Instituto Politécnico Nacional

Escuela Superior de Cómputo

Semestre 2023-1

PRACTICA 1

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

07 Septiembre 2022

Índice general

Índice de figuras

1 | Introducción

1.1. Resumen

La práctica consta del análisis de complejidad de dos algoritmos iterativos y sus respectivas gráficas.

Python

Algoritmo

Complejidad

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]
La importancia de conocer el costo y las soluciones de un algoritmo es que sepamos si las respuestas dadas son las esperadas. Esto nos ayuda a resolver los problemas de manera concisa. (cita)
Dentro de esta práctica, podremos ver dos algoritmos puestos a prueba de su complejidad temporal. Dando respuesta a sus diversas soluciones, desde las más eficientes hasta las que hacen que el algoritmo no pueda proporcionar las respuestas esperadas.

2 | Desarrollo

2.1. Conceptos Básicos

Omega Ω es la connotación que define el mejor caso del algoritmo.

Big O es la connotación que define el peor caso del algoritmo.

Teta Θ es la connotación que indica que no existe ni el peor ni el mejor caso en el algoritmo.

Analisis a Priori es una expresión matemática que indica el tiempo entre costo y pasos ejecutados de un algoritmo.

Analisis a Posteriori es el algoritmo codificado con el que se comprueba que la expresión matemática anterior expresa el costo del algoritmo.

Algoritmo de Euclides Algoritmo creado por el matematico griego Euclides, que permite una busqueda eficaz y rapida del maximo comun divisor de dos numeros naturales. [Khan Academy, 2020]

2.2. Algoritmo 1: Coincidencia de Arreglos

2.2.1. Resumen del problema

Devolver dos elementos identicos en dos arreglos provenientes de dividir un arreglo de 0 a 3n posiciones. Al momento de detectar ambos elementos, el algoritmo se detendrá, imprimiendo los valores de posición de la primera coincidencia.

2.2.2. Pseudocodigo Algoritmo 1

El algoritmo parte de un arreglo generado de 0 a 3n, el cual en su consiguiente es dividido en 2 para identificar todo el arreglo por mitades y así dar verificar si hay algún elemento repetido entre ambos arreglos. Hace un doble **cilo for** para recorrer ambos arreglos e ir indentando cada elemento, comparando con el del ciclo debajo.

Algorithm 1: Coincidencia de Arreglos

Result: $resultado = [position1, position2]$
for $auxiliar1 \leftarrow 0$ **to** $mitad_1$ **do**
    $j \leftarrow 0$;
    for $auxiliar2 \leftarrow 0$ **to** $mitad_2$ **do**
        if $auxiliar1 = auxiliar2$ **then**
            $resultado[0] \leftarrow [TRUE]$;
            $resultado[1] \leftarrow i$;
            $resultado[2] \leftarrow j + arreglo.length/2$;
            $salir \leftarrow TRUE$;
            break;
        else
            continue;
        $j+ = 1$;
    $i+ = 1$;
    if $salir == TRUE$ **then**
        break;
    else
        continue;

2.3. Algoritmo 2: Algoritmo de Euclides

2.3.1. Resumen del problema

Encontrar el peor caso dentro del algoritmo de Euclides. El algoritmo debe funcionar de manera que para el peor caso m y n sean numeros enteros positivos consecutivos de las serie de Fibonacci. Para determinar su complejidad hay que graficar la curva del peor caso y puntos para el no peor caso.

2.3.2. Pseudocodigo Algoritmo 2

El algoritmo consiste en un **ciclo while** que evalua que m y n tengan maximo comun divisor. Este algoritmo se ejecuta dentro de un **ciclo for** que avanza hasta 1000 pasos con numeros dados aleatoriamente o de la serie de Fibonacci. Siguiendo el algoritmo de Euclides, el proceso del ciclo avanza hasta que n=0, esto quiere decir que el maximo comun divisor tomará el lugar de m siendo este el MCD de ambos numeros enteros ingresados a la función.

Algorithm 2: Algoritmo de Euclides

Result: $max(m, n)$

while n *!= 0* **do**

    $r = m//n$;

    $m = n$;

    $n = r$;

return n;

3 | Experimentación y Resultados

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

3.1. Algoritmo 1

El algoritmo codificado en *Python* fue testeado en un entorno virtual de Linux, donde pudimos obtener resultados del mejor caso, peor caso y del algoritmo funcionando en su totalidad normal.

3.1.1. Mejor Caso

Para el mejor caso definimos que la coincidencia se encontrará justo en el primer elemento de cada arreglo. Esto ocasiona que el algoritmo ejecute sus ciclos una sola vez ya que la condición dentro del ciclo *for* terminaría el programa. En la gráfica podemos observar que aunque el tamaño del arreglo cambie el numero de operaciones es constante, lo cual da una linea recta desplazada desde el número de operaciones hasta el arreglo con n=100.

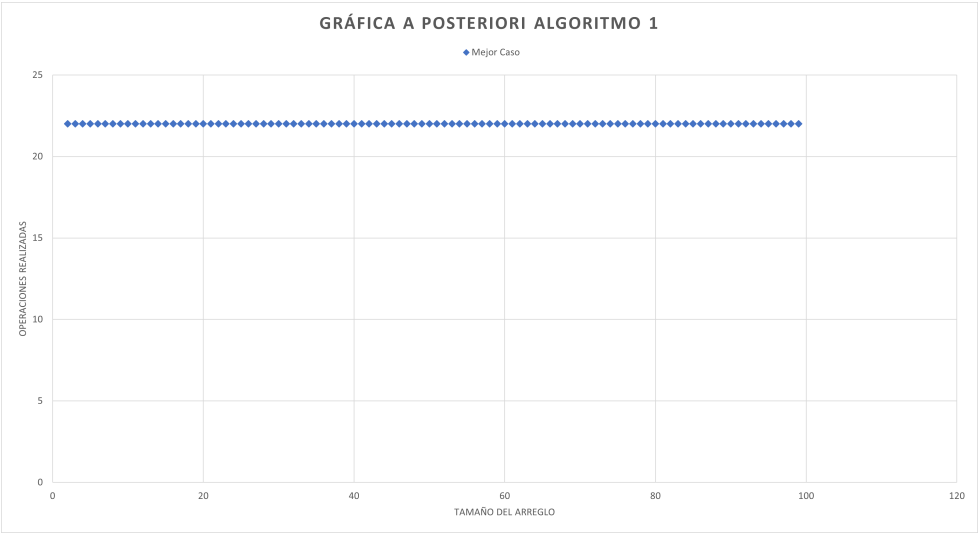

Figura 3.1: Gráfica Mejor Caso

3.1.2. Peor Caso

En el Peor Caso, definimos que sería cuando no haya ninguna coincidencia en los dos arreglos, lo que ocasionaría que conforme el arreglo incremente su tamaño en n, los pasos ejecutados serán cada vez mayores, ya que al recorrer ambos arreglos y no encontrar nada, se convertiría en un sin fin de pasos. La gráfica muestra esta conclusió con una función cuadratica que parte desde 15 pasos hasta casi 8 mil pasos en el tamaño n=100.
La complejidad del algoritmo es:

$$O(n^2)$$

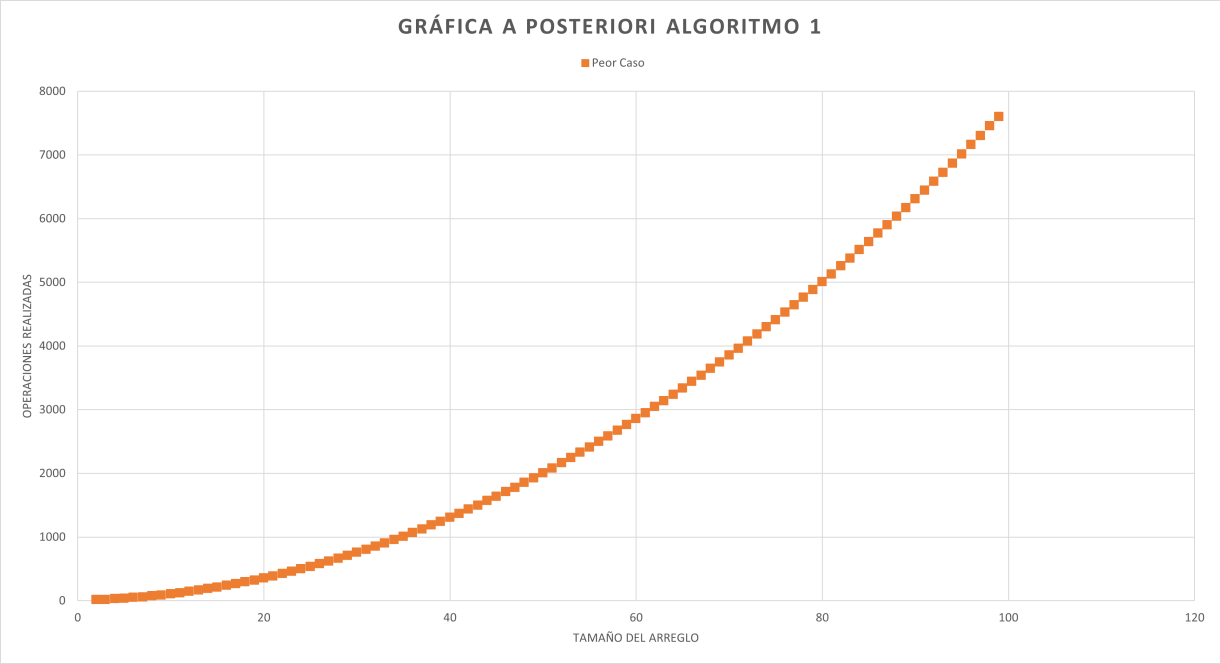

Figura 3.2: Gráfica Peor Caso

3.1.3. Análisis Normal

Obteniendo datos del algoritmo evaluado de manera "normal", podemos atestiguar que entre las cotas el algoritmo funciona de manera adecuada, siendo que tiende a un avance de pocas operaciones mientras escala de tamaño.

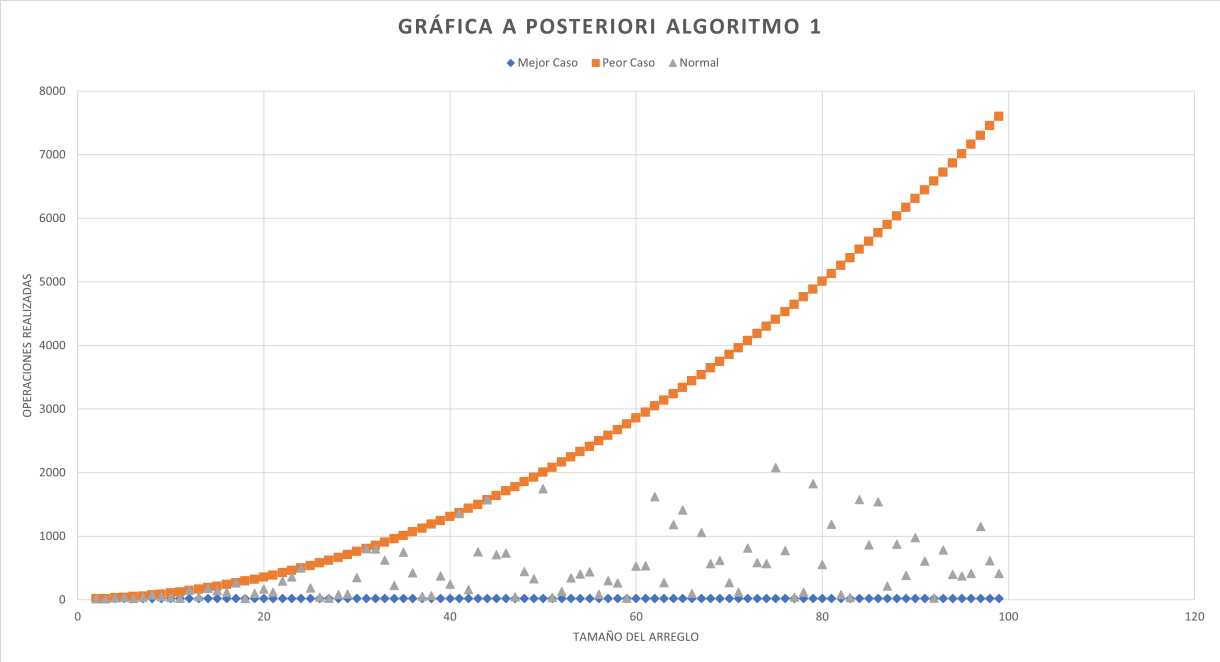

Figura 3.3: Gráfica Algoritmo 1

3.2. Algoritmo 2

El algoritmo de Euclides fue codificado en *Python*, compilado en un entorno virtual de Linux donde se pudo observar su comportamiento en el peor caso y en el caso normal. Estos resultados fueron capturados en un archivo *.csv* que posteriormente fueron graficados con la herramienta de Office Excel.

3.2.1. Peor Caso

El peor caso considera números enteros positivos consecutivos de la serie de Fibonacci. El resultado de esto nos dio una función logarítmica que desplaza desde el origen hasta el infinito. Esto nos muestra que al principio el algoritmo con la serie de Fibonacci identifica una cantidad grande de MCD que va achicando la cantidad conforme avanza la función. Su orden de complejidad es:

$$O(\log n)$$

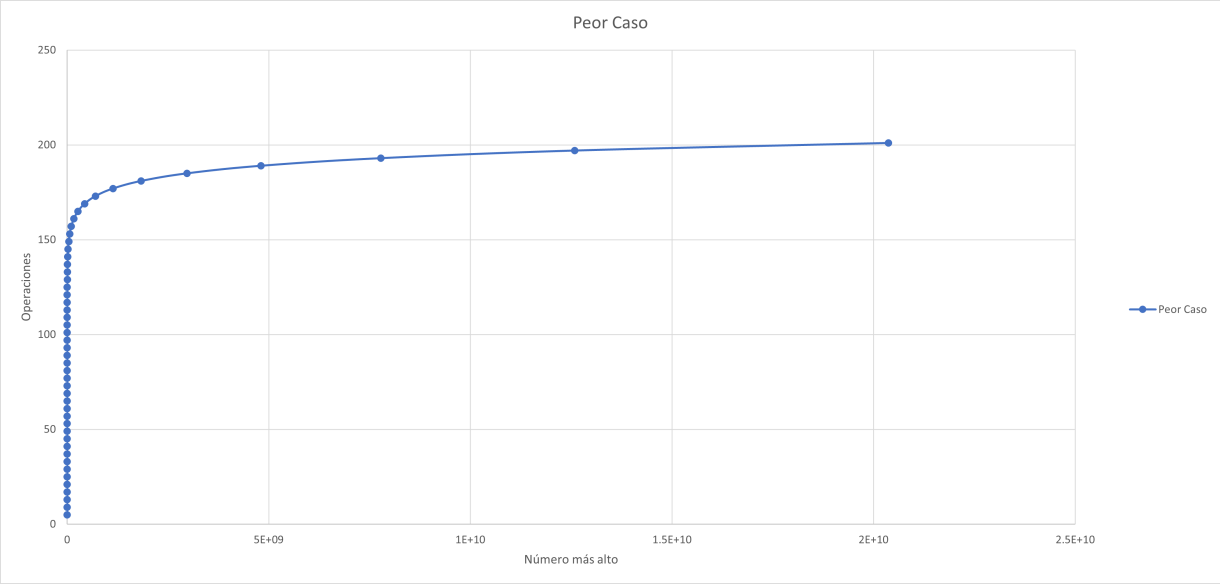

Figura 3.4: Gráfica Peor Caso

3.2.2. Caso Normal

Para el caso normal del algoritmo, usamos valores aleatorios entre el 2 y el número máximo del valor del peor caso. El resultado de esto nos dio puntos que se encuentran por debajo de la función logarítmica del peor caso.

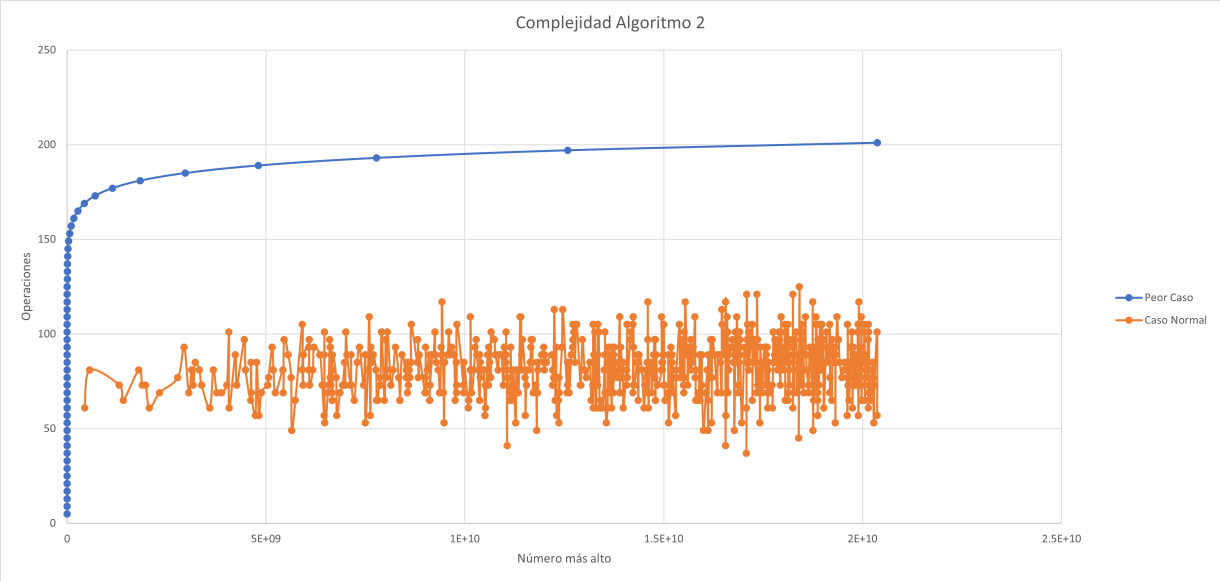

Figura 3.5: Gráfica Complejidad Algoritmo 2

4 | Conclusiones

4.1. Conclusiones Generales

Concluimos que el analisis para un algoritmo es una práctica que debería aplicarse en cualquier algoritmo para poder conocer que tan viable y que tan certera es la respuesta dada. Presentamos dificultades dentro del primer algoritmo a la hora de analizar los pasos que daban los ciclos dentro del código, esto lo resolvimos contando cada iteración dentro de nuestra función. Una vez que despejamos la mente, nos dimos cuenta que el segundo algoritmo puede realizarse con una función recursiva, que funciona de manera similar al ciclo while. No decidimos probar esta manera ya que aún no sabemos como manejar una analisis concreto a una función recursiva.

4.2. Isaac Sánchez - Conclusiones

Esta práctica resulto para mi un tanto complicada ya que no estaba familiarizado con los conceptos de complejidad y estaba un poco oxidado en programar. Al termino de la práctica los conceptos quedaron mas claros gracias al algoritmo de Euclides, este con complejidad log(n) me ayudo comprender porque un algoritmo es menos viable.

Figura 4.1: Isaac Sánchez

4.3. Axel Trevino - Conclusiones

La complejidad es una buena medida (inversa) de qué tan eficiente es un programa, y esto se puede ver en las gráficas: un algoritmo simple y sin eficientizar -el ejercicio 1- tuvo una complejidad de O(n∧2), mientras que un algoritmo eficientizado -el de Euclides- bajó a log(n), incluso si el primero si tiene mejor caso y no el segundo.

Figura 4.2: Axel Treviño

Bibliografía

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

[Khan Academy, 2020] Khan Academy (2020). El algoritmo de euclides. https://es.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm. [Online; accessed 06-September-2022].