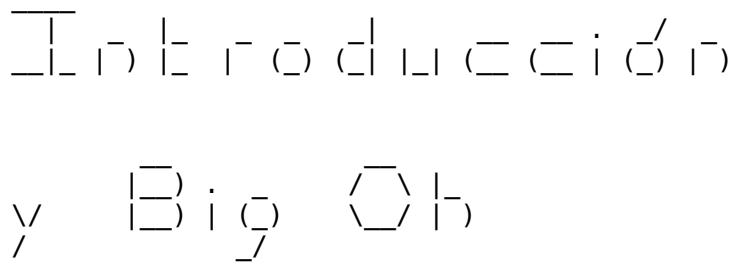
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬▬ 1. Introducción.
▬▬▬▬ 2. Diferencias Entre Algoritmos y Programas.
▬▬▬▬ 3. Tiempo de Ejecución de un Programa.
▬▬▬▬ 4. Tipos de Funciones.
▬▬▬▬ 5. Notación Asintótica.
▬▬▬▬ 6. Big Oh.
▬▬▬▬ 7. Omega.
▬▬▬▬ 8. Theta.
▬▬▬▬ 9. Es Posible Encontrar Theta en Todos los Casos?.
▬▬▬▬ 10. La Tiranía de la Velocidad.
▬▬▬▬ 11. El Problema del TSP.
▬▬▬▬ 12. Resumen.
▬▬▬▬ 13. Ejercicios.

▬▬▬▬ 1. Introducción.

Un algoritmo es un conjunto de instrucciones para lograr una tarea. Un programa es cuando escribimos un algoritmo en algún tipo de lenguaje computacional. Sin embargo estos dos términos tienden a usarse casi como sinónimos. Desde esta perspectiva, cualquier pedazo de código podría llamarse un programa, pero en este documento, trataremos de cubrir los más interesantes. Se han escogido los algoritmos que son eficientes, interesantes, o ambas cosas.

Para muchos de los algoritmos presentados se pueden encontrar implementaciones en muchos lenguajes. Pero implementar un algoritmo en un programa es algo completamente inútil si no se conocen las restricciones de tiempo y eficiencia. Cuando se escoge una estructura de datos y una forma particular de programación, se pueden obtener resultados muy rápidos, o no tan rápidos. Entender estos problemas es el objetivo principal de este documento.

▬▬▬ 2. Diferencias Entre Algoritmos y Programas.

Un algoritmo puede escribirse de muchas maneras. Desde un párrafo que describe las tareas que se deben hacer, hasta un pseudocódigo. El programa es más sencillo de definir. Tiene que poder ejecutarse en un computador.

Estas son las principales diferencias entre un algoritmo y un programas.

Algoritmo.	Programa.
Se usa en el diseño.	Se usa en la implementación.
Lo hace un analista.	Lo hace un programador.
Se describe de muchas formas.	Se describe mediante un lenguaje de programación
Independiente del sistema operativo y el hardware.	Es mucho más dependiente.

▬▬▬ 3. Tiempo de Ejecución de un Programa.

Cuando se resuelve un problema, con frecuencia se puede elegir entre varios algoritmos. Cómo elegir uno de ellos es una pregunta difícil de responder. Hay algunos objetivos que se deben cumplir y estos objetivos pueden ser opuestos. Se suele escoger un algoritmo que tenga alguna de estas cualidades.

1. Que sea el algoritmo más fácil de entender, codificar y corregir.

2. Que el algoritmo use eficientemente los recursos del computador. En especial que se ejecute con la mayor rapidez posible.

Cuando se escribe un programa que se va a ejecutar una o pocas veces, el primer objetivo es el más importante. En tal caso, es muy probable que el costo del tiempo de programación exceda en mucho al costo de ejecución del programa.

En cambio, cuando se presenta un problema cuya solución se va a utilizar muchas veces, el costo de ejecución del programa puede superar en mucho al de la escritura, en especial si en la mayor parte de las ejecuciones se dan entradas de gran tamaño. Entonces es mucho mejor realizar un algoritmo complejo, siempre que el tiempo de ejecución del programa sea significativamente menor que el del programa más sencillo de entender y de escribir.

En muchas situaciones se acostumbra iniciar con un algoritmo simple, con el objetivo de determinar el beneficio real que se obtendría al escribir un programa más complicado.

El tiempo de ejecución de un programa depende de muchos factores. Entre ellos se pueden enunciar los siguiente:

1. Los datos de entrada al programa.

2. La calidad del código generado por el compilador utilizado.

3. La naturaleza y rapidez de las instrucciones de máquina empleadas para la ejecución del programa.

4. La complejidad de tiempo del algoritmo base del programa.

El hecho que el tiempo de ejecución dependa de la entrada, indica que el tiempo de ejecución debe definirse como una función de la entrada. Con frecuencia, el tiempo de ejecución no depende de la entrada, sino que depende propiamente del “tamaño” de la entrada.

Por ejemplo, en un algoritmo de ordenamiento se da como entrada una lista de elementos. Si la entrada son los elementos:

[2, 1, 3, 1, 5, 8],

Se desea producir como salida una lista ordenada de menor a mayor, de la forma:

[1, 1, 2, 3, 5, 8],

Evidentemente los elementos de la lista de salida se encuentran ordenados de menor a mayor. La medida natural de la entrada es el número de elementos para ordenar, en este caso 6 elementos. En otras palabras, el largo de la lista.

En general, entre mayor sea el número de elementos que se deben ordenar, mayor será el tiempo de ejecución del programa. Esta medida del número de elementos es una medida apropiada para determinar el tiempo de ejecución del programa.

Se acostumbra entonces, denominar la función T(n) al tiempo de ejecución de un programa, con una entrada de tamaño “n”. Por ejemplo, algunos programas pueden tener un tiempo de ejecución de la forma:

$T(n) = n^2 + 4*n + 3$

Las unidades de T(n) se dejan sin especificar, pero se pueden considerar como el número de instrucciones ejecutadas en un computador idealizado.

Se tomará la parte de mayor creciemiento de la sumas, y se dice que es:

$T(n) = n^2 + 4*n + 3$

$T(n)$ es $O(n^2)$

▬▬▬ 4. Tipos de Funciones.

A continuación se presenta una lista con las funciones más usadas para medir el tiempo de ejecución de una función. Se muestra en orden de menor a mayor.

$O(1)$	tiempo constante.
$O(\log n)$	tiempo logarítmico.
$O(\sqrt{n})$	tiempo raíz de n
$O(n)$	tiempo lineal
$O(n * \log n)$	tiempo casi lineal.
$O(n^2)$	tiempo cuadrado
$O(n^3)$	tiempo cúbico
$O(2^n)$	tiempo exponencial
$O(3^n)$	tiempo exponencial
$O(k^n)$	tiempo exponencial

Una mención aparte es esta función de tiempo, de la cual hablaremos después.

$O(n!)$ tiempo factorial

▬▬▬ 5. Notación Asintótica.

Para hacer referencia a la velocidad de crecimiento de los valores de una función se usará la notación conocida como “notación asintótica”. La cual utiliza fundamentalmente tres elementos que se mencionan a continuación.

▬▬▬ 6. Big Oh.

Por ejemplo, cuando el tiempo de ejecución de un programa se puede presentar como:

$T(n) = 5*n^2 + 4*n + 8$

$T(n)$ es $O(n^2)$

Y se lee, “el tiempo de ejecución de un programa es O mayúscula de n al cuadrado”.

Y significa que existen constantes positivas “c” y “n_0”, tales que::

Para $n \geq n_0$ se cumple que:

$T(n) \leq c*n^2$

●● Ejemplo.

Supongamos que:

$T(0) = 1$

$T(1) = 4$

y en general:

$T(n) = (n + 1)^2$ es $O(n^2)$

$T(n) = n^2 + 2*n + 1$ es $O(n^2)$

Puesto que:

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

$= (1*n^2 + 2*n + 1)$

$< 1*n^2 + 2*n^2 + 1*n^2$

$= 4*n^2$

es $O(n^2)$

Para $n \geq 1$ se tiene que:

$T(n) \leq 4*n^2$

$(n + 1)^2 \leq 4*n^2$

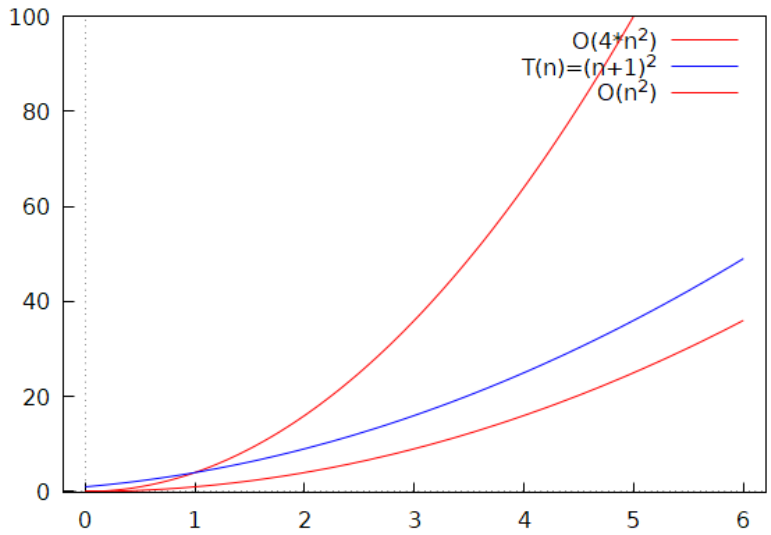
Veamos una tabla:

n	$T(n)=(n + 1)^2 \leq$	$4*n^2$
0	1	0
1	4	4
2	9	16
3	16	36
4	25	64
5	36	100
6	48	144
7	64	196
8	81	256
9	100	324
10	121	400

» Gráfico.

» Relación entre T(n) y O(n)

```
load(draw)$
f(n):=4*(n^2);
t(n):=(n+1)^2;
o(n):=n^2;


ini:0;
fin:6;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,100],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(4*n^2)",
   explicit(f(n), n,ini,fin),

   color=blue,
   key="T(n)=(n+1)^2",
   explicit( t(n) ,n,ini,fin),

   color=red,
   key="O(n^2)",
   explicit(o(n), n,ini,fin)

)$
```



●● Ejemplo.

Si el tiempo de ejecución está dado por:

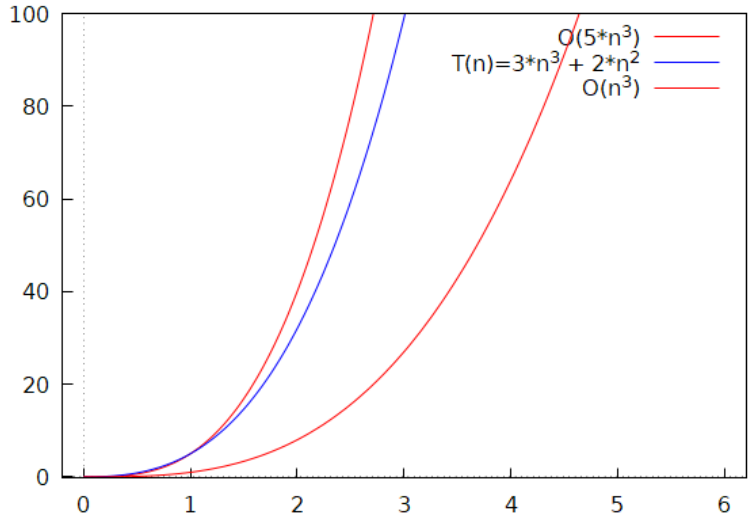
$T(n) = 3*n^3 + 2*n^2$ es $O(n^3)$

Para $n \geq 0$, se tiene que:

```
T(n) = 3*n³ + 2*n²

     < 3*n³ + 2*n³

     = 5*n³

     = O(n³)
```

n	T(n)= 3*n³ + 2*n²	≤ 5*n³
0	0	0
1	5	5
2	32	40
3	99	135
4	224	320
5	425	625
6	720	1080
7	1127	1715
8	1664	2560
9	2349	3645
10	3200	5000

```
» Gráfico.
» Relación entre T(n) y O(n)

load(draw)$
f(n):=5*(n^3);
t(n):=3*n^3 + 2*n^2;
o(n):=n^3;


ini:0;
fin:6;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,100],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(5*n^3)",
   explicit(f(n), n,ini,fin),

   color=blue,
   key="T(n)=3*n^3 + 2*n^2",
   explicit( t(n) ,n,ini,fin),

   color=red,
   key="O(n^3)",
   explicit(o(n), n,ini,fin)

)$
```



▬▬▬ 7. Omega.

Cuando se dice que T(n) es O(f(n)), se sabe que f(n) es una cota superior para la velocidad de crecimiento de T(n).

Para especificar una cota inferior para la velocidad de crecimiento de T(n) se usa la notación:

T(n) es Omega(g(n))

Que se lee “T(n) es omega de g(n)”.

Y lo que significa es que existe una constante “c” tal que:

Para $n \geq n_0$ se cumple que:

$T(n) \geq c*g(n)$

●● Ejemplo.

Sea

$T(n) = 3*n^3 + 2*n^2$ es Omega(n^3)

Puesto que:

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

$> 1*n^3$

= Omega(n^3)

Para n ≥ 0, se tiene que:

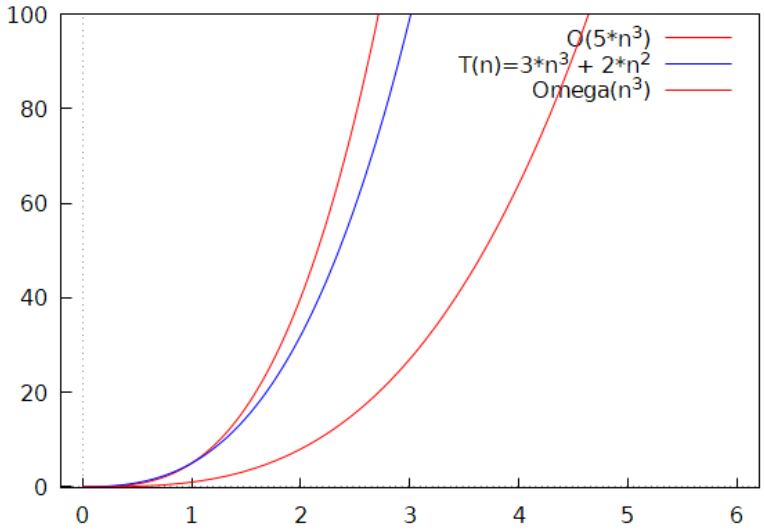
$T(n) \geq 1*n^3$

$3*n^3 + 2*n^2 \geq 1*n^3$

n	$T(n)= 3*n^3 + 2*n^2 \geq$	$1*n^3$
0	0	0
1	5	1
2	32	8
3	99	27
4	224	64
5	425	125
6	720	216
7	1127	343
8	1664	512
9	2349	729
10	3200	1000

» Gráfico.
» Relación entre T(n) y Omega(n)

```
load(draw)$
o(n):=5*(n^3);
t(n):=3*n^3 + 2*n^2;
omega(n):=n^3;

ini:0;
fin:6;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,100],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(5*n^3)",
   explicit(o(n), n,ini,fin),

   color=blue,
   key="T(n)=3*n^3 + 2*n^2",
   explicit( t(n) ,n,ini,fin),

   color=red,
   key="Omega(n^3)",
   explicit(omega(n), n,ini,fin)

)$
```



▬▬▬ 8. Theta.

La función f(n) = Theta(g(n)) si existen contantes c_1, c_2, n_0 tales que:

Para $n \geq n_0$ se tiene que:

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

●● Ejemplo.

Anteriormente habíamos probado que:

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

$\geq 1*n^3$

$= Omega(n^3)$

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

$< 3*n^3 + 2*n^3$

$= 5*n^3$

$= O(n^3)$

Entonces:

$1*n^3 \leq 3*n^3 + 2*n^2 \leq 5*n^3$

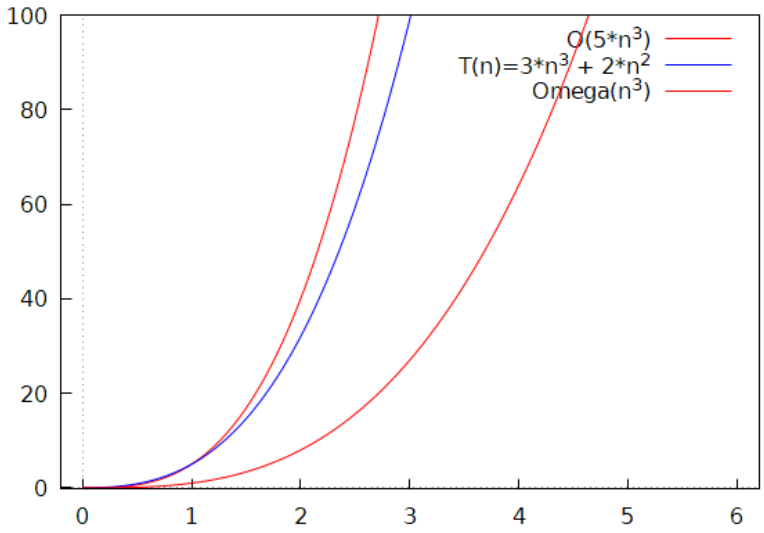
$Omega(n^3) \leq T(n) \leq O(n^3)$

Por lo tanto:

T(n) es Theta(n^3)

» Gráfico.
» Relación entre T(n) y 0(n^3)

```
load(draw)$
o(n):=5*(n^3);
t(n):=3*n^3 + 2*n^2;
omega(n):=n^3;


ini:0;
fin:6;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,100],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(5*n^3)",
   explicit(o(n), n,ini,fin),

   color=blue,
   key="T(n)=3*n^3 + 2*n^2",
   explicit( t(n) ,n,ini,fin),

   color=red,
   key="Omega(n^3)",
   explicit(omega(n), n,ini,fin)

)$
```



▬▬▬ 9. Es Posible Encontrar Theta en Todos los Casos?.

En algunas ocasiones, se puede encontrar el valor de Theta, en otras no. Veamos los siguientes ejemplos.

●● Ejemplo.

Sea $T(n) = 2*n^2 + 3*n + 4$.

Encontremos los valores de O, Omega y Theta.

Tenemos que:

$1*n^2 \leq 2*n^2 + 3*n + 4 \leq 9*n^2$

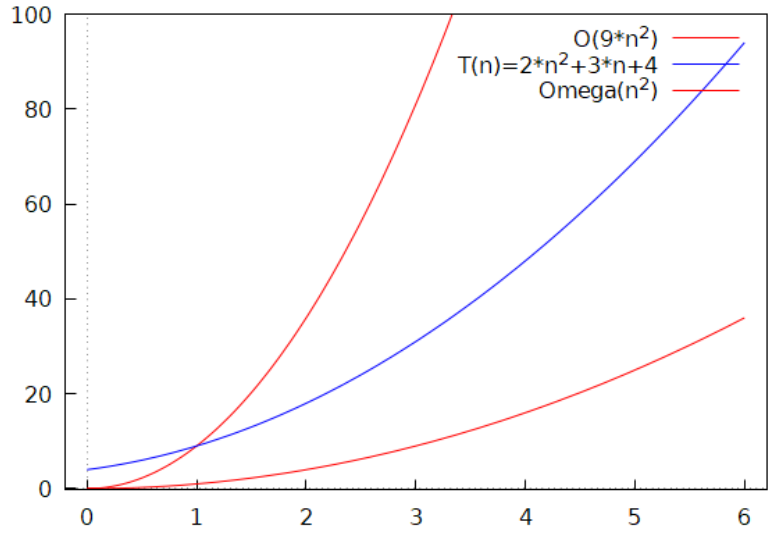
$\Omega(n^2) \leq T(n) \leq O(n^2)$

Por lo tanto:

$T(n)$ es $\Theta(n^2)$

» Gráfico.
» Ejemplo de Omega

```
load(draw)$
oh(n):=9*n^2;
t(n):=2*n^2+3*n+4;
omega(n):=n^2;


ini:0;
fin:6;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,100],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(9*n^2)",
   explicit(oh(n), n,ini,fin),

   color=blue,
   key="T(n)=2*n^2+3*n+4",
   explicit( t(n) ,n,ini,fin),

   color=red,
   key="Omega(n^2)",
   explicit(omega(n), n,ini,fin)

)$
```



●● Ejemplo.

Sea T(n) = n² * log(n) + n;

Se tiene que:

1*n²*log(n) ≤ n²*log(n) + n ≤ 2*n²*log(n)

Theta(n²*log(n)) ≤ T(n) ≤ O(n²*log(n))

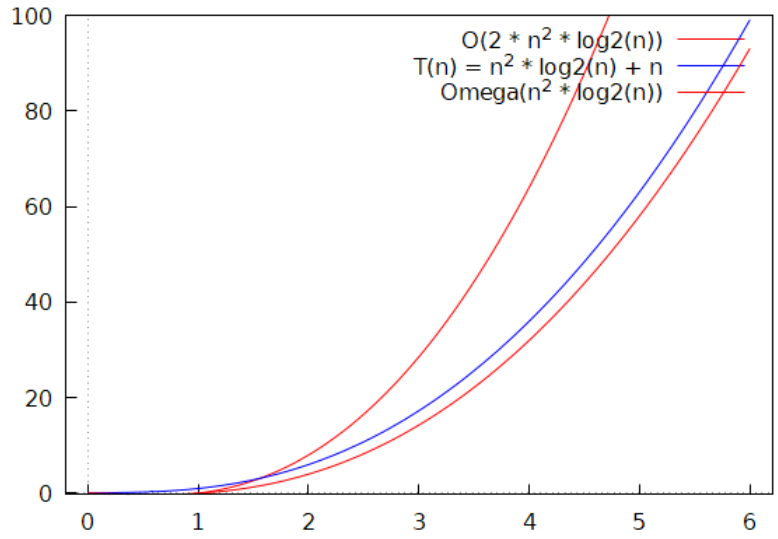
Por lo tanto:

T(n) es Theta(n²*log(n))

» Gráfico.
» Ejemplo de Omega

```
load(draw)$
log2(x):=log(x)/log(2);
oh(n):=2 * n^2 * log2(n);
t(n):=n^2 * log2(n) + n;
omega(n):=n^2 * log2(n);

ini:0;
fin:6;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,100],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(2 * n^2 * log2(n))",
   explicit(oh(n), n,ini,fin),
```

```
   color=blue,
   key="T(n) = n^2 * log2(n) + n",
   explicit( t(n) ,n,ini,fin),

   color=red,
   key="Omega(n^2 * log2(n))",
   explicit(omega(n), n,ini,fin)

)$
```



●● Ejemplo.

Sea:

T(n) = n!

T(n) = n*(n-1)*(n-2)*...*1

En este caso tenemos:

1*1*1*...*1 ≤ n*(n-1)*(n-2)*...*1 ≤ n*n*n*...*n

1 ≤ n! ≤ n^n

Luego:

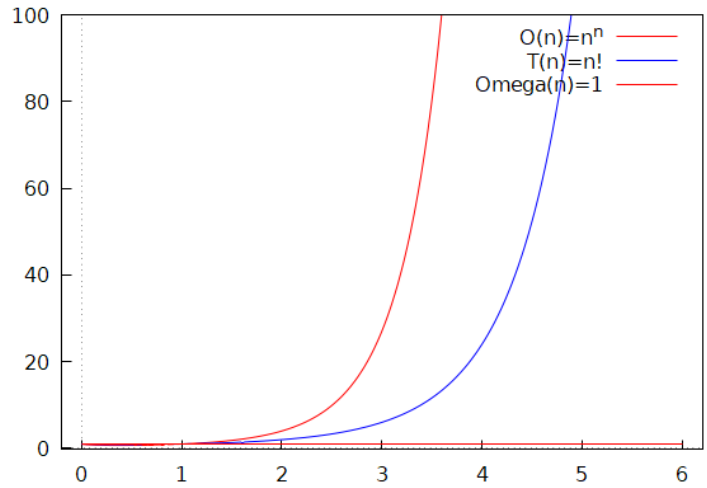
Omega(1) ≤ n! ≤ O(n^n)

Y por lo tanto no existe el valor de Theta!!!

No siempre es posible encontrar el valor de Theta. De hecho en muchos casos vamos a tener que calcular únicamente el valor de la Big Oh.

>>> Gráfico.
>>> Ejemplo: No existe Theta para T(n) = n!

```
load(draw)$
log2(x):=log(x)/log(2);

oh(n):=n^n;
t(n):=n!;
omega(n):=1;

ini:0;
fin:6;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,100],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(n)=n^n",
   explicit(oh(n), n,ini,fin),

   color=blue,
   key="T(n)=n!",
   explicit( t(n) ,n,ini,fin),

   color=red,
   key="Omega(n)=1",
   explicit(omega(n), n,ini,fin)

)$
```



●● Ejemplo.

Sea

T(n) = log(n!)

Entonces:

log(1) ≤ log(n!) ≤ log(nⁿ)

log(1) ≤ log(n!) ≤ n*log(n)

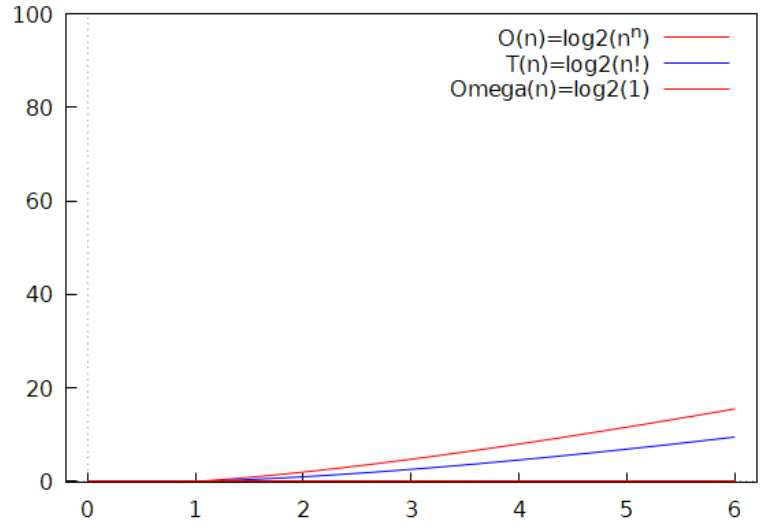
Por lo tanto:

Omega(log(1)) ≤ log(n!) ≤ O(n*log(n))

Y no existe el valor de Theta!!!

>>> Gráfico.
>>> Ejemplo cuando no existe Theta.

```
load(draw)$
log2(x):=log(x)/log(2);

oh(n):=log2(n^n);
t(n):=log2(n!);
omega(n):=log2(1);

ini:0;
fin:6;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,100],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=red,
   key="O(n)=log2(n^n)",
   explicit(oh(n), n,ini,fin),

   color=blue,
   key="T(n)=log2(n!)",
   explicit( t(n) ,n,ini,fin),

   color=red,
   key="Omega(n)=log2(1)",
   explicit(omega(n), n,ini,fin)

)$
```



▬▬▬ 10. La Tiranía de la Velocidad.

Aquí presentamos algunos tiempos de la Big Oh que a menudo se mencionan. Se encuentran ordenados del más rápido al más lento.

• O(1).

Tiempo constante

• $O(\log_2(n))$, $O(\log_2 n)$, O(log n).

Tiempo logarítmico.
Siempre que se indique la función logaritmo, supondremos que tiene base 2.

• O(n)

Tiempo lineal.

• $O(n * \log_2 n)$, O(n*log n).

Tiempo “n” logaritmo de “n”.

• $O(n^2)$

Tiempo cuadrático

• $O(2^n)$, $O(3^n)$, $O(k^n)$

Tiempo exponencial.

• O(n!)
Tiempo factorial
Un algorimo verdaderamente lento!

Vamos a suponer que tenemos un computador lento que hace 1000000 instrucciones por segundo. Un millón de instrucciones por segundo.

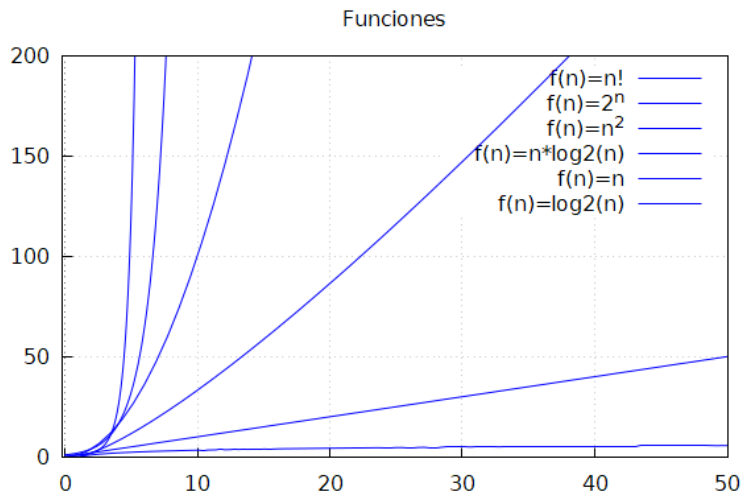
Observe la siguiente tabla de valores, los tiempos se han redondeado y se indican en segundos (s), minutos(m), horas (h), días(d), años(a) y siglos(sig).

BigOh	n=1,000	n=100,000	n=1,000,000
O(log n)	0.00001s	0.000017s	0.00002s
O(n)	0.001s	0.1s	1s
O(n*log n)	0.01s	1.7s	20s
$O(n^2)$	1s	3h	12d
$O(n^3)$	17m	32años	317siglos
$O(2^n)$	10^{285}siglos	10^{10000}siglos	10^{100000}siglos
O(n!)	Float(2508)	Float(456,214)	Float(???)

» Gráfico.
» Diferentes Funciones Tiempos de Ejecución.

```
load(draw)$
f_logn(n):=log(n)/log(2);
f_n(n):=n;
f_nlogn(n):= n*(log(n)/log(2));
f_np2(n):=n^2;
f_2pn(n):=2^n;
f_nf(n):=n!;

ini:0$
fin:100$
wxdraw2d
(
    xrange = [-0.20,50],
    yrange = [-0.20,200],
    grid=true,
    xaxis=true,
    yaxis=true,

    title="Funciones",
    color=blue,
    key="f(n)=n!",
    explicit( f_nf(n)   ,n,ini,fin),
    key="f(n)=2^n",
    explicit( f_2pn(n)  ,n,ini,fin),
    key="f(n)=n^2",
    explicit( f_np2(n)  ,n,ini,fin),
    key="f(n)=n*log2(n)",
```

```
    explicit( f_nlogn(n),n,ini,fin),
    key="f(n)=n",
    explicit( f_n(n)    ,n,ini,fin),
    key="f(n)=log2(n)",
    explicit( f_logn(n) ,n,ini,fin)
)$
```



Existen otros tiempos de algoritmos, pero estos son los más comunes. Además esto es una simplificación del proceso, pues no es siempre posible pasar de la notación de la Big Oh a los tiempos que tarda una computadora de una forma tan sencilla.

Hasta este punto, es importante resaltar lo siguiente:

• La velocidad de un algoritmo no se mide en segundos, sino en el crecimiento del número de operaciones.

• En general hablamos de que tan rápido crece el tiempo de ejecución del algoritmo, en razón del incremento del tiempo de ejecución de sus entradas.

• El tiempo de ejecución de los algoritmos se expresa mediante la notación de la Big Oh.

▬▬▬ 11. El Problema del TSP.

A menudo se tiene la pregunta: como son los problemas donde O(n!). Así que les mostramos un problema cuyo tiempo de corrida es muy lento. Este problema se conoce como “el vendedor viajero” o “la vendedora viajera”, o en inglés “traveling salesperson” (TSP).

Supongamos que tenemos 4 ciudades y una tabla con las distancias entre cada ciudad. La tabla se muestra a continuación. En esta tabla se indica la distancia en km entre cada ciudad. Vamos a suponer simetría, es decir que ir de C(i) a C(j) tiene el mismo valor de ida que de vuelta.

El vendedor o vendedora, tiene que viajar por las cinco ciudades, y luego regresar a la ciudad donde vive. Supondremos que vive en la ciudad C1. No puede pasar dos

veces por una misma ciudad, ni quedarse en una misma ciudad. Se desea hacer el recorrido viajando el menor número de kilómetros posible.

Distancia entre las ciudades.

	C1	C2	C3	C4
C1	M	132	217	164
C2	132	M	290	201
C3	217	290	M	113
C4	164	201	113	M

Por ejemplo el recorrido de:

C1 → C2 → C3 → C4 → C1

Tiene un costo de:

132 + 290 + 113 + 164 = 699 km

Pero otro recorrido como:

C1 → C3 → C2 → C4 → C1

Tiene un costo de:

217 + 290 + 201 + 164 = 872 km

Con 4 ciudades, se tiene que revisar 6 rutas, pues el número de caminos está dado por:

(4-1)! = 3! = 3 * 2 * 1 = 6 rutas.

Estas son las seis posibles rutas que existen para 4 ciudades:
[1,2,3,4,1]
[1,2,4,3,1]
[1,3,2,4,1]
[1,3,4,2,1]
[1,4,2,3,1]
[1,4,3,2,1]

En general, se puede demostrar que para “n” ciudades existen (n-1) posibles rutas. Se debe calcular la distancia para cada una de estas rutas y seleccionar la ruta con el menor número de kilómetros. En la siguiente tabla se muestra el número de rutas que existen, y por lo tanto de operaciones, que se deben evaluar conforme crece el número de ciudades.

Ciudades n Rutas n! Rutas

5	(5-1)!	24
10	(10-1)!	362,880
15	(15-1)!	87,178,291,200
30	(30-1)!	8841,761,993,739,701,954,543,616,000,000

Se puede ver que en general para “n” ciudades se deben revisar (n-1)! rutas. De esta forma es un problema O(n!), o bien un problema con tiempo factorial. Desafortunadamente no hay otra forma de resolver este algoritmo. Es usual entonces, usar otros algoritmo más rápidos que produzcan soluciones aproximadas a la solución correcta, pero este es otro tema.

▬▬▬ 12. Resumen.

• Es importante medir el tiempo de ejecución de un programa.

• Usualmente estamos interesados en el “peor caso” de ejecución del programa.

• Para medir el tiempo de ejecución se utiliza la notación asintótica.

• La velocidad de los algoritmos no se mide en segundos, ni en ninguna otra unidad de tiempo.

• La velocidad de los algoritmos se mide en términos del crecimiento del número de operaciones del algoritmo.

• La velocidad se escribe usando una notación llamada la gran O.

• Hay una relación entre la gran O y el tiempo que tarda un algoritmo.

▬▬▬ 13. Ejercicios.

●● Ejercicio 1.

Se tienen las funciones:

$f(n) := 2*n+1;$

$g(n) := n;$

Para un valor $n \geq 1$, encuentre el valor entero de “c” más pequeño para que se cumpla:

$f(n) \leq c*g(n)$

$2*n + 1 \leq c*n$

Puede hacer esto de dos formas:

• Pruebe con diferentes valores de “c” y hace una tabla para ver que se cumple la condición.

• Encuentre el valor de “c” por medios algebraicos.

●● Ejercicio 2.

Se tienen las funciones:

f(n) := 2*n;

g(n) := n^2;

Para un valor n ≥ 1, encuentre el valor entero de “c” más pequeño para que se cumpla:

f(n) ≤ c*g(n)

2*n ≤ c*n^2

• Pruebe con diferentes valores de “c” y hace una tabla para ver que se cumple la condición.

• Encuentre el valor de “c” por medios algebraicos.

●● Ejercicio 3.

Tenemos las siguientes funciones de f(n) y g(n).

Determine si

f(n) = O(g(n)) o g(n) = O(f(n))

Utilice:

$f(n) = (n^2 - n)/2$

g(n) = 6*n

●● Ejercicio 4.

Tenemos las siguientes funciones de f(n) y g(n).

Determine si

f(n) = O(g(n)) o g(n) = O(f(n))

Utilice:

$f(n) = n + 2*\sqrt{(n)}$

$g(n) = n^2$

●● Ejercicio 5.

Tenemos las siguientes funciones de f(n) y g(n).

Determine si

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

Utilice:

$f(n) = n + n*\log(n)$

$g(n) = n*\sqrt{(n)}$

●● Ejercicio 6.

Tenemos las siguientes funciones de f(n) y g(n).

Determine si

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

Utilice:

$f(n) = n^3 + 3*n + 4$

$g(n) = n^3$

●● Ejercicio 7.

Tenemos las siguientes funciones de f(n) y g(n).

Determine si

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

Utilice:

$f(n) = n * \log(n)$

$g(n) = (n * \sqrt{(n)})/2$

●● Ejercicio 8.

Tenemos las siguientes funciones de f(n) y g(n).

Determine si

f(n) = O(g(n)) o g(n) = O(f(n))

Utilice:

f(n) = n + log(n)

g(n) = √(n)

●● Ejercicio 9.

Tenemos las siguientes funciones de f(n) y g(n).

Determine si

f(n) = O(g(n)) o g(n) = O(f(n))

Utilice:

f(n) = 2*(log(n))²

g(n) = log(n) + 1

●● Ejercicio 10.

Tenemos las siguientes funciones de f(n) y g(n).

Determine si

f(n) = O(g(n)) o g(n) = O(f(n))

Utilice:

f(n) = 4*n*log(n) + n

g(n) = (n² - n)/2

●● Ejercicio 11.

Ponga las siguientes funciones en orden de de menor a mayor. Si dos funciones son del mismo orden, debe poner un circulo para unirlas.

O(2ⁿ)

O(log n)

$O(n^2)$

$O((\log n)^2)$

O(n!)

●● Ejercicio 12.

Ponga las siguientes funciones en orden de de menor a mayor. Si dos funciones son del mismo orden, debe poner un circulo para unirlas.

O(log(log(n)))

$O(5*n^3 - n^2 + n)$

$O(n^3)$

$O(\sqrt{n})$

O(n)

●● Ejercicio 13.

Ponga las siguientes funciones en orden de de menor a mayor. Si dos funciones son del mismo orden, debe poner un circulo para unirlas.

$O(n^3 + \log(n))$

$O(2^{(n-1)})$

O(n * log(n))

O(6)

$O((3/2)^n)$