Taller-1 Big-O

1st Hugo Alejandro Latorre Portela
Fundación Universitaria Konrad Lorenz
Bogotá, Colombia
hugoa.latorrep@konradlorenz.edu.co

***Abstract*—El análisis Big O es esencial en informática y programación, y desempeña un papel vital en la industria del software, ya que permite a los profesionales evaluar el rendimiento de los algoritmos en función del tamaño de los datos de entrada. Los beneficios incluyen optimizar el rendimiento de las aplicaciones, predecir y mitigar problemas de escalabilidad y mejorar la usabilidad al demostrar la capacidad de escribir código eficiente. Para los estudiantes, aprender este enfoque también desarrolla habilidades analíticas, lógicas y de resolución de problemas, lo que lo convierte en una parte importante de la formación en informática y programación.**

***Index Terms*—Optimización, recursos, lógica, complejidad, análisis**

I. Introduction

El análisis de la complejidad del algoritmo (llamado ”análisis Big O”) es muy importante en informática y programación para evaluar qué tan bien se desempeña un algoritmo en diferentes tamaños de datos. La denominación ”Big O” describe el peor de los casos e indica su eficiencia en términos de tiempo y memoria. Guíe a los profesionales para seleccionar algoritmos apropiados basados en escenarios y optimizar el rendimiento y los recursos. Big O juega un papel importante en la toma de decisiones y desarrolla habilidades clave en programación y ciencia de datos.

II. Análisis de codigos

A. CODE 1

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

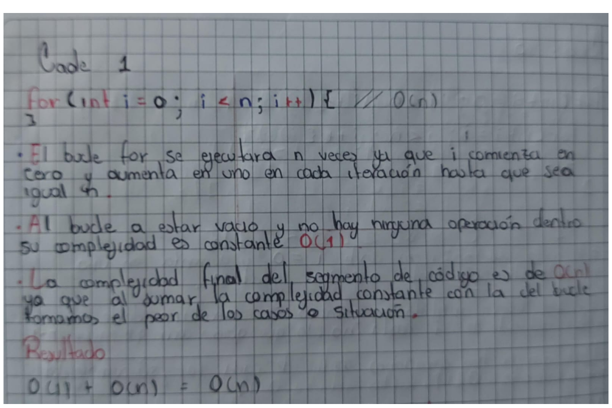

Fig. 1. Análisis code 1.

B. CODE 2

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

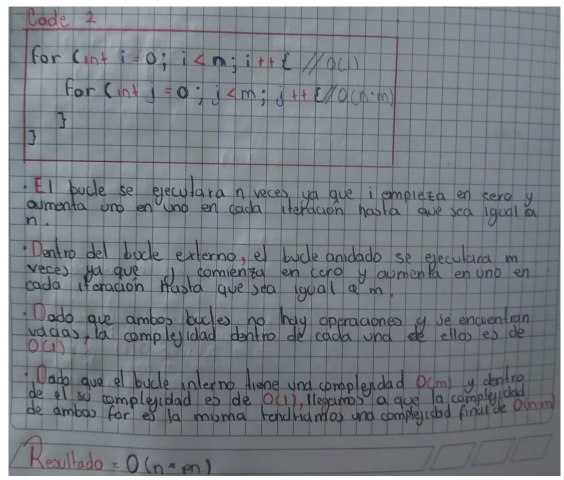

Fig. 2. Análisis code 2.

C. CODE 3

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

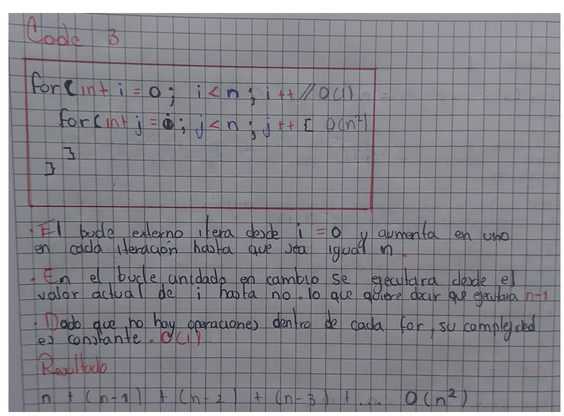

Fig. 3. Análisis code 3.

D. CODE 4

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

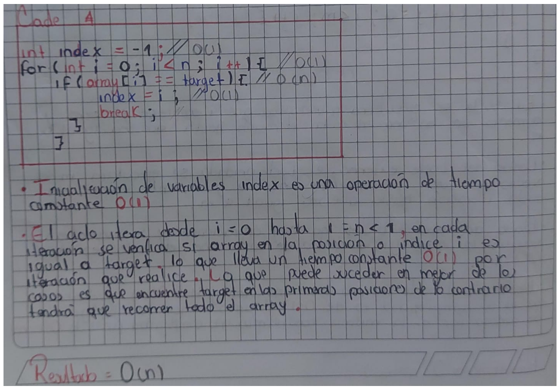

Fig. 4. Análisis code 4.

E. CODE 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;
    }}
```

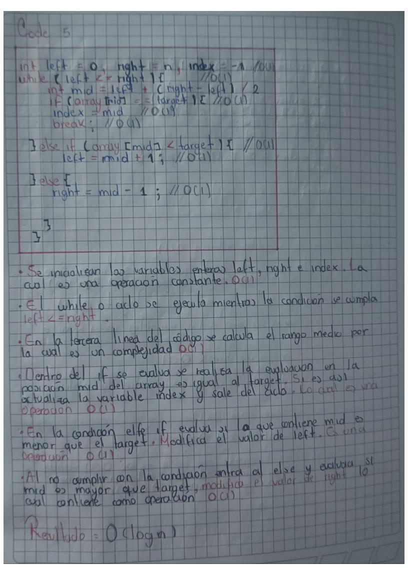

Fig. 5. Análisis code 5.

F. CODE 6

```
int row = 0, col = matrix[0].length - 1, indexRow =
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--;
    }}
```

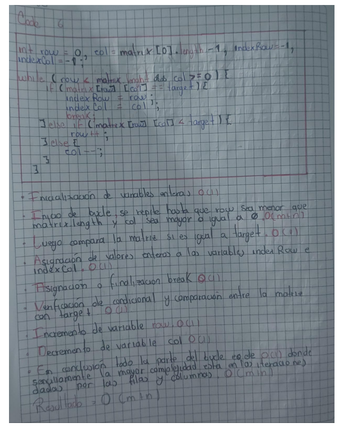

Fig. 6. Análisis code 6.

G. CODE 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;
            } } } }
```

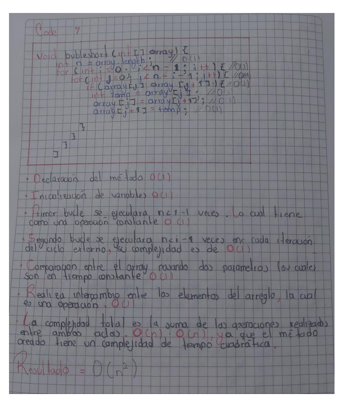

Fig. 7. Análisis code 7.

H. CODE 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;}}
```

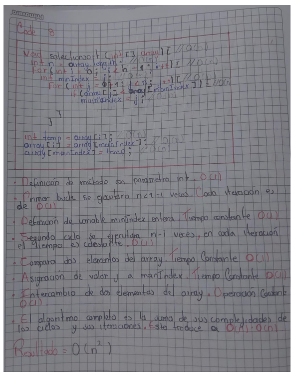

Fig. 8. Análisis code 8.

I. CODE 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;}}
```

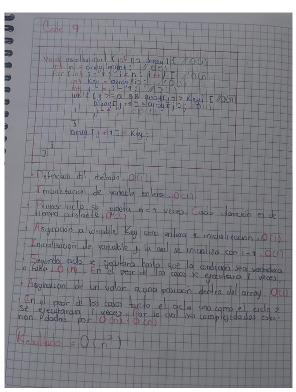

Fig. 9. Análisis code 9.

J. CODE 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);
    }}
```

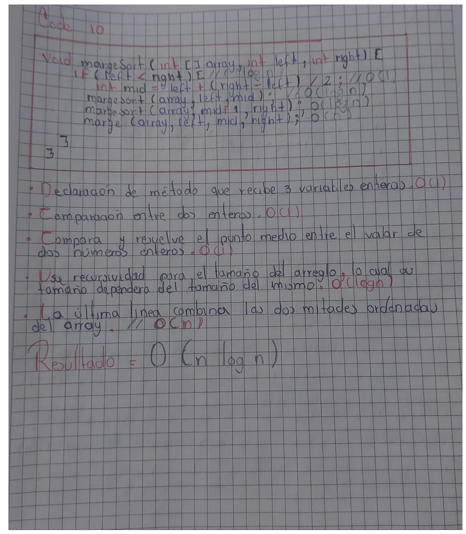

Fig. 10. Análisis code 10.

K. CODE 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);
    }}
```

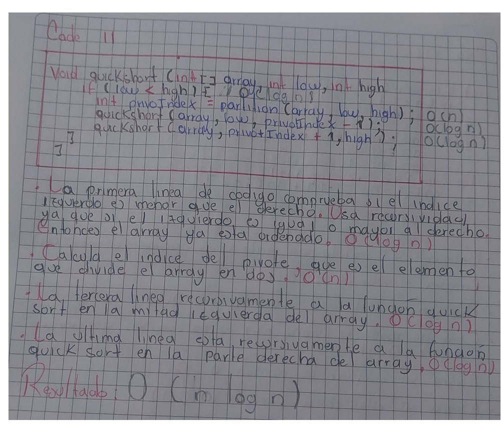

Fig. 11. Análisis code 11.

L. CODE 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];}
```

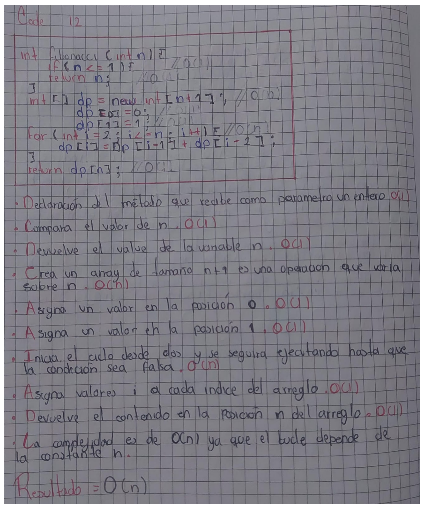

Fig. 12. Análisis code 12.

M. CODE 13

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

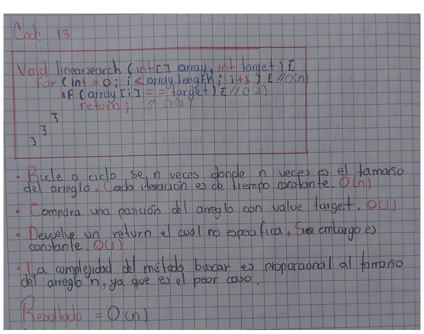

Fig. 13. Análisis code 13.

N. CODE 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 enc
        } else if (sortedArray[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Elemento no encontrado
}
```

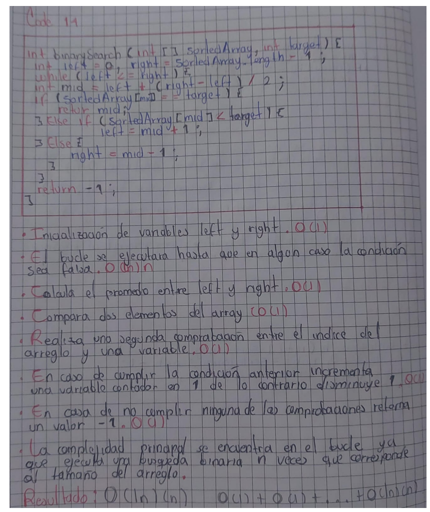

Fig. 14. Análisis code 14.

O. CODE 15

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

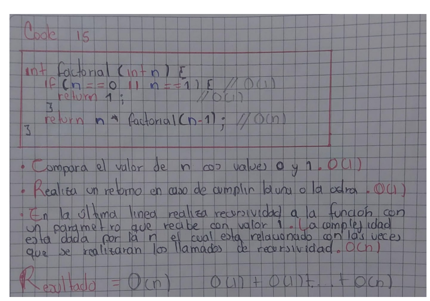

Fig. 15. Análisis code 15.

III. Conclusiones

El análisis de la complejidad algorítmica (llamado ”análisis Big O”) es la piedra angular de la informática y la programación. Esto le permite evaluar cómo se desempeña su algoritmo dados los datos de entrada cambiantes y predecir su desempeño en el peor de los casos. Esta herramienta es importante para tomar decisiones informadas, elegir el algoritmo adecuado para una situación específica y optimizar el uso del tiempo y los recursos disponibles. La capacidad de realizar análisis Big O no sólo es valiosa para los profesionales de TI, sino que también promueve el pensamiento crítico y la toma de decisiones informadas al resolver problemas.

References

[1] Khan Academy, ”Big-O Notation,” [Online]. Available: https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation. [Accessed: Fecha de Acceso].