Tarea 2

1)

```
void algoritmo1(int n){
    int i, j = 1;                                    1
    for(i = n * n; i > 0; i = i / 2){                log2(n²) + 2
        int suma = i + j;
        printf("Suma %d\n", suma);                   log2(n²) + 1
        ++j;
    }
}
```

Complejidad computacional = $O(\log 2\ n)$

¿Qué se obtiene al ejecutar algoritmo1(8)? Explique

El algoritmo1(8) imprime una serie de sumas, comenzando por el valor de 64 y disminuyendo a la mitad el valor de i en cada iteración, hasta que i es menor o igual a 0. En cada iteración, j se incrementa en 1, por lo que el valor de la suma impresa también aumenta en cada iteración. Para que quede claro en la primera interacción es 64+1, 32+2, 16+3…. hasta 1+7.

```
Suma 65
Suma 34
Suma 19
Suma 12
Suma 9
Suma 8
Suma 8
```

2)

```
int algoritmo2(int n){
    int res = 1, i, j;                               1

    for(i = 1; i <= 2 * n; i += 4)                   (n/2) + 1.
        for(j = 1; j * j <= n; j++)                  (n/2) * √n
            res += 2;                                (n/2) * √n − 1

    return res;                                      1
}
```

Complejidad computacional= $O(n^{(3/2)})$

¿Qué se obtiene al ejecutar algoritmo1(8)? Explique

Al ejecutar la función algoritmo2(8) se obtiene un resultado de 17. El primer for se ejecuta 4 veces, mientras que el segundo for se ejecuta hasta que j x j <=n (en este caso n =8). Entonces j se ejecuta hasta j=3. En resumen, el primer for se ejecuta 4 veces, mientras que el segundo for 2 veces. En total serian 8 veces por lo cual la variable res aumenta en 2 -→ 8x2=16 +1=17 (el +1 se coloca porque la variable res inicializa en 1.

3)

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

1

$n-1$

$(n-1)*n$

$\sum_{i=1}^{n-1} i+1$

$\sum_{i=1}^{n-1} i$

Complejidad computacional= $O(n^3)$

4)

```
int algoritmo4(int* valores, int n){
    int suma = 0, contador = 0;
    int i, j, h, flag;

    for(i = 0; i < n; i++){
      j = i + 1;
      flag = 0;
      while(j < n && flag == 0){
        if(valores[i] < valores[j]){
          for(h = j; h < n; h++){
            suma += valores[i];
```

1

$n+1$

n

$n-i-1$

$n-i$

$\sum_{j=1}^{n-1} n-j$

$\sum_{j=1}^{n-1} (n-j)-1$

En el peor de los casos el if se ejecutaría en todas las veces que entra el while

```
            }
        }
        else{
            contador++;
            flag = 1;
        }
        ++j;
    }
}
return contador;
}
```

$n - i$

1

En el mejor de los casos: Complejidad computacional= $O(n)$

En el mejor de los casos, cuando los valores del arreglo están ordenados de forma no decreciente, la primera condición del while se evalúa como falsa en la primera iteración, y el algoritmo avanza al siguiente valor de i. Por lo tanto, el algoritmo solo realiza una comparación en el primer ciclo for y no ejecuta el ciclo while ni el ciclo for interno. La complejidad en este caso sería O(n).

En el peor de los casos: Complejidad computacional= $O(n^3)$ debido a que la complejidad del ciclo externo es O(n), la del ciclo interno es O(n^2),

En el peor de los casos, cuando los valores del arreglo están ordenados en orden inverso, el ciclo while se ejecuta completamente en cada iteración del ciclo for externo. En este caso, la complejidad sería O(n^3), ya que se tienen tres ciclos anidados.

¿Qué calcula esta operación?

Esta función cuenta el número de veces que hay un elemento en el arreglo que no es menor que el siguiente elemento, y cuenta cuántos elementos tienen esta propiedad.

5)

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

1

$\left(\frac{n}{\frac{n}{5}} + 1\right) + 1$

$\left(\frac{n}{\frac{n}{5}} + 1\right)$

Independiente del numero “n” que se ingrese, el ciclo While se ejecutará 7 veces y se ingresará al ciclo 6 veces, por lo tanto podemos decir que tiene un complejidad computacional O(1).

6)

Tamaño de entrada	tiempo	Tamaño entrada	tiempo
5	0m0.115s	35	0m5.248s
10	0m0.176s	40	0m37.679s
15	0m0.130s	45	…….
20	0m0.130s	50	…….
25	0m0.147s	60	…….
30	0m0.555s	100	…….

El programa corrió desde 5 hasta 40 el valor más alto de entrada fue el 40, con 0m37.679s, cuando el tamaño de entrada fue 45 se quedó pensando y no corrió el programa. Con los resultados obtenidos podemos observar que es como una función exponencial, entre aumentar las entradas los tiempos aumentan. La complejidad computacional es O(2^n), ya que se generan dos llamadas a la misma función, lo que significa que el tiempo de ejecución aumenta exponencialmente con el valor de entrada.

7)

Tamaño de entrada	tiempo	Tamaño entrada	tiempo
5	0m0.080s	45	0m0.082s
10	0m0.081s	50	0m0.082s
15	0m0.081s	100	0m0.082s
20	0m0.082s	200	0m0.082s
25	0m0.081s	500	0m0.082s
30	0m0.082s	1000	0m0.082s
35	0m0.081s	5000	0m0.082s
40	0m0.082s	10000	0m0.082s

Integrantes: Brayan Zuluaga y Jhon Mejias Materia: Estructura de datos

La complejidad computacional de este código es O(n)

8)

Tamaño de la entrada	Solución propia	Solución profesor
100	0m0.080s	0m0.097s
1000	0m0.112s	0m0.081s
5000	0m0.882s	0m0.097s
10000	0m3.253s	0m0.097s
50000	1m21.911s	0m0.190s
100000		0m0.331s
200000		0m0.818s

(a) ¿qué tan diferentes son los tiempos de ejecución y a qué cree que se deba dicha diferencia?

Es mucha la diferencia y es debida a la manera en la cual se encuentran los números primos. En nuestro programa hacemos un ciclo for para ir enontrando los valores si estos pasan una operación, en cambio el de los profesores determina si el valor i de un ciclo que se hace después es true a esPrimos se agrega a la lista

(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?

La complejidad computacional en la parte donde encontramos el número primo es de O(n^2) debido a que después del ciclo for donde se recorre hasta “max” tenemos un ciclo dentro que recorre todos los divisores de cada número para verificar si es un número primo.

La complejidad computacional del ejemplo de los profesores es de O(√n). Esto se debe a que el ciclo while recorre todos los números desde 2 hasta la raíz cuadrada de n. Por lo tanto, el número de iteraciones depende del valor de la raíz cuadrada de n y no de n directamente.

Anexos:

```
def fibonacciRecursion(numero):
    salida = 0
    if numero <= 0:
        return 0
    elif numero ==1:
        salida = 1
    else:
        salida = fibonacciRecursion(numero-1) + fibonacciRecursion
(numero-2)
    return salida

#print(fibonacciRecursion (45))

def fibonacciCiclos(numero):
    salida = 0
    if numero <= 0:
        salida = 0
    elif numero ==1:
        salida = 1
    else:
        a= 0
        b= 1
        for i in range (2, numero+1):
            salida = a + b
            a = b
            b = salida
    return salida

print(fibonacciCiclos(1000))
```

```
Mostrar primos
### Punto 4

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

def sumarDigitos(n):
  suma = 0
  while n > 0:
    suma += n % 10
    n //= 10
  return suma

def mostrarPrimos(N):
  print("Números primos entre 1 y %d" % N)
  primos, sumPrimo = [], []
  for i in range(2, N):
    if esPrimo(i):
      primos.append(i)
      if esPrimo(sumarDigitos(i)):
        sumPrimo.append(str(i))
  for i in range(len(primos)):
    if i < len(primos) - 1:
      print("--> %d," % primos[i])
    else:
      print("--> %d" % primos[i])
  print()
  print("Números entre 1 y %d con suma de dígitos con valor
primo:" % N)
  print(", ".join(sumPrimo))
```

```
# Descomentar la siguiente línea con el valor que se desea hacer
la ejecución.

mostrarPrimos(200000)
```

$$\sum_{i=1}^{n-1} i + 1$$