- **Introducción**

Partimos recordando el concepto de qué es un algoritmo. Un algoritmo es una secuencia de instrucciones cuyo objetivo es la **resolución de un problema en un tiempo dado**. Dicha resolución puede ser planteada desde distintos puntos de vista, aplicando distintas estrategias, y por consiguiente, llegando **a soluciones algorítmicas distintas**.

Desde el punto de vista computacional, es necesario disponer de alguna forma de comparar una solución algorítmica respecto a otra **en términos de eficiencia**, en especial al atacar problemas de gran tamaño.

Saber si un algoritmo es mejor que otro puede estudiarse desde dos puntos de vista: un algoritmo es mejor **cuanto menos tarde en resolver un problema,** o bien, **cuanta menos memoria necesite.**

A la idea del tiempo que consume un algoritmo para resolver un problema le llamamos **complejidad temporal** y a la idea de la memoria que necesita el algoritmo le llamamos **complejidad espacial**.

Por ahora nos focalizaremos en la complejidad temporal y eventualmente, cuando sea necesario, haremos referencia a la complejidad espacial.

- **El tamaño de un problema**

La idea que subyace tras el concepto de complejidad temporal de un algoritmo es, básicamente, medir cuánto tarda en resolver el problema.

Para resolver cualquier problema, son necesarios unos datos de entrada sobre los que trabaja el algoritmo. Sin embargo, debemos tener en cuenta algunas consideraciones.

Tomemos como ejemplo un típico algoritmo que ordena los elementos de un vector. El algoritmo consta de una serie de instrucciones que se repiten una y otra vez (bucles), y probablemente, de una serie de comparaciones que hacen que se ejecute uno u otro camino dentro del algoritmo.

Pregunta: Tardará lo mismo un algoritmo en ordenar un vector con 100 elementos que uno con 10.000 elementos? Obviamente no; el tamaño del vector incide directamente en el tiempo que tarda el algoritmo en resolverse.

Es por esta razón que debemos comenzar a hablar del **tamaño del problema**,

Todo problema tiene un **tamaño**, que es un valor o un conjunto de valores que se pueden obtener de los datos de entrada y que si varían, normalmente tienen una repercusión en el tiempo que tardará el algoritmo en finalizar.

Volviendo al ejemplo del problema de ordenar un vector, el tamaño del problema nos lo da el número de elementos del vector.

Otro ejemplo: en un algoritmo que halle el término n-ésimo, el tamaño del problema nos lo da el propio término número n que queremos hallar.

La complejidad se calcula en función de un tamaño genérico, no concreto; esto es, la complejidad de un algoritmo de ordenación, por ejemplo, se calcula pensando en un array de longitud *n*, no de longitud 50, 500 ó 5.000. Dicho esto, **la complejidad no se expresa como un número sino como una función *T(n).***

Otra consideración a tener en cuenta a la hora de tratar con la complejidad es que el tiempo que tarda un algoritmo en resolver un problema depende de la arquitectura del ordenador donde se ejecuta. Parece obvio que el mismo algoritmo ejecutado en un ordenador el doble de rápido que otro tardará la mitad en encontrar la solución.

De esto se deduce que el tiempo medido en segundos, milisegundos etc., no nos sirve porque el resultado variaría de un ordenador a otro. Por lo tanto hablaremos de ***unidades de tiempo*.**

Además es obvio también que el mismo algoritmo tardará más o menos tiempo en solucionar un problema de un tamaño u otro. Es decir, el algoritmo no puede tardar lo mismo en ordenar un array de 100 elementos que uno de 10.000 elementos.

Lo que realmente queremos saber es **cómo crece el número de instrucciones necesarias para resolver el problema con respecto al tamaño del problema,** esta es la esencia de la complejidad.

- **Operaciones Elementales (OE)**

En virtud de lo expuesto, es necesario lograr una simplificación que nos permita en lugar de medir tiempos, contar las instrucciones que debe realizar el algoritmo. Para ello, partimos de la suposición de que cada instrucción se ejecuta en un tiempo constante.

Se considera como **1 OE:**

- Operaciones aritméticas básicas
- Asignaciones a variables de tipo predefinido por el compilador,
- Saltos (llamadas a funciones),
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas(vectores y matrices).

El tiempo de **1 OE es de orden 1**

A continuación veremos algunos ejemplos:

Algoritmo 1: Suma de n enteros

```
1: def sumadeN(n):                2 OE
2:       theSum = 0               1 OE
3:       for i in range(1,n+1):   3 OE + n OE
4:       theSum = theSum + I      2 OE + 4OE
5:   return theSum                2 OE
```

$$T(n)=2+1+3+\sum_{n} 6+2$$

T(n)=2+1+3+6n+2

T(n)=8+6n

En este algoritmo hay un bucle que se ejecuta *n* veces, y en su interior hay una instrucción (línea 4). Eso quiere decir que la línea 4 se ejecuta *n* veces y por último se ejecuta el return.

En consecuencia, el número de OE que se ejecutan en total es *T(n)= 8+6n .*

La idea que subyace es que podemos saber cómo se comporta el algoritmo conforme el tamaño del problema va creciendo. En este caso, si representamos gráficamente *8+6n* con respecto a *n* veremos que **esta función es una recta**.

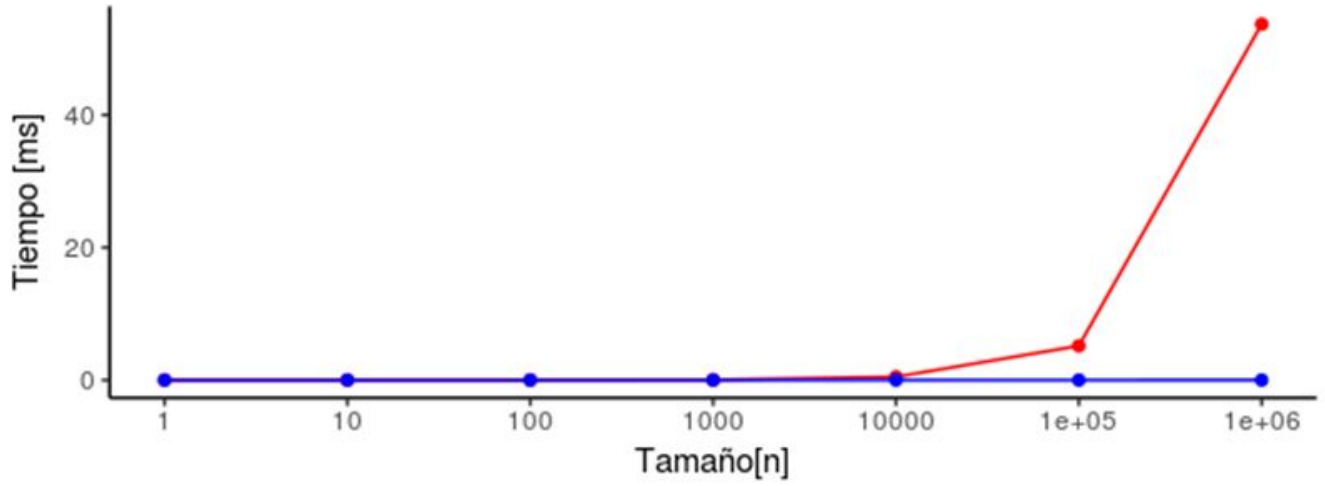
Algoritmo 2: Suma de n enteros

```
1: def sumDeN2(n):                2 OE
2:     return (n*(n+1))/2         5 OE
```

T(n) = 2+5 = 7

La complejidad del Algoritmo 2 es 5. En este caso la representación gráfica es una **función constante**.

Para este algoritmo podemos suponer que cuando lo traslademos a un lenguaje de programación concreto sobre un ordenador concreto, lo que nos importa es saber de qué manera aumenta el tiempo con respecto al tamaño del problema.



Independientemente de los valores numéricos, resulta evidente que cada gráfica crece de manera distinta. A medida que *n* se vaya haciendo grande, las dos gráficas se distanciarán cada vez más.

Eso es lo que nos importa realmente de un algoritmo: saber cómo crece el número de instrucciones a realizar conforme lo apliquemos cada vez a problemas más grandes, más que el tiempo medido en segundos.

- **Peor caso, mejor caso**

En los tres ejemplos que hemos visto hasta ahora, la complejidad del algoritmo es totalmente dependiente del tamaño del problema, pero no todos los algoritmos se comportan de igual manera frente a un problema del mismo tamaño.

En la mayor parte de los algoritmos, también influye el propio contenido de los datos. Es posible que para un problema determinado de tamaño *n*, unas veces el algoritmo tarde más y otras tarde menos,

Consideremos como ejemplo la búsqueda de un elemento en un vector. El elemento puede estar en cualquier posición, en la primera posición se consideraría **el mejor caso**, en la última posición o incluso no estar en el vector serían considerados **el peor caso.**

Si por ejemplo pasamos a la función *search* un vector de *tamaño=100* enteros, y un elemento *x=17*. Es evidente que el tamaño del problema es *n=100,* ya que es lo que determina el número de instrucciones que se ejecutarán. Sin embargo, es posible que el valor x=*17* esté situado en la *posición 30* del vector, con lo que el bucle se realizará *30 veces*, o quizá en la *posición 50*, o quizá no esté en el vector, con lo que el bucle se ejecutará *n=100* veces, recorriendo todo el vector.

En éste caso, nos conviene distinguir dos métricas: qué es **lo peor** que nos puede pasar para un problema de tamaño n, y qué es **lo mejor** que nos puede pasar para un problema de tamaño *n*.

Lo mejor que nos puede pasar es que encontremos el valor *x* en la primera iteración. En ese caso, el bucle se ejecuta una sola vez, o sea que *n=1*.

Lo peor que puede pasar es que el valor *x* no se encuentre en el vector, así que el bucle se ejecutará *n* veces, recorriendo todo el vector.

Para expresar esto, se utiliza una notación específica, diremos que para este algoritmo, su complejidad en el peor caso es **O(n),** ya que cuando *n* es lo suficientemente grande, un valor constante no afecta el resultado**.** A esta notación se le denomina **"O Grande",** del inglés **"Big-Oh"**, o simplemente **"Complejidad en el peor caso"**. Aunque es una "**O**", realmente viene de la letra griega Omicron.

En general, cuando decimos "complejidad" a secas, casi siempre nos referimos a la **complejidad en el peor caso,** es decir, cuánto va a tardar el algoritmo **como mucho**.

- **Asíntotas y órdenes de complejidad**

La idea de la complejidad de un algoritmo, es conocer cómo se comporta el tiempo de ejecución conforme el tamaño del problema va creciendo, especialmente para valores muy grandes y especialmente en el peor de los casos.

En ese contexto, podemos hacer otra "simplificación". El hecho es que podemos encontrar ciertas similitudes entre las funciones que definen la complejidad de los algoritmos.

Si comparamos todos los algoritmos cuya complejidad es **lineal,** es decir una recta, por ejemplo: *10n+3 ó 32n+12 ó 56n+1* y los comparamos con todos aquellos cuya complejidad es **cuadrática,** por ejemplo: $2n^2+3$ *ó* $4n^2+n$ *ó* $6n^2+4n+3$; y dibujamos sus funciones de complejidad en una gráfica

observaremos que conforme *n* se va haciendo grande, tienen un patrón de crecimiento bien diferenciado.

En ese contexto, podemos agrupar todas las complejidades que crecen de la misma manera. A ese agrupamiento le vamos a llamar **orden de complejidad**.

Esto es, para todas las funciones que agrupemos en un mismo orden, encontraremos una asíntota que al multiplicarla por un valor nos acote a nuestra función superiormente cuando estemos tratando el peor caso.

Por ejemplo, todas las complejidades cuadráticas están acotadas asintóticamente por n^2. Eso quiere decir que para cualquiera de las complejidades cuadráticas que hemos visto antes, por ejemplo $6n^2+4n+3$, existe un valor real *c* que hace que $6n^2+4n+3 \leq cn^2$ cuando *n* se hace muy grande, es decir, cuando $n \rightarrow \infty$.

La notación para expresar esto es como sigue: Por ejemplo, para decir que $6n^2+4n+3$ está determinado por la asíntota superior n^2, decimos "**que $6n^2+4n+3$ pertenece al orden de n^2**" o "**que $6n^2+4n+3$ es del orden de n^2**" y lo escribimos formalmente de ésta manera: $6n^2+4n+3 \in O(n^2)$

En lugar de hallar los tiempos exactos que tarda un algoritmo en solucionar uno o varios problemas, lo que analizamos es la asíntota que representa a todos los algoritmos cuyo tiempo crece de igual forma cuando el tamaño del problema tiende a infinito. Eso hace que el cálculo de la complejidad se simplifique mucho.

Lo que se busca en general no es la función de complejidad real, sino el **orden** al que pertenece la complejidad del algoritmo. Cada secuencia de instrucciones se cuenta como una sola instrucción. Lo que influye en la complejidad son principalmente los **bucles** (for/While) y las **condiciones** (if/else) que tienen efecto sobre los bucles.

Cuando decimos que la complejidad de un algoritmo es de un orden concreto, por ejemplo n^2, podemos estar seguros de que para cualquier valor de *n*, aún en el peor caso, el valor que obtengamos nunca será mayor que cn^2, siendo *c* un real mayor que 1.

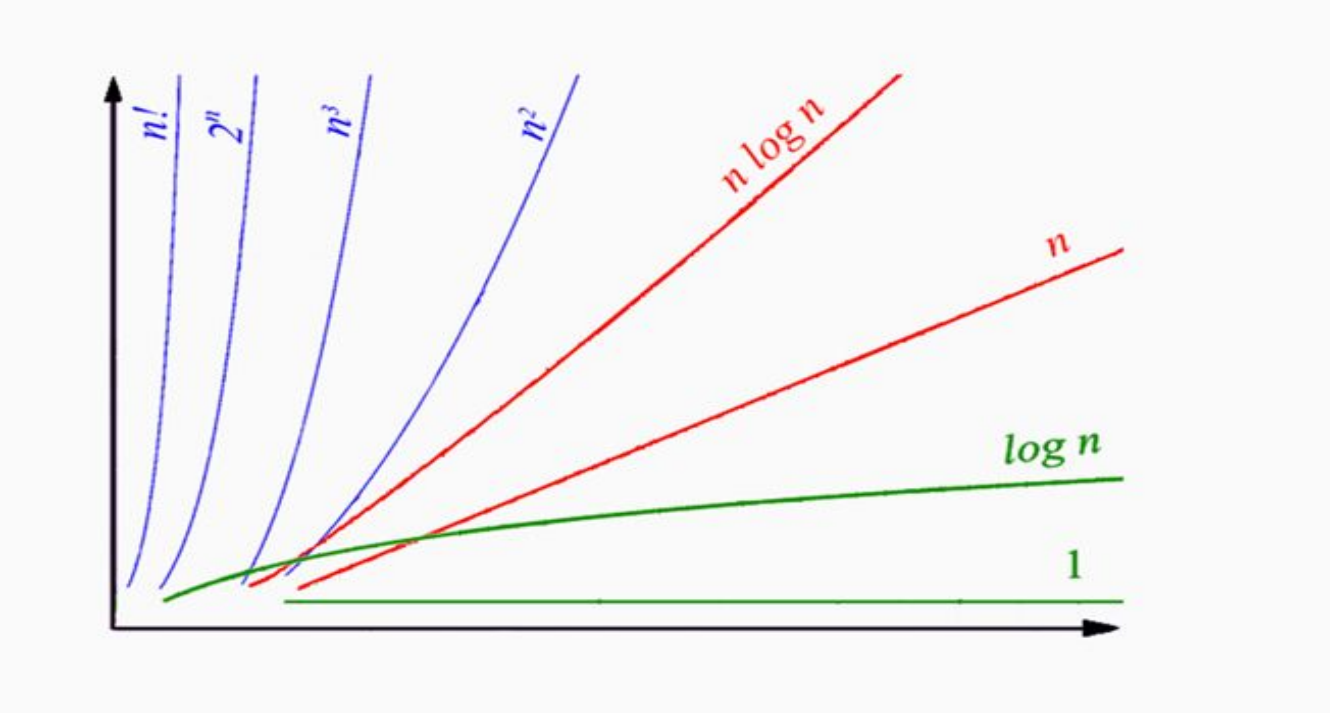
Órdenes de complejidad más comunes

Los órdenes de complejidad que se suelen manejar son los siguientes:

Orden	Nombre	Comentario
O(1)	constante	Todos aquellos algoritmos que responden en un tiempo constante, sea cual sea la talla del problema. Son los que aplican alguna fórmula sencilla, por ejemplo, hallar el máximo de dos valores

O(log n)	logarítmico	Los que el tiempo crece con un criterio logarítmico, independientemente de cuál sea la base mientras ésta sea mayor que 1. Por eso, normalmente, ni siquiera se indica la base. No suelen ser muchos, y normalmente están bien considerados, ya que implican que un bucle realiza **menos** iteraciones que la talla del problema, lo cual no suele ser muy común. Por ejemplo, la búsqueda dicotómica en un vector ordenado.
O(n)	lineal	El tiempo crece linealmente con respecto a la talla. Por ejemplo, encontrar el máximo de un vector de talla n.
O(n·log(n))	Enelogarímico o n por logaritmo de n	Éste orden tiene muchos nombres. Es un orden relativamente bueno, porque la mayor parte de los algoritmos tienen un orden superior. En éste orden está, por ejemplo, el algoritmo de ordenación Quicksort, o la transformada rápida de Fourier.
$O(n^c)$, con c>1	polinómico	Aquí están muchos de los algoritmos más comunes. Cuando c es 2 se le llama **cuadrático**, cuando c es 3 se le llama **cúbico**, y en general, polinómico. Intuitivamente podríamos decir que éste orden es el último de los aceptables (siempre y cuando c sea relativamente bajo). A partir del siguiente, los algoritmos son complicados de tratar en la práctica cuando n es muy grande.
$O(c^n)$, con c>1	exponencial	Aunque pudiera no parecerlo, es mucho peor que el anterior. Crece muchísimo más rápidamente.
O(n!)	factorial	Es el típico de aquellos algoritmos que para un problema complejo prueban todas las combinaciones posibles.

- **Representación Gráfica**



Por ejemplo, el orden exponencial contiene a: $O(2^n)$, $O(3^n)$, $O(4^n)$... etc. Lo mismo ocurre con (n^c), que contiene a $O(n^2)$, $O(n^3)$, $O(n^{1000})$..*etc..*

Para la complejidad en el mejor caso se utilizan los mismos órdenes, sólo que en lugar de utilizar la letra O (Omicron), para denotarla se utiliza Ω (Omega).

Así pues, para el algoritmo de ejemplo que obtenía si un valor *x* estaba en un array de tamaño *n*, decimos que su complejidad, en el peor caso está en el orden O(n) y en el mejor caso en el orden Ω(1).

Por último, respecto a la complejidad espacial, mencionar que podemos actuar de la misma manera, sólo que en lugar de contar instrucciones, contamos las piezas de memoria **dinámica** que se utilizan. Sin embargo, su estudio suele tener menos interés, porque el tiempo es un recurso mucho más valioso que el espacio.