Taller 1 Notación Big-O

Andres Ricardo Caceres Ortiz 506222067
Fundación Universitaria Konrad Lorenz

I. Introducción

En el marco de este taller, se llevará a cabo un análisis exhaustivo de la notación Big O en una serie de códigos proporcionados por el instructor. Estos códigos están escritos en el lenguaje de programación Java y serán transcritos al papel con el fin de facilitar una evaluación más profunda. Durante este proceso, se identificarán y resaltarán las palabras clave, los operadores lógicos y las declaraciones de variables, con el objetivo de lograr una comprensión más precisa y detallada.

II. Desarrollo del análisis

A continuación, se describirá la notación de cada uno de los algoritmos.

II-A. Código Uno

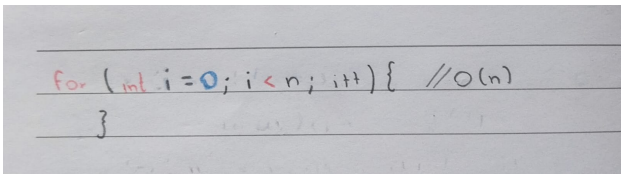

Figura 1. Ejemplo código 1

Este algoritmo incluye un ciclo "for"que se ejecuta n veces, lo que implica que su notación Big O es O(n). En resumen, la notación es O(n).

II-B. Código Dos

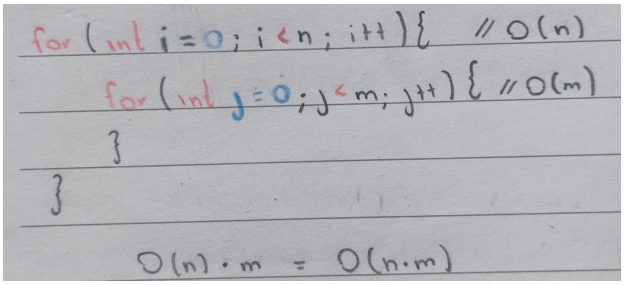

Figura 2. Ejemplo código 2

Este codigo incluye dos ciclos "for.[el] primero de ellos se ejecuta n veces, esto implica que su notacion es O(n). El segundo bucle se ejecuta m veces, es decir su notacion es O(m).Dado que este bucle está anidado dentro del bucle anterior, se ejecutará m veces por cada iteración del bucle exterior. Por lo tanto, en total, se ejecutará n * m"veces. En conclucion la notacion BigO del codigo es O(n*m).

II-C. Código tres

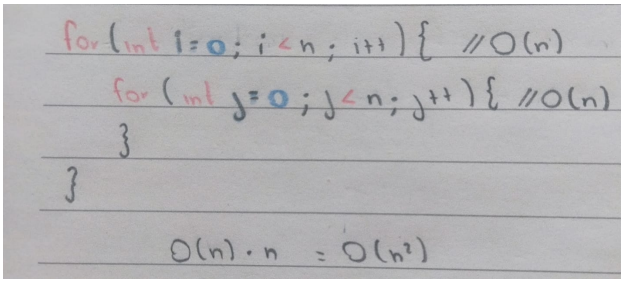

Figura 3. Ejemplo código 3

En este algoritmo, podemos observar que hay dos bucles anidados. Para el primero de ellos, la notación Big O sería O(n), lo que significa que su complejidad crece linealmente con respecto al tamaño de la entrada. Para el segundo bucle anidado, la notación Big O sería O(n)*n. Esto se debe a que el segundo bucle itera sobre la misma entrada que el primero, multiplicando así la complejidad. Por lo tanto, el Big O de este algoritmo es O(n)*n ya que la cantidad total de iteraciones crece cuadráticamente con respecto a n. Notacion BigO= $O(n^2)$.

II-D. Código cuatro

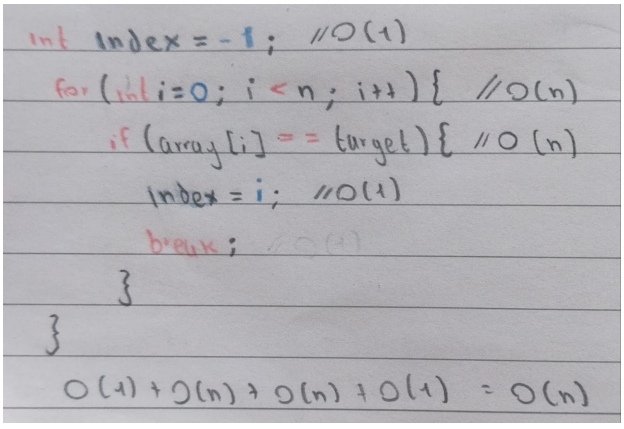

Figura 4. Ejemplo código 4

int index = -1;: Esta línea tiene una complejidad de O(1) porque simplemente declara e inicializa una variable, lo que es una operación de tiempo constante.
for (int i = 0; i <n; i++) : El bucle for ejecuta n iteraciones, por lo que su complejidad es O(n).
if (array[i] == target) : Dentro del bucle, esta línea realiza una operación de comparación entre array[i] y target. En el peor

de los casos, la comparación es O(1), ya que es una operación constante.
index = i;: Esta línea también es una operación de tiempo constante, ya que simplemente asigna un valor a la variable index. Es O(1).
break;: El break simplemente sale del bucle cuando se cumple la condición, lo que es una operación de tiempo constante, O(1).
Big O general para el código en su conjunto sigue siendo O(n) en el peor de los casos debido al bucle principal que recorre los elementos del arreglo.

II-E. Código cinco

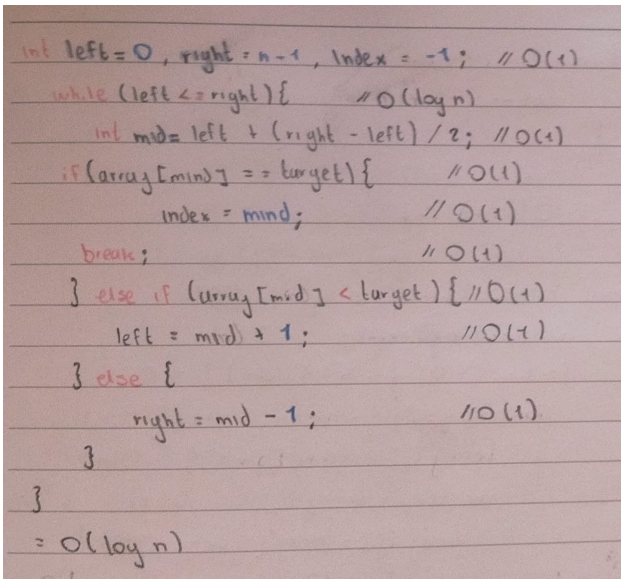

Figura 5. Ejemplo código 5

int left = 0, right = n - 1, index = -1;: Esta línea declara e inicializa tres variables. Lo que se realiza en tiempo constante, por lo que es O(1).
while (left <= right) : Aquí comienza un bucle while que se ejecutará hasta que left sea mayor que right. En el peor de los casos, el bucle se ejecutará log(n) veces.
int mid = left + (right - left) / 2;: En cada iteración del bucle, se calcula el valor medio (mid) utilizando una operación de tiempo constante, O(1).
if (array[mid] == target) : Esto es una operación de comparación entre array[mid] y target, lo cual es una operación de tiempo constante, O(1).
index = mid;: Esta línea es una asignación de tiempo constante, O(1).
break;: Salir del bucle es una operación de tiempo constante, O(1).
else if (array[mid] <target) left = mid + 1; : En este caso, se actualiza la variable left en tiempo constante, O(1).
else right = mid - 1; : Similarmente, se actualiza la variable right en tiempo constante, O(1).

La complejidad total en el peor de los casos de este algoritmo es O(log n), lo que significa que el tiempo de ejecución crece de manera logarítmica con respecto al tamaño del arreglo.

II-F. Código número seis

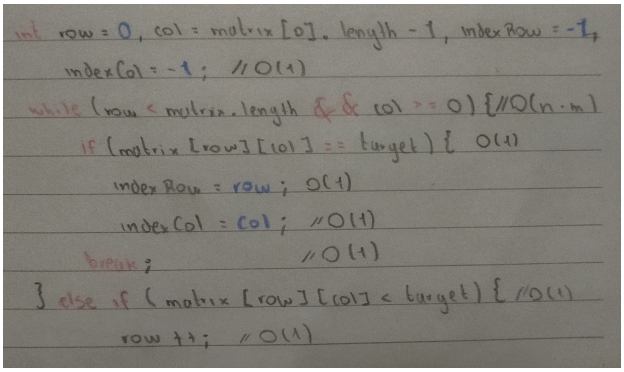

Figura 6. Ejemplo código 6

II-G. Código número siete

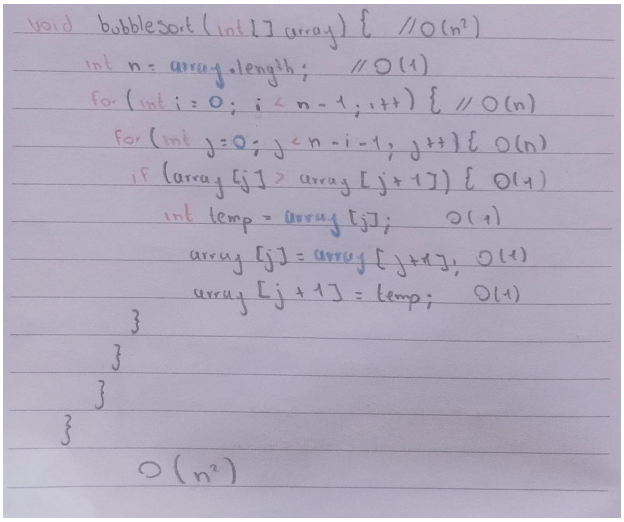

Figura 7. Ejemplo código 7

int n = array.length. La notacion de esta linea es O(1)
for (int i = 0; i <n - 1; i++).Esta linea for se ejecuta n veces, lo que implica que su notación Big O es O(n).
for (int j = 0; j <n - i - 1; j++). Para el segundo bucle anidado, la notación Big O sería O(n)*n
if (array[j] >array[j + 1]). Para esta linea su complejidad es O(1) Por lo tanto, la complejidad total del algoritmo de ordenamiento es $O(n^2), dondeneslalongituddelarreglo.$

II-H. Código ocho

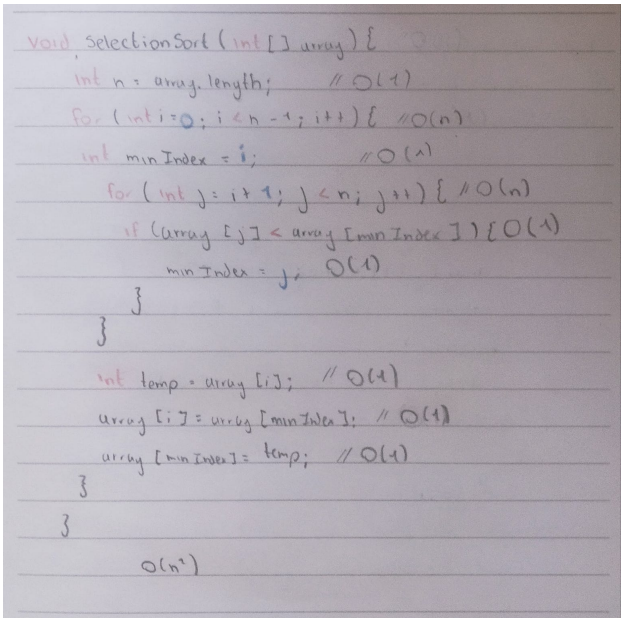

Figura 8. Ejemplo código 8

int n = array.length;: Esto toma tiempo constante para calcular la longitud del array, por lo que es O(1).
for (int i = 0; i <n - 1; i++) : El bucle externo se ejecuta n - 1 veces, donde "n.es la longitud del array. Por lo tanto, esto es O(n) en el peor caso.
int minIndex = i;: Esto es una asignación simple y toma tiempo constante, O(1).
for (int j = i + 1; j <n; j++) : El bucle interno se ejecuta n - i veces en la iteración ï"del bucle externo, donde ï.es el índice actual del bucle externo. En el peor caso, esto suma a una serie aritmética, lo que da como resultado O(n) para todas las iteraciones del bucle interno en total.
for (int j = i + 1; j <n; j++) : El bucle interno se ejecuta n - i veces en la iteración ï"del bucle externo, donde ï.es el índice actual del bucle externo. En el peor caso, esto suma a una serie aritmética, lo que da como resultado O(n) para todas las iteraciones del bucle interno en total.
if (array[j] <array[minIndex]) : La comparación simple toma tiempo constante, O(1).
minIndex = j;: Esto es una asignación simple y toma tiempo constante, O(1).
int temp = array[i];: Esto es una asignación simple y toma tiempo constante, O(1).
array[i] = array[minIndex];: Esto es una asignación simple y toma tiempo constante, O(1).
array[minIndex] = temp;: Esto es una asignación simple y toma tiempo constante, O(1). La complejidad total en el peor de los casos de este algoritmo es $O(n^2)$.

II-I. Código número nueve

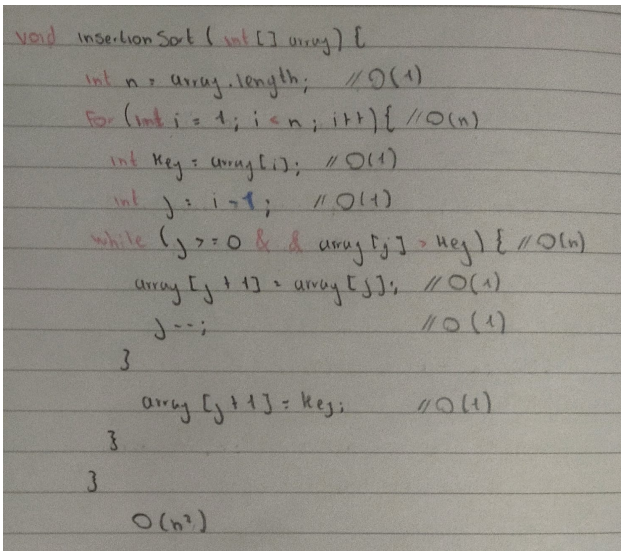

Figura 9. Ejemplo código 9

La primera línea, donde se calcula la longitud del array n, tiene una complejidad de tiempo constante O(1) ya que simplemente se accede a la longitud del array, lo que no depende del tamaño del array.
El bucle externo que recorre el array desde el segundo elemento hasta el último tiene una complejidad de O(n), ya que su número de iteraciones depende directamente del tamaño del array.
Las operaciones dentro del bucle externo, como copiar el valor actual a key, calcular j, y las operaciones dentro del bucle interno, tienen todas una complejidad de tiempo constante O(1), ya que no dependen del tamaño del array.
El bucle interno, que realiza comparaciones y posiblemente desplaza elementos en el array, también tiene una complejidad de O(n) en el peor de los casos, ya que en el peor escenario, puede realizar una inversión completa del array previamente ordenado.
En general, la complejidad temporal del algoritmo de ordenación por inserción es dominada por el bucle externo y el bucle interno en el peor de los casos, lo que resulta en una complejidad total de O(n^2).

II-J. Código número diez

Aquí se explicará el proceso A.

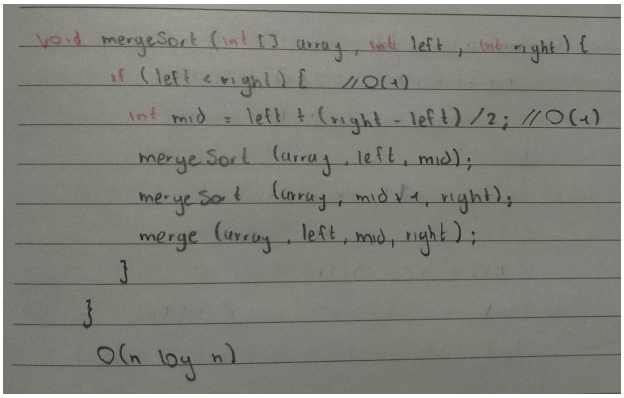

Figura 10. Ejemplo código 10

En la segunda línea, se calcula el punto medio mid utilizando una operación de resta y división. Estas operaciones también tienen una complejidad constante O(1) porque se realizan una vez, independientemente del tamaño del array.
Las dos llamadas recursivas a mergeSort dividen el array en dos mitades: una izquierda y una derecha. Cada llamada recursiva se hace en una mitad del tamaño original, por lo que se dividen en subproblemas de tamaño n/2, donde n es el tamaño original del array. Esto crea un árbol de recursión binario con una altura de log(n). Entonces, la complejidad de estas llamadas recursivas se puede expresar como O(log n).
Finalmente, la llamada a la función de mezcla merge depende del tamaño de las dos mitades que se están fusionando. En el peor caso, si se deben fusionar todos los elementos del array, esta operación tendrá una complejidad de O(n), ya que todos los elementos deben ser comparados y ordenados.
Entonces, en total, la complejidad temporal es O(n log n) en el peor caso.

II-K. Código número once

Aquí se explicará el proceso A.

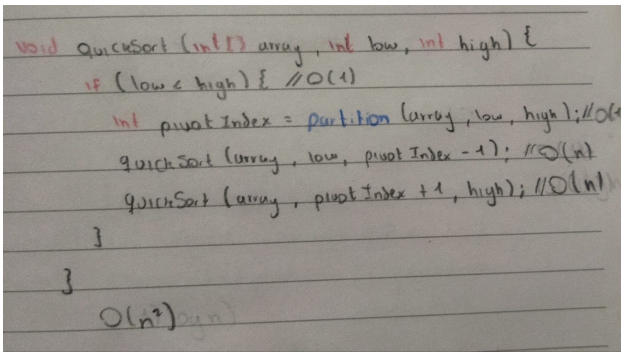

Figura 11. Ejemplo código 11

La primera línea de la función quickSort es una simple comparación if (low <high) que verifica si se debe continuar con la recursión. Esta comparación tiene una complejidad constante O(1) ya que no depende del tamaño del array.
En la segunda línea, se llama a la función partition para determinar la posición del pivote. La complejidad de la función de partición puede variar dependiendo de su implementación, pero en el peor caso, cuando la función de partición no está equilibrada y siempre elige el peor elemento como pivote, puede tener una complejidad cuadrática O(n^2).
$En el peor caso, si la particines desequilibrada, puede llevar a una comple$

II-L. Código doce

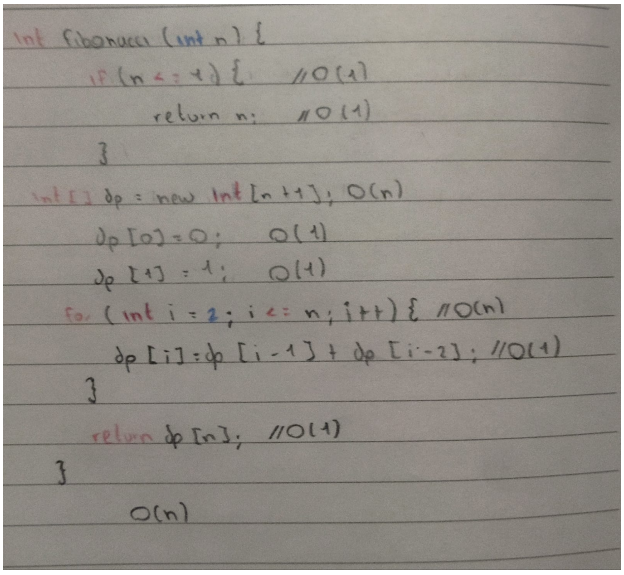

Figura 12. Ejemplo código 12

En resumen, el tiempo de ejecución total del algoritmo es O(n), ya que la parte principal del algoritmo es el bucle que se ejecuta n veces para calcular los números de Fibonacci hasta n y almacena cada resultado en un array. Las demás líneas del código son de tiempo constante y no afectan significativamente el análisis del Big O.

- O(1) + O(1) + O(1) + O(1) + O(1) + O(n) + O(n)
- =O(n) es el peor de los casos.

II-M. Código número trece

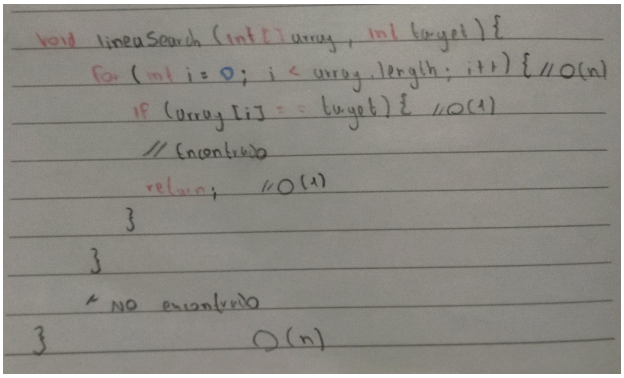

Figura 13. Ejemplo código 13

El bucle for recorre todo el array desde el principio hasta el final. La longitud del array es n, por lo que el bucle se ejecuta n veces, lo que resulta en una complejidad de O(n).

Dentro del bucle, se realiza una comparación simple para verificar si el elemento actual del array es igual al objetivo. Esta comparación es de tiempo constante O(1) ya que no depende del tamaño del array.
Si se encuentra el elemento objetivo, la función return se ejecuta y la función retorna inmediatamente, lo cual también es una operación de tiempo constante O(1)
En resumen, el tiempo de ejecución del algoritmo de búsqueda lineal es O(n) en el peor caso, ya que recorre todos los elementos del array en busca del elemento objetivo. Las demás líneas del código son de tiempo constante y no afectan significativamente el análisis del Big O.

II-N. Código número catorce

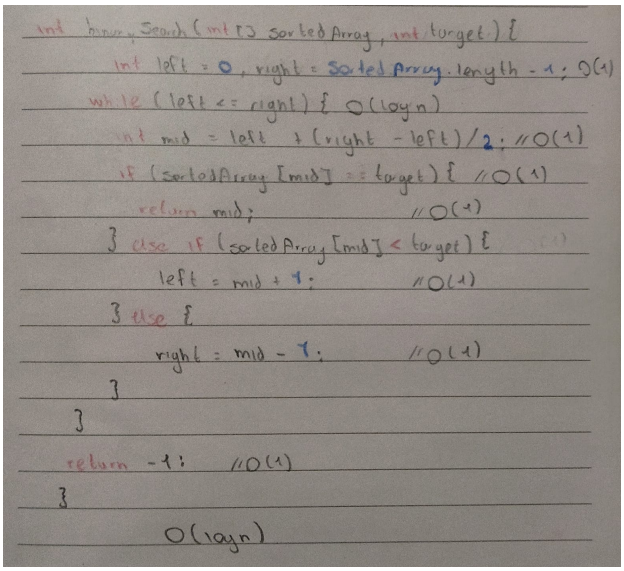

Figura 14. Ejemplo código 14

La función inicia con dos variables, left y right, que representan los límites del intervalo de búsqueda. La inicialización de estas variables es de tiempo constante O(1).
Luego, entra en un bucle while que ejecuta la búsqueda binaria. En cada iteración del bucle, se divide el intervalo en dos mitades y se compara el elemento en el punto medio con el objetivo. Si se encuentra el elemento objetivo, la función retorna inmediatamente. El bucle se ejecuta hasta que left sea mayor que right, lo que significa que se ha revisado todo el intervalo.
Dentro del bucle, todas las operaciones son de tiempo constante O(1) ya que no dependen del tamaño del array.
Si el elemento objetivo no se encuentra después de revisar todo el intervalo, la función retorna -1, lo cual es una operación de tiempo constante O(1).
En resumen, el tiempo de ejecución del algoritmo de búsqueda binaria es O(log n) en el peor caso, ya que en cada iteración, el rango de búsqueda se reduce a la mitad. Las demás líneas del código son de tiempo constante y no afectan significativamente el análisis del Big O

II-Ñ. Código número quince

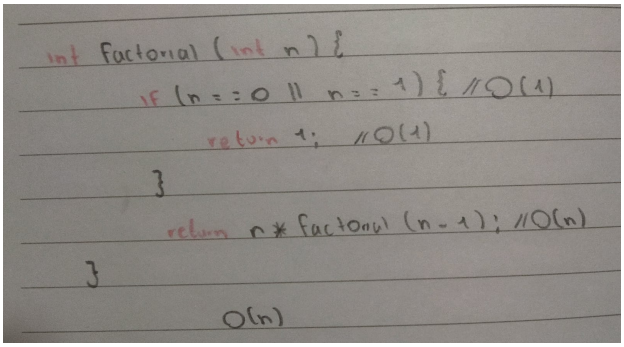

Figura 15. Ejemplo código 15

La comprobación if (n == 0 || n == 1) es una operación de tiempo constante, ya que solo implica verificar si n es igual a 0 o 1. Por lo tanto, esta línea es O(1).
Se realiza una llamada recursiva a factorial(n - 1) y luego se multiplica el resultado por n. La llamada recursiva tiene una complejidad de O(n) en el peor caso, ya que podría realizar hasta n llamadas recursivas antes de llegar al caso base. Por lo tanto, esta línea es O(n) en el peor caso.
En resumen, la complejidad de tiempo total de la función factorial es O(n), ya que la operación de tiempo constante O(1) en la línea 1 es eclipsada por la llamada recursiva O(n) en la línea 3 en términos de complejidad dominante.

- O(1) + O(1) + O(n)
- =O(n) es el peor de los casos.

III. Conclusiones

A lo largo de este documento, hemos explorado el análisis del Big O en diferentes algoritmos y escenarios de código. El Big O es una herramienta esencial para evaluar la eficiencia de un algoritmo y comprender cómo su rendimiento se relaciona con el tamaño de entrada.La notación Big O nos permite expresar la complejidad temporal de un algoritmo en términos de su peor caso en función del tamaño de entrada.Además de analizar algoritmos, el Big O también se aplica al análisis de fragmentos de código, lo que nos permite evaluar la eficiencia de secciones específicas de un programa. Gracias a el analisi se aprende a comparar qué tan rápido son diferentes soluciones para un problema y cuál es la mejor.