UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE CIENCIAS FÍSICO – MATEMÁTICAS

MATEMÁTICAS COMPUTACIONALES

“REPORTE DE ALGORITMOS DE FIBONACCI Y NÚMEROS PRIMOS”

Nombre: Diego Rubio Romero

Matrícula: 1738544

Carrera: Lic. En Matemáticas

Profesor: Lic. José Anastacio Hernández Saldaña

Fecha: Viernes 13 de Octubre del 2017

Lugar: Ciudad Universitaria

Introducción

En el siguiente reporte, se hablará acerca de dos algoritmos: La sucesión de Fibonacci y Números Primos, repasando estos mediante la descripción de cada uno y a su vez la descripción de sus algoritmos, y por úlitmo una comparativa de rendimiento entre los algoritmos a ver.

1. Números Primos

Un número primo es un número natural, mayor o igual a dos, que se puede identificar si es divisible únicamente por dos números, es decir, un número que puede ser dividido única y exclusivamente por el uno, y por sí mismo.

Como característica todos los numéros primos son impares, a excepción del dos (Es el único número primo par).

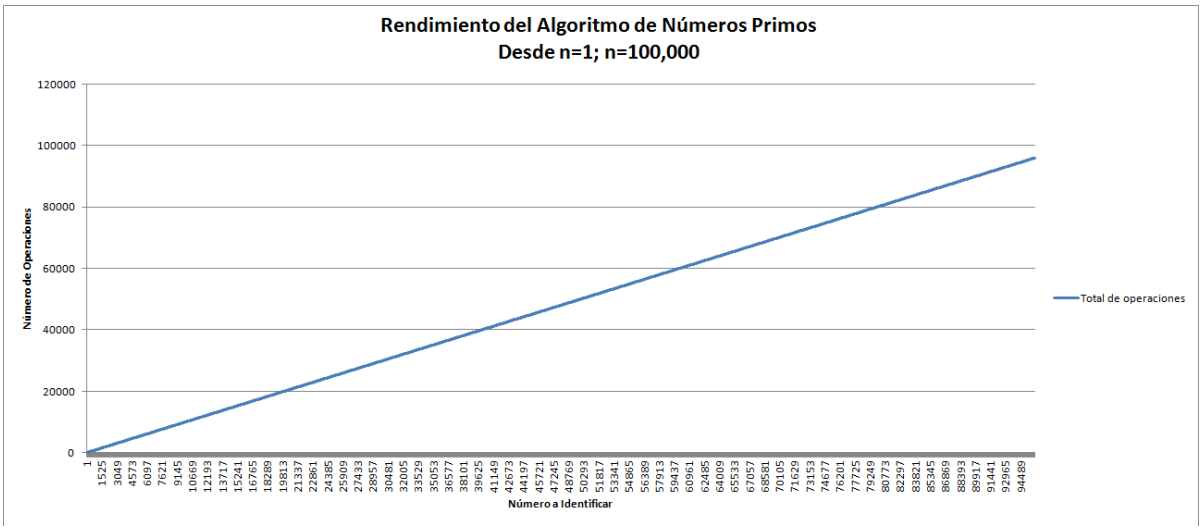
El siguiente algoritmo en Python, tiene como objetivo identificar si un número es primo o no, usando la definición de número primo, osea que tenga exactamente dos divisores y que este sea el mismo y el uno.

Código Fuente:

```
def NumeroPrimo (NumeroAIdentificar):
    NumeroDeDivisores=0
    if NumeroAIdentificar <2:
        print("Intente con un numero mayor o igual a 2")
    for dividir in range(1, NumeroAIdentificar +1):
        if NumeroAIdentificar  % divisor==0:
            NumeroDeDivisores  = NumeroDeDivisores  + 1
    if NumeroDeDivisores!=2:
        print(“No es un número primo”)
    else:
        print(“Si es un número primo”)
```

La complejidad en el peor de los casos de este algoritmo es de $O(n)$

La siguiente gráfica describe el desempeño del algoritmo:



Conclusión

Pudimos observar que un número era primo si este tenía únicamente dos divisores: El mismo y el uno.

Además mediante prácticas al probar el código, determinamos que su complejidad era lineal, de tal manera que se requería exactamente la cantidad de operaciones análoga al n-ésimo término ingresado. Obteniendo así una gráfica de desempeño que demostró lo dicho anteriormente.

2. Sucesión de Fibonacci.

Es una sucesión infinita de números naturales que empieza en 1 y 1, y a partir de ahí los siguientes términos son el resultado de estar sumando los dos números anteriores del que se desee obtnere, tomando la siguiente forma: F(n)= 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,…

A los elementos de esta sucesión se les llama números de Fibonacci.

Como característica esta sucesión no tendría nada de particular sino fuera porque aparace repetidamente en la naturaleza y, además, tiene numerosas aplicaciones en ciencias de la computación, matemáticas y teoría de juegos, entre otras.

A continuación se mencionarán tres algoritmos distintos en Python que mediante un método distinto, tienen como objetivo, obtener la sucesión de Fibonacci:

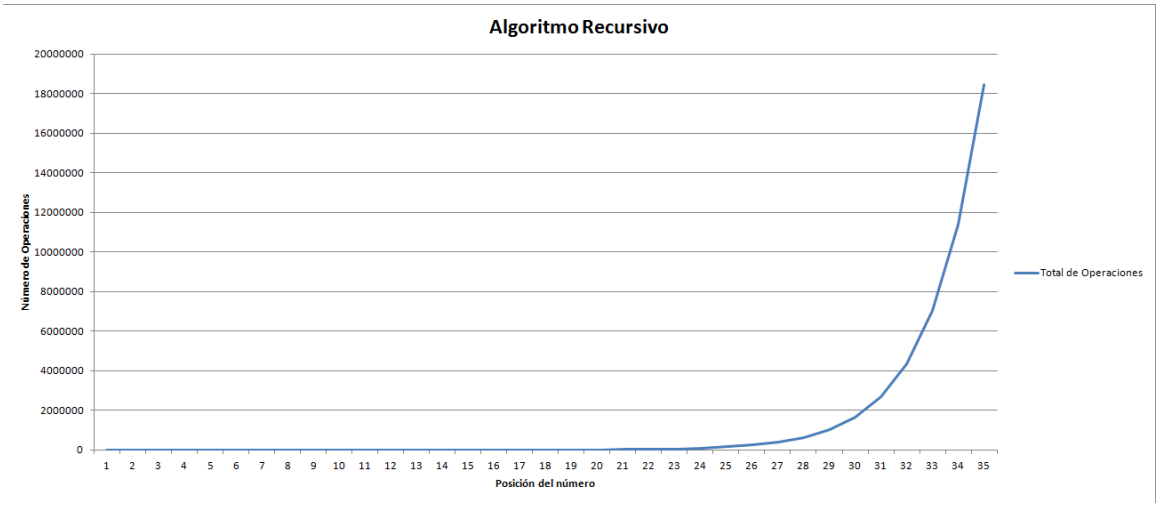
2.1 Algoritmo Recursivo

Hace uso de la recursividad en la función, repitiendo el ciclo hasta obtener la el numero de Fibonacci ‘n’ deseado.

Código Fuente:

```
def fibonacci(PosicionDelNumero):
        Numero = PosicionDelNumero
        if Numero ==0 or Numero ==1:
                return 1
        return fibonacci(Numero-2) + fibonacci(Numero -1)
```

La siguiente gráfica describe el desempeño del algoritmo:



2.2 Algoritmo Iterativo

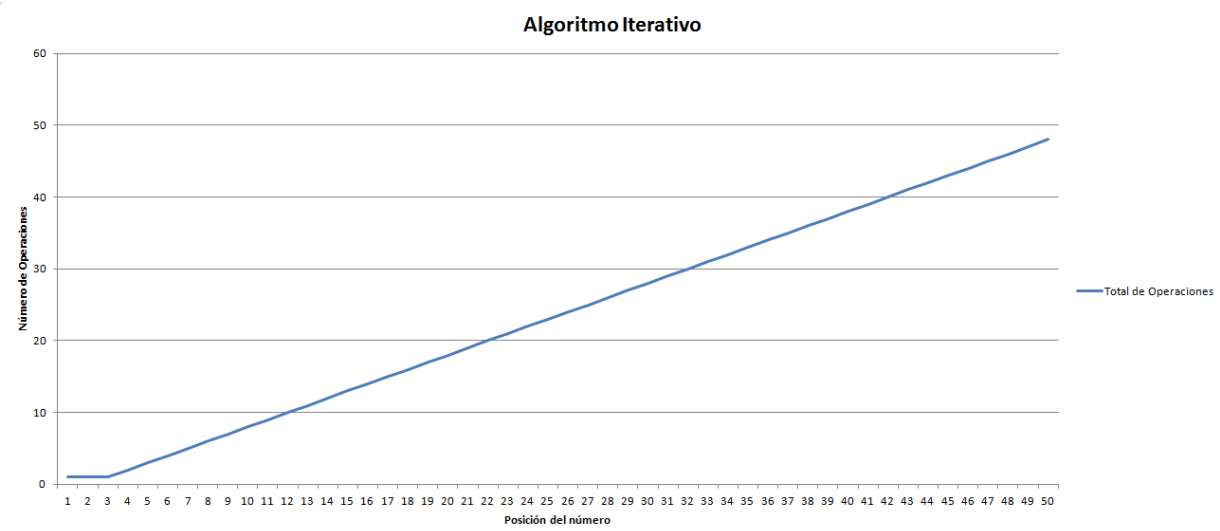
Este algoritmo hace uso de estar intercambiando datos entre las variables, asignando un valor anterior y pre-anterior a dos variables temporales, que tienen como fin, obtener un numero de fibonacci.

Código Fuente:

```
def fibonacci(PosicionDelNumero):
        Num = PosicionDelNumero
        NumeroDeFibonacci=0
        NumTemp1=1
        NumTemp2=1
        if num==0 or num==1:
                return 1
        for i in range(2,num+1):
                NumeroDeFibonacci =NumTemp1+NumTemp2
                NumTemp2=NumTemp1
                NumTemp1= NumeroDeFibonacci
        return NumeroDeFibonacci
```

La complejidad en el peor de los casos de este algoritmo es de $O(n-2)$; n>3

La siguiente gráfica describe el desempeño del algoritmo:



2.3 Algoritmo Recursivo Con Memoria

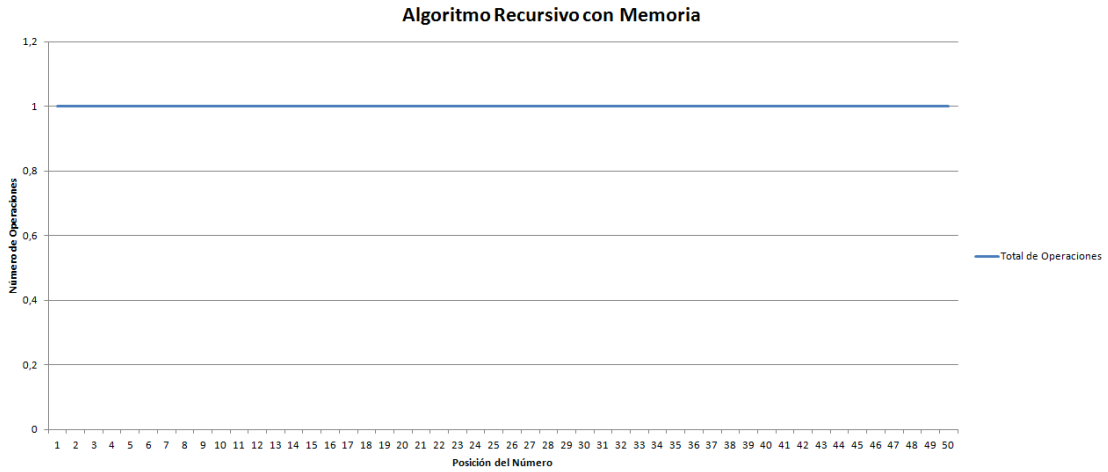
Este algoritmo declara un arreglo(Memoria) en donde todas los números de fibonacci con su respectiva posición, de preferencia en orden ascendente, guarda la posición y su valor dentro del arreglo(Memoria) por si se llega a dar el caso de que sea requerido su uso en el momento que se este ejecutando la consola, todo esto haciendo uso de la recursividad ya vista en 2.1.

Código Fuente:

```
Memoria={ }
def fibonacci(PosicionDelNumero):
        n= PosicionDelNumero
        global Memoria
        if n==0 or n==1:
                return 1
        if n in Memoria:
                return Memoria[n]
        else:
                NumeroDeFibonacci=fibonacci(n-2) + fibonacci(n-1)
                Memoria[n]= NumeroDeFibonacci
                return NumeroDeFibonacci
```

La complejidad en el peor de los casos de este algoritmo es de $O(1)$, siempre y cuando se corra de manera ascendente, es decir de 1 en adelante.

La siguiente gráfica describe el desempeño del algoritmo:



Conclusión

En conclusión pudimos crear, y observar mediante prácticas los diferentes algoritmos vistos previamente, todos cumpliendo un mismo objetivo, obtener el n-ésimo numero de fibonacci, mediante diferentes métodos, que fueron el recursivo, iterativo, o el uso de un arreglo, para guardar posiciones anteriores, o un número en específico.

Además mediante los contadores, pudimos comprobar el rendimiento y eficacia de cada uno, mediante prácticas corriendo todos los algoritmos hasta una posición en específico para todos, y así observar cual era mejor de todos estos, además de obtener la complejidad de estos.

Los resultados finales arrojaron que el Algoritmo Iterativo es más eficiente que el recursivo, pero comparando este primero con el Recursivo con Memoria, se pudo notar una gran diferencia entre ambos, dando una mayor eficacia al Recursivo con Memoria que si se corría de manera ascendente, así se obtenían los valores previos a un sucesor, si es que se requería, además de que estos eran almacenados en un arreglo, para cada resultado, haciendo una única operación (suma) para obtener el siguiente término.