Reporte de test de primalidad y sucesión de fibonacci

José Manuel Tapia Avitia.
Matrícula: 1729372
jose.tapiaav@gmail.com
https://github.com/jose-tapia/1729372MC

12 de octubre de 2017

El presente reporte tiene la finalidad de analizar y mostrar algoritmos relacionados a los números primos y la sucesión de fibonacci, con el objetivo de comprender e implementarlos en Python.

1. Test de primalidad

Durante siglos se han realizado investigaciones referente a los números primos, con la intención de descubrir nuevas propiedades para su uso en criptografía, por ejemplo.

Definimos la primalidad de un número como la propiedad de que un número sea primo o no.

Para poder utilizar números primos, primero debemos de encontrarlos. Para ellos, se han desarrollado diversos algoritmos para probar si un número es primo o no, llamados test's de primalidad.

Por definición, se dice que un número es primo si los únicos números que lo dividen son el 1 y el mismo. Dado esto, un posible algoritmo es comprobar si los números entre 2 y uno antes del número no le dividen. Es decir,

$$\text{n es primo } \Leftrightarrow \nexists m \in [2, n-1] \text{ tal que } m \mid n$$

En donde, se puede demostrar que si existíera dicho m, entonces $m \leq \sqrt{n}$. Por lo tanto, un número n es primo si para $m = 2,\ 3,\ 4,\ \ldots\ , \sqrt{n}$, m no divide a n.

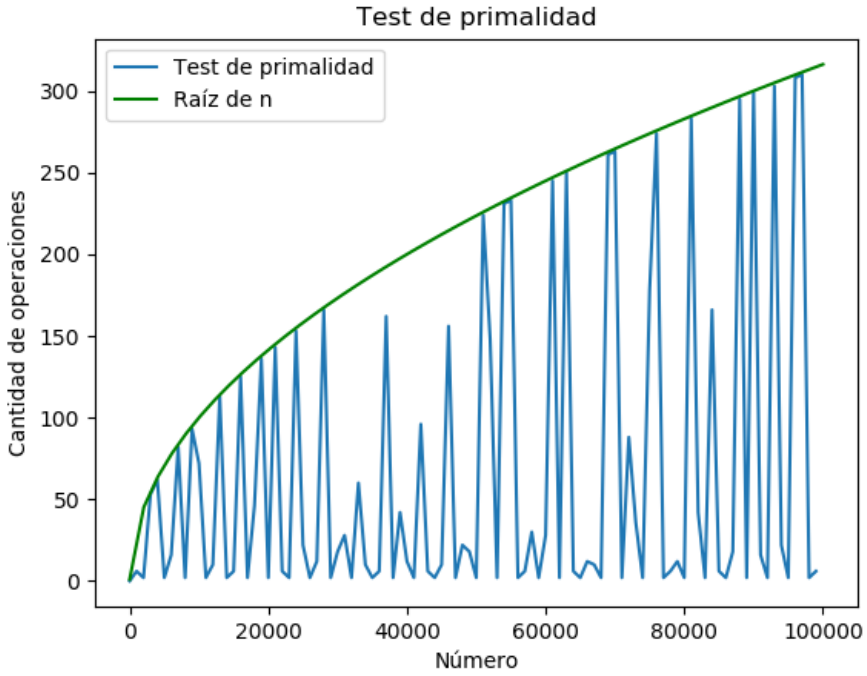
Esto nos lleva al siguiente algoritmo para comprobar si n es primo o no:

- Si n es menor o igual a 1, entonces no es primo.
- Inicializar m=2.
- Mientras m sea menor o igual a $\sqrt{n}$, checar si m divide a n.
- Si sí, n no es primo, en caso contrario, aumentar m.
- Si m es mayor a $\sqrt{n}$, entonces n es un número primo.

En Python, el código sería el siguiente:

```
def esPrimo(n):
    if n<=1:
        return False
    m=2
    while(m*m<=n):
        if n%m==0:
            return False
        m+=1
    return True
```

En donde, a lo más realizaría $\sqrt{n}$ operaciones en el peor de los casos, lo cual sucede generalmente si n es primo. Para n número compuesto, realiza menos operaciones. En la siguiente gráfica, se analizó la cantidad de operaciones que realiza el test de primalidad para los primeros 100,000 números naturales.

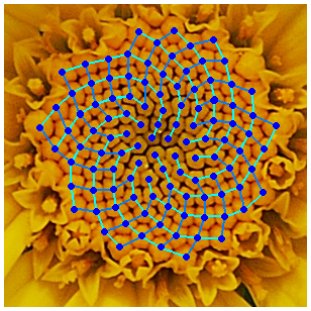


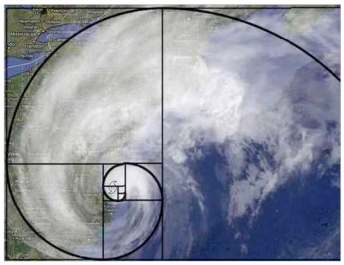
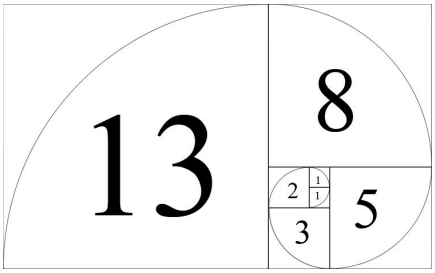
Se logra apreciar la cota superior de $\sqrt{n}$. Ya que la cantidad de operaciones llega a ser hasta $\sqrt{n}$, la complejidad del algoritmo es $\mathcal{O}(\sqrt{n})$.

2. Sucesión de fibonacci

Dentro del área de las matemáticas, existen ciertas sucesiones con propiedades sumamente interesantes, una de ellas, la sucesión de fibonacci, formulada por Leonardo de Pisa.

Dicha sucesión se encuentra de manera natural en la fauna y flora de diversos ambientes.



La sucesión de fibonacci se define de multiples formas, la que manejaremos será la siguiente:

$$F_0 = 1,\ F_1 = 1 \text{ y } F_{n+1} = F_{n-1} + F_n \text{ para } n \geq 1.$$

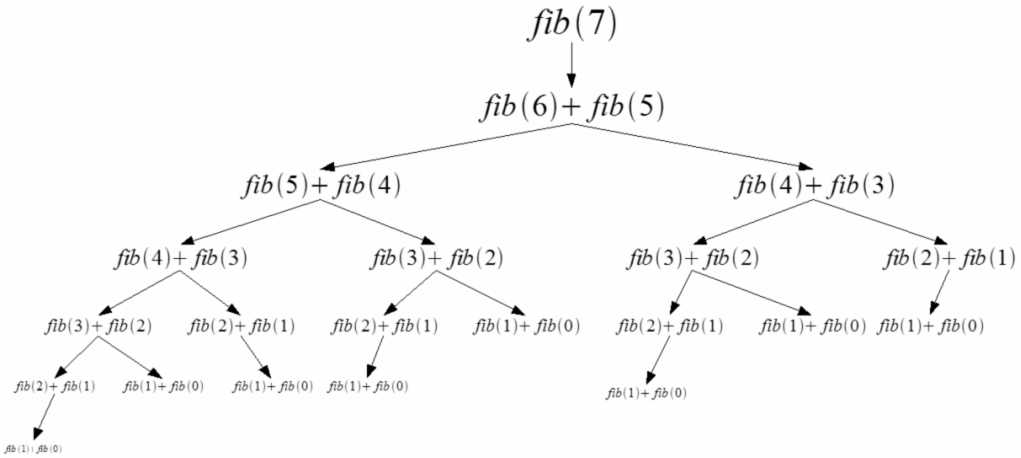
En donde F_n es el n-ésimo número de fibonacci. Los primeros términos de la sucesión son 1, 1, 2, 3, 5, 8, 13, 21, 34, ... , 55, 89, 144, ... Podemos notar que la sucesión de fibonacci define los primeros dos valores y los siguientes se consiguen de los dos anteriores.

Nuestro objetivo es realizar métodos que calculen números de fibonacci.

Para ello podemos definir una función que dado una n nos calcule el n-ésimo número de fibonacci de manera recursiva. La función sería como sigue:

```
def fiborec(n):
    if n==0 or n==1:
        return 1
    return fiborec(n-1)+fiborec(n-2)
```

El problema es que la función recursiva se tarda demasiado ya que repite muchas operaciones, para visualizarlo se presenta la siguiente imagen:



Teniendo en cuenta esto, podemos optimizarlo con un arreglo de tal forma que calculemos cada término solamente una vez. La implementación sería:

```
def fiboiter(n):
    fib=[1,1]
    for k in range(2,n+1):
        fib.append(fib[k-1]+fib[k-2])
    return fib[n]
```

La función genera un arreglo de longitud $n+1$, en donde se inicializa con los primeros dos términos bases, basados en estos, generar los siguientes. Al no ser recursiva, se le denomina iterativa.

En comparación de la función pasada, la cantidad de operaciones es abrumadoramente menor, aunque tiene sus desventajas ante otros algoritmos.

El siguiente algoritmo, al igual que el primero, es recursivo, sin embargo, no vuelve a calcular los valores que ya se han calculado, acotando la cantidad de operaciones. La implementación es como sigue:

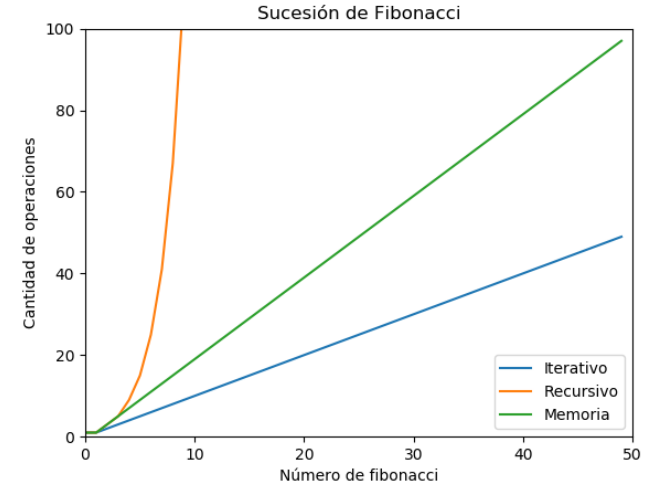
```
#Diccionario como variable global
memo={}

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

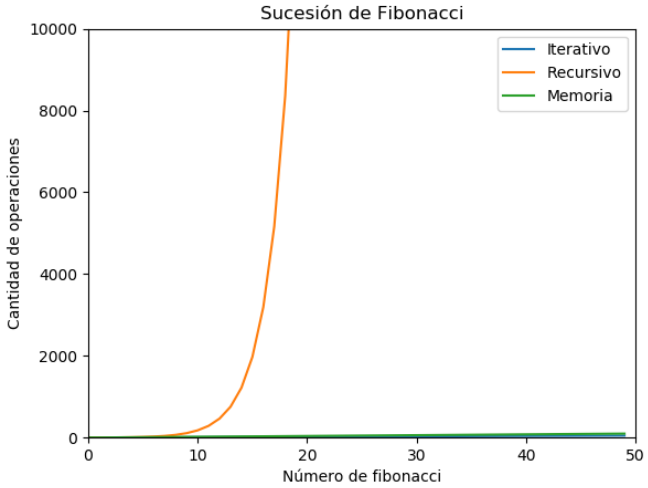
El primer algoritmo podemos apreciar que logra obtener el n-ésimo término de la sucesión sumando puros 1's. Por lo tanto, la complejidad para obtener el n-ésimo término se deben realizar aproximadamente F_n operaciones, por lo que la complejidad es $\mathcal{O}(F_n)$.

En el segundo algoritmo, se calcula un arreglo con los primeros n términos realizando aproximadamente n sumas, se tiene que la complejidad para calcular el n-ésimo término es $\mathcal{O}(n)$.

Para el tercer algoritmo, al tener en un diccionario global y calcular cada término una vez, para el n-ésimo término se realizaron aproximadamente n operaciones, por lo que la complejidad es $\mathcal{O}(n)$. La ventaja que se tiene ante los otros algoritmos, es que al volver a requerir el término n-ésimo, no se vuelven a realizar n sumas, al ya tener almacenada el número, lo regresa sin realizar ninguna operación.

Viendo gráficamente las complejidades de los tres algoritmos en práctica para los primeros 50 términos, se tiene:





En la primera gráfica se puede ver la notable diferencia de complejidades de los algoritmos, con una ligera ventaja para el algoritmo iterativo. En la segunda se puede apreciar como el algoritmo recursivo crece de manera exponencial, con una diferencia abrumadora ante los otros algoritmos.

Sin dejarnos llevar por la ventaja que muestra el algoritmo iterativo, el algoritmo con memoria tiene la ventaja de tener almacenadas los números de fibonacci, evitando el volver a calcular los números de fibonacci.

3. Detalles de implementación

Se mostró como, partiendo de la definición de número primo, se puede dar un método para conseguir realizar dicha tarea, después de diversas observaciones se puede modificar dicho método para conseguir optimizarlo eficientemente.

Se mostraron diversas formas de implementar una tarea en particular, en este caso, el calcular el n-ésimo término de fibonacci. Se mostraron diversas ideas para calcularlo, ya sea con la manera más intuitiva de todas, con recursividad, que se logra implementando de manera casi directa de la definición de la sucesión de fibonacci. La siguiente, de almacenar los números en un arreglo para no volver a repetir el calculo. Y por último, una optimización del primer algoritmo, agregando la estructura de diccionario que tiene Python para conseguir un algoritmo eficiente.