**Búsqueda secuencial**: recorre uno a uno todos los elementos del arreglo o base de datos hasta dar con el valor buscado. Esto es, desde el primer elemento hasta el elemento n. Por lo tanto, por normal general, una búsqueda secuencial será menos eficiente cuanto mayor sea el número de datos en los que se deba buscar.  
No obstante, también puede ocurrir que se intente buscar justo el primer elemento del arreglo…

**Búsqueda binaria**: se halla el elemento central del conjunto de datos y se compara  con el valor buscado. ¿Es el elemento central mayor o menor que el valor buscado? De esta forma, en cada iteración, se elimina una mitad del conjunto de datos.

Se repite el proceso. Por lo tanto, se debe analizar primero el tamaño del conjunto de datos para determinar el valor central (longitud del vector).

**Big O Notation (O Grande, Notación Asintótica o Notación Landau):** Técnica/metodología para planificar el comportamiento de algoritmos.

La notación Big O es una herramienta muy funcional para determinar la complejidad de un algoritmo que estemos utilizando, permitiéndonos medir su rendimiento en cuanto a uso de espacio en disco, recursos (memoria y ciclos del reloj del CPU) y tiempo de ejecución, entre otras, ayudándonos a **identificar el peor escenario donde el algoritmo llegue a su más alto punto de exigencia**.

1. **O(1): Complejidad Constante**: La complejidad constante nos indica que, sin importar el tamaño de entrada o salida, el tiempo de ejecución y recursos utilizados por nuestro algoritmo siempre será el mismo.

Por ejemplo: una función que devuelva el último elemento de un array tiene una complejidad constante.

Resumiendo, O(1) quiere decir que da exactamente igual que la colección tenga uno o un millón de elementos: insertar o recuperar un elemento siempre tarda más o menos lo mismo.

1. **O(n): Complejidad Linear:** Decimos que un algoritmo tiene complejidad linear, cuando su tiempo de ejecución y/o uso de recursos es directamente proporcional (es decir que se incrementa linealmente) al tamaño del valor de entrada necesario para la ejecución del algoritmo.

Por ejemplo: una función que imprima todos los elementos de un array tiene una complejidad linear, esto es, cuantos más elementos haya en el array, más tiempo le llevará.

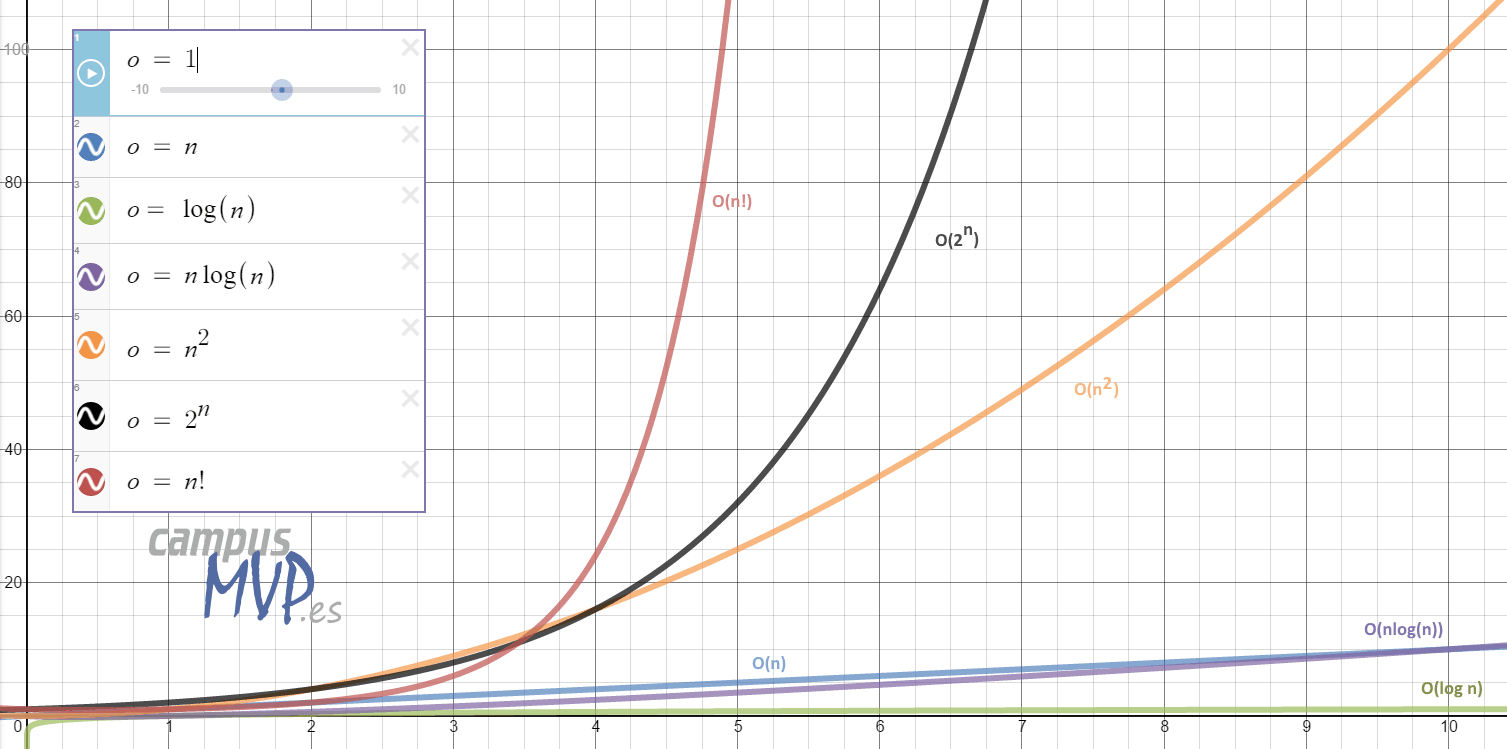
1. **O(log n): Complejidad Logarítmica** La complejidad logarítmica es dada cuando el tiempo de ejecución o uso de recursos es directamente proporcional al resultado logarítmico del tamaño del valor de entrada. Es decir, si tenemos un dato de entrada cuyo tamaño es 10 y nos toma 1 segundo en la implementación del algoritmo, significa que por un valor de entrada cuyo tamaño es 100, nos debe tomar 2 segundos en realizar el algoritmo, un valor de 1000 nos debe tomar 3 segundos y así consecuentemente.

Por ejemplo: una función búsqueda binaria.

1. **O(n ^ 2): Complejidad Cuadrática:** Encontramos complejidad cuadrática en los algoritmos, cuando su rendimiento es directamente proporcional al cuadrado del tamaño del valor de entrada. Es decir, si tenemos como dato de entrada un arreglo con un tamaño de 4 elementos que queremos comparar para ver si hay elementos repetidos, tendremos que hacer 16 comparaciones en total para completar nuestro algoritmo. Esta complejidad es común encontrarla en algoritmos de ordenamiento de datos como el [método de la burbuja](https://www.geeksforgeeks.org/bubble-sort/), el [de inserción](https://es.wikipedia.org/wiki/Ordenamiento_por_inserci%C3%B3n) y el [método de selección](https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n), entre algunos otros.
2. **O(2 ^ n): Complejidad Exponencial** Cuando un algoritmo tiene complejidad exponencial, su rendimiento se incrementa al doble cada vez que se agregue un nuevo dato al valor de entrada, por ende, incrementando su tamaño de manera exponencial. Esto quiere decir que si tenemos un arreglo con 1 elemento y nos toma 10 segundos ejecutar el algoritmo, con 2 elementos nos deberá tomar 100 segundos, con 3 nos deberá tomar 1000, continuando de manera sucesiva.

Un ejemplo bastante común de complejidad exponencial es la sucesión de [Fibonacci](https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci).

Es muy importante conocer el tipo de función/curva que se asocia con la ejecución de una función ya que nos permitirá **saber de antemano si el rendimiento de nuestra aplicación se puede resentir** en caso de que el tamaño de los datos a manejar aumente mucho. Así, entre dos algoritmos/funciones que hagan lo mismo, deberíamos elegir el que tenga una complejidad asintótica menor, para lo cual consultaremos la notación Big-O asociada al mismo.



Enlaces de interés: <https://www.youtube.com/watch?v=MaY6FpP0FEU>

**Análisis Big O Notation**

En cuanto a rendimiento o complejidad, ¿Cómo de bueno es un algoritmo de búsqueda para unos casos determinados? ¿Cómo de malo?

En los algoritmos de búsqueda, algunos funcionan muy bien para grandes cantidades de datos, pero mal en cantidades pequeñas.

Se parte de la lista l\_data = [3, 56, 21, 33, 874, 123, 66, 1000, 23, 45, 65, 56]

Se busca el número 874, utilizando un algoritmo secuencial y uno binario. Se suman el número de iteraciones necesarias para hallar el número.

**Además, para el análisis Big O**, se añaden 10 elemento aleatorios (cuyo valor está entre 0,1000). Cada vez que se añade un elemento, se calcula el tiempo necesario para cada algoritmo en hallar el número 874.

Aquí los resultados:

Texto

Descripción generada automáticamente

1. Algoritmo de búsqueda secuencial: el rendimiento del algoritmo dependerá de la posición en la que se encuentre el número (874) dentro del arreglo de datos. Por lo tanto, por más elementos que se añadan a la lista, el resultado siempre será el mismo.
2. Algoritmo de búsqueda binario: al añadir únicamente 10 elementos aleatorios, el número de iteraciones necesarias para hallar el número 874 es de 2, 3 o 4. No obstante, no se aprecia variación en el tiempo necesario para hallar dicho valor.

Para poder hallar un ejemplo en el que el tiempo requerido por el algoritmo de búsqueda binario cambie, se crea una lista de 10\*\*e número aleatorio, que se añade a la lista detallada en el ejercicio. Aquí los resultados:

{Nº de elementos de la lista} – {Nº de iteraciones} – {Tiempo requerido}:

10 - 4 - 0.0

100 - 6 - 0.0

1000 - 9 - 0.0

10000 - 12 - 0.004345893859863281

100000 - 14 - 0.07643747329711914

1000000 - 17 - 1.1598639488220215

10000000 - 17 - 26.049776792526245

\****Nota***: *en la siguiente iteración da MemoryError*

La primera conclusión es que n crece mucho más rápido que el tiempo requerido por el algoritmo para arrojar el resultado. Esto es, para apreciar un impacto considerable en la eficiencia del algoritmo se debe aumentar drásticamente el tamaño del arreglo de los datos. Por lo tanto, esto indica un comportamiento que sigue una ley logarítmica.

A continuación, se muestran los resultados:

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **O** | **log (n)** |  |
| 1000 | 0 | 3 |  |
| 10000 | 0,0043445890 | 4 |  |
| 100000 | 0,0764374733 | 5 |  |
| 1000000 | 1,1598639488 | 6 |  |
| 10000000 | 26,0497767925 | 7 |  |

Graficando los resultados, se obtiene lo siguiente:

A partir de los puntos obtenidos en el programa (n, O) se dibuja la línea de tendencia logarítmica. Una R tan grande indica que el resultado no es muy ajustado.