Instituto Politécnico Nacional

Escuela Superior de Cómputo

Semestre 2023-1

PRACTICA 2

Grupo: 3CV11

Materia: Analisis de Algoritmos

Alumnos:

Isaac Sánchez Verdiguel

isanchezv1603@alumno.ipn.mx

Axel Treviño Palacios

atrevinop1500@alumno.ipn.mx

Maestro:

Benjamin Luna Benoso

26 Octubre 2022

Índice general

Índice de figuras

1 | Introducción

1.1. Resumen

La practica consta del análisis a priori y posteriori de dos algoritmos; sucesión de Fibonacci y encontrar números perfectos. Ambos algoritmos fueron desarrollados mediante el lenguaje de programación **Python** en un ambiente virtual de **Linux**.
Palabras Clave: Python, Algoritmo, Temporalidad, Análisis.

1.2. Introducción

Los algoritmos son una parte fundamental de la ciencia de la computación, ya que estos al ser computables pueden dar solución o una idea más concreta acerca de la solución de un problema.
Un algoritmo no siempre dará una solución correcta, lo cual jamás será malo, porque esto nos ayudará a poder minimizar su radio de error. Una característica casi obligatoria para el buen funcionamiento de un algoritmo es su **rendimiento y eficacia**. El rendimiento adecuado se encuentra en la solución más rápida y menos costosa [Cormen, 2009].
Los algoritmos presentados en esta práctica presentan una complejidad temporal polinomial y no polinomial, siendo calculados por cuanto tarda en ejecutarse dependiendo la cantidad de valores de entrada [drkbugs, 2021].

2 | Desarrollo

2.1. Conceptos Básicos

La **complejidad temporal**, dentro del análisis de algoritmos, es el número de operaciones que ejecuta un algoritmo en cierto tiempo. Su denotación es T(n) y puede ser analizada mediante dos tipos de análisis:

- Análisis de priori: entrega una función que muestra el tiempo de cálculo de un algoritmo.
- Análisis a posteriori: es la prueba en tiempo real del algoritmo, midiendo su costo mediante valores de entrada.

El análisis de complejidad temporal define que un algoritmo alcanza su máximo potencial cuando los valores de entrada son mayores al tiempo estimado de ejecución, siendo que es factible poder completar sus ejecuciones en menor tiempo posible.
Los algoritmos expuestos en esta práctica son: sucesión de Fibonacci y Números Perfectos
La sucesión de Fibonacci se comprende como la sucesión infinita de números naturales, donde cada término es la suma de los dos anteriores [Wikipedia, 2021].
Definida por la ecuación:

$$f(0) = 0 \tag{2.1}$$

$$f(1) = 1 \tag{2.2}$$

$$f(n) = f_(n-1) + f_(n-2) \tag{2.3}$$

Mientras que un numero perfecto se define como un número entero positivo que es además igual a la suma de sus divisores positivos [Valeria Superprof, 2022]. Por ejemplo: 6 = 1 + 2+ 3

2.2. Algoritmo 1: Sucesión de Fibonacci (iterativa)

2.2.1. Resumen del problema

Ejecutar y realizar los analisis a priori y posteriori de un algoritmo que sea capaz de calcular la sucesión de Fibonacci n veces de forma iterativa, y realizar un análisis a posteriori de otro algoritmo que ejecute la sucesión de Fibonacci de forma recursiva.

2.2.2. Pseudocodigo Algoritmo 1 Fibonacci Iterativo

El algoritmo consta de dos números (n1, n2) los cuales servirán para realizar el ciclo de la sucesión de Fibonacci. La función admite como parametro la variable "lim"que define hasta que número se calculará la sucesión.

Algorithm 1: Fibonacci Iterativo

Result: $n2$
$n1 = 1$;
$n2 = 1$;
for $i \leftarrow 0$ **to** $limit - 2$ **do**
    $n2+ = n1$;
    $n1 = n2 - n1$;
return n2;

2.3. Algoritmo 2: Sucesión de Fibonacci (recursiva)

2.3.1. Pseudocodigo Algoritmo 2 Fibonacci Recursivo

El algoritmo recursivo de la sucesión de Fibonacci, llama la función con parámetros *lim, n1, n2*, siendo el límite, el primer término y el segundo término respectivamente. Dentro de la función se evalúa que el limite sea mayo que 2 para poder mostrar más términos de la sucesión, en caso de que esta condición se cumpla, se retorna nuevamente la función hasta que el límite sea menor a 2.

Algorithm 2: Fibonacci Recursivo

Result: $n2$
$n1 = 1$;
$n2 = 1$;
if *lim <2* **then**
    return 1;
else
    return n2 + fibonacci(lim - 1, n2, n1 + n2) - n1;

2.4. Algoritmo 3: Encontrar Números Perfectos

2.4.1. Resumen del problema

Realizar el análisis a priori y posteriori de un algoritmo que defina si un número es perfecto o no mediante la suma de sus divisores menores. En base al algoritmo para definir un número perfecto, se implementa otro que muestre los números perfectos encontrados.

2.4.2. Pseudocodigo Algoritmo 3 Encontrar Número Perfecto

El algoritmo consta de una función la cual recibe como parametro un número a consultar si es perfecto o no. El proceso para descubrirlo es aplicar, dentro de un ciclo *for* que va de 0 hasta el menor numero divisible (dado por la función piso), un condicional que verifique que el modulo del número es igual a cero, si cumple la condición a m se le suma un 1. Al finalizar el ciclo *for* se consulta si m es igual al número dado, en caso de que sí significa que es un número perfecto.

Algorithm 3: Encontrar Número Perfecto

Result: $verificador = Booleano$
$m = 0$;
for $i \leftarrow 0$ **to** $ceil(n/2) + 1$ **do**
    if n *modulo* $i == 0$ **then**
        $m+ = 1$;
    else
        continue;
if $m == n$ **then**
    return verificador = True;
else
    return verificador = False;

2.5. Algoritmo 4: Mostrar Números Perfectos

2.5.1. Pseudocodigo Algoritmo 4 Encontrar Número Perfecto

El algoritmo, junto con la función **Perfecto()** muestra los números perfectos existentes en más de mil números. Funciona de tal manera que con un *ciclo while* verifique y almacene el número perfecto. Siendo m el conteo de números perfectos e i el número perfecto.

Algorithm 4: Mostrar Número Perfecto

Result: $perfectos = []$
$m = 0;$
$i = 0;$
while $m < n$ **do**
    print("Probando i") ;
    if *Perfecto(i)* **then**
        $m = m + 1;$
        print("Perfecto m: i") ;
    else
        continue;
    $i+ = 1$;

3 | Experimentación y Resultados

Aquí se presentaran los resultados del **Análisis a Priori y Posteriori** de cada algoritmo solicitado respectivamente.

3.1. Algoritmo 1: Fibonacci Iterativo

El algoritmo codificado en *Python* fue testeado en un entorno virtual de Linux, donde el algoritmo pudo calcular al menos 900 elementos de la serie de Fibonacci.

3.1.1. Analisis a Priori

La figura 3.1 presenta el análisis a priori realizado sobre el código fuente del algoritmo de la sucesión de Fibonacci recursiva. Concluyendo que el algoritmo presenta una complejidad lineal $f(n) = O(n)$ porque en cada iteración solamente se asignan dos variables y se realiza una suma.

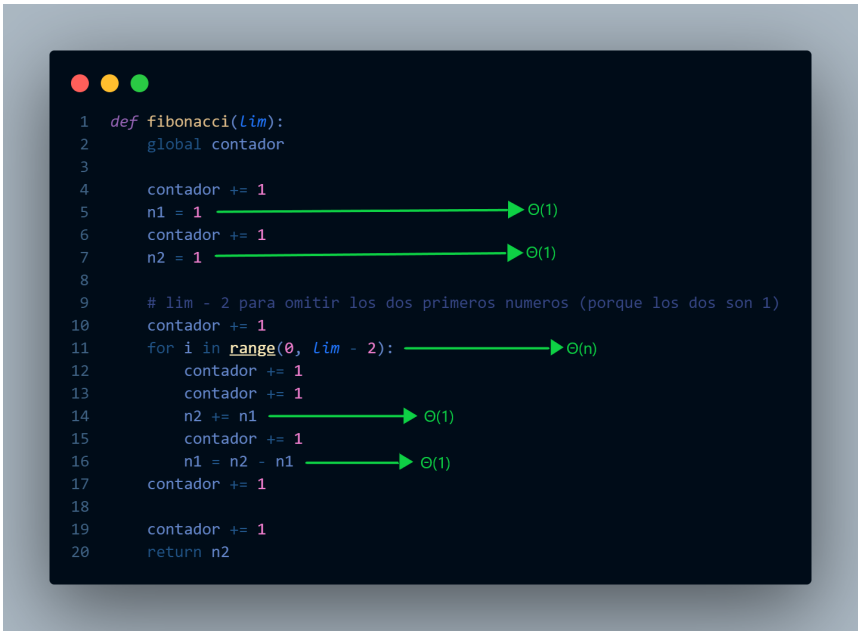

Figura 3.1: Analisis a Priori: Fibonacci Iterativo

3.1.2. Analisis a Posteriori

En el análisis posteriori se verifica el análisis a priori, efectivamente es una función lineal que incrementa en tiempo como se vayan sumando más entradas. Para la gráfica mostrada a continuación (figura 3.2) se muestran pocos elementos para notar el crecimiento en la función.

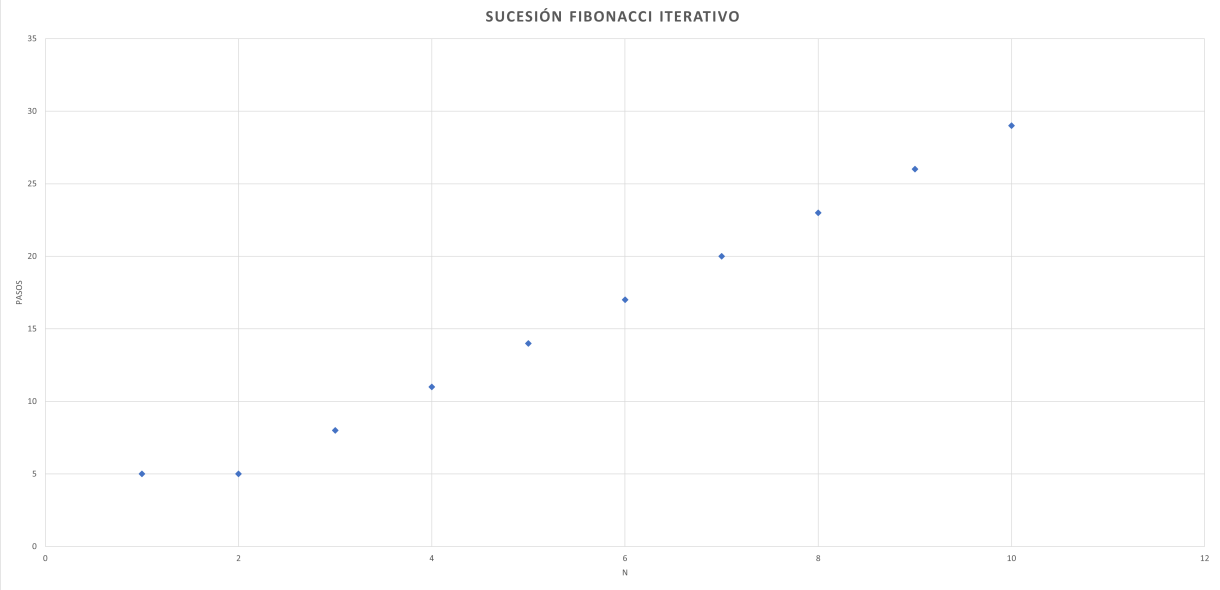

Figura 3.2: Analisis a Posteriori: Fibonacci Iterativo

3.2. Algoritmo 2: Fibonacci Recursivo

El algoritmo codificado en *Python* fue testeado en un entorno virtual de Linux, donde el algoritmo pudo calcular al menos 900 elementos de la serie de Fibonacci con un tiempo de ejecución creciente.

3.2.1. Analisis a Posteriori

El algoritmo analizado en posteriori, dio como resultado un tiempo de ejecucion similar al algoritmo en su forma iterativa. En la figura 3.3 se observa que el crecimiento es lineal y constante, dando como resultado una función lineal. Se concluye que el algoritmo incrementa su tiempo de ejecución a la par de los valores de entrada.

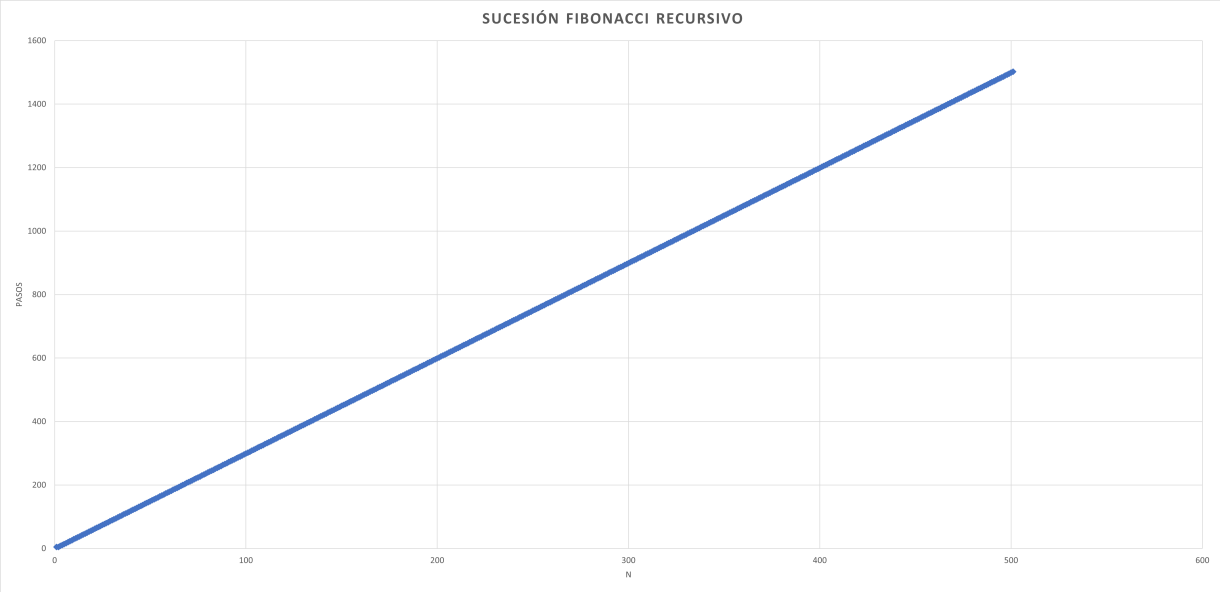

Figura 3.3: Análisis a Posteriori: Fibonacci Recursivo

3.3. Algoritmo 3: Encontrar Numero Perfecto

El algoritmo codificado en *Python* fue testeado en un entorno virtual de Linux en donde el algoritmo, en base a un numero como parametro de la funcion, pudo verificar si se trataba de un numero perfecto o no.

3.3.1. Análisis a Priori

En el análisis a priori del algoritmo para encontrar un número perfecto concluyó que el algoritmo presenta una complejidad lineal, esto debido a que el for ciclico solamente realiza una operación y acumula una sola variable en cada iteración. El análisis fue realizado sobre el código fuente, se puede observar en la figura 3.4.

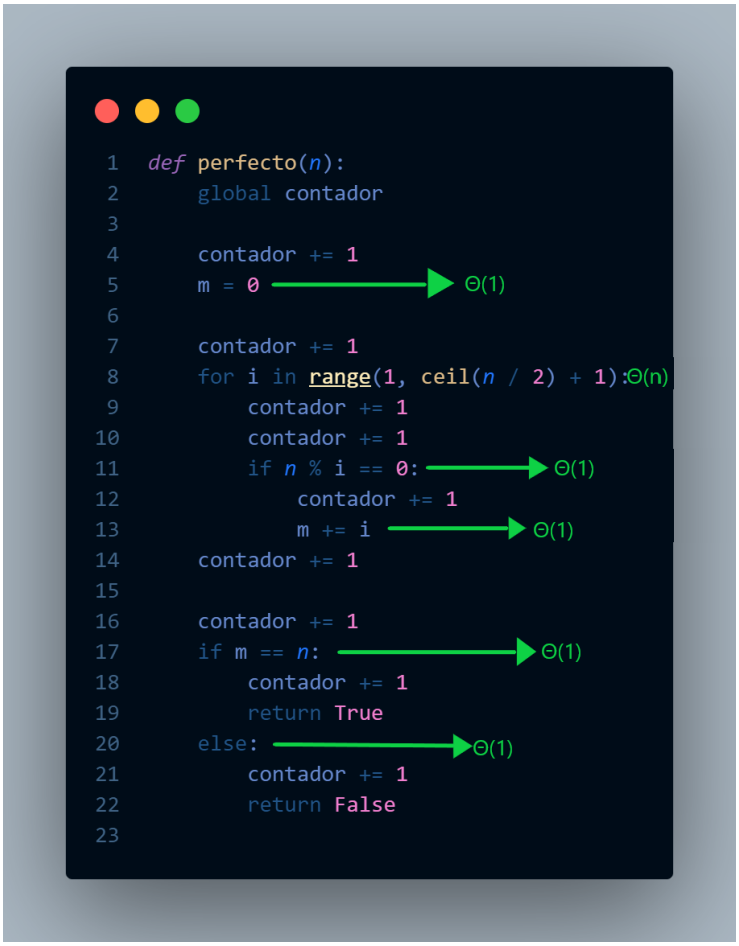

Figura 3.4: Análisis a Priori: Encontrar Perfecto

3.3.2. Analisis a Posteriori

Con los resultados, se observa que el algoritmo requiere más pasos conforme el número de entradas incrementa. En la figura 3.5 tal podemos observar que esto se cumple dentro de la función.

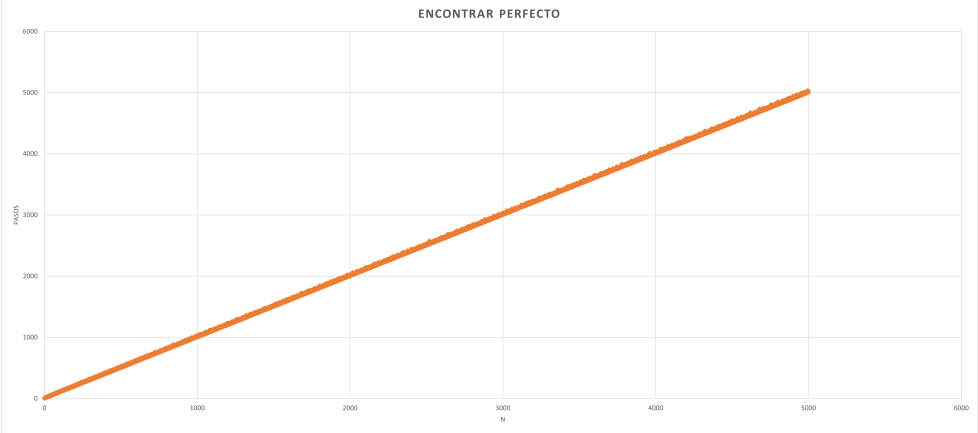

Figura 3.5: Análisis a Posteriori: Encontrar Perfecto

3.4. Algoritmo 4: Mostrar Numeros Perfectos

El algoritmo codificado en *Python* fue testeado en un entorno virtual de Linux en donde el algoritmo verifica junto con la función *Perfecto()* que un número sea perfecto para después mostrarlo en pantalla. Esta función numera los números perfectos.

3.4.1. Analisis a Posteriori

El algoritmo analizado a posteriori dio como resultado que existen 6 números perfectos en un conjunto de 5 mil números. En la figura 3.6 se observa que se ocuparon más de 350 millones de pasos para poder encontrar el sexto número perfecto.

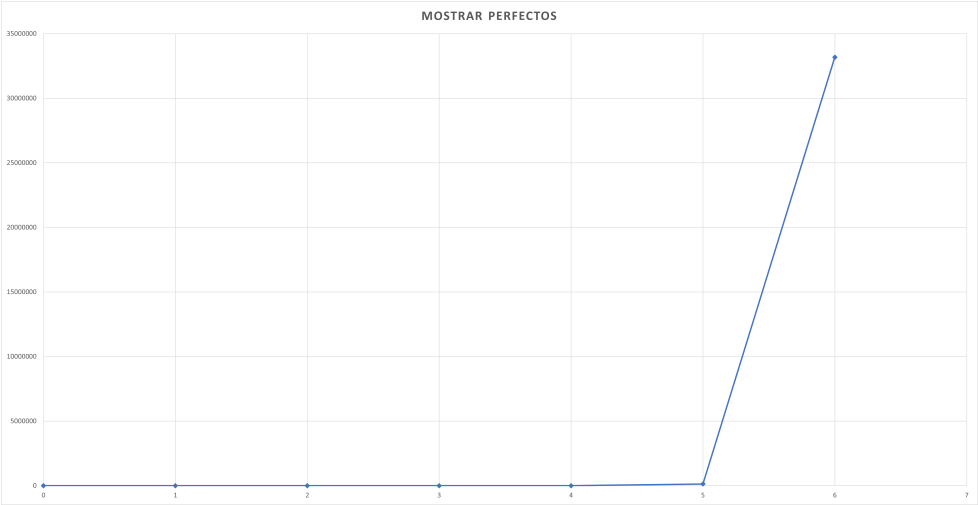

Figura 3.6: Análisis a Posteriori: Mostrar Perfecto

4 | Conclusiones

4.1. Conclusiones Generales

La práctica fue concluida con satisfacción, ya que nos resultó bastante didactico el modo de emplear un algoritmo tan común como el de Fibonacci para hacer un análisis certero de su rendmiento temporal.
Tuvimos complicaciones en la ejecución de los algoritmos porque queriamos medir su complejidad con cantidades muy grandes de números, presentando así problemas para poder continuar con la ejecución de los algoritmos.
Comprendimos de mejor manera un análisis a priori a un algoritmo iterativo.

4.2. Isaac Sánchez - Conclusiones

Esta práctica me permtió desarrollar mis habilidades a la hora de realizar algoritmos recursivos, ya que es un concepto que no dominaba. De igual manera, el análisis en la parte de priori fue más sencilla de plasmar para dar una idea. Mis conceptos que aprendí de mejor manera fue: recursividad y como encontrar un número perfecto.

Figura 4.1: Isaac Sánchez

4.3. Axel Trevino - Conclusiones

Durante esta práctica vimos que la recursividad no siempre es buena, ya que en el de Fibonacci aparte de necesitar más optimización no pudimos correrlo con números grandes porque causábamos un stack overflow

Figura 4.2: Axel Treviño

Bibliografía

[Cormen, 2009] Cormen, T. H. (2009). *Introduction to Algorithms*. The MIT Press, London.

[drkbugs, 2021] drkbugs (2021). ¿qué es la complejidad temporal? https://www.drk.com.ar/qu%C3%A9-es-la-complejidad-temporal/. [Online; accessed 25-October-2022].

[Valeria Superprof, 2022] Valeria Superprof (2022). Números perfectos: Qué son, cómo calcularlos y ejemplos. https://www.superprof.mx/blog/utilidad-numeros-perfectos/. [Online; accessed 25-October-2022].

[Wikipedia, 2021] Wikipedia (2021). Sucesión de fibonacci. https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci. [Online; accessed 25-October-2022].