Estructuras de Datos
Profesores: Gonzalo Noreña - Carlos Ramírez
Estudiantes: Samuel Francisco Moncayo Moncayo, Samuel Rueda

Tarea #2

Punto 1:

A) 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;
   }
}
```

Línea 1: 1

Línea 2:

$2log_2$ n+2

Línea 3:

$2log_2 n + 1$

Línea 4:

$2log_2 n + 1$

Línea 5:

$2log_2 n + 1$

T(n): 1+ ($2log_2$ n+2) + ($2log_2$ n+1) + ($2log_2 n + 1$) + ($2log_2 n + 1$)

=$8log_2 n + 6$

Complejidad: O(log_2 n)

B)
Al ejecutar el código, empieza en el ciclo (for) con i = por ende cuando n = 8 al momento de operar será obteniendo que i = 64 y j = 1, mientras el valor de i sea mayor a 0 el ciclo ópera (i) = i / 2, imprimiendo la suma entre i + j que sería 65, asimismo, cuando el ciclo avanza j aumentará 1 y i será igual a i/. K es el número de veces que sea capaz de realizarse la división entera del número, llegando a la conclusión de que el ciclo va a iterar 2^{6+1} veces, ya que, cuando llega a $2^6 = 64$ el número es reducido a 2, sin embargo, como el ciclo itera mientras que i > 0 consideramos que k = 7, a medida que se desarrolla el ciclo se irá imprimiendo la suma hasta (k) veces, en la que al final obtenemos 2 veces el número (n).

respuesta:
65 = 64 + 1
34 = 32 +2
19 = 16 +3
12 = 8 + 4
9 = 4 + 5
8 = 2 + 6
8 = 1 +7

Punto 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;
}
```

línea 1:

1

Línea 2:

($\frac{n}{2}$) + 1

Línea 3:

($\frac{n}{2}$+ n) $\sqrt{n}$

Línea 4:

($\frac{n}{2}$) $\sqrt{n}$

Línea 5:

1

T(n)= 1+ ($\frac{n}{2}$ + 1) +($\frac{n}{2}$ + n $\sqrt{n}$) +($\frac{n}{2}$) $\sqrt{n}$ +1

$O(n^{1/3})$

¿Qué se obtiene al ejecutar algoritmo 2(8)? Explique.

se obtiene que res = 16

si n es = 8

Quiere decir que i va aumentar en 4 unidades hasta que el valor sea <= 16

por lo cual el cuerpo del primer ciclo se ejecutaría 4 veces. Dentro del primer ciclo va un segundo ciclo en el cual se revisa si j al cuadrado es menor o igual a n(8) por lo cual en la primera iteración sería 1 <= 8 en la segunda 4 <= 8 y en la tercera 9 <= 8 en la última iteración la condición no se cumple por lo cual en cada iteración de i el código res + 2 se va a ejecutar 2 veces por lo cual 4*2*2 siendo 4 las veces que se ejecuta el primer ciclo 2 las veces que se ejecuta el segundo ciclo por cada iteración del primero y el último 2 siendo las unidades en las que aumenta la variable res.

Punto 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");
}
```

Línea 1:

1

Línea 2:

n

Línea 3:

*(n-1) * (n+1)*

Línea 4:

n^2*-1* $(\sum_{i=1}^{n} i + 1)$ *= n+1*$(n/2)$

línea 5:

n^2*-1*$(\frac{n^2}{2})$

$T(n^2)$

Punto 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;
}
```

Línea 1: 1

Línea 2: 1

Línea 3: *n + 1*

LÍnea 4: *n*

Línea 5: *n*

Línea 6:

Línea 7

¿Qué calcula esta operación?

el algoritmo recibe un arreglo y su respectivo tamaño. El programa suma los valores de cada elemento del arreglo aun contador en caso de que estén ordenados de menor a mayor. Si uno de los números es menor a su anterior inmediato el contador debe sumar uno y continuar recorriendo el arreglo sin sumar el valor del elemento. Finalmente el programa retorna el valor de las sumas (el contador)

}

Punto 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;
  }
}
```

Línea 1:

1

Línea 2:

7

Línea 3:

6

Línea 4:

6

T(n) = 1 + 7 + 6 + 6 = 20 – Es una función constante por lo que O(1) sin importar el dato introducido

Punto 6:

Fibonacchi.py > ...

```python
#!/usr/bin/env python3


def calcularFibonacci(numero):
    #Función para calcular el numero de la secuencia de fibonacci
    if numero <= 1:
        return numero
    else:
        #Al llamarse a si misma, evalua con recursión simulando el trabajo de 2 ciclos en una lista
        #Evalua el valor anterior de la secuencia para sumarlo con el nuevo valor
        return (calcularFibonacci(numero-1) + calcularFibonacci(numero-2))

```

Tamaño de Entrada	Tiempo	Tamaño de Entrada	Tiempo
5	0m0.024s	35	0m12.813s
10	0m0.035s	40	2m14.130s
15	0m0.105s	50	CPU max
20	0m0.040s	55	CPU max
25	0m0.149s	60	CPU max
30	0m2.036s	100	CPU max

¿Cuál es el valor más alto para el cuál pudo obtener su tiempo de ejecución?

R/ El valor más alto que logró fue 40 en el tamaño de entrada

¿Qué puede decir de los tiempos obtenidos?

R/ El tamaño de entrada es directamente proporcional al tiempo de ejecución, por ello entre más valores tenga que determinar la función, más tiempo demorara su lectura

¿Cuál cree que es la complejidad del algoritmo?

R/ Probablemente sea cúbica o mayor, ya que, cuando los valores de entrada son mayores aumenta la cantidad de números a sumar, sin contar el hecho de que cada vez serán números mucho más grandes, ya que, triplica o incluso más los números propuestos.

Punto7:

Código:

```
1
2
3   def calcularFibonacci(numero):
4       numeroFibo = [0,1]
5       x=0
6       while x < numero:
7           # A medida que aumenta x, toman valores en la lista de numerofibo.
8           numFibonacci = numeroFibo[x] + numeroFibo[x+1]
9           # Suma ambos indices de la lista y añaden el nuevo valor a la misma
10          # Al regresar x tomaria el valor que le correspondia a [x+1] y x+1 el valor de numFibonacci
11          numeroFibo.append(numFibonacci)
12          x +=1
13      return numeroFibo[-1]
14  print(calcularFibonacci(10))
```

Tamaño de Entrada	Tiempo	Tamaño de Entrada	Tiempo
5	0m0.037s	45	0m0.108s
10	0m0.038s	50	0m0.054s
15	0m0.047s	100	0m0.062s
20	0m0.067s	200	0m0.035s
25	0m0.044s	500	0m0.049s
30	0m0.057s	1000	0m0.057s
35	0m0.040s	5000	0m0.052s
40	0m0.049s	10000	0m0.050s

Punto 8:

Tamaño Entrada	Tiempo solución propia	Tiempo solución profesores
100	**0m0.039s**	0m0.030s
1000	**0m0.112s**	0m0.031s
5000	**0m2.419s**	0m0.044s
10000	**0m10.650s**	0m0.044s
50000	**CPU MAX**	0m0.511s
100000	**CPU MAX**	0m1.077s
200000	**CPU MAX**	**0m2.913s**

Responda las siguientes preguntas:

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

R/ En los dos primeros tamaños de entrada no se distanciaron mucho del otro, sin embargo, cuando los datos de entrada empiezan a rondar más allá de 1000, mi programa saca a relucir su ineficiencia a diferencia del otro código, que sin ningún tipo de problema ejecuta el programa, la gran determinante entre los dos es la cantidad de ciclos anidados

que utilizo en mi código, volviendo mucho más complejo mi programa, concluyendo que la complejidad y por ende el tiempo de ejecución de mi código es mucho mayor al de los profes.

(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/

Código propio:

```
        num = True
        for divisor in range(2, numeroPrimo):
            #Evalua cada divisor del numero reque
            #En caso de no ser exacta el numero e
            if numeroPrimo % divisor == 0:
                num = False
        if num is True:
            print("--> %d,"%(numeroPrimo))
```

Línea 1: *1*

Línea 2: *n-1*

Línea 3: n-2

Línea 4:

Mejor Caso:

n = 1

Peor Caso:

n-2

Línea 5: 1

Línea 6:

Mejor caso:

0

Peor caso :

1

Peor caso: T(n) 1 +(n-1) + (n-2) +(n-2)+1 = O(n)

Mejor caso: T(n) 1 +(n-1) + (n-2) +1 +0 = O(n)

Código Profes:

```
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
```

Mejor caso : O(1)

Peor caso : O(n^1/2)