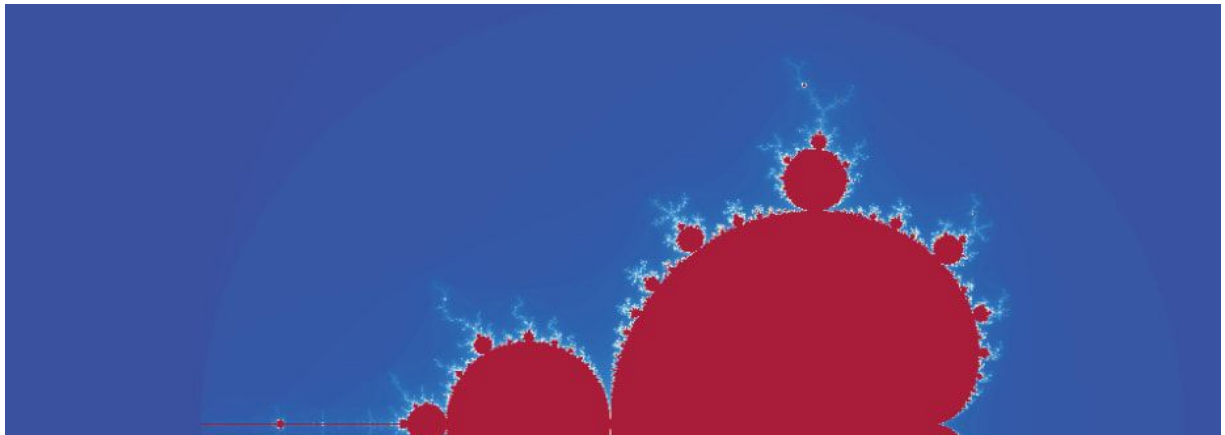
Complejidad algorítmica



Big O

Algoritmo

Conjunto de instrucciones o reglas definidas y no-ambiguas, ordenadas y finitas que permite, típicamente, solucionar un problema, realizar un cómputo, procesar datos y llevar a cabo otras tareas o actividades.

Un determinado problema computacional puede ser resuelto por un conjunto infinito de algoritmo.

¿Qué es Big O?

Es una métrica que revisa la **eficiencia** de un algoritmo. ¿Que tan bueno es un algoritmo?

Básicamente cuando expresamos la complejidad en Big O, estamos indicando **como el tamaño de un conjunto de datos afecta al rendimiento** de un algoritmo.

Tiempo de ejecución de un algoritmo

En BigO no podemos hablar de tiempo, ya que el tiempo dependerá de la máquina en la cual ejecutamos el algoritmo. Es una métrica que no depende de donde o quien ejecute el problema.

Cantidad de información

No todos los algoritmos funcionan igual con la misma cantidad de información.

Algunos algoritmos funcionan bien cuando trabajan con poca cantidad de datos y mal con mucha cantidad de datos a tratar.

Y otros algoritmos al revés, son más eficientes cuanto tienen que trabajar con una gran cantidad de datos

Notación Big O

O(n)

O = orden de la función

n = representa el tamaño de un conjunto

Ejemplo

Queremos enviar un dato a alguien que está en América.

¿Qué medio o mecanismo usarías?

- Llamar por teléfono al amigo
- Enviar un email
- Subir el mensaje a mi nube y compartirlo
- Coger un avión, visitar a mi amigo, y dar el mensaje en persona

Respuesta: Depende

Tipos de complejidad

- **Complejidad en tiempo**: que tan eficiente es mi algoritmo para que tarde menos tiempo en realizar la tarea.
- **Complejidad en espacio**: cuanta memoria o espacio en disco usa mi algoritmo.

Tipos de problema

Problema Tratable - Intratable

Tratable

Problema para el cual existe un algoritmo de complejidad polinomial
Se los conoce como problemas “P” (de orden polinomial)

Intratable

Problema para el que no se conoce un algoritmo de complejidad polinomial
Se los conoce como problemas “NP” (de orden no determinístico polinomial)

Tipos de Big O

Tipos

O(1) = complejidad constante

O(log n) = complejidad logarítmica

O(n) = complejidad lineal

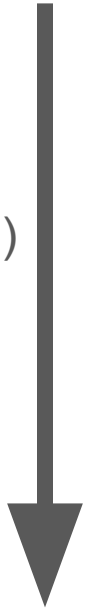
O(n * log n) = complejidad cuasi-lineal (logarítmica-lineal)

$O(n^2)$ = complejidad cuadrática

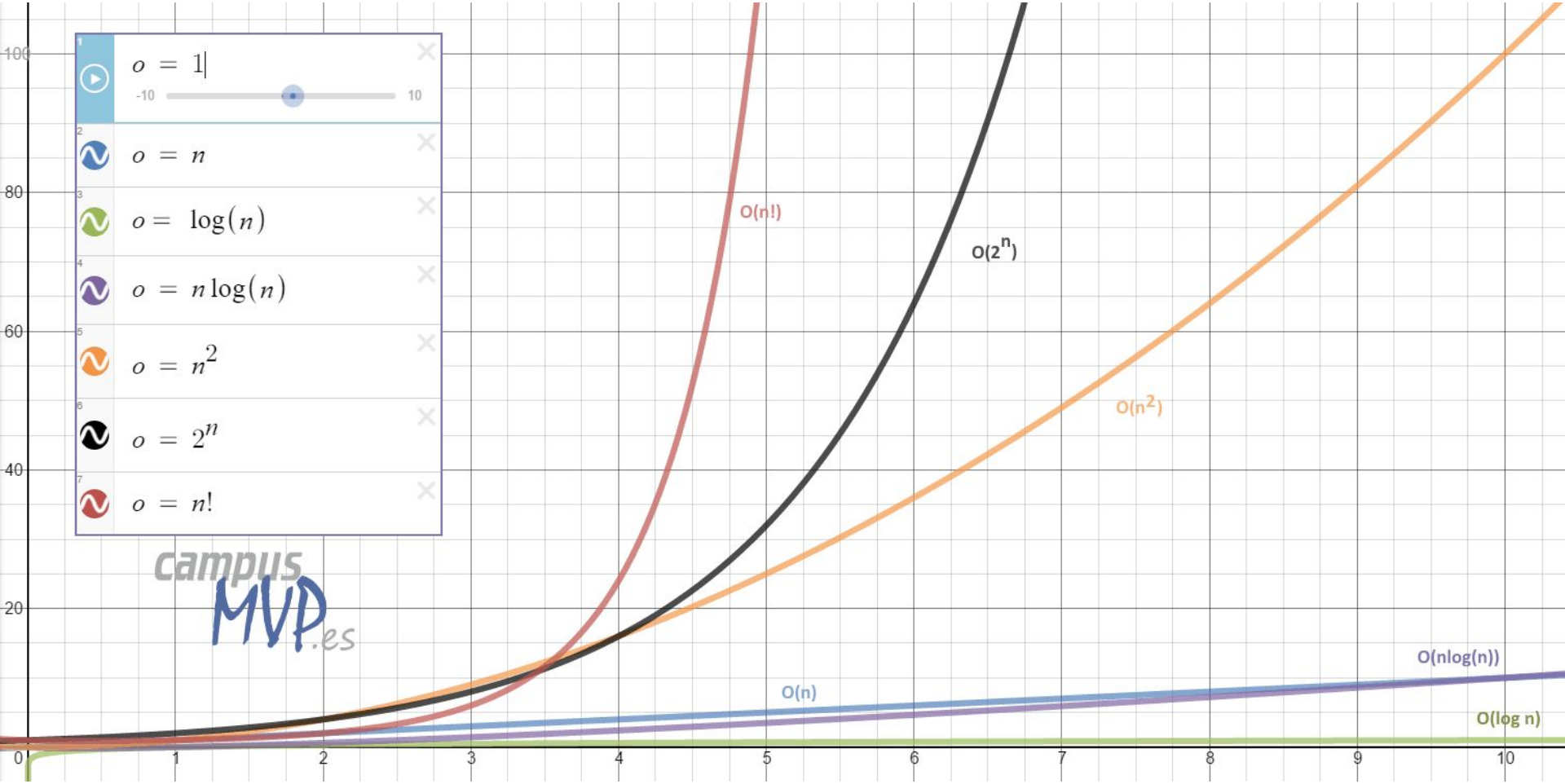
$O(n^a)$ = complejidad polinomial (a>=2) Ej $O(n^3)$ = cúbica

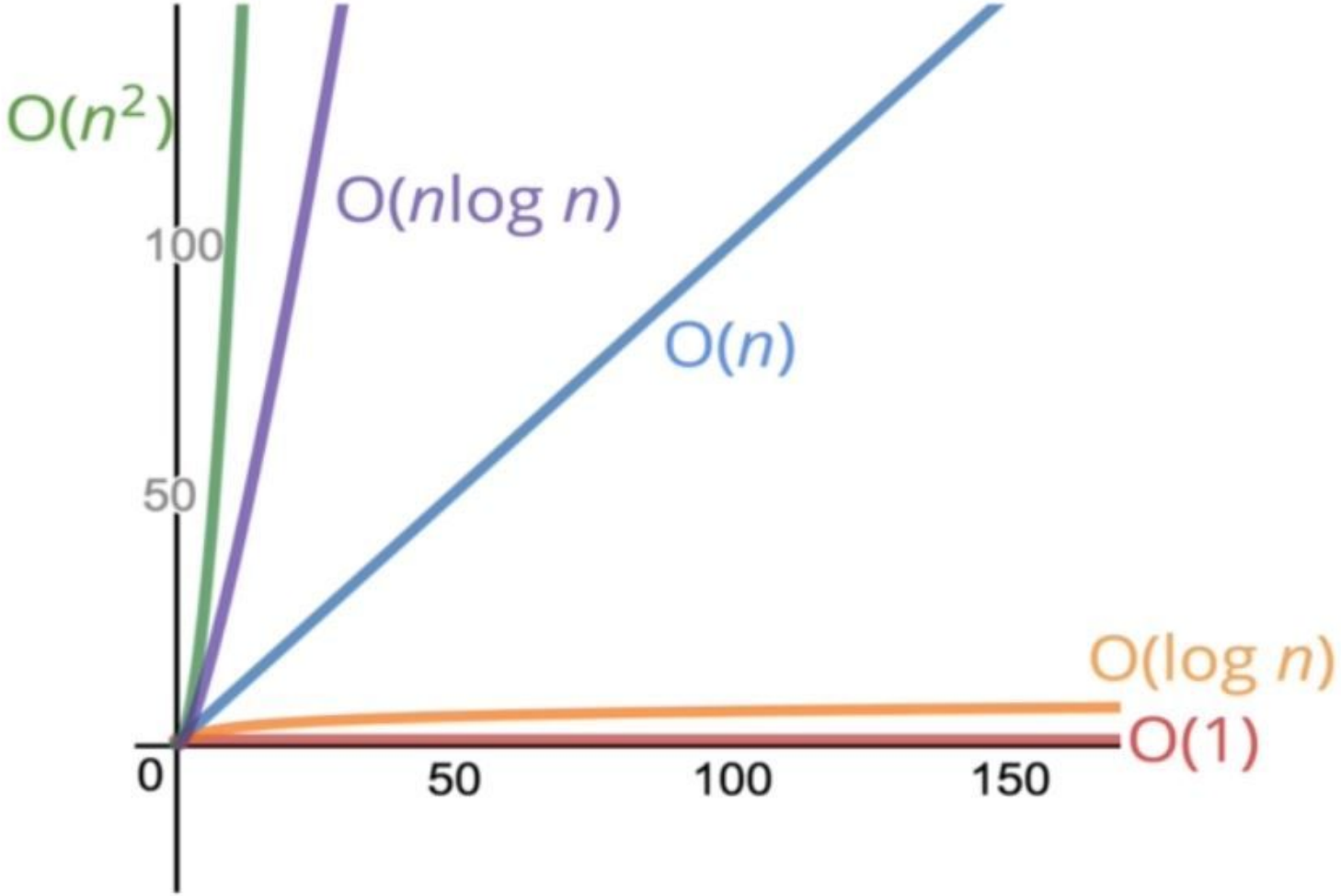
$O(a^n)$ complejidad exponencial (a>=2) Ej $O(2^n)$

O(n!) = complejidad factorial = explosión combinatoria.



CUANTO MÁS BAJEMOS PEOR ES EL ALGORITMO


o = 1
-10
10
o = n
o = log(n)
o = n log(n)
o = n^2
o = 2^n
o = n!
O(n!)
O(2^n)
O(n^2)
O(nlog(n))
O(n)
O(log n)
campus MVP.es
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10


O(n^2)
O(nlog n)
O(n)
O(log n)
O(1)
100
50
0
50
100
150

Resolvemos el ejercicio del envío del mensaje

El envío por avión tiene complejidad O(1), siempre tarda lo mismo

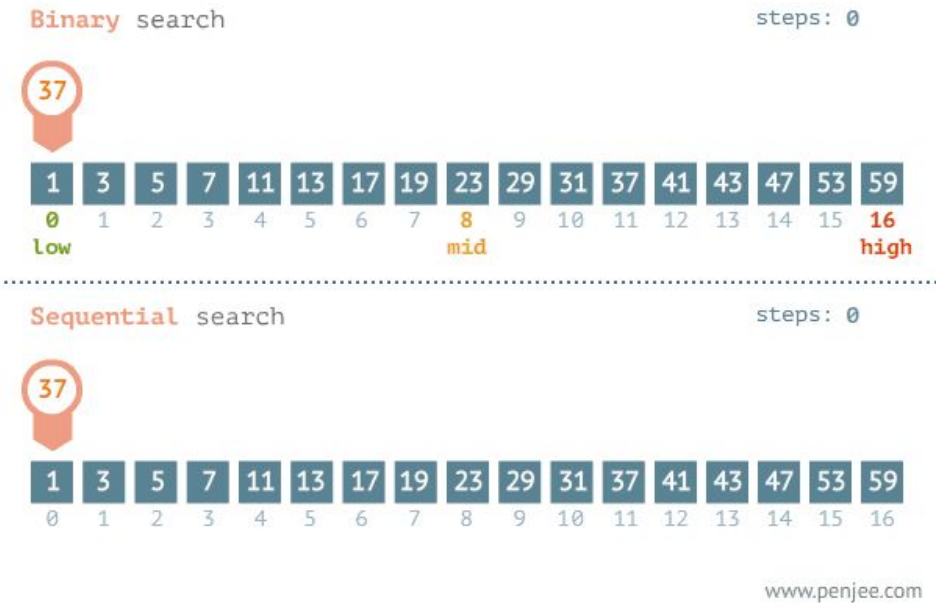
El envío del archivo de forma digital tendrá una complejidad O(n), tiene complejidad lineal, cuanto más grande es el archivo, más tarda en enviarse.

O(1): constante

O(1): constante. La operación no depende del tamaño de los datos. Es el caso ideal, pero a la vez probablemente el menos frecuente.

O(log n): logarítmica

O(log n): logarítmica. por regla general se asocia con algoritmos que "trocean" el problema para abordarlo, como por ejemplo una búsqueda binaria.



O(log n)=logaritmica

O(n)=lineal

O(n): lineal.

O(n): lineal. El tiempo de ejecución es directamente proporcional al tamaño de los datos. Crece en una línea recta.

Cuanto más crecen los datos a tratar, mas cree la complejidad.

O(n * log n): cuasi lineal

O(n* log n): en este caso se trata de funciones similares a las anteriores, pero que rompen el problema en varios trozos por cada elemento, volviendo a recomponer información tras la ejecución de cada "trozo".

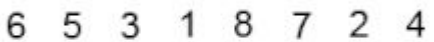
Por ejemplo, el algoritmo de orden Quicksort.

$O(n^2)$: cuadrática.

$O(n^2)$: cuadrática. Es típico de algoritmos que necesitan realizar una iteración por todos los elementos en cada uno de los elementos a procesar.

Por ejemplo el algoritmo de ordenación de burbuja. Si tuviese que hacer la iteración más de una vez serían de complejidad O(n3), O(n4), etc... pero se trata de casos muy raros y poco optimizados.

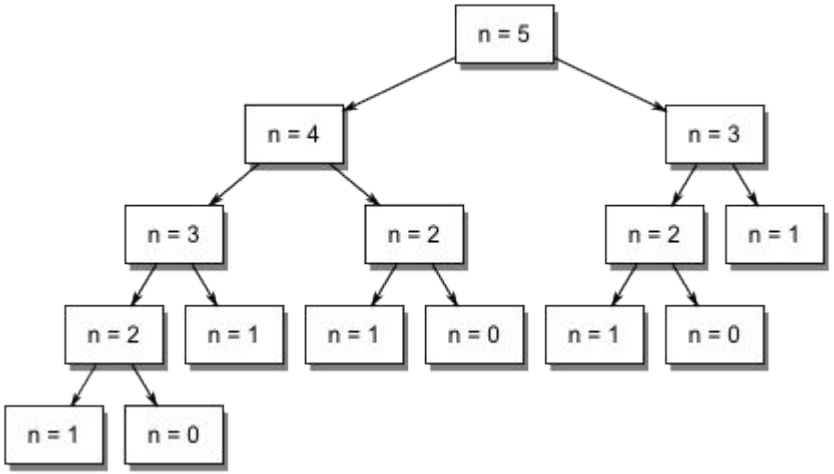
Esta complejidad se da normalmente cuando usamos for anidados.



$O(2^n)$: exponencial.

$O(2^n)$: exponencial. Se trata de funciones que duplican su complejidad con cada elemento añadido al procesamiento. Son algoritmos muy raros pues en condiciones normales no debería ser necesario hacer algo así.

Un ejemplo sería, por ejemplo, el cálculo recursivo de la serie de Fibonacci, que es muy poco eficiente (se calcula llamándose a sí misma la función con los dos números anteriores: F(n)=F(n-1)+F(n-2)).



O(n!); explosión combinatoria.

O(n!); explosión combinatoria. Un algoritmo que siga esta complejidad es un algoritmo totalmente fallido.

Una explosión combinatoria se dispara de tal manera que cuando el conjunto crece un poco, lo normal es que se considere computacionalmente inviable. Solo se suele dar en algoritmos que tratan de resolver algo por la mera fuerza bruta. No deberías verlo nunca en un software "real".

Términos no dominantes

Siempre vamos a simplificar la eficiencia.

Descartando las constantes y los términos de orden inferior. Ej:

$O(2n) = O(n)$

$O(n/2) = O(n*1/2) = O(n)$

$O(4+n) = O(n)$

$O(n/2 + n^2 + n^3) = O(n^3)$

$O(n+n^2) = O(n^2)$

$O(n+\log(n)) = O(n)$

$O(5*2^n+n^{100}) = O(2^n)$

Ejemplo

```
public void algoritmo(){
        for(int a : datos){
                System.out.println(a);
        }
        for(int b : datos){
                System.out.println(b);
        }
}
```

O(n+n) = O(2n) = O(n) = lineal

Ejemplo

```
public void algoritmo(){
        for(int a : datos){
                for(int b : datos){
                        System.out.println(b);
                }
        }
}
```

$O(n_1 * n_2)$

Si $n_1 = n_2$ = $O(n^2)$ = cuadrática

Ejemplo

O(log n) logarítmica

$$\frac{n}{2^x} = 1 \text{ (parada)}$$

$$2^x = n$$

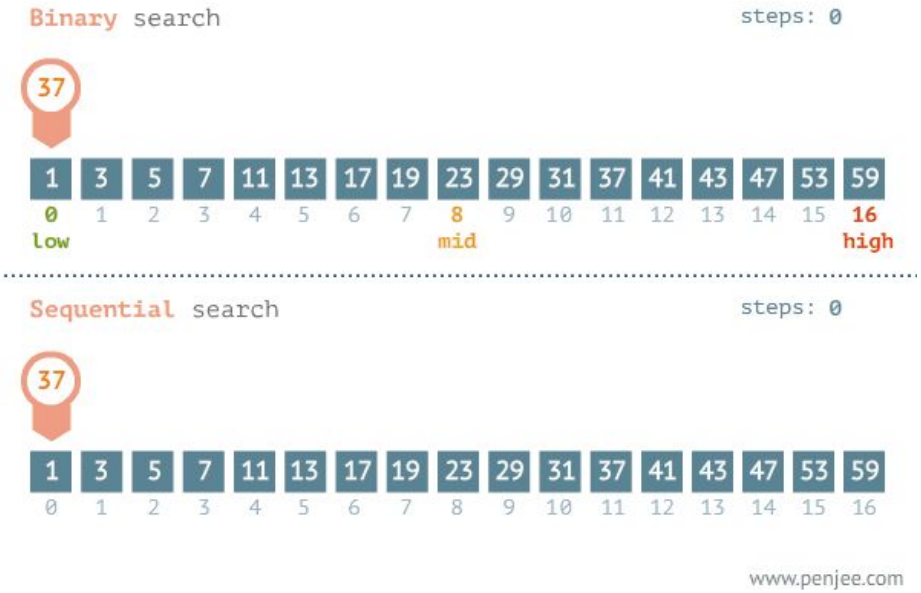
$$\log_2 n = x$$

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos. Funciona al dividir repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una.

Es un requisito que la información esté ordenada.

Puedes ver el algoritmo aquí

https://devs4j.com/2018/02/02/busqueda-binaria-en-java-paso-a-paso/



Ejemplo

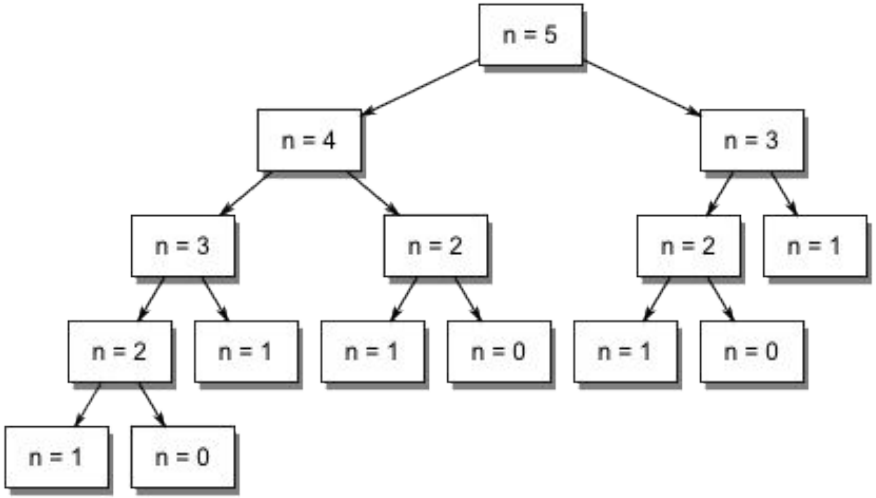
$O(2^n)$ exponencial

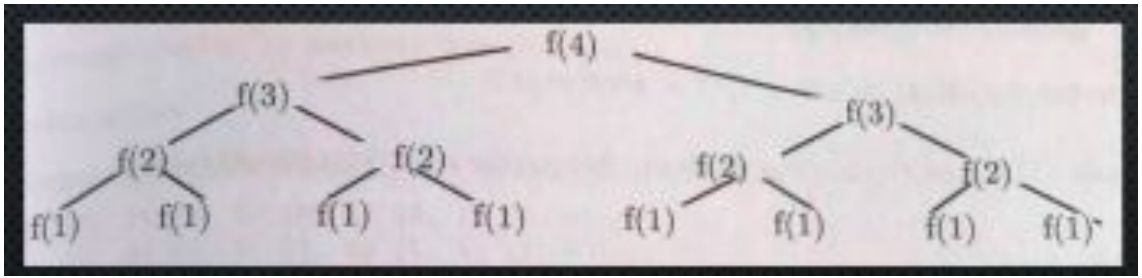
```
public int f(int n){

        if(n<=1) return 1;

        return f ( n-1 ) + f (n+1);

}
```





Ejemplo

El problema de calcular una ecuación de segundo grado

$$ax^2 + bx + c = 0\,, \quad a \neq 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

O(1) constante

Ejemplo: problema viajero

Quieres recorrer la 28 capitales europeas. Pero quieres encontrar el camino más corto para poder recorrerlas todas una sola vez.

Una solución sería probar todas las combinaciones posibles… pero eso sería un **error** ya que hay 304888344611713860501504000000 combinaciones posibles

28! = 304888344611713860501504000000

NOTA: Existen otros algoritmos que resuelven mejor este problema

O(n!) explosión combinatoria

¿Por qué tener en cuenta esto?

- Si trabajamos con dispositivos con pocos recursos (baja memoria, poco procesador)
- Si trabajamos con grandes cantidades de datos
- Muy importante para sistemas en tiempo real
- Si queremos consumir menos energía (batería) o que el dispositivo se caliente menos.
- Si queremos que nuestra aplicación funcione más rápido

Estructuras de datos más comunes

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Algoritmos de ordenación

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)