**Concurrencia**: se refiere a la habilidad de distintas partes de un programa, algoritmo problemade ser ejecutado en desorden o en orden parcial, sin afectar al resultado

final.

**Cómputo**: cuenta o cálculo.

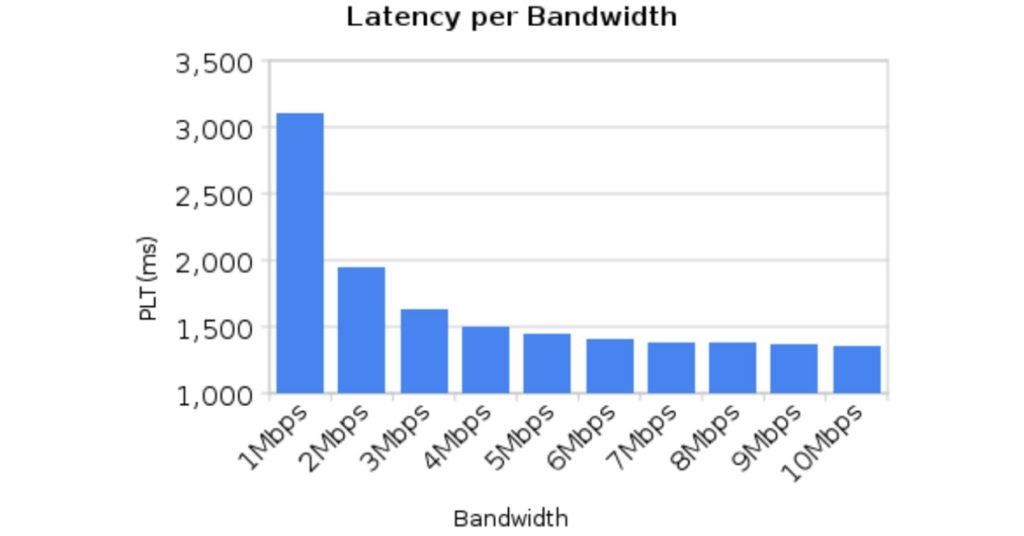
**Computabilidad**: analiza los límites de la solución de problemas a partir de algorítmo. Cuando un problema no puede resolverse a través de un cómputo, suponen un límite para la computación.

**Latencia**: tiempo que tarda un paquete en transmitirse dentro de una red. La latencia influye, por ejemplo, en el tiempo que tardar en cargar una web.

Los factores que generan latencia son:

1. La tecnología de acceso a Internet: ADSL o Fibra.
2. Distancia entre los dos puntos que quieran establecer la comunicación.
3. Capacidad del dispositivo desde el que nos conectamos.

A partir de los 10 Mbps de conexión, no notaremos apenas diferencia al aumentar la velocidad de nuestra conexión, debido a que ya son otros factores los que afectan en la reducción de la latencia.



En el gráfico se aprecia que, a pesar de aumentar la velocidad de conexión, la latencia se estabiliza. Esto es, con una velocidad de conexión de 9Mbps la latencia es la misma que con 10Mbps.

**Eficiencia algorítmica**: el término eficiencia algorítmica es usado para describir aquellas propiedades de los algoritmos que están relacionadas con la cantidad de recursos utilizados por el algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza. La eficiencia algorítmica puede ser vista como análogo a la ingeniería de productividad de un proceso repetitivo o continuo.

Se necesita un método para evaluar la eficiencia de un algoritmo, **independientemente de:**

* **Del ordenador donde se ejecute.**
* **Del lenguaje informático en el que esté programado.**
* **Del sistema operativo en el que se ejecute el programa.**
* **De otros factores externos al propio algoritmo.**

**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.

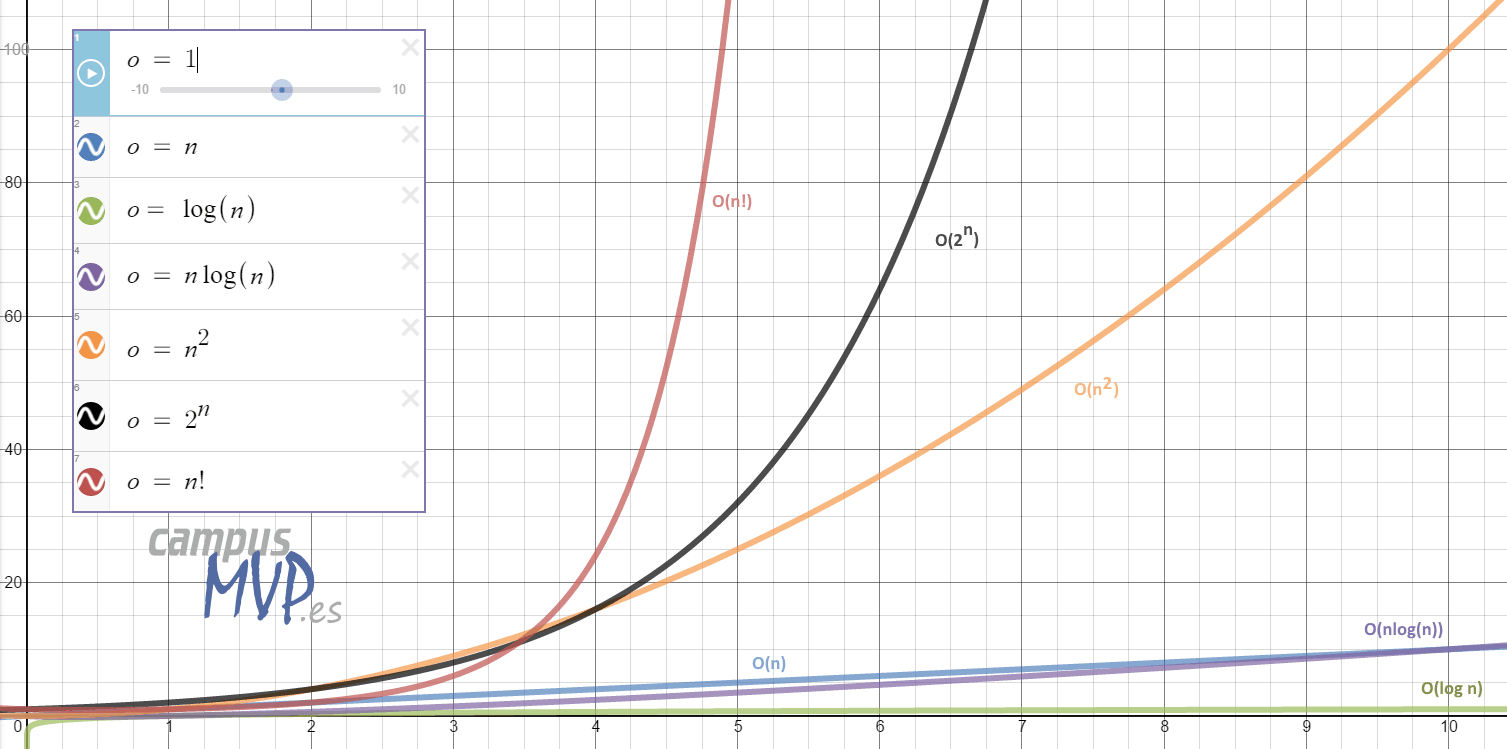
1. **O(log n): Complejidad Logarítmica** La complejidad logarítmica es dada cuando el tiempo de ejecución o uso de recursos es directamente proporcional al resultado logarítmico del tamaño del valor de entrada. Es decir, si tenemos un dato de entrada cuyo tamaño es 10 y nos toma 1 segundo en la implementación 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>

Imagen que contiene Texto

Descripción generada automáticamente