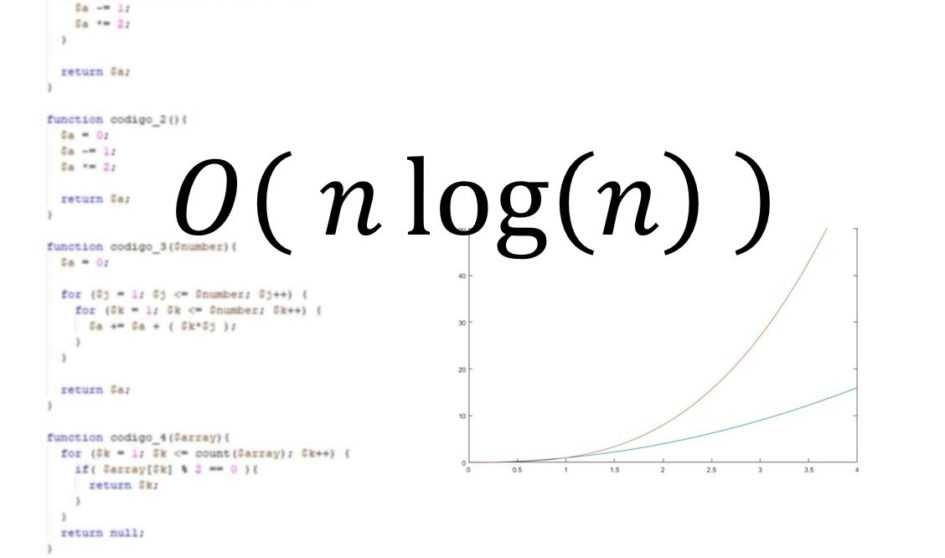
**[Complejidad Algorítmica – Ejemplos Python, PHP](http://micaminomaster.com.co/grafo-algoritmo/complejidad-algoritmica-ejemplos-python-php/" \o "Complejidad Algorítmica – Ejemplos Python, PHP)**



Es en realidad una métrica teórica que mide la eficiencia computacional de un algoritmo. Si tenemos varios algoritmos que solucionan un mismo problema, entonces esta métrica nos ayudará a definir cuál de los algoritmos es mejor en términos computacionales.

Aunque no es un concepto difìcil de entender, la obtención y el estudio de la complejidad requieren ciertas destrezas matemáticas. En esta entrada voy a exponerles de manera sencilla sobre la complejidad algorítmica, los tipos de complejidad y ejemplos en código Python y Php.

**¿Qué es?**

Primero que todo hablemos de Algoritmo. Un algoritmo es una secuencia de instrucciones cuyo objetivo es la resolución de un **problema**. Existen muchos problemas que tiene algoritmos que los solucionen. Además un mismo problema puede tener varias soluciones (¿o no tener ninguna?). Si tenemos varias soluciones de un mismo problema, ¿Cuál sería la mejor solución? Para eso necesitamos evaluar dicho algoritmo, el resultado de la evaluación lo llamaremos **complejidad algorítmica**.

Para medir un algoritmo bien podemos tratarlo desde dos puntos de vista. Bien sea quien resuelva el problema en menos tiempo o quien utilize menos memoria del computador. A la idea del tiempo la llamaremos **complejidad temporal** y el punto de vista de recursos en memoría la llamaremos **complejidad espacial**.

De los dos tipos mencionados, la de menos relevancia es la de complejidad espacial. Debido a los altos recursos que puede tener una máquina. Además el tiempo es mucho más valioso. Podemos decir que de los dos, el tiempo es el único que no se puede comprar. Así que simplemente cuando nos referimos a complejidad algorítmica, nos estamos refiriendo a la complejidad temporal.

Básicamente la medición del algoritmo es cuánto tarda en resolver un problema.

**¿El tamaño importa?**

Hablemos de algunos conceptos que poseen todos los algorítmos. Durante su proceso, digamos un algoritmo de ordenamiento, posee una serie de instrucciones que se repiten, se le conocen como bucles. También una serie de elecciones o comparaciones, (if.. else..) que hacen que siga ciertas instrucciones o no siga otras. Todos estos elementos hacen parte del algoritmo . Poseen, también, datos de entrada y datos de salida.

De todas estas características las que podemos variar para medir nuestra complejidad serían los datos de entrada. Ya que no es lo mismo, en un algoritmo de ordenamiento, ordenar 10 elementos que ordenar 1’000.000 de elementos. El algoritmo debería poder ordenar los elementos sin importar el tamaño del vector, pero si incide directamente en el tiempo en resolver el problema.

Con esto podemos empezar a medir los distintos algoritmos de ordenamiento. Si ordenamos un vector de diez millones y vemos cuál se demora menos, podríamos decir cuál tiene mayor o menor complejidad algorítmica.

**La complejidad algorítmica no es un número. Es una función**

Ahora, consideremos que hacemos experimentos del algoritmo con 10’000.000 de entradas. Si lo realizamos en un computador muy potente la respuesta del programa sería obviamente mucho más rápido que en un computador con pocos recursos. Así que el algoritmo va a tener siempre un tiempo de respuesta diferente para cada computador.

Para solucionarlo no mediremos el tiempo que se tarda en responder un algoritmo. Más bien vamos a contar el número de instrucciones, suponiendo que cada instrucción se ejecuta en un mismo tiempo constante. De esta manera mediremos cuántas instrucciones necesarias se toma el algoritmo para resolver el problema con respecto al tamaño del problema.

Analizemos el siguiente código escrito tanto en PHP como Python:

Código 1

PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function codigo\_1($number){      $a = 0;    for ($j = 1; $j <= $number; $j++) {      $a += $a + $j;    }      for ($k = $number; $k >= 1; $k--) {      $a -= 1;      $a \*= 2;    }      return $a;  } |

Código 1

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def codigo\_1( number ):    a = 0    for j in range(1, number+1):      a += a + j      for k in range(number, 0, -1):      a -= 1      a \*= 2    return a |

En este algoritmo tenemos unas instrucciones y varios bucles. Empezemos con contar las instrucciones.

Tenemos una asignación de la variable *a*, así que tenemos 1 instrucción.

Instrucciones = 1

El siguiente es un bucle con una instrucción adentro. Dependiento del valor de la variable *$number,* se realiza una instruccion *n* veces. Por ejemplo si *$number*tiene el valor de 3 entonces las instrucciones que se realizan son 3 veces. Entonces tenemos:

Instrucciones = (1) + (n)

Instrucciones = n + 1

En el segundo bucle sucede lo mismo. Pero esta vez son 2 instrucciones dentro del ciclo de código. El número de instrucciones quedaría:

Instrucciones = 2n + n + 1

Instrucciones = 3n + 1

Entonces la complejidad algorítmica sería 3n + 1 porque es el número de instrucciones que tiene que realizar para solucionar el problema.

Lo que interesa es saber cómo puede crecer en unidades de tiempo la resolución de un problema. En este ejemplo es claro que el tiempo crece linealmente con respecto al valor de entrada.

**Código y Complejidad**

Es posible calcular visualmente la complejidad de algunos algoritmos sencillos. Veamos algunos ejemplos:

Código 2

PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function codigo\_2(){    $a = 0;    $a -= 1;    $a \*= 2;      return $a;  } |

Código 2

Python

|  |  |
| --- | --- |
| 1  2  3  4 | def codigo\_2():    a = 0    a -= 1    a \*= 2 |

En el Código 2 la Complejidad sería:

F(x) = 3

Código 3

PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function codigo\_3($number){    $a = 0;      for ($j = 1; $j <= $number; $j++) {      for ($k = 1; $k <= $number; $k++) {        $a += $a + ( $k\*$j );      }    }      return $a;  } |

Código 3

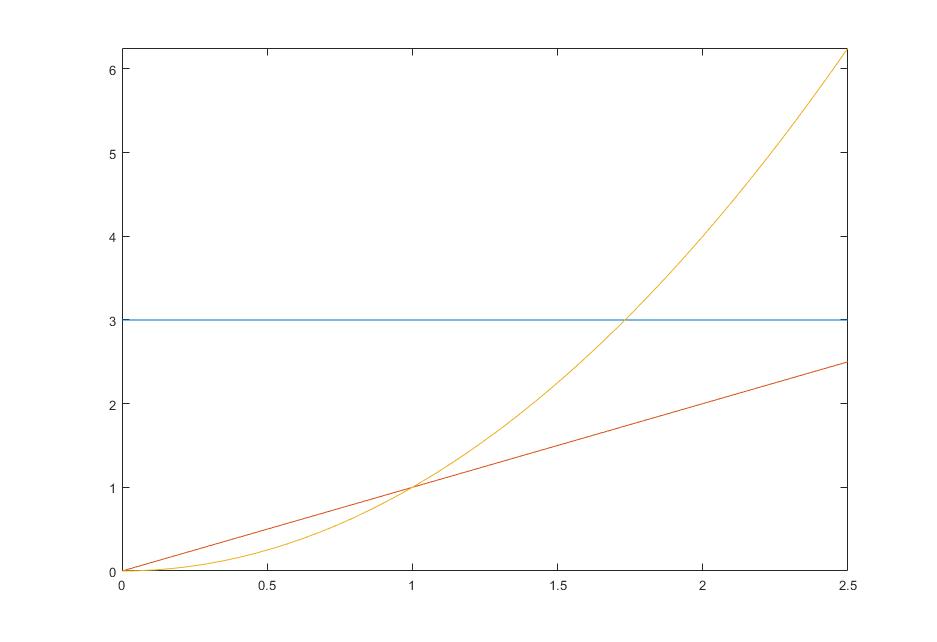
Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def codigo\_3( number ):    a = 0;      for j in range(1, number+1):      for k in range(1, number+1):        a += a + ( k\*j )      return a |

En el código 3 tenemos un bucle anidado. Cada vez que ejecute un ciclo el otro se ejecuta *n* veces también. Lo cual sería *n* veces *n*. La complejidad quedaría:

F(x) = n^2 + 1

Si graficamos estas 3 funciones, el código 1, 2 y 3. Podemos ver exactamente cual sería el que posee menor complejidad algorítmica:



**Figura 1. Comparacón de distintas complejidades algorítmicas**

Esto nos deja que independientemente de los valores, resulta evidente que entre más datos, la curva se va haciendo cada vez más grande. Esto es lo que realmente nos interesa ver en una complejidad algorítmica, cómo se comporta el algoritmo con volúmenes de entradas grandes en el tiempo.

**Nada de suerte. Escenario de peor caso**

Algunas algoritmos pueden tener distintos tiempos de ejecución, teniendo el mismo tamaño de los datos y los mismos recursos computacionales. Esto es debido a que dependiendo de los datos a veces se llegue a una solución en la primera iteración, o bien tener que recorrer todos los datos.

El siguiente código tiene como fin encontrar el primer número par de una lista de números:

Código 4

PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function codigo\_4($array){    for ($k = 1; $k <= count($array); $k++) {      if( $array[$k] % 2 == 0 )        return $k;      }    }    return null;  } |

Código 4

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def codigo\_4(array):    for k in range(len(array)):      if( array[k] % 2 == 0 ):        return k      return null |

Dependiendo de los valores que se guarden en el vector, así será el tiempo que dure el algoritmo en resolver el problema. Si el array es una secuencia de números enteros, entonces solo hará falta una iteración. Si es una lista de números impares, entonces recorrerá todas las iteraciones.

En el mejor de los escenarios el número par será el primero de la lista, lo que concluiría el algoritmo. En el peor caso ni siquiera tenga un número par, por que recorrería todos las instrucciones.

Para el mejor caso tendríamos una complejidad algorítmica:

F(x) = 1

Para el peor de los casos la comjidad algorítmica sería:

F(x) = n

Para expresar el peor caso usaremos una notación conocida como “**O Grande**” y se escribe:

O(n)

Que significa complejidad en el peor caso. Se escribe como “O” pero en realidad es la letra griega Omicron.

**Órden de Complejidad**

Hasta aquí, es posible medir cualquier algoritmo. Pero incluso así es complicado comparar unos algoritmos con otros. Además se requiere poder categorizar que tan complejo es el algoritmo de otros. Para esto recurriremos al órden de complejidad.

Como vimos atrás. Para medir un algoritmo necesitamos recurrir al escenario de peor caso y con datos demasiado grandes, cuando me refiero a datos demasiado grandes, quiero decir ¡infinitamente grande! Viendolo de esta manera las ecuaciones de O Grande podriamos simplificarlas, eliminando algunas características que en datos de entrada titánicas son irrelevantes.

Si comparamos varios algoritmos lineas tales como:

O(3n + 1)

O(20n )

O(15n + 150)

O(n)

Podemos ver en sus gráficas que sin importar la variable que corta al eje Y o la pendiente de la curva, todas crecen con la misma inclinación. Todas estas tipo de ecuaciones podemos agruparlas y refenirnos a ellas como **Órden Lineal**.

Lo mismo podemos hablar de las funciones cuadráticas, veamos el siguiente grupo:

O(n^2 + 27)

O(20n^2 + 3 )

O(n^2 + n + 45)

O(100n^2 + 101n)

A pesar de que sus correspondientes gráficas tenga inicios y posiciones diferentes. Todos son una parábola y tendrán un comportamiento similar con respecto al aumento de la cantidad de entradas. Estas presentan un comportamiento muy diferente al conjunto anterior. Este grupo de O grandes las llamaremos de **Órden Cuadrática**. También las podemos expresar como:

O(100n^2 + 101n) \in O(n^2)

De esta manera podemos clasificar a un algoritmo en un grupo concreto y resultará mucho más fácil y rápido ver la complejidad de un algoritmo. Cuando nos advierten de una complejidad algorítmica de un algorítmo podemos tener una idea de cómo será su comportamiento.

Si por ejemplo tenemos un algoritmo de orden O(100n^4 + 3n^3 + 53), entonces podemos definir su orden de cualquiera de las siguientes formas:

O(100n^4 + 3n^3 + 53) \subset O(100n^4 + 3n^3) \subset O(100n^4) \subset O(n^4)

Hasta aquí concluye la definición de la complejidad algorítmica.

**Órden de Complejidad más conocidas**

**Constante: O(1)**

Es la más sencilla y siempre presenta un tiempo de ejecución constante. Ejemplo:

Constante

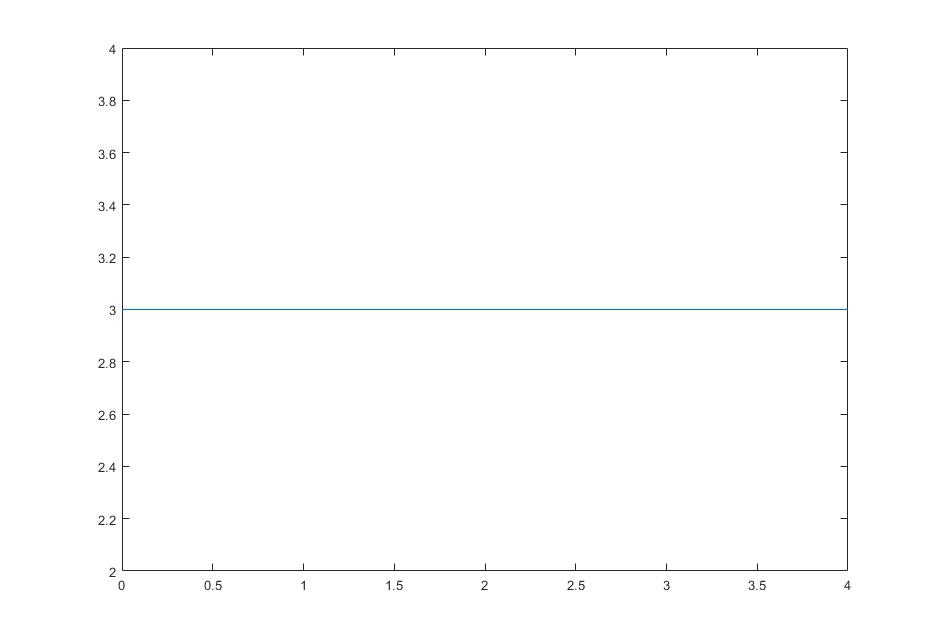
PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function constante(){    $x = 50;    $x++;      return $x;  } |

Constante

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | def constante():    x = 50    ++x      return x |



**Figura 2. Órden de comlpejidad – Constante**

**Lineal: O(x)**

El tiempo crece linealmente mientras crece los datos. Ejemplo:

Lineal

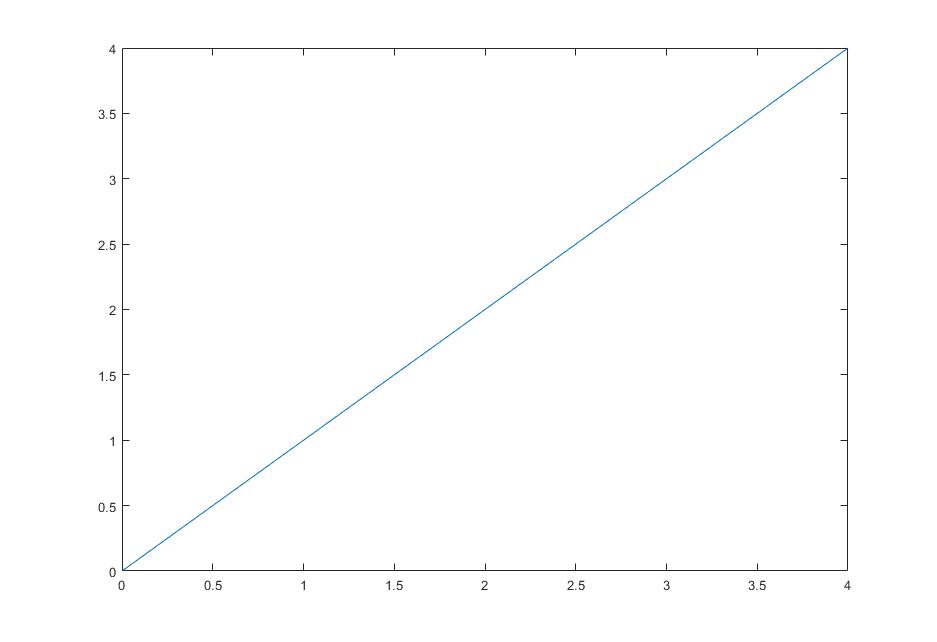
PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function lineal($number){    $result = 0;    for ($x = 0; $x <= $number; $x++) {      $result++;    }      return $result;  } |

Lineal

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def lineal(number):    result = 0    for x in range(0, number):      ++result      return result |



**Figura 3. Órden de Complejidad – Lineal**

**Polinómico: O(x^c), c > 0**

Son los algoritmos más comunes. . Cuando c es 2 se le llama **cuadrático**, cuando es 3 se le llama **cúbico**, y en general, polinómico. Cuando *n* es muy grande suelen ser muy complicados. Estos algoritmos suelen tener bucles anidados. Si tienen 2 bucles anidados sería un cuadrático. Ejemplos:

Polinómico

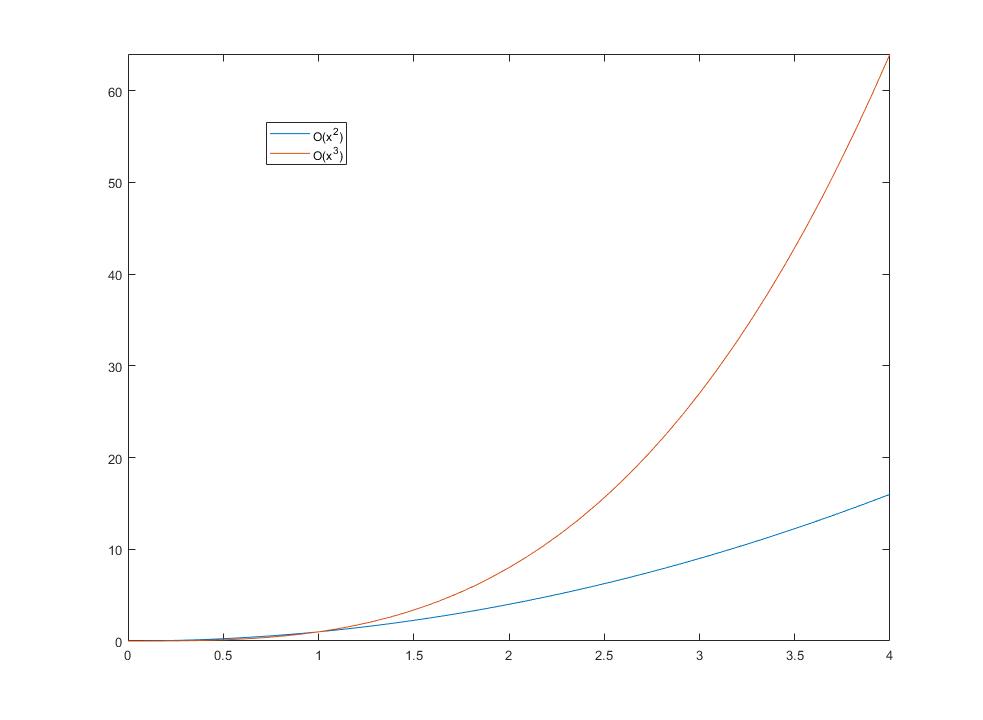
PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | funtion polinomico($number){    $x = 1;    for ($i=0; $i < $number; $i++) {      for ($j=0; $j < ; $j++) {        $x += $j + $i;      }    }      for ($i=0; $i < $number; $i++) {      for ($j=0; $j < ; $j++) {        for ($k=0; $k < ; $k++) {          $x += $j \* $i \* $k;        }      }    }      return $x;  } |

Polinómico

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def polinomico(number):    x = 0    for i in xrange(1,number):      for j in xrange(1,number):        x += i + j      for i in xrange(1,number):      for j in xrange(1,number):        for k in xrange(1,number):          x += i \* j \* k      return x |



**Figura 4. Comportamiento de örden Polinómico**

**Logarítmico: O(\log x)**

No suelen ser muchos. Estos algoritmos indican que el tiempo es menor que el tamaño de los datos de entrada. No importa indicar la base del logaritmo. Un ejemplo es una**búsqueda dicotómica**. Este algoritmo busca un elemento en un array ordenado dividiendo el array en 2 mitades, identifica en cual de las mitades se encuentra, luego divide esa parte en 2 mitades iguales y busca nuevamente hasta encontrar el elemento, es un algoritmo recursivo:

Algoritmo dicotómica - logarítmica

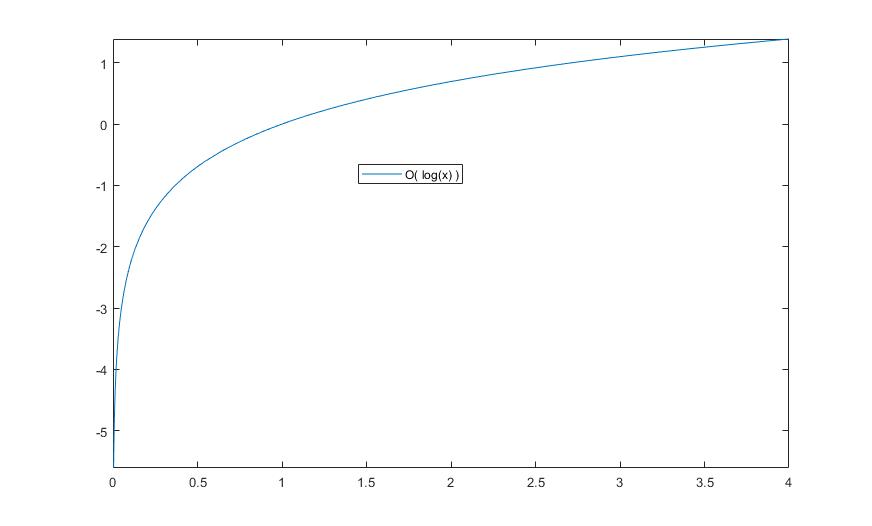
PHP

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function bin($a,$x,$low,$high){    $ans = -1;    if( $low==$high) $ans = -1;    else{      $mid = ( $low + floor( ($high-$low)/2) );      if (x < a[mid]){ $ans = bin($a,$x,$low,$mid); }      else if( x > a[mid] ){ $ans = bin($a,$x,$mid+1,$high); }      else $ans = $mid;    }    return $ans;  } |

Algoritmo dicotómica - logarítmica

Python

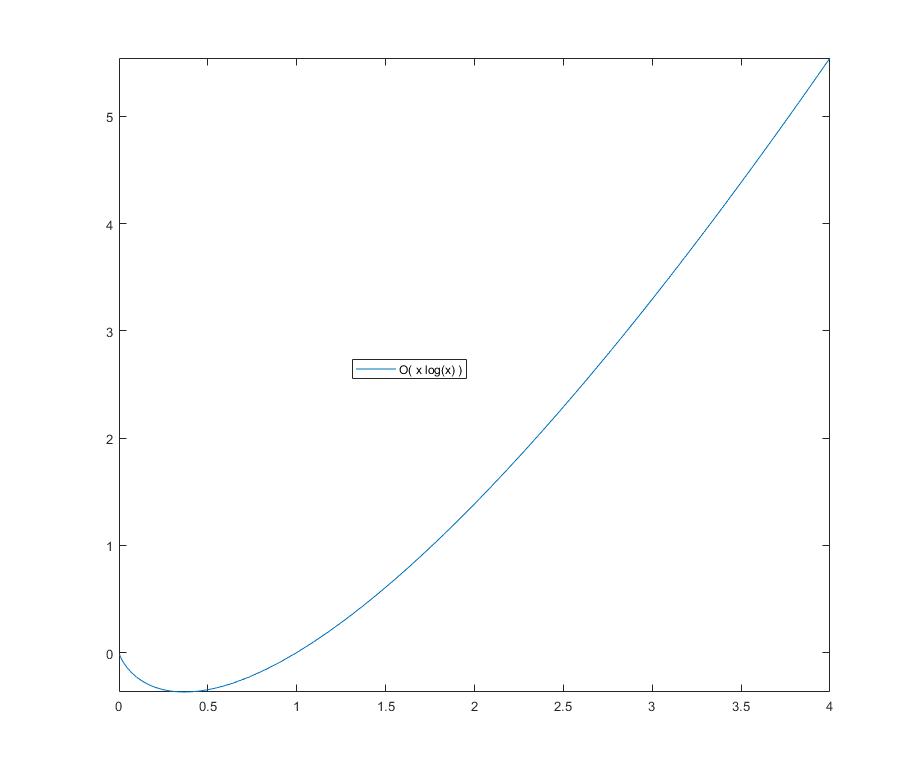
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def bin(a,x,low,high):    ans = -1    if low==high: ans = -1    else:      mid = (low+((high-low)//2))      if x < a[mid]: ans = bin(a,x,low,mid)      elif x > a[mid]: ans = bin(a,x,mid+1,high)      else: ans = mid    return ans |



**Figura 5. Comportamiento de órden logarítmica**

**Enelogarítmico: O(n \log x)**

Tan bueno como el anterior, en este orden encontramos el algoritmo QuickSort. El ejemplo podemos verlo en Wikipedia en este enlace <https://en.wikipedia.org/wiki/Quicksort>. No incluyo el código debido a las distintas versiones, estudios y discusiones de este algoritmo. Más se merece una entrada, pero mientras les comparto el comportamiento de este órden de complejidad:



**Figura 6. EComportamiento de órden enelogarítmica**

**Exponencial: O(c^x)**

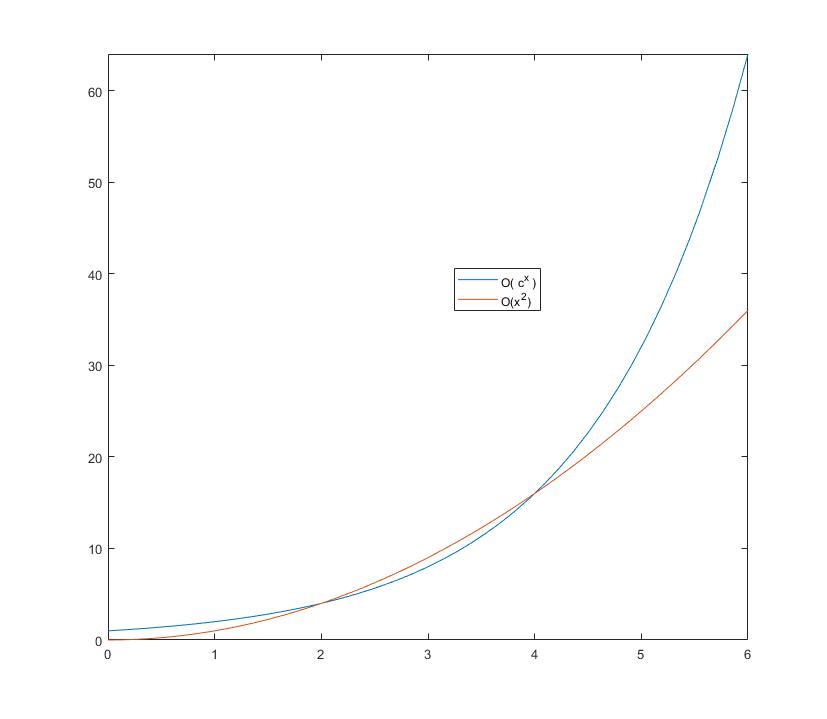
Es uno de los peores complejidades algorítmicas. Sube demasiado a medida que crece los datos de entrada. Puede verse en la Figura 7 como crece una función de este tipo. Un ejemplo de este código es la solución de Fibonacci, el cual genera bucles 2 recursividades en cada ejecución. Ejemplo:

PHP

|  |  |
| --- | --- |
| 1  2  3  4 | function exponencial($n){    if (n==1 || n==2) return 1;    return exponencial($n-1)+exponencial($n-2);  } |

PHP

|  |  |
| --- | --- |
| 1  2  3  4 | def exponencial(n):    if n==1 or n==2:      return 1    return exponencial(n-1)+exponencial(n-2) |



**Figura 7. Comportamiento de exponencial (linea roja) comparada con la cuadrática**

**Comparación**

Para resaltar el nivel de importancia de categorizar un algoritmo en un órden de complejidad, tenemos que ver en realidad que tan diferentes pueden ser. Para compararlos entre si, supongamos que todos ellos requieren 1 hora de ordenador para resolver un problema de tamaño N=100.

|  |  |  |  |
| --- | --- | --- | --- |
| **O(f(n))** | **N=100** | **t=2h** | **N=200** |
| **log n** | 1 h | 10000 | 1.15 h |
| **n** | 1 h | 200 | 2 h |
| **n log n** | 1 h | 199 | 2.30 h |
| **n**2 | 1 h | 141 | 4 h |
| **n**3 | 1 h | 126 | 8 h |
| **2**n | 1 h | 101 | 1030 h |

**Fuentes**

* La tecla escape – <http://latecladeescape.com/h/2015/07/que-es-la-complejidad-de-un-algoritmo>
* Servidor WServidor Web de los Laboratorios del DITeb de los Laboratorios del DIT – <http://www.lab.dit.upm.es/~lprg/material/apuntes/o/index.html>
* QuickSort Wikipedia – <https://en.wikipedia.org/wiki/Quicksort>