Universidad Autónoma de Nuevo León Facultad de Ciencias Físico Matemáticas

Artemio Guajardo Aparicio Grupo: 001 Matrícula: 1590417 Fecha: 13/10/2017

Algoritmos para números primos y para la sucesión de Fibonacci

Introducción

Desde siempre hemos sabido que hay diferentes maneras de implementar un algoritmo en diferentes lenguajes. En nuestra primer etapa cuando trabajamos con algoritmos de ordenamiento observamos que varios compañeros teníamos los algoritmos acomodados de forma diferente a la hora de implementarlos y muchas veces afectaba en el rendimiento de estos. En esta ocasión ese será el caso pues veremos como con tres formas de implementar un mismo algoritmo varia de manera considerable la eficiencia del programa, es decir, para unos la cantidad de operaciones realizadas es mucho mayor que para otros. También observaremos un método para implementar un algoritmo para saber si un número es primo o no. Todo lo anterior mencionado será realizado a base de teoremas.

Algoritmo para saber si un número es primo

Sabemos que en matemáticas, un número primo es un número natural mayor que 1 que tiene únicamente dos divisores distintos: él mismo y el 1. Por el contrario, los números compuestos son los números naturales que tienen algún divisor natural aparte de sí mismos y del 1 y por lo tanto, pueden factorizarse. El número 1, por convenio, no se considera ni primo ni compuesto.

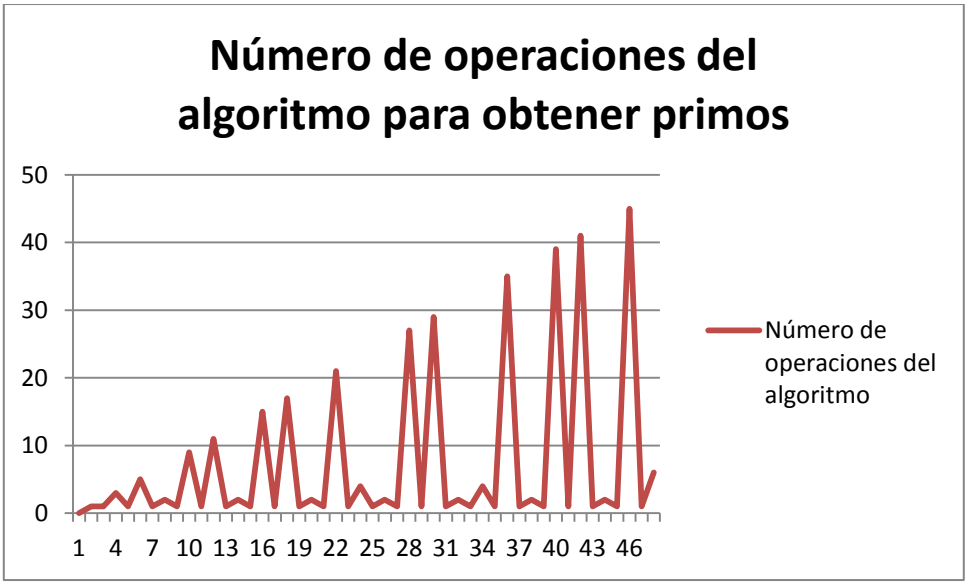
Lo anterior lo podemos implementar de manera muy sencilla en Python pues solo tenemos las condiciones de que para que un número sea primo tiene que ser mayor que 1, dividirse entre el mismo y dividirse entre 1.

Mencionado esto, su implementación es la siguiente:

```
cnt=0
def primo(n):
    global cnt
    for i in range(2, n):
        cnt+=1
        if ((n%i)==0):
            return ("No es primo")
    return ("Es primo")
```

Conocemos una propiedad que nos dice que si un número no tiene divisores menores o iguales que su raíz cuadrada es primo, lo cual se puede utilizar para optimizar el código. Sin embargo por problemas técnicos me fue imposible utilizar ese método. De igual forma el método utilizado es muy eficiente.

En la implementación la variable cnt (contador) la utilizamos para obtener el número de operaciones realizadas con cada número que se introduzca, en este caso se obtuvieron resultados para los valores del 1 al 50 con el fin de obtener la siguiente gráfica:



En la cual podemos observar que el algoritmo utiliza una gran cantidad de operaciones cuando un número es primo, en el caso contrario no pasa de 6 operaciones.

Algoritmo para obtener términos de la sucesión de Fibonacci

Ahora trabajaremos con lo mencionado en la introducción, es decir, haremos tres tipos de implementaciones para un mismo algoritmo.

En matemática, la sucesión de Fibonacci es la siguiente sucesión infinita de números naturales:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597 \ldots$$

La sucesión comienza con los números 0 y 1, y a partir de estos, «cada término es la suma de los dos anteriores», es la relación de recurrencia que la define.

Lo anterior es la clave para implementar este algoritmo a Python: "Cada término es la suma de los dos anteriores".

Algoritmo Fibonacci Recursivo

La recursividad es la forma en la cual se especifica un proceso basado en su propia definición, siendo ésta característica discernible en la construcción a partir de un mismo tipo. Dicho esto y descrita anteriormente la sucesión podemos implementar el algoritmo creando una función donde tengamos que si n es igual a 0 o n es igual a 1 (n es nuestro parámetro, el n-ésimo valor), el termino debe ser igual a 1, lo cual va a realizar la función mediante una condición y si n es diferente de 0 y 1 entonces podemos utilizar la condición que dice que cada termino es la suma de los dos anteriores.

Implementando esto obtenemos:

```
cnt=0
def fibonacciR(n):
        global cnt
        cnt+=1
        if n==0 or n==1:
                return (1)
        else:
                return fibonacciR(n-2)+fibonacciR(n-1)
```

Podemos observar que aquí también utilizamos la variable cnt (contador) y en este caso se utilizó también para obtener las operaciones del algoritmo para los primeros 40 términos. Se obtuvo la gráfica siguiente:

En la cual podemos observar que la cantidad de operaciones utilizadas en este algoritmo es demasiado elevada, de hecho se tuvo que realizar para los primeros 40 términos y no de los primeros 50 debido a la cantidad considerable de tiempo que tardaba en llegar a estas cifras. El número máximo de operaciones para esta cantidad de términos es 204,668,309.

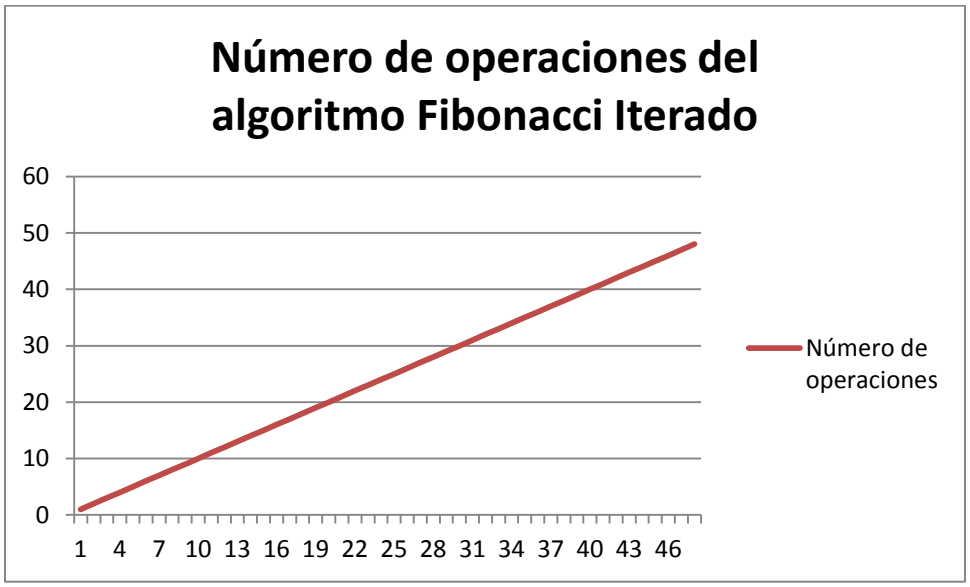
Algoritmo Fibonacci Iterado

Para este algoritmo utilizaremos una implementación muy parecida a la anterior, con la diferencia de que aquí se hará uso de tres variables con los valores 0, 1 y 1 respectivamente para poder utilizarlas dentro de un ciclo for con la condición "Cada termino es la suma de los dos anteriores". Para visualizarlo de una mejor manera dejo su implementación:

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

```
        cnt+=1
        r=r1+r2
        r2=r1
        r1=r
    return r, cnt
```

Nuevamente usamos la función contador para obtener la siguiente gráfica:



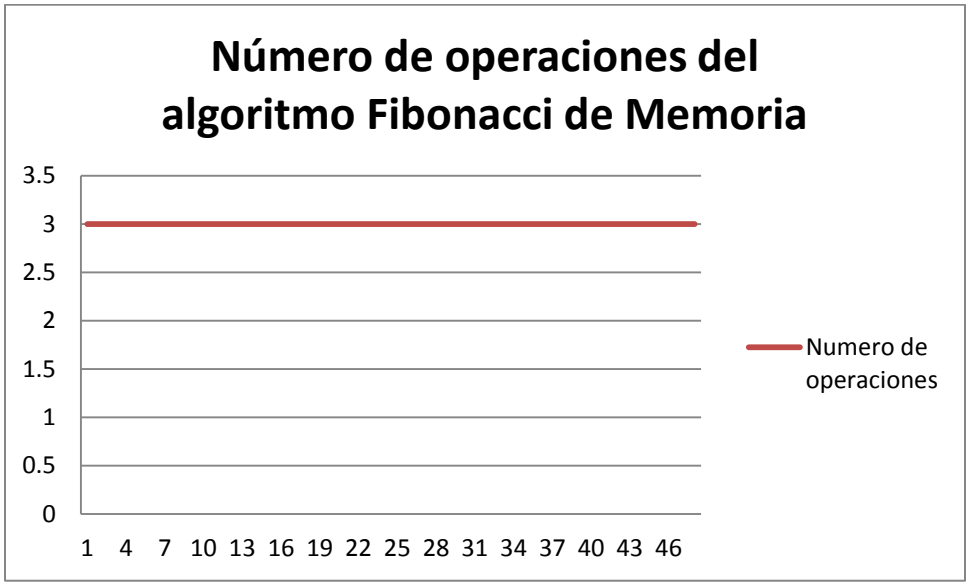
En la cual podemos observar que si se obtuvieron los primeros 50 términos, debido a que este algoritmo utiliza muchas menos operaciones que el anterior (El número máximo de operaciones para esta cantidad de términos es de 48).

Algoritmo Fibonacci de Memoria

Esta forma de obtener términos de la sucesión difiere en las otras dos en que aquí se utiliza un espacio de memoria para datos. Utilizando una condición vamos a verificar si el término que se desea obtener ya está en el espacio de memoria antes mencionado, si no está entonces haremos uso de la condición “Cada termino es la suma de los dos anteriores”, pero si ya lo tenemos lo guardaremos en ese espacio de memoria, lo cual hará que la función utilice menos operaciones que las dos anteriores. Todas las demás condiciones son iguales a las comunes de los algoritmos anteriores. Con el fin de visualizarlo mejor se deja aquí su implementación a Python:

```
memo={}
cnt=0
def fibonacci(n):
        global memo, cnt
        cnt+=1
        if n==0 or n==1:
        return(1)
        if n in memo:
        return memo[n]
        else:
        val=fibonacci(n-2)+fibonacci(n-1)
        memo[n]=val
        return val
```

Como podemos observar, también se hace uso de la variable contador, por lo cual obtuvimos también una gráfica que es la siguiente:



Podemos observar que este algoritmo es mucho más eficiente que los anteriores pues siempre realiza solo 3 operaciones debido a lo mencionado al inicio de la descripción de este algoritmo. Aquí se permitió obtener el número de operaciones para los primeros 50 términos fácilmente.

Conclusiones:

De todo lo anterior se puede concluir que lo afirmado en la introducción es correcto. Hay varias formas de implementar un mismo algoritmo, unas más eficientes que otras pues pudimos observar que el algoritmo Fibonacci de memoria utilizo menos operaciones que el Fibonacci iterado, mientras que estos dos utilizaron muchas menos operaciones que el algoritmo Fibonacci Recursivo. Mientras que a veces varía completamente la cantidad de operaciones dentro de un mismo algoritmo como lo pudimos ver en el algoritmo para saber si un número es primo.