

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

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

```
void algoritmo1(int n){
    1 int i, j = 1; → 2
    2 for(i = n * n; i > 0; i = i / 2) { → 2 log₂ n + 2
        3     int suma = i + j; → 3
        4     printf("Suma %d\n", suma); → 3(2 log₂ n + 1)
        5     ++j; → 3(2 log₂ n + 1)
    }
}
```

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

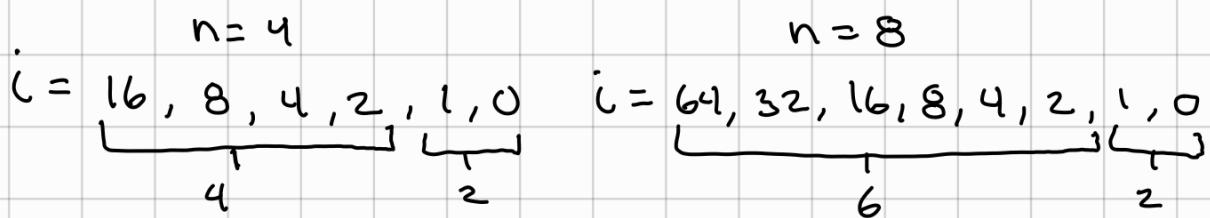
- Indique el número de veces que se ejecuta cada línea y determine cuál es la complejidad del siguiente algoritmo:

```
int algoritmo2(int n){
    1 int res = 1, i, j; → 3
    2 for(i = 1; i <= 2 * n; i += 4) → n/2 + 1
        3     for(j = 1; j * j <= n; j++) → (n/2)(√n + 1)
            4         res += 2; → n √n
}
```

## Punto 1

Se analizó con base a  $n=4$  y  $n=8$

### Línea 2:



Se puede apreciar que los pasos al  $\log_2 i_0 + 2$ , y como se sabe  $i_0 = n^2$ , reemplazando:

$$\log_2 n^2 + 2 = 2 \log_2 n + 2$$

### Línea 3, 4, 5

Se ejecutarán cuantas veces ingrese el ciclo, es decir

$$2 \log_2 n + 1$$

### $T(n)$ en el programa:

$$T(n) = 2 + 2 \log_2 n + 2 + 3(2 \log_2 n + 1)$$

$$T(n) = 8 \log_2 n + 7$$

$$T(n) = O(\log_2 n)$$

### Respuesta:

Al ejecutar algoritmo(8) se imprimirá el acumulador suma, en el que se suman  $i+j$ , es decir,  $64+1, 32+2, 16+3\dots 0+8$ , y tomará un total de  $8 \log_2 8 + 7 = 31$  pasos

## Punto 2

### Línea 2

llamaremos m condición de parada del for

$$n = 6$$

$$m = 12$$

$$i = 1, 5, 9, 13$$
  
$$\underbrace{1}_{3}, \underbrace{5}_{1}$$

$$n = 10$$

$$m = 20$$

$$i = 1, 5, 9, 13, 17, 21$$
  
$$\underbrace{1}_{5}, \underbrace{17}_{1}$$

Como se aprecia la linea del ciclo se repetirá  $\frac{n}{2} + 1$

### Línea 3

Primeramente se analiza el ciclo anidado

$$n = 16$$

$$j = 1, 2, 3, 4, 5$$
  
$$\underbrace{1}_{4}, \underbrace{2}_{1}$$

$$n = 9$$

$$j = 1, 2, 3, 4$$
  
$$\underbrace{1}_{3}, \underbrace{2}_{1}$$

$$n = 8$$

$$j = 1, 2, 3$$
  
$$\underbrace{1}_{2}$$

Como se aprecia la linea se repetirá  $\sqrt{n} + 1$  veces

Ahora bien, como este anidado, se esa linea se repetirá cuantas veces se ingrese al ciclo en el cual se anida, por lo tanto

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

### Línea 4

Se repite cuantas veces se ingrese en los ciclos superiores, es decir

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

## T(n) en el Programa

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

$$T(n) = n^{3/2} + n + 4$$

$$T(n) = O(n^{3/2})$$

## Respuesta

Al ejecutar algoritmo 2(8) se obtiene el valor del acumulador res, que suma 2 unidades cada vez que se repite, por lo tanto el valor que se obtiene al ejecutar el programa con  $n=8$  es 17, ya que se repite  $\frac{n\sqrt{n}}{2}$  veces, esto teniendo en cuenta que  $\sqrt{8} \approx 2$  entonces  $8 \cdot 2 / 2 = 8$ , por lo que al ejecutarse 8 veces se suma  $2 \cdot 8$  al los, es decir  $res = 1 + 16$

```

    return res; 1
}

```

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

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

```

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

```

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; → 2
    int i, j, h, flag; → 4

    for(i = 0; i < n; i++) { → n+1
        j = i + 1; → n
        flag = 0; → n Mejor
        while(j < n && flag == 0){ → 2n
            if(valores[i] < valores[j]){ → n
                for(h = j; h < n; h++){ → 0
                    suma += valores[i]; → 0
                }
            }
            else{ → 0
                contador++; → n
                flag = 1; → n
            }
            ++j; → n
        }
    }
    return contador; → 1
}

```

¿Qué calcula esta operación? Explique.

### Punto 3

Analizaremos este punto con base a  $n = 4$

#### Línea 2:

$n = 4$  por lo tanto, el for se repite  $n$  veces  
 $i = \underbrace{4, 3, 2, 1}_4$

Línea 3: Primero se analiza el for aislado

$n = 4$   
 $j = \underbrace{1, 2, 3, 4, 5}_5$  por lo tanto, este for se repite  $n + 1$  veces

Ahora bien, como es anidado se repetirá  $(n-1)(n+1)$  veces

Línea 4: como este for depende del valor de  $i$  entonces se genera una sumatoria que se repite  $n$  veces con la misma  $i$  por el segundo for

$n = 4$        $i = 3$        $i = 2$       En total son  
 $i = \underbrace{4, 3, 2, 1}_4$        $K = \underbrace{1, 2, 3, 4}_4$        $K = \underbrace{1, 2, 3}_3$        $12 n$  pasos que equivalen a  $12 \cdot 4 = 48$

#### Desarrollo de sumatorias

$$\begin{aligned} n \cdot \sum_{i=2}^n i+1 &= n \sum_{i=2-1}^{n-1} (i+1) + 1 = n \sum_{i=1}^{n-1} i+2 = n \left( \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 2 \right) \\ n \left( \frac{(n-1) \cdot n}{2} + (n-1) \cdot 2 \right) &= n \left( (n-1) \left( \frac{n}{2} + 2 \right) \right) = n \left( \frac{(n-1)(n+4)}{2} \right) = n \left( \frac{n^2 + 3n - 4}{2} \right) \\ &= \frac{n^3 + 3n^2}{2} - 2n \end{aligned}$$

## Línea 5

Para esta línea se ejecuta cuantas veces entre en el último for, es decir:

$$n \cdot \sum_{i=2}^n i$$

## Desarrollo de sumatoria

$$\begin{aligned} n \cdot \sum_{i=2}^n i &= n \cdot \sum_{i=1}^{n-1} i + 1 = n \left( \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \right) = n \left( \frac{(n-1) \cdot n}{2} + n-1 \right) \\ &= n \left( \frac{n^2 - n}{2} + n - 1 \right) = n \left( \frac{n^2}{2} + \frac{n}{2} - 1 \right) = \frac{n^3 + n^2}{2} - n \end{aligned}$$

## T(n) en todo el programa:

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

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

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

$$T(n) = O(n^3)$$

# # Punto 4

## Proceso del while

Evaluar un caso particular:

$$n=4$$

$$\begin{array}{cccc} i=0 & i=1 & i=2 & i=3 \\ j = \underbrace{1, 2, 3, 4}_{4} & j = \underbrace{2, 3, 4}_{3} & j = \underbrace{3, 4}_{2} & j = \underbrace{4}_{1} \end{array}$$

## Desarrollo de la sumaatoria:

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

## Comprobación:

Si se evalúa el  $n=4$  entonces:

$$\frac{(4-1) \cdot 4}{2} + 4 = 10$$

## Proceso del for

$$n=4 \wedge i=0$$

$$\begin{array}{ccc} j=1 & j=2 & j=3 \\ h = \underbrace{1, 2, 3, 4}_{4} & h = \underbrace{2, 3, 4}_{3} & h = \underbrace{3, 4}_{2} \end{array}$$

Pasos totales para h cuando  $i=0 \rightarrow j=1, 2, 3, 4$  es 9 pasos

## Desarrollo de sumatoria Aislada

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

## Comprobación:

Si  $n=4$  entonces:

$$\begin{aligned} &= \frac{(4-1) \cdot 4}{2} + (4-1) \\ &= 9 \end{aligned}$$

## Sumatoria total

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

## Propiedades Sumatoria:

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

$$2 \quad \sum_{k=1}^m a_k + b_k = \sum_{k=1}^m a_k + \sum_{k=1}^m b_k$$

$$3 \quad \sum_{k=1}^m 1 = m$$

$$4 \quad \sum_{k=0}^m 1 = m+1$$

Si evaluamos el  $n=4$   
nos damos cuenta que es una  
sumatoria dos 1 hasta  $n$ ,  
dando resultado lo pasos

## Analisis del Peor caso

El peor caso es cuando siempre se cumple que  $\text{valores}[i] < \text{valores}[j]$ , ya que cumple la función del if, y no culmina el while hasta que  $j = n$ , ya que el flag nunca cambia, un ejemplo de este caso es un arreglo que esté organizado de menor a mayor, por ejemplo  $\{1, 2, 3, 4, 5\}$ , en que  $\text{valor}[i]$  siempre es menor al resto del arreglo

### $T(n)$ en el Peor caso:

$$\begin{aligned} T(n) &= 6 + (n+1) + 2n + \sum_{i=0}^{n-1} i+1 + 2 \left( \sum_{i=0}^{n-1} i \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j+1 \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j \right) + 1 \\ T(n) &= 8 + 3n + \left( \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 1 \right) + 2 \left( \frac{(n-1)n}{2} \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j+1 \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j \right) \\ T(n) &= 8 + 3n + \left( \frac{(n-1)\cdot n}{2} + n \right) + 2 \left( \frac{n^2-n}{2} \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j+1 \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j \right) \\ T(n) &= 8 + 3n + \left( \frac{n^2-n}{2} + n \right) + (n^2-n) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j+1 \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j \right) \\ T(n) &= 8 + 3n + \left( \frac{n^2}{2} - \frac{n}{2} + n \right) + (n^2-n) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j+1 \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j \right) \\ T(n) &= 8 + 3n + \frac{n^2}{2} + \frac{n}{2} + n^2 - n + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j+1 \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j \right) \\ T(n) &= \frac{3n^2 + sn + 16}{2} + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j+1 \right) + \sum_{i=0}^{n-1} \left( \sum_{j=1}^{n-1} j \right) = \mathcal{O}(n^3) \end{aligned}$$

Es  $\mathcal{O}(n^3)$  ya que la doble sumatoria dará como resultado un  $n^3$

### Mejor Caso:

El mejor caso, es que siempre se de que  $\text{valores}[i] \geq \text{valores}[j]$ , es decir un arreglo que esté organizado de mayor a menor, o que en su efecto tengo valores iguales, ya que al comparar se ejecuta el contenido del else y el valor de flag cambia, lo que termina el while. Por ejemplo el arreglo  $\{5, 4, 4, 3, 2\}$  daría el mejor caso.

$$T(n) = 2 + 4 + (n+1) + 9n + 1$$

$$T(n) = 10n + 8 = O(n^1)$$

### ¿Qué calcula esta operación?

El algoritmo 4 recibe un arreglo (valores) y el tamaño de ese arreglo (n). Lo que hace este algoritmo de forma inicial es definir las variables de contador y variables para recorrer dicho arreglo. En el primer for se define  $i = 0$  hasta que  $i < n$ , de este modo se puede recorrer el arreglo con el índice [i], acto seguido se define  $j = i + 1$ , que será usado para obtener el valor inmediatamente seguido del índice [i]. Ahora bien, con esto definido se procede a entrar al while, donde se hace la comparación del valor en la posición  $\text{valores}[i]$  con el resto del arreglo, buscando valores que sean mayores, si se encuentra un valor menor que sea posterior a  $\text{valores}[i]$  entonces se sumará en el acumulador suma el valor de  $\text{valores}[i]$  cuantas veces  $h < n$ , es decir, cuantas posición hay entre  $\text{valores}[i]$  hasta el ultimo valor del arreglo. En el caso que se encuentre un valor posterior a  $\text{valores}[i]$  que sea mayor o igual a él, se sumará 1 en el contador y se cambia  $\text{flag} = 1$ , por lo que dará por acabado el while, al culminar todo este proceso se retorna contador, es decir, el total de veces que se encontró un valor que es mayor al  $\text{valores}[i]$ .

Un ejemplo de esto podría ser el arreglo  $\{2, 3, 1\}$  de tamaño  $n = 3$ , por lo que al comparar en  $i = 0$  y  $j = 1$ , nos damos cuenta que  $\text{valores}[i] < \text{valores}[j]$ , porque  $2 < 3$ , de este modo se suma 2 al contador suma un total de 2 veces, ya que es el numero de valores posteriores a él, de esta forma  $\text{suma} = 4$ . Ahora el while ya dio su primer recorrido y cambia  $j = 2$ , ahora compara  $\text{valores}[0]$  con  $\text{valores}[2]$ , en este caso  $\text{valores}[i] > \text{valores}[j]$ , por lo que se suma 1 el contador y se termina el while. Si se continua el proceso se obtiene al finalizar contador = 1, e internamente suma quedará como  $\text{suma} = 7$ .

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

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

### 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	0m0.126s	35	0m7.339s
10	0m0.120s	40	1m11.051s
15	0m0.129s	45	12m7.298s
20	0m0.127s	50	62m46.298s
25	0m0.236s	60	
30	0m0.174s	100	

Dato impreciso ya que  
se tuvo que cortar el proceso

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?

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	0m0.175s	45	0m0.189s
10	0m0.169	50	0m0.192s
15	0m0.185	100	0m0.166
20	0m0.127s	200	0m0.184s
25	0m0.236s	500	0m0.192s
30	0m0.174s	1000	0m0.199s
35	0m0.186s	5000	0m0.964s
40	0m0.174s	10000	0m3.386s

## Punto S

$$n = 25$$

$$i = \underbrace{0, 5, 10, 15, 20, 25}_{7}, 30$$

$$n = 10$$

$$i = \underbrace{0, 2, 4, 6, 8, 10}_{7}, 12$$

$$n = 50$$

$$i = \underbrace{0, 10, 20, 30, 40, 50}_{7}, 60$$

$$n = 8$$

$$i = \underbrace{0, 1, 2, 3, 4, 5, 6, 7}_{10}, 8, 9$$

Si realizamos este proceso haciendo  $n$  cada vez más grande entonces nos daremos cuenta que siempre se verificará 7 veces

## Peor Caso: ( $n < s$ )

Cuando  $n < s$  el while sería infinito pues al hacer la división entera entre  $s$ , el valor será 0.

## Mejor Caso ( $n \geq s$ )

En el mejor caso para la mayoría de  $n$ 's

$$T(n) = 20, \text{ por lo que su complejidad sería } T(n) = O(n^0)$$

## Casos especiales

Hay ciertos valores de  $n$  en que los pesos quedarán entre 7 y 10, Sin embargo no cambia el hecho que es constante  $T(n) = O(n^0)$

## Punto 6:

```
def fibonacciRecursion(number):
    ans = None
    if number < 2:
        ans = number
    else:
        ans = (fibonacciRecursion(number - 1) + fibonacciRecursion(number - 2))
    return ans
fibonacciRecursion(5)
```

El fibonacci de recursión permitió ejecutar con un valor de tiempo razonable hasta fibo(45) con un tiempo de 12 minutos. Después de ese valor se realizó la ejecución de fibo(50), no obstante después de una hora de ejecución no se terminaba el proceso por lo que se tuvo que cortar.

Por otro lado los tiempos de ejecución fueron cortos hasta fibo(30), después de este fibo el valor del tiempo de ejecución crecerá exponencialmente a medida que crece "n", haciendo que haya una relación directamente proporcional entre n y el tiempo de ejecución, lo que nos permite concluir que la complejidad es  $O(2^n)$ .

## Punto 7:

```
def fibo(num):
    inicial = 0
    siguiente = 1
    ans = 0
    for i in range(num + 1):
        if i < 2:
            ans = i
        else:
            secuencia = inicial + siguiente
            inicial = siguiente
            siguiente = secuencia
            ans = siguiente
    return ans
```

Complejidad  $O(n)$

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	0m0.200s	0m0.111s
1000	0m0.232s	0m0.127s
5000	0m1.256s	0m0.110s
10000	0m1.943s	0m0.128s
50000	0m4.711s	0m0.204s
100000	3m9.902s	0m0.330s
200000	13m1.747s	0m0.733s

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?

(a). Los tiempos de ejecución en los dos primeros casos de prueba no se separaban mucho entre ellos, sin embargo a medida que se aumentaba el dato de entrada se aprecia que los datos se empiezan a distanciar, siendo más efectivo el tiempo de ejecución de los profesores que el de los estudiantes. Esta diferencia se debe a que en una función auxiliar se utilizo un ciclo for que hacia doble trabajo, ya que primero convertia los enteros a cadena y despues de tomar cada elemento de la cadena, lo convertia de nuevo en un entero para hacer la suma entre esos. Además otro factor que lo vuelve menos efectivo es que en vez de utilizar la lista que ya esta con todos los numeros primos para imprimir los numeros que sumando sus cifras también son primos, se creó otra lista haciendo que el ciclo tenga que iterar mas veces de las necesarias. Finalmente para los valores de esa ultima lista extra que se creó también se hizo un ciclo que recorre sus elementos para imprimirlos con el formato.

(b). La siguiente es la solución de los estudiantes:

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

Esta función "esPrimo" es lineal y tiene complejidad O(n).

Explicación:

El ciclo for recorre todos los elementos de "num" partiendo desde "2" hasta "num - 1". Por lo tanto el numero de iteraciones depende directamente de "num", haciendo que el tiempo de ejecución crezca linealmente.

Por otro lado, la solución de los profesores tiene complejidad  $O(\sqrt{n})$  ya que el "While" corre hasta que " $i * i$ " sea menor o igual que "n" o que ans sea False. Como "i" se incrementa en cada iteración genera que el numero de iteraciones depende de la raiz cuadrada de "n". Esto hace que al final sea en terminos de tiempo mas corta que la solucion de los estudiantes ya que la anterior lo que hacia era depender directamente de "num" en cambio es este tiempo de ejecución crece mas lento con el valor de "n".