Ejercicio laboratorio 3

1. implementar el programa de búsqueda binaria y estimar su complejidad de acuerdo con el teorema del Maestro.

```
language-c

int BusquedaBinaria(int arr[], int ini, int fin, int elem)
{
    if (fin >= ini)
    {
        int mid = ini + (fin - ini) / 2;

        if (arr[mid] == elem)
            return mid;

        if (arr[mid] > elem)
            return BusquedaBinaria(arr, ini, mid - 1, elem);

        return BusquedaBinaria(arr, mid + 1, fin, elem);
    }
    return -1;
}
```

Teorema del maestro

T(n) = a * T($\frac{n}{b}$) + O(n^c) - - (1)

O(n^c) = O(1)
O(n^0) = O(1)
C = 0

- El tamaño de la entrada se divide a la mitad en cada llamada recursiva, con esto:
  $\frac{n}{b}$ = $\frac{n}{2}$

- Despejando B
  b = 2

- El algoritmo se llama 1 sola vez por cada llamada recursiva, asi:
  a = 1

- Sustituyendo a = 1, b = 2, c = 0 el caso a = b^c, tenemos
  1 = 2^0
  1 = 1

- Con esto podemos concliur que la complejidad del algoritmo es
  O(n^0 * $\log n$)
  O($\log n$)

2. Implementar el programa del stoogesort y estimar su complejidad de acuerdo con el teorema del Maestro.

language-c

```
void stooge_sort(int arr[], int bajo, int alto)
{
        // Complejidad O(8) = O(1)
```

```
    if (bajo >= alto) // O(2)
        return;

    if (arr[bajo] > arr[alto]) // O(2)
    {
        int aux = arr[bajo]; // O(1)
        arr[bajo] = arr[alto]; // O(1)
        arr[alto] = aux; // O(1)
    }

    if (alto - bajo + 1 >= 3)
    {
        int tercera = (alto - bajo + 1) / 3; // O(1)
        stooge_sort(arr, bajo, alto - tercera);
        stooge_sort(arr, bajo + tercera, alto);
        stooge_sort(arr, bajo, alto - tercera);
    }
}
```

Teorema del maestro

T(n) = a * T($\frac{n}{b}$) + O(n^c) - - (1)

O(n^c) = O(1)
O(n^0) = O(1)
C = 0

- El tamaño de la entrada se divide de la siguiente manera: $\frac{n}{b}$ = $\frac{2n}{3}$

- Despejando B
  b = $\frac{3}{2}$

- El algoritmo se llama 3 veces por cada llamada recursiva, así:
  a = 3

- Sustituyendo a = 3, b = $\frac{3}{2}$, c = 0 en el caso a > b^c, tenemos
  3 > $\frac{3}{2}^0$
  3 > 1

- Con esto podemos concluir que la complejidad del algoritmo es
  O($n^{\log_{\frac{3}{2}}(3)}$)

3. Implementar el programa del factorial recursivo y estimar su complejidad de acuerdo con el teorema del Maestro.

```c
int factorial(int n)
{
    return n == 0 ? 1 : n * factorial(n - 1); // O(1)
}
```

Teorema del maestro

T(n) = a * T($\frac{n}{b}$) + O(n^c) - - (1)

O(n^c) = O(1)
O(n^0) = O(1)
C = 0

El algoritmo se llama 1 sola vez por cada llamada recursiva, así:
a = 1

- El tamaño de la entrada se reduce en una unidad en cada llamada recursiva, con esto sustituyendo en (1):
  T(n) = T(n - 1) + O(1)
  b = 1

- Sustituyendo a = 1, b = 1, c = 0 el caso a = b^c, tenemos
  1 = 1^0
  1 = 1

- Con esto podemos concluir que la complejidad del algoritmo es
  O($n^0 * \log n$)
  O($\log n$)

4. Implementar un programa que genere la secuencia de Fibonacci y estimar su complejidad de acuerdo con el teorema del Maestro.

```c
int fibonacci(int n)
{
    if (n == 0)
```

```
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Teorema del Maestro

T(n) = a * T($\frac{n}{b}$) + O(n^c) - - (1)

O(n^c) = O(1)
O(n^0) = O(1)
C = 0

El algoritmo se llama 1 sola vez por cada llamada recursiva, así:
a = 2

- El tamaño de la entrada se reduce en 3 unidades en cada llamada recursiva como se muestra en la siguiente función:
  F(n) = F(n - 1) + F(n - 2)

- Con esto sustituyendo en (1):
  T(n) = T(n - 1) + T(n - 2) + O(1)
  b = 3

- Sustituyendo a = 2, b = 3, c = 0 el caso a > b^c, tenemos
  2 = 1^0
  2 > 1

- Con esto podemos concluir que la complejidad del algoritmo es
  O($n^{\log_3(2)}$)