

# Complejidad Algoritmica - Estructura de Datos

Daniel Osorio Valencia - 614222007

## I. INTRODUCCION

A continuación se realizará el análisis e interpretación para cada línea de código con el fin de determinar el grado de complejidad. Lo anteriormente mencionado nos permitirá comprender como esta complejidad afecta el código y "Big O"

### A. CÓDIGO 1

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

- El for que encontramos tiene un límite "n", entonces tenemos  $O(n)$ .

**Respuesta:**  $O(n)$

### B. CÓDIGO 2

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

- En primer for que encontramos tiene un límite "n", entonces tenemos  $O(n)$ .

- En segundo for el limite es "m", entonces tenemos  $O(m)$ .

**Respuesta:**  $O(m) + O(n) = O(n*m)$

### C. CÓDIGO 3

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

- En primer y segundo "for" tiene un límite de "n", entonces tenemos:  $O(n)$  en cada "for".

**Respuesta:**  $O(n) + O(n) = O(n^2)$

### D. CÓDIGO 4

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

- La línea `int index = -1` es una asignación, por lo cual, es de tiempo constante:  $O(1)$ .

- En el for tiene un límite de "n", entonces tenemos:  $O(n)$ .

- El if tiene una complejidad constante:  $O(1)$ .

- La línea donde le da valor al index, es constante:  $O(1)$ .

- Y el break, también es constante de  $O(1)$ .

**Respuesta:**  $O(1) + O(n) + O(1) + O(1) + O(1) = O(n)$

### E. CÓDIGO 5

```
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;  
    }  
}
```

- En la primera línea es una asignación, por tanto tenemos:  $O(1)$ . - El while es para hacer una búsqueda binaria, entonces tenemos:  $O(\log n)$ .

- La tercera línea es una asignación y es constante:  $O(1)$ .

- El if tiene una complejidad constante de  $O(1)$ .

- La línea donde le da valor al index y el break, tienen una complejidad constante:  $O(1)$ .

- El if de nuevo tiene una complejidad constante:  $O(1)$ .

- La else también es constante  $O(1)$ .

**Respuesta:**  $O(1) + O(\log n) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = O(\log n)$ .

## F. CÓDIGO 6

```
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--;
    }
}
```

- En la línea primera línea tenemos inicializaciones, así que tenemos:  $O(1)$ .
- En el while que encontramos que tiene una complejidad de  $O(n+m)$  donde  $n$  es el número de filas y  $m$  el número de columnas.
- En el if tenemos una complejidad de  $O(1)$ .
- En las líneas donde se da valor a la variable indexRow y indexCol, tienen una complejidad constante de  $O(1)$ .
- El else if y el aumento de 1 en la variable row, tienen la misma complejidad constante de  $O(1)$ .
- Y el else también tiene complejidad constante de  $O(1)$ .

**Respuesta:**  $O(1) + O(n+m) + O(1) + O(1) + O(1) + O(1) + O(1) = O(n+m)$

## G. CÓDIGO 7

-

```
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;
            }
        }
    }
}
```

- En la primera línea donde se inicializa la variable  $n$ , es constante:  $O(1)$ .
- El for se ejecuta hasta  $n-1$ , entonces tenemos:  $O(n)$ .
- El otro for será de  $n-i-1$  veces, así que también será:  $O(n)$ .
- El if tiene una complejidad constante:  $O(1)$ .
- las últimas 3 líneas tiene una complejidad constante:  $O(1)$

**Respuesta:**  $O(1) + O(n) + O(n) + O(1) + O(1) + O(1) + O(1) = O(n^2)$

## H. CÓDIGO 8

- En la primera línea con la variable  $n$  inicializada, tenemos:  $O(1)$ .
- El for va hasta  $n-1$ , así que será de:  $O(n)$ .

```
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;
    }
}
```

- En la línea donde se inicializa la variable minIndex, respecto al tamaño del  $i$ , tiene una complejidad constante:  $O(1)$ .
- El segundo for que está anidado se va a ejecutar hasta  $n$ , entonces también es:  $O(n)$ .
- El if que evalúa que la posición del array tiene una complejidad constante:  $O(1)$ .
- las últimas 3 líneas tiene una complejidad constante:  $O(1)$

**Respuesta:**  $O(1) + O(n) + O(1) + O(n) + O(1) + O(1) + O(1) = O(n^2)$

## I. CÓDIGO 9

-

```
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;
    }
}
```

- En la primera línea con la variable  $n$  inicializada, tenemos:  $O(1)$ .
- El primer for se va a ejecutar hasta  $n$ , entonces es:  $O(n)$ .
- Las inicializaciones de Key y  $j$  tiene una complejidad constante:  $O(1)$ .
- El while será hasta  $n$  veces, entonces será:  $O(n)$ .

**Respuesta:**  $O(1) + O(n) + O(1) + O(n) = O(n^2)$

## J. CÓDIGO 10

-

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

- En la primera línea se define mergeSort recibiendo 3 parámetros, un array y 2 números, siendo constante:  $O(1)$ .

- El if compara si derecha es mayor que izquierda así que es constante:  $O(1)$ .

- La declaración del int mid, tiene una complejidad constante de  $O(1)$ .

- Al llamar recursivamente a la función mergeSort hace que su complejidad sea dependiente de la profundidad de la recursión y también está relacionada con el resultado de la operación mid es por ello que esta línea tiene una complejidad de  $O(\log n)$ .

- Al volver a llamar a la función tendremos que su complejidad es la misma:  $O(\log n)$ .

- Y la función merge, que se encarga de fusionar los 2 al arrays, por lo tanto serpa:  $O(n)$ .

**Respuesta:**  $O(1) + O(1) + O(1) + O(\log n) + O(\log n) + O(n) = O(\log n)$

## REFERENCES

### K. CÓDIGO 11

```
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);
    }
}
```

- En la primera línea tenemos la definición de la función quickSort y tiene una complejidad constante:  $O(1)$ .

- El if evalúa que high sea mayor que low, así que es:  $O(1)$ .

- Inicializamos una variable llamada pivotindex, la cual es elegida dentro de los elementos del array para poder dividir el array, la complejidad de esta línea va a depender de la elección del pivote, por lo tanto tiene una complejidad de  $O(n)$ .

- En la penúltima línea se llama recursivamente a la función para que organice el sub-array izquierdo generado del principal, su complejidad es de  $O(n \log n)$ .

- En la última línea se llama nuevamente de manera recursiva a la función para que organice el sub-array derecho generado del principal, su complejidad es de  $O(n \log n)$ .

**Respuesta:**  $O(1) + O(1) + O(n) + O(n \log n) + O(n \log n) =$

$O(n \log n)$

### L. CÓDIGO 12

- En la primera línea a fibonacci tomando un número entero n, siendo constante:  $O(1)$ .

- En el if sólo se calcula que si n cero o uno, por ende, su complejidad es constante de  $O(1)$ .

- En la siguiente sección se usa programación dinámica, y como en la raíz se declara n, esta parte tendrá una complejidad:  $O(n)$ .

```
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];
}
```

- El for se ejecutará hasta n, por lo cual, será complejidad:  $O(n)$ .

- Y en la última se devuelve el valor de la posición n del array, siendo:  $O(1)$ .

**Respuesta:**  $O(1) + O(1) + O(n) + O(n) + O(1) = O(n)$

### M. CÓDIGO 13

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

- La función linearSearch, que toma un array y un valor entero a buscar, siendo complejidad:  $O(1)$ .

- En la siguiente línea al tener que recorrer todo el array, será de complejidad  $O(n)$ .

- Dentro del for encontramos un if que sólo evalúa si el array en la posición i es igual al dato ingresado, entonces tenemos que:  $O(1)$ .

- Y tenemos un return constante:  $O(1)$ .

**Respuesta:**  $O(1) + O(n) + O(1) + O(1) = O(n)$

### N. CÓDIGO 14

```
14) int binarySearch(int[] sortedArray, int target) {
    int left = 0, right = sortedArray.length - 1;
    while (left <= right) {
        int mid = (left + right) / 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
}
```

- La función binarysearch, recibe por parámetros por lo cual es constante:  $O(1)$
- En la siguiente línea se inicializan variables, siendo también:  $O(1)$ .
- El while se va a ejecutar dependiendo de la longitud del array, así que será  $O(\log n)$
- Se calcula el índice medio dentro de la sección actual del array, tiene una complejidad constante de  $O(1)$ .
- Dentro del bucle y los if, else if y else se realizan comparaciones simples, por ende, sus complejidades son:  $O(1)$ .

**Respuesta:**  $O(1) + O(1) + O(\log n) + O(1) + O(1) + O(1)$   
**=  $O(\log n)$**

## 0. CÓDIGO 15

-

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

- En la línea uno inicialización de una función; siendo su complejidad constante:  $O(1)$ .
- En el if también tendrá una complejidad constante:  $O(1)$ .
- Finalmente se devuelve la variable n multiplicado por la función factorial, obteniendo una complejidad:  $O(n)$ .

**Respuesta:**  $O(1) + O(1) + O(n) = O(n)$

## II. CONCLUSION

Con la realización de este trabajo podemos concluir la importancia de analizar, comprender y manejar una buena estructura en el desarrollo de algoritmos que nos permitan generar códigos mas limpios y eficientes para las tareas, trabajos o proyectos que se esten creando, permitiendo así, tener visibilidad de todo el panorama y tomar decisiones en cuanto a tiempo y dinero.

## III. BIBLIOGRAFÍA

Joel Ayala de la Vega, Irene Aguilar Juárez, Farid García Lamont Hipólito Gómez Ayala (2019). Introducción al análisis de algoritmos. <http://ri.uaemex.mx/bitstream/handle/20.500.11799/105198/Libro>

José A. Mañas. (2017). Análisis de Algoritmos – Complejidad. <https://www.dit.upm.es/pepe/doc/adsw/tema1/Complejidad.pdf>