in\ \boldsymbol{O}(\boldsymbol{n})$
  - Hay que encontrar $c\ >\ 0$ y $n_0 > 0$ tales que $5n\ +\ 2\ \leq\ cn, \forall n\ \geq n_0$
  - Para ello, elegimos una constante adecuado (por ejemplo $c = 6$)
  - Buscamos un $n > 0$ para que se cumpla $5n\ +\ 2\ \leq\ cn$
  - Encontramos que con $c = 6$ se cumple para todo $n\ \geq\ 2$, entonces $n_0 = 2$.
    - Hay infinitas parejas mas, pero basta con encontrar una.

¿Qué es la notación big O?

- Ejemplo: demostrar que $\mathbf{5n + 2 \in O(n^2)}$
  - Hay que encontrar $c > 0$ y $n_0 > 0$ tales que $5n + 2 \leq cn^2, \forall n \geq n_0$
  - Elegimos $c = 1$, buscamos que valores de n cumplen $5n + 2 \leq n^2$
    - Resolvemos la desigualdad $n^2 - 5n - 2 \geq 0$
    - Las raíces de la función cuadrática son -0.37 y 5.37
    - Por lo tanto, siempre será positiva para $n_0 = 6$



- Ejemplo: demostrar que $3n^2 + 2n - 2 \in O(n)$

Cota inferior Ω

Provee un límite inferior a la tasa de crecimiento de una función que representa el tiempo computacional de un algoritmo.

$$\boxed{f(n) \in \Omega(g(n)) \quad \Longleftrightarrow \quad f(n) \geq g(n)}$$

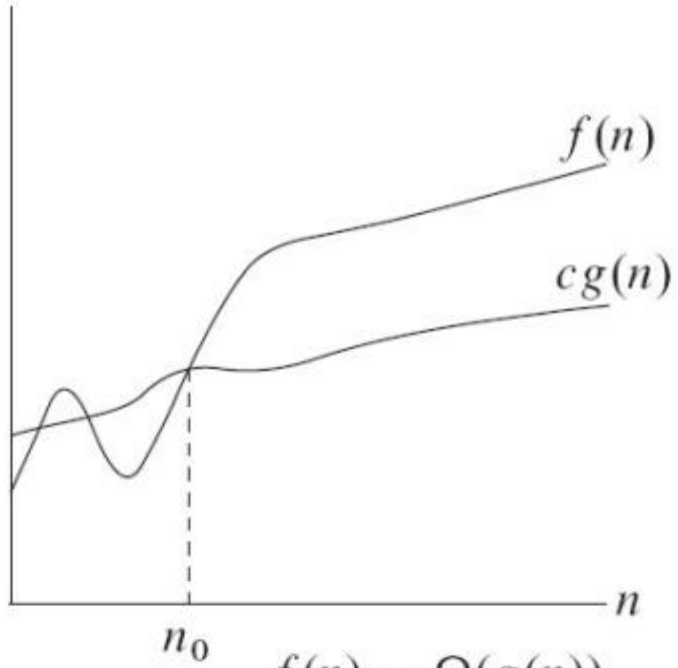
- $f(n)$ es asintóticamente mayor o igual que $g(n)$
- $g(n)$ es una cota inferior de $f(n)$

- $2n + 5 \in \Omega(3 \log n)$
- $2n + 5 \in \Omega(4n + 10)$
- $2n + 5 \in \Omega(1)$
- $2n + 5 \in \Omega(n)$ ← **Nos fijamos en la cota inferior mas alta (simplificado)**

Cota inferior Ω

- Definición formal

$$\Omega(g(n)) = \left\{ f(n) : \exists\ c > 0 \text{ y } n_0 > 0\ /\ 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0 \right\}$$



$f(n) = \Omega(g(n))$

- La ide principal es que a partir de n_0, la función $f(n)$ siempre supera a $c.g(n)$

Cota ajustada Θ

Provee un límite ajustado a la tasa de crecimiento de una función que representa el tiempo computacional de un algoritmo.

$$\boxed{f(n) \in \Theta(g(n)) \quad \Longleftrightarrow \quad f(n) = g(n)}$$

- $f(n)$ es asintóticamente igual que $g(n)$
- $g(n)$ es una cota ajustada de $f(n)$

Ejemplo:

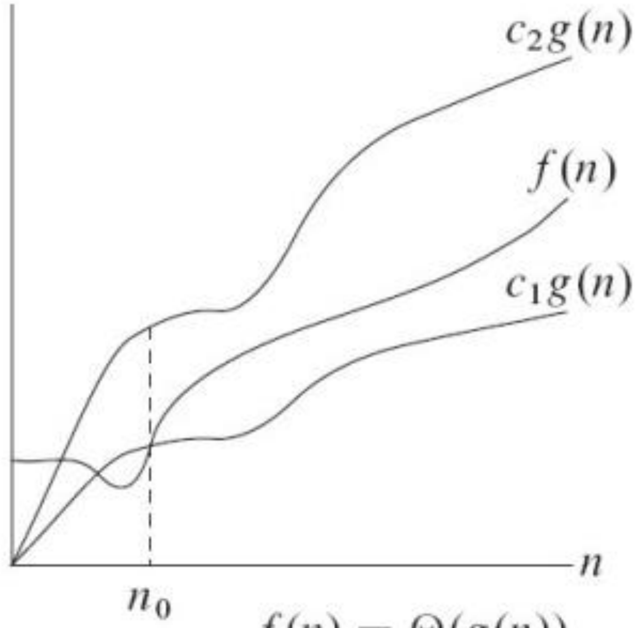
- $2n + 5 \in \Theta(8n + 10)$
- $2n + 5 \in \Theta(n)$

Cota ajustada Θ

- Definición formal

$$\Theta(g(n)) = \Big\{ f(n) : \exists\ c_1 > 0,\ c_2 > 0 \text{ y } n_0 > 0\ /$$

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n),\ \forall n \ge n_0 \Big\}$$



$f(n) = \Theta(g(n))$

- La ide principal es que a partir de n_0, la función $f(n)$ siempre queda en medio de $c_1 . g(n)$ y $c_2 . g(n)$