



Notación Big O

Complejidad Algorítmica

01 {

[Complejidad Algorítmica]

- Complejidad Algorítmica
- Notación Big O
- Complejidad en el Tiempo

}

Complejidad Algorítmica {

En el mundo del software, no todo es escribir código y programar como si no hubiese un mañana.

Más allá de entender nuestro objetivo y lo que queremos hacer al programar, existen otras variables que tenemos que tener en cuenta al momento de desarrollar software.

Una de las más importantes, es el **rendimiento** de nuestra aplicación en cuanto a la implementación de un **algoritmo**.

}

Complejidad Algorítmica {

El **rendimiento** de nuestro algoritmo ayudará a que **el usuario no tenga que esperar durante horas a que la tarea**, para la que nuestro software fue desarrollado, **se cumpla**.

Existen varios factores que interfieren al hablar de rendimiento en nuestro algoritmo, como lo son: el uso de espacio en disco, de memoria para su ejecución, y el **tiempo de ejecución** en general.

Una de las herramientas más importantes para determinar el rendimiento de nuestro algoritmo es la **Notación Big O**.

}

Notación Big O {

La **Notación Big O** es una **notación matemática para definir el rendimiento o complejidad de un algoritmo**, podemos determinar su **complejidad en el tiempo** o en el espacio.

Como hoy en día lo que nos sobra es espacio, entonces le daremos más importancia al **tiempo**.

Es una representación relativa de la complejidad de un algoritmo. Con la **Notación Big O** expresamos el **tiempo de ejecución** en términos de: qué tan rápido crece en relación con la **entrada**, a medida que la entrada se hace más grande.

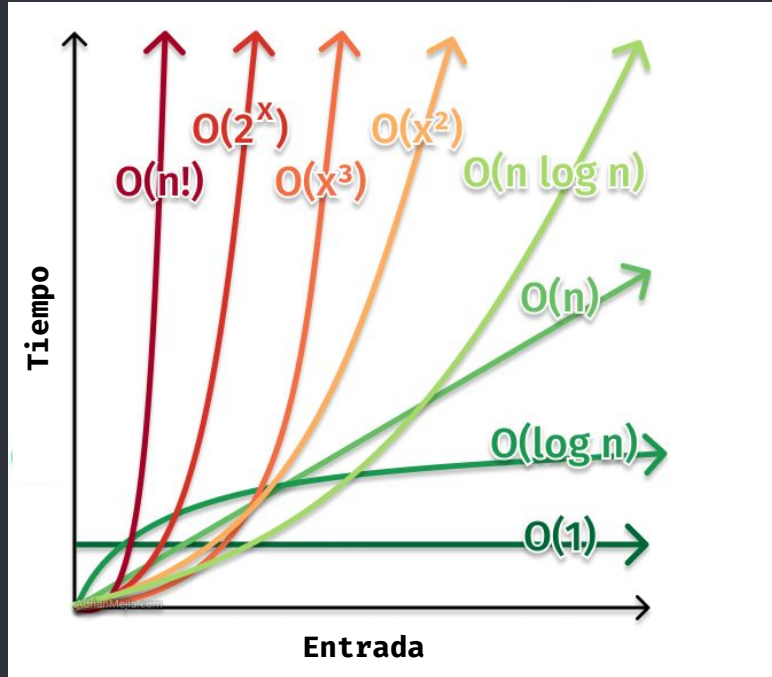
}

Notación Big O {

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**.

}

Complejidad en el Tiempo {



Términos más utilizados {

$O(1)$ → constante.

$O(n)$ → lineal.

$O(\log n)$ → logarítmica.

$O(n^2)$ → cuadrática.

$O(2^n)$ → exponencial.

}

0(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.**

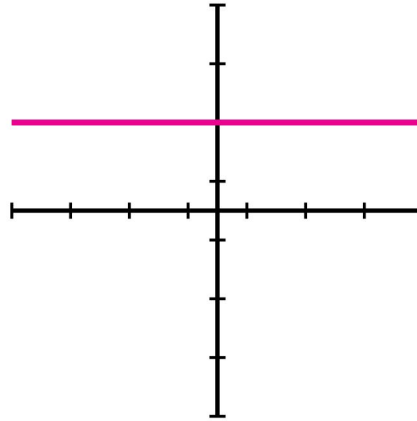
Podemos verlas como **funciones “estáticas”** debido a que siempre se comportarán de la misma manera, no importa las veces que sea ejecutada ni donde.

Un algoritmo con complejidad constante es aquel que siempre toma una misma cantidad de tiempo para su ejecución, es decir que es independiente del número de datos de entrada, es el tipo de complejidad deseada, el caso ideal.

}

$O(1)$ Complejidad Constante {

Función constante



$O(1)$ Complejidad Constante {



```
def saludar():  
    print("Hola")
```



```
def es_mayor(edad):  
    if edad >= 18:  
        return True  
    else:  
        return False
```

}

$O(1)$ Complejidad Constante {



```
def imprimir_primer(lista):  
    primero = lista[0]  
    print(primer)
```

}

$O(n)$ Complejidad Lineal {

Decimos que un algoritmo tiene **complejidad lineal**, 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.

}

$O(n)$ Complejidad Lineal {

Decimos que un algoritmo tiene **complejidad lineal**, 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.

}

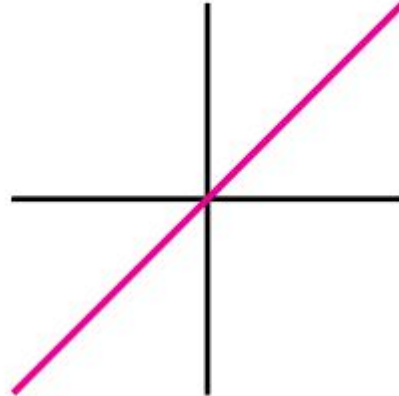
$O(n)$ Complejidad Lineal {

Decimos que un algoritmo tiene **complejidad lineal**, 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.

}

$O(n)$ Complejidad Lineal {

Función lineal



}

$O(n)$ Complejidad Lineal {



```
def imprimirCadena(cadena):  
    for letra in cadena:  
        print(letra)
```

}

$O(n)$ Complejidad Lineal {



```
def imprimirLista(lista):  
    for elemento in lista:  
        print(elemento)
```

}

$O(\log n)$ Complejidad Logarítmica {

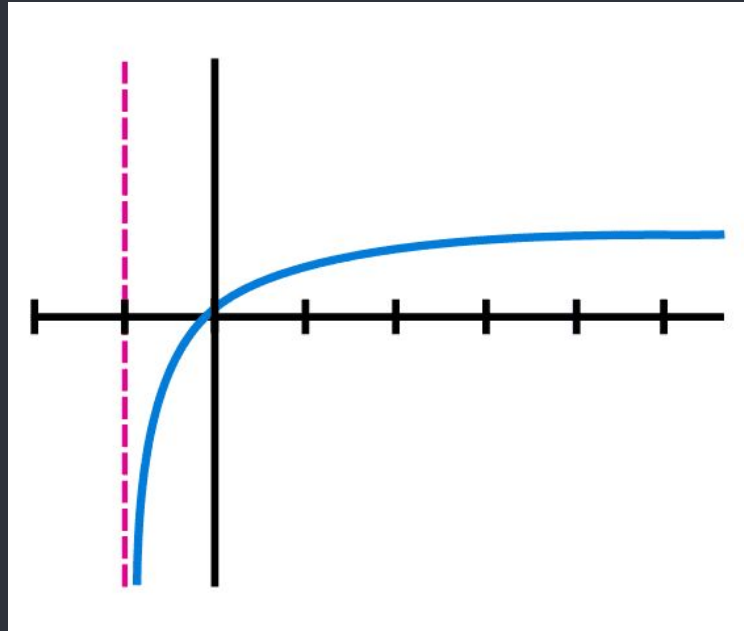
La **complejidad logarítmica** es dada cuando **el tiempo de ejecución o uso de recursos es directamente proporcional al resultado logarítmico del tamaño del valor de entrada.**

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.

Los algoritmos de esta complejidad generalmente la consiguen porque **reducen el tamaño de los datos de entrada con cada paso ejecutado.**

}

$O(\log n)$ Complejidad Logarítmica {



$O(\log n)$ Complejidad Logarítmica {



```
def sucesion_factor_2(numero):  
    i = 1  
    while (i < numero):  
        print(i)  
        i *= 2
```

}

$O(\log n)$ Complejidad Logarítmica {

```
def busqueda_binaria(lista, elemento_buscado):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        print("INICIO: ", inicio)
        print("FIN: ", fin)
        medio = (inicio + fin) // 2 # Division entera para conseguir la mitad
        print("MEDIO: ", medio)
        if lista[medio] == elemento_buscado:
            # Si el elemento medio es el buscado
            print("ENCONTRADO")
            return lista[medio]
        elif lista[medio] < elemento_buscado:
            # Si el elemento buscado es mayor que el medio
            print("Busquemos en la mitad mayor (derecha)")
            inicio = medio + 1
        else:
            # Si el elemento buscado es menor que el medio
            print("Busquemos en la mitad menor (izquierda)")
            fin = medio - 1
    print("NO ENCONTRADO")
    return None
```

$O(n^2)$ Complejidad Cuadrática {

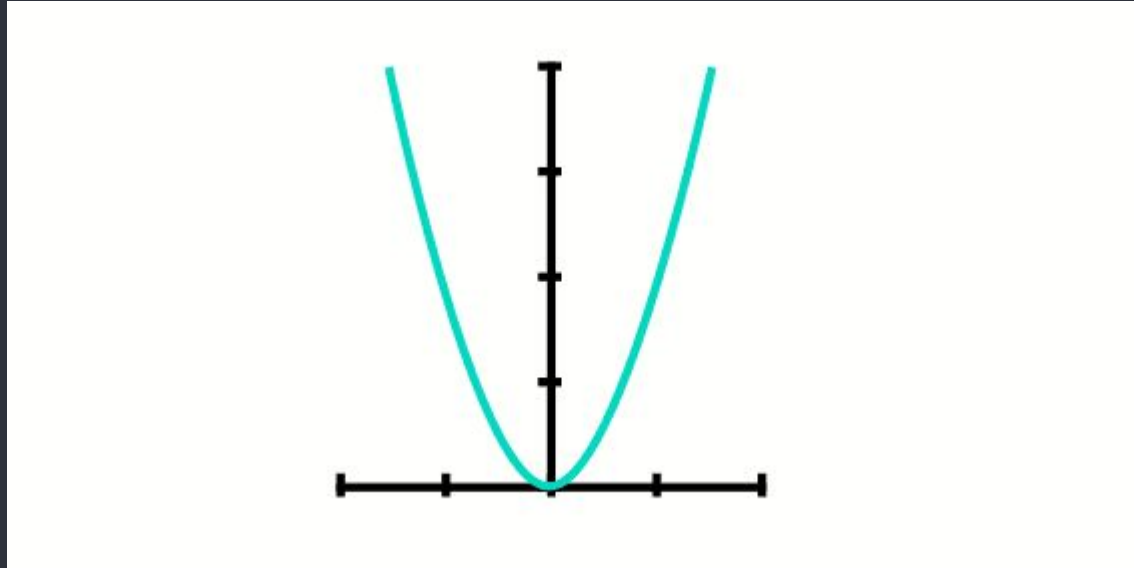
Encontramos **complejidad cuadrática** en los algoritmos, cuando **su rendimiento es directamente proporcional al cuadrado del tamaño del valor de entrada**.

Es decir, si tenemos como dato de entrada una lista 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.


Son aquellos en los que su tiempo crece de **forma cuadrática**, generalmente esto es ocasionado por **ciclos anidados**.

}

$O(n^2)$ Complejidad Cuadrática {



$O(n^2)$ Complejidad Cuadrática {



```
def compararElementos(lista):  
    for x in range(len(lista)):  
        for y in range(len(lista)):  
            if (x != y and lista[x] == lista[y]):  
                print("El elemento {} es igual a {}".format(lista[x], lista[y]))
```

}

$O(2^n)$ Complejidad Exponencial {

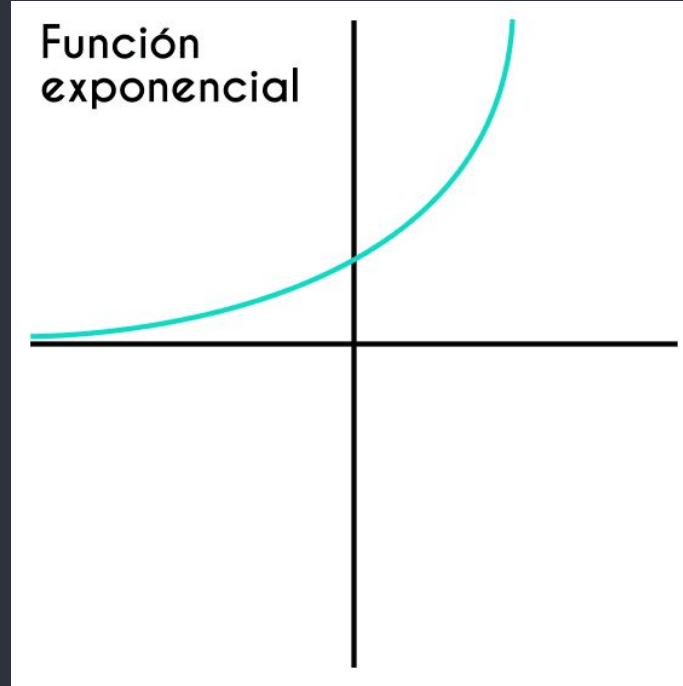
Cuando un algoritmo tiene **complejidad exponencial**, su **rendimiento se incrementa al doble cada vez que se agregue un nuevo dato al valor de entrada**, por ende, incrementando su tamaño de manera **exponencial**.

Esto quiere decir que si tenemos una **lista con 1 elemento** y nos toma **10 segundos** ejecutar el algoritmo, con **2 elementos** nos deberá tomar **100 segundos**, continuando de manera sucesiva.

}

$O(2^n)$ Complejidad Exponencial {

1
2
3
4
5
6
7
8
9
10
11
12
13 }
14



$O(2^n)$ Complejidad Exponencial {



```
def fibonacci(numero):  
    if (numero <= 1):  
        return numero  
    return fibonacci(numero - 1) + fibonacci(numero - 2)
```

}

$O(2^n)$ Complejidad Exponencial {

0	1	1	2	3	5	8	13	21
34	55	89	144	233	377			
610	987	1597	2584	...				

}

$O(2^n)$ Complejidad Exponencial {

1
2
3
4
5
6
7
8
9
10
11
12
13
14

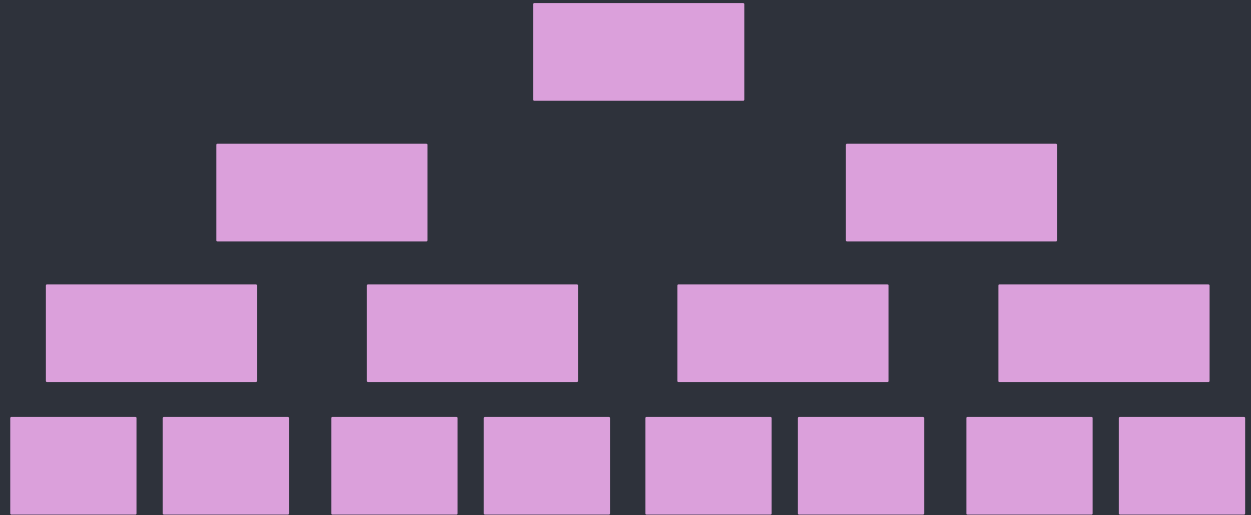
$$2^0=1$$

$$2^1=2$$

$$2^2=4$$

$$2^4=8$$

}





Búsqueda

Algoritmos de Búsqueda

Decorative footer elements consisting of several horizontal bars in white, red, and cyan at the bottom of the slide.

02 {

[Búsqueda]

- Búsqueda Lineal
- Búsqueda Binaria

}

Búsqueda Lineal {

La **búsqueda lineal o secuencial** es la forma más intuitiva de buscar un elemento en una lista.

La lógica es la siguiente, se tiene una **lista de n elementos**, y se quiere **obtener el índice de un elemento buscado**.

Para esto podemos realizar un **ciclo for que recorra la lista elemento por elemento**, y dentro de este **ciclo for** tener un **condicional if** que nos evalúe **si el elemento actual es el elemento que buscamos**.

Es un **método para encontrar un elemento dentro de una lista**. Comprueba secuencialmente cada elemento de la lista hasta que se encuentra una coincidencia o se ha buscado en toda la lista.

}

Búsqueda Lineal {

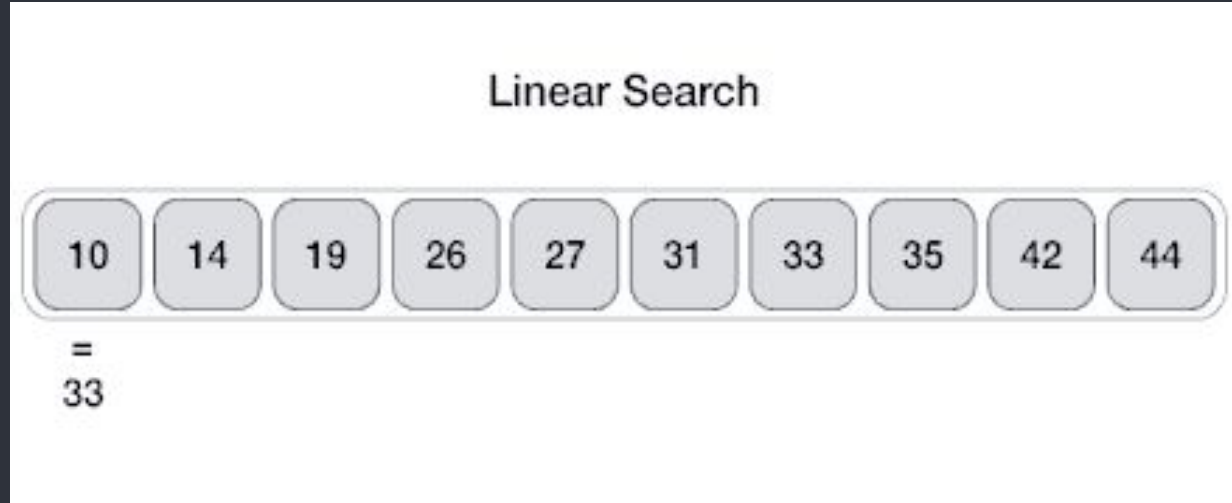
Esta es la **forma más intuitiva** de realizar una búsqueda sobre una **lista**, sin embargo, **no es muy eficiente**.

Si se tiene un arreglo de **10.000 elementos**, se tendría que revisar cada uno para poder encontrar el que buscamos, en el peor de los casos esto nos puede tomar **10.000 iteraciones**.

En la **búsqueda lineal**, buscamos el elemento de destino en la lista **en orden secuencial** uno por uno **desde el primer elemento de la lista hasta el último**.

}

Búsqueda Lineal {



}

Búsqueda Lineal {

Mejor caso

El valor a encontrar está en la primera posición de la lista.

Peor de caso

El valor a encontrar está en la última posición de la lista, o no se encuentra en la lista.

Cuándo se debe usar

- Cuando la lista **no está ordenada**.
- Cuando la lista es **pequeña**.

}

Búsqueda Lineal {



```
def busqueda_binaria(lista, elemento_buscado):  
    elemento_encontrado = None  
    for elemento in lista:  
        if elemento == elemento_buscado:  
            elemento_encontrado = elemento  
    return elemento_encontrado
```

}

Búsqueda Binaria {

La **búsqueda binaria**, también conocida como **búsqueda de medio intervalo**, es un algoritmo de búsqueda que **encuentra la posición de un valor buscado dentro de una lista ordenada**.

La **búsqueda binaria** compara el valor buscado con el **elemento intermedio** de la lista.

Si no son iguales, se elimina la mitad en la que el objetivo no puede estar y **la búsqueda continúa en la mitad restante**, tomando nuevamente el **elemento del medio** para compararlo con el valor objetivo.

}

Búsqueda Binaria {

Este es un algoritmo **recibe dos parámetros:** una **lista ordenada**, y un **elemento a buscar**.

Es importante resaltar que **la búsqueda binaria sólo puede hacerse si la lista está ordenada**.

En pocas palabras, este algoritmo de búsqueda aprovecha una colección de elementos que ya está ordenada al ignorar la mitad de los elementos después de una sola comparación.

}

Búsqueda Binaria {

El algoritmo comienza por comparar el valor que se busca con el **elemento central** de la lista.

- Si el valor que se busca es igual al elemento central, el algoritmo ha **terminado**.
- Si el valor que se busca es menor al elemento central, el algoritmo **busca el valor en la mitad inferior** de la lista.
- Si el valor que se busca es mayor al elemento central, el algoritmo **busca el valor en la mitad superior** de la lista.

El algoritmo **continúa dividiendo la lista por la mitad hasta que encuentra el valor que se busca** o hasta que la lista esté **vacía**.

Búsqueda Binaria {

Search for 47

0	4	7	10	14	23	45	47	53
---	---	---	----	----	----	----	----	----

}

Búsqueda Binaria {

Mejor caso

El valor buscado está en la posición media (mitad) de la lista.

Peor de caso

El valor buscado se encuentra en la primera o última posición de la lista, o no se encuentra en la lista.

Cuándo se debe usar


- Cuando la lista **está ordenada**.
- Cuando la lista es **grande**.

}

Búsqueda Binaria {

```
def busqueda_binaria(lista, elemento_buscado):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        print("INICIO: ", inicio)
        print("FIN: ", fin)
        medio = (inicio + fin) // 2 # Division entera para conseguir la mitad
        print("MEDIO: ", medio)
        if lista[medio] == elemento_buscado:
            # Si el elemento medio es el buscado
            print("ENCONTRADO")
            return lista[medio]
        elif lista[medio] < elemento_buscado:
            # Si el elemento buscado es mayor que el medio
            print("Busquemos en la mitad mayor (derecha)")
            inicio = medio + 1
        else:
            # Si el elemento buscado es menor que el medio
            print("Busquemos en la mitad menor (izquierda)")
            fin = medio - 1
    print("NO ENCONTRADO")
    return None
```

Búsqueda Binaria {



```
def busqueda_binaria_recursiva(lista, elemento_buscado, inicio, fin):
    print(lista[inicio:fin + 1])
    if inicio > fin:
        return None
    medio = (inicio + fin) // 2 # Division entera para conseguir la mitad
    if lista[medio] == elemento_buscado:
        # Si el elemento medio es el buscado
        print("ENCONTRADO")
        return lista[medio]
    elif lista[medio] < elemento_buscado:
        # Si el elemento buscado es mayor que el medio
        print("Busquemos en la mitad mayor (derecha)")
        return busqueda_binaria_recursiva(lista, elemento_buscado, medio + 1, fin)
    else:
        # Si el elemento buscado es menor que el medio
        print("Busquemos en la mitad menor (izquierda)")
        return busqueda_binaria_recursiva(lista, elemento_buscado, inicio, medio - 1)
```