**Complejidad logarítmica:**

Representa cantidad de **recursos** (temporales) que requiere un algoritmo para resolver un problema y es por ello que permite determinar la **eficiencia** del algoritmo. Los criterios que se emplea para determinar la complejidad algorítmica **no proporcionan medidas absolutas** sino medidas relativas al tamaño del problema.

El tiempo empleado por el algoritmo se mide en pasos, tiene que ser independiente: de la máquina, del lenguaje de programación, del compilador, de cualquier otro elemento hardware o software que influya en el análisis. Para conseguir esta independencia una posible medida abstracta puede consistir en determinar cuantos pasos se efectúan al ejecutarse el algoritmo.

**Complejidad logarítmica - Búsqueda binaria**

Al realizar la búsqueda en un **vector** (array) hay que recorrer el vector, elemento a elemento, hasta encontrar el elemento en cuestión o hasta que se acabe el vector.

La **complejidad** de los algoritmos se calcula teniendo en cuenta el peor caso posible. Para el caso de una búsqueda en un vector de tamaño *n*, el peor caso es que el elemento no se encuentre en el vector, con lo que habría que recorrer los *n* elementos del vector antes de poder decir que el elemento no se encuentra en el mismo. A la complejidad de esta operación de la denomina de orden lineal (O(n)).

El algoritmo de [**búsqueda binaria**](http://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda) (o búsqueda dicotómica) es un algoritmo de búsqueda en vectores ordenados que permite disminuir la complejidad de la búsqueda en dichos vectores.

La intuición detrás de la búsqueda binaria es la misma que cuando se busca en un diccionario; cuando se va a buscar una palabra que empieza por la letra "s", nadie comienza a leerse las palabras que empiezan por la "a", luego la "b", y así en adelante. Normalmente, se abre el diccionario por la mitad y, dependiendo de si la letra inicial por la que hemos abierto es mayor o menor que la que buscamos, descartamos la mitad de las palabras de la enciclopedia y ya solo buscamos en la mitad restante.

Para calcular la complejidad de este algoritmo, inicialmente en número de elementos por analizar es n. Tras la primera división, el número será como mucho n/2 (pues nos hemos quedado con la mitad de elementos); tras la segunda división, el número será como mucho n/4; y así sucesivamente. El peor caso se da cuando el elemento a buscar no se encuentra en el vector (es decir, cuando tras dividir los elementos por analizar nos quedemos con un número menor a 1). Por lo tanto, el número máximo de llamadas a realizar es el menor número *m* tal que: \[\frac{n}{2^m}.

Transformando esta fórmula a un [logaritmo](http://es.wikipedia.org/wiki/Logaritmo) en base 2, tenemos que: \[n < 2^m\] y que \[\log n < m\] es decir, que el número m depende, no del tamaño n del vector, sino del logaritmo de dicho n.

Es por esto que el algoritmo de **búsqueda binaria** tiene una **complejidad de orden logarítmico (O (log n)).**

**Complejidad Exponencial – Fibonacci**

Fibonacci, llamada sucesión de números 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,..., como solución a un problema de reproducción de conejos, satisface la ecuación en recurrencia lineal:

Fn = Fn-1 + Fn-2   
donde F1 = 1 = F2

Pero ¿Cómo de rápido crece la función de Fibonacci? Para responder a esa pregunta. Calcularemos las raíces de la ecuación característica asociada a la ecuación en recurrencia F(n) = F(n-1) + F(n-2).

x2 = x + 1, o sea, las raíces de x2 - x - 1 = 0

Una de ellas, es el llamado número de oro (Golden Ratio), cuyo valor aproximado es 1.61803 y su valor exacto es c = (1+√5)/2. Claramente, se verifica c2 = c + 1 y por tanto, para todo n mayor que dos, también cn = cn-1 + cn-2

O sea, la progresión geométrica {cn} satisface la misma ecuación en recurrencia que la función de Fibonacci F(n) = F(n-1) + F(n-2). Ahora, por inducción Como, c0 = 1 = F(2), c = c1 < 2 = F(3). Obtenemos que cn-2 < F(n). En conclusión, la función de Fibonacci crece, como mínimo, exponencialmente. Ahora, enlazando las dos desigualdades, para toda n se tiene cn-4 < F(n-2) < S(n) y también la función S(n) crece, como mínimo, exponencialmente.

Por tanto, **la programación recursiva de la función de Fibonacci** tiene una complejidad, como mínimo, **exponencial**. Y eso, independientemente, de lo bien que gestione el compilador o intérprete correspondiente la programación recursiva. Veamos a continuación que una programación iterativa, sin embargo, tiene una complejidad mucho mejor.