**BÚSQUEDA BINARIA**

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.

La búsqueda binaria consiste en localizar el término buscado comparándolo con la mediana del conjunto de elementos previamente ordenados, reduciendo así sucesivamente el intervalo de búsqueda.

Está altamente recomendado para buscar en arreglos de gran tamaño y requiere de dos requisitos:

1. Que la lista o arreglo se encuentre ordenada ascendentemente
2. Que los elementos de la lista sean único.

La búsqueda binaria comienza por comparar el elemento del medio del arreglo con el valor buscado. Si el valor buscado es igual al elemento del medio, su posición en el arreglo es retornada. Si el valor buscado es menor o mayor que el elemento del medio, la búsqueda continua en la primera o segunda mitad, respectivamente, dejando la otra mitad fuera de consideración.

***¿PORQUE LA BÚSQUEDA BINARIA ES DE COMPLEJIDAD LOGARÍTMICA?***

La búsqueda tiene una complejidad logarítmica porque el proceso se realiza al estilo de un árbol binario donde la raíz es el elemento medio del arreglo y los hijos vendrían a ser el elemento medio de la primera mitad para la hijo izquierdo y segunda mitad seria el hijo de lado derecho, el subárbol izquierdo o derecho son recorridos de acuerdo a si el valor buscado es menor o mayor que el valor presente en el nodo actual, representando la eliminación sucesiva de los elementos

En el peor de los casos se realizan iteraciones (del ciclo de comparaciones), donde la notación  denota la parte entera por debajo de la función. Esta cantidad de iteraciones es alcanzada cuando la búsqueda alcanza el nivel más profundo del árbol, equivalente a una búsqueda binaria que se reduce a un solo elemento, y en cada iteración, siempre elimina el arreglo más **pequeño** de los dos si no tienen la misma cantidad de elementos.

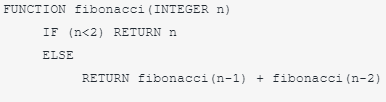
**ALGORITMO DE LA SERIE FIBONACCI**

En [matemáticas](https://es.wikipedia.org/wiki/Matem%C3%A1ticas), la sucesión o serie de Fibonacci es la [sucesión](https://es.wikipedia.org/wiki/Sucesi%C3%B3n_matem%C3%A1tica) infinita de [números naturales](https://es.wikipedia.org/wiki/N%C3%BAmero_natural):

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,…

La sucesión comienza con los números 0 y 1,​ y a partir de estos, «cada término es la suma de los dos anteriores», es la [relación de recurrencia](https://es.wikipedia.org/wiki/Relaci%C3%B3n_de_recurrencia) que la [define](https://es.wikipedia.org/wiki/Definici%C3%B3n_(matem%C3%A1tica)).

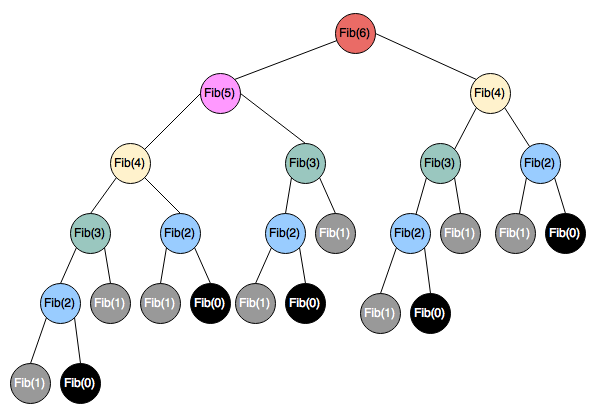
Una primera solución para obtener un término n de la sucesión de Fibonacci será utilizando recursividad:



¿PORQUE TIENE COMPLEJIDAD EXPONENCIAL?

El problema es su complejidad. No sólo en espacio. Ya que todas las llamadas anteriores, a la propia función, han de recordarse, calcularse y asignarse para poder hallar F(n). Sino también en número de operaciones elementales. Que se traduce en complejidad de tiempo. Ya que se suman todas las operaciones elementales necesarias para calcular todos los anteriores.

Analizando a fondo lo que hace esta función. Recursivamente, estará llamando a sus dos anteriores. Cada una de estas llamadas, si es mayor que 1, de nuevo estará llamando a sus dos anteriores. Analicemos el árbol de llamadas para un fibonacci de orden 6:



Entonces para calcular **1** vez Fib(6), es necesario calcular **1** vez Fib(5), **2** veces Fib(4), **3** veces Fib(3), **5** veces Fib(2), **8** veces Fib(1) y cinco veces Fib(0) (Como el algoritmo considera constante Fib(1), solo tenemos 5 llamadas a Fib(0). Si Fib(1) se calculara con su antecesor, tendríamos **13** llamadas). ¡Estamos repitiendo operaciones! Y lo peor, estas operaciones se repiten justamente siguiendo la secuencia de Fibonacci

En conclusión, la función de Fibonacci crece, como mínimo, exponencialmente.