

Complejidad de algoritmos Big O Taller 1

Amir Rodriguez Mejia 506222032
Fundación Universitaria Konrad Lorenz

I. INTRODUCCIÓN

En el presente taller vamos a aplicar la complejidad Big O en diferentes algoritmos de programación, con el fin de determinar su notación línea por línea, para hallar el peor de los casos, tomando el proceso o línea que podría consumir más recursos al momento de ejecutarse de manera individual o en conjunto conformando una aplicación. También estaremos ilustrando los códigos de manera escrita con el fin de identificar con colores la asignación de variables de color azul, condicionales de color rojo y el interior del código de color negro, con el fin de reconocer la conformación del mismo.

II. EJERCICIO DE ANÁLISIS DE COMPLEJIDAD DE ALGORITMOS

En esta sección se describirá la metodología utilizada.

II-A. Código número uno

```
Código 1:
for (int i=0; i<n; i++) { // O(n)
}
```

Figura 1. Ejemplo código 1

En el algoritmo solo tenemos un ciclo for, donde su entrada es n, significa que su complejidad es $O(n)$ y a su vez es el peor de los casos.

II-B. Código número dos

```
Código 2:
for (int i=0; i<n; i++) { // O(n)
  for (int j=0; j<m; j++) { // O(m)
  }
}
```

Figura 2. Ejemplo código 2

Dado que en el código hay 2 bucles pero con entradas "n" y "m". es decir que el primero es $O(n)$ y el segundo en su interior es $O(m)$, al ser $n=m$ significa que la complejidad es $O(n^2)$.

- $O(n)+O(n)$
- $=O(n^2)$ es el peor de los casos.

II-C. Código número tres

```
Código 3:
for (int i=0; i<n; i++) { // O(n)
  for (int j=i; j<n; j++) { // O(n)
  }
}
```

Figura 3. Ejemplo código 3

Dado que es un ciclo for anidado donde la entrada del primero controlado por la variable "i", tiene una entrada n, es decir que su complejidad es $O(n)$, lo mismo sucedería con el interno, controlado por la variable j, ya que tiene una entrada "n", es decir que su complejidad es $O(n)$.

- $O(n)*O(n)$
- $=O(n^2)$ es el peor de los casos.

II-D. Código número cuatro

```
Código 4
int index = -1; // O(1)
for (int i=0; i<n; i++) { // O(n)
  if (array[i] == target) { // O(n)
    index = i; // O(1)
    break; // O(1)
  }
}
```

Figura 4. Ejemplo código 4

En el anterior código tenemos una declaración de una variable que se encuentra inicializada en un valor que es constante, es decir que para esta línea su complejidad es $O(1)$. Continuando, hay un ciclo for el cual tiene una entrada n, es decir que su complejidad es $O(n)$ y en su interior hay un condicional if, el cual al no estar en un ciclo anidado que se ejecuta n veces no se aplicaría $O(n^2)$, es decir que la complejidad es $O(n)$. Por último se le asigna el valor de

i a la variable index, siendo una complejidad constante $O(1)$ y el break que tiene una ejecución constante, significa que su complejidad es $O(1)$.

- $O(1) + O(1) + O(1) + 2 O(n)$
- $=O(n)$ es el peor de los casos.

II-E. Código número cinco

```

Código 5 //O(1)
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 { //O(1)
        right=mid-1; //O(1)
    }
}

```

Figura 5. Ejemplo código 5

Dado que en el código la primera línea se encuentra la asignación y declaración de variables, esto se realiza en un tiempo de ejecución constante, lo que equivale a una complejidad $O(1)$. En cuanto al ciclo while, en cada iteración se ajusta el rango de búsqueda, dividiéndolo en dos partes, lo cual es característico de la búsqueda binaria. Esto resulta en una complejidad de aproximadamente $O(\log n)$ debido a la reducción exponencial del rango en función de los datos procesados. La línea de código donde se encuentra la variable mid implica un cálculo constante, lo que también es una operación $O(1)$.

Con respecto al condicional if, es relevante notar que los datos de entrada se manejan de manera constante, llevando a una complejidad de $O(1)$. La asignación de la variable index en la misma línea, así como la instrucción break, ambas presentan una complejidad de $O(1)$.

Luego, la línea de código else if nuevamente se basa en una serie de datos de entrada constantes, lo que resulta en una complejidad $O(1)$. Dentro de esta condición, se lleva a cabo otra asignación de variable, la cual es una operación de tiempo constante. Finalmente, la sección else también presenta una complejidad constante, al igual que la asignación de variable en su interior.

- $O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + \text{Log}(n)$
- $=\text{Log}(n)$ es el peor de los casos.

II-F. Código número seis

```

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

```

Figura 6. Ejemplo código 6

Dado que la primera línea se encuentra vinculada a la asignación de las variables de manera constante, su complejidad es $O(1)$, para luego contar con un ciclo while, el cual esta su condición esta tomando las posiciones en sus filas y columnas, tendiendo una complejidad $O(m+n)$. En el caso del condicional, independientemente del tamaño de la matriz, no se ve alterada por alguna modificación en algún tipo de búsqueda, siendo una complejidad de $O(1)$, en su interior hay asignación de variables como indexRow", indexCol", tienen una complejidad constante $O(1)$, con respecto al break, también presenta una complejidad constante $O(1)$. En el caso la opción else if, no se ve alterada por alguna modificación en algún tipo de búsqueda, siendo una complejidad de $O(1)$, ahora, en su interior hay un incremento de manera constante el y su complejidad es $O(1)$, por último la sección else con su decrementador, tienen una complejidad constante, que equivale a $O(1)$.

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

II-G. Código número siete

```

Código 7
void bubbleSort(int[] array) {
    int n=array.length; //O(1)
    for(int i=0; i<n-1; i++) { //O(n)
        for(int j=0; j<n-i-1; j++) { //O(n^2)
            if(array[j]>array[j+1]) { //O(1)
                int temp=array[j]; //O(1)
                array[j]=array[j+1]; //O(1)
                array[j+1]=temp; //O(1)
            }
        }
    }
}

```

Figura 7. Ejemplo código 7

En el código número 7, primero debemos tener en cuenta la declaración inicial de la variable 'n' como entero. Esta declaración tiene una complejidad constante. Luego, encontramos un ciclo 'for' que recorre 'n' veces un arreglo que posee columnas y filas. Esto da lugar a una complejidad de $O(n)$. Después, en el segundo ciclo 'for', el cual está anidado y cuenta con una variable de incremento, además de depender de las iteraciones del primer ciclo, su complejidad es $O(n^2)$. La siguiente línea de código presenta un condicional que depende de las iteraciones del ciclo 'for' anidado. En otras palabras, su complejidad también es $O(n^2)$.

- $O(1) + O(1) + O(1) + O(1) + O(n^2) + O(n^2)$
- $=O(n^2)$ es el peor de los casos.

II-H. Código número ocho

```

Código 8
void selectionSort (int[] array) {
    int n = array.length; // O(1)
    for (int i = 0; i < n - 1; i++) { // O(n)
        int minIndex = i; // O(1)
        for (int j = i + 1; j < n; j++) { // O(n^2)
            if (array[j] < array[minIndex]) { // O(n^2)
                minIndex = j; // O(1)
            }
        }
        int temp = array[i]; // O(1)
        array[i] = array[minIndex]; // O(1)
        array[minIndex] = temp; // O(1)
    }
}

```

Figura 8. Ejemplo código 8

En el código número 8, encontramos varias operaciones con diferentes complejidades. En primer lugar, tenemos la asignación de un arreglo a una variable entera, lo cual tiene una complejidad constante ($O(1)$). Luego, nos encontramos con un ciclo 'for' que recorre el arreglo según sus filas y columnas, resultando en una complejidad lineal $O(n)$.

En la línea siguiente, observamos la asignación de una variable ('minIndex') que también tiene una complejidad constante ($O(1)$). Sin embargo, en la línea posterior, nos encontramos con un ciclo 'for' anidado. Al depender de las iteraciones del ciclo exterior, esto hace que su complejidad sea cuadrática, es decir, $O(n^2)$.

El mismo patrón se repite en el condicional siguiente. Debido a que sus iteraciones dependen tanto del ciclo externo como del ciclo anidado, la complejidad resulta ser cuadrática $O(n^2)$. Es importante notar que el condicional en su interior también contiene una asignación de variable, lo cual es una operación de complejidad constante ($O(1)$).

En relación a las asignaciones de valores que se encuentran al final del ciclo 'for' exterior, su complejidad sigue siendo constante ($O(1)$), ya que no dependen del tamaño del arreglo.

- $O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(n) + O(n^2) + O(n^2)$
- $=O(n^2)$ es el peor de los casos.

II-I. Código número nueve

```

Código 9
void insertionSort (int[] array) {
    int n = array.length; // O(1)
    for (int i = 1; i < n; i++) { // O(n)
        int key = array[i]; // O(1)
        int j = i - 1; // O(1)
        while (j >= 0 && array[j] > key) { // O(n^2)
            array[j + 1] = array[j]; // O(1)
            j--; // O(1)
        }
        array[j + 1] = key; // O(1)
    }
}

```

Figura 9. Ejemplo código 9

En el código número 9, analizaremos la complejidad de cada componente. En primer lugar, la asignación de un arreglo a la variable 'n' tiene una complejidad constante ($O(1)$), ya que no depende del tamaño del arreglo.

Luego, observamos el ciclo 'for', que recorre el arreglo 'n' veces debido a sus filas y columnas, resultando en una complejidad lineal $O(n)$.

Continuando, encontramos la asignación constante de dos variables, 'key' y 'j', lo cual también tiene una complejidad constante ($O(1)$). Pasamos al ciclo 'while', que se encuentra anidado dentro del ciclo 'for'. Dado que sus iteraciones dependen del ciclo exterior, su complejidad se multiplica. Sin embargo, hay una clarificación necesaria aquí: la complejidad del ciclo 'while' no es cuadrática $O(n^2)$. Su complejidad depende del número de inversiones en el arreglo y, en promedio, su ejecución a lo largo de todas las iteraciones del ciclo 'for'. Esto da lugar a una complejidad promedio menor que $O(n^2)$, pero aún sigue siendo un factor que contribuye a la complejidad total del algoritmo.

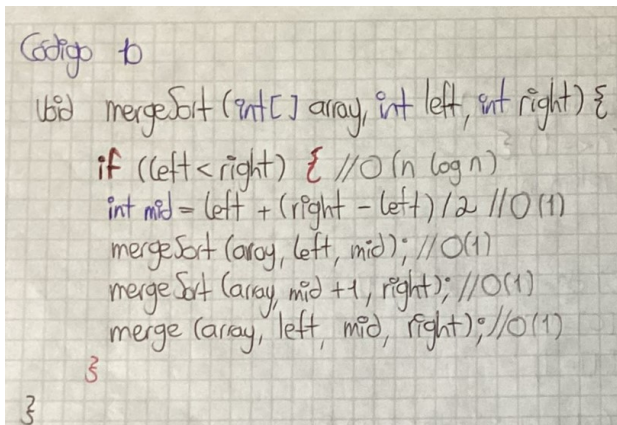
Dentro del ciclo 'while', encontramos una asignación de valores y un decremento de una variable, ambas operaciones con complejidad constante ($O(1)$).

Finalmente, en el ciclo 'for' exterior, hay otra asignación de valores constante en la parte inferior, lo que lleva a una complejidad constante ($O(1)$).

- $O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(n) + O(n^2)$
- $=O(n^2)$ es el peor de los casos.

II-J. Código número diez

Aquí se explicará el proceso A.



```

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

Figura 10. Ejemplo código 10

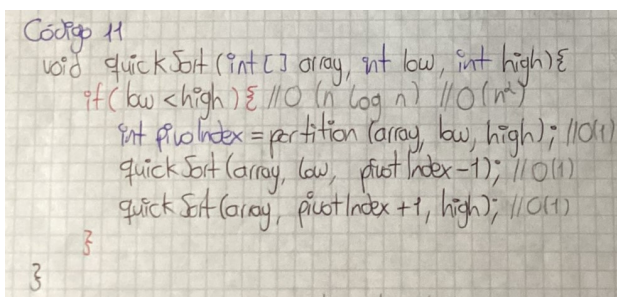
Al examinar el código número 10, podemos analizar la función mergeSort. Su propósito principal radica en dividir una lista de elementos en partes más pequeñas, ordenar esas partes y luego combinarlas para obtener una lista final ordenada. Es importante entender que la complejidad de Merge Sort es logarítmica debido a las divisiones recursivas del arreglo en partes más pequeñas. Cada división reduce el tamaño del problema a la mitad, lo que equivale a una complejidad $O(\log n)$.

La primera línea con la condición `if (left < right)` no contribuye directamente a la complejidad logarítmica. Su función es determinar si el arreglo se puede dividir aún más. Las tres líneas de código siguientes, que realizan operaciones de complejidad constante ($O(1)$), ya que no contribuyen a la modificación del algoritmo en sí. La complejidad logarítmica de Merge Sort se deriva del proceso de división logarítmica y la combinación lineal de las partes ordenadas. La división en sí misma se determina como una complejidad de $O(\log n)$ debido a la reducción exponencial del problema.

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

II-K. Código número once

Aquí se explicará el proceso A.



```

Código 11
void quickSort(int[] array, int low, int high) {
    if (low < high) { // O(n log n) // O(n^2)
        int pivotIndex = partition(array, low, high); // O(1)
        quickSort(array, low, pivotIndex - 1); // O(1)
        quickSort(array, pivotIndex + 1, high); // O(1)
    }
}
  
```

Figura 11. Ejemplo código 11

En el código número 11, encontramos un algoritmo conocido como "QuickSort". Al verificar este algoritmo, descubrimos

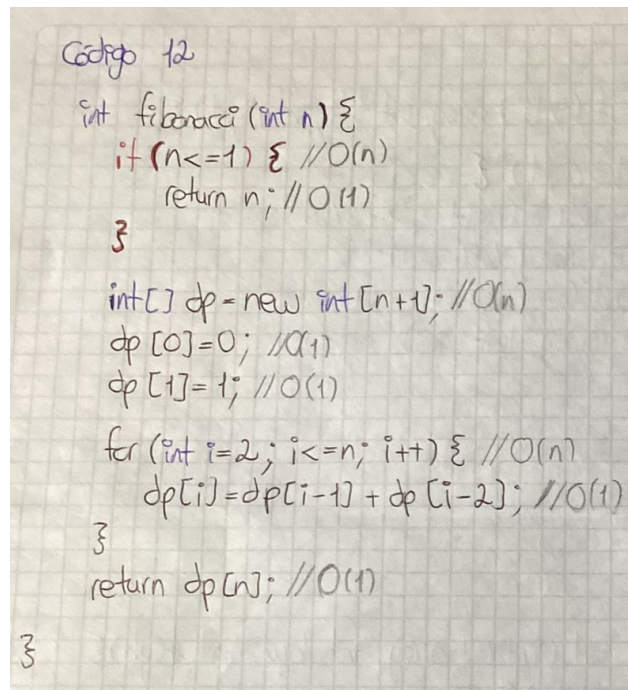
que se trata de otro método de ordenación. Su enfoque reside en la división inteligente de listas o arreglos en partes más pequeñas, permitiendo una reorganización eficiente.

El proceso que realiza QuickSort consiste en seleccionar un elemento llamado "pivote" del arreglo y reorganizar los elementos del arreglo en torno a este pivote. Aquí radica su complejidad, ya que en el caso en que el algoritmo no cuente con mucha eficiencia lógica, su complejidad sería $O(n^2)$. Sin embargo, cuando se optimiza correctamente, su complejidad cambia a $O(n \log n)$.

Debido a esta variabilidad en la complejidad, en el condicional podría concluir que sus complejidades pueden ser 2 entre esas tenemos $O(n \log n)$ hasta $O(n^2)$, dependiendo de la eficiencia de la implementación. Las tres operaciones de reordenamiento en su interior son de complejidad constante y se denotan como $O(1)$.

- $O(1) + O(1) + O(1) + (O(n \log n) \text{ o } O(n^2))$
- $= (O(n \log n) \text{ o } O(n^2))$ es el peor de los casos.

II-L. Código número doce



```

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

    int[] dp = new int[n + 1]; // O(n)
    dp[0] = 0; // O(1)
    dp[1] = 1; // O(1)

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

    return dp[n]; // O(1)
}
  
```

Figura 12. Ejemplo código 12

En el código número doce, se presenta una implementación del cálculo del n -ésimo término de la secuencia de Fibonacci utilizando programación dinámica.

Comenzando por el condicional inicial, se evalúa si el valor de n es igual o menor que 1. Dado que esta operación no depende de n , su complejidad es constante, denotada como $O(1)$. La operación de retorno también es de complejidad constante, ya que simplemente devuelve el valor de n . Por lo tanto, la sección condicional y de retorno en conjunto tiene una complejidad constante.

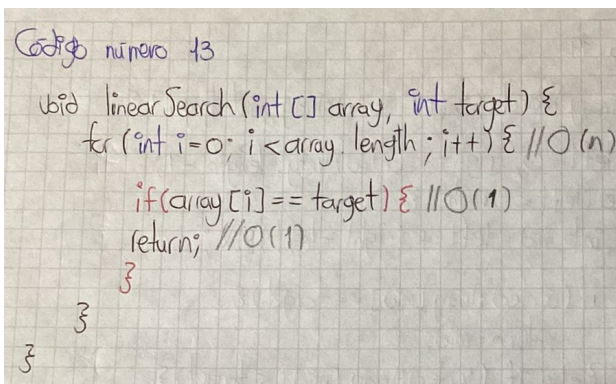
A continuación, se crea un arreglo 'dp' de tamaño $n + 1$. La creación del arreglo lleva tiempo constante, $O(1)$, ya que solo

se realiza una vez y no depende del valor de n . Sin embargo, la inicialización de cada elemento en el arreglo puede equivaler a una complejidad de $O(n)$, ya que se asigna un valor en cada posición del arreglo. Por lo tanto, la creación e inicialización del arreglo en conjunto tiene una complejidad de $O(n)$.

El ciclo 'for' itera desde 2 hasta n . Dado que la iteración ocurre $n - 1$ veces, la complejidad de este ciclo es $O(n)$. Ahora, en el interior del ciclo hay una asignación, la cual es constante y es igual a $O(1)$. Por último, el retorno del valor 'dp[n]' tiene una complejidad constante, ya que simplemente accede al valor almacenado en el arreglo, escrito como $O(n)$

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

II-M. Código número trece



```

Código número 13
void linearSearch(int[] array, int target) {
    for (int i = 0; i < array.length; i++) { // O(n)
        if (array[i] == target) { // O(1)
            return; // O(1)
        }
    }
}

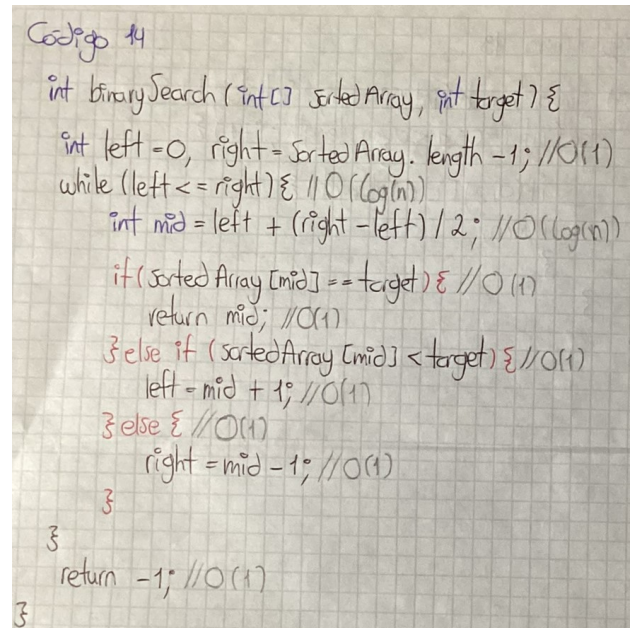
```

Figura 13. Ejemplo código 13

En el código 13, tenemos un ciclo for el cual como parámetro para evaluar su iteración depende de un arreglo, el cual contiene filas y columnas las cuales equivalen a n veces, siendo una complejidad $O(n)$, con respecto al if en su interior no tiene afectación en general con el algoritmo, ya que su función es evaluar de manera constante si se cumple cierta condición, es decir que su complejidad es $O(1)$ igual que el retorno que tiene en su interior.

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

II-N. Código número catorce



```

Código 14
int binarySearch(int[] SortedArray, int target) {
    int left = 0, right = SortedArray.length - 1; // O(1)
    while (left <= right) { // O(log(n))
        int mid = left + (right - left) / 2; // O(log(n))
        if (SortedArray[mid] == target) { // O(1)
            return mid; // O(1)
        } else if (SortedArray[mid] < target) { // O(1)
            left = mid + 1; // O(1)
        } else { // O(1)
            right = mid - 1; // O(1)
        }
    }
    return -1; // O(1)
}

```

Figura 14. Ejemplo código 14

Binary Search, también conocida como búsqueda binaria, es un algoritmo de búsqueda eficiente utilizado para encontrar la posición de un elemento en una lista ordenada. es decir que tiene un funcionamiento similar al los algoritmos anteriores QuickSort y MergeSort, ya que comparten la idea de dividir las listas n cantidades de veces hasta encontrar la información deseada.

Ahora, en la primera línea de código que se tiene en cuenta es la asignación de variables las cuales tienen una complejidad de $O(1)$, con respecto a la línea de código con el ciclo while, el cual tiene una complejidad $O(\log(n))$, debido a que tiene múltiples iteraciones hasta encontrar la información deseada, con respecto a la línea de código `int mid`, la cual tiene una complejidad constante denotada como $O(1)$. A partir del if, tienen una complejidad constante, ya que las condiciones se comparan con una variable asignada y su función es determinar si se cumple o no, es decir que sus valores de entrada no afectan los algoritmos y lo mismo sucedería con la asignación de cada opción, las cuales son constantes y son denotadas como $O(1)$. Por último tenemos el retorno, el cual lo genera de manera constante, es decir que se puede denotar como $O(1)$.

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

II-Ñ. Código número quince

Handwritten code on graph paper for calculating factorial. The code is written in C++ style. It starts with 'Código 15' in purple. The function signature is 'int factorial (int n) {'. Inside, there is an 'if' statement: 'if (n==0 || n==1) { //O(1)'. This is followed by 'return 1; //O(1)'. Then a closing brace '}' for the if statement. The main return statement is 'return n * factorial (n-1); //O(n)'. Finally, a closing brace '}' for the function.

```

Código 15
int factorial (int n) {
    if (n==0 || n==1) { //O(1)
        return 1; //O(1)
    }
    return n * factorial (n-1); //O(n)
}
  
```

Figura 15. Ejemplo código 15

En el código número quince tenemos una condición la cual tiene la función de un valor que ingresa de manera constante es decir que se puede denotar como $O(1)$, con respecto al retorno, si tiene una complejidad constante $O(1)$, ya que el dato que está ingresado puede causar que varíe y pero al momento de su llamado convierte su complejidad en $O(n)$.

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

III. CONCLUSIONES

Concluimos que al analizar la complejidad de 15 algoritmos por medio de la notación Big O de manera gráfica y escrita, con una breve explicación línea por línea indicando su complejidad, La notación "Big O" se utiliza para describir el límite superior asintótico de la complejidad de tiempo de un algoritmo en función del tamaño de entrada, en este método lo ideal es determinar cual es la operación o línea de código que consume más recursos en la maquina, con el fin de no solo economizar recursos, sino que también prepara el funcionamiento del código con mayor precisión, analizando diferentes problemas o que pueden ser evitados a comparación de la creación de un código con complejidad constante. De igual manera, al analizar la complejidad en el peor caso, se obtiene una base sólida para comparar algoritmos y tomar decisiones informadas sobre cuál algoritmo es más adecuado para una situación dada.