

# Taller 1 Big - O

Camila Andrea Galindo Ruiz - 506222700

Miguel Daniel Ruiz Silva - 506222719

Jose David Ramirez Beltran - 506222723

16 de septiembre de 2023

## 1. Introducción

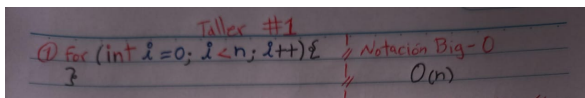
En la programación y el análisis de algoritmos, entender la complejidad temporal es esencial para evaluar el rendimiento y la eficiencia de los programas. La complejidad temporal se refiere a cuánto tiempo lleva ejecutar un algoritmo en función del tamaño de entrada. Una forma común de expresar la complejidad temporal es mediante la notación Big O.

En el análisis se examinaron varios fragmentos de código y se evaluó su complejidad temporal en términos de la notación Big O. Cada código se desglosó en sus operaciones individuales, y se calcularon los tiempos necesarios para realizar estas operaciones en función del tamaño de entrada. A partir de esto, se determinaron las complejidades temporales de los algoritmos presentados.

El análisis presentado destaca la importancia de elegir algoritmos y enfoques adecuados para tareas específicas. La complejidad temporal puede variar significativamente según el enfoque utilizado, y es fundamental seleccionar el algoritmo más eficiente para el problema en cuestión.

## 2. Código 1

```
for (int i = 0; i < n; i++) { // O(n)
}
```

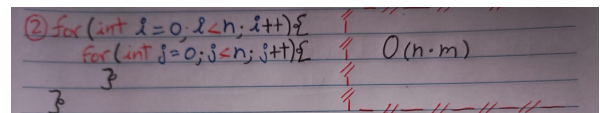


- La declaración de la variable  $i$  y la inicialización de su valor a 0 tienen una complejidad constante de  $O(1)$ .

- La condición del bucle  $i < n$  se evalúa  $n + 1$  veces, lo que tiene una complejidad de  $O(n)$ .
- El incremento de la variable  $i$  en cada iteración del bucle también tiene una complejidad de  $O(n)$ .
- Por lo que se puede decir que la complejidad total del bucle for es  $O(n)$ .

## 3. Código 2

```
for (int i = 0; i < n; i++) { // O(n)
    for (int j = 0; j < m; j++) { // O(m)
    }
}
```

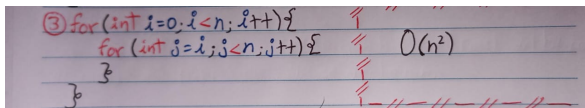


- La declaración de las variables  $i$  y  $j$  y la inicialización de sus valores en 0 tienen una complejidad constante de  $O(1)$ .
- La condición del bucle externo  $i < n$  son evaluadas  $n + 1$  veces, lo que tiene una complejidad de  $O(n)$ .
- La condición del bucle interno  $j < m$  se evalúa  $m + 1$  veces en cada iteración del bucle externo, lo que tiene una complejidad de  $O(m)$ .
- El incremento de las variables  $i$  y  $j$  en cada iteración de sus bucles también tienen una complejidad de  $O(n)$  y  $O(m)$ .

- Por esto la complejidad total del código es  $O(n * m)$ .

## 4. Código 3

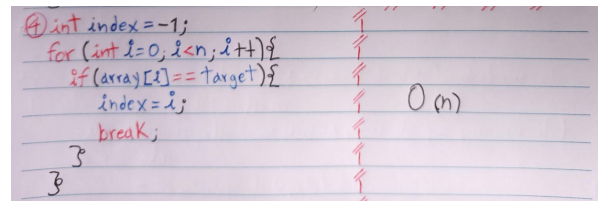
```
for (int i = 0; i < n; i++) { // O(n)
    for (int j = i; j < n; j++) { // O(n - i)
    }
}
```



- La declaración de las variables  $i$  y  $j$  y la inicialización de sus valores en 0 e  $i$ , tienen una complejidad constante de  $O(1)$ .
- La condición del bucle externo  $i < n$  se evalúa  $n + 1$  veces, lo que tiene una complejidad de  $O(n)$ .
- La condición del bucle interno  $j < n$  se evalúa  $n - i + 1$  veces en cada iteración del bucle externo, lo que tiene una complejidad de  $O(n - i)$ .
- El incremento de las variables  $i$  y  $j$  en cada iteración de sus bucles también tienen una complejidad de  $O(n)$  y  $O(n - i)$ .
- Por esto la complejidad total del código es  $O(n^2/2 + n/2) = O(n^2)$ .

## 5. Código 4

```
int index = -1; // O(1)
for (int i = 0; i < n; i++) { // O(n)
    if (array[i] == target) { // O(1)
        index = i; // O(1)
        break; // O(1)
    }
}
```



- La declaración de la variable  $index$  y la inicialización de su valor en  $-1$  tienen una complejidad constante de  $O(1)$ .
- La declaración de la variable  $i$  y la inicialización de su valor en 0 también tienen una complejidad constante de  $O(1)$ .
- La condición del bucle  $i < n$  se evalúa  $n + 1$  veces, lo que tiene una complejidad de  $O(n)$ .
- La comparación  $array[i] == target$  y la asignación  $index = i$  tienen una complejidad constante de  $O(1)$ .
- La instrucción `break` también tiene una complejidad constante de  $O(1)$ .
- Por esto la complejidad total del código es  $O(n)$ .

## 6. Código 5

```
int left = 0, right = n - 1, index = -1; // O(1)
while (left <= right) { // O(log(n))
    int mid = left + (right - left) / 2; // O(1)
    if (array[mid] == target) { // O(1)
        index = mid; // O(1)
        break; // O(1)
    } else if (array[mid] < target) { // O(1)
        left = mid + 1; // O(1)
    } else {
        right = mid - 1; // O(1)
    }
}
```

```

⑤ int left=0, right=n-1, index=-1;
while (left <= right) {
    int mid = left + (right - left) / 2;
    if (array[mid] == target) {
        index = mid;
        break;
    } else if (array[mid] < target) {
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
}

```

- La declaración de las variables *left*, *right* e *index* y la inicialización de sus valores tienen una complejidad constante de  $O(1)$ .
- La condición del bucle while se evalúa hasta que la variable *left* es mayor que la variable *right*.
- En cada iteración del bucle el rango de búsqueda se reduce a la mitad, es decir que el número de iteraciones es proporcional al logaritmo en base 2 del tamaño del rango de búsqueda inicial.
- Por esto, la complejidad del bucle while es  $O(\log(n))$ , donde  $n$  es el tamaño del arreglo *array*.
- La declaración de la variable *mid* y la inicialización de su valor tienen una complejidad constante de  $O(1)$ .
- Las comparaciones *array[mid] == target* y *array[mid] < target*, y también las asignaciones *index = mid*, *left = mid + 1* y *right = mid - 1*, también tienen una complejidad constante de  $O(1)$ .
- La instrucción *break* también tiene una complejidad constante de  $O(1)$ .
- Por esto la complejidad total del código sería  $O(\log(n))$ .

## 7. Código 6

```

int row = 0, col = matrix[0].length - 1, indexRow = -1, indexCol = -1;
while (row < matrix.length && col >= 0) {
    if (matrix[row][col] == target) {

```

```

        indexRow = row;
        indexCol = col;
        break;
    } else if (matrix[row][col] < target) {
        row++;
    } else {
        col--;
    }
}

```

```

⑥ int row = 0, col = matrix[0].length - 1, indexRow = -1, indexCol = -1;
while (row < matrix.length && col >= 0) {
    if (matrix[row][col] == target) {
        indexRow = row;
        indexCol = col;
        break;
    } else if (matrix[row][col] < target) {
        row++;
    } else {
        col--;
    }
}
}

```

- Las líneas `int row = 0, col = matrix[0].length - 1, indexRow = -1, indexCol = -1;` son asignaciones y operaciones de tiempo constante:  $O(1)$ .
- El bucle `while (row < matrix.length && col >= 0)` se ejecuta en función de las dimensiones de la matriz. Supongamos que la matriz tiene "m" filas y "n" columnas. Este bucle se ejecutará en el peor caso hasta que *row* sea igual a "m" y *col* sea igual a 0. Por lo tanto, tiene una complejidad de tiempo de  $O(m + n)$  en el peor caso.
- Dentro del bucle, las comparaciones *matrix[row][col] == target* y *matrix[row][col] < target* son operaciones de tiempo constante:  $O(1)$ .

## 8. Código 7

```
void bubbleSort(int[] array) {
    int n = array.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}
```

7. void bubbleSort (int[] array) {  
 int n = array.length;  
 for (int i = 0; i < n - 1; i++) {  
 for (int j = 0; j < n - i - 1; j++) {  
 if (array[j] > array[j + 1]) {  
 int temp = array[j];  
 array[j] = array[j + 1];  
 array[j + 1] = temp;  
 }  
 }  
 }  
}

Notación: Big:  $O(n(n-1)/2)$

- El primero esta en complejidad  $O(1)$  porque solo se asigna la longitud del arreglo a una variable.
- El primer bucle for itera  $n - 1$  veces ( $n$  es la longitud del arreglo). Por esto su  $O(n)$ .
- El segundo bucle for itera  $n - i - 1$  veces en cada iteración de  $i$ . Hay una comparación y un intercambio que se realizan si se cumple la condicion. En el peor caso, todas estas operaciones se realizan en cada iteración. Lo que nos da una complejidad de  $\frac{n(n-1)}{2}$ . Esto finalmente se simplifica y nos daría  $O(n^2)$
- En el if las operaciones son constantes por lo que no dependen del arreglo esto hace que su complejidad sea de  $O(1)$

la notacion Big-O:

En el peor caso, el arreglo esta completamente desordenado. El bucle externo y el bucle interno se ejecutan completamente, por lo que el numero total de operaciones es  $(n - 1) * (n - 1)$  lo que nos da un Big -  $O(n^2)$ .

## 9. Código 8

```
void selectionSort(int[] array) {
    int n = array.length;
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[minIndex]) {
                minIndex = j;
            }
        }
        int temp = array[i];
        array[i] = array[minIndex];
        array[minIndex] = temp;
    }
}
```

8 void selectionSort (int[] array) {  
 int n = array.length;  
 for (int i = 0; i < n - 1; i++) {  
 int minIndex = i;  
 for (int j = i + 1; j < n; j++) {  
 if (array[j] < array[minIndex]) {  
 minIndex = j;  
 }  
 }  
 int temp = array[i];  
 array[i] = array[minIndex];  
 array[minIndex] = temp;  
 }  
}

Big-O  $O(n^2)$

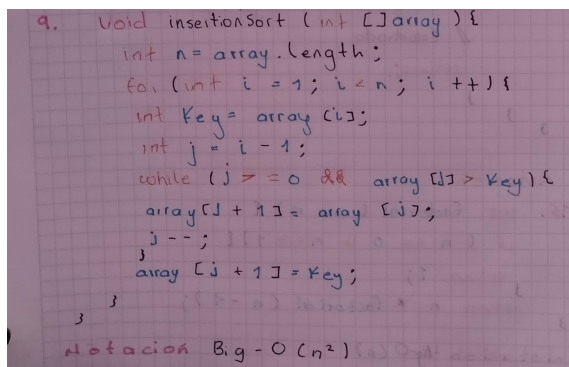
- La línea `int n = array.length;` es una asignación y tiene una complejidad de tiempo constante:  $O(1)$ .
- El primer bucle for (`int i = 0; i < n - 1; i++`) se ejecuta " $n - 1$ " veces, donde  $n$  es el número de elementos en el arreglo. Por lo tanto, tiene una complejidad de tiempo de  $O(n)$  en el peor caso.

- Dentro del primer bucle, se tiene otro bucle `for (int j = i + 1; j < n; j++)`, que también se ejecuta en función del tamaño del arreglo. Este bucle se ejecuta " $n - i - 1$ " veces en cada iteración del bucle externo, donde  $i$  es el índice del bucle externo. Por lo tanto, la complejidad de este bucle interno es  $O(n)$ , pero como está anidado dentro del bucle externo, la complejidad total del bucle interno es  $O(n^2)$ .
- Dentro del bucle interno, hay una comparación `if (array[j] < array[minIndex])`, que es una operación de tiempo constante:  $O(1)$ .
- Luego, se realiza un intercambio de elementos en el arreglo. Estas operaciones de intercambio también son de tiempo constante:  $O(1)$ .

En resumen, el algoritmo tiene una complejidad de tiempo de  $O(n^2)$  donde  $n$  es el número de elementos en el arreglo.

## 10. Código 9

```
void insertionSort(int[] array) {
    int n = array.length;
    for (int i = 1; i < n; i++) {
        int key = array[i];
        int j = i - 1;
        while (j >= 0 && array[j] > key) {
            array[j + 1] = array[j];
            j--;
        }
        array[j + 1] = key;
    }
}
```



9. void insertionSort (int []array) {  
 int n = array.length;  
 for (int i = 1; i < n; i++) {  
 int key = array[i];  
 int j = i - 1;  
 while (j >= 0 && array[j] > key) {  
 array[j + 1] = array[j];  
 j--;  
 }  
 array[j + 1] = key;  
 }  
}

Notación Big-O ( $n^2$ )

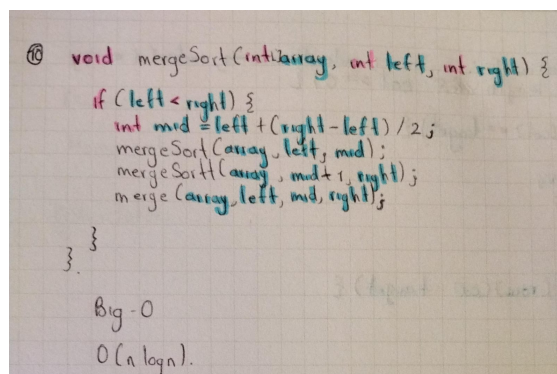
- La complejidad de la segunda línea es de  $O(1)$ , ya que simplemente asigna el valor de la longitud del arreglo a una variable.
- El bucle `for` itera desde 1 hasta  $n-1$ . Por lo tanto, la complejidad de este bucle es  $O(n)$ .
- Las operaciones dentro del bucle `for` son constantes. La asignación `int key = array[i];`, la asignación `int j = i - 1;` y la asignación `array[j + 1] = key;` tienen todas complejidad  $O(1)$ .
- El bucle `while` en el peor caso, se ejecuta hasta que  $j$  llega a  $-1$ , lo que significa que realiza un máximo de  $i$  iteraciones. Dado que  $i$  puede variar desde 1 hasta  $n - 1$ , el bucle `while` se ejecuta en total  $1 + \dots + (n - 1)$  veces, que es proporcional a  $\frac{n*(n-1)}{2}$ , es decir,  $O(n^2)$ .

La notación Big-O:

La complejidad total del algoritmo en el peor caso es  $(n^2)$  debido a que el bucle `while` puede ejecutarse hasta  $\frac{n*(n-1)}{2}$  veces en total.

## 11. Código 10

```
void mergeSort(int[] array, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(array, left, mid);
        mergeSort(array, mid + 1, right);
        merge(array, left, mid, right);
    }
}
```



10 void mergeSort (int[] array, int left, int right) {  
 if (left < right) {  
 int mid = left + (right - left) / 2;  
 mergeSort (array, left, mid);  
 mergeSort (array, mid + 1, right);  
 merge (array, left, mid, right);  
 }  
}

Big-O  
 $O(n \log n)$ .



- La función mergeSort toma tres argumentos: el arreglo array, el índice izquierdo left, y el índice derecho right. Esta función se llama recursivamente hasta que left sea menor que right. Cuando left es igual o mayor que right, la función se detiene. Por lo tanto, la recursión se realizará hasta que el rango de elementos a ordenar se reduzca a un solo elemento.

```

11. void quickSort ( int [ ] array, int low,
    int high ) {
    if ( low < high ) {
        int pivotIndex = partition ( array, low, high );
        quickSort ( array, low, pivotIndex - 1 );
        quickSort ( array, pivotIndex + 1, high );
    }
}
Notacion: Big-O(n^2)

```

- En cada llamada recursiva, se calcula el índice medio mid como  $\frac{left + (right - left)}{2}$ . Esto se hace en tiempo constante:  $O(1)$ .
- Luego, se realiza una llamada recursiva a mergeSort para ordenar la mitad izquierda del arreglo (`array[left..mid]`) y otra llamada para ordenar la mitad derecha (`array[mid+1..right]`).
- Por último la función merge combina las dos mitades ordenadas en una sola matriz. La complejidad de esta función depende de la longitud total de las dos mitades, que es "`right - left + 1`".
- El tiempo que lleva combinar dos mitades de longitud  $n$  es  $O(n)$ , y dado que el algoritmo se divide en mitades logarítmicamente (se ejecuta  $\log_2(n)$  veces), la complejidad total del merge sort es  $O(n \log n)$ .

- La línea del if verifica si low es menor que high. Esto tiene una complejidad constante  $O(1)$ .
- La línea 3 llama a una función partition de QuickSort que tiene una complejidad de  $O(n)$ , ya que implica recorrer el arreglo para dividirlo en dos subarreglos.
- Las siguientes dos líneas son llamadas recursivas a quickSort para los dos subarreglos generados por la partición. En el mejor caso, el arreglo se divide en mitades iguales en cada nivel, lo que lleva a una profundidad del árbol de recursión de  $\log_2(n)$ . Por lo tanto, la cantidad de trabajo en cada nivel se multiplica por la cantidad de niveles, dando como resultado una complejidad de  $O(n \log n)$ .

Notación Big- O:

La complejidad del algoritmo es en el peor caso, donde la partición siempre genera un subarreglo de tamaño 0 y otro de tamaño  $n - 1$ , la complejidad sería  $O(n^2)$ , lo que puede suceder si la elección del pivote es siempre el elemento más pequeño o más grande en un arreglo ya ordenado.

## 12. Código 11

```

void quickSort(int[] array, int low, int high) {
    if (low < high) { // O(1)
        int pivotIndex = partition(array, low, high); // O(n)
        quickSort(array, low, pivotIndex - 1); // Recursion
        quickSort(array, pivotIndex + 1, high); // Recursion
    }
}

```

## 13. Código 12

```

int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    int dp[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
}

```

```

    }
    return dp[n];
}

```

```

12 int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }

    int [] dp = new int [n + 1];
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}

Big - O
O(n)

```

- La función fibonacci toma un solo argumento, "n", que representa el término de la secuencia de Fibonacci que se desea calcular.
- La primera condición `if (n <= 1)` verifica si `n` es menor o igual a 1. Si es así, la función retorna `n`. Esto es una operación de tiempo constante:  $O(1)$ .
- Se crea un arreglo `dp` de tamaño "`n+1`" para almacenar los valores de la secuencia de Fibonacci. Las operaciones de inicialización `dp[0] = 0` y `dp[1] = 1` son operaciones de tiempo constante:  $O(1)$ .
- En la siguiente línea se utiliza un bucle `for` para calcular los valores de la secuencia de Fibonacci desde el tercer término hasta el "`n`-ésimo" término. El bucle se ejecuta "`n - 1`" veces en total. En cada iteración del bucle, se realiza una operación de asignación `dp[i] = dp[i - 1] + dp[i - 2]`. Esto también es una operación de tiempo constante:  $O(1)$ .

La complejidad de tiempo de la función fibonacci es llevada bajo el bucle `for` que itera "`n - 1`" veces. Por lo tanto, la complejidad de tiempo en el peor caso es  $O(n)$ .

## 14. Código 13

```

void linearSearch(int[] array, int target) {
    for (int i = 0; i < array.length; i++) { // O(n)
        if (array[i] == target) { // O(1)
            // Encontrado
            return; // O(1)
        }
    }
    // No encontrado
}

```

```

13. void linearSearch (int [] array, int target) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == target) {
            // Encontrado
            return;
        }
    }
}

Notación: Big - O(n)

```

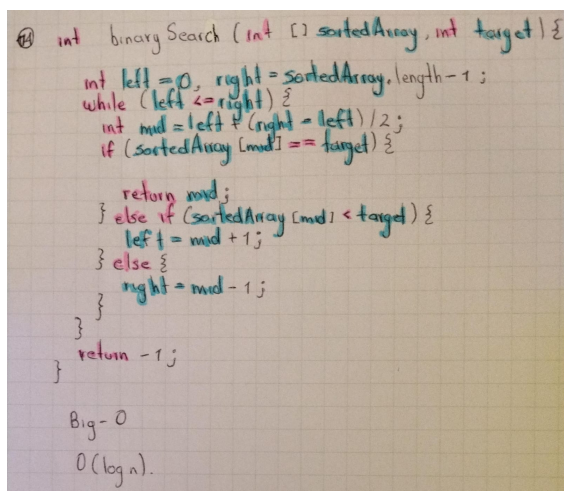
- El bucle `for` itera a través de todo el arreglo con `array.length` elementos. Esto da como resultado una complejidad de  $O(n)$ , ya que la cantidad de iteraciones depende directamente del tamaño del arreglo.
- La comparación `if` verifica si el elemento actual es igual al valor objetivo. Esta operación tiene una complejidad constante  $O(1)$ , ya que no depende del tamaño del arreglo.
- La línea `return;` detiene la función en caso de encontrar el elemento buscado. Esto también tiene una complejidad constante  $O(1)$ .
- En el caso de no encontrar el elemento en todo el arreglo, simplemente se ejecuta la línea `// No encontrado`, la cual también tiene una complejidad constante  $O(1)$ .

Notación Big - O:

La complejidad total del algoritmo es de  $O(n)$  en el peor caso, ya que el bucle itera a través de todo el arreglo, y las operaciones dentro del bucle son de complejidad constante  $O(1)$ .

## 15. Código 14

```
int binarySearch(int[] sortedArray, int target) {
    int left = 0, right = sortedArray.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (sortedArray[mid] == target) {
            return mid; // Índice del elemento encontrado
        } else if (sortedArray[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Elemento no encontrado
}
```



Big - O  
 $O(\log n)$ .

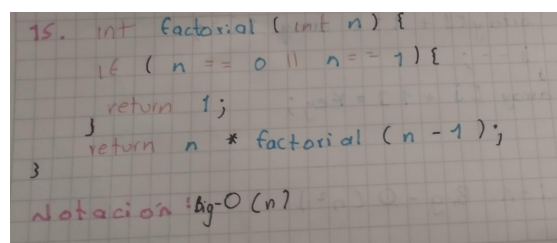
- Las líneas `int left = 0, right = sortedArray.length - 1;` son asignaciones y tienen una complejidad de tiempo constante:  $O(1)$ .
- El bucle `while (left <= right)` se ejecuta mientras la condición `left <= right` sea verdadera. En cada iteración, el tamaño del rango en el que se está buscando se reduce a la mitad, ya que se actualizan los valores de `left` o `right` en función de si el elemento buscado es menor o mayor que el elemento en la posición `mid`.
- Dentro del bucle, se calcula el valor medio `mid` del rango de búsqueda actual. Esto se hace en tiempo constante:  $O(1)$ .

- Luego, se compara el en la posición `mid` con el elemento buscado `target`. Esta comparación es de tiempo constante:  $O(1)$ .

En resumen, la complejidad de tiempo de la búsqueda binaria es  $O(\log n)$  en el peor caso, donde  $n$  es la longitud del arreglo ordenado

## 16. Código 15

```
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```



Notación Big - O:  $O(n)$

- La línea del `if` verifica si `n` es 0 o 1. Esta operación tiene una complejidad constante  $O(1)$ , ya que no importa cuál sea el valor de `n`.
- La línea `return 1;` devuelve 1 cuando `n` es 0 o 1. Esto también tiene una complejidad constante  $O(1)$ .
- La línea `return n * factorial(n - 1);` es donde se realiza la recursión. En cada llamada recursiva, se multiplica `n` por el resultado de la función `factorial(n - 1)`. Dado que se realiza una llamada recursiva con un valor `n - 1` y así sucesivamente hasta llegar a `n = 0` o `n = 1`. Esto nos dice que habrá `n` llamadas recursivas en total, cada una realiza una multiplicación. Por lo tanto, la complejidad total de esta parte es  $O(n)$ .

Notación Big - O:



La complejidad total de la función es  $O(n)$ , ya que el número de llamadas recursivas es directamente proporcional al valor de  $n$ , y cada llamada realiza una cantidad constante de trabajo (la multiplicación).

## 17. Conclusiones

En general, la notación Big-O ayuda a comprender cómo el rendimiento de los algoritmos se relaciona con el tamaño de entrada. Es importante considerar la eficiencia del algoritmo en el contexto de los datos con los que se trabaja y las necesidades específicas del problema que se está resolviendo. Esto para verificar la eficiencia del algoritmo, como se pueden optimizar los recursos, estimar el menor tiempo de ejecución, verificar si puede haber una optimización de código, y de más factores que aporten al rendimiento.

## 18. Bibliografía

Jose Manuel A. (14 de Junio de 2016),  
<https://www.campusmvp.es/recursos/post/Rendimiento-de-algoritmos-y-notacion-Big-O.aspx>  
Geremias B. (01, Marzo de 2020)  
<https://github.com/gbaudino/MetodosDeOrdenamiento>  
<https://github.com>