Tarea2

Integrantes:

Jean Karlo Buitrago Orozco

Daniel Felipe Moncada Tello

Punto 1

```
void algoritmo1(int n){
        int i, j = 1;   ------------------------------------        2
        for(i = n * n; i > 0; i = i / 2){   ----------------     log2 n^2 + 2
                int suma = i + j; ---------------------     log2 n^2 + 1
                printf("Suma %d\n", suma);  ---     log2 n^2 + 1
                ++j; ------------------------------------     log2 n^2 + 1
        }
}
```

$$2 + (\log_2 n^2 + 2) + (\log_2 n^2 + 1) + (\log_2 n^2 + 1) + (\log_2 n^2 + 1)$$

$$=$$

$$8 \log_2 n + 7$$

La complejidad del algoritmo1 es $O(\log_2 n)$

Punto 2

```
int algoritmo2(int n){
        int res = 1, i, j; ------------------------------------        3
        for(i = 1; i <= 2 * n; i += 4) --------------------     ⌈n/2⌉ +1
                for(j = 1; j * j <= n; j++) ------------  n/2 × √n + 1
                        res += 2; ---------------------     n/2 × √n
        return res; ----------------------------------------        1
}
```

La complejidad del algoritmo2 es $O(n\sqrt{n})$

Punto 3

```
void algoritmo3(int n){
        int i, j, k; ----------                                   3
        for(i = n; i > 1; i--) ----------                         n
                for(j = 1; j <= n; j++) ----------                n − 1 (n + 1)
                        for(k = 1; k <= i; k++) ----------        ∑_{i=1}^{n−1} i + 1
                                printf("Vida cruel!!\n"); ------  ∑_{i=1}^{n−1} i
}
```

La complejidad del algoritmo3 puede ser expresada como $O(n^3)$ al contener 3 ciclos anidados.

Punto 4

```
int algoritmo4(int* valores, int n){
        int suma = 0, contador = 0;                  2
        int i, j, h, flag;                           4
        for(i = 0; i < n; i++){                      n + 1
                j = i + 1;                           n
                flag = 0;                            n
                while(j < n && flag == 0){
                        if(valores[i] < valores[j]){
                                for(h = j; h < n; h++){
                                        suma += valores[i];
                                }
                        }
                        else{
                                contador++;
                                flag = 1;
                        }
                        ++j;
                }
        }
```

return contador;

}

5.

Indique el número de veces que se ejecuta cada línea y determine cuál es la complejidad del siguiente algoritmo:

```
1  void algoritmo5(int n){
2    int i = 0;
3    while(i <= n){
4      printf("%d\n", i);
5      i += n / 5;
6    }
7  }
```

Número de ejecuciones

1

7

6

6

Número total de ejecuciones = 1 + 7 + 6 + 6 = 20

Complejidad del algoritmo: O(1)

El algoritmo5 es constante para $\{7\ si\ n = 5\ \vee n \geq 10\ $; entonces la complejidad del algoritmo puede ser expresada como $O(1)$ o constante amortizada (desde un análisis amortizado).

6. Escriba en Python una función que permita calcular el valor de la función de Fibonacci para un número n de acuerdo a su definición recursiva. Tenga en cuenta que la función de Fibonacci se define recursivamente como sigue:

Fibo(0) = 0

Fibo(1) = 1

Fibo(n) = Fibo(n − 1) + Fibo(n − 2)

Obtenga el valor del tiempo de ejecución para los siguientes valores (en caso de ser posible):

Tamaño entrada	Tiempo	Tamaño entrada	Tiempo
5	0.127s	35	4.445s
10	0.130s	40	36.673s
15	0.134s	45	6m17.562s
20	0.142s	50	No lo ejecuta
25	0.153s	60	No lo ejecuta
30	0.506s	100	No lo ejecuta

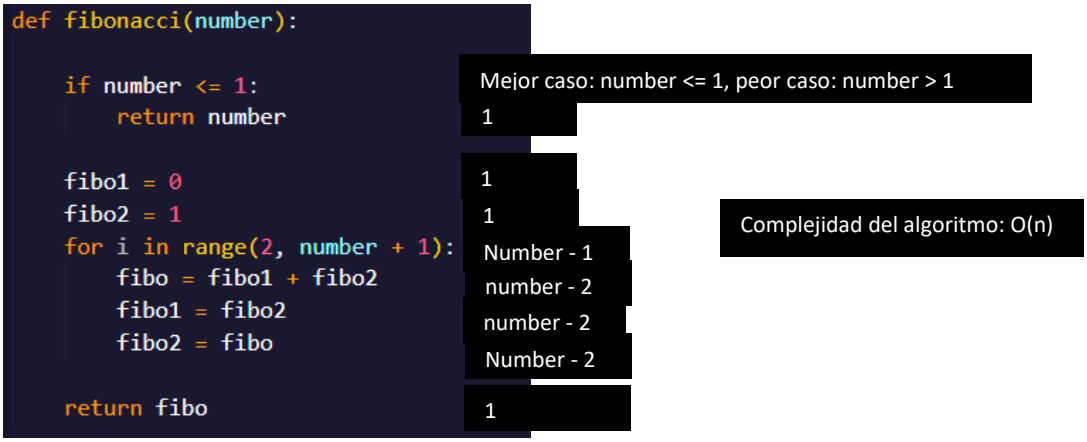
¿Cuál es el valor más alto para el cuál pudo obtener su tiempo de ejecución? Qué puede decir de los tiempos obtenidos? ¿Cuál cree que es la complejidad del algoritmo?

R/ el valor mas alto fue 45, a partir de ahí me quede esperando 1 hora a que ejecutara el valor 50 y nada, puedo concluir que esta forma de realizar la función de Fibonacci realmente no es nada recomendable para ejecutar “números grandes”, creo que la complejidad del algoritmo es exponencial la cual como vimos en clase se representa como $O(2^n)$.

7. Escriba en Python una función que permita calcular el valor de la función de Fibonacci utilizando ciclos y sin utilizar recursión. Halle su complejidad y obtenga el valor del tiempo de ejecución para los siguientes valores:

Tamaño entrada	Tiempo	Tamaño entrada	Tiempo
5	0.142s	45	0.125s
10	0.145s	50	0.127s
15	0.136s	100	0.157s
20	0.142s	200	0.124s
25	0.126s	500	0.127s
30	0.157s	1000	0.126s
35	0.126s	5000	0.126s
40	0.128s	10000	0.126s

R/



8. Ejecute la operación mostrarPrimos que presentó en su solución al ejercicio 4 de la Tarea 1 y también la versión de la solución a este ejercicio que se subirá a la página del curso con los

siguientes valores y mida el tiempo de ejecución:

Tamaño Entrada	Tiempo Solución Propia	Tiempo Solución Profesores
100	0.112s	0.118s
1000	0.127s	0.121s
5000	0.221s	0.129s
10000	0.506s	0.146s
50000	8.505s	0.286s
100000	25.158s	0.527s
200000	1m25.928s	1.170s

Responda las siguientes preguntas:

(a) ¿qué tan diferentes son los tiempos de ejecución y a qué cree que se deba dicha diferencia?
R/ Se puede ver que la diferencia a partir de la 3ra entrada aumenta y con mucha diferencia, dicha diferencia debe ser porque el código de la solución de los profes, esta mucho mas optimizado y requiere de menos procesos para dar la solución del tamaño de entrada.

(b) ¿cuál es la complejidad de la operación o bloque de código en el que se determina si un número es primo en cada una de las soluciones?

```
def numeroPrimo(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

n – 2

Mejor caso: n%2 = 0, Peor caso: que n = primo

1

1

Complejidad del algoritmo del peor caso = O(n)

Complejidad del mejor caso = O(1)

```
def esPrimo(n):
  if n < 2: ans = False
  else:
    i, ans = 2, True
    while i * i <= n and ans:
      if n % i == 0: ans = False
      i += 1
  return ans
```

La complejidad para el mejor de los casos es cuando el número ingresado(n) es menor a 2 y sería $O(1)$; Para el peor de los casos el algoritmo presentaría una complejidad de $O(n)$.