

SOLUCIÓN TAREA 2- SANTIAGO ARANGO HENAO – RICHARD OGGIONI

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;
    }
}
```

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

Solución:

Primera Línea: Se ejecuta 2 veces.

Segunda Línea: Toca analizar el aumento de i hasta que sea mayor que 0, en cada iteración

$i = i / 2$, sería entonces $1 = i / 2^k$, solucionando esta ecuación: da como resultado $\log_2 n + 1 = k$

Tercera Línea: Como se ejecuta las veces que entra al ciclo, y el ciclo se ejecuta $\log_2 n$, esta se ejecutará $\log_2 n$

Cuarta Línea y Quinta línea: También se ejecutará $\log_2 n$.

Por lo tanto, $T(n) = 2 + \log_2 n + 1 + \log_2 n + \log_2 n + \log_2 n = 3 + 4(\log_2 n)$.

La complejidad del algoritmo es $O(\log_2 n)$.

Al ejecutar algoritmo1 (8) se obtiene:

65 33 17 9 5 3 2 1, ya que el código, primero eleva al cuadrado el parámetro, consecutivamente le suma uno y saca la mitad hasta que el ciclo acabe.

i=64 j=1 suma=65

i=32 j=2 suma=34

i=16 j=3 suma=19

i=8 j=4 suma=12

i=4 j=5 suma=9

i=2 j=6 suma=8

i=1 j=7 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;  
}
```

Página 1 de 4

estructuras de Datos
profesor: Gonzalo Noreña - Carlos Ramírez

Tarea 2

```
    return res;  
}
```

Solución:

Primera línea se ejecuta 3 veces.

Segunda línea se ejecuta hasta que $i \leq 2 * n$, entonces por tanteo:

Si $n = 5$, i empieza en 1, después en 5 porque la i aumenta en 4 cada iteración, posteriormente le suma $i += 4$ que sería 9, allí acabaría la iteración, en total se hicieron 3 iteraciones, en 2 entra y la última comprueba pero no entra.

Probemos con $n = 10$, pasaría lo mismo solo que iteraría 4 veces, 3 veces entra y la otra verifica.

Con $n = 50$, empezaría $i = 1, 4, 9, 16, 25, 36, 49, 64$, entraría 7 veces (hasta 49) pero en la última solo verificaría, pero no entra.

De todo esto podemos deducir que el primer ciclo va:

$$\frac{2n}{4} = \frac{n}{2}$$

En el segundo ciclo, iría mientras la condición $j * j \leq n$, nuevamente por tanteo:

$n = 50$ y la j incrementa 1 en 1, es decir, $(j^2) + 1$.

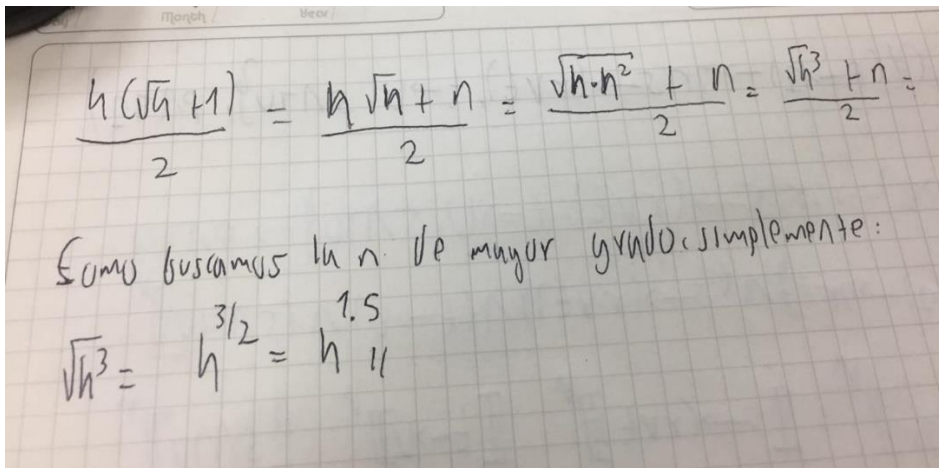
En la primera j sería, 1, 4, 9, 16, 25, 36, 49, **64**. Iteraría 8 veces pero en la última no entra al for, solo verifica la condición mas no entra.

Si fuera $n = 16$, iría $j^2 = 1, 4, 9, 16$. Iteraría 4 veces pero en la ultima no entra al ciclo.

Podemos deducir entonces que el segundo for va hasta:

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

En conclusión, operando:



Handwritten derivation on grid paper:

$$\frac{n(\sqrt{n} + 1)}{2} = \frac{n\sqrt{n} + n}{2} = \frac{\sqrt{n} \cdot n^2 + n}{2} = \frac{\sqrt{n}^3 + n}{2}$$

Como buscamos la n de mayor grado, simplemente:

$$\sqrt{n}^3 = n^{3/2} = n^{1.5}$$

Así que la complejidad del algoritmo es, $O(n^{3/2})$.

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");
}
```

Solución:

Primera línea se ejecuta 3 veces.

Segunda línea se ejecutaría, el ciclo se ejecuta cuando $i > 1$ pero $n = i$.

Con $n = 5$, se iteraría 4 veces pero sin entrar a la última, así: 5,4,3,2,1

Con $n = 10$, pasa lo mismo, se iteraría 9 veces, pero la última no entra.

Con $n = 20$, se iteraría 19 veces.

Como se puede apreciar, el primer for va hasta que:

$$n - 1$$

Tercera línea, se ejecutaría tantas veces multiplicado con el ciclo anterior, este for va hasta que $j \leq n$, empezando con $j = 1$ y sumando $j++$, sería el mismo for anterior solo que, al contrario, de esta manera:

$$(n - 1)(n + 1) = (n^2) - 1$$

La cuarta línea, sería todo lo anterior pero este for se cumple cuando $k \leq i$, empezando desde $k = 1$ y en cada iteración $k++$, fíjese que i es mayor que 1 y la condición va hasta que $k \leq i$, es decir hasta que hagan match, fácilmente se puede hacer con una sumatoria, de esta manera:

$$\sum_{k=1}^i k + 1 = \frac{n(n+1)}{2}((n^2) - 1)$$

$$\frac{n(n+1)}{2} (n^2-1) = \frac{(n^2+n)(n^2-1)}{2} = \frac{n^4 - n^2 + n^3 - n}{2}$$

Así que la complejidad del siguiente algoritmo es $O(n^4)$

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.

La primera línea se ejecuta 2, solo inicializa 2 variables y le asigna su valor.

La segunda línea 4 veces solo inicializa 4 variables

La tercera línea, $n + 1$ veces.

La cuarta línea se ejecuta n veces ya que $j = i + 1$, cuando el ciclo for cumple la condición.

Lo mismo pasa con la línea 5.

Con la línea 6 cambia la cosa ya que nos encontramos un mejor y peor caso, el mejor caso es cuando la condición no se cumple, sería:

$$(n+1)(n+1)$$

El primer $n+1$ por el primer for y este ciclo while $n+1$ veces.

En cambio, por el peor caso sería:

$$\sum_{i=0}^n j = \frac{(n+0)(n-0+1)}{2} = \frac{n(n+1)}{2}$$

Y recordar que las veces que se ejecuta el while depende del for también:

$$\frac{n(n+1)}{2}(n+1) = \frac{((n^2)+n)}{2} * \frac{(n+1)}{1} = \frac{(n^3)+(n^2)+(n^2)+n}{2}$$

En la séptima línea:

$$\sum_{i=0}^n j$$

En la octava línea, si consideramos únicamente el peor caso sería

$(\sum_{i=0}^n j)$ Por el while y se multiplica con la sumatoria de este for, así:

$$\sum_{i=0}^n \sum_{h=j}^n j$$

El resultado de esta doble sumatoria sería n^2 considerando solo la variable de mayor grado para la complejidad.

En la novena línea sería:

$$\sum_{h=j}^n j$$

Ya que esta dentro del for.

Y el resto de las líneas solo se ejecutan 1 vez.

Por lo tanto, la complejidad del algoritmo es $O(n^3)$ por la sexta línea que produce este cubo.

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;  
    }  
}
```

Solución:

- Primera línea se ejecuta 1 vez
- Segunda línea se ejecuta $(n + 1)$ veces, ya que si observamos el programa va desde i el cual tiene un valor de cero hasta n , donde n es el valor numérico que nosotros colocamos.
- Tercera línea se ejecuta n veces que serian las veces q entra al ciclo
- Cuarta línea se ejecuta n veces y aquí lo mismo, las veces q se entra al ciclo

$$1+n+1+n+n = 3n+2$$

De esta manera definimos que la complejidad del programa es $O(n)$.

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		35	
10		40	
15		45	
20		50	
25		60	
30		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?

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0m0.015s	35	0m3.083s
10	0m0.015s	40	0m26.013s
15	0m0.031s	45	14m5.598s
20	0m0.046s	50	No se puede
25	0m0.125s	60	No se puede
30	0m0.367s	100	No se puede

- A. ¿Cuál es el valor más alto para el cual pudo obtener su tiempo de ejecución?
R// Hasta 45 el programa ya se demora 14 minutos, con 50 sería una ejecución demasiado larga.
- B. ¿Qué puede decir de los tiempos obtenidos?
R// A medida que avanza los tamaños de entrada, el tiempo de ejecución también aumenta, se demora cada vez más porque la función Fibonacci, si se ingresa un 5, primero saca 4 y 3, después a 4 le saca 3 y 2, al 3 le saca 2 y 1, así sucesivamente hasta que sea cero, como si fuera un árbol que cada vez se va desglosando, de esta manera cuando el valor es 35 o 40, el tiempo de ejecución se demora demasiado sacando la serie Fibonacci de cada número.
- C. Según cómo funciona la función de Fibonacci, mientras mas grande sea la entrada, mayor será el tiempo, por lo tanto creería que la complejidad es exponencial. Como si en el sistema de coordenadas, el eje x son los valores de entrada y el eje y son valores del tiempo de ejecución, además

que Fibonacci no puede hacerse con 0 y la grafica de la exponencial no toca el eje $x=0$.

los tiempos obtenidos: ¿Cual cree que es la complejidad del algoritmo?

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		45	
10		50	
15		100	
20		200	
25		500	
30		1000	
35		5000	
40		10000	

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0m0.015s	45	0m0.047s
10	0m0.016s	50	0m0.046s
15	0m0.031s	100	0m0.097s
20	0m0.031s	200	0m0.105s
25	0m0.030s	500	0m0.106s
30	0m0.031s	1000	0m0.113s
35	0m0.047s	5000	0m0.116s
40	0m0.123s	10000	0m0.114s

La complejidad del algoritmo es:

- Primera línea y segunda línea: 2 veces
- Tercera línea se ejecuta n veces.
- Cuarta, quinta, sexta y séptima se ejecutan: 4 veces.

La complejidad del siguiente algoritmo es: $T(n) = 2 + n + 4 = 6 + n$.

Por lo tanto, la complejidad computacional es lineal: $O(n)$.

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		
1000		
5000		
10000		
50000		
100000		
200000		

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?

Tamaño Entrada	Tiempo Solución Propia	Tiempo Solución Profesores
100	0m0.053s	0m0.045s
1000	0m0.128s	0m0.066s
5000	0m1.614s	0m0.084s
10000	0m5.580s	0m0.136s
50000	1m46.296s	0m0.973s
100000	6m29.602s	0m2.767s
200000	25m5.869s	0m6.344s

- A. ¿Qué tan diferentes son los tiempos de ejecución y a que 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?

R//:

- a) Los tiempos de ejecución son extremadamente diferentes se puede llegar a notar en el cuadro como se va elevando cada vez más llegando a un punto en el que uno se demora minutos y el del profe siguen siendo segundos, aunque aumente la cantidad.

Nosotros creeríamos que la principal diferencia entre los códigos es el manejo de los ciclos y sus cantidades, además de algún q otro condicional q permite que conseguir de mejor manera los valores necesitados sin necesidad de colocar tantísimos como hay en el de nosotros para q el programa consiga los primos. Por estas cosas el programa del profe llega a ser menos complejo y más versátil a la hora de ejecutar el código, haciendo provecho de buena manera o mejor manera el programa.

- b) Nosotros diríamos que en el bloque del profesor la complejidad sería $O(n)$ la verdad solo el punto que hace los primos es bastante corto y al separarlo en una sola función es mejor y evita juntarlo con el mostrar primos queda mejor organizado y se vea menos complejo, aunque también hay q tener en cuenta q se debe de cumplir la condición para entrar al while y eso también dependerá de la cantidad de n números que queramos revisar si son primos.

A diferencia, el de nosotros ejecuta un ciclo extra algo q ya aumenta su complejidad, además de unos cuantos condicionales más, llegados a este punto podemos pensar que la complejidad del programa de primos llega a ser $O(n^2)$ aunque también dependerá de la cantidad de n q metamos.

Profe:

```
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
```

Nosotros:

```
def mostrarPrimos(N):  
  
    print("Numeros primos entre 1 y 100")  
  
    listaPrimos = []  
  
    for i in range(2,N):  
        j = 2  
        bandera = 0  
        contador = 0  
        while bandera == 0 and j < i:  
            if i % j == 0:  
                contador += 1  
            if contador == 1:  
                bandera = 1  
            j+=1  
  
        if bandera == 0 or i == 2:  
            listaPrimos.append(i)
```