Tecnicatura Universitaria en Programación

Programación I

Porf. Nicolas Quirós

“Análisis de Algoritmos”

Alumnos:

Agustin Emiliano Sotelo Carmelich (agustinemiliano22@gmail.com)

Bruno Giuliano Vapore (brunogvapore@gmail.com)

Fecha de Entrega: 9 de Junio de 2025

Índice

1. Introducción

En este trabajo integrador nos proponemos explorar el análisis de algoritmos, un pilar fundamental dentro del estudio de la programación. Este tema resulta especialmente relevante porque nos permite comprender cómo la forma en que diseñamos e implementamos nuestros algoritmos puede afectar directamente su eficiencia y rendimiento, especialmente cuando se enfrentan a grandes volúmenes de datos.

El objetivo principal de este trabajo es comparar dos enfoques distintos para resolver un mismo problema: el cálculo del promedio de los números positivos en una lista. Para ello, implementamos dos versiones del algoritmo: una iterativa, basada en bucles, y otra recursiva, en la que la función se llama a sí misma para recorrer la lista. A través de esta comparación buscamos no solo validar que ambas soluciones son funcionales, sino también evaluar cuál ofrece un mejor rendimiento y en qué contextos resulta más adecuada.

2. Marco Teórico

El Análisis de Algoritmos, según Brassard y Bratley (1997), puede definirse como una herramienta esencial que se utiliza para determinar cuál resulta ser el algoritmo más adecuado entre varios algoritmos distintos para resolver el mismo problema. Los autores mencionan que "*Sólo después de haber determinado la eficiencia de los distintos algoritmos será posible tomar una decisión bien informada. (…) En su mayor parte es una cuestión de juicio, intuición y experiencia*" (p. 111). No obstante, existen técnicas que vamos a poder aplicar para evaluar la eficiencia del algoritmo.

La eficiencia del algoritmo, como una de sus principales características, determina que los recursos deban ser utilizados en forma óptima, en relación al tiempo de ejecución y el espacio de memoria utilizado. Entonces, podemos definir el análisis de algoritmos como una habilidad y tarea que deben desarrollar los programadores para evaluar y comparar algoritmos, para el perfeccionamiento del programa desarrollado en un contexto determinado.

Para poder abordar el análisis y la comparación de algoritmos en este trabajo, consideramos necesario repasar algunos conceptos clave trabajados a lo largo de la materia:

- Algoritmo: Es un conjunto finito y ordenado de instrucciones que permite resolver un problema computacional o realizar una tarea específica. Puede pensarse como una "receta" que indica paso a paso lo que debe hacer la computadora.
- Estructuras secuenciales (Módulo 1): Son aquellas instrucciones que se ejecutan una tras otra, en el mismo orden en que aparecen en el código, sin ningún tipo de bifurcación o repetición.
- Estructuras condicionales (Módulo 3): Permiten tomar decisiones dentro del programa. A través de expresiones como if–else, se puede ejecutar un bloque de código u otro, dependiendo de si se cumple una condición.

- Estructuras repetitivas (Módulo 4): Conocidas también como bucles, nos permiten repetir un bloque de código múltiples veces. Los más comunes son for y while. Son fundamentales para procesar datos en secuencias, como listas o rangos.

- Recursividad (Módulo 6): Es una técnica en la que una función se llama a sí misma para resolver un problema. Para que esta técnica funcione correctamente y no produzca un bucle infinito, es necesario definir un caso base que detenga la recursión. La recursividad es especialmente útil en problemas que tienen una estructura naturalmente recursiva, como el cálculo de factoriales o la búsqueda en estructuras jerárquicas.

Para llevar a cabo el análisis de algoritmos, se puede optar por realizar un análisis empírico o un análisis teórico. El primero, se hace mediante la ejecución de varias pruebas con diferentes tamaños de entrada, registrando los tiempos de cada ejecución. Para ello, se utilizan en el algoritmo algunas variables que ayudan a medir el tiempo. Estos datos son recopilados y son volcados posteriormente en un gráfico para un análisis visual de los resultados.

La desventaja de este tipo de análisis es que depende de factores que inciden en los tiempos de ejecución. Por ejemplo, el hardware donde se ejecuta y la eficiencia del compilador. También requiere implementación.

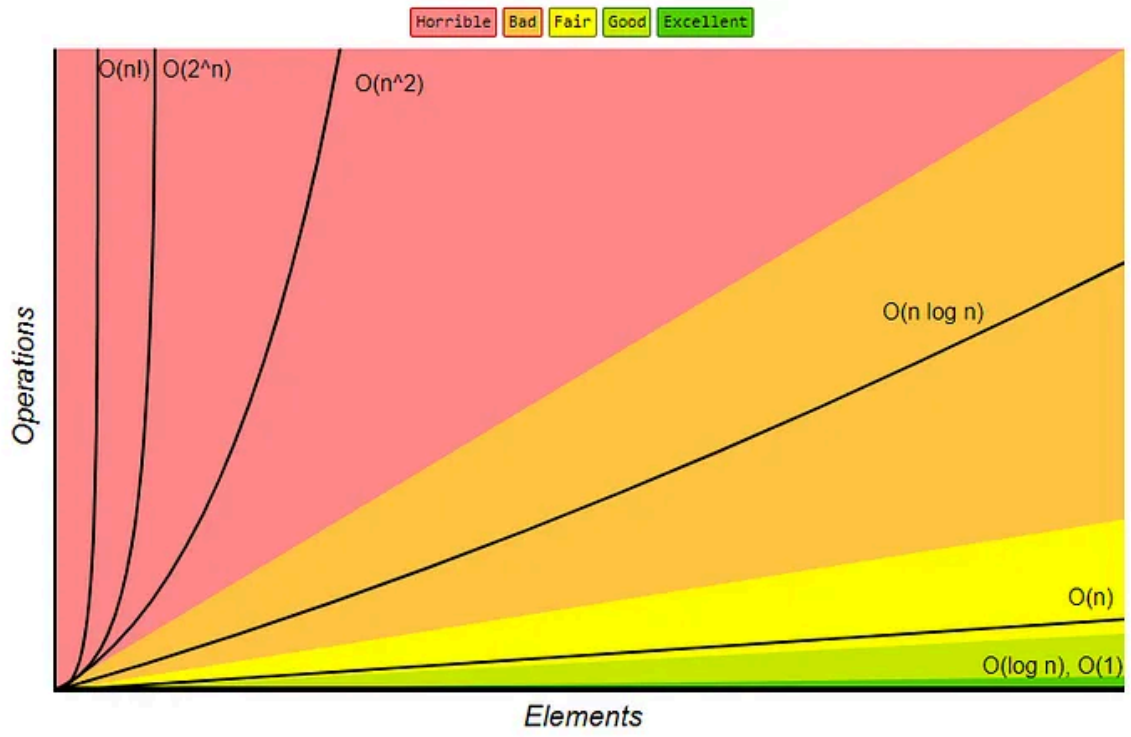
Para el análisis teórico, en cambio, se obtiene una función matemática T(n) que describe el número de operaciones que realiza un algoritmo en función del tamaño de entrada n, permitiendo un panorama general de la eficiencia del algoritmo. Esto permite identificar la complejidad computacional y predecir el rendimiento del algoritmo sin ejecutar el código ni la necesidad de implementar el código.

Para el cálculo de T(n), cada operación básica se cuenta como una unidad de tiempo. Por ejemplo, asignación de valores; acceso al elemento en una lista; evaluación de expresión aritmética o lógica; y devolución de un valor en una función.

Finalmente, podemos realizar un análisis asintótico mediante la notación Big O. La notación Big O nos permite describir cómo se comporta un algoritmo a medida que crece el tamaño de la entrada. Este método facilita la comparación de las funciones temporales T(n), aproximando cada función que necesitemos comparar a una cota superior o límite superior (el Big O).

Por ejemplo, un algoritmo con complejidad O(n) presenta un crecimiento lineal del tiempo de ejecución respecto al tamaño de la entrada. Esta herramienta es fundamental para comparar diferentes enfoques y tomar decisiones informadas sobre cuál conviene implementar en cada caso.

Entonces, la notación Big O describe el peor caso del crecimiento de un algoritmo en función del tamaño de entrada. Cuanto mayor sea la complejidad de la ordenada superior, menor es la eficiencia del algoritmo. El orden establecido para la notación Big O, de menor a mayor, es el siguiente: constante O(1); logarítmica O (log n); lineal O(n); lineal-logarítmica O(n log n), cuadrática O(n²), exponencial O(2^n) y factorial O(n!). El gráfico recuperado de Ferguson (2022) ilustra lo mencionado:



Por otro lado, el Big Omega (Ω) describe el mejor caso, estableciendo una cota o límite inferior, mientras que el Big Theta (Θ) describe el caso promedio, entre ambos límites.

En el lenguaje Python, que se utiliza para la aplicación del caso práctico, existen diferentes herramientas y técnicas para medir los tiempos de ejecución. Por ejemplo time y timeit.

- time.time(): devuelve el tiempo en segundos desde el inició de la época (1 de enero de 1970, 00:00:00). Podemos guardarla en variables para calcular la diferencia del tiempo.
- timeit: *"Este módulo proporciona una forma sencilla de cronometrar pequeños fragmentos de código Python. Tiene tanto una Interfaz de línea de comandos como una invocable. Evita una serie de trampas comunes para medir los tiempos de ejecución."* (Python Software Foundation, s.f.)

Estas herramientas se pueden aplicar en diferentes estructuras de control, donde se realizan diferentes operaciones aritméticas o lógicas, y pudiendo utilizar diferentes estructuras de datos. Por ejemplo listas (estructura ordenada de elementos, con acceso por índices); tuplas (secuencia inmutable de datos); conjuntos o sets (colección de elementos no ordenados y sin elementos duplicados) y diccionarios (asociación de claves únicas con valores), entre otros.

En el caso práctico se especificarán las estructuras y herramientas utilizadas para el cumplimiento del objetivo propuesto y la implementación de la optimización de algoritmos.

3. Caso Práctico

I. Breve descripción del problema a resolver:

Para la aplicación del caso práctico, se decidió analizar algoritmos para resolver el cálculo de promedio de todos los números positivos dentro de una lista dada. Para ello, implementaremos y compararemos dos algoritmos: uno iterativo y otro recursivo. Luego, mediremos su eficiencia (tiempo que tardan en ejecutarse) para listas de diferentes tamaños.

II. Situación concreta:

En el cálculo de promedios de calificaciones estudiantiles, por ejemplo, es habitual encontrarse con valores atípicos o inválidos, como calificaciones negativas. Estos valores pueden originarse por errores en la carga de datos o representar ausencias, frecuentemente codificadas con valores como -1. Para garantizar un análisis estadístico riguroso y representativo, resulta fundamental excluir estas calificaciones no válidas y considerar únicamente aquellos valores que sean positivos y válidos al momento de calcular el promedio.

III. Elección del problema:

Este problema fue de nuestra elección debido a su relevancia en el ámbito del procesamiento y análisis de datos reales. En contextos educativos, como la gestión de calificaciones estudiantiles, es común encontrarse con datos incompletos, erróneos o codificados de forma especial (por ejemplo, ausencias representadas con valores negativos). Abordar esta situación permite trabajar con un caso práctico y cercano, que a su vez pone en evidencia la importancia de realizar una limpieza adecuada de los datos antes de aplicar cualquier método estadístico. Asimismo, resulta relevante para optimizar los códigos que se presenten

y definir el diseño de algoritmos en base a los recursos disponibles y utilizados en diferentes contextos.

IV. Explicación de decisiones de diseño

A. Elección de algoritmos (iterativo y recursivo):

Decidimos implementar y comparar los enfoques iterativo y recursivo porque representan las principales formas de resolver problemas repetitivos en programación. La iteración, a través de bucles como for o while, es un método directo y ampliamente utilizado por su eficiencia. En cambio, la recursión, que consiste en funciones que se llaman a sí mismas, puede ofrecer soluciones más limpias y conceptualmente claras en ciertos casos, con menos código pero más consumo de memoria. Compararlos nos permitió analizar sus ventajas y desventajas, y comprender mejor en qué situaciones conviene aplicar cada uno.

B. Elección de la estructura de datos "lista":

Optamos por utilizar listas porque son una de las estructuras de datos más simples y comunes en Python. Son fáciles de declarar, recorrer y modificar, lo que nos permitió concentrarnos en la lógica del problema sin añadir complejidad. Además, se adaptan muy bien a nuestro caso, ya que necesitábamos trabajar con una lista de calificaciones numéricas.

V. Validación del funcionamiento:

Para garantizar que nuestros algoritmos funcionen correctamente, realizamos dos tipos de validación complementaria: pruebas de funcionalidad y análisis de rendimiento.

A. Pruebas unitarias (verificación de la funcionalidad):

Antes de evaluar el rendimiento de los algoritmos, implementamos pruebas unitarias específicas (asserts) para cada función desarrollada. Estas pruebas consisten en verificar que las funciones produzcan los resultados esperados frente a

distintos casos de entrada: listas con únicamente valores positivos, listas con valores negativos, listas mixtas y listas vacías. La correcta ejecución de todas las pruebas nos permitió confirmar que la lógica implementada en ambas versiones del algoritmo (iterativa y recursiva) era la adecuada. Estos tests pueden ejecutarse desde la terminal al correr directamente los archivos funciones_iterativas.py o funciones_recursivas.py, donde se mostrarán los resultados correspondientes.

B. Análisis de rendimiento (evaluación de la eficiencia):

Una vez validada la funcionalidad, procedimos a medir la eficiencia de cada algoritmo mediante la librería timeit de Python. Esta herramienta nos permitió calcular con precisión el tiempo de ejecución para listas de distintos tamaños. El objetivo fue observar cómo se comportan los algoritmos a medida que aumentaba la cantidad de datos, y si dicho comportamiento coincidía con las expectativas teóricas según la notación Big O. Además, esta etapa nos permitió detectar una limitación conocida de la recursión en Python: el RecursionError, que ocurre al superar el límite de llamadas recursivas permitidas. Los resultados obtenidos de estas mediciones se presentan en forma de tabla en la terminal al ejecutar el archivo prueba_rendimiento_principal.py.

VI. Dificultades presentadas y cómo se corrigieron los errores:

Durante el desarrollo del trabajo, la principal dificultad encontrada estuvo relacionada con el uso de la función time.time() y con el límite de recursión impuesto por el lenguaje Python.

A. Función time

Al utilizar time.time() no nos permitía observar en forma precisa el lapso de tiempo entre las variables utilizadas para medir el inicio y fin de ejecución del algoritmo. La siguiente imagen muestra en qué sección de la función se estableció las variables para luego calcular el tiempo.

```python
def promedio(notas):
    start = time.time()
    suma = 0
    for i in range(len(notas)):
        suma += notas[i]
    promedio = round(suma / len(notas), 2)
    end = time.time()
    return promedio, (end-start) * 1000
```

Corrección aplicada:

Se investigó e implementó al código la función timeit, que permitió determinar el número de veces que el código podía ejecutarse.

B. Límite de recursión

Al ejecutar la función recursiva con listas de gran tamaño, se supera el número máximo de llamadas recursivas permitidas por defecto, lo que generaba un RecursionError (error de recursión).

Corrección aplicada:

Para solucionar este inconveniente, utilizamos la función sys.setrecursionlimit() con el objetivo de aumentar temporalmente el límite máximo de llamadas recursivas. Esta medida nos permitió ejecutar pruebas con listas de mayor tamaño y continuar con el análisis de rendimiento. Sin embargo, también reconocemos que esta solución tiene limitaciones, ya que la recursión, por su propia naturaleza, consume más memoria al utilizar la pila de llamadas. Por lo tanto, en contextos reales con volúmenes de datos muy grandes, la versión iterativa suele ser preferible por su mayor estabilidad y eficiencia.

Diseño de la función iterativa:

```python
# Encuentra el promedio de números positivos usando un bucle (paso a paso).
def calcular_promedio_positivos_iterativo(lista_de_numeros):
    """
    Calcula el promedio de los números positivos en una lista.
    Usa un bucle.

    Argumentos:
        lista_de_numeros (list): Una lista que contiene números (enteros o decimales).

    Devuelve:
        float: El promedio de los números positivos. Si no hay números positivos devuelve 0.
    """
    # Se inicializan las variables
    suma_de_positivos = 0
    cantidad_de_positivos = 0

    # Recorremos cada número
    # Se revisa cada elemento de la lista - uno por uno.
    for numero_actual in lista_de_numeros:
        # Este número es positivo?
        if numero_actual > 0:
            # Si es positivo se suma a la variable
            suma_de_positivos += numero_actual
            # aumenta el contador
            cantidad_de_positivos += 1

    if cantidad_de_positivos > 0:
        # Cálculamos el promedio:
        return suma_de_positivos / cantidad_de_positivos
    else:
        # Si no encontramos ningún número positivo o la lista estaba vacía, el promedio es 0.
        return 0

# Investigamos cómo comprobar si el bucle funciona cómo debería
# Para ver si nuestra función devuelve el resultado esperado utilizamos "assert"
# Lo que hace es probar/depurar el código. Es una verificación de una condición que debería ser verdadera.
# Evalúa la condición: Python mira si la condición es True o False.
# Si es False lanza un error llamado AssertionError.
if __name__ == "__main__":
    print("Pruebas Unitarias")
    # 1 + 2 + 3 + 4 + 5 / 5 = 3
    assert calcular_promedio_positivos_iterativo([1, 2, 3, 4, 5]) == 3.0, "Test 1 Fallido: Lista con solo positivos"
    # Una lista de sólo números negativos el promedio debería ser = 0
    assert calcular_promedio_positivos_iterativo([-1, -2, -3]) == 0, "Test 2 Fallido: Lista con solo negativos"
    #(10+20+30)/3 = 20.0
    assert calcular_promedio_positivos_iterativo([10, -5, 20, 0, 30]) == 20.0, "Test 3 Fallido: Lista mixta"
    # lista vacía el promedio es 0
    assert calcular_promedio_positivos_iterativo([]) == 0, "Test 4 Fallido: Lista vacía"
    # 7/1 = 7.0
    assert calcular_promedio_positivos_iterativo([7]) == 7.0, "Test 5 Fallido: Un solo positivo"
    # número negativo el promedio es = 0
    assert calcular_promedio_positivos_iterativo([-7]) == 0, "Test 6 Fallido: Un solo negativo"
    print("Todas las pruebas para la función pasaron correctamente.")
```

Diseño de la función recursiva:

```python
# Función recursiva para calcular el promedio
import sys
"""
El módulo sys te da acceso a herramientas para interactuar con el Python.

La usamos específicamente para sys.setrecursionlimit(), que nos permite aumentar el número máximo de veces que
una función puede llamarse a sí misma (recursión). Hacemos esto para que nuestras pruebas de rendimiento con listas
muy grandes no fallen por el límite de recursión por defecto de Python.
"""
sys.setrecursionlimit(200000)

def calcular_promedio_positivos_recursivo(lista_de_numeros, suma_acumulada=0, conteo_acumulado=0, indice_actual=0): #
inicializamos las variables
    """
    - Calcula el promedio de los números en una lista
    - Usa un enfoque recursivo (la función se llama a sí misma)

    Argumentos:
        - lista_de_numeros (list): Una lista que contiene números
        - suma_acumulada (float): La suma de positivos hasta ahora (uso interno)
        - conteo_acumulado (int): La cantidad de positivos hasta ahora (uso interno)
        - indice_actual (int): La posición actual que estamos revisando en la lista (uso interno)

    Retorna:
        float: El promedio de los números positivos y si no hay devuelve 0.
    """
    # Condición que detiene la recursión
    # si el indice actual es igual al tamaño de la lista (revisó toda la lista)
    if indice_actual == len(lista_de_numeros):
        if conteo_acumulado > 0:
            return suma_acumulada / conteo_acumulado # se calcula el promedio
        else:
            # devuelve 0 si no hay numeros positivos o si está vaciá
            return 0

    # paso recursivo, se llama la función así misma
    numero_en_posicion_actual = lista_de_numeros[indice_actual] # obtengo el numero en la posicion actual


    if numero_en_posicion_actual > 0: # si es positivo..

        return calcular_promedio_positivos_recursivo(
            lista_de_numeros,
            suma_acumulada + numero_en_posicion_actual,     # se suma el numero actual
            conteo_acumulado + 1,                           # aumento el contador
            indice_actual + 1                               # paso al sgte numero
        )
    else:
        # si es 0 o negativo
        return calcular_promedio_positivos_recursivo(
            lista_de_numeros,
            suma_acumulada,
            conteo_acumulado,
            indice_actual + 1
        )

# chequeamos si nuestro codigo funciona bien
if __name__ == "__main__":
    print("\nTest y pruebas unitarias")
    # numeros positivos (1 + 2 + 3 + 4 + 5) / 5 = 3.0
    assert calcular_promedio_positivos_recursivo([1, 2, 3, 4, 5]) == 3.0, "Test 1 Recursivo Fallido: Lista con solo
positivos"
    # numeros negativos = 0
    assert calcular_promedio_positivos_recursivo([-1, -2, -3]) == 0, "Test 2 Recursivo Fallido: Lista con solo
negativos"
    # (10+20+30)/3 = 20.0
    assert calcular_promedio_positivos_recursivo([10, -5, 20, 0, 30]) == 20.0, "Test 3 Recursivo Fallido: Lista mixta"
    # lista vacía = 0
    assert calcular_promedio_positivos_recursivo([]) == 0, "Test 4 Recursivo Fallido: Lista vacía"
    # 7/1 = 7.0
    assert calcular_promedio_positivos_recursivo([7]) == 7.0, "Test 5 Recursivo Fallido: Un solo positivo"
    # numero negativo = 0
    assert calcular_promedio_positivos_recursivo([-7]) == 0, "Test 6 Recursivo Fallido: Un solo negativo"
    print("Todas las pruebas para la función recursiva pasaron correctamente.")
```

4. Metodología Utilizada

Se consultaron diversas fuentes para consolidar los conocimientos del Análisis de Algoritmos, los tipos de análisis y la notación Big O. En primer lugar, se utilizaron los recursos proporcionados por la cátedra de Programación I de la Tecnicatura Universitaria en Programación. Posteriormente, fue necesario consultar diversas fuentes disponibles en internet para profundizar en el tema y en conceptos relacionados, así como integrar lo visto en módulos anteriores.

Una vez realizado la investigación previa, se seleccionó entre varias opciones el caso práctico en el que podía implementarse el Análisis de Algoritmo, habiendo realizado en lenguaje Python las primeras líneas de código para obtener el promedio de números positivos, así como también la aplicación de la función time.time() para realizar algunas pruebas de su funcionalidad.

En esta etapa se hizo necesario retomar la investigación para resolver los problemas que fueron surgiendo en el desarrollo de los algoritmos (previamente explicados), haciendo uso de las librerías de Python.

Durante la investigación y el desarrollo de algoritmos, la comunicación entre los integrantes del grupo fue constante. Se realizaron reuniones diarias para el avance en la investigación y toma de decisiones. Así como también se utilizó GitHub como repositorio remoto para compartir los cambios efectuados en los archivos Python a través de la herramienta Visual Studio Code.

Una vez implementado el código deseado, se llevó a cabo un análisis teórico y la notación Big O de ambos algoritmos. Por último, se utilizó el módulo de timeit para efectuar el análisis empírico, procediendo a tomar nota de los resultados obtenidos.

5. Resultados Obtenidos

Análisis teórico y Notación Big O:

Función iterativa:

```
def calcular_promedio_positivos_iterativo(lista_de_numeros):
    suma_de_positivos = 0                                   T(n) = 1
    cantidad_de_positivos = 0                               T(n) = 1

    for numero_actual in lista_de_numeros:                  T(n) = 5*n
        if numero_actual > 0:
            suma_de_positivos += numero_actual
            cantidad_de_positivos += 1

    if cantidad_de_positivos > 0:                           T(n) = 3
        return suma_de_positivos / cantidad_de_positivos
    else:
        return 0
```

T(n) = 1 + 1 + 5n + 3 = 5n + 5

Notación Big O: O(n) → es lineal, depende de n, complejidad eficiente.

Función recursiva:

```
def  calcular_promedio_positivos_recursivo(lista_de_numeros,  suma_acumulada=0,
conteo_acumulado=0, indice_actual=0):
    if indice_actual == len(lista_de_numeros):                          T(n) = 4
        if conteo_acumulado > 0:
            return suma_acumulada / conteo_acumulado
        else:
```

```
        return 0

    numero_en_posicion_actual = lista_de_numeros[indice_actual]          T(n) = 1

    if numero_en_posicion_actual > 0:                                    T(n) = 5
        return calcular_promedio_positivos_recursivo(
            lista_de_numeros,
            suma_acumulada + numero_en_posicion_actual,
            conteo_acumulado + 1,
            indice_actual + 1
        )
    else:
        return calcular_promedio_positivos_recursivo(
            lista_de_numeros,
            suma_acumulada,
            conteo_acumulado,
            indice_actual + 1
        )
```

El caso base se ejecuta una sola vez, mientras que el resto del código n veces según la lista de números proporcionada.

Entonces, T(n) = (5 + 1) * n + 4 = 6n + 4.

Notación Big O = O(n) → lineal, depende de n, complejidad eficiente.

No obstante, la pequeña diferencia con la función anterior se debe a que la función recursiva consume más memoria al llamarse a sí misma, aumentando los tiempos de ejecución real.

Análisis Empírico (timeit):

Para el análisis empírico, se elaboró un algoritmo general en el cual se llamó a las dos funciones anteriormente desarrolladas y se utilizó el módulo timeit para registrar los tiempos de ejecución de cada una, conforme se detalla a continuación:

```python
# Archivo "main" que ejecuta las pruebas
import random         # Para generar números aleatorios de forma sencilla.
import timeit         # Para medir el tiempo de ejecución.
import sys            # Para ajustar el límite de recursión si es necesario.

# Importamos nuestras funciones:
from funciones_iterativas import calcular_promedio_positivos_iterativo
from funciones_recursivas import calcular_promedio_positivos_recursivo

# Pruebas - Medición
"""
Ajustamos el límite de recursión de Python. Por defecto, Python detiene la recursión
después de un cierto número de llamadas (normalmente 1000) para evitar que el programa
se quede sin memoria. Lo aumentamos para poder probar con listas más grandes.
Un número demasiado alto puede causar problemas de memoria
"""
sys.setrecursionlimit(200000)

# Definimos los diferentes tamaños de lista para ejecutar nuestras pruebas, queremos notar y ver
# la diferencia de tiempos.
tamanos_de_lista = [100, 1000, 5000, 10000, 50000, 100000]

# inicializamos listas vacías para guardar los tiempos para cada función
tiempos_funcion_iterativa = []
tiempos_funcion_recursiva = []

ejecuciones_por_medicion = 100 # cuantas veces se repetira cada medicion (nos aseguramos de obtener resultados mas precisos)
```

```python

# --- INICIANDO PRUEBAS DE RENDIMIENTO ---

print("\n- INICIANDO PRUEBAS DE RENDIMIENTO -")

for tamano_actual in tamanos_de_lista:
    # Creamos una lista de números aleatorios entre -1 y 10, simulando notas de alumnos
    lista_para_prueba = [random.randint(-1, 10) for _ in range(tamano_actual)]

    print(f"\nProbando con una lista de tamaño: {tamano_actual} elementos.")

    # - Medición de la función iterativa -
    # Preparamos el código que 'timeit' ejecutará.
    configuracion_iterativa = f"""
from funciones_iterativas import calcular_promedio_positivos_iterativo
lista_prueba = {lista_para_prueba}
"""

    # Medimos el tiempo total y calculamos el promedio por ejecución
    tiempo_total_iterativa = timeit.timeit(
        stmt="calcular_promedio_positivos_iterativo(lista_prueba)",
        setup=configuracion_iterativa,
        number=ejecuciones_por_medicion
    )
    tiempo_promedio_iterativa = tiempo_total_iterativa / ejecuciones_por_medicion

    tiempos_funcion_iterativa.append(tiempo_promedio_iterativa)
    print(f"  Tiempo Iterativo (promedio de {ejecuciones_por_medicion} ejecuciones): {tiempo_promedio_iterativa:.6f} segundos")


```

```python
    # - Medición de la función recursiva -
    configuracion_recursiva = f"""
from funciones_recursivas import calcular_promedio_positivos_recursivo
import sys
sys.setrecursionlimit(max(sys.getrecursionlimit(), {tamano_actual + 100}))
lista_prueba = {lista_para_prueba}
"""
    try:
        # Intentamos medir el tiempo. Puede fallar por 'RecursionError'
        tiempo_total_recursiva = timeit.timeit(
            stmt="calcular_promedio_positivos_recursivo(lista_prueba)",
            setup=configuracion_recursiva,
            number=ejecuciones_por_medicion
        )
        tiempo_promedio_recursiva = tiempo_total_recursiva / ejecuciones_por_medicion
        tiempos_funcion_recursiva.append(tiempo_promedio_recursiva)
        print(f"  Tiempo Recursivo (promedio de {ejecuciones_por_medicion} ejecuciones): {tiempo_promedio_recursiva:.6f} segundos")
    except RecursionError:
        # Informamos si hubo un error de recursión
        print(f"  Error de Recursión,lLa lista de tamaño {tamano_actual} es demasiado grande para la recursión.")
        tiempos_funcion_recursiva.append(float('nan'))

print("\n- PRUEBAS DE RENDIMIENTO FINALIZADAS -")

# --- Presentación Final de los Resultados en una Tabla en la Terminal ---
# Esta sección toma todos los tiempos que guardamos y los organiza en una tabla fácil de leer
# que se mostrará directamente en la terminal.
print("\n--- RESUMEN DE TIEMPOS DE EJECUCIÓN ---")

# Imprimimos los encabezados de la tabla.
# '{:<15}' significa: "reserva 15 espacios y alinea el texto a la izquierda".
# Esto ayuda a que las columnas se vean ordenadas.
print("{:<15} {:<25} {:<25}".format("Tamaño Lista", "Tiempo Iterativo (s)", "Tiempo Recursivo (s)"))
# Imprimimos una línea de guiones para separar los encabezados de los resultados.
print("-" * 65)
```

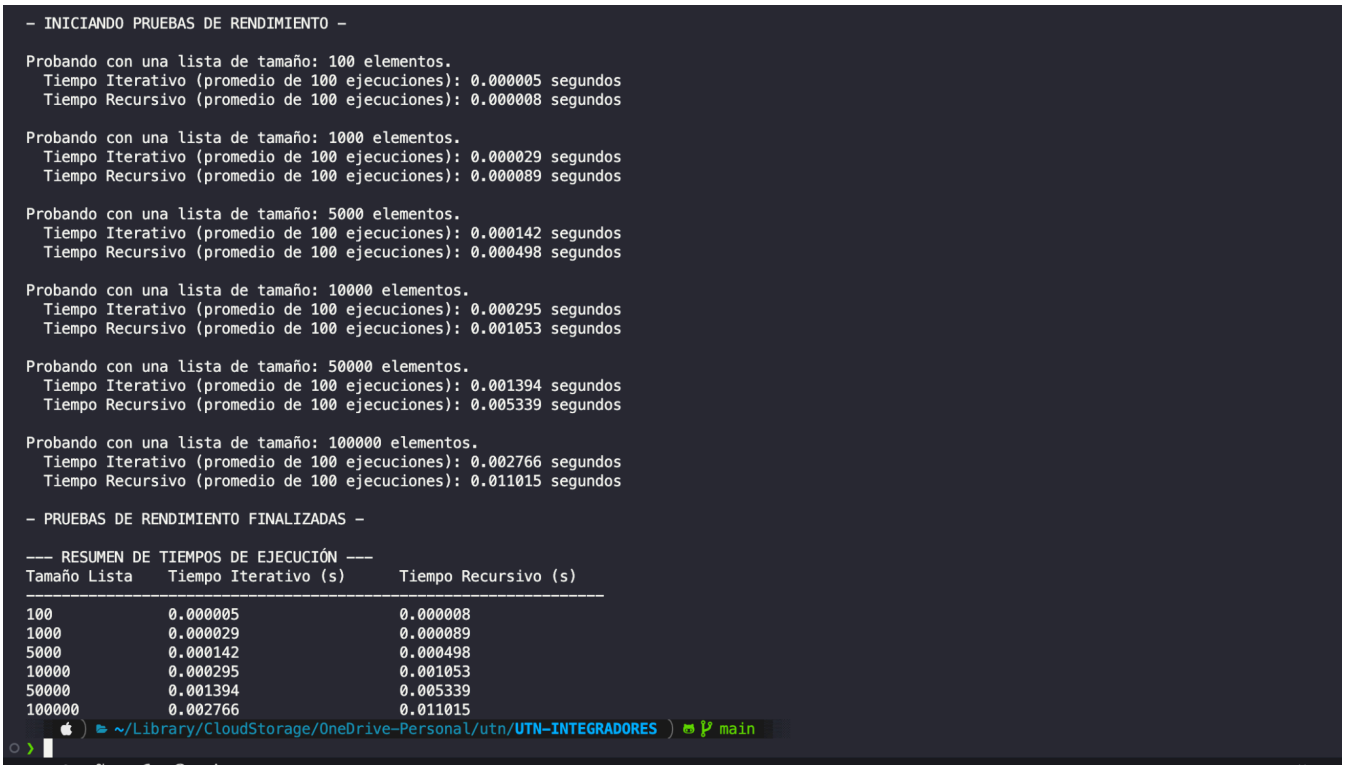
```python

# Recorremos nuestras listas de 'tamanos_de_lista' y los tiempos que guardamos
# 'enumerate()' nos da el índice 'i' y el valor 'tamano' en cada vuelta.
for i, tamano in enumerate(tamanos_de_lista):
    # Obtenemos el tiempo iterativo y lo formateamos a 6 decimales.
    tiempo_iterativo_formateado = f"{tiempos_funcion_iterativa[i]:.6f}"

    # Obtenemos el tiempo recursivo, condición:
    # Si el tiempo es 'nan' = Not a number (que guardamos cuando hubo un error de recursión),
    # mostramos el texto "(Error de Recursión)".
    # Si no es 'nan', lo formateamos} a 6 decimales.
    tiempo_recursivo_formateado = f"{tiempos_funcion_recursiva[i]:.6f}" if not float('nan') == tiempos_funcion_recursiva[i] else "(Error de Recursión)"

    # Imprimimos la fila completa de la tabla, usando el mismo formato para las columnas.
    print(f"{tamano:<15} {tiempo_iterativo_formateado:<25} {tiempo_recursivo_formateado:<25}")
```

Una vez ejecutado el código, pudimos obtener la siguiente salida por consola:



- INICIANDO PRUEBAS DE RENDIMIENTO -

Probando con una lista de tamaño: 100 elementos.

Tiempo Iterativo (promedio de 100 ejecuciones): 0.000005 segundos

Tiempo Recursivo (promedio de 100 ejecuciones): 0.000008 segundos

Probando con una lista de tamaño: 1000 elementos.

Tiempo Iterativo (promedio de 100 ejecuciones): 0.000029 segundos

Tiempo Recursivo (promedio de 100 ejecuciones): 0.000089 segundos

Probando con una lista de tamaño: 5000 elementos.

Tiempo Iterativo (promedio de 100 ejecuciones): 0.000142 segundos

Tiempo Recursivo (promedio de 100 ejecuciones): 0.000498 segundos

Probando con una lista de tamaño: 10000 elementos.

Tiempo Iterativo (promedio de 100 ejecuciones): 0.000295 segundos

Tiempo Recursivo (promedio de 100 ejecuciones): 0.001053 segundos

Probando con una lista de tamaño: 50000 elementos.

Tiempo Iterativo (promedio de 100 ejecuciones): 0.001394 segundos

Tiempo Recursivo (promedio de 100 ejecuciones): 0.005339 segundos

Probando con una lista de tamaño: 100000 elementos.

Tiempo Iterativo (promedio de 100 ejecuciones): 0.002766 segundos

Tiempo Recursivo (promedio de 100 ejecuciones): 0.011015 segundos

- PRUEBAS DE RENDIMIENTO FINALIZADAS -

--- RESUMEN DE TIEMPOS DE EJECUCIÓN ---

Tamaño Lista Tiempo Iterativo (s) Tiempo Recursivo (s)

---

100 0.000005 0.000008

1000 0.000029 0.000089

5000 0.000142 0.000498

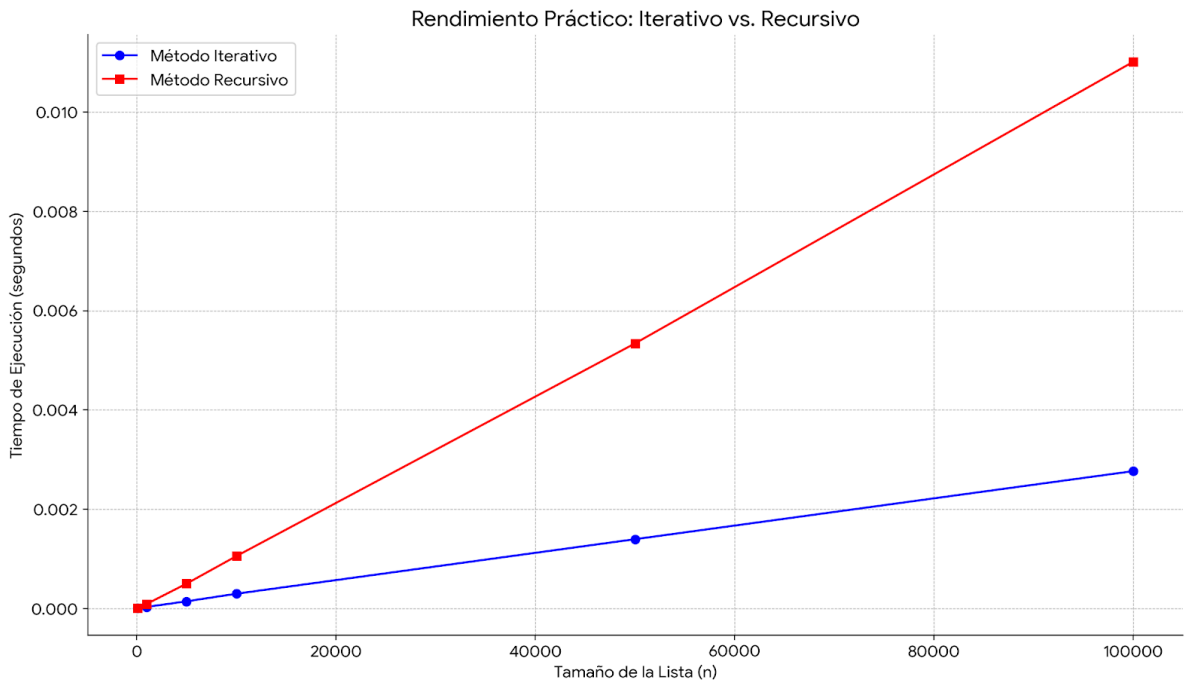
10000 0.000295 0.001053

50000 0.001394 0.005339

100000 0.002766 0.011015

Una vez obtenidos los resultados, podemos armar la siguiente tabla y realizar el gráfico correspondiente para una mejor visualización.

Tamaño Lista	Tiempo Iterativo (s)	Tiempo Recursivo (s)
100	0.000005	0.000008
1000	0.000029	0.000089
5000	0.000142	0.000498
10000	0.000295	0.001053
50000	0.001394	0.005339
100000	0.002766	0.011015



Con los resultados obtenidos, podemos observar claramente que a medida que el tamaño de los datos de entrada aumentaba, la función recursiva fue creciendo en mayor proporción que la función iterativa.

Ambas funciones tienen un crecimiento lineal, que en notación Big O se representa como O(n). A medida que el tamaño de la lista crece, el tiempo de ejecución aumenta de manera proporcional.

Sin embargo, el método iterativo es más rápido, manteniéndose por debajo del método recursivo. Esto muestra que, para cualquier tamaño de lista, la versión iterativa termina el trabajo en menos tiempo.

También se confirma que una función que se llama a sí misma, puede resultar en un aumento del tiempo de ejecución, volviéndose más lenta.

Enlace al repositorio Git:

https://github.com/enkai12/UTN-INTEGRADORES/tree/main/Programacion1

6. Conclusiones

El Análisis de Algoritmos es una habilidad muy importante que permite evaluar diferentes soluciones propuestas para resolver un mismo problema, permitiendo un análisis más profundo y facilitando la tarea de optimizar algoritmos, dependiendo del contexto donde se aplique. Como estudiantes y futuros programadores, es importante poder replicar el análisis de algoritmos en cualquier proyecto que se trabaje.

El trabajo en equipo resulta imperativo, no sólo para desarrollar habilidades de comunicación y cooperación en la resolución de un problema que se presenta, sino también para adquirir conocimiento personal y la oportunidad de mejorar los algoritmos.

A futuro, se podría profundizar en analizar otras soluciones algorítmicas que se pueden aplicar al mismo problema y la investigación de otras funciones que permitan analizar cómo el algoritmo utiliza los recursos disponibles. Finalmente, pudimos comprobar con la aplicación del caso práctico que ambas soluciones propuestas eran funcionales, no obstante la función iterativa nos ofrece la solución más óptima en cuanto a los tiempos de ejecución.

7. Bibliografía

Brassard, G., & Bratley, P. (1997). *Fundamentos de algoritmia* (R. García-Bermejo, Trans.). Pearson Educación.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Ferguson, N. (1 de Noviembre de 2022). *Big-O Notation Explained in Plain English | by Natasha Ferguson*. Medium. Recuperado el 07/06/2025, de https://medium.com/@teamtechsis/big-o-notation-explained-in-plain-english-983b0f7227aa

Lutz, M. (2011). *Programming Python*. O'Reilly Media, Incorporated.

Python Software Foundation. (s.f.). *timeit — Measure execution time of small code snippets*. Python documentation. Recuperado el 06/06/2025, de https://docs.python.org/3/library/timeit.html

Singanamonimallesh. (25 de Septiembre de 2024). *Asymptotic Analysis and Asymptotic Notations*. Medium. Recuperado el 29/05/2025, de https://medium.com/@singanamonimallesh/asymptotic-analysis-and-asymptotic-notations-5b222c4f90e2

Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press. Disponible en: https://archive.org/details/Automate-Boring-Stuff-Python-2nd

8. Anexos

Enlace al video explicativo: https://youtu.be/wNkBKJ2LyiE