Integrador “Analisis de Algoritmos”

Guijarro, Luciano Ezequiel

Programacion I

Profesora: Julieta Trapé

Entrega: 09/06/2025

Introducción

Para este trabajo elegí el tema **Análisis de Algoritmos** porque me pareció interesante entender cómo se mide el rendimiento de un código. Muchas veces uno programa algo que “funciona”, pero no se pregunta si es eficiente o si podría hacerse mejor. Este tema justamente se trata de eso: analizar cuánto tarda un algoritmo en ejecutarse, cuánta memoria usa, y ver si hay maneras más rápidas o más ordenadas de hacerlo.

Me pareció útil porque es algo que se puede aplicar a cualquier programa, sea chico o grande. Saber cómo se comporta un algoritmo te ayuda a tomar mejores decisiones cuando tenés que elegir entre distintas formas de resolver un problema.

Con este trabajo me propuse entender bien los distintos tipos de análisis (como el empírico y el teórico), aplicar ejemplos simples en Python para medir rendimiento, y poder comparar cómo cambian los resultados según cómo esté escrito el código.

Marco Teórico

El análisis de algoritmos es básicamente ver cómo se comporta un código: cuánto tarda, cuánta memoria usa, y si es eficiente o no. Esto es clave en programación, porque no alcanza con que algo simplemente funcione — también tiene que funcionar bien.

1. Análisis empírico

Es cuando uno prueba el código directamente y mide cuánto tarda en ejecutarse. En Python, se suele usar el módulo time (Python Software Foundation, 2024) para ver cuánto demora un algoritmo en diferentes casos. Por ejemplo, se puede hacer un bucle que sume los números del 1 al 1 millón, y ver si eso tarda más que usar una fórmula directa.

Este tipo de análisis sirve para experimentar con datos reales. Aunque no da una respuesta exacta para todos los casos posibles, te permite ver cómo se comporta el código con entradas de distintos tamaños.

2. Análisis teórico (complejidad / Big-O)

Acá ya se trata más de pensar que de probar. Se analiza cuántas operaciones hace un algoritmo según la cantidad de datos que le pasamos, sin necesidad de ejecutarlo. Para esto se usa algo que se llama **notación Big-O**, que ayuda a entender cómo crece el tiempo de ejecución o el uso de memoria.

Por ejemplo:

- Un bucle que suma del 1 al n tiene una complejidad de **O(n)**, porque hace una operación por cada número (Cormen et al., 2009).
- En cambio, una fórmula como n*(n+1)/2 se hace en un solo paso, sin importar el valor de n. Esa es una operación **O(1)**, o sea constante.

Este análisis teórico permite comparar diferentes algoritmos y decidir cuál es más conveniente en base a cómo escalan cuando se les da mucha información.

Aplicado en Python

En Python se puede hacer tanto el análisis empírico como el teórico de manera bastante accesible. Por un lado, se puede usar time.time() o time.perf_counter() para medir tiempos de ejecución reales. Por otro, entendiendo la lógica del algoritmo se puede pensar en su eficiencia con Big-O, lo cual ayuda mucho a escribir mejores programas.

3. Caso Práctico

La idea fue probar cómo funciona el análisis de algoritmos usando ejemplos simples en Python. Para eso, armé dos casos donde se podía medir el tiempo que tardaba el programa en hacer una tarea, y además pensar teóricamente cuán eficiente era cada uno.

Uno de los casos fue sumar los números del 1 hasta un número grande usando un bucle. El otro fue hacer lo mismo pero con una fórmula matemática directa (la de Gauss). En ambos casos, usé herramientas de Python para medir cuánto tardaban en ejecutarse.

Ejemplo 1: Análisis empírico con un bucle

import time

```
def suma_con_bucle(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total

# Medimos el tiempo de ejecución
n = 10000000
inicio = time.time()
resultado = suma_con_bucle(n)
fin = time.time()

print("Resultado:", resultado)
print("Tiempo de ejecución:", fin - inicio, "segundos")
```

Este código recorre todos los números desde 1 hasta n y los va sumando. Como hace una operación por cada número, su complejidad es O(n).

Ejemplo 2: Análisis teórico con fórmula de Gauss

```
import time

def suma_gauss(n):
    return n * (n + 1) // 2

# Medimos el tiempo de ejecución
n = 10000000
inicio = time.time()
resultado = suma_gauss(n)
```

```
fin = time.time()


print("Resultado:", resultado)
print("Tiempo de ejecución:", fin - inicio, "segundos")
```

Esta versión usa una fórmula matemática que hace todo en un solo paso, sin bucles. Eso la convierte en una solución mucho más rápida, con complejidad O(1).

Elegí estos ejemplos porque son simples y ayudan a ver bien la diferencia entre una solución con bucle y una solución optimizada. Además, ya había usado el primero en clase, así que me pareció buena idea comparar ambos para entender la eficiencia.

No usé librerías raras ni cosas complicadas, solo el módulo time de Python que ya viene incorporado.

Probé los dos códigos con el mismo número (n = 10.000.000) y funcionaron perfecto: ambos dieron el mismo resultado. La diferencia más clara estuvo en el tiempo de ejecución. El bucle tardó varios segundos, mientras que la fórmula fue prácticamente instantánea.

4. Metodología Utilizada

Primero arranqué viendo qué era el análisis de algoritmos. Leí algunos artículos, busqué en la documentación oficial de Python y miré algunos ejemplos en páginas como GeeksforGeeks y Real Python. También repasé lo que habíamos visto en clase, especialmente sobre cómo medir tiempos de ejecución.

Después pensé en un ejemplo fácil para arrancar, así que usé uno parecido al de clase: un bucle que suma los números del 1 al que yo le diga. Como ya lo conocía, me pareció bueno para medir el tiempo que tardaba y ver qué tan eficiente era. Más adelante le agregué otro ejemplo más optimizado, usando la fórmula de Gauss, para comparar.

Fui probando el código en el IDE que uso siempre (Thonny y a veces VS Code), haciendo pequeñas pruebas para ver que los resultados fueran correctos. Usé time.time() para ver cuánto tardaba en ejecutarse cada función.

Todo lo hice yo solo, así que fui probando, corrigiendo errores si salían y ajustando lo necesario hasta que me pareció que estaba listo. Después de eso,

anoté los resultados y armé los comentarios para explicar bien qué hace cada parte del código.

5. Resultados Obtenidos

Con el caso práctico logré entender bien cómo medir el rendimiento de un algoritmo en Python. Usando el módulo time, pude comparar cuánto tarda una función que suma con un bucle frente a otra que lo hace con una fórmula directa (la de Gauss). Ambos códigos funcionaron perfecto, y los resultados fueron iguales, así que eso ya me confirmó que estaban bien.

Lo más interesante fue ver cómo cambia el tiempo que tarda en ejecutarse dependiendo del método que se usa. La versión con bucle se notaba más lenta cuando el número era grande, mientras que la de Gauss tardaba prácticamente nada. Eso me ayudó a ver en la práctica por qué importa pensar en la eficiencia de un algoritmo, aunque a veces haga lo mismo.

También me sirvió para practicar cómo escribir funciones, medir tiempos y dejar todo ordenado y comentado. No tuve errores graves, solo un par de detalles con los prints y los comentarios que fui acomodando mientras lo probaba. No llegué a usar muchas estructuras complejas ni librerías pesadas, pero para el objetivo del trabajo eso estuvo bien.

6. Conclusiones

Haciendo este trabajo aprendí bastante sobre cómo funciona el análisis de algoritmos, tanto desde lo teórico como en la práctica. Aunque al principio parecía medio complicado eso de pensar en la eficiencia, con ejemplos simples pude entender bien cómo medir tiempos y por qué es importante escribir código que no solo funcione, sino que también sea rápido y eficiente.

Me di cuenta de que muchas veces hay formas más inteligentes de resolver un problema. Por ejemplo, la fórmula de Gauss hace lo mismo que un bucle, pero muchísimo más rápido. Ver eso en números reales fue clave para terminar de entenderlo.

También me sirvió para practicar cómo organizar un proyecto, comentar bien el código, probar diferentes entradas y anotar los resultados. Siento que esto me va a servir mucho si en algún momento tengo que hacer programas más grandes o con más datos.

Una mejora que podría hacer en el futuro sería comparar más algoritmos entre sí o sumar visualizaciones para que los resultados se vean más fácil. Pero para el objetivo de este trabajo, creo que quedó completo y claro.

Bibliografía / Referencias

- Python Software Foundation. (2024). *Python Documentation*. Disponible en: https://docs.python.org/3/
- GeeksforGeeks. (s.f.). *Python – Time Functions*. Disponible en: https://www.geeksforgeeks.org/python-time-module/
- Real Python. (s.f.). *Measuring Time in Python*. Disponible en: https://realpython.com/python-timer/
- Big-O Cheat Sheet. (s.f.). Disponible en: https://www.bigocheatsheet.com/