SOLUCIÓN TAREA 2- SANTIAGO ARANGO HENAO – RICHARD OGGIONI

Text, letter

Description automatically generated

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 , solucionando esta ecuación: da como resultado

Tercera Línea: Como se ejecuta las veces que entra al ciclo, y el ciclo se ejecuta

, esta se ejecutará

Cuarta Línea y Quinta línea: También se ejecutará .

Por lo tanto, T(n) = 2 + +4(

La complejidad del algoritmo es O (.

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

Text

Description automatically generated with low confidence

Solución:

Primera línea se ejecuta 3 veces.

Segunda línea se ejecuta hasta que i <= 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:

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

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

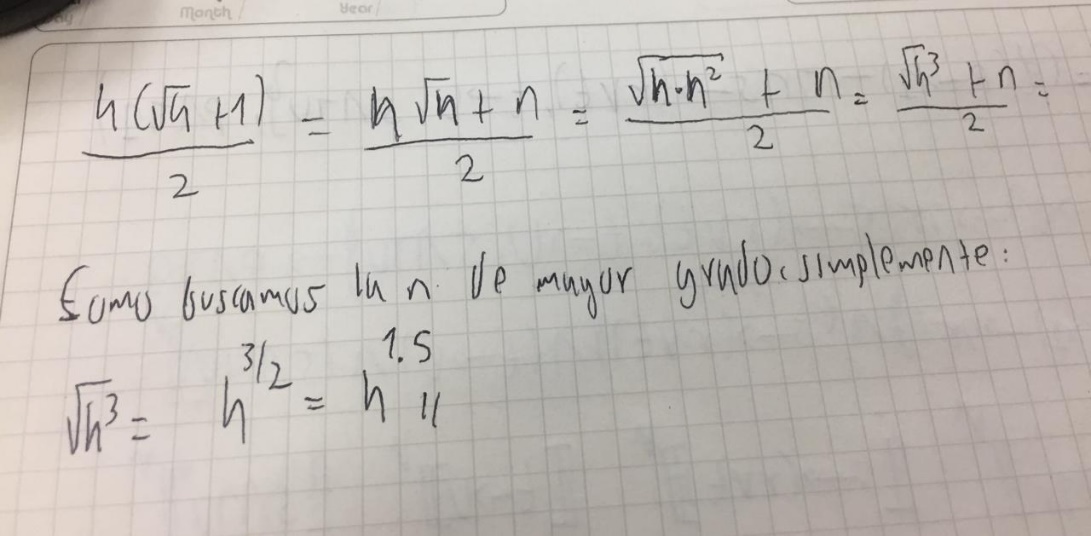
En la primera j sería, 1, 4, 9, 16, 25, 36, 49, **64.** Iteraría 8 veces pero en la última no entra al for, solo verifica la condición mas no entra.

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:

)

En conclusión, operando:



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

Text, letter

Description automatically generated

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.

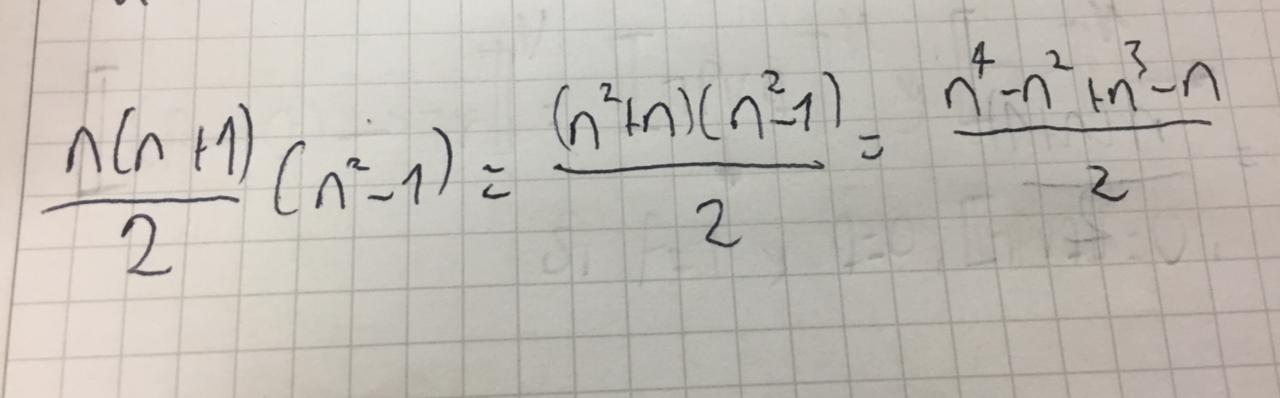
Como se puede apreciar, el primer for va hasta que:

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

= (n^2)- 1

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

= ((n^2)- 1)



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

Text, letter

Description automatically generated

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:

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

(n+1) ==

En la séptima línea:

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

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

En la novena línea seria:

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.

Table

Description automatically generated

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).

Table

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

1. ¿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.

1. ¿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.

1. 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.

Table

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| 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).

Table

Description automatically generated

Tabla

Descripción generada automáticamente

1. ¿Qué tan diferentes son los tiempos de ejecución y a que cree que se deba dicha diferencia?
2. ¿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//:

1. 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.

1. Nosotros diríamos que en el bloque del profesor la complejidad seria 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(n2) aunque también dependerá de la cantidad de n q metamos.

Profe:

Texto, Carta

Descripción generada automáticamente

Nosotros:

