**Problemas Tema 1**

Adrián Borges Cano y Marco González Martínez

**3) Analizar la eficiencia del siguiente código:** Texto, Carta

Descripción generada automáticamente



1: 1 Declaración (1 ejecución)

2: 3 Declaraciones (1 ejecución)

3: 1 Asignación (1 ejecución)

4: 1 Bucle for + 1 Asignación (1 ejecución)

CUERPO DEL BUCLE: 1 Asignación + 1 Suma (n ejecuciones)

5: 1 Condicional + 1 SUMA + 1 División Entera + 1 Comparación (1 ejecución)

CUERPO DEL CONDICIONAL-T: 1 Devolución

CUERPO DEL CONDICIONAL-F:

7: 1 Resta + 1 División Entera + 1 Suma + 1 Asignación(1 ejecución)

8: 1 Bucle for + 1 Asignación (1 ejecución)

CUERPO DEL BUCLE: (n ejecuciones)

9: 1 Bucle for + 2 Mult + 1 Asignación (n ejecuciones)

CUERPO DEL BUCLE: (3n^2 ejecuciones)

10: 1 Función + 1 Suma + 1 Asignación (3n^2 ejecuciones)

13: 1 Función Recursiva + 1 Suma + 1 Asignación (1 ejecución)

14: 1 Devolución (1 ejecución)

Coste = 1+3+1+(1+2n)+(3+max(1,4+(1+3n+(2+Coste(Minimo))n(3n))+T(Calculo)+2+1))

Coste = 17 + 5n + 3n^2 · (2+Coste(Minimo)) + T(n/2)

Coste = 17 + 5n + 12n^2 + T(n/2)

Para calcular el coste de la llamada recursiva a la función, es necesario analizar que parámetros toman juego en las iteraciones en los bucles de la función. En este caso, la variable *z* no aparece, en ninguna condición de los bucles. El número de iteraciones en este caso depende de la diferencia entre los parámetros (y-x), que llamaremos n por comodidad. Al llamar recursivamente a la función, se introduce como parámetro *x* la variable *t*, a la que se le ha asignado *x + (y-x)/2* así que el valor de *t* pasará a ser la media entre *x* e *y*. De esto se puede deducir que *(y-t) = (x-t)/2*. Y la *n* de la que depende la cantidad de iteraciones pasará a ser n/2 al aumentar el nivel de la recursión.

def minimo(x,y):

if x>=y: # 1 Comparación

return y # 1 Devolución

else:

return x # 1 Devolución

# Coste = 1 + max(1,1)

# Coste = 2

Cálculo del coste final (analizando la recursividad):



**7) Realiza un programa que pida un número positivo al usuario (N) y le diga cuantos primos hay entre 1 y ese número N, y cuantos perfectos hay entre 1 y ese número N. Realiza un análisis de eficiencia y de complejidad.**

def divisores(n):

if n==1: return [1]

i=1

k=n//i

divisores = []

while i<k:

if n%i==0:

divisores.append(i)

if i!=n/i:

divisores.append(n//i)

k=n//i

i+=1

return divisores

def esprimo(n):

ldivisores=divisores(n)

return len(ldivisores)==2 and ldivisores[0]!=ldivisores[1]

def esperf(n):

return sum(divisores(n))==2\*n

n = int(input('Introduce un número positivo: '))

tot\_primos=0

tot\_perfectos=0

for j in range(1,n):

tot\_primos += esprimo(j)

tot\_perfectos += esperf(j)

print(f"Hay {tot\_primos} números primos entre {1} y {n}")

print(f"Hay {tot\_perfectos} números perfectos entre {1} y {n}")

Asumiendo Coste(append) = 1, Coste(len) = 1, Coste(sum) = n

divisores:

6: 1 Condicional + 1 Comparación (1 ejecución)

CUERPO DEL CONDICIONAL-T: 1 Devolución (1 ejecución)

CUERPO DEL CONDICIONAL-F: (1 ejecución)

7: 1 Asignación (1 ejecución)

8: 1 División entera (1 ejecución)

9: 1 Asignación (1 ejecución)

10: Bucle-while (1 ejecución)

CUERPO DEL BUCLE: 1 Comparación ( ejecuciones)

11: 1 Condicional + 1 Módulo + 1 Comparación ( ejecuciones)

CUERPO DEL CONDICIONAL-T: ( ejecuciones)

12: 1 Coste(append) ( ejecuciones)

13: 1 Condicional + 1 División + 1 Comparación ( ejecuciones)

CUERPO DEL CONDICIONAL-T: ( ejecuciones)

14: 1 Coste(append)+ 1 División Entera ( ejecuciones)

15: 1 Asignación + 1 División Entera ( ejecuciones)

16: 1 Asignación + 1 Suma ( ejecuciones)

17: 1 DEVOLUCIÓN (1 ejecución)

Coste = 2 + max(1, (1+1+1+1+ ·(1+3+1+3+2+2+2)+1))

Coste = 7 + 14·

esprimo:

24: 1 Asignación + Coste(divisores) (1 ejecución)

25: 1 Devolución + Coste(len)+ 2 Comparaciones (1 ejecución)

Coste = 1 + 7 + 14 + 1 + 1 + 2

Coste = 11 + 14 + 1

esperf:

32: 1 Devolución + Coste(divisores) + Coste(sum) + 1 Comparación + 1 Mult (1 ejecución)

Coste = 3 + n + 7 + 14

Coste = 10 + 14 + n

34: 1 Asignación (1 ejecución)

36: 1 Asignación (1 ejecución)

37: 1 Asignación (1 ejecución)

39: 1 Bucle for (n ejecuciones)

CUERPO DEL BUCLE (n ejecuciones)

40: 1 Asignación + 1 Suma + T(esprimo) (n ejecuciones)

41: 1 Asignación + 1 Suma + T(esperf) (n ejecuciones)

Coste = 1 + 1 + 1 + n·(2 + 11 + 14log2(n) + n + 2 + 10 + 14 + n)

Coste = 3 + 25n + 14n +2n^2 => O(n) = n^2

En cada iteración del bucle de la función divisores, a la variable k se le asigna el valor de n//i, y posteriormente i es incrementada. De esta forma, la condición del bucle (i<k) se romperá cuando el valor de i>=k, que como k=n/i el mínimo valor que lo puede llegar a romper es   
, por lo que el número de veces que se ejecuta el bucle es .

**9) Realizar una función recursiva que calcule el siguiente sumatorio: S= 1+2+3+4+….+n-1+n. Realiza un análisis de eficiencia y de complejidad.**

def sumatorio (n):

"""

int --> int

OBJ: calcular el sumatorio de 1 hasta n

PRE: n>=1

"""

# Caso de partida: El sumatorio de 1 hasta 1, trivialmente es 1

if n==1: # 1 Comparación

return 1 # 1 Devolución

# Caso general: El sumatorio de 1 hasta n es equivalente a la suma de n más el sumatorio de 1 hasta n - 1

# Se puede desglosar según la progresión de la suma de los n primeros de la siguiente forma:

# 1 + 2 + ... + n-1 + n

else:

return n+(sumatorio(n-1)) # 2 + T(n-1)

# T(n) = 3 + T(n-1)

Cálculo del coste final (analizando la recursividad):

