Reporte de practica: Estructuras de datos

Nombre: Diana Carolina Lozoya Pérez **Matricula:**1725012

Materia: Matemáticas Computacionales **Grupo:**001 **Fecha:** 13/10/17

Resumen: Este reporte tiene como objetivo mostrar y describir un código creado para calcular un elemento deseado, en base a la sucesión de Fibonacci. Se mostrarán 3 maneras diferentes para poder obtener lo esperado, además una gráfica comparando los códigos anteriormente programados, así como un código para obtener números primos.

I. Fibonacci (1)

Con Fibonacci (1) se busca obtener los elementos correspondientes de la sucesión de Fibonacci. Para poder generar nuestro código primero es necesario definir una función para darle instrucciones acerca de cómo obtener el elemento deseado. Primeramente, declararemos una variable global, que será la encargada de contar el número de iteraciones realizadas por el programa para poder obtener dicho elemento que nosotros le solicitemos, después la inicializamos en cero para que lleve la cuenta bien. Posteriormente comparamos si nuestro número de elemento es igual a cero o a 1, ya que en la sucesión de Fibonacci se inicia a partir del segundo término, diferente de 1. Luego declaramos otras variables para poder iterar dentro de un ciclo y así obtener el elemento deseado, aquí definimos que el elemento deseado es el número que resulta de la suma de los dos anteriores.

Código en Python:

```
#Fibonacci 1
def fibo(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


#intervalo de 1 al 35
for i in range (2, 35):
        cnt=0
        print(i, fibo(i), cnt )
```

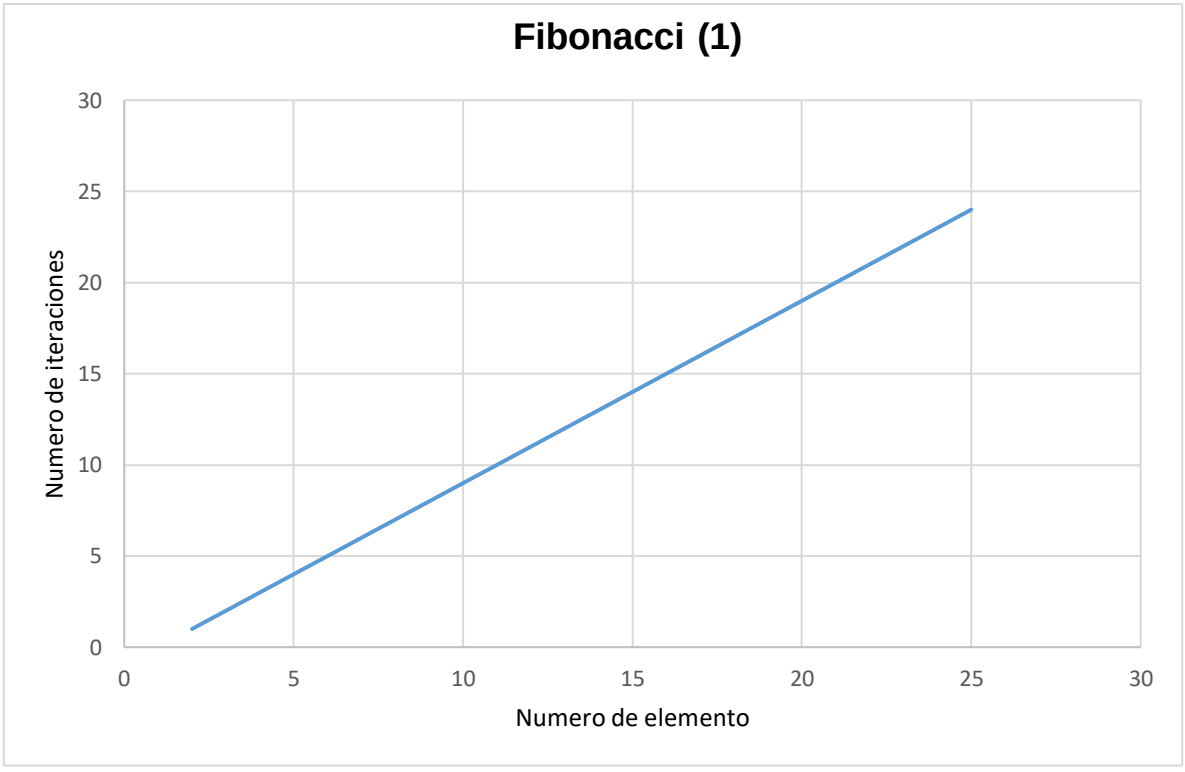
Después se probó el código y con un ciclo se generó una tabla que reflejaba el elemento generado desde el 1 hasta el 35 en la sucesión de Fibonacci, posteriormente, realizaremos una gráfica para ver su comportamiento, comparando el número de operaciones con el número de elementos pedidos. A continuación, se muestra la tabla y la gráfica, ambas hechas en Excel:

Tabla:

Elemento	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Iteraciones	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

18	19	20	21	22	23	24	25
17	18	19	20	21	22	23	24

Gráfica:



Donde podemos observar la comparación de número de elementos junto con el número de iteraciones llevadas para cada elemento que deseábamos saber. Esta grafica concuerda con lo esperado (una línea recta) debido a la manera en la que se establecieron las instrucciones para las operaciones.

II. Fibonacci (2)

Con Fibonacci (2) se busca obtener los elementos correspondientes de la sucesión de Fibonacci. Para poder generar nuestro código primero es necesario definir una función para darle instrucciones acerca de cómo obtener el elemento deseado. Primeramente, declararemos una variable global, que será la encargada de contar el número de iteraciones realizadas por el programa para poder obtener dicho elemento que nosotros le solicitemos, después la inicializamos en cero para que lleve la cuenta bien. Posteriormente comparamos si nuestro número de elemento es igual a cero o a 1, ya que en la sucesión de Fibonacci se inicia a partir del segundo término, diferente de 1. Si no, devolvemos el valor de dos posiciones menos más el valor de una posición menos con el fin de obtener el elemento deseado, aquí definimos que el elemento deseado es el número que resulta de la suma de los dos anteriores.

Código en Python:

```
#Fibonacci (2)
cnt=0

def fibonacci(n):
    global cnt
    cnt+=1
    if n==0 or n==1:
        return(1)

    else:
        return fibonacci(n-2)+ fibonacci(n-1)


#intervalo de 2 al 35

for i in range (2, 35):
        cnt=0
        print(i, fibonacci(i), cnt )
```

Después probamos el código con un rango de 2 al 35, ya que, debido a la manera de operar de la función, un elemento que fuera mayor que el elemento 35 tarda mucho en ejecutarse. De esta grafica esperamos que la comparación entre el número de elementos y el número de iteraciones represente una curva, por la manera en la que está definida la función.

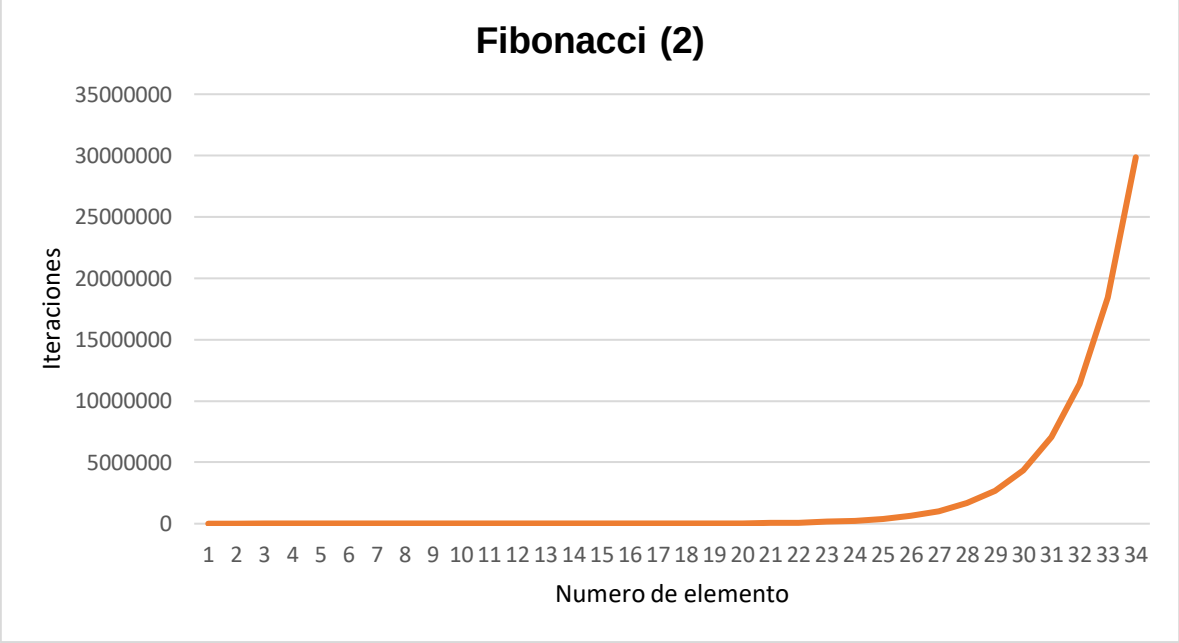
Tabla:

Elemento	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Iteraciones	3	5	9	15	25	41	67	109	177	287	465	753	1219	1973	3193	5167	8361

19	20	21	22	23	24	25	26	27	28	29	30
13529	21891	35421	57313	92735	150049	242785	392835	635621	1028457	1664079	2692537

31	32	33	34	35
4356617	7049155	11405773	18454929	29860703

Gráfica:



III. Fibonacci (3)

Con Fibonacci (3) se busca obtener los elementos correspondientes de la sucesión de Fibonacci. Para poder generar nuestro código primero es necesario definir un arreglo vacío, después una función para darle instrucciones acerca de cómo obtener el elemento deseado. Primeramente, declararemos una variable global, que será la encargada de contar el número de iteraciones realizadas por el programa para poder obtener dicho elemento que nosotros le solicitemos, después la inicializamos en cero para que lleve la cuenta bien, aquí mismo declaramos el arreglo anteriormente definido como global. Posteriormente comparamos si nuestro número de elemento es igual a cero o a 1, ya que en la sucesión de Fibonacci se inicia a partir del segundo término, diferente de 1. Si no, definimos una nueva variable que será igual a la suma de los dos anteriores elementos, después devolvemos el valor de esta variable.

Código en Python:

```
#Fibonacci(3)

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

#intervalo del 2 al 35
for i in range(2,35):
    cnt=0
    print(i,fibonacci(i),c)
```

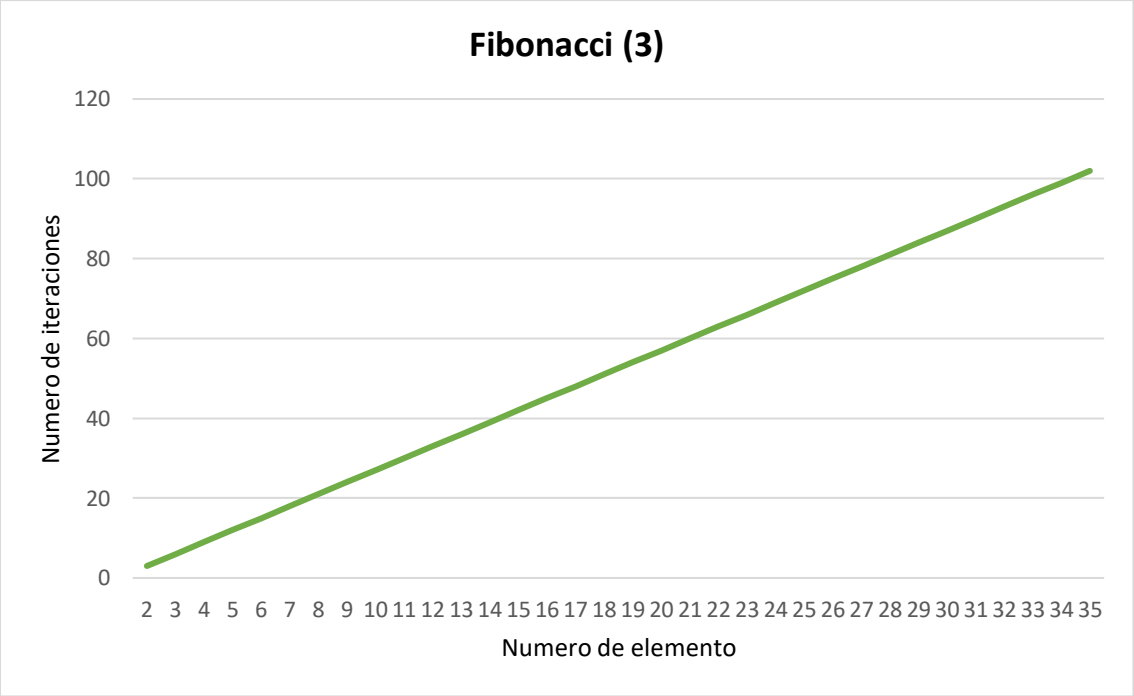
De este código esperamos que su representación cuando se comparen las iteraciones con el numero de elemento, sea una línea recta, ya que se está utilizando un código diferente a los dos primeros. Se probo de igual manera con un rango de 2 a 35 de numero de elemento.

Tabla:

Elemento	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Iteraciones	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
60	63	66	69	72	75	78	81	84	87	90	93	96	99	102

Gráfica:



IV. Algoritmo para hallar primos

Este algoritmo fue programado con el objetivo de poder determinar si un numero dado es primo. Para construir el algoritmo primero definimos la función que será la encargada de ejecutar todas las instrucciones posteriormente.

Comenzamos con declarar nuestra variable contadora como global para poderla iterar dentro de un ciclo, en el cual vamos a extraer la raíz cuadrada del número que queremos determinar si es primo o no, después dependiendo de este resultado vamos a proceder, si es divisible entre su raíz no es primo si no si lo es.

Dentro del programa se agrega un ciclo para poder determinar cuáles de los números dentro del rango de 2 al 35, de donde nos interesa obtener el valor del contador para poder realizar una gráfica y mostrar cómo se comporta el algoritmo.

Código en Python:

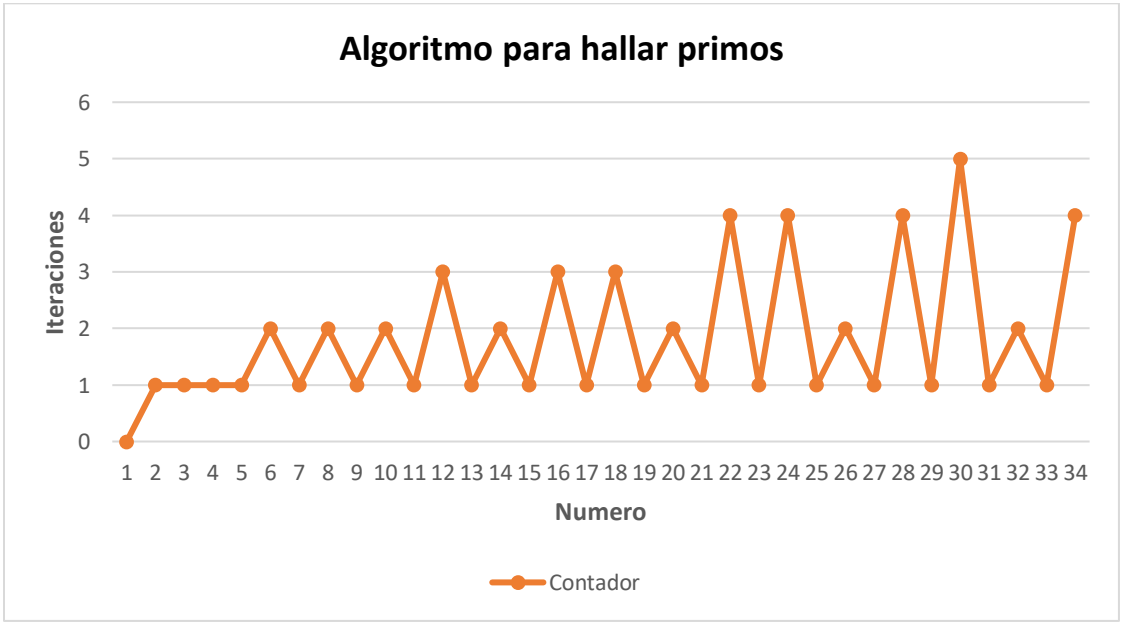
```
#Algoritmo para determinar si un numero es primo
def primo(n):
    global cnt

    for i in range (2,round(n**(1/2)+1)):
            cnt=cnt+1
            if ((n%i)==0):
                return ("No es primo")

    return ("Es primo")

for i in range (2,36):
    cnt=0
    primo(i)
    print(i,cnt)
```

Tabla:

Numero	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Contador	0	1	1	1	1	2	1	2	1	2	1	3	1	2	1	3	1	3	1	2	1	4	1	4

26	27	28	29	30	31	32	33	34	35
1	2	1	4	1	5	1	2	1	4

Gráfica:

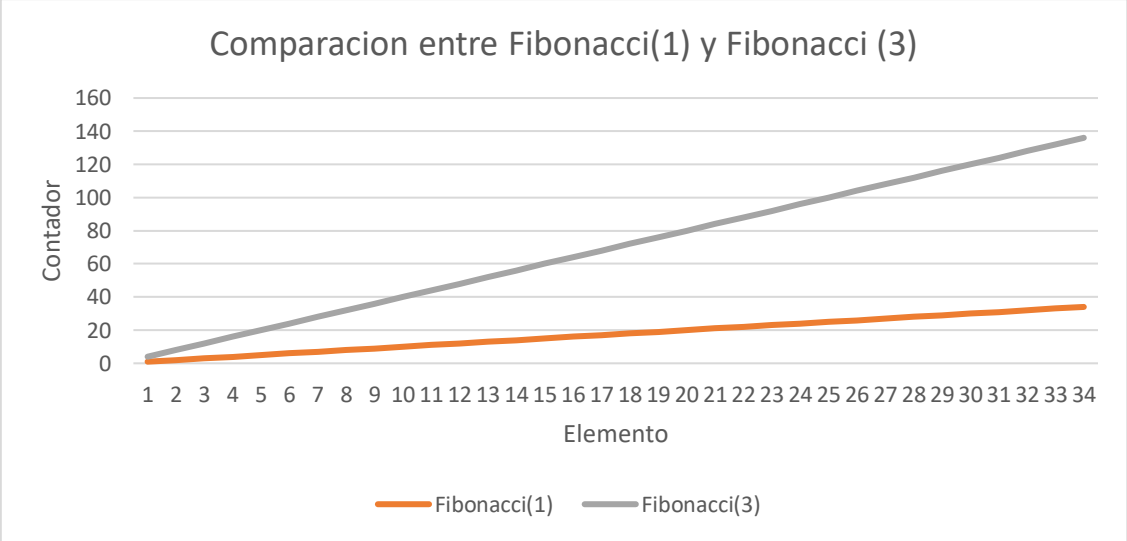


V. Conclusión

Los tres algoritmos Fibonacci, son tres maneras de hallar el termino deseado en esta sucesión.

- En Fibonacci (1) el código que empleamos resulta ser recursivo, y en comparación los otros dos, resulta ser más eficiente que el (2) En este algoritmo podemos ver, de manera gráfica que su representación es una línea recta, es decir el contador va aumentando de manera ‘’normal’’ al ir subiendo el número de elemento que deseamos encontrar. Cabe destacar que este algoritmo no utiliza memoria.
- En Fibonacci (2) el código empleado resulta ser iterativo, además de que representa ser más ineficiente que (1) y (3), se ve grandemente reflejado en las cantidades que la variable contadora arroja al momento de ser ejecutado.
- En Fibonacci (3) el código usado, lo podemos diferenciar de (1) porque utilizamos un arreglo, el cual se declara previamente. Este tipo de código resulta ser recursivo, pero este si utiliza una memoria (el arreglo) para el almacén de datos. Este resulta ser nuestra segunda mejor opción, ya que la mejor es Fibonacci (1).

En la siguiente grafica podemos visualizar lo anteriormente dicho:



Nótese que no se comparó Fibonacci (3) ya que no tiene caso, es decir a simple vista podemos descartarlo por el alto número de operaciones que realiza, por lo que en la gráfica nos centramos en (1) y (2), ya que quizá esta comparación no la podamos realizar de manera instantánea. Sin embargo, concuerda con lo anteriormente dicho, Fibonacci (1) es nuestra mejor opción y lo podemos comprobar de manera gráfica aun sin incluir el (3).

De esta manera podemos concluir que existen maneras diversas de obtener un mismo resultado, pero habrá que ponerlas aprueba para poder saber cuál es la mejor.

Después, en el algoritmo para hallar primos, podemos deducir que cuando nosotros queremos determinar si un número es primo, regularmente con los números pares hace muy pocas iteraciones, es decir solo 1, y cuando son impares realiza un poco más, pero, sin embargo cuando se trata de un numero primo, tiende a realizar más operaciones, sin embargo, estas mismas iteraciones no se disparan en valores más grandes, por lo que podemos concluir que el algoritmo es eficaz ya demás es fácil de programar y de entender, además la gráfica mostrada anteriormente de este algoritmo nos ayuda para poder darnos una idea de cómo funciona y como trabaja.