

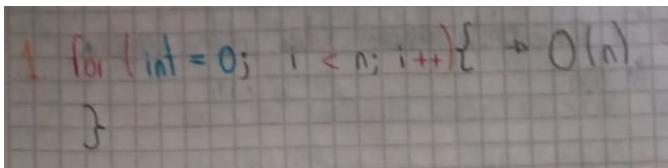
Taller 1, Complejidad Algoritmica

Daniel Gonzalez

I. INTRODUCCION

En este trabajo obtuve que hacer un proceso de análisis de cada algoritmo para poder terminar que se estaba haciendo en cada línea del código de para luego poder analizar la complejidad de esta línea de código y al final mirar de qué manera se está afectando una complejidad a la otra o qué complejidad era mayor a otra. **CODE 3**

A. CODE 1

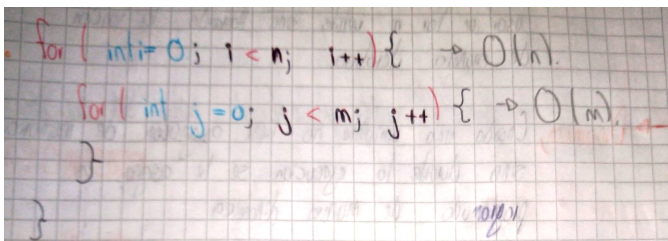


```
1. for (int i = 0; i < n; i++) { } → O(n)
```

- El for que encontramos tiene un límite de $i < n$, por lo tanto, su complejidad es de $O(n)$.

$O(n)$

B. CODE 2



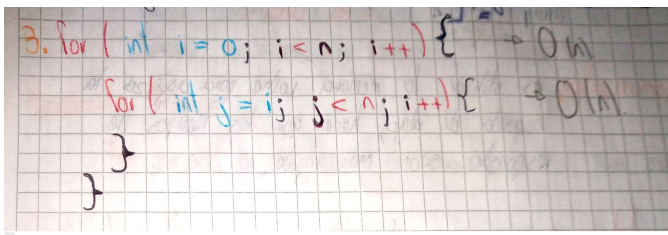
```
for (int i = 0; i < n; i++) { } → O(n)
for (int j = 0; j < m; j++) { } → O(m)
} → O(n*m)
```

- En primer for que encontramos tiene un límite de $i < n$, por lo tanto, su complejidad es de $O(n)$.

- En segundo for que encontramos tiene un límite de $j < m$, por lo tanto, su complejidad es de $O(m)$.

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

C. CODE 3



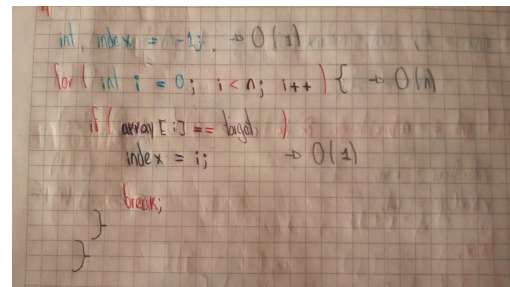
```
3. for (int i = 0; i < n; i++) { } → O(n)
    for (int j = i; j < n; j++) { } → O(n)
    } → O(n^2)
```

- En primer for que encontramos tiene un límite de $i < n$, por lo tanto, su complejidad es de $O(n)$.

- En segundo for que encontramos tiene un límite de $j < n$, por lo tanto, su complejidad es de $O(n)$.

$O(n) + O(n) = O(n^2)$

D. CODE 4



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

- La línea `int index = -1` es una asignación y tiene una complejidad de tiempo constante: $O(1)$.

- En el for que encontramos tiene un límite de $i < n$, por lo tanto, su complejidad es de $O(n)$.

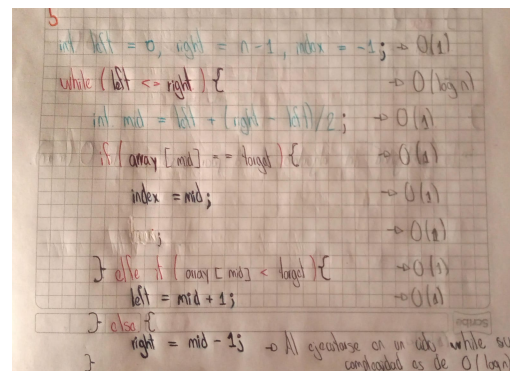
- El if que evalúa que en la posición i del array sea igual al target, tiene una complejidad constante de $O(1)$.

- La línea donde le da valor al index, tiene una complejidad constante de $O(1)$.

- Por último, el break, también tiene una complejidad constante de $O(1)$.

$O(1) + O(n) + O(1) + O(1) + O(1) = O(n)$

E. CODE 5



```
int left = 0, right = n - 1, index = -1; → O(1)
while (left <= right) { } → O(log n)
    int mid = (left + right) / 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)
    }
```

- En la línea `int left = 0, right = n - 1, index = -1` es una asignación y tiene una complejidad de tiempo constante: $O(1)$.

- El bucle while (`left <= right`) se usa para hacer un proceso

de búsqueda binaria, porque lo que tiene una complejidad de $O(\log n)$.

- En la línea `int mid = left + (right - left) / 2;` es una asignación y tiene una complejidad de tiempo constante: $O(1)$.
- El `if` que evalúa que en la posición `i` del array sea igual al `target`, tiene una complejidad constante de $O(1)$.
- La línea donde le da valor al `index` y el `break`, tienen una complejidad constante de $O(1)$.
- El `if` que evalúa que en la posición `i` del array sea menor al `target` y la línea donde le da valor al `left`, tiene una complejidad constante de $O(1)$.
- La `else` y donde se le da valor a la variable `right`, tiene una complejidad constante de $O(1)$.

$O(1) + O(\log n) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = O(\log n)$.

F. CODE 6

```
int row = 0; col = matrix[0].length - 1; indexRow = -1;
indexCol = -1;
while (row < matrix.length && col >= 0) {
    indexRow = row;
    indexCol = col;
    break;
} else if (matrix[row][col] < target) {
    row++;
} else {
    col--;
}
```

- En la línea `int row = 0, col = matrix[0].length - 1, indexRow = -1, indexCol = -1`, donde tenemos diferentes inicializaciones, esto tiene una complejidad de $O(1)$.
- 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, entonces en el peor de los casos el algoritmo tendría que recorrer todos los espacios de la matriz ósea $n+m$.
- El primer `if` que encontramos tiene una complejidad de $O(1)$, debido a que su tiempo de ejecución no depende de ninguna nada.
- 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 $O(1)$.
- Por último el `else` y la disminución de la variable `col`, tienen la misma complejidad constante de $O(1)$.

$O(1) + O(n+m) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = O(n+m)$

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;
            }
        }
    }
}
```

- En la primera línea donde se inicializa la variable `n`, respecto al tamaño del array, tiene una complejidad constante de $O(1)$.
- El primer `for` que tenemos se va a ejecutar `n-1` veces entonces su complejidad en el peor caso es de $O(n)$.
- El segundo `for` que está anidado se va a ejecutar `n-i-1` veces entonces su complejidad en el peor caso es de $O(n)$.
- El `if` que evalúa que la posición `j` del array sea mayor a la posición `j` más 1 del array, tiene una complejidad constante de $O(1)$.
- Por último en la línea donde se declara la variable `int temp`, la línea donde se dice que el nuevo valor de la posición `j` del array es `j + 1` y la línea donde se dice que la posición `j + 1` del array es igual a la variable `temp`, estas 3 líneas tienen una complejidad constante de $O(1)$.

$O(1) + O(n) + O(n) + O(1) + O(1) + O(1) + O(1) = O(n^2)$

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;  
    }  
}
```

- En la primera línea donde se inicializa la variable n, respecto al tamaño del array, tiene una complejidad constante de $O(1)$.

- El primer for se va a ejecutar hasta $n-1$, aquí la complejidad del código es de $O(n)$.

- En la línea donde se inicializa la variable minIndex, respecto al tamaño del i, tiene una complejidad constante de $O(1)$.

- El segundo for que esta anidado se va a ejecutar hasta n, aquí la complejidad del código es de $O(n)$.

- el if que evalúa que la posición del array j sea menor que la posición minIndex del array, si es verdad el valor de minIndex ahora va a hacer j, eso tiene una complejidad constante de $O(1)$.

- Por último, en la línea donde se declara la variable int temp, la línea donde se dice que el nuevo valor de la posición i del array es minIndex y la línea donde se dice que la posición del minIndex array es igual a la variable temp, estas 3 líneas tiene una complejidad constante de $O(1)$. $O(1) + O(n) + O(1) + O(n) + O(1) + O(1) + O(1) = O(n^2)$.

I. CODE 9

-

- En la primera línea donde se inicializa la variable n, respecto al tamaño del array, tiene una complejidad constante de $O(1)$.

- El primer for se va a ejecutar hasta n, aquí la complejidad del código es de $O(n)$.

- La inicialización de la variable Key y la variable j tiene una complejidad constante de $O(1)$.

- El bucle while se puede ejecutar en el peor caso n veces en total, esto quiere decir que la complejidad de $O(n)$.

$O(1) + O(n) + O(1) + O(n) = O(n^2)$

```
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;  
    }  
}
```

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);  
    }  
}
```

- En la primera línea definimos una función llamada mergeSort la cual recibe 3 parámetros un array y 2 números, esta línea tiene una complejidad constante de $O(1)$.

- El if compara si derecha es mayor que izquierda tiene una complejidad constante de $O(1)$.

- 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 se maneja la misma lógica de la anterior línea y por lo tanto su complejidad es de $O(\log n)$.

- Por último, tenemos a la función merge, que se encarga de fusionar los 2 al arrays resultantes esa formación tiene una complejidad de $O(n)$.

-

$O(1) + O(1) + O(1) + O(\log n) + O(\log n) + O(n) = O(\log n)$

REFERENCES

K. CODE 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, cuál es la fuente se encarga de organizar los datos de un array de menor a mayor, tiene una complejidad constante de $O(1)$.

- El if que encontramos nos evalúa que el dato high sea mayor que low, tiene una complejidad constante de $O(1)$.

- 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 generador del principal, su complejidad es de $O(n \log n)$.

$$O(1) + O(1) + O(n) + O(n \log n) + O(n \log n) = O(n \log n)$$

L. CODE 12

-

```

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];
}

```

- En la primera línea definimos la llamada fibonacci que toma un número entero n, eso tiene una complejidad constante de $O(1)$.

- En el if calculamos los casos base de la sucesión fibonacci en donde si n es cero o uno no hay que hacer más cálculos, la complejidad de esto es constante de $O(1)$.

- En la siguiente sección del código creamos un array llamado dp, usando programación dinámica, En la raíz se declara con n uno para poder recibir cualquier número de posiciones en el array, inicializamos los valores de la posición cero y la posición uno que sería con estos valores base de cero y uno, esta parte tiene una complejidad constante de $O(n)$.

- En el for calculamos el siguiente valor del array en la posición i a partir de las dos anteriores posiciones, este for se ejecuta hasta n veces lo que quiere decir que su complejidad es de $O(n)$.

- Por último, se devuelve el valor de la posición n del array, eso tiene una complejidad de $O(1)$.

$$O(1) + O(1) + O(n) + O(n) + O(1) = O(n)$$

M. CODE 13

-

```

13 linear
int linearSearch (int [] array, int target)
for (int i = 0; i < array.length; i++) {
    if (array[i] == target) {
        return i;
    }
}
}

```

- Se pierde la función tema linearSearch, que toma un array de elementos y un valor entero a buscar, la complejidad de esta línea es constante de $O(1)$.

- En la siguiente línea tenemos que recorrer todo el array por lo que la complejidad de esta línea sería de $O(n)$, donde n es array.Length.

- Dentro del for encontramos un if que va a evaluar que es si el array en la posición i es igual al dato ingresado, su complejidad es constante de $O(1)$.

- por último, tenemos un return que tiene la complejidad constante de $O(1)$.

$$O(1) + O(n) + O(1) + O(1) = O(n)$$

N. CODE 14

- cuando declaramos una función llamada binarysearch,

```
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;
        } else if (sortedArray[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

la cual recibe por parámetros un array ordenado y un valor entero que se va a buscar dentro del array, la complejidad de esto es constante y es de $O(1)$

- en la siguiente línea inicializamos variables seríamos dos int, la complejidad es constante y es de $O(1)$.

- el bucle while se va a ejecutar desde que la variable izquierda sea igual o menor a la variable derecha, su complejidad depende de la longitud del array, por lo que su complejidad es de $O(\log n)$

se calcula el índice medio dentro de la sección actual del array, tiene una complejidad constante de $O(1)$.

Dentro del bucle se realizan comparaciones para determinar si el valor de la posición mid es igual mayor o menor que el valor objetivo, todas las comparaciones realizadas en el if, else if o else, son constantes y tiene una complejidad de $O(1)$.

$$O(1) + O(1) + O(\log n) + O(1) + O(1) + O(1) = \mathbf{O(\log n)}$$

O. CODE 15

-

- En la línea uno inicializamos una función que recibe como parámetro una variable entera n, su complejidad es de $O(1)$.

- En el if o el vamos que el número ingresado sea cero o uno, si es cualquiera de esos 2 regresamos un uno y acabamos el programa, su complejidad es de $O(1)$.

- En el último renglón devolvemos la variable n multiplicado por la función factorial de esta manera estarían haciendo programación recursiva, Eso tiene una complejidad de $O(n)$, donde n son todas las llamadas recursivas.

$$O(1) + O(1) + O(n) = \mathbf{O(n)}$$

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

II. CONCLUSION

La conclusión de este tipo de trabajos es ver más allá de una simple ejecución de un programa y fijarse en los tiempos de ejecución de un algoritmo sencillo e imaginar cómo se aplica esto aún no es muchísimo más complicado, en pocas palabras hace ser más conscientes de la forma de programar y ser más conscientes de los costos en tiempo y dinero de nuestro algoritmo.

III. BIBLIOGRAFÍA

* Smith, J. (2022). Rendimiento de algoritmos y notación Big O. CampusMVP. <https://www.campusmvp.es/recursos/post/Rendimiento-de-algoritmos-y-notacion-Big-O.aspx>

* Khan Academy. (s.f.). Binary Search. Khan Academy. <https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search>