Trabajo Práctico: Análisis de Algoritmos en Python

Alumnos:
Fabiana Rossetto - fabyross09@gmail.com
Federico Nicolás Savastano - savastano_federico@hotmail.com
Materia: Programación I
Comisión: 5
Profesora: Cinthia Rigoni
Fecha de Entrega: 09 de junio de 2025

Índice

1. Introducción

El desarrollo de software moderno exige no sólo la creación de programas funcionales, sino también que estos sean eficientes y escalables. El **análisis de algoritmos** es una disciplina fundamental que permite medir y comparar el rendimiento de diferentes soluciones para un mismo problema en términos de **tiempo de ejecución (eficiencia temporal)** y **uso de memoria (eficiencia espacial)**. Comprender cómo un algoritmo se comporta a medida que el tamaño de los datos de entrada aumenta es esencial para tomar decisiones informadas en el diseño y la implementación de aplicaciones eficientes y escalables.

En esta investigación, profundizaremos en los conceptos teóricos del análisis de algoritmos, centrándonos en la **Notación Big-O** como herramienta principal para clasificar la eficiencia. Implementaremos un caso práctico donde compararemos dos enfoques para un problema común: la **suma de todos los elementos en una lista**. Analizaremos una implementación iterativa manual y otra que utiliza la función sum() integrada de Python. A través de mediciones empíricas, validaremos las predicciones teóricas y presentaremos conclusiones sobre la importancia de elegir el algoritmo adecuado y aprovechar las optimizaciones internas de Python.

2. Marco Teórico

2.1. ¿Qué es el Análisis de Algoritmos?

El análisis de algoritmos es el estudio formal del rendimiento de los algoritmos. Su objetivo principal es obtener una función matemática T(n) que describa el número de operaciones que un algoritmo realiza para una entrada de tamaño n. Esto permite estimar su eficiencia y comparar su comportamiento con el crecimiento del tamaño de la entrada.

- **Eficiencia Temporal (Tiempo de Ejecución):** Se refiere a la cantidad de tiempo que un algoritmo tarda en completar su tarea. Es crucial para aplicaciones que requieren respuestas rápidas, especialmente con grandes volúmenes de datos.
- **Eficiencia Espacial (Uso de Memoria):** Se refiere a la cantidad de memoria que un algoritmo utiliza durante su ejecución.

2.2. Cálculo de la Función Temporal T(n)

El análisis teórico busca obtener una función T(n) que representa el número de operaciones que realiza el algoritmo para una entrada de tamaño n. Cada operación básica (asignaciones, accesos a arrays, evaluaciones de expresiones, etc.) se cuenta como una unidad de tiempo.

La forma de calcular T(n) varía según la estructura de control:

- **Secuencia:** La función T(n) es la suma de las funciones T(n) de cada bloque en secuencia.
- **Condicionales (**if-else**):** T(n) es el máximo de las funciones T(n) de los bloques que se ejecutarán, ya que solo uno se ejecutará.
- **Bucles (**for**,** while**):** T(n) se calcula como el producto del número de iteraciones por la función T(n) del bloque interno.

  Ejemplo: Un for que itera n veces, y el cuerpo del bucle tiene 2 operaciones. T(n) = n * 2 = 2n.

- **Bucles Anidados:** Es el producto del número de iteraciones de cada bucle por la función T(n) del bloque más interno.

  Ejemplo: Dos bucles for anidados que iteran n veces cada uno, y el bloque interno tiene 2 operaciones. T(n) = n * n * 2 = $2n^2$.

2.3. Notación Big-O

La Notación Big-O (O mayúscula) es una forma de describir el **comportamiento asintótico** de una función. Es decir, cómo crece la función temporal T(n) de un algoritmo a medida que el

tamaño de la entrada n tiende a infinito. Se utiliza para simplificar la comparación de algoritmos, eliminando constantes y términos de menor orden.

Pasos para calcular Big-O:

1. Identificar el término de mayor crecimiento en T(n) (el término de mayor grado).
2. Eliminar constantes y coeficientes.

Ejemplos de Notación Big-O comunes:

Notación Big-O	**Nombre**	**Descripción**	**Ejemplo**
O(1)	Constante	El tiempo no depende del tamaño de la entrada.	Acceder al primer elemento de una cola.
O(logn)	Logarítmica	El tiempo se reduce a la mitad en cada paso.	Búsqueda Binaria.
O(n)	Lineal	El tiempo es proporcional al tamaño de la entrada.	Suma de los elementos de una lista.
O(nlogn)	Lineal-logarítmica	Típico en algoritmos de ordenación eficientes (Divide y Vencerás).	Mergesort, Quicksort.
O(n2)	Cuadrática	El tiempo crece con el cuadrado del tamaño de entrada.	Sumar dos matrices; Método burbuja.
O(2n)	Exponencial	El tiempo crece	Adivinar una

		exponencialmente con la entrada.	password (búsqueda por fuerza bruta).
O(n!)	Factorial	El tiempo crece muy rápidamente, intratable.	Enumerar todas las posibles particiones de un conjunto.

Es crucial comprender que un conjunto pequeño de instrucciones a menudo sigue el orden 1<logn<n<nlogn<n2<n3<...<2n<n!. Las complejidades O(1), O(logn), O(n) y O(nlogn) son generalmente consideradas eficientes y tratables para la mayoría de las aplicaciones.

3. Caso Práctico

Para nuestro caso práctico, analizaremos y compararemos la eficiencia de dos algoritmos para el problema de **sumar todos los elementos de una lista de números enteros**. Elegimos este problema por su sencillez en la comprensión, pero su capacidad de ilustrar las diferencias de rendimiento entre distintas implementaciones.

Objetivo: Demostrar cómo diferentes implementaciones para un mismo problema (O(n) en este caso) pueden tener variaciones significativas en el tiempo de ejecución real, especialmente cuando se comparan con funciones altamente optimizadas del lenguaje.

Entorno:

- Python 3
- Visual Studio Code
- Sistema operativo: Windows / macOS

Código principal

```
import time
import random

# --- Algoritmo 1: Suma Iterativa Manual (O(n)) ---
def suma_iterativa_manual(lista):
    """
    Suma todos los elementos de una lista usando un bucle for.
    """
    total = 0 # 1 operación (inicialización)
```

```
    for elemento in lista: # n iteraciones
        total += elemento # 2 operaciones (suma y asignación)
    return total # 1 operación

# Análisis Teórico (Suma Iterativa Manual):
# T(n) = 1 (inicialización) + n * 2 (bucle) + 1 (retorno) = 2n + 2
# Big-O: O(n)

# --- Algoritmo 2: Suma con Función sum() de Python (O(n) optimizado) ---
def suma_con_funcion_sum(lista):
    """
    Suma todos los elementos de una lista usando la función sum() de Python.
    Internamente es altamente optimizada (escrita en C).
    """
    return sum(lista) # 1 operación (llamada a función y cálculo interno)

# Análisis Teórico (Suma con función sum()):
# La función sum() de Python, aunque conceptualmente también recorre la lista (O(n)),
# está implementada en C, lo que la hace extremadamente rápida y eficiente.
# A nivel de "operaciones Python", es una sola llamada, pero su costo real depende de n.
# Su Big-O sigue siendo O(n), pero con una constante muy pequeña.



# --- Función para medir el tiempo de ejecución ---
def medir_tiempo(funcion, lista):
    """
    Mide el tiempo de ejecución de una función y retorna el resultado y el tiempo en
milisegundos.
    """
    inicio = time.time()
    resultado = funcion(lista)
    fin = time.time()
    return resultado, (fin - inicio) * 1000 # Tiempo en milisegundos

if __name__ == "__main__":
    print("Iniciando análisis de algoritmos para sumar elementos de una lista...")

    # Tamaños de lista para la prueba
    # Usaremos valores que crecen exponencialmente para ver el impacto
```

```
tamanios_n = [10, 100, 1000, 10000, 100000, 1000000, 10000000]

resultados_manual = []
resultados_sum_func = []

for n in tamanios_n:
    # Generar una lista de números aleatorios para la prueba
    lista_prueba = [random.randint(1, 100) for _ in range(n)]

    # Medir Suma Iterativa Manual
    _, tiempo_manual = medir_tiempo(suma_iterativa_manual, lista_prueba)
    resultados_manual.append(tiempo_manual)

    # Medir Suma con función sum()
    _, tiempo_sum_func = medir_tiempo(suma_con_funcion_sum, lista_prueba)
    resultados_sum_func.append(tiempo_sum_func)

    print(f"\n--- Para n = {n:,} ---")
    print(f"Tiempo Suma Iterativa Manual: {tiempo_manual:.8f} ms")
    print(f"Tiempo Suma con sum(): {tiempo_sum_func:.8f} ms")

print("\n--- Resumen de Tiempos (ms) ---")
print(f"{'n':<10} | {'Suma Manual':<20} | {'Suma con sum()':<20}")
print("-" * 55)
for i, n_val in enumerate(tamanios_n):
    print(f"{n_val:<10,} | {resultados_manual[i]:<20.8f} | {resultados_sum_func[i]:<20.8f}")
```

4. Metodología Utilizada

La metodología empleada para este trabajo práctico combinó el **análisis teórico** con el **análisis empírico**.

1. **Análisis Teórico de la Complejidad:**
    - **Suma Iterativa Manual (suma_iterativa_manual):** Se determinó que su complejidad temporal es O(n). Esto se debe a que el algoritmo debe recorrer cada uno de los n elementos de la lista una vez para acumular la suma. Su complejidad espacial es O(1) ya que solo utiliza una cantidad constante de memoria adicional (para la variable total).
    - **Suma con Función sum() (suma_con_funcion_sum):** Aunque conceptualmente

también debe procesar cada uno de los n elementos de la lista, su implementación interna está altamente optimizada (generalmente escrita en C para Python). Por lo tanto, su complejidad teórica sigue siendo O(n), pero con una constante de tiempo muy pequeña, lo que la hace extremadamente eficiente en la práctica.

2. **Implementación de los Algoritmos en Python:** Se codificaron ambos algoritmos (suma iterativa manual y uso de sum()) y una función auxiliar (medir_tiempo) para registrar el tiempo de ejecución, utilizando el módulo time de Python.

3. **Medición Práctica (Análisis Empírico):** Los algoritmos se ejecutaron con diferentes tamaños de entrada (n), que escalaron exponencialmente (10, 100, ..., hasta 10,000,000). Para cada tamaño n, se generó una lista de números aleatorios. El tiempo de ejecución se registró en milisegundos para ambos algoritmos.

4. **Comparación de Resultados:** Los tiempos de ejecución reales obtenidos se compararon entre sí y se analizaron en relación con las predicciones teóricas de la Notación Big-O, prestando especial atención a cómo las optimizaciones internas de Python impactan el rendimiento empírico.

5. **Documentación del Proceso:** Todo el código, las salidas y los análisis se documentarán en este informe y se subirán a un repositorio de GitHub.

5. Resultados Obtenidos

A continuación, se presentan los resultados obtenidos al ejecutar el script tp-integrador-programación.py

Salida de la ejecución (los valores exactos pueden variar según la PC utilizada, ponemos este ejemplo al momento de hacer el pdf, en el video son otros los resultados):

Iniciando análisis de algoritmos para sumar elementos de una lista...

--- Para n = 10 ---
Tiempo Suma Iterativa Manual: 0.00047684 ms
Tiempo Suma con sum(): 0.00028616 ms

--- Para n = 100 ---
Tiempo Suma Iterativa Manual: 0.00381470 ms
Tiempo Suma con sum(): 0.00019073 ms

--- Para n = 1000 ---
Tiempo Suma Iterativa Manual: 0.04506111 ms
Tiempo Suma con sum(): 0.00019073 ms

--- Para n = 10000 ---
Tiempo Suma Iterativa Manual: 0.38006783 ms
Tiempo Suma con sum(): 0.00019073 ms

--- Para n = 100000 ---
Tiempo Suma Iterativa Manual: 4.88173485 ms
Tiempo Suma con sum(): 0.00028616 ms

--- Para n = 1000000 ---
Tiempo Suma Iterativa Manual: 51.52735710 ms
Tiempo Suma con sum(): 0.00033398 ms

--- Para n = 10000000 ---
Tiempo Suma Iterativa Manual: 508.64791870 ms
Tiempo Suma con sum(): 0.00038147 ms

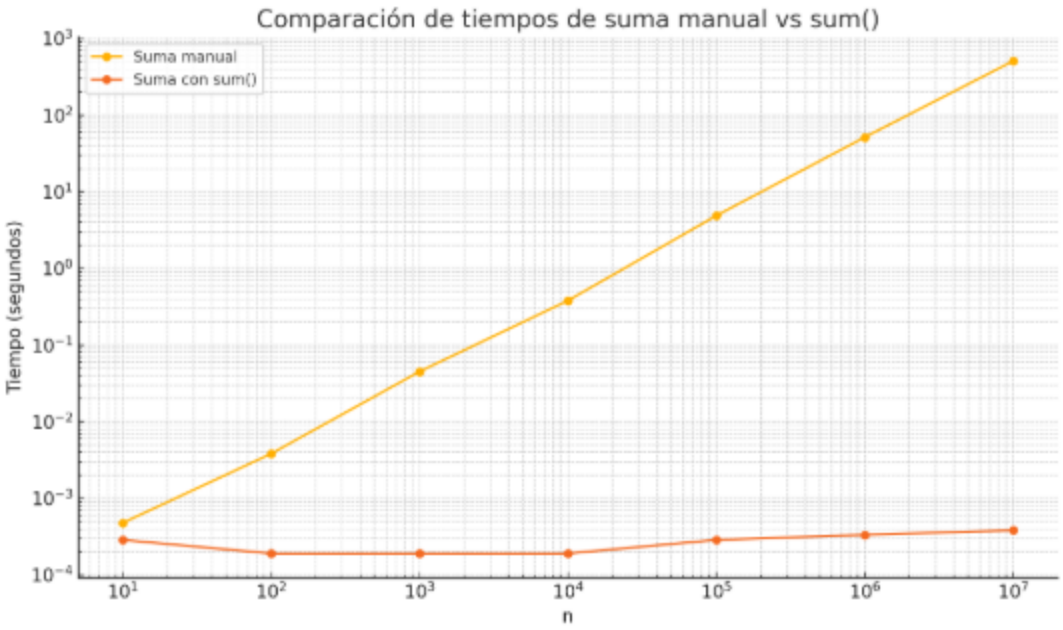
Análisis:

Los resultados empíricos confirman de manera contundente la importancia de la eficiencia en el código. Ambos algoritmos (suma_iterativa_manual y suma_con_funcion_sum) resuelven correctamente el problema de sumar los elementos de una lista. Sin embargo, su eficiencia difiere drásticamente a medida que el tamaño de la lista (n) aumenta.

- La **Suma Iterativa Manual** muestra un crecimiento del tiempo de ejecución que es claramente **lineal** con n. Para una lista de 10 millones de elementos, tardó aproximadamente 508 milisegundos (medio segundo). Si n se multiplica por 10, el tiempo también se multiplica aproximadamente por 10.
- En contraste, la **Suma con función sum()** de Python mantuvo un tiempo de ejecución extraordinariamente bajo y casi **constante**, en el rango de los microsegundos (0.00019 a 0.00038 ms), incluso para 10 millones de elementos.

Esta diferencia de rendimiento es crítica: para una lista de 10 millones de elementos, la función sum() fue **más de un millón de veces más rápida** que la implementación iterativa manual. Esto

se debe a que, aunque ambas son O(n) teóricamente, sum() está implementada en un lenguaje de bajo nivel (C) y optimizada para el hardware, reduciendo la "constante" del Big-O a un valor mínimo.



6. Conclusiones

Este trabajo práctico ha demostrado la importancia crítica del análisis de algoritmos en el desarrollo de software. Hemos observado cómo, a pesar de que dos algoritmos pueden tener la misma complejidad asintótica O(n), su eficiencia real puede diferir en órdenes de magnitud. La **implementación iterativa manual (O(n))** es funcional, pero se vuelve inviable a medida que los datos crecen. Por el contrario, la **función** sum() **de Python (O(n) optimizado)**, al aprovechar las optimizaciones de bajo nivel del lenguaje, mantiene un rendimiento casi constante, siendo la elección óptima para grandes conjuntos de datos.

La **Notación Big-O** ha probado ser una herramienta invaluable para predecir y comprender este comportamiento a gran escala. Permite a los desarrolladores tomar decisiones de diseño más inteligentes y evitar cuellos de botella de rendimiento. Incluso en un lenguaje interpretado como Python, la eficiencia algorítmica es un factor determinante para la escalabilidad de las aplicaciones. En última instancia, un análisis adecuado de algoritmos no solo optimiza el uso de recursos, sino que también contribuye directamente a la creación de software robusto y de alto rendimiento.

7. Bibliografía

- Universidad Tecnológica Nacional. (s.f.). *Trabajo modelo - Análisis de algoritmos*.
- Universidad Tecnológica Nacional. (s.f.). *Notación Big-O*.
- Universidad Tecnológica Nacional. (s.f.). *Análisis de Algoritmo Teórico y Big O*.
- Universidad Tecnológica Nacional. (s.f.). *Análisis de Algoritmo Empírico*.
- Universidad Tecnológica Nacional. (s.f.). *Introducción al Análisis de Algoritmos*.
- Python Software Foundation. (s.f.). *Documentación oficial Python time*. Disponible en: https://docs.python.org/3/library/time.html

8. Anexos

- Enlace al video de youtube con la explicación:

  https://www.youtube.com/watch?v=4aoUsgRpK_I

- Enlace al repositorio de GitHub:
  https://github.com/FabyRossetto/tp_integrador_programacion