Análisis de la eficiencia algorítmica en Python

Programación I
Profesora: Julieta Trapé

Alumnos:
Matias Orellana (matias.orellan@gmail.com)
Nicolas Pagola (nicolaspagola@outlook.com)

9 de junio 2025

Introducción

En el presente trabajo práctico se analiza la eficiencia de distintos enfoques algorítmicos para resolver un mismo problema: la suma de los elementos de una lista. Con este objetivo, se implementaron dos algoritmos en Python: uno basado en recursión y otro mediante un bucle iterativo. A partir de la generación de listas con valores aleatorios, se evaluó el desempeño de ambas soluciones utilizando mediciones de tiempo de ejecución. Los resultados obtenidos fueron comparados mediante representaciones gráficas que permiten visualizar el comportamiento de cada algoritmo a medida que aumenta el tamaño de la entrada. Este análisis se complementa con una evaluación teórica del orden de complejidad de cada implementación, lo cual permite anticipar y explicar las diferencias observadas en la práctica.

El trabajo busca reflexionar sobre la importancia del análisis asintótico como herramienta para seleccionar algoritmos eficientes, especialmente en contextos donde la optimización de recursos resulta crítica.

Marco Teórico

Un algoritmo es un conjunto de instrucciones precisas y ordenadas que permiten resolver un problema o realizar una tarea específica. Estas instrucciones deben ser claras, finitas y ejecutables, de modo que al seguirlas paso a paso se obtenga un resultado determinado. Los algoritmos se utilizan en diversas áreas, especialmente en programación, donde sirven como base para el desarrollo de soluciones informáticas.

El análisis de algoritmos tiene como objetivo entender qué tan eficiente es un algoritmo. Esto, en el ámbito de programación, se consigue al calcular cuánto tarda en ejecutarse y cuánta memoria utiliza. Resulta importante porque a veces una solución es eficiente si la muestra de datos es chica, pero se vuelve lenta cuando los datos crecen.

Hay dos cosas clave que se analizan:

- Eficiencia temporal: cuánto tiempo tarda en ejecutarse un algoritmo.
- Eficiencia espacial: cuánta memoria extra necesita para funcionar.

La forma más rudimentaria de comparar la eficiencia es medir el tiempo de ejecución del algoritmo también llamado análisis empírico, en Python podemos usar la librería time para determinar el tiempo de ejecución de un algoritmo específico. Esta medición permite comparar dos algoritmos siempre que se ejecuten en el mismo lenguaje, en el mismo hardware y con set de datos similares, al menos. Es un método simple y concreto para determinar que algoritmo se ajusta mejor a las necesidades.

El problema que presenta es el gran inconveniente de someter los algoritmos a grandes cantidades de datos por el tiempo y los recursos que consumiría. Para sortear este inconveniente utilizamos el análisis asintótico.

Análisis Teórico

El análisis teórico de un algoritmo es el estudio de la función que representa ese algoritmo. Para esto partimos del cálculo de la función temporal de un algoritmo que representa el número de operaciones que realiza un algoritmo para una entrada de tamaño n.

La función temporal se determina asignando un número de operaciones primitivas al algoritmo siguiendo parámetros como por ejemplo:

- Asignar valor a una variable: x=2
- Indexar un elemento en un *array*: vector [3]
- Devolver un valor en una función: *return* x
- Evaluar una expresión aritmética: x+3
- Evaluar una expresión lógica: 0<i

Una vez identificada la función temporal, podemos pasar al análisis asintótico del mismo. Para esto hay que tener en cuenta que este tipo de análisis se realiza de manera comparativa, es decir, parten de la comparación de dos funciones. Esto se verá en la expresión que represente las notaciones.

El análisis asintótico refiere al estudio del comportamiento de un algoritmo cuando su entrada de datos tiende al infinito dejando de lado el tiempo exacto para la muestra y poniendo énfasis en la forma en la que el tiempo de ejecución crece a medida que aumenta el tamaño de la muestra. Es decir, es un análisis más abstracto sobre la

eficiencia de un algoritmo en función del número de operaciones, que se relaciona con el tiempo de ejecución.

Existen tres tipos de notaciones para representar este análisis: Big-O, Theta y Omega.

La notación **Big-O** permite expresar cómo crece el tiempo o uso de memoria de un algoritmo en el peor caso. Formalmente, se dice que f(n) pertenece a O(g(n)) si existen constantes positivas c y n_0 tales que para todo $n \geq n_0$, se cumple que $f(n) \leq c \cdot g(n)$. En términos simples, esto significa que f(n) no crece más rápido que g(n) a partir de cierto punto. Por ejemplo, si un algoritmo tiene f(n) = 3n + 2, se lo considera O(n) porque su crecimiento es lineal.

La notación **Theta (Θ)** representa la cota ajustada. Describe el crecimiento exacto de un algoritmo cuando su tiempo de ejecución está acotado tanto por arriba como por abajo por una misma función. Se utiliza para expresar el **caso promedio** o cuando el comportamiento del algoritmo es consistente en todos los casos. Formalmente, f(n) = Θ(g(n)) si existen constantes positivas c_1, c_2 y n_0 tales que $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ para todo $n \geq n_0$. Por ejemplo, si un algoritmo es Θ(n log n), su tiempo de ejecución crecerá proporcionalmente a n log n de manera constante.

La notación **Omega (Ω)** indica la cota inferior del tiempo de ejecución, es decir, el mejor caso: cuánto tiempo como mínimo tomará el algoritmo. Se dice que f(n) = Ω(g(n)) si existen constantes positivas c y n_0 tales que $f(n) \geq c \cdot g(n)$ para todo $n \geq n_0$. Por ejemplo, un algoritmo con Ω(n) garantiza que, al menos, necesitará tiempo lineal en el mejor de los casos.

En resumen, Big-O describe el límite superior o peor caso, Theta representa el comportamiento exacto o promedio, y Omega indica el límite inferior o mejor caso. Estas notaciones permiten comparar algoritmos de forma abstracta, sin depender del hardware o del lenguaje de programación utilizado.

Caso Práctico

Para llevar a cabo el caso práctico se generaron varias listas con números aleatorios utilizando la función *randint*. Luego, se realizó la suma de los elementos mediante dos algoritmos distintos: uno de forma recursiva y otro de manera iterativa mediante un bucle. Durante la ejecución de ambas funciones se midió el tiempo de procesamiento utilizando la librería *time*, y los resultados obtenidos fueron graficados con la ayuda de *pandas*, con fines comparativos.

A través del análisis asintótico se puede anticipar que la función suma_loop, que utiliza un bucle for para recorrer la lista y sumar sus elementos, resulta más eficiente, ya que su tiempo de ejecución crece linealmente con el tamaño de la entrada: Time(suma_loop(lista)) = O(n)

$$\lim_{n \to \infty} \frac{T(n)}{n} = C$$

A su vez tiene un bajo uso de memoria ya que no necesita crear nuevas listas, solo utiliza una variable acumuladora.

En cambio, la función suma_recursiva suma los elementos de forma recursiva, generando una nueva sublista en cada llamada al utilizar el slicing (lista[1:]), lo que conlleva una complejidad cuadrática: Time(suma_recursiva(lista)) = $O(n^2)$

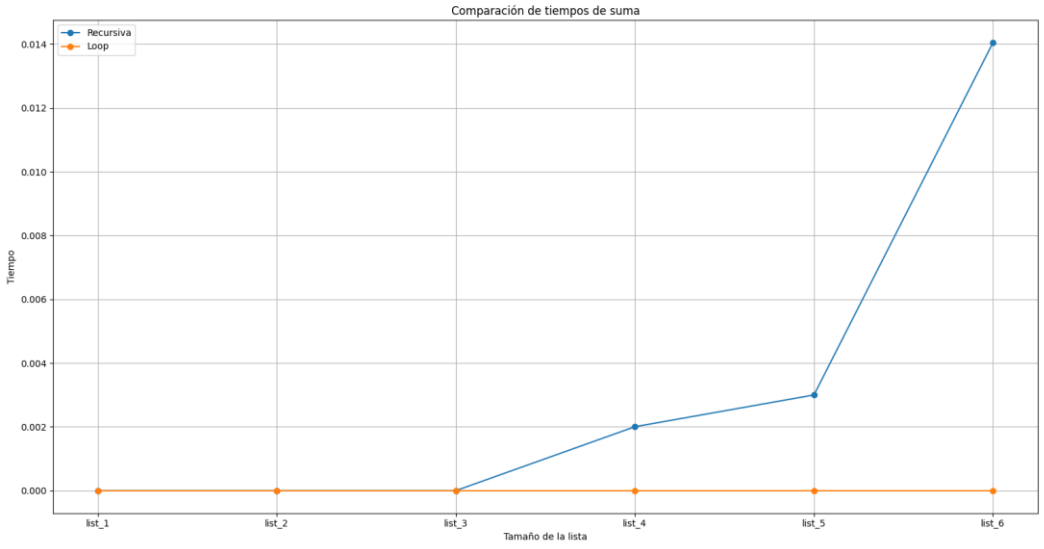
$$\lim_{n \to \infty} \frac{T(n)}{n^2} = C$$

La necesidad de crear una copia parcial de la lista en cada llamada provoca un uso de memoria alto.

Resultados Obtenidos

Se crean 6 listas con números al azar para luego realizar la sumatoria de ellos mediante ambos métodos, aquí los resultados:

	Lista	Recursiva	Loop
0	list_1	0.000000	0.0
1	list_2	0.000000	0.0
2	list_3	0.000000	0.0
3	list_4	0.002004	0.0
4	list_5	0.003000	0.0
5	list_6	0.014052	0.0



Conclusiones

Tal como se observa en el gráfico, el método Loop efectúa un tiempo de ejecución constante sin importar el tamaño de la muestra, a diferencia de la función Recursiva que presenta un formato logarítmico a medida que crece el tamaño de la muestra, volviéndose menos eficiente.

En términos de eficiencia espacial también resulta más eficiente el algoritmo mediante bucle ya que no necesita crear una nueva lista de elementos cada vez que se ejecuta.

Bibliografía

- **Geeks for Geeks** – Explicaciones simples y ejemplos en Python
  https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/
- **Notación Big O | Análisis de algoritmos de forma sencilla**
  https://youtu.be/MyAiCtuhiqQ
- **Programiz** – Buen resumen de notación Big-O
  https://www.programiz.com/dsa/asymptotic-notations
- **Python Docs – módulo time**
  https://docs.python.org/3/library/time.html
- **Wikipedia – Análisis de algoritmos**
  https://es.wikipedia.org/wiki/An%C3%A1lisis_de_algoritmos