Integrantes: Harold David Guerrero 8976203
Juan Esteban Salas 8975783

Esta es la Tarea 2 del curso *Estructuras de Datos*, 2023-1. La actividad se debe realizar en **parejas** o de forma **individual**. Sus soluciones deben ser entregadas a través de BrightSpace a más tardar el día **15 de Febrero a las 22:00**. En caso de dudas y aclaraciones puede escribir por el canal `#tareas` en el servidor de *Discord* del curso o comunicarse directamente con los profesores y/o el monitor.

Condiciones Generales

- Para la creación de su código y documentación del mismo use nombres en lo posible cortos y con un significado claro. La primera letra debe ser minúscula, si son más de 2 palabras se pone la primera letra de la primera palabra en minúscula y las iniciales de las demás palabras en mayúsculas. Además, para las operaciones, el nombre debe comenzar por un verbo en infinitivo. Esta notación se llama *lowerCamelCase*.

  Ejemplos de funciones: `quitarBoton`, `calcularCredito`, `sumarNumeros`

  Ejemplos de variables: `sumaGeneralSalario`, `promedio`, `nroHabitantes`

- Todos los puntos de la tarea deben ser realizados en un único archivo llamado `tarea2.pdf`.

Ejercicios de Complejidad Teórica

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

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

2
Log2(n**2)
n**2 +1
n
n

T(n) = 2 + n+1 + n**2+1 + n + n = Log(n**2) + 3n + 6

T(n) E O Log2(n)

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

Se obtiene este resultado ya que la variable n que en este caso es 8, se multiplica por ella misma y se divide el resultado entre 2 cada vez que se reinicia el ciclo. Posteriormente este resultado se suma con la variable j que está definida en 1 y que luego se le suma 1 unidad cada vez que se reinicia el ciclo. Esta suma se imprime en la terminal como “Suma %d\n" siendo el “%d” el valor de esta.

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

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

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

   for(i = 1; i <= 2 * n; i += 4)
       for(j = 1; j * j <= n; j++)
           res += 2;
   return res;
}
```

1

(n/2) + 1
$\sqrt{n} * \left(\frac{n}{2}\right)$
$\sqrt{n}$
1

¿Qué se obtiene al ejecutar algoritmo2(8)? Explique.

Respuesta:

$$= 1 + \left(\frac{n}{2}\right) + 1 + \sqrt{n} * \left(\frac{n}{2}\right) + \sqrt{n} + 1$$

$$= 3 + \frac{\left((\sqrt{n} * n + 2 * \sqrt{n})\right)}{2} = \frac{(\sqrt{n}(n+2))}{2} + 3$$

T(n) E O $(n * \sqrt{n})$

Se obtiene 17.

Al ejecutar la función con n = 8 se obtiene como resultado 17, esto sucede ya que primero se define la variable que se retornara al final en 1 y “j” e “i” como enteros. Luego entra en un ciclo que define i = 1 y que toma la condición de i debe ser menor o igual al producto de n por 2 y cada vez que se reinicia el ciclo, i tendrá un incremento en 4. Una vez se cumple la condicione entra al segundo ciclo que define j = 1 y pone la condición de que el cuadrado de j debe ser menor o igual a n y cada vez que se reinicia aumenta j en 1. Si se cumple esto se le sumara a la variable res 2 unidades. Por lo tanto, aquí un ejemplo de que pasa si n = 8:

```
i=1  1 <= 16:    #Los valores que inician en el primer ciclo
        j=1  1 <= 8:     #Los valores que inician en el segundo ciclo
                res=3
        j=2  4 <= 8:     #Continuidad del segundo ciclo hasta que la condición no se cumpla
                res=5
#Aqui termina la primera ronda y el primer ciclo se vuelve a ejecutar
i=5  5 <= 16:    # i tiene un incremento de 4 unidades
        j=1  1 <= 8:     #De igual manera el segundo ciclo será igual al de la ronda anterior
                res=7
        j=2  4 <= 8:
                res=9
i=9  9 <= 16:
        j=1  1 <= 8:
                res=11
        j=2  4 <= 8:
                res=13
i=14  14 <= 16: #Depues de esta ronda la condición ya no se cumple
        j=1  1 <= 8:
```

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

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

Respuesta:

Complejidad:

O (n^3)

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

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

¿Qué calcula esta operación? Explique.

```
int algoritmo4(int* valores, int n) {
        int suma = 0, contador = 0;                    → 2
        int i, j, h, flag;              → 4

        for (i = 0; i < n; i++) {
            j = i + 1;                                 } n + 1
            flag = 0;
            while (j < n && flag == 0) {               → Σ_{j=i+1}^{n+1} j + 1
                if(valores[i] < valores[j]) {          Peor caso: 1 Mejor caso: Σ_{j=i+1}^{n+1} j + 1
                    for (h = j; h < n; h++) {          Peor caso: 0 Mejor caso: Σ_{h=j+1}^{n+1} Σ_{j=i+1}^{n+1} j + 1
                        suma += valores[i];            Peor caso: 0 Mejor caso: Σ_{h=j+1}^{n+1} Σ_{j=i+1}^{n+1} j + 1
                    }
                }
                else {                     → 1
                    contador++;            → 1
                        flag = 1;          → 1
                }
                ++j;                       → Peor Caso: 1   Mejor Caso: Σ_{j=i+1}^{n+1} j + 1
            }
        }
        return contador;                   → 1
```

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

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

I = 0, n/5, 2n/5, 3n/5, 4n/5, n, 6n/5

T(n) = 1 + 7 + 6 + 6 = 20

O (1)

La complejidad de este algoritmo se debe a que en la linea del While el numero de iteraciones si depende de la variable n, por lo cual, podemos decir que la cantidad de iteraciones del ciclo While es constante y depende tanto de n como de i.

Complejidad Teórico-Práctica

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	0.62 seg	35	7.43 seg
10	1.009 seg	40	65.47 seg
15	1.15 seg	45	18 min
20	1.25 seg	50	1 hora, 55 min
25	3.0 seg	60	…
30	3.915 seg	100	…

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. Como podemos ver los resultados son una función exponencial lo cual a medida que el tamaño de la entrada aumenta asi mismo lo hará el tiempo. Cuando el tamaño de la entrada era 60 se debería de demorar más de 5 horas probablemente, ese el valor más alto que pudimos obtener ya que mi pc a medida que aumentaba la entrada asi mismo aumentaba el uso de la CPU.

La complejidad del sistema se debe a que en la función con nombre fibo(n) lo que hacemos es volver a llamar a la función en el código.

Codigo de forma recursiva

```python
# Punto 6


def fibo(n):
    ans = None
    if n <= 1:
        ans = n
    else:
        ans = fibo(n - 1) + fibo(n - 2)
    return ans
#print(fibo(10))
```

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	0.124 seg	45	0.154 seg
10	0.134 seg	50	0.157 seg
15	0.145 seg	100	0.165 seg
20	0.147 seg	200	0.173 seg
25	0.146 seg	500	0.179 seg
30	0.150 seg	1000	0.182 seg
35	0.152 seg	5000	0, 195 seg
40	0.153 seg	10000	0.26 seg

R. Complejidad del algoritmo:

La complejidad del algoritmo se debe a que la función se vuelve a llamar en cada iteración hasta llegar a n. Debido a esto el nivel de complejidad del algoritmo seria exponencial.

```python
# Punto 7

def fibo(n):
    lista = [0, 1]
    for i in range(n+1):
        if i <= 1:
            resp = lista[i]
        else:
            lista.append((lista[i-1]) + (lista[i-2]))
            resp = lista[i]

    return resp
print(fibo(5000))
```

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	0.159 seg	0.112 seg
1000	0. 178 seg	0.129 seg
5000	1.902 seg	0.144 seg
10000	4.858 seg	0.161 seg
50000	1 min 42 seg	0.240 seg
100000	11 min 39 seg	0.629 seg
200000	20 min 24 seg	1.088 seg

Responda las siguientes preguntas:

Codigo:

```python
# Punto 8

# Mi codigo


def mostrarPrimos(N):
    prim = []
    prim2 = []
    for i in range(2, N + 1):
        if esPrimo(i):
            prim.append(i)
    print("Números primos entre 1 y %d:" % N)
    for nPrim in range(0, len(prim)):
        if nPrim == len(prim)-1:
            print("--> %d" % prim[nPrim])
        else:
            print("--> %d," % prim[nPrim])
    for i in prim:
        numString = str(i)
        sum = 0
        for j in numString:
            sum = sum + int(j)
        if esPrimo(sum):
            prim2.append(i)
    print("Números entre 1 y", str(N), "con suma de dígitos con valor primo")
    print(*prim2, sep=', ')


def esPrimo(i):
    ans = True
    for r in range(2, i):
        if i % r == 0:
            ans = False
    return ans


#mostrarPrimos(200000)
```

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

La diferencia de los tiempos de los tiempos es bastante notoria, podemos ver como en la entrada de (200000) la diferencia de tiempo entre nuestra solución y la de los profes tiene una gran diferencia, en este caso con mi código se demoró 20 min y el de los profes 1 seg. Esta diferencia se debe a que en el código de los profesores se usan 3 funciones que separan los 3 procedimientos: Verificar si es primo, sumar los dígitos e imprimir los datos.

(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?

Respuesta de nuestro código:

La complejidad del código es que en la función esPrimo(i) depende de i

```
def esPrimo(i):
    ans = True                      1
    for r in range(2, i):           i + 1
        if i % r == 0:              i
            ans = False             i
    return ans                       1
```

```
# Profes

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