

Giovanny González
Cod. 506231055

Taller #1

- Código #1

```
for (int i = 0; i < n; i++) {  $\rightarrow O(n)$ 
}
```

* En la línea que contiene el **for** al ejecutarse una cantidad n realiza una complejidad de tiempo lineal $O(n)$

- Código #2

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

* Se tienen 2 **for** anidados

* El primer **for** se ejecuta una cantidad de n veces $O(n)$

* El segundo **for** se ejecuta una cantidad de m veces $O(m)$

* Se realiza una complejidad polinómica $O(n) \cdot O(m) \rightarrow O(n \cdot m)$

- Código #3

```
for (int i=0; i<n; i++) {  $\rightarrow O(n)$   
    for (int j=i; j<n; j++) {  $\rightarrow O(n)$   
    }  
}
```

* Se cuenta con 2 **for** anidados con una complejidad $O(n)$

* En la primera línea el **for** se ejecuta n cantidad de veces y en cada iteración.

* En la segunda línea el **for** inicia en el valor actual de i y se ejecuta hasta n aumentando en cada paso.

- Código #4:

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

* Se cuenta con una complejidad de tiempo $O(n)$ en el peor caso

* Pasaría lo siguiente: ~~$O(1) + O(n) + O(1)$~~

* La complejidad es de tiempo lineal $O(n)$

- Código #5

```
int left = 0, right = n - 1, index = -1;  $\rightarrow O(1)$ 
while (left <= right) {  $\rightarrow O(\log n)$ 
    int mid = left + (right - left) / 2;  $\rightarrow O(1)$ 
    if (array[mid] == target) {  $\rightarrow O(1)$ 
        index = mid;  $\rightarrow O(1)$ 
        break;
    } else if (array[mid] < target) {  $\rightarrow O(1)$ 
        left = mid + 1;  $\rightarrow O(1)$ 
    } else {  $\rightarrow O(1)$ 
        right = mid - 1;
    }
}
```

* Se observa en el algoritmo es de búsqueda binaria.
* Por cada iteración el rango de búsqueda disminuye por la mitad $O(\log n)$ con una complejidad de tiempo log-lineal

-Codigo #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;                            ->  
    } else if (matrix[row][col] < target) { -> O(1)  
        row++;                             -> O(1)  
    } else {                               -> O(1)  
        col--;                             -> O(1)  
    }  
}
```

* El código es de complejidad lineal en 2 variables $O(m+n)$.

* Lo anterior sale del código por una matriz que ejecuta operaciones que acude a las 2 dimensiones.

-Codigo # 7

```
void bubbleSort(int[] array) {  
    int n = array.length;            $\rightarrow O(1)$   
    for (int i = 0; i < n - 1; i++) {  $\rightarrow O(n)$   
        for (int j = 0; j < n - i - 1; j++) {  $\rightarrow O(n)$   
            if (array[j] > array[j + 1]) {  $\rightarrow O(1)$   
                int temp = array[j];  $\rightarrow O(1)$   
                array[j] = array[j + 1];  $\rightarrow O(1)$   
                array[j + 1] = temp;  $\rightarrow O(1)$   
            }  
        }  
    }  
}
```

* El algoritmo genera una ordenación de burbuja organiza un array en orden creciente.

* Se tiene complejidad de tiempo $O(n^2)$ complejidad cuadrática ya se obtienen 2 bucles anidados en el código que transmiten el arreglo.

-Codigo #8:

```
void selectionSort(int[] arrayX) {  
    int n = arrayX.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)  
            if (arrayX[j] < arrayX[minIndex]) { → O(1)  
                minIndex = j;        → O(1)  
            }  
        }  
        int temp = arrayX[i];        → O(1)  
        arrayX[i] = arrayX[minIndex]; → O(1)  
        arrayX[minIndex] = temp;     → O(1)  
    }  
}
```

* El código es de ordenación por selección, organiza el arreglo en un orden creciente, el peor caso sería $O(n^2)$ que es una complejidad cuadrática por lo que se ejecuta en el arreglo.

- 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)  
            array[j+1] = array[j];   → O(1)  
            j--;                     → O(1)  
        }  
        array[j+1] = key;           → O(1)  
    }  
}
```

* Los que generan una alta complejidad algorítmica son los bucles anidados es de ordenación por inserción que imprime $O(n^2)$ que es una complejidad cuadrática. El código organiza en forma creciente

-Codigo #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);  
    }  
}
```

Annotations in the image:
- A bracket on the right side of the recursive calls and the merge call is labeled $O(n \log n)$.
- An arrow points from the mid calculation to $O(n)$.

* El código es de ordenación y se llama Merge Sort que implementa la frase divide y conquistarás, lo que organiza de una forma creciente.

* El Merge Sort contiene una complejidad de $T(n) = 2 \cdot T(n/2)$ da una complejidad $O(n \log n)$.

* Complejidad logarítmica lineal $O(n \log n)$.

-Codigo # 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);  
    }  
}
```

* El codigo Quick Sort implementa el divide y conquistarás para que se organice de manera creciente se clasifica en subarreglos en un elemento y despues se organiza los subarreglos de forma frecuente

* Lo que marca variedad con el merge Sort es por la doble complejidad $O(n^2)$ o $O(n \log n)$.

-Codigo # 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 algoritmo realiza el calculo de n-ésimo número de la secuencia Fibonacci; la complejidad del código esta por el arreglo dp y tambien por el bucle for, eso hace que sea una complejidad de tiempo $O(n)$.

- Código #13:

```
void linearSearch(int [ ] array, int target) {  
    for (int i = 0; i < array.length; i++) {  $\rightarrow O(n)$   
        if (array[i] == target) {  $\rightarrow O(1)$   
            // encontrado  $\rightarrow O(1)$   
            return;  
        }  
    }  
    // No encontrado  
}
```

* se utiliza la búsqueda lineal para así encontrar un elemento exacto, revisa todo uno a uno.

* Es de complejidad tiempo en una búsqueda lineal $O(n)$ complejidad lineal n es la longitud de todo el arreglo

- Código # 14

```
int binarySearch(int[] sortedArray, int target) {  
    int left = 0, right = sortedArray.length - 1;  $\rightarrow O(1)$   
    while (left <= right) {  $\rightarrow O(\log n)$   
        int mid = left + (right - left) / 2;  $\rightarrow O(1)$   
        if (sortedArray[mid] == target) {  $\rightarrow O(1)$   
            return mid // Índice del elemento encontrado  $\rightarrow O(1)$   
        } else if (sortedArray[mid] < target) {  $\rightarrow O(1)$   
            left = mid + 1;  $\rightarrow O(1)$   
        } else {  $\rightarrow O(1)$   
            right = mid - 1;  
        }  
    }  
    return -1; // Elemento no encontrado  
}
```

* El código es de una búsqueda binaria en un arreglo ordenado, en el rango se divide frecuentemente la búsqueda a la mitad, esto se debe por la disminución algorítmica, es de complejidad $O(\log n)$

- Código #15

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

* se utiliza una función para que encuentre el factorial de un número entero por medio de la recursión, en cada llamada recursiva disminuye la complejidad factorial del número más diminuto, y continua hasta llegar a la base.

* En la llamada recursiva implementa multiplicación constante con un valor bajo de n , por ende, se encuentran n llamadas recursivas para el peor caso, por eso es una complejidad lineal $O(n)$.