# **Complejidad algorítmica.**

## **Notación asintótica**

* O(1) **Complejidad Constante**
* O(n) **Complejidad Lineal**
* O(log n) **Complejidad Logarítmica**
* O(n log n) **Complejidad Logarítmica** quicksort y mergesort.
* O(n^2) **Complejidad Cuadrática** bubble sort y selection sort.
* O(2^n) **Complejidad Exponencial**

# **Complejidad algorítmica.**

La complejidad algorítmica representa la cantidad de recursos (temporales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo.

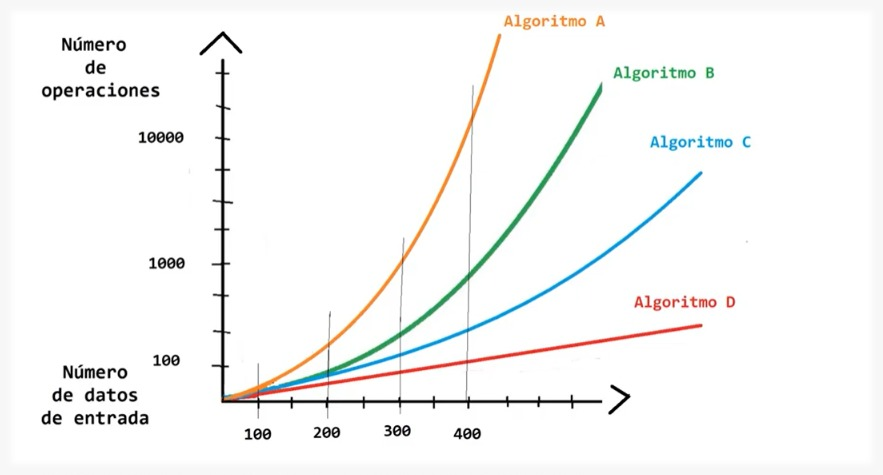
Los criterios que se van a emplear para evaluar la complejidad algorítmica no proporcionan medidas absolutas sino medidas relativas al tamaño del problema.

La cuestión no es la cantidad de tiempo real que se tarda sino cómo el tiempo va creciendo, cómo va aumentando a medida que aumenta el número de operaciones que se llevan a cabo.

A a la hora de medir la eficiencia de un algoritmo no nos interesa el tiempo real porque depende de la máquina, depende del sistema operativo, depende del uso de la CPU que se esté llevando a cabo en ese momento, dependen de muchos factores externos al propio algoritmo y lo que nos interesa es tener una medida con la que comparar cualquier algoritmo independientemente de todos estos factores externos

## **Notación asintótica**

La notación asintótica nos va a permitir representar de una forma general la cantidad de operaciones o instrucciones que requiere un algoritmo para una cantidad dada de datos de entrada.



Hoy en día los ordenadores son muy potentes y pueden ejecutar programas en fracciones de tiempo inapreciables.

Por lo tanto, no va a ser tan importante lo que tarde un algoritmo en ejecutarse con una cantidad de datos de entrada pequeña que apenas se va a apreciar y tampoco sería muy útil para distinguir un algoritmo de otro sino que lo que va a interesar va a ser cómo se comporta ese algoritmo a medida que los datos de entrada aumentan. Esto va a ser importante porque a medida que crecen los datos el tiempo de ejecución puede crecer de una manera o de otra y si ese crecimiento de un algoritmo es muy grande los tiempos de ejecución se pueden disparar y pueden convertir la ejecución de ese algoritmo en impracticable.

Hay tres notaciones asintoticas la big O, la Omega, y la Theta.

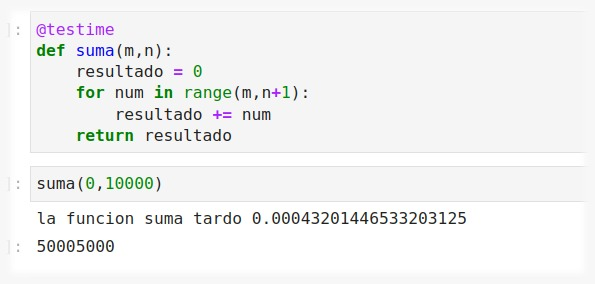
Es importante resaltar que la notación O es útil para indicar la complejidad asintótica en el peor de los casos, en tanto que la notación omega lo hace para la complejidad asintótica en el mejor caso. Por otro lado, la notación theta establece que la complejidad del algoritmo es igualmente ajustada tanto en el peor como en el mejor caso.

En general, estas notaciones son herramientas útiles para analizar algoritmos y elegir el más adecuado para un determinado problema, ya que permiten evaluar su complejidad de tiempo y espacio en función al tamaño de entrada y determinar si son eficientes o no.

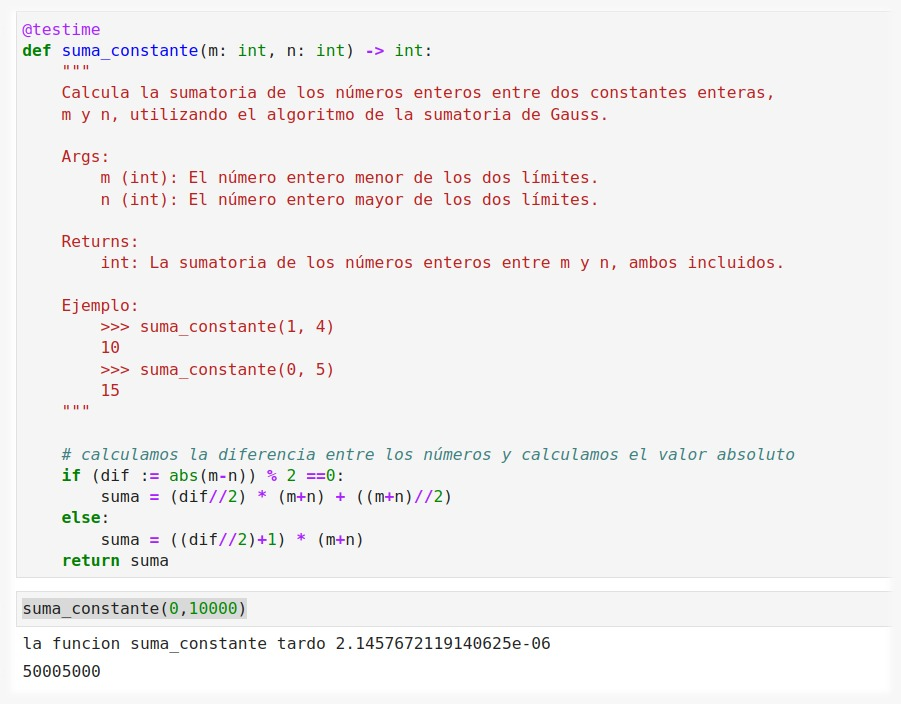
* **O(1):** se refiere a un algoritmo que tiene una complejidad constante, es decir, no importa el tamaño del problema, el tiempo de ejecución es el mismo. Ejemplos de algoritmos con complejidad O(1) son el acceso a un elemento en un arreglo y la búsqueda de una clave en un diccionario.
* **O(n):** se refiere a algoritmos que tienen un tiempo de ejecución proporcional al tamaño del problema. Ejemplos de algoritmos con complejidad O(n) son los algoritmos de búsqueda lineal y la suma de todos los elementos en una lista.
* **O(log n):** se refiere a algoritmos que reducen a la mitad el tamaño del problema en cada paso. Ejemplos de algoritmos con complejidad O(log n) son la búsqueda binaria y la inserción en un árbol de búsqueda binario balanceado.
* **O(n log n):** se refiere a algoritmos que tienen un tiempo de ejecución logarítmico para una parte del problema y proporcional para el resto. Ejemplos de algoritmos con complejidad O(n log n) son los algoritmos de ordenamiento quicksort y mergesort.
* **O(n^2):** se refiere a algoritmos que tienen un tiempo de ejecución proporcional al cuadrado del tamaño del problema. Ejemplos de algoritmos con complejidad O(n^2) son los algoritmos de ordenamiento bubble sort y selection sort.
* **O(2^n):** se refiere a algoritmos que tienen un tiempo de ejecución exponencial. Ejemplos de algoritmos con complejidad O(2^n) son el algoritmo de la mochila y la solución de algunos problemas de optimización combinatoria.

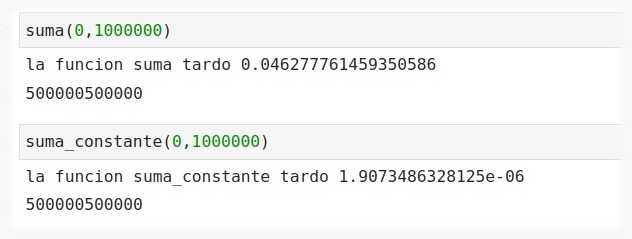
Antes vamos a crear una funcion decoradora para medir el tiempo de las funciones.

Vamos a generar 2 funciones que sumen los elementos de un rango ej del 0 al 100, una usando un algoritmo que va iterar por cada elemento y la otra implentara el algoritmo de gaus.

vemos que el resultado de sumar un rango de 0 10000 es de 50005000 y tardo 0.0004 seg

ahora implementaremos el algoritmo de sumatoria de gaus

vemos que da el mismo resultado pero tardo aprox 100 veces menos que la función suma, pero la diferencia se amplia al aumentar el rango :



observamos que la funcion suma aumento considerablemente su tiempo en cambio la función de gaus se mantuvo constante.

## **Complejidad Constante**

Ahora si los datos crecen pero el número de operaciones no crece sino que permanece constante estaremos ante un orden de complejidad constante que es lo que hemos visto en la función suma constante que aunque la cantidad de números a sumar crecía el número de operaciones permanecía en una simplemente aplicando la fórmula.

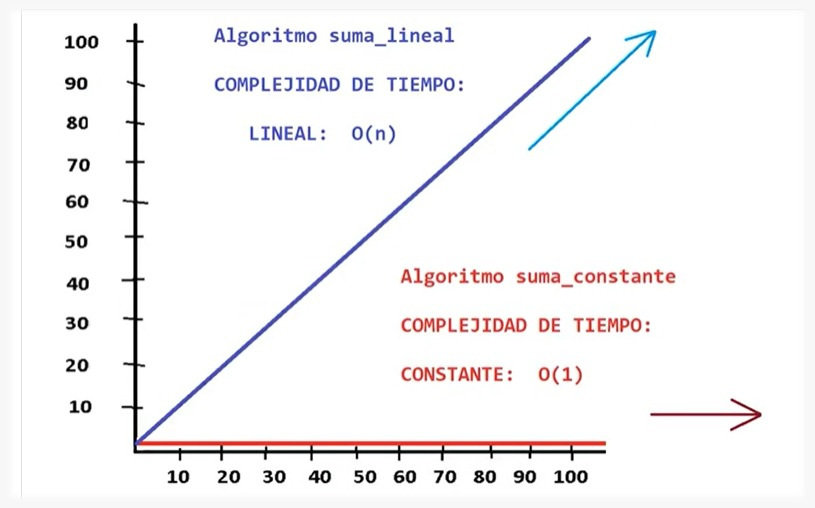
O sea el comportamiento de este algoritmo a medida que los datos crecían por lo tanto si el número de operaciones es de 3 o 5 o incluso 100 no va a tener tanta importancia porque sabemos que siempre va a ser ese mismo número de operaciones un número de operaciones constante y que si los datos de entrada son un millón o mil millones el número de operaciones que va a llevar a cabo ese algoritmo seguirá siendo constante es decir de 3 o 5 o 100 y ese número va a ser insignificante frente a los 1000 millones de datos de entrada por lo tanto podemos ignorar ese número constante y lo vamos a representar de esta forma con un 1.

## **Complejidad Lineal**

Si por el contrario a medida que los datos aumentan, el número de operaciones aumenta en una proporción directa a esos datos diremos que estamos ante un orden de complejidad lineal. Esto es lo que hemos visto en la función suma que a medida que aumentaba la cantidad de números a sumar aumentaba de forma directamente proporcional las operaciones a realizar.

|  |
| --- |
| def suma(m,n):  resultado = 0 una asignacion  for num in range(m,n+1): n operaciones  resultado += num una operacion  return resultado |

La función suma lineal nos encontramos primero con una asignación que va a contar como una operación luego tenemos un bucle for que va a alterar n veces siendo n los datos de entrada por lo tanto va a suponer tantas operaciones como n elementos de entrada se le pasen al algoritmo por lo tanto ya tenemos una más n que tenemos dentro del bucle for pues dos operaciones una suma y una asignación con lo que podemos decir que vamos a tener dos operaciones por cada n dato de entrada esto lo podemos representar de forma simplificada como 1 + 2 n cuando nos encontremos con dos tipos de complejidad como en este caso que tenemos una complejidad constante más una complejidad lineal nos vamos a quedar con el término mayor y podemos ignorar el menor si n es un millón y la constante es uno o tres un millón uno o un millón tres no sería significante además de que si n crece y es dos millones pues el un millón tres por ejemplo ya no tendría sentido como medición por lo tanto nos quedamos sólo con la complejidad de n y podemos además eliminar la constante del término mayor porque en el fondo no deja de ser n

Un ejemplo de complejidad Lineal es la búsqueda de un valor en una secuencia

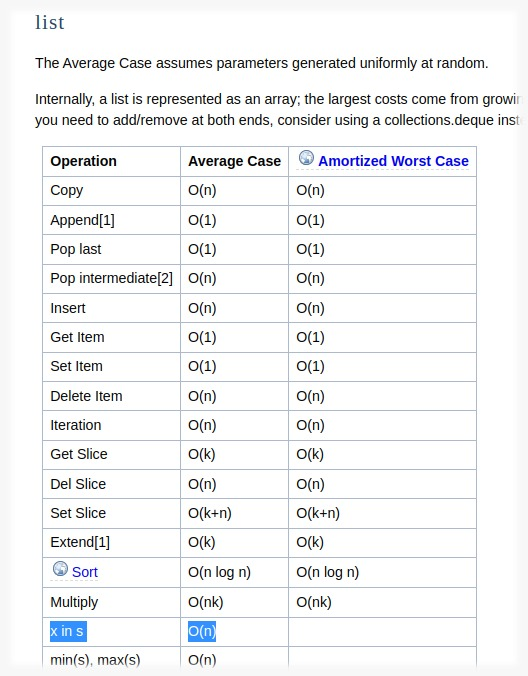


Este algoritmo de búsqueda lineal tiene una complejidad de O(n) ya que deberá recorrer n elementos de la lista para encontrar el elemento

Python posee el operador in que podríamos utilizar pero debajo del operador in también usa un bucle con lo que tenemos también una complejidad de O(n)

En este link podemos encontrar la complejidad de las funciones y métodos de python <https://wiki.python.org/moin/TimeComplexity>

En la página se puede observar que el operador in en lista tienen una complejidad de O(n) mientras que en conjuntos tiene una complejidad O(1) o sea constante.



## **Co****mplejidad polinómica**

En general, la complejidad polinómica se refiere a cualquier algoritmo cuyo tiempo de ejecución aumenta a medida que la entrada crece según una función polinómica.

n otras palabras, el tiempo de ejecución del algoritmo se expresa como una función de la forma

donde k es un número entero positivo y a, b, c, ... , z son constantes.



El orden de complejidad cuadrático O(n^2) es un ejemplo común de complejidad polinómica. Un algoritmo con complejidad O(n^2) tiene un tiempo de ejecución que aumenta proporcionalmente al cuadrado del tamaño de la entrada.

Por ejemplo, si la entrada tiene n elementos, el tiempo de ejecución será aproximadamente n^{2} operaciones. Un ejemplo de un algoritmo cuadrático sería un algoritmo que recorre todos los pares de elementos de una lista de tamaño n y realiza una operación en cada par.

Los algoritmos con complejidad polinómica generalmente se consideran menos eficientes que los algoritmos con complejidad logarítmica o lineal. Sin embargo, algunos problemas solo se pueden resolver con algoritmos polinómicos. En general, cuanto mayor sea el valor de k en la función de complejidad polinómica, menos eficiente será el algoritmo para entradas grandes.

Supongamos que tenemos una matriz de n x n (es decir, con n filas y n columnas), y queremos calcular la suma de todos sus elementos. Podríamos hacerlo de la siguiente manera:

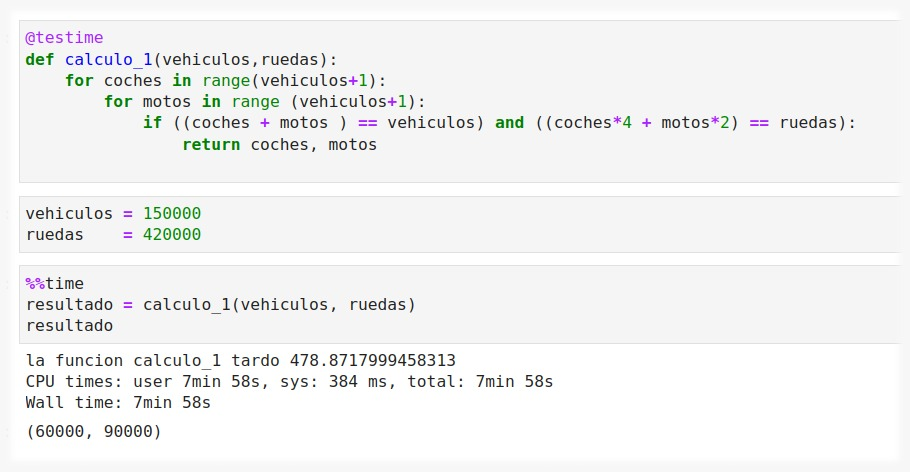
|  |
| --- |
| def suma\_matriz(matriz):  suma = 0  for i in range(len(matriz)):  for j in range(len(matriz)):  suma += matriz[i][j]  return suma |

Este algoritmo utiliza dos bucles for anidados para recorrer la matriz y sumar cada uno de sus elementos. Si contamos las operaciones que realiza este algoritmo, veremos que para cada elemento de la matriz se hacen dos operaciones (una para acceder al elemento y otra para sumarlo a la variable suma), y como hay n x n elementos en total, se hacen n x n x 2 operaciones en total. Por lo tanto, la complejidad de este algoritmo es 𝑂(𝑛2)

Si tenemos una matriz pequeña, este algoritmo funcionará bien y no tendremos problemas de rendimiento. Sin embargo, si la matriz es muy grande (por ejemplo, 1000 x 1000), el número de operaciones será enorme y el algoritmo será muy lento. En ese caso, podríamos buscar formas más eficientes de calcular la suma de los elementos de la matriz (por ejemplo, usando alguna propiedad matemática de las matrices).

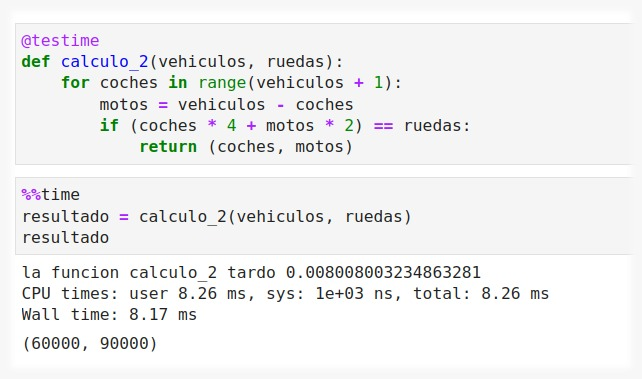
## **Ejemplo de Complejidad Cuadrática**

En este ejemplo se presenta un problema que consiste en contar el número de coches y motos que pasan por una autopista, a partir de dos datos: el número total de vehículos y el número total de ruedas.



Sin embargo, esta solución es ineficiente, ya que tiene una complejidad cuadrática y tardaría mucho tiempo en ejecutarse si el número de vehículos es elevado.

En su lugar, se propone una solución más eficiente que consiste en utilizar una sola variable, en este caso la cantidad de coches, y luego calcular la cantidad de motos a partir de esa variable y los datos de entrada. De esta manera, solo es necesario recorrer el número total de vehículos una vez, lo que reduce significativamente la complejidad del algoritmo y lo hace más eficiente.



Este código recibe como entrada dos números enteros: el número de vehículos que han pasado por la autopista y el número de ruedas que se han contado en total. Luego, mediante un bucle for, va recorriendo todos los posibles valores de la variable coches, desde 0 hasta la cantidad de vehículos que han pasado.

En cada iteración del bucle, se calcula el valor de la variable motos restando coches de vehiculos. Después, se comprueba si la ecuación coches \* 4 + motos \* 2 == ruedas se cumple. Si es así, se devuelve una tupla con los valores de coches y motos. Si no se encuentra ninguna solución, se devuelve None.

#### **Conclución :**

Se ve claramente que cuando los datos son grandes el primer algoritmo de complejidad O(n^2) tarda mas de 7 minutos.

Mentras que el seguno algoritmo de complejidad lineal O(n) tarda apenas un poco mas de 8 milesegundos

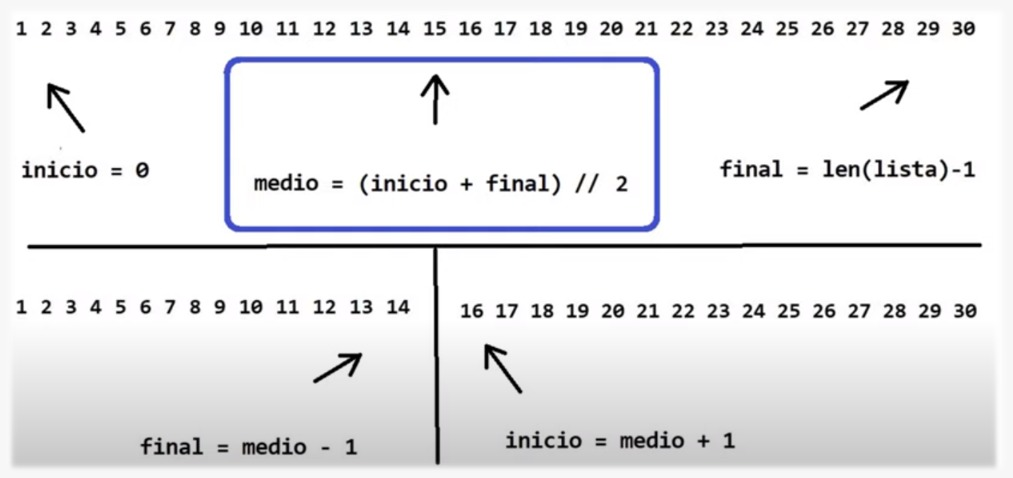
## **Complejida O(log n)**

Para esta complejidad veremos el algoritmo de búsqueda binaria, que es un algoritmo utilizado para buscar un elemento específico en una lista ordenada.

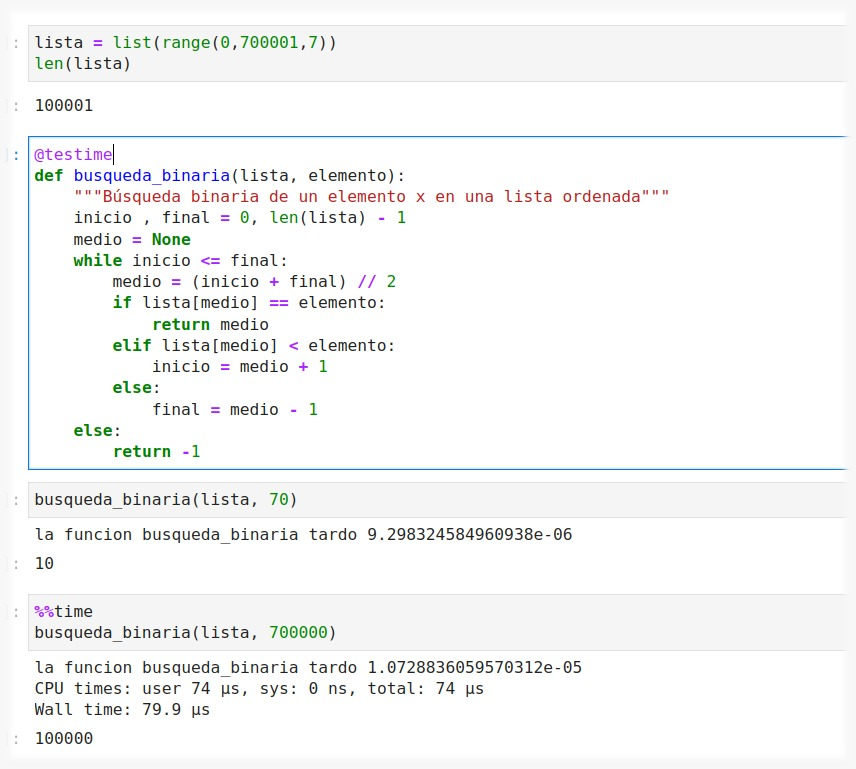
- El algoritmo funciona dividiendo la lista en dos partes y comprobando si el elemento buscado está en la mitad de la lista.

- Si no está allí, el algoritmo descarta la mitad de la lista que no contiene el elemento buscado

- Repite el proceso en la otra mitad de la lista.

El algoritmo es muy eficiente, ya que su complejidad es logarítmica, lo que significa que el número de operaciones necesarias para encontrar el elemento buscado aumenta lentamente a medida que el tamaño de la lista aumenta.

Por lo tanto, es mucho más rápido que el algoritmo de búsqueda lineal, que debe revisar cada elemento en la lista.

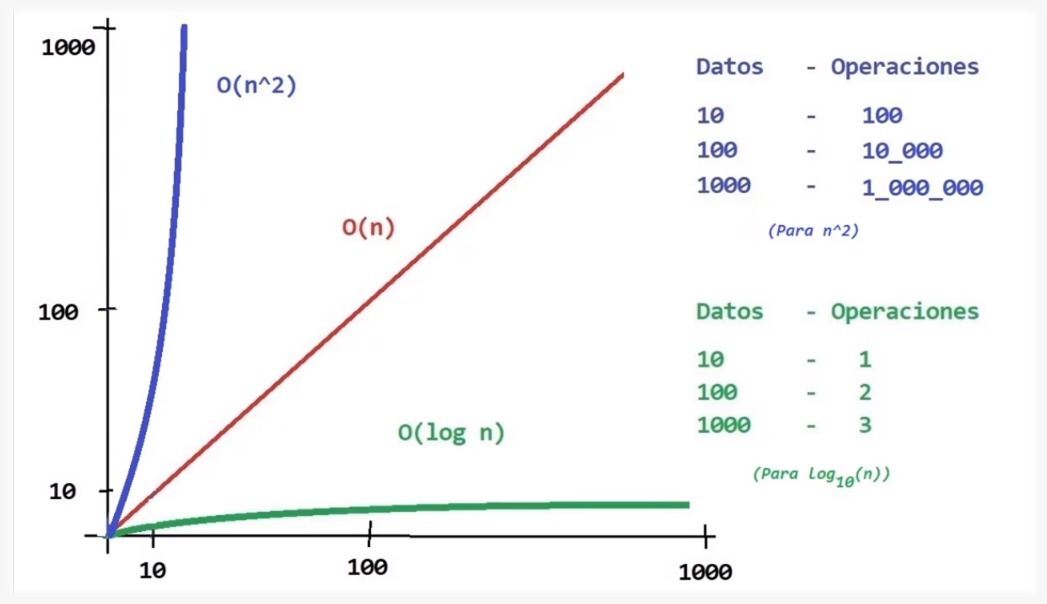
El algoritmo de la búsqueda binaria tiene una complejidad de O(log n).

Veremos qué significa esto, pero para ello tenemos que tener claro que es un logaritmo. Si vamos a la wikipedia <https://es.wikipedia.org/wiki/Logaritmo>

vemos que el logaritmo de un número en una base determinada es el exponente al cual hay que elevar la base para obtener dicho número.

Por ejemplo, el logaritmo en base 10 de 1000 es 3 porque 1000 es igual a 10 a la potencia de 3.

Y aquí vemos la representación de unas funciones logarítmicas. Por lo tanto, si 10^3 = 1000, el log\_10(1000) = 3 . Con lo cual, en una función con complejidad cúbica y una cantidad de datos de entrada de 10, necesitaríamos 1000 operaciones. Mientras que para una función con complejidad logarítmica y una cantidad de datos de entrada de 1000, necesitaríamos 3 operaciones. Aquí tenemos un gráfico con la representación de tres funciones con distinta complejidad, lineal, cuadrática y logarítmica.

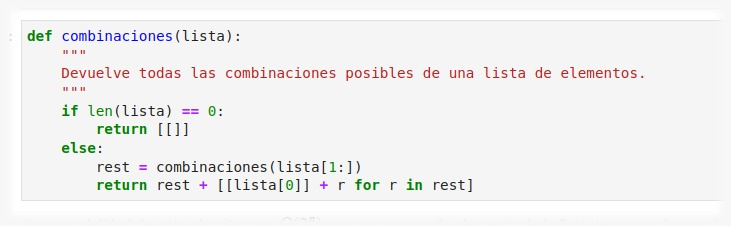


Y vemos cómo mientras la cuadrática se dispara, la logarítmica crece en número de operaciones de una forma muy lenta, apenas una operación más por cada vez que se multiplican por 10 los datos de entrada.

Por lo tanto, la complejidad logarítmica va a ser muy eficiente, no tanto como la constante, pero bastante más eficiente que la lineal. Y por supuesto, muchísimo más eficiente que la cuadrática o cualquier polinomial. Al igual que ocurría en las otras complejidades, vamos a ignorar las constantes por no ser significativas a la hora de medir el comportamiento de una función a medida que crecen los datos de entrada. En el caso de las logarítmicas, podemos ignorar la base. Por lo tanto, no nos importará si una función tiene una complejidad logarítmica en base 2, o en base 5, o en base 10. Podemos ignorar el dato de la base y diremos simplemente que tiene una complejidad logarítmica.

## **Complejidad Exponencial O(2^n)**

Supongamos que tenemos una función que busca todas las combinaciones posibles de una lista de elementos, y queremos calcular su complejidad. El siguiente código implementa esta función:

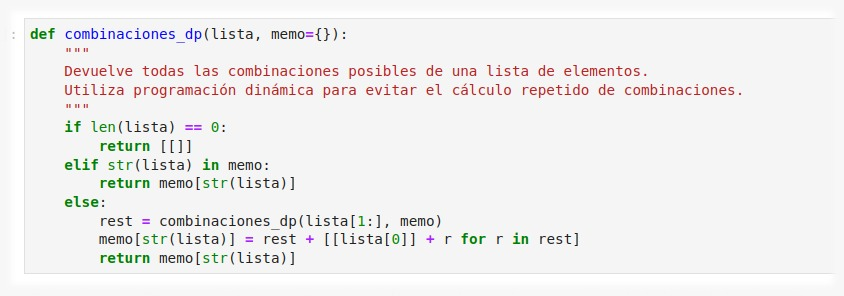


La complejidad de este algoritmo es O(2^n), ya que para cada elemento de la lista, tenemos dos opciones: incluirlo en una combinación o no incluirlo.

Por lo tanto, para una lista de tamaño n, el número de combinaciones posibles es 2^n.

Posible solución con complejidad mucho mejor:

Una posible solución más eficiente sería utilizar la programación dinámica para evitar el cálculo repetido de combinaciones. Por ejemplo, podemos almacenar las combinaciones ya calculadas en un diccionario para evitar volver a calcularlas. El siguiente código implementa una solución con programación dinámica:



En este caso, utilizamos un diccionario memo para almacenar las combinaciones ya calculadas. Si la lista ya se ha procesado anteriormente, simplemente devolvemos la lista almacenada en el diccionario, en lugar de volver a calcularla. La complejidad de esta solución es O(n \* 2n), ya que para cada elemento de la lista, debemos recorrer el diccionario en busca de combinaciones ya calculadas, lo que lleva un tiempo proporcional a n, y luego realizar una operación de lista para crear nuevas combinaciones. En general, esto es mucho más eficiente que el algoritmo exponencial original.

