Serie de Fibonacci y Números primos

Luis Enrique López Nerio Universidad Autónoma de Nuevo León
luiselopeznerio@gmail.com

12 de Octubre del 2017

Abstract

En este reporte se describirán brevemente diferentes algoritmos del área de teoría de números, específicamente algoritmos sobre la serie de Fibonacci y tests de primalidad para números. La serie de Fibonacci es una serie muy estudiada por los matemáticos durante varios siglos, por lo que es bueno contar con algoritmos que nos ayuden a calcular sus término de una manera eficiente computacionalmente; dentro de los algoritmos de Fibonacci se introducirá el concepto de recursividad.

Los números primos también son objetos muy estudiados por los matemáticos dentro del área de teoría de números, los primos son importantes por diversas razones siendo una de ellas su utilización en el área de criptografía, dentro de este reporte se dara un algoritmo para probar si un número es primo o no y se detallara su complejidad computacional.

I. Serie de Fibonacci

Los n+úmeros de Fibonacci son la secuencia de números $\{F_n\}_{n=1}^{\infty}$ que es definida como $F_n = F_{n-1} + F_{n-2}$ donde $F_1 = 1$ y $F_2 = 1$ los primeros terminos de la serie de Fibonacci son $1, 1, 2, 3, 5, 8, 13, 21$, en la definición de la serie se puede notar que esta serie depende siempre de los dos terminos pasados, esta caracteristica es importante en la series computacionales ya que esta relacionado con el concepto de recursividad.

Un proceso recursivo es aquel en el cual un objeto es definido en términos de otros objetos del mismo tipo, esto permite mediante algún tipo relación recursiva la clase entera de objetos puede ser construida a partir de unos cuantos valores iniciales y un pequeÃśo número de reglas. Esta definición nos permite ver que la serie de Fibonacci puede ser construida de forma recursiva, ademas al ser construida con pocas reglas esto permite escribir codigo que es mas conciso y entendible.

A continuación se presentara el codigo para calcular números de Fibonacci utlizando python y recursividad:

```
1 def fibnc(n): ## enesimo termino
2     global ope_fib
3     ope_fib += 1
4     if n == 0  or n == 1:
5         return 1
6     else:
7         return fibnc(n-2) + fibnc(n-1)
```

Codigo 1.Fibonacci calculado con recursividad

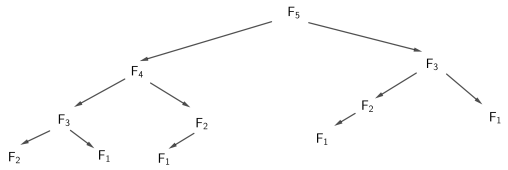

Figure 1: *Llamadas de función al realizar fibonacci(5)*

Ahora describiremos un poco la complejidad computacional de este algoritmo, esto se har'a utilizando la variable global para contar operaciones, si calculamos la complejidad computacional del algoritmo al calcular el quinto número de Fibonacci veremos que le toma 15 operaciones funcionar, si se quiere calcular el 10 número de Fibonacci nos tomara 177 operaciones. El problema con nuestro programa recursivo es que para llamadas recursivas repite proceso, esto es, al calcular F_5 calcula todas las llamadas recursivas y al volver a calcular F_5 en lugar de tomar un valor en teoría ya calculado, repite el proceso de recursión, esto lo podemos ver en la imagen3 .

Para evitare esto ahora se presenta el código para calcular el n-esimo número de Fibonacci pero no realizando llamadas recursivas de la función sino más bien con solo ciclos simples, esto nos dará un proceso con mejor rapidez pero sacrificaremos la simplicidad del código que utiliza recursividad.

```
def fibnc_simple(n):
    global ope_fib_sim
    if n == 0 or n ==1:
        return 1
    r, r1,r2=0,1,1
    for i in range(2,n+1):
        ope_fib_sim+=3
        r = r1+r2
        r2=r1
        r1=r
    return r
```

Codigo 2.Fibonacci calculado sin recursividad

Con este codigo, calcular el número de Fibonacci de 5 nos tomaria 12 operaciones y para calcular el número de operaciones para calcular el número de Fibonacci 10 nos tomaria 27, lo cual es mas rapido que lo que nos tomo al realizar esta tarea con el codigo recursivo, sin embargo este codigo se puede hacer todavia mas eficiente, a continuación se presenta otra alternativa para calcular el n-esimo número de Fibonacci.

En el codigo de Fibonacci con recursividad hablamos acerca de su complejidad, dijimos que no era muy buena ya que tenia que calcular el número de Fibonacci de algunos números multiples veces, en lugar de utilizar un valor que ya se habia calculado el programa tenia que obtener el número realizando procesos repetidios y esto repercutia de gran manera en la complejidad, para arreglar esto algo que podemos hacer es almacenar en memoria los distintos valores de F_n que sean calculados, de esta manera una vez que son calculados ya no es necesario calcularlos de nueva cuenta sino llamarlos de donde estan almacenados, a continuación se presenta el codigo

para realizar esto.

```
d={0:1, 1:1 }
def fibnc_eficiente(n,d): ## enesimo termino
    global ope_fib_efi
    ope_fib_efi += 1
    if n in d :
        return d[n]
    else:
        ans = fibnc_eficiente(n-2,d) + fibnc_eficiente(n-1,d)
        d[n] = ans
        return ans
```

Codigo 3.Fibonacci calculado con recursividad y diccionarios

Realizando un codigo especial para guardar el número de operaciones realizan estos tres algoritmos nos podemos generar la grafica 2, el codigo para generar esta grafica se presenta a continuación.

```
numero = []
op_simple=[]
op_eficiente=[]
op_recursivo=[]
for i in range(0,30):

    ope_fib = 0
    ope_fib_efi=0
    ope_fib_sim=0
    fibnc_simple(i)
    fibnc_eficiente(i,d)
    fibnc(i)
    numero.append(i)
    op_simple.append(ope_fib_sim)
    op_eficiente.append(ope_fib_efi)
    op_recursivo.append(ope_fib)

plt.xlabel('N-esimo n\'umero de Fibonacci')
plt.ylabel("# de Operaciones")
plt.plot(numero,op_simple)
plt.plot(numero,op_eficiente)
plt.plot(numero,op_recursivo)
```

Codigo 4.- Para obtner grafica de complejidad

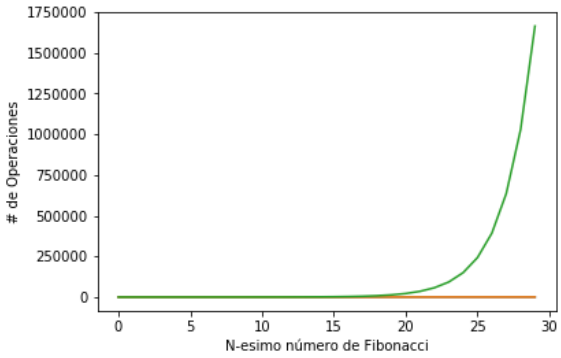

Figure 2: *Grafica de complejidad computacional en los tres algoritmos*

II. Números primos

En esta sección se presentara un algoritmo para determinar si un número dado es primo o no, a este tipo de algoritmos les llamamos test de primalidad, el test que presentaremos es un muy sencillo, tests más complejos pueden ser desarrolados. Este tipo de problemas son de gran importancia en las matemáticas ya que los primos son un area de la teoria de números que cuenta con gran cantidad de problemas abiertos y que se entrelazan con otras areas de matemáticas.

Decimos que un número es primo si los unicos divisores eneteros del número son el 1 y el mismo número, esto nos da a entender que ningun número entre el 1 y el mismo número puede ser factor del número original, entonces algo que podemos hacer es iterar desde el 1 hasta el número buscando si un número es factor, si esto sucede entonces salimos del ciclo y concluimos que el número no es primo, si realiza todo el ciclo sin encontrar un factor entonces podemos concluir que el número es primo.

Se puede demostrar matematicamente que no es necesario probar cada uno de los factores del 1 al número para demostar que es primo, basta con iterar hasta el techo de la raiz del número y si no encontramos factores entonces podemos concluir que es primo, en el siguiente codigo queda plasmada la idea de este test de primalidad:

```
import math
import matplotlib.pyplot as plt


def primo (num):
    contador = 0
    test = "Primo"
    for i in range(2, math.ceil( math.sqrt(num) ) +1):
        contador += 1
        if (num % i) == 0:
            test = "No primo"
            break
    return (contador,test)

numero_operaciones=[]
numero_checado=[]
```

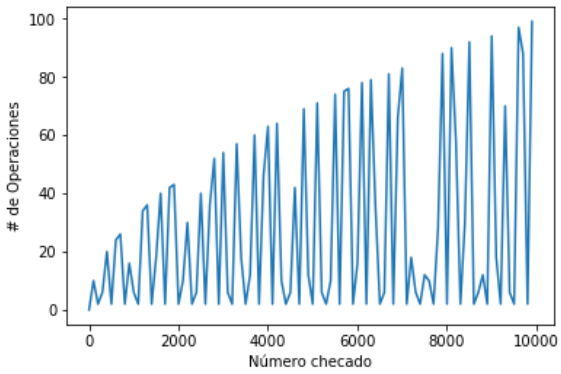

Figure 3: *Grafica de complejidad, test de primalidad*

```
numero_test=[]
for numero in range(1,10001,100):
    t = primo(numero)
    numero_checado.append(numero)
    numero_operaciones.append(t[0])
    numero_test.append(t[1])
for i in range(0, len(numero_checado)):
    print(numero_checado[i],numero_operaciones[i], numero_test[i])

plt.plot(numero_checado,numero_operaciones)
```

Codigo 5.- Codigo para calcular si un número es primo o no

A continuación se presenta la grafica para de la complejidad computacional en el calculo de los números primos del 1 al 1001 con incrementos de 10:

III. Conclusión

Mi conclusión es que utilizando de manera adecuada las herramientas con las que contamos ya sean estructuras dentro del programa o propiedades matematicas del objeto de estudio podemos simplificar drasticamente la complejidad de un problema. Para el algoritmo de Fibonacci pudimos transformar un problema que era exponencial en uno lineal tan solo haciendo uso de memoria.

En el test de primalidad hicimos uso de propiedades desarrolladas dentro de la teoría de números para no tener que probar todos los casos de factores, como futura mejora queda optimizar todavia mas el codigo mediante propiedades de la teoría de números para cada uno de los problemas, el de la serie de Fibonacci y el de test de primalidad.