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`.

SOLUCIÓN

- Juan David Tabares Perez (8977554)
- Juan Pablo Andrade (8978239)

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;
   }
}
```

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

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

Al ejecutar algoritmo1(8) nos produce:

```
C:\Users\juand\CLionProjects\untitled\cmake
Suma 65
Suma 34
Suma 19
Suma 12
Suma 9
Suma 8
Suma 8

Process finished with exit code 0
```

En la primera iteración, la variable suma toma el valor de 64 + 1 = 65, luego se imprime. En la segunda iteración, i se actualiza a 32, y suma toma el valor de 32 + 2 = 34, esto se repite hasta que i es menor o igual que 0 por lo que se imprimen 7 sumas.

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;
}
```

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

    for(i = 1; i <= 2 * n; i += 4) ⟶ n/2 veces
        for(j = 1; j * j <= n; j++) ⟶ √n veces
            res += 2; ⟶ (n/2)·√n ·2
    return res; ⟶ 1 vez
}
```

Complejidad = O(n^(3/2))

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");
}
```

```
void algoritmo3(int n){                                  Complejidad = O(n^3)
    int i, j, k; ⟶ 1 vez
    for(i = n; i > 1; i--) ⟶ n - 1 veces
        for(j = 1; j <= n; j++) ⟶ (n-1)·n veces
            for(k = 1; k <= i; k++) ⟶ n(n+1)/2 - n veces
                printf("Vida cruel!!\n"); ⟶ n(n+1)/2 - 1 veces
}
```

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;
}
```

```
int algoritmo4(int* valores, int n){
   int suma = 0, contador = 0; ⟶ 1 vez
   int i, j, h, flag; ⟶ 1 vez

   for(i = 0; i < n; i++){ ⟶ n veces
     j = i + 1; ⟶ n-1 veces
     flag = 0; ⟶ n veces
     while(j < n && flag == 0){ ⟶ n-1 veces   Peor caso
        if(valores[i] < valores[j]){ ⟶ (n-1)/2 veces
           for(h = j; h < n; h++){
             suma += valores[i]; } n(n-1)(n-2)/6 veces
           }
        }
        else{
           contador++; } n(n-1)/2 veces   Peor caso
           flag = 1;
        }
        ++j; ⟶ n(n+1)/2 veces   Peor caso
     }
   }
   return contador; ⟶ 1 vez
}
```

Complejidad = O(n^3)

¿Qué calcula esta operación? Explique.

El programa en si lo que hace es recorrer un arreglo de enteros "valores" de tamaño "n" y cuenta cuántas veces un número en una posición "i" del arreglo es mayor que un número en una posición "j" del arreglo, donde "j" es mayor que "i". Para hacerlo, se comparan todos los números en las posiciones siguientes a la posición "i" hasta que se encuentra un número mayor o se llega al final del arreglo, al final se retorna la variable contador que cueta el número de veces que la condición valores[i] < valores[j] no se cumplió.

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;
  }
}
```

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

Complejidad = O(n)

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	2.69 seg	35	2.90 seg
10	1.81 seg	40	29.70 seg
15	0.00018 seg	45	380.25 seg
20	0.0036 seg	50	
25	0.033 seg	60	
30	0.27 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?

Solución

El valor mas alto que pude obtener haciendo cada una de las ejecuciones con las entradas planteadas fue de 45, ya que a partir de ahí el tiempo de respuesta era cada y cada vez mas grande haciendo imposible conocer el tiempo de ejecución con parámetros grandes, frente a la complejidad del algoritmo podría decir que es

exponencial y por lo tanto su notación será O(2^n) en donde podemos decir que n es el numero de elementos de la secuencia Fibonacci que se van a calcular.

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 ejecuciónpara los siguientes valores:

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	1.99 seg	45	4.19 seg
10	2.29 seg	50	4.40 seg
15	2.79 seg	100	7.09 seg
20	3.50 seg	200	1.15 seg
25	3.30 seg	500	2.92 seg
30	3.59 seg	1000	9.28 seg
35	3.69 seg	5000	0.00048 seg
40	3.79 seg	10000	0.0024 seg

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	0.0004612000000000088 seg	0.0002537999999998597 seg
1000	0.022468000000000002 seg	0.0016261000000000053 seg
5000	0.7942192 seg	0.0187654 seg
10000	2.7032195999999997 seg	0.030350799999999997 seg
50000	68.73163319999999 seg	0.18395820000000002 seg
100000	260.6961703 seg	0.3747411 seg
200000	1270.6131241 seg	0.9347817999999999 seg

Responda las siguientes preguntas:

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

(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?

Solución

a) Los tiempos de ejecución del programa **"mostrarPrimos"** en la solución propia son muy diferentes a la ejecución del programa **"mostrarPrimos"** en la solución de los profesores, ya que, como se puede observar en la tabla, los tiempos de la solución propia hasta el parámetro 5000 se logra una diferencia significativa en cuanto a tiempo (segundos). Mientras que, en la solución de los profesores, se logra un manejo relativamente "perfecto" del tiempo de ejecución por cada uno de los parámetros.

Revisando la solución de los profesores y haciendo una comparación en cuanto a la utilización de recursos, para la solución del ejercicio en la función **"sumarDigitos"** de la solución propia, utilizo un bucle for y convierto el número en una cadena para luego iterar sobre los caracteres de la cadena y sumar los dígitos. Mientras que, en la solución de los profesores, se utiliza un ciclo while para sumar los dígitos del número, por lo cual su complejidad sería proporcional al número de dígitos del número. Otro punto importante que hace la diferencia es que, en la solución propia, en la función **"mostrarPrimos"**, utilizo dos ciclos for, uno para encontrar los números primos y otro para encontrar los números con suma de dígitos con valor primos, mientras que en la solución de los profesores solo se usa un ciclo for para encontrar los números primos y los números con suma de dígitos con valor primos al mismo tiempo.

b) Precisando en la solución propia podemos decir que la complejidad del algoritmo que halla el numero primo se divide en varia partes, en donde el bucle **for** ejecuta **num - 2** iteraciones ya que comienza en 2 y termina en **num – 1**, ademas se utiliza la operación modulo **%** y la comparación == esto quiere decir que el número total de operaciones y divisiones es de (num-2) * 2, esto en términos de

notación seria O(num) ya que el tiempo de ejecución aumenta linealmente dependiendo del tamaño de la entrada **num**

La gran diferencia que existe, a nuestro entendimiento con el código de los profesores es que se ejecuta hasta la raíz cuadrada de num usar i * i en la condición de la iteración while permite comparar el cuadrado de i con n, lo que vendría a ser lo mismo a comparar i con la raíz cuadrada de n. Si i * i es mayor que n, entonces i es mayor que la raíz cuadrada de n y se sale del ciclo. Si i * i es menor o igual que n, entonces i es menor o igual que la raíz cuadrada de n, lo que significa que se debe seguir iterando hasta que se hayan probado todos los posibles divisores menores o iguales que la raíz cuadrada de n lo que claramente reduce mucho el número de iteraciones y por lo tanto su notación es la misma solo que se pone en términos de la raíz cuadrada de num O($\sqrt{num}$).