TEMA / MATERIA | DIA | MES | AÑO

Taller 1

Codigo 1.1

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

constante

- Esta linea de código crea un bucle "for" que se ejecuta "n" veces, con "i" incrementandose de 0 a "n-1". Dado que el cuerpo del bucle esta vacio.
- El incremento es constante y no depende del valor "n" el numero de iteraciones sera el mismo.

Codigo 2.

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

la complejidad total

$O(n*m)$

- Crea dos bucles anidados el bucle exterior se ejecutará "n" veces y, en cada iteración el bucle interno se ejecuta "m" veces. Por lo tanto tambien debido a esto la complejidad es $O(n*m)$.

Código 3

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

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

$= O(n^2)$

El bucle exterior se ejecuta n veces y en cada iteración el bucle interior se ejecuta desde i hasta n.

Código 4.

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

O(1)+O(1)+O(n).+O(1)+O(1)
=O(n)

El bucle for se ejecuta n veces y en cada iteración se realizan 0 operaciones constantes. la asignación y el break también son operaciones constantes

6. int left=0 right=n-1 index=-1 inicializa las variables para marcar los límites del intervalo y para almacenar la posición del elemento el bucle se ejecuta mientras el intervalo no esta vacío.

Código 5.

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

O(1)+O(logn)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)
=O(logn)

Código 6.

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

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

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

Inicia los valores row y col para comenzar desde la izquierda superior derecha de la matriz. El bucle se ejecuta mientras las posiciones estén dentro la matriz. Hasta que 'row' y 'col' se decrementen el bucle se ejecuta n+m veces n son las filas y m columnas

Código 7.

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

O(1) + O(n) + O(n) + O(1) + O(1) + O(1) + O(1) = O(n²)

int n = array.length; obtiene la longitud del arreglo el bucle exterior se ejecuta n veces siendo n la longitud del arreglo, en bucle interior se ejecuta n-1-j veces. if (array[j] > array[j+1] Compara si el elemento actual es mayor que el siguiente crea una variable temporal para realizar el intercambio entre los dos elementos

8. Código

```
void selectionSort(int[] array) { //O(1)
  int n = array.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 (array[j] < array[minIndex]) { //O(1)
        minIndex = j;   //O(1)

      } }
    int temp = array[i];   //O(1)
    array[i] = array[minIndex];  //O(1)
    array[minIndex] = temp;   //O(1)
  } }
```

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

Calcula la longitud del arreglo y almacena en "n", el bucle exterior se ejecuta "n-1" veces donde "n" es la longitud del arreglo, y el índice del elemento más pequeño de la sección no ordenada del arreglo, el bucle interior se ejecuta "n-1-i" compara si el elemento en la posición actual del bucle interior es menor que la posición mínima actual.

9. Código

```
void insertionSort(int[] array) { //O(1)
  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(n)
```

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

Calcula la longitud del arreglo y lo almacena en "n", el bucle exterior se ejecuta desde la segunda hasta la última posición, almacena el valor del elemento en "key" y como índice del elemento anterior, el bucle interior se ejecuta mientras "j" esté dentro del arreglo y el elemento j sea mayor que "key".

Código 10

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

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

if(left <right: verifica si hay más de un elemento en el segmento actual. Si no es así, detiene la recursión.

int mid = left + (right - Left)/2 : calcula el índice medio para dividir el arreglo en dos mitades.

mergeSort(array, left, mid) : llamada recursiva para ordenar la mitad izquierda del arreglo.

mergeSort (array, mid+1, right : llamada recursiva para ordenar la mitad derecha del arreglo.

Código 11

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

= O(1) + O(n) + O(n log n) + O(n log n) = O(n²)

if (low <high) : verifica si la sección actual del arreglo tiene más de un elemento. Si no se detiene la recursión.

int pivotIndex = partition (array, low, high) : llama a la función "partition" para dividir la sección actual en dos partes alrededor de un pivote.

quickSort (array, low, pivotIndex -1) : llamada recursiva para ordenar la mitad izquierda de la partición.

Código 12

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

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

if(n<=1) : verifica si el valor de "n" es 0 o 1
int[] dp= new int[n+1] : almacena los datos calculados
dp[0] : inicializa el primer valor de fibonacci
dp[1] : inicializa el segundo valor de fibonacci
for(int i=2 ; i<=n; i++) : el bucle itera "n" veces, cada vez calculando el valor de fibonacci para :
dp[i] = dp[i-1] + dp[i-2] : calcula el valor de fibonacci para : sumando los dos valores anteriores en "dp"
return dp[n] : devuelve el valor de Fibonacci calculado para "n"

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; } //O(1)
    }
}
```

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

El bucle se ejecuta 'n' veces, compara si el elemento actual del bucle es igual al elemento objetivo.
return: si se encuentra el elemento, se devuelve inmediatamente.

Código 14.

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

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

Inicializa los índices left y right para definir el rango de la búsqueda, el bucle se ejecuta left sea menos o igual a right en cada iteración, el rango se reduce a la mitad. Calcula el índice medio para dividir el rango de la búsqueda en dos mitades, compara si el elemento en "mid" es igual al objetivo.

Código 15.

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

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

Verifica si n es igual a 0 o 1, si es así devuelve 1 return n * factorial (n-1): realiza una multiplicación entre 'n' y el resultado de la llamada recursiva.

La multiplicación es constante en tiempo, pero debido a la recursión se llama a la función "factorial" n veces.