

Código 1.

```
for (int i = 0; i < n; i++) { } // bucle for. La variable  
i se incrementa en 1 entonces esto hará que el código se  
ejecute n veces más que si i++ fuera menor que n
```

$O(n)$

Código 2

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

(1) La linea representada

se ejecutará n veces

(2) ese código for se eje-

cutorá m veces

Código 3

```
for (int i = 0; i < n; i++) { // O(n)  
    for (int j = i; j < n; j++) { // O(n^2)
```

}

$O(n^2)$

Este debido a los dos bucles

nestados

Código 4

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

$O(n)$

Este porque solo recorre todo

el arreglo por lo tanto el dato

Código 5

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

$O(\log n)$

dado n es el tamaño del

arreglo

Código 6

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

$(1) = O(1)$
 $(2) = O(m+n)$
 $(3) = O(1)$
 $(4) = O(1)$

donde m es el numero de filas
y n es el numero de columnas en la matriz

Código 7

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

$O(n^2)$

donde n es la longitud del arreglo

Código 8

```
void selectionSort (int[] array) { // O(n^2)  
    int n = array.length // O(1)  
    for (int i = 0; i < n - 1; i++) { // O(n)  
        int minIndex = i;  
        for (int j = i + 1; j < n; j++) { // O(n^2)  
            if (array[j] < array[minIndex]) // O(1)  
                minIndex = j;  
        }  
        int temp = array[i];  
        array[i] = array[minIndex];  
        array[minIndex] = temp;  
    }  
}
```

$O(n^2)$

donde n es la longitud del arreglo
(Esto se que hay dos bucles anidados que recorren el arreglo)

Código 9.

```
void insertionSort(int[] array) { O(1)
    int m = array.length; O(1)
    for (int i = 1; i < m - 1; i++) O(m)
        int key = array[i]; O(1)
        int j = i - 1; O(1)
        while (j >= 0 && array[j] > key) { O(m^2)
            array[j + 1] = array[j]; O(1)
            j--;
        }
        array[j + 1] = key; O(1)
    }
```

Condición de longitudes del arreglo, porque hay dos bucles anidados que recorren todo el arreglo y realizan desplazamiento de elementos

Código 10.

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

O(n)
donde n es la longitud del arreglo que se está funcionando

Código 11

```
void quickSort(int[] array, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(array, low, high); O(n)
        quickSort(array, low, pivotIndex - 1);
        quickSort(array, pivotIndex + 1, high);
    }
}
```

Se función solo se ejecutara si low < n menor que high, lo que significa que lo que se esta ordenando tiene mas de un elemento.

Si el resultado es como $O(n^2)$

La recursión se repite en los subarreglos hasta que cada subarreglo tenga un solo elemento

Código 12

```
int fibonacci(int n) {
    if (n == 1) { O(1)
        return n;
    }
    int dp[] = new int[n + 1]; O(n)
    dp[0] = 0; O(1)
    dp[1] = 1; O(1)
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2]; O(n)
    }
    return dp[n]; O(1)
}
```

O(n)

Definida la trivalidez y el cálculo factorial utilizando programación dinámica

Código 13

```
void linearSearch (int [ ] array, int target) { // O(1)
    for (int i = 0; i < array.length; i++) { // O(n)
        if (array [i] == target) { // O(1)
            return i;
        }
    }
}
```

$O(n)$

donde $n \rightarrow$ la longitud
del arreglo y debido a que debemos
recorrer el arreglo para encontrar
el elemento

Código 14

```
int binarySearch (int [ ] sortedArray, int target) { // O(1)
    int left = 0, right = sortedArray.length - 1; // O(1)
    while (left <= right) { // O(log n)
        int mid = left + (right - left) / 2; // O(1)
        if (sortedArray [mid] == target) { // O(1)
            return mid;
        } else if (sortedArray [mid] < target) {
            left = mid + 1; // O(1)
        } else {
            right = mid - 1; // O(1)
        }
    }
}
```

$O(\log n)$

donde $n \rightarrow$ la longitud del arreglo
y esto hace significativamente mas
eficiente la búsqueda para arreglo grande

Código 15

```
int factorial (int n) { // O(1)
    if (n == 0 || n == 1) { // O(1)
        return 1;
    }
}
```

return $n * \text{factorial}(n-1)$; // O(n)

$t(n)$

Entiendo porque la función llama
repetidamente a otra que n llega a
ser 1 menos en el que se vaya.