Taller Big O

Arturo Daza - 506231039

***Resumen*—Este trabajo escrito se centra en el análisis de diversos algoritmos comunes en el ámbito de la programación y la informática. Se proporciona un análisis detallado de cada algoritmo, incluyendo una descripción de su funcionamiento, las palabras clave reservadas y operadores lógicos utilizados, y una evaluación de su complejidad temporal en el peor caso.**

I. Introducción

En el mundo de la ciencia de la computación y la programación, la eficiencia de un algoritmo es un aspecto fundamental. La Notación Big-O, también conocida como la notación del orden de complejidad, es una herramienta esencial para evaluar y comparar la eficiencia de diferentes algoritmos. Esta notación nos permite describir cómo crece el tiempo de ejecución o el uso de recursos de un algoritmo a medida que aumenta el tamaño de entrada. En este trabajo, exploraremos ejemplos de algoritmos comunes y evaluaremos su complejidad temporal utilizando la Notación Big-O, lo que nos permitirá comprender mejor cómo se comportan estos algoritmos en términos de eficiencia en diferentes situaciones.

En este trabajo se analizaron diferentes algoritmos que incluyen:

1. Búsqueda Lineal (Linear Search): Un algoritmo sencillo para encontrar un elemento en un arreglo no ordenado.
2. Búsqueda Binaria (Binary Search): Un algoritmo eficiente para buscar en un arreglo ordenado.
3. Ordenamiento Burbuja (Bubble Sort): Un algoritmo de ordenamiento básico que compara y ordena elementos adyacentes.
4. Ordenamiento por Selección (Selection Sort): Un algoritmo de ordenamiento que selecciona iterativamente el elemento más pequeño.
5. Ordenamiento por Inserción (Insertion Sort): Un algoritmo de ordenamiento que construye la solución final elemento a elemento.
6. Cálculo del Factorial (Factorial): Un algoritmo recursivo para calcular el factorial de un número.

II. Desarollo

II-A. *Primer punto*

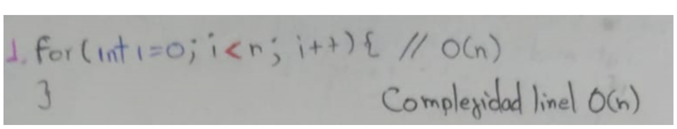

Figura 1. Primer punto

Analisis de complejidad:

- La declaración e inicialización de la variable int i = 0 es una operación de tiempo constante: O(1).
- El bucle for se ejecuta mientras i sea menor que n. Esto implica que el bucle se ejecutará n veces. Por lo tanto, la complejidad temporal del bucle for es O(n).

La complejidad total es O(1) + O(n), que se simplifica a O(n).

II-B. *Segundo punto*

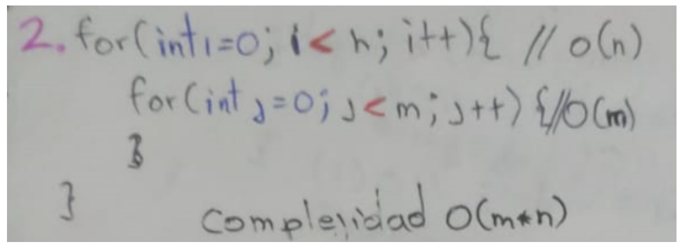

Figura 2. Segundo punto

Analisis de complejidad:

- La declaración e inicialización de las variables int i = 0 y int j = 0 en sus respectivos bucles son operaciones de tiempo constante: O(1).
- El bucle externo for se ejecuta mientras i sea menor que n. Esto implica que el bucle externo se ejecutará n veces.
- Dentro del bucle externo, el bucle interno for se ejecuta mientras j sea menor que m. Esto implica que el bucle interno se ejecutará m veces en cada iteración del bucle externo. Por lo tanto, el bucle interno se ejecutará un total de n * m veces.

*La complejidad total es O(1) + O(n*m), que se simplifica a O(n*m).*

II-C. *Tercer punto*

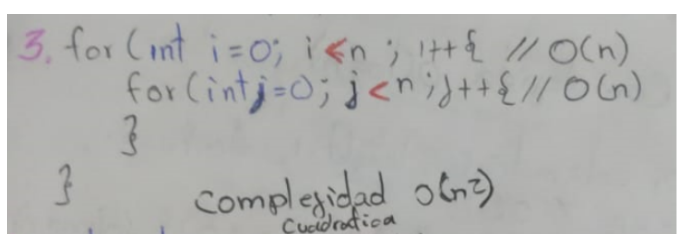

Figura 3. Tercer punto

Analisis de complejidad:

- La declaración e inicialización de las variables int i = 0 y int j = i en sus respectivos bucles son operaciones de tiempo constante: O(1).

- El bucle externo for se ejecuta mientras i sea menor que n. Esto implica que el bucle externo se ejecutará un máximo de n veces.
- Dentro del bucle externo, el bucle interno for se ejecuta mientras j sea menor que n

La complejidad total es O(1) + O(n*n), que se simplifica a O(n**2).

II-D. Cuarto punto

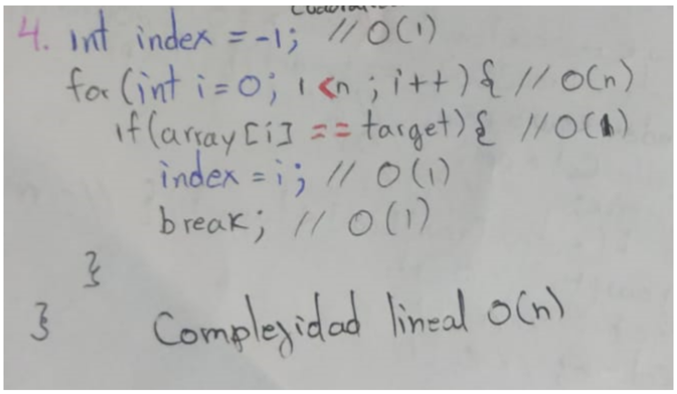

Figura 4. Cuarto punto

Analisis de complejidad:

- La declaración e inicialización de la variable int index = -1 es una operación de tiempo constante: O(1).
- El bucle for se ejecuta mientras i sea menor que n. Esto implica que el bucle se ejecutará un máximo de n veces.
- la comprobación if (array[i] == target) es una operación de tiempo constante: O(1).
- La operación index = i es una operación de tiempo constante: O(1).

La complejidad total es O(1) + O(1) + O(n) +O(1), que se simplifica a O(n).

II-E. Quinto punto

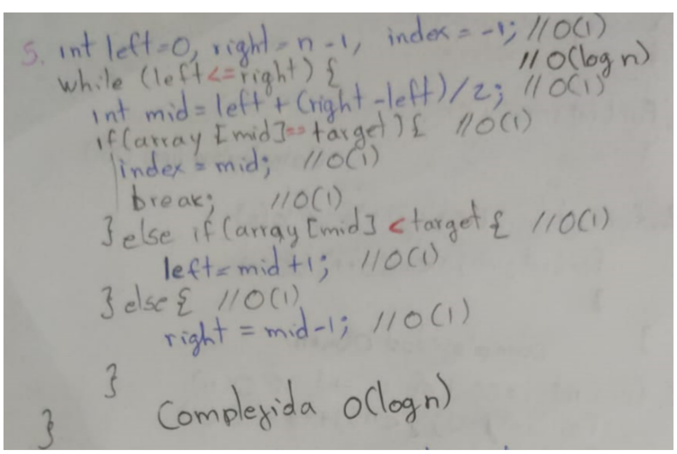

Figura 5. Quinto punto

Analisis de complejidad:

- La declaración e inicialización de las variables left, right, y index son operaciones de tiempo constante: O(1).
- El bucle while se ejecuta mientras left sea menor o igual que right. La cantidad de iteraciones depende de cuántas veces se divida el intervalo entre left y right a la mitad porlotanto esde complejidad O(log n)
- la variable mid se calcula como left + (right - left) / 2, lo que es una operación de tiempo constante: O(1).
- La comprobación if (array[mid] == target) es una operación de tiempo constante: O(1).
- Las operaciones left = mid + 1 y right = mid - 1 también son operaciones de tiempo constante: O(1).

La complejidad total es O(1) + O(log n)+ O(1) +O(1)+O(1)+O(1), que se simplifica a O(log n).

II-F. Sexto Punto

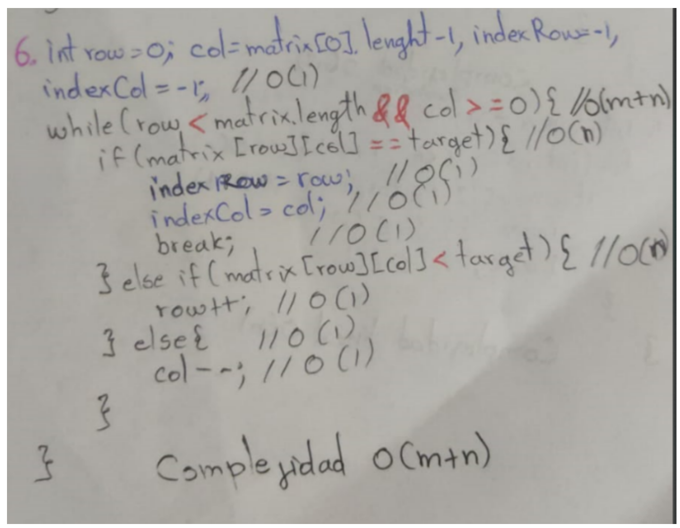

Figura 6. Sexto punto

Analisis de complejidad:

- La declaración e inicialización de las variables row, col, indexRow, y indexCol son operaciones de tiempo constante: O(1).
- El bucle while se ejecuta mientras row sea menor que la longitud de matrix (número de filas) y col sea mayor o igual a cero. La cantidad de iteraciones depende de las dimensiones de la matriz, por lo tanto tiene una complejidad de O(m+n).
- La comprobación if (matrix[row][col] == target) es una operación de tiempo constante: O(1).
- Las operaciones row++ y col– también son operaciones de tiempo constante: O(1).

La complejidad total es O(1) + O(m+n)+ O(1) +O(1), que se simplifica a O(m+n).

II-G. Septimo Punto

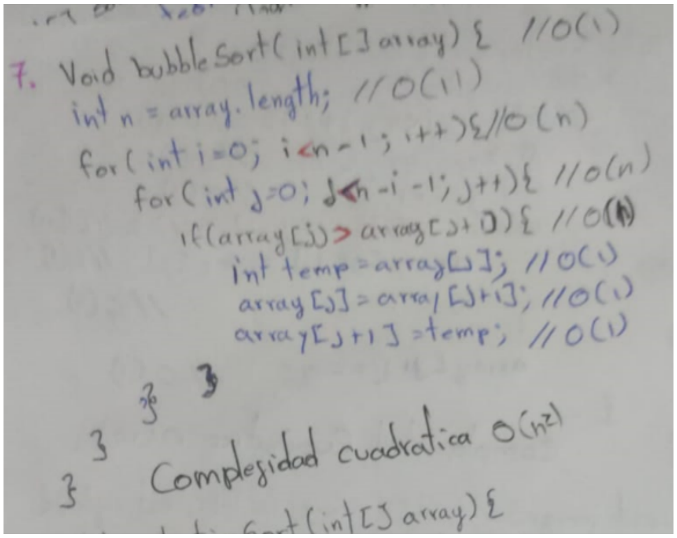

Figura 7. Septimo punto

Analisis de complejidad:

- La declaración e inicialización de la variable n con la longitud de array es una operación de tiempo constante: O(1).
- El bucle externo for se ejecuta mientras i sea menor que n - 1, por lo que es de comlepejidad O(n).
- El bucle interno for se ejecuta mientras j sea menor que n - i - 1, por lo que también es de comlepejidad O(n).
- La comprobación if (array[j] >array[j + 1]) es una operación de tiempo constante: O(1).
- as operaciones de intercambio de elementos en el arreglo (array[j] = array[j + 1] y array[j + 1] = temp) también son operaciones de tiempo constante: O(1).

*La complejidad total es O(1) + O(n**2)+ O(1) +O(1) +O(1) +O(1) , que se simplifica a O(n**2).*

II-H. Octavo Punto

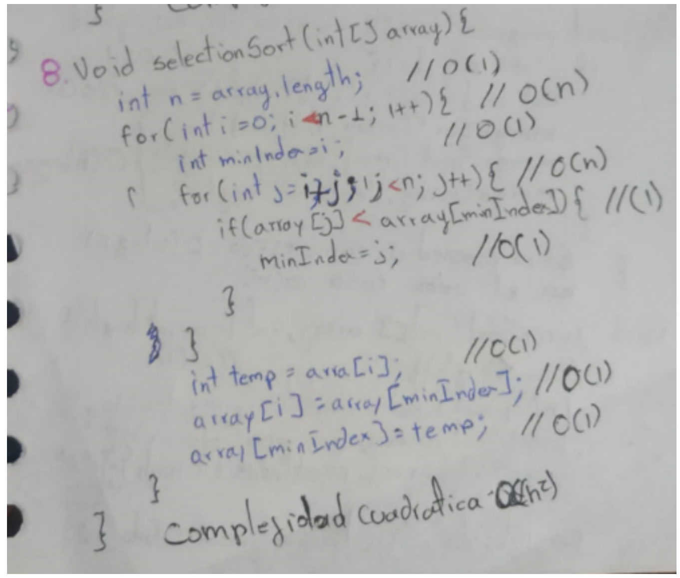

Figura 8. Octavo punto

- La declaración e inicialización de la variable n con la longitud de array es una operación de tiempo constante: O(1).
- El bucle externo for se ejecuta mientras i sea menor que n - 1, por lo que es de comlepejidad O(n).
- Se se inicializa la variable minIndex con el valor de i, lo que es una operación de tiempo constante: O(1).
- El bucle interno for se ejecuta mientras j sea menor que n, por lo que también es de comlepejidad O(n).
- Las operaciones de intercambio de elementos en el arreglo (array[i] = array[minIndex] y array[minIndex] = temp) son operaciones de tiempo constante: O(1).

*La complejidad total es O(1) + O(n**2)+ O(1) +O(1) +O(1) +O(1) , que se simplifica a O(n**2).*

II-I. Noveno Punto

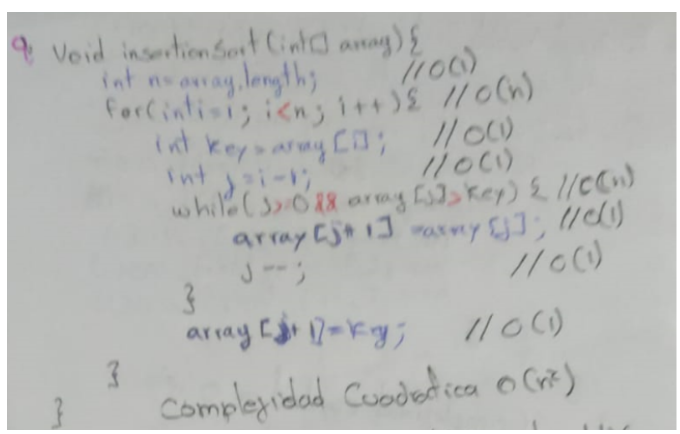

Figura 9. Noveno punto

Analisis de complejidad:

- La declaración e inicialización de la variable n con la longitud de array es una operación de tiempo constante: O(1).
- El bucle externo for se ejecuta desde i = 1 hasta i = n - 1, , por lo que es de comlepejidad O(n).
- Dentro del bucle externo, se inicializa la variable key con el valor de array[i], lo que es una operación de tiempo constante: O(1).
- Se inicializa la variable j con el valor de i - 1, también una operación de tiempo constante: O(1).
- El bucle interno while se ejecuta mientras j sea mayor o igual a cero y array[j] sea mayor que key. En cada iteración del bucle. El número de iteraciones del bucle interno depende de la posición de key en el arreglo. Por lo que es comlejidad de O(n).
- La asignación array[j + 1] = key después del bucle interno es una operación de tiempo constante: O(1).

*La complejidad total es O(1) + O(n**2)+ O(1) +O(1) +O(1) +O(1) , que se simplifica a O(n**2).*

II-J. Decimo Punto

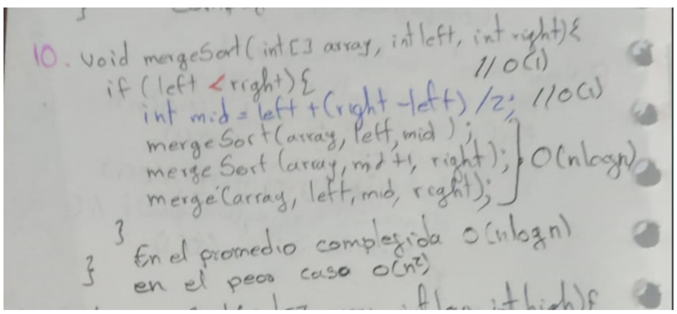

Figura 10. Decimo punto

Analisis de complejidad:

- La función mergeSort toma como entrada un arreglo array, y los índices left y right, que denotan el rango del arreglo que se va a ordenar.
- La función mergeSort se llama recursivamente en dos mitades del arreglo. Esta recursión divide el arreglo repetidamente en mitades hasta que se alcance un caso base (cuando left sea igual a right).
- La función merge se llama para combinar las mitades ordenadas del arreglo.
- El algoritmo Merge Sort divide el arreglo en mitades logarítmicas y realiza la fusión en cada nivel de la recursión. La complejidad temporal del Merge Sort es O(n log n)

Por lo que es un algorito que dividde el problema y la función de complejidad total es O(1) + O(n log n), lo que se simplifica a O(n log n) .

II-K. Onceavo Punto

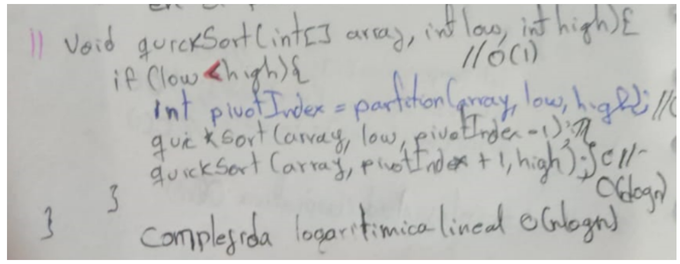

Figura 11. Onceavo punto

Analisis de complejidad:

- La comprobación if (low <high)es una operación de tiempo constante: O(1).
- La declaracion int pivotIndex = partition(array, low, high) es una operación de tiempo constante: O(1).;
- El algoritmo Quick Sort divide el arreglo en subarreglos de manera eficiente y, en promedio, tiene una complejidad temporal de O(n log n) y O(n**2) en el peor caso, donde "n.es el tamaño del arreglo.

*La complejidad total es O(1) + O(1)+ O(n**2) (por que tomamos el peor caso) , que se simplifica a O(n**2).*

II-L. Doceavo Punto

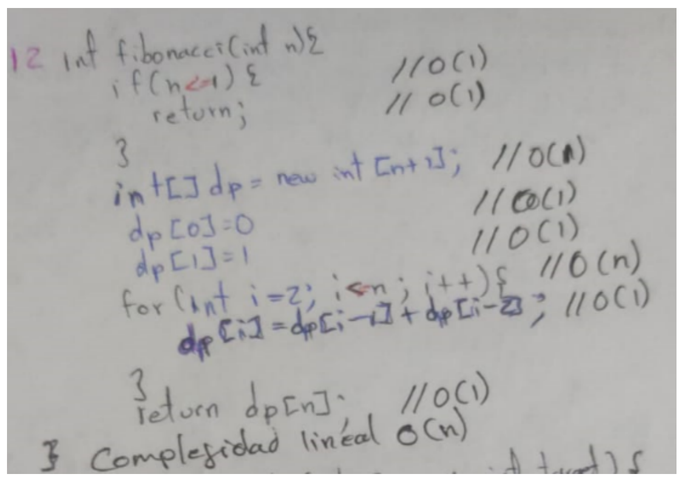

Figura 12. Doceavo punto

Analisis de complejidad:

- En el caso base, si n es menor o igual a 1, la función retorna n. Esto es una operación de tiempo constante: O(1).
- Se crea un arreglo dp de tamaño n + 1, lo que es una operación de tiempo constante O(1).
- Se inicializan los valores de dp[0] y dp[1] en O(1) cada uno.
- Luego, se itera desde i = 2 hasta i = n en el bucle for, por lo que es de comlepejidad O(n).
- Finalmente, se retorna dp[n], lo que es una operación de tiempo constante: O(1).

La complejidad total es O(1) + O(1)+ O(1) +O(n) +O(1), que se simplifica a O(n).

II-M. Terceavo Punto

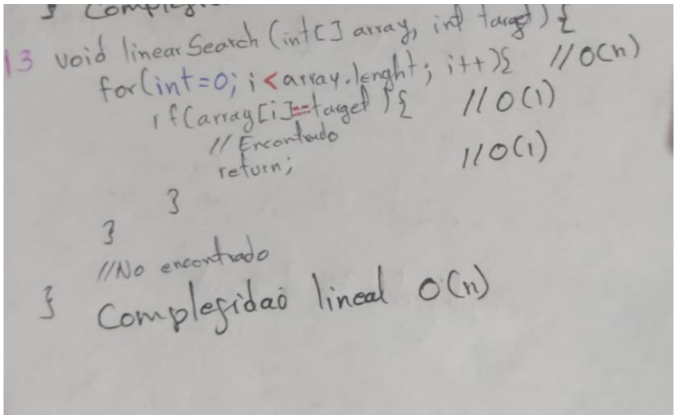

Figura 13. Terceavo punto

Analisis de complejidad:

- El bucle for itera desde i = 0 hasta i = array.length - 1, es decir, a través de todo el arreglo, por lo que es de comlepejidad O(n).
- La condición if (array[i] == target). Esto es una operación de tiempo constante O(1).
- Si se encuentra target, la función retorna de inmediato, lo que detiene la búsqueda.
- i no se encuentra target, la función continuará hasta el final del bucle for y luego retornará,

La complejidad total es O(n) + O(1)+ O(1), que se simplifica a O(n).

II-N. Catorceavo Punto

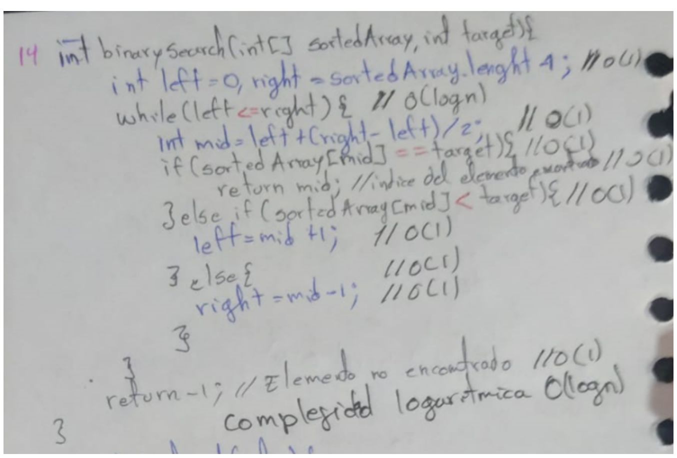

Figura 14. Catorceavo punto

Analisis de complejidad:

- Se inicializan las variables left y right con los índices extremos del arreglo, lo que es una operación de tiempo constante: O(1).
- l bucle while se ejecuta mientras left sea menor o igual que right. La cantidad de iteraciones depende de cuántas veces se divida el intervalo entre left y right a la mitad antes de encontrar target. Por lo que divide el problema es de complejidad o(log n)
- Dentro del bucle, la variable mid se calcula como left + (right - left) / 2, lo que es una operación de tiempo constante O(1).
- La comprobación if (sortedArray[mid] == target) es una operación de tiempo constante O(1).
- Las operaciones left = mid + 1 y right = mid - 1 también son operaciones de tiempo constante O(1).

La complejidad total es O(1) + O(log n)+ O(1) + O(1), por lo que la implemetación de búsqueda binaria en el peor de los casos es de complejidad O(log n).

II-Ñ. Quinceavo Punto

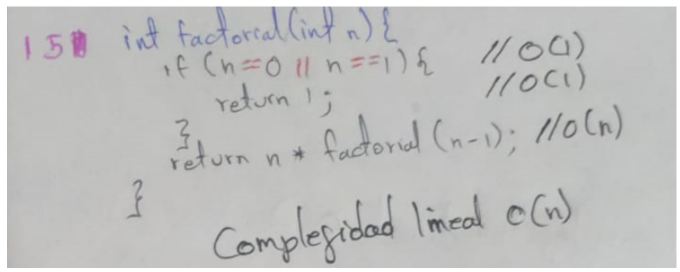

Figura 15. Quinceavo punto

Analisis de complejidad:

- La función verifica si n es igual a 0 o 1, en cuyo caso retorna 1. Esto es una operación de tiempo constante O(1).
- Si n no es igual a 0 o 1, la función recursivamente llama a factorial(n - 1) y multiplica el resultado por n. Cada llamada recursiva disminuye n en 1 y realiza una multiplicación, lo que lleva a una serie de llamadas recursivas, que tiene complejidad lineal de O(n).

La complejidad total es O(1) + O(n), que se simplificaen el peor de los casos a O(n).

III. Conclusión

En este trabajo, hemos explorado varios algoritmos fundamentales utilizados en programación e informática, analizando en detalle su funcionamiento, palabras clave reservadas, operadores lógicos y, lo más importante, su complejidad temporal en el peor caso. A través de este análisis, hemos obtenido una visión más profunda de cómo estos algoritmos se comportan en términos de eficiencia y rendimiento en función del tamaño de los datos de entrada.