Giovanny González
Cod. 5062231055

Taller #1

- Codigo #1

```
for(int i=0;i<n;i++){ -> O(n)
}
```

*En la linea que contiene el for al ejecutarse una cantidad n realiza una complegidad de tiempo lineal O(n)

- Codigo #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).O(m) -> O(n.m)

- Codigo #3

```
for(int i=0;i<n;j++){   ->O(n)
    for(int j=i;j<n;j++){->O(n)

    }

}
```

* Se cuenta con 2 for anidados con una complejidad O(n)
* En la primera linea el for se ejecuta n cantidad de veces y en cada iteración.
* En la segunda linea el for inicia en el valor actual de i y se ejecuta hasta n aumentando en cada paso.

- Codigo #4:

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

    }

}
```

* Se cuenta con una complejidad de tiempo O(n) en el peor caso
* Pasaria lo siguiente: ~~O(1)~~ + O(n) + ~~O(1)~~
* La complejidad es de tiempo lineal O(n)

- Codigo #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;
    }else if(array[mid]<target){   ->O(1)
      left=mid+1;                  ->O(1)
    }else{                         ->O(1)
    right=mid-1;
    }
}
```

* Se observa en el algoritmo es de busqueda binaria.
* Por cada iteración el rango de busqueda 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(mtn)
    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--;
    }
}
```

* El codigo es de complejidad lineal en 2 variables O(mtn).

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

- Codigo # 7

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

* El algoritmo genera una ordenacion de burbuja organiza un arrgay en orden creciente.
* se tiene complejidad de tiempo $O(n^2)$ complejidad cuadratica ya se obtienen 2 buclesanidados en el codigo que transmiten el arreglo.

- Codigo #8:

```
Void selection sort(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;
    }
}
```

```
-> O(1)
-> O(n)
-> O(1)
-> O(n)
-> O(1)
-> O(1)
-> O(1)
-> O(1)
-> O(1)
```

* El codigo es de ordenación por selección, organizan el arreglo en un orden creciente, el peor caso seria $O(n^2)$ que es una complejidad cuadratica por lo que se ejecuta en el arreglo.

- Codigo #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 algoritmica son los bucles anidados es de ordenacion por insecion que imprime $O(n^2)$ que es una complejidad cuadratica. El codigo organiza en forma creciente

-Codigo #10

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

* El codigo es de ordenación y se llama MergeSort implementa una frase que es divide y conquistaras se organiza de forma creciente.
* El MergeSort contiene una complejidad de T(n) = 2T(n/2) da una complejida O(nlogn).
* Complejidad logarítmica lineal O(nlogn).

- Codigo # 11.

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

* El codigo Quick Sort implementa el divide y conquistaras para que se organize 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(n²) o O(nlogn).

-Codigo # 12

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

*El algoritmo realiza el calculo de n-ésimo número de la secuencia fibonacci la complejidad del codigo esta por el arreglo dp y tambien por el bucle for, eso hace que sea una complejidad de tiempo O(n).

- Codigo #13:

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

* se utiliza la busqueda lineal para asi encontrar un elemento exacto, revisa todo uno a uno.
* Es de complejidad tiempo en una busqueda lineal O(n) Complejidad lineal n es la longitud de todo el arreglo

- Codigo # 14

```
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 // Indice del elemento encontrado
        } else if (SortedArray[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Elemento no encontrado
}
```

-> O(1)
-> O(logn)
-> O(1)
-> O(1)
-> O(1)
-> O(1)
-> O(1)
-> O(1)

*El codigo es de una busqueda binaria en un arreglo ordenado, en el rango se divide frecuentemente la busqueda a la mitad, esto se debe por la disminución algoritmica, es de complejidad O(logn)

- Codigo #15

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