



Análisis de Algoritmos

Alumnos:

- Agustina Grille Comisión 15
- Cristian Borda Comisión 10

Materia: Programación I

Profesor/a:

- Ariel Enferrel
- Cinthia Rigoni

Tutores:

- Maximiliano Sar Fernández
- Brian Lara

Fecha de entrega: 09/06/2025

Índice:

- 2. Introducción
- 3. Marco Teórico
- 5. Caso Práctico
- 7. Metodología Utilizada
- 8. Resultados Obtenidos
- 9. Conclusiones
- 10. Bibliografía

Introducción:

El presente trabajo aborda el análisis de algoritmos como herramienta fundamental en el desarrollo de programas eficientes y escalables. Se eligió este tema debido a que representa un aspecto esencial pero muchas veces subestimado de la programación: no solo es importante que un algoritmo funcione correctamente, sino también que lo haga de forma óptima frente al crecimiento de los datos. El caso práctico seleccionado; la comparación entre una suma iterativa, una versión recursiva y una solución mediante fórmula matemática. Permite evidenciar con claridad cómo distintas implementaciones para un mismo problema pueden presentar rendimientos muy diferentes.

Dentro del ámbito de la programación, la eficiencia algorítmica cobra especial relevancia al trabajar con estructuras repetitivas o grandes volúmenes de información. En este sentido, la recursividad se introduce como una técnica que, aunque elegante y expresiva, puede no siempre ser la opción más eficiente dependiendo del problema y del contexto en que se aplique.

El objetivo principal de este trabajo es aplicar herramientas teóricas y prácticas para analizar, comparar y evaluar el rendimiento de tres enfoques distintos para resolver una misma tarea. A través de la implementación en Python, la medición de tiempos reales y el uso de conceptos como la notación Big-O, se busca fomentar el pensamiento crítico en torno a la eficiencia algorítmica y la toma de decisiones informadas en el desarrollo de software.

Marco Teórico:

Un algoritmo es una secuencia finita y ordenada de pasos que resuelve un problema o realiza una tarea. En el contexto de la programación, los algoritmos son la base de toda solución computacional: indican cómo transformar una entrada (input) en un resultado esperado (output), utilizando instrucciones claras, precisas y ejecutables.

Cuando evaluamos un algoritmo, no alcanza con saber si funciona correctamente. También es fundamental analizar su eficiencia, es decir, cómo se comporta su rendimiento a medida que crece la cantidad de datos procesados. Esta eficiencia puede medirse desde dos perspectivas clave:

- Eficiencia temporal: el tiempo que tarda el algoritmo en ejecutarse.
- Eficiencia espacial: la cantidad de memoria que utiliza durante su ejecución.

Ambas medidas permiten tomar decisiones fundamentadas al momento de elegir o diseñar un algoritmo para un problema determinado.

Complejidad algorítmica

La complejidad de un algoritmo se refiere a la cantidad de recursos que necesita en función del tamaño de la entrada, generalmente representado como “n”. Esta complejidad puede expresarse mediante fórmulas matemáticas, pero en programación se utiliza habitualmente la notación Big-O (O grande).

Notación Big-O

La notación Big-O describe el crecimiento del tiempo de ejecución (o del uso de memoria) en el peor de los casos, dejando de lado constantes o factores secundarios. Algunas de las complejidades más comunes son:

- O(1): Tiempo constante. El tiempo de ejecución no cambia, sin importar el tamaño de la entrada.
- O(n): Tiempo lineal. El tiempo crece proporcionalmente al tamaño de la entrada.
- O(n^2): Tiempo cuadrático. El tiempo crece exponencialmente al aumentar la entrada.
- O(log n): Tiempo logarítmico. Crece más lento que lineal, como en la búsqueda binaria.
- O(2^n): Tiempo exponencial. Aumenta de forma extremadamente rápida, como en algunas soluciones recursivas sin optimización.

Ejemplo aplicado al trabajo

En este trabajo Implementamos tres formas distintas de calcular la suma de los primeros n números naturales. Cada una tiene su propia complejidad:

- Suma iterativa (usando un bucle): $O(n)$. Recorre todos los números del 1 al n , sumando uno por uno.
- Suma recursiva: $O(n)$. Llama a sí misma n veces, acumulando resultados en la pila de llamadas.
- Suma con fórmula matemática: $O(1)$. Aplica una expresión cerrada que se resuelve en un solo paso, sin importar el valor de n .

Recursividad

La recursividad es una técnica de programación donde una función se llama a sí misma para resolver subproblemas más pequeños. Es especialmente útil para resolver problemas de estructura repetitiva o que se pueden dividir en partes similares (como los algoritmos de búsqueda, árboles, y cálculo de factoriales).

Sin embargo, la recursividad mal utilizada puede ser ineficiente y consumir mucha memoria, ya que cada llamada ocupa espacio en la pila. En el caso de la suma recursiva, se realizan n llamadas anidadas, lo que implica un costo tanto en tiempo como en espacio similar al enfoque iterativo, pero con mayor carga en la pila de ejecución.

Comparación entre enfoques

- La versión recursiva es útil para fines didácticos y estructurales, pero no resulta la más eficiente en este caso.
- La versión iterativa es simple y clara, adecuada para la mayoría de los lenguajes.
- La versión con fórmula es la más óptima en todos los sentidos, ya que su complejidad temporal es constante.

Conclusión del marco teórico

Conocer la complejidad de un algoritmo permite al programador anticipar su comportamiento antes de ejecutarlo. Utilizar herramientas como la notación Big-O, analizar las estructuras utilizadas (bucles, recursión, expresiones cerradas) y medir el rendimiento real son prácticas fundamentales para escribir código eficiente, escalable y mantenable.

Caso Practico:

El problema propuesto consiste en calcular la suma de los primeros n números naturales. Si bien existen diversas formas de resolver esta tarea, nuestro objetivo con este trabajo fue implementar y comparar tres enfoques distintos: uno iterativo (usando un bucle for), otro recursivo (usando llamadas anidadas a una misma función) y uno basado en una fórmula matemática directa (fórmula de Gauss). A través de estas tres versiones, buscamos analizar la eficiencia temporal de cada estrategia y reflexionar sobre sus ventajas y desventajas, tanto desde la teoría como desde la ejecución práctica del código.

Desarrollamos el programa en cuatro módulos para mayor prolijidad y entendimiento del código. Lo que nos permite mantener un orden y evita errores.

algoritmos.py: contiene las tres implementaciones del cálculo de suma.

```
analysis.py main.py algoritmos.py
algoritmos.py > ...
1 def suma_iterativa(n): ##Primer algoritmo que analizaremos
2     total = 0
3     for i in range(1, n + 1): ##Iteraremos la suma con un ciclo for
4         total += i
5     return total ##Retornamos el total una vez que se termina de recorrer el numero
6
7 def suma_formula(n): ##Segundo algoritmo a analizar
8     return n * (n + 1) // 2 ##Retornamos directamente el producto de la formula de Gauss para la suma de n
9
10 def suma_recursiva(n): ##Tercer algoritmo a analizar
11     if n == 0: ##Valido si n es 0
12         return 0 ##Retorno 0, cierro el llamado recursivo
13     return n + suma_recursiva(n - 1) ##Aplico recursividad
```

tiempo.py: permite medir el tiempo de ejecución de cada función.

```
tiempo.py > ...
1 import time
2
3 def medir_tiempo(funcion, n): ##Defino metodo de medición de tiempos de ejecución
4     inicio = time.time()
5     resultado = funcion(n)
6     fin = time.time()
7     return fin - inicio, resultado #Retorno medición analizada para inicio y final + resultado
```

analysis.py: contiene una función para comparar los resultados.

```

❸ analysis.py > ...
1 def comparar_algoritmos(tiempo_iterativa, tiempo_formula, tiempo_recurativa):#Se incia la funcion de comparacion de los algoritmos.
2     tiempos = {
3         "Iterativa": tiempo_iterativa,
4         "Fórmula": tiempo_formula,
5         "Recursiva": tiempo_recurativa
6     }
7     mejor = min(tiempos, key=tiempos.get)
8     print(f"El algoritmo más eficiente fue: {mejor}") #Se imprime el algoritmo de mayor eficiencia.
9     for nombre, tiempo in tiempos.items(): #Inicio del ciclo for
10        print(f"- {nombre}: {tiempo:.6f} segundos")# Se imprime el tiempo de ejecucion de los algoritmos.
11

```

main.py.: ejecuta el programa y muestra los resultados.

```

1 from algoritmos import suma_iterativa, suma_formula, suma_recurativa #Se importan las funciones de los otros modulos.
2 from tiempo import medir_tiempo
3 from analisis import comparar_algoritmos
4
5 def main():#Nuestra función principal
6     n = 10000 #Inicializo variable con valor 10000
7
8     print(f"Calculando la suma de los primeros {n} números...\n")
9
10    tiempo_iterativa, resultado_iterativa = medir_tiempo(suma_iterativa, n)#Ejecutamos la función suma_iterativa con n, medimos el tiempo que tarda en ejecutarse,
11    #y guardamos el resultado de la suma y el tiempo en dos variables
12    print(f"Suma iterativa: Resultado = {resultado_iterativa}, Tiempo = {tiempo_iterativa:.6f} segundos")
13
14    tiempo_formula, resultado_formula = medir_tiempo(suma_formula, n)#Medimos cuánto tarda suma_formula(n) y guardamos el tiempo y el resultado
15    print(f"Suma con fórmula: Resultado = {resultado_formula}, Tiempo = {tiempo_formula:.6f} segundos")
16
17    tiempo_recurativa, resultado_recurativa = medir_tiempo(suma_recurativa, n)#Medimos cuánto tarda suma_recurativa(n) y guardamos el tiempo y el resultado
18    print(f"Suma recursiva: Resultado = {resultado_recurativa}, Tiempo = {tiempo_recurativa:.6f} segundos")
19
20    print("\nAnálisis de eficiencia:")
21    comparar_algoritmos(tiempo_iterativa, tiempo_formula, tiempo_recurativa)#Comparamos los resultados de los tiempos de cada algoritmos
22
23 if __name__ == "__main__":
24     main()

```

Se optó por desarrollar tres enfoques distintos con el propósito de observar cómo varía la eficiencia algorítmica de una misma operación. La fórmula matemática fue incluida como ejemplo de algoritmo de tiempo constante ($O(1)$), mientras que la iteración y la recursión representan soluciones de complejidad lineal ($O(n)$). Esta comparación es didáctica y permite comprender las implicancias de diseño a nivel de rendimiento.

Metodología Utilizada:

Para el desarrollo de este trabajo integrador se siguió una metodología basada en la investigación teórica, la implementación práctica en Python y el análisis empírico de resultados.

Se consultaron fuentes bibliográficas y materiales provistos por la cátedra para comprender los conceptos fundamentales del análisis de algoritmos, como la eficiencia temporal y espacial, la notación Big-O y las características de la recursividad.

El problema fue abordado desde tres enfoques diferentes (iterativo, recursivo y fórmula matemática), y cada algoritmo fue programado en Python de forma modular. El código se dividió en cuatro archivos para mantener una estructura clara y reutilizable:

- `algoritmos.py`: contiene las tres implementaciones de suma.
- `tiempo.py`: mide el tiempo de ejecución con la librería `time`.
- `analisis.py`: realiza el análisis comparativo de los tiempos.
- `main.py`: integra todos los módulos y ejecuta el programa completo.

Se utilizaron las siguientes herramientas y recursos para realizar el código y el proyecto:

- Lenguaje: Python 3.11.9
- Librería estándar: `time` (para medir tiempos de ejecución)
- Entorno de desarrollo: Visual Studio Code.
- Control de versiones: Git y GitHub
- Formatos de entrega: Link al repositorio de GitHub en el cual cargamos los archivos .Py del programa, el PDF del trabajo, la presentación en PowerPoint (PPT) y el link del video en el archivo `raedme`.

El trabajo fue realizado en conjunto por los dos integrantes, Agustina grille y Cristian Alexis Borda. Se estableció una dinámica de colaboración continua a través de GitHub, subiendo y revisando el código de forma conjunta. El desarrollo de los archivos PDF y PPT se dividió entre ambos participantes, coordinando tanto los textos como la estética y presentación del material. El enfoque colaborativo permitió distribuir tareas, revisar el trabajo mutuamente y asegurar la coherencia en los entregables finales.

Resultados obtenidos:

Se realizaron múltiples pruebas con distintos valores de entrada. En todos los casos, los tres algoritmos devolvieron resultados numéricamente correctos. Sin embargo, se observaron diferencias significativas en los tiempos de ejecución:

La versión iterativa mostró un buen rendimiento general mientras que la fórmula matemática fue la más eficiente, como se esperaba.

```
PS C:\Users\Agus\Desktop\UTN\Programación 1\ProyectoIntegrador\trabajo-practico-integrador-programación-i> & C:/Users/Agus/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/Agus/Desktop/UTN/Programación 1/ProyectoIntegrador/main.py"
Calculando la suma de los primeros 900 números...

Suma iterativa: Resultado = 405450, Tiempo = 0.000036 segundos
Suma con fórmula: Resultado = 405450, Tiempo = 0.000002 segundos
Suma recursiva: Resultado = 405450, Tiempo = 0.000134 segundos

Análisis de eficiencia:
El algoritmo más eficiente fue: Fórmula
- Iterativa: 0.000036 segundos
- Fórmula: 0.000002 segundos
- Recursiva: 0.000134 segundos
PS C:\Users\Agus\Desktop\UTN\Programación 1\ProyectoIntegrador\trabajo-practico-integrador-programación-i> []
```

La versión recursiva fue funcional para valores moderados de n (hasta 9000 aprox.), pero a partir de cierto punto generó un `RecursionError` debido al límite de profundidad del intérprete de Python. Por esta razón, en pruebas más grandes, se redujo el valor de n para la función recursiva.

El proyecto completo, con el código fuente organizado en módulos, está disponible en:

<https://github.com/agustinagrille/trabajo-practico-integrador-programacion-i>

Conclusión:

El desarrollo de este trabajo nos permitió comprender en profundidad el valor del análisis de algoritmos dentro del proceso de programación. Más allá de que todos los algoritmos que implementamos, resolvían correctamente el mismo problema (sumar los primeros n números naturales), se observó que su eficiencia y comportamiento eran muy diferentes según la técnica utilizada: iteración, recursividad o fórmula matemática.

El mayor aprendizaje que conseguimos fue reconocer que, si bien la recursividad es una herramienta poderosa y elegante, su aplicación debe evaluarse con cuidado, especialmente en lenguajes como Python donde existen límites en la profundidad de llamadas recursivas. Por el contrario, las soluciones iterativas ofrecen una alternativa más predecible en cuanto a consumo de recursos, y las soluciones matemáticas pueden ser insuperables en términos de rendimiento cuando son posibles.

Este trabajo también sirvió para aplicar conceptos clave de la teoría de algoritmos como la notación Big-O, la eficiencia temporal y espacial, y la medición empírica de tiempos de ejecución. Estas herramientas resultan sumamente útiles tanto en programación general como en situaciones reales que requieren procesamiento de grandes volúmenes de datos.

Como mejora futura, nos interesaría extender el análisis a otros algoritmos con distintas complejidades (por ejemplo, ordenamientos o búsquedas), o implementar versiones optimizadas con técnicas como memorización o programación dinámica. También podría explorarse el uso de herramientas gráficas para representar visualmente el crecimiento de la complejidad algorítmica.

Durante el trabajo surgieron algunas dificultades relacionadas con la recursión: al probar con valores grandes de n, Python lanzó un RecursionError debido al límite predeterminado de profundidad. Esta situación se resolvió ajustando el valor de entrada para la función recursiva y analizando el error como parte del aprendizaje. Esta experiencia reforzó la importancia de comprender las limitaciones técnicas del lenguaje y de validar siempre el funcionamiento real del código más allá de la teoría.

Bibliografía:

UTN - Cátedra de Programación I. (2025). Análisis Teórico de Algoritmos. Apuntes de clase en PDF.

UTN - Cátedra de Programación I. (2025). Análisis de algoritmo Teórico y Big O. Material complementario.

Universidad Tecnológica Nacional. (2025). Guía completa sobre recursividad en Python.

Cimino, C. (2024, enero 22). RECURSIVIDAD en PROGRAMACIÓN – La explicación definitiva YouTube. <https://www.youtube.com/watch?v=Sh1xrDAOPyY>

Cimino, C. (2024, febrero 2). RECURSIVIDAD SIMPLE – Ejemplo en Python YouTube. <https://www.youtube.com/watch?v=pLUp8t1e97U>

Python Software Foundation. (2024). time — Time access and conversions. En Python 3 documentation.

<https://docs.python.org/3/library/time.html>

Yang, C. (2024). Big O Cheat Sheet.

<https://www.bigcheatsheet.com>