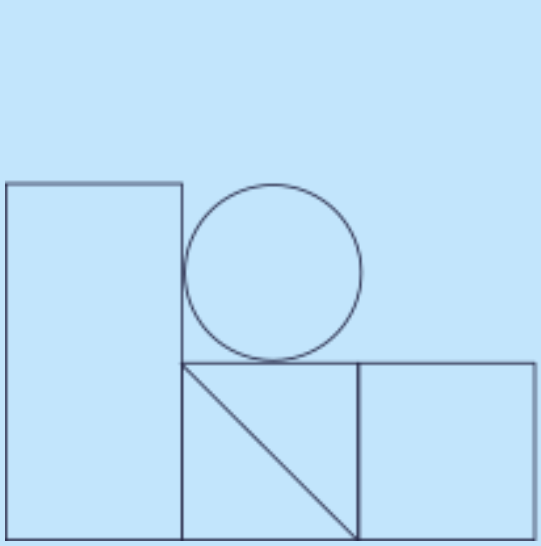
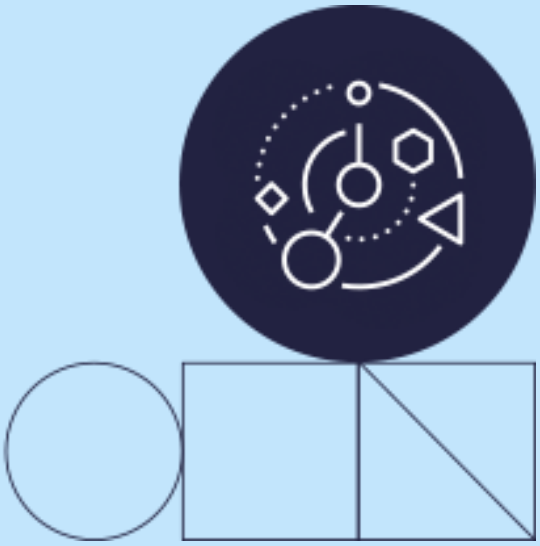
IBM SkillsBuild

IBM

Programando en Python

Complejidad de los algoritmos





Índice

Introducción

Cuando hablamos de la complejidad de un algoritmo, nos referimos en realidad a 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 difícil de entender, la obtención y el estudio de la complejidad requieren ciertas destrezas matemáticas. En esta sección vamos a exponer de manera sencilla la complejidad algorítmica, los tipos de complejidad y ejemplos en código Python.

¿Qué es un algoritmo?

En primer lugar, 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, al resultado de la evaluación lo llamaremos complejidad algorítmica.

Para medir un algoritmo podemos tratarlo desde dos puntos de vista. Bien sea quien resuelva el problema en menos tiempo o quien utilice menos recursos del sistema. A la idea del tiempo la llamaremos complejidad temporal y a la de recursos del sistema 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 consiste en medir cuánto tarda en resolver un problema.

¿El tamaño importa?

Hablemos de algunos conceptos que poseen todos los algoritmos.

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 forman 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 sí 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 con varios algoritmos 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 ordenador muy potente la respuesta del programa sería obviamente mucho más rápido que en un ordenador con pocos recursos. Así que el algoritmo va a tener siempre un tiempo de respuesta diferente para cada ordenador.

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.

Analicemos el siguiente código escrito en Python:

```
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. Empecemos 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 dentro. Dependiendo del valor de la variable **$number**, se realiza una instrucción **n** veces. Por ejemplo, si **$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:

```
def codigo_2():
  a = 0
  a -= 1
  a *= 2
```

En el Código 2 la complejidad sería:

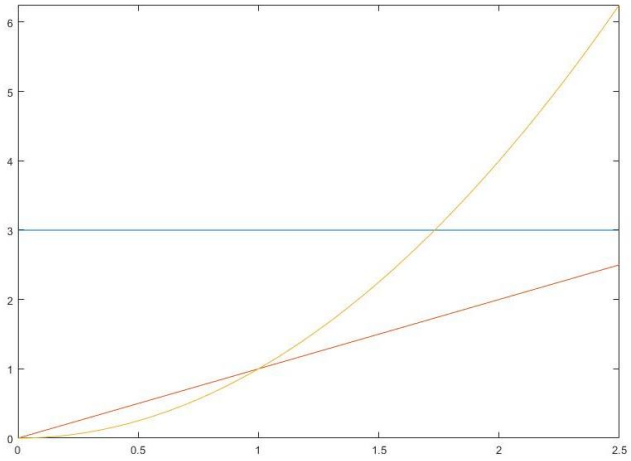
F(x) = 3

```
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 cuál sería el que posee menor complejidad algorítmica:



Esto nos deja que independientemente de los valores, resulta evidente que, cuantos 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.

Escenario de peor caso

Algunos 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:

```
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, porque recorrería todas las instrucciones.

Para el mejor caso tendríamos una complejidad algorítmica:

F(x)=1

Para el peor de los casos la complejidad 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 lo complejo que es un algoritmo con respecto a otros. Para esto recurriremos al orden de complejidad.

Como vimos antes. Para medir un algoritmo necesitamos recurrir al escenario de peor caso y con un gran volumen de datos.

Viéndolo de esta manera podríamos simplificar las ecuaciones de Big-O eliminando algunas características que en ingreso masivo de datos son irrelevantes.

Si comparamos varios algoritmos 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 de ecuaciones podemos agruparlas y referirnos a ellas como **orden 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 tengan 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 **orden cuadrática**. También las podemos expresar como:

O($100m^2$+101n+) € 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 algoritmo 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:

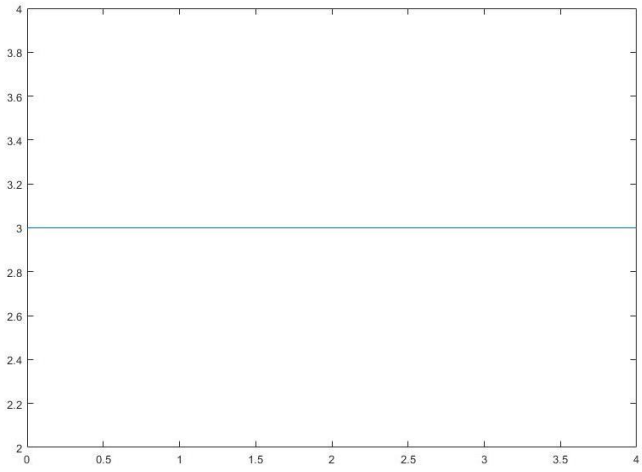
$$O(100n^4 + 3n^3 + 53) \subset O(100n^4 + 3n^3) \subset O(100n^4)$$

Orden de complejidad más conocidas

Constante: $O(1)$

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

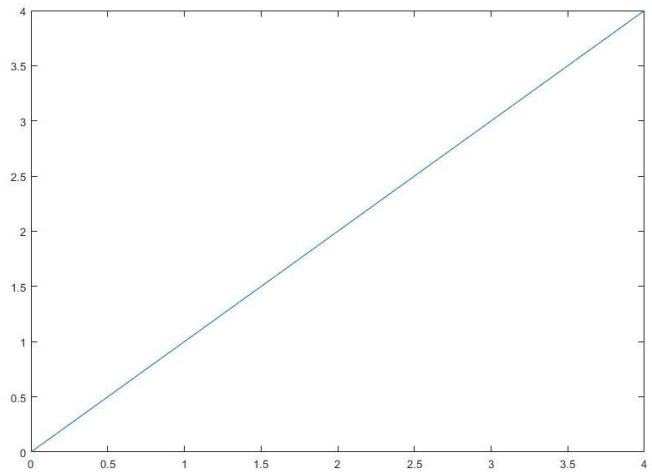
```
def constante():
  x = 50
  ++x
  return x
```



Lineal: $O(x)$

El tiempo crece linealmente mientras crece los datos. Ejemplo:

```
def lineal(number):
  result = 0
  for x in range(0, number):
    ++result

  return result
```

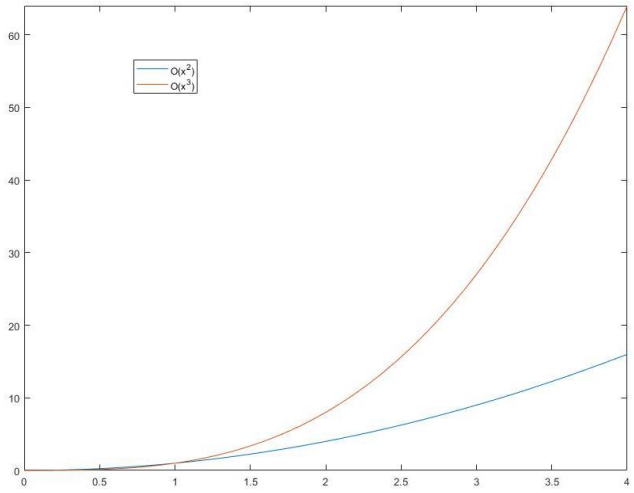


Polinómico: $O(x^c), c > 0$

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:

```
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
```

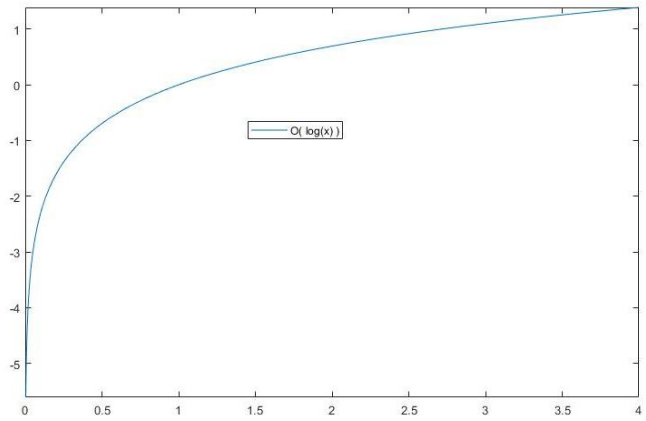


Logarítmico: $O(\log x)$

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 cuál de las mitades se encuentra, luego divide esa parte en 2 mitades iguales y busca nuevamente hasta encontrar el elemento, es un algoritmo recursivo:

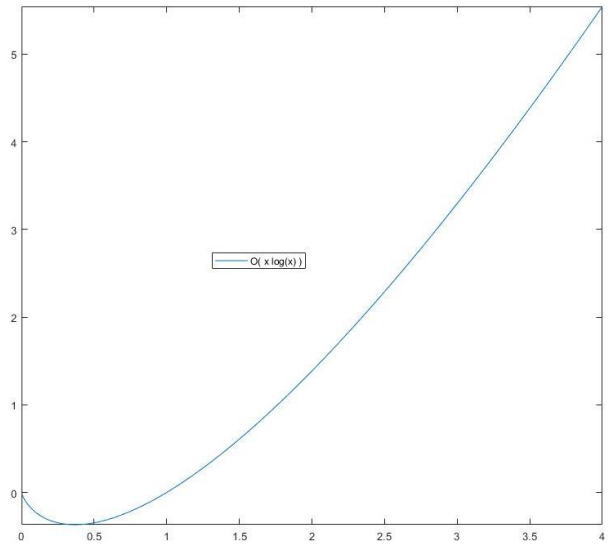
```
def bin(a,x,low,high):
  ans = -1
  if low==high: ans = -1
  else:
    mid = (low+((high-low)//2))
    if x < a[mid]: ans = bin(a,x,low,mid)
    elif x > a[mid]: ans =
bin(a,x,mid+1,high)
    else: ans = mid
  return ans
```



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.

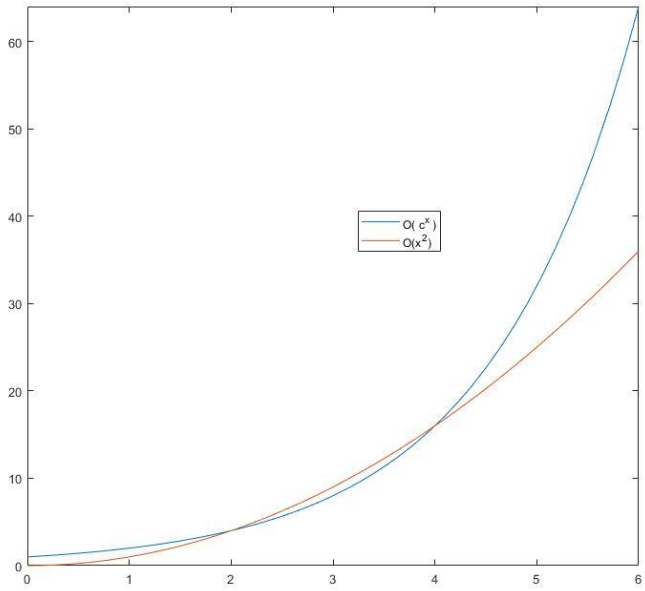
No se incluye el código debido a las distintas versiones, estudios y discusiones de este algoritmo. Pero compartiremos el comportamiento de este orden de complejidad:



Exponencial: $O(c^x)$

Es una de las peores complejidades algorítmicas. Sube demasiado a medida que crece los datos de entrada. Puede verse en la Figura 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:

```
def exponencial(n):
  if n==1 or n==2:
    return 1
  return exponencial(n-1)+exponencial(n-
2)
```



Comparación

Para resaltar el nivel de importancia de categorizar un algoritmo en un orden de complejidad, tenemos que ver en realidad hasta qué punto pueden ser complejos los algoritmos. Para compararlos entre sí 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	10^{30} h

Como podemos observar, a medida que aumenta la complejidad, el tiempo necesario para completar la tarea crece mucho, pudiendo llegar a aumentar enormemente en algunos casos en cuanto hay más datos a manejar.

Si nos encontramos con una función con complejidad cuadrática (O(n2)) o exponencial (O(2n)) por regla general será señal de que el algoritmo necesita una revisión urgente, y mejor no utilizarlo.

Lo interesante de esta notación es que nos permite comparar varios algoritmos equivalentes sin preocuparnos de hacer pruebas de rendimiento que dependen del hardware utilizado. Es decir, ante resultados equivalentes podemos elegir el algoritmo de mayor rendimiento, que lo será siempre independientemente del hardware si el conjunto de datos es lo suficientemente grande (en conjuntos pequeños un hardware más rápido puede dar resultados más rápidos para un algoritmo menos eficiente, pero a medida que el conjunto crece ya no será así).

Por eso, a partir de ahora, cuando veamos que una función documenta explícitamente su complejidad usando la notación Big-O, prestaremos mucha atención y deberemos tener en cuenta las implicaciones que puede tener en nuestra aplicación el hecho de utilizarla. Del mismo modo, cuando creemos un algoritmo para resolver un problema en nuestra aplicación, es interesante anotar en la documentación del mismo (aunque sea en los comentarios de la cabecera) cuál es la complejidad algorítmica del mismo en notación Big-O. Eso ayudará a cualquier programador que venga detrás, tanto a usarlo como a tener que optimizarlo, y sabrá cuál es la lógica a batir.

Aquí tenemos una tabla muy completa con la complejidad de los principales algoritmos para estructuras de datos, ordenación de matrices, operaciones en grafos y operaciones de montón (heap).