Student Name: Juan Diego Collazos - Juan David Niño
Student ID: 8977837 - 8978144

Estructura De Datos

Tarea 2 - Ejercicios de Complejidad Teórica

Ejercicio 1

```c
void algoritmo1(int n){
    int i, j = 1;                           // 2
    for(i = n * n; i > 0; i = i / 2){       // (Log2(n^2)) + 1
        int suma = i + j;                   // (Log2(n^2))
        printf("Suma %d\n", suma);          // (Log2(n^2))
        ++j;                                // (Log2(n^2))
    }
}
```

Función:

$$T(n) = 4\lceil log2(n^2)\rceil + 3$$

Simplificado:

$$T(n) = 8\lceil log2(n)\rceil + 3$$

Complejidad computacional (Big O):

$$O(log(n))$$

Qué se obtiene al ejecutar `algoritmo1(8)`? Explique.

iteraciones	i	j	print()
—	—	1	—
Entra al ciclo for			
1	64	1	”Suma 65”
2	32	2	”Suma 34”
3	16	3	”Suma 19”
4	8	4	”Suma 12”
5	4	5	”Suma 9”
6	2	6	”Suma 8”
7	1	7	”Suma 8”
Sale del ciclo for			
—	0	7	—
Termina el programa			

Resultado

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

Ejercicio 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) * raiz2(n)) + 1
            res += 2;                   // (n/2) * raiz2(n)
    return res;                         // 1
}
```

Función que define todo el programa:

$$T(n) = n\lfloor\sqrt{n}\rfloor + \frac{n}{2} + 6$$

Complejidad computacional (Big O):

$$O(n)$$

Qué se obtiene al ejecutar algoritmo2(8)? Explique:

La cantidad de veces que entran al ciclo for más interno está descrito por la Función:

$$f(n) = \frac{n\lfloor\sqrt{n}\rfloor}{2}$$

Teniendo en cuenta que la variable de 'res' es inicializada en 1 he incrementa en 2 cada vez que entra al ciclo interior, la función que describe a 'res' al finalizar el programa es:

$$g(n) = n\lfloor\sqrt{n}\rfloor + 1$$

Por lo tanto si la entrada es 8 el programa debería retornar:

$$g(8) = 8\lfloor\sqrt{8}\rfloor + 1$$

$$g(8) = 8 * 2 + 1$$

$$g(8) = 17$$

Ejercicio 3

```
void algoritmo3(int n){
    int i, j, k;                            // 3
    for(i = n; i > 1; i--)                  // n
        for(j = 1; j <= n; j++)             // n(n - 1) + 1
            for(k = 1; k <= i; k++)         // n(n - 1 + (((n-1)*n)/2)) + 1
                printf("Vida cruel!!\n");   // n(n - 1 + (((n-1)*n)/2))
}
```

La cantidad de iteraciones ciclo k está definido por la Función:

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

Simplificado:

$$f(n) = n(n-1+\frac{n(n-1)}{2})$$

$$f(n) = \frac{n^3+n^2+2n}{2}$$

Función que define todo el programa:

$$T(n) = 2(\frac{n^3+n^2+2n}{2}) + n(n-1) + n + 5$$

Simplificado:

$$T(n) = n^3 + 2n^2 + 2n + 5$$

Complejidad computacional (Big O):

$$O(n^3)$$

Ejercicio 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){          // ((n(n-1))/2) + 1
            if(valores[i] < valores[j]){    // ((n(n-1))/2)
                for(h = j; h < n; h++){     // (1/6)(n)(n^2 - 1) + 1
                    suma += valores[i];     // (1/6)(n)(n^2 - 1)
                }
            }
            else{
                contador++;
                flag = 1;
            }
            ++j;                            // ((n(n-1))/2)
        }
    }
    return contador;                        // 1
}
```

La cantidad de iteraciones ciclo h está definido por la Función:

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

Simplificado:

$$f(n) = \frac{1}{6}(n^3 - n)$$

Función que define todo el programa en el peor de los casos:

$$T(n) = \frac{2}{6}(n^3 - n) + \frac{3}{2}(n^2 - n) + 3n + 9$$

Simplificado:

$$T(n) = \frac{1}{3}n^3 + \frac{3}{2}n^2 + \frac{7}{6}n + 9$$

Complejidad computacional (Big O):

$$O(n^3)$$

¿Qué calcula esta operación? Explique.
R// Esta operación recorre una matriz unidimensional con 2 contadores i y j, i se posiciona mientras j comienza en i + 1 y recorre en resto de los elementos de la lista comparando a cada elemento con el que se encuentra en la posición i, si el elemento en la posición i resulta ser menor del elemento en la posición j, se suman los elementos desde la posición j hasta el fin de la lista a una variable llamada suma. sí, por el contrario, el elemento en la posición i mayor o igual del elemento en la posición j se suma uno a la variable contador y sale del ciclo interno sumándole 1 a i. al finalizar toda la lista en procedimiento retorna la variable contador.

Ejercicio 5

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

Teniendo en cuenta de que este programa esta escrito en el lenguaje de programación C, en C la razón de 2 números entero siempre es entera, en base ello existen 3 rangos de n que considerar:

1. En caso de que $0 \leq n < 5$, la razón $n/5$ es igual a 0, y por tanto el ciclo sera infinito, ya que en todas la iteración se le sumara 0 a i.
2. En caso de que $5 \leq n < 10$, la razón $n/5$ es igual a 1, entonces el ciclo iterara n + 1 veces, ya que sumara 1 a i en cada iteración.
3. En caso de que $n \geq 10$, la razón $n/5$ es proporcional al n, entonces el ciclo solo tendrá 6 iteraciones.

Función que define todo el programa:

$$T(n) = \begin{cases} \infty & : 0 \leq n < 5 \\ n+1 & : 5 \leq n < 10 \\ 6 & : n \geq 10 \end{cases}$$

Complejidad computacional (Big O):

$$\begin{cases} O(\infty) & : 0 \leq n < 5 \\ O(n) & : 5 \leq n < 10 \\ O(1) & : n \geq 10 \end{cases}$$

Ejercicio 6

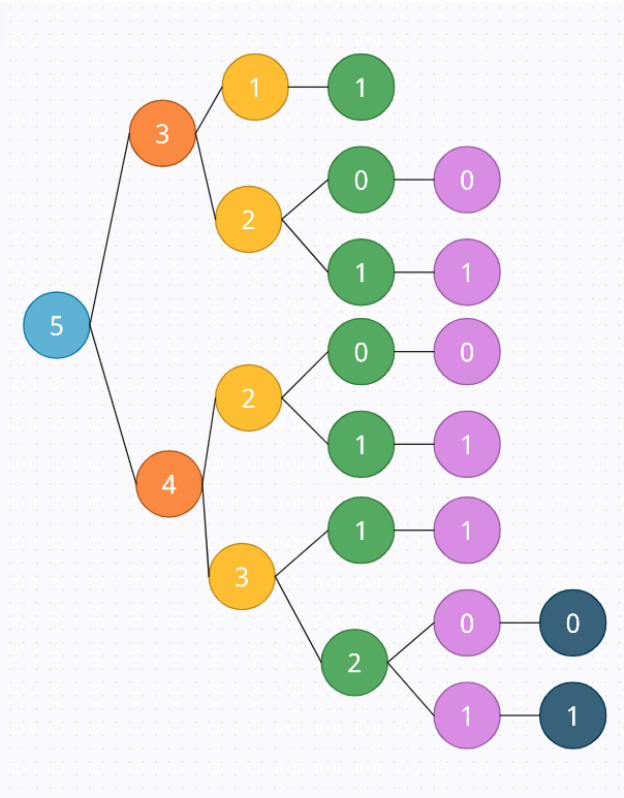
```
def fibonacci(n):
  if n == 0:
    ans = 0
  elif n == 1:
    ans = 1
  else:
    ans = figo(n - 1) + figo(n - 2)

  return ans
```

Entrada-Tiempo:

Entrada	Tiempo	Entrada	Tiempo
5	0.171s	35	6.920s
10	0.173s	40	1min 11.506s
15	0.172s	45	13min 19.457s
20	0.173s	50	—
25	0.222s	60	—
30	0.769s	100	—

Estimacion de la complejidad computacional:
Teniendo en cuenta que en esta versión recursiva de la secuencia Fibonacci los subprocesos se dividen de una manera semejante a árbol binario, como se muestra a continuación:



Es correcto asumir qué la complejidad de este algoritmo es:

$$O(2^n)$$

Sin embargo no todos los niveles del árbol se encuentran completos para todas las entradas, por ello hace falta un refinamiento de la cota superior, aproximadamente:

$$O(c^n) : 1{,}5 < C < 2$$

Ejercicio 7

```
def fibonacci(n):
  if n == 0:                  # 1
    ans = 0
  else:
    ans, a, b = 1, 0, 1     # 3
    for i in range(1, n):   # n
      ans = a + b         # n - 1
      a = b               # n - 1
      b = ans             # n - 1

  return ans                  # 1
```

Entrada-Tiempo:

Entrada	Tiempo	Entrada	Tiempo
5	0.158s	45	0.143s
10	0.142s	50	0.143s
15	0.143s	100	0.143s
20	0.160s	200	0.159s
25	0.143s	500	0.143s
30	0.159s	1000	0.142s
35	0.143s	5000	0.158s
40	0.143s	10000	0.159s

Función que define todo el programa:

$$T(n) = 3(n-1) + n + 5$$

Simplificando:

$$T(n) = 4n + 2$$

Complejidad computacional (Big O):

$$O(n)$$

Ejercicio 8

Tamaño Entrada	Tiempo Solución Propia	Tiempo Solución Profesores
100	0.142s	0.160s
1000	0.143s	0.143s
5000	0.190s	0.158s
10000	0.283s	0.189s
50000	2.705s	0.299s
100000	8.919s	0.529s
200000	30.213s	1.267s

(a) ¿Qué tan diferentes son los tiempos de ejecución y a qué cree que se deba dicha diferencia?
R// En valores entre 100 y 1000 sus tiempos de ejecución son bastante parecidos, pero cuanto más crece más diferencia existe, esto se debe a que el programa que implementamos para definir si un número es primo verifica si es divisible por todos los números primos menores a él (que se encuentran alojados en un arreglo). En valores pequeños cuando existe menor cantidad números primos la diferencia no es mucha, pero entre más crece la entrada más números tiene que revisar.

(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?
R// En la solución implementada por nosotros la complejidad esta en términos del tamaño de lista que contiene los primos:

```
1  def esPrimo(num, listprim):           # n = len(listprim)
2    ans, i = True, 0                    # 2
3    while i < len(listprim) and ans:    # n + 1
4      if num % listprim[i] == 0:        # n
5        ans = False
6      i += 1                            # n
7    return ans                          # 1
```

Función que define todo el bloque de código en el peor de los casos (definiendo n como el tamaño de la lista):

$$T(n) = 3n + 4$$

Complejidad computacional (Big O):

$$O(n)$$

Para la otra solución la complejidad esta en términos de la variable n:

```
1  def esPrimo(n):
2    if n < 2: ans = False               # 1
3    else:
4      i, ans = 2, True                  # 2
5      while i * i <= n and ans:         # raiz2(n)
6        if n % i == 0: ans = False      # raiz2(n) - 1
7        i += 1                          # raiz2(n) - 1
```

```
8   return ans
```

Función que define todo el programa en el peor de los casos:

$$T(n) = 3\sqrt{n} - 1$$

Complejidad computacional (Big O):

$$O(\sqrt{n})$$