Trabajo Integrador de programación

Análisis de Algoritmos

Análisis de la eficiencia algorítmica en Python

Alumnos: Brian Emanuel Rios (brian.rios977@gmail.com)

Mauricio Osvaldo Rios (mauricio.rios991@gmail.com)

Materia: Programación I

Profesor: Prof. Nicolás Quirós

Tutor: Francisco Quarñolo

Fecha de Entrega: 9 de junio de 2025

Video a Youtube: https://youtu.be/_MnDbXsOsEY

Repo GitHub: https://github.com/brianrios97/UTN-TUPaD-P1/tree/main/TP%20Integrador%20de%20Programacion

---

Introducción:

En este trabajo vamos a aplicar los conocimientos adquiridos en cuanto a la eficiencia de los algoritmos, aplicando conocimiento teórico y empírico sobre el tema con dos ejercicios prácticos.

El análisis de algoritmos nos permite medir y comparar la eficiencia de los programas en términos de tiempo de ejecución y uso de memoria. Comprender como se comporta un algoritmo frente a un aumento de datos es esencial para desarrollar aplicaciones eficientes y escalables en nuestros programas.

En esta investigación vamos a repasar conceptos básicos sobre algoritmos, dejaremos el enfoque en claro sobre el objetivo de nuestro análisis y haremos dos ejemplos prácticos junto con su conclusión

Programación I

Marco Teórico:

Antes de empezar nuestro análisis primero debemos tener bien en claro que es un algoritmo, un algoritmo es un procedimiento o conjunto de instrucciones bien definidas que permite resolver un problema o realizar una tarea especifica

Análisis de Algoritmos: Fundamentos y Técnicas de Optimización:

El análisis de algoritmos es una rama fundamental de las ciencias de la computación que estudia el rendimiento de los algoritmos en términos de eficiencia temporal y espacial. Este estudio permite comprender qué tan bien se comporta un algoritmo cuando se enfrenta a grandes volúmenes de datos, constituyendo una herramienta indispensable para el desarrollo de software escalable y optimizado.

Objetivos del Análisis de Algoritmos:

Eficiencia Temporal: Evalúa cuánto tiempo tarda el algoritmo en ejecutarse según el tamaño de los datos de entrada
Eficiencia Espacial: Determina cuánta memoria adicional necesita el algoritmo para completar su ejecución
Escalabilidad: Analiza cómo se comporta el algoritmo al aumentar significativamente el tamaño de los datos

Enfoques de Análisis

Análisis Teórico (Asintótico):	**Análisis Empírico (Experimental)**
Utiliza la notación Big-O para expresar el crecimiento del algoritmo en función del tamaño de entrada. Proporciona una comprensión matemática del comportamiento algorítmico.	**Mide el tiempo de ejecución real en diferentes condiciones y tamaños de datos. Ofrece resultados prácticos y tangibles del rendimiento.**
Este enfoque permite comparar algoritmos independientemente del hardware o lenguaje de programación utilizado.	**Es especialmente útil para validar las predicciones teóricas y optimizar implementaciones específicas.**

Introducción al Análisis Teórico

El análisis teórico es un enfoque matemático para evaluar la eficiencia de un algoritmo sin necesidad de implementarlo. Este método se basa en el pseudocódigo (usaremos Python como lenguaje) del algoritmo y permite calcular una función temporal T(n), que representa el número de operaciones que realiza el algoritmo para una entrada de tamaño n.

Ventajas del análisis teórico:

- **No depende del hardware o software específico.**
- **Permite considerar todas las posibles entradas, no solo un conjunto limitado.**
- **Es más general y abstracto que el análisis empírico.**

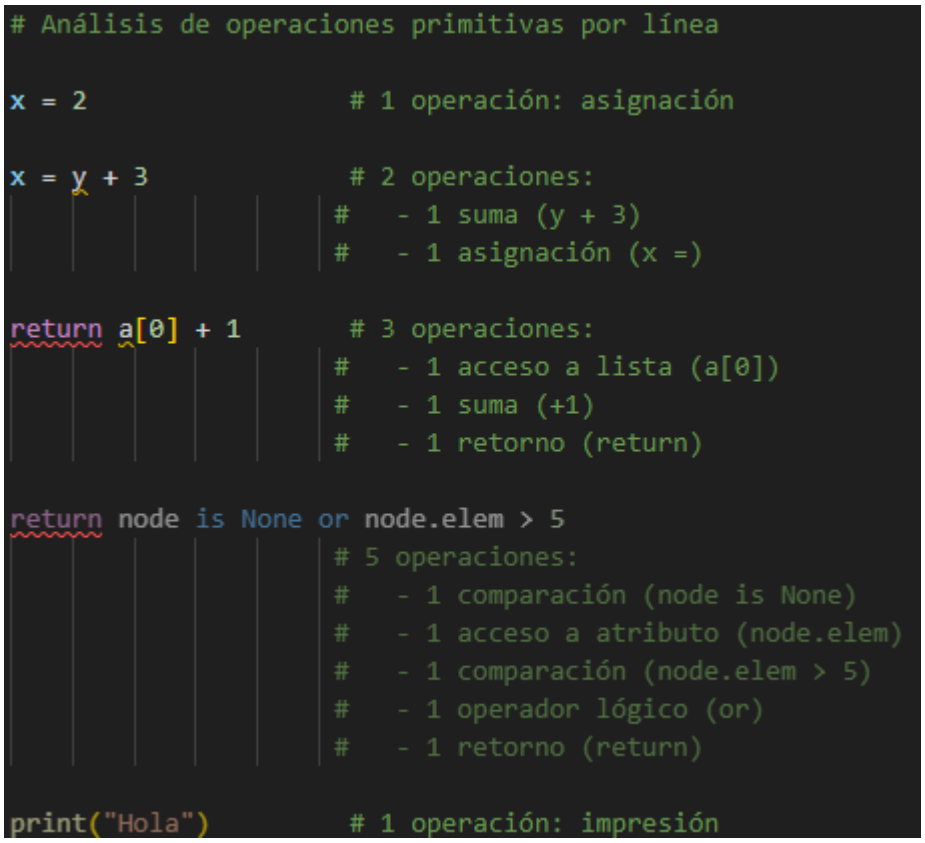
Cálculo de la Función Temporal T(n)

Operaciones primitivas:

• Se consideran operaciones primitivas aquellas como asignaciones, accesos a arreglos, evaluaciones de expresiones, entre otras.

• Cada una de estas operaciones se cuenta como una unidad de tiempo.

Ejemplo:



```
# Análisis de operaciones primitivas por línea

x = 2                   # 1 operación: asignación

x = y + 3               # 2 operaciones:
                        #   - 1 suma (y + 3)
                        #   - 1 asignación (x =)

return a[0] + 1         # 3 operaciones:
                        #   - 1 acceso a lista (a[0])
                        #   - 1 suma (+1)
                        #   - 1 retorno (return)

return node is None or node.elem > 5
                        # 5 operaciones:
                        #   - 1 comparación (node is None)
                        #   - 1 acceso a atributo (node.elem)
                        #   - 1 comparación (node.elem > 5)
                        #   - 1 operador lógico (or)
                        #   - 1 retorno (return)

print("Hola")           # 1 operación: impresión
```

Estructuras de Control:

Las estructuras de control influyen en la complejidad temporal de un algoritmo. A continuación, se explica cómo se calcula la función T(n) en cada caso:

Secuencia:

Cuando un algoritmo está compuesto por varios bloques que se ejecutan uno tras otro, la función temporal total es la suma de las funciones temporales de cada bloque

```python
# Bloque B1: asignar valor a una variable
a = 5       # O(1)          # T(B1) = O(1)
# Bloque B2: asignar valor a una variable
b = 10      # O(1)          # T(B2) = O(1)
# Bloque B3: asignar valor a una variable
c = 15      # O(1)          # T(B3) = O(1)

# La funcion t(n) para esta funcion es:
# T(n) = T(B1) + T(B2) + T(B3)
# T(n) = O(1) + O(1) + O(1)
# T(n) = O(3)

# Complejidad temporal: O(1) (constante)
# Complejidad espacial: O(1) (constante, solo se usan variables escalares)
```

$$\boldsymbol{Formula}\ \boldsymbol{T(n)} = \boldsymbol{T(B_1)} + \boldsymbol{T(B_2)} + \ldots + \boldsymbol{T(B_k)}$$

Condicionales (if - elif - else):

En este tipo de estructura, solo se ejecuta uno de los bloques posibles. Por lo tanto, la función T(n) se calcula tomando el máximo de los tiempos posibles de cada bloque.

```python
def analizar_numero(x):
    if x < 0:
        resultado = "Negativo"          # 1 comparación + 1 asignación # T(B1)
    elif x == 0:
        resultado = "Cero"              # 2 comparaciones (incluye x < 0) + 1 asignación # T(B2)
    else:
        resultado = "Positivo"          # 2 comparaciones (x < 0 y x == 0) + 1 asignación # T(B3)
    return resultado                    # 1 operación (retorno) # T(B4)
# Análisis de la complejidad temporal
# T(n) = T(B1) + T(B2) + T(B3) + T(B4)
# T(n) = O(1) + O(1) + O(1) + O(1)
# T(n) = O(4)
# Complejidad temporal: O(1) (constante)
# Complejidad espacial: O(1) (constante, solo se usa una variable para el resultado)
```

$$\boldsymbol{Formula}\ \boldsymbol{T.if-else(n)} = \boldsymbol{max\{T(B_1), T(B_2), \ldots, T(B_k)\}}$$

Bucles:

La función T(n) para un bucle se calculará como el número de veces que se ejecuta el bucle (número de iteraciones) por la función T(n) del bloque interno B.

```
n = 2                  # T(B1) = O(1) → Asignación simple
suma = 0               # T(B2) = O(1) → Asignación simple
for i in range(1, n+1):    # T(B3) = O(n) → Bucle con n iteraciones
    suma = suma + i        # T(B4) = O(2) por iteración (suma + asignación)
print(suma)            # T(B5) = O(1) → Operación de impresión
# Análisis de la complejidad temporal
# T(n) = T(B1) + T(B2) + T(B3) + T(B4) + T(B5)
# T(n) = O(1) + O(1) + O(n) + O(2) + O(1)
# T(n) = O(n) (lineal)
# Total: 1 + 1 + n * 2 + 1 = 2n + 3 operaciones
# Función O(n) - Suma de los primeros n números enteros
# Complejidad espacial:
# S(n) = O(1) (constante)
# Complejidad espacial: O(1) (constante)
```

$$\boldsymbol{T\ loop\ (n)\ =\ T(B)\ *\ número\ de\ iteraciones}$$

Bucles anidados:

Cuando hay bucles dentro de otros, se multiplica el número de iteraciones de cada uno. Por ejemplo, si un bucle externo itera n veces y el interno también n veces, y el cuerpo del bucle interno tarda 2 unidades de tiempo, entonces:

```
# Bucles anidados
for i in range(n):             # Bucle externo → n iteraciones → O(n)
    for j in range(n):         # Bucle interno → n iteraciones por cada i → O(n)
        print(i * j)           # 2 operaciones por iteración: multiplicación + impresión → O(1)
# COMPLEJIDAD TEMPORAL:
# - El cuerpo interno se ejecuta n × n veces → n² veces
# - En cada iteración se realizan 2 operaciones constantes → O(1)
# T(n) = n × n × 2 = 2n² → O(n²)
# COMPLEJIDAD ESPACIAL:
# - Solo se usan variables escalares (i, j) → no se usan estructuras de datos
# S(n) = O(1) (constante)
```

$$\boldsymbol{Su\ función\ temporal\ T(n)\ es: T(n)\ =\ n\ *\ n\ *\ 2\ =\ 2n^2}$$

Es decir, 2 del bucle interno, por el número de iteraciones del bucle interno, por el número de iteraciones del bucle externo

Programación I

Introducción a la Notación Big-O

La notación Big-O describe cómo escala el tiempo de ejecución de un algoritmo en función del tamaño de entrada n. Es la herramienta fundamental para expresar la complejidad algorítmica de manera matemática y comparable.

Esta notación nos permite abstraer los detalles de implementación y hardware para centrarnos en el comportamiento fundamental del algoritmo cuando los datos crecen hacia el infinito.

Pasos para calcular Big-O:

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

Ejemplos:

- **$T(n) = 3n^2 + 2n + 1 \rightarrow O(n^2)$.**
- **$T(n) = 5n + 10 \rightarrow O(n)$.**
- **$T(n) = 7 \rightarrow O(1)$.**

Tipo de complejidades temporales:

O(1) – Constante	**Tiempo independiente del tamaño de entrada. Ejemplo: acceso directo a un índice de array.**
O(log n) - Logarítmica	Crecimiento muy lento. Ejemplo: búsqueda binaria en estructuras ordenadas.
O(n) - Lineal	Tiempo proporcional al tamaño. Ejemplo: recorrido completo de una lista.
$O(n^2)$ - Cuadrática	Crecimiento cuadrático. Ejemplo: algoritmos de ordenamiento básicos como bubble sort
O(n log n) - Cuasilineal	Optimizado para ordenamientos eficientes como merge sort y quick sort
$O(2^n)$ - Exponencial	Muy costoso computacionalmente. Ejemplo: Fibonacci recursivo sin optimización
O(n!) - Factorial	Computacionalmente prohibitivo. Aparece en problemas de permutación completa

Caso Práctico

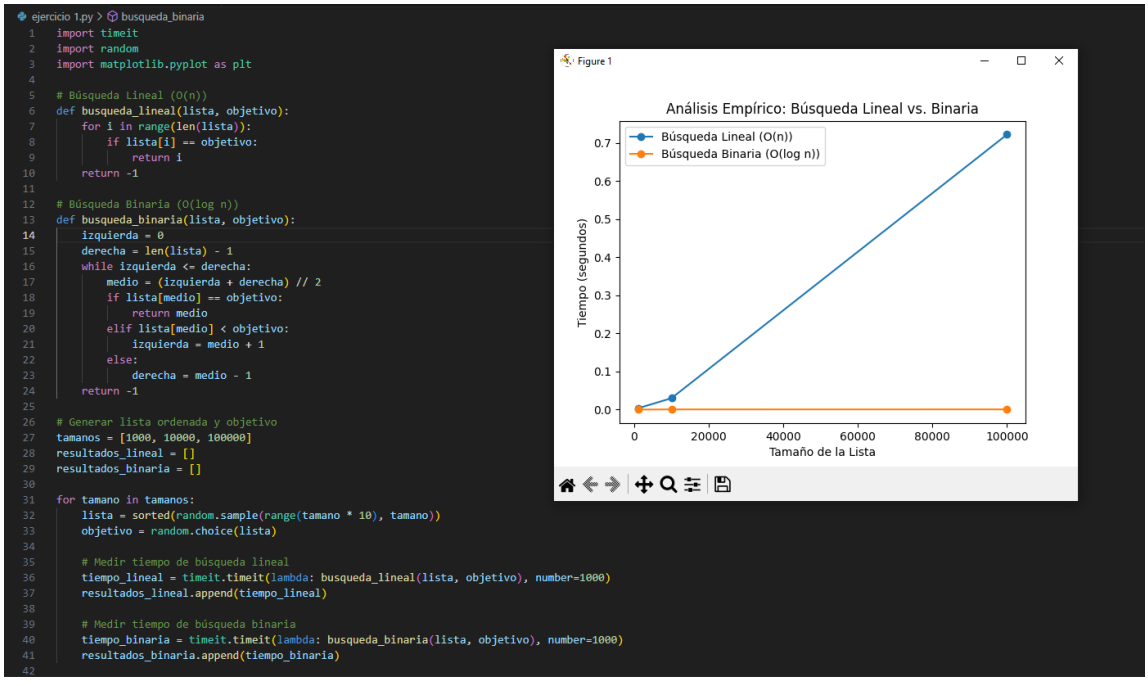
Caso 1: Análisis Empírico de Algoritmos de Búsqueda

Objetivo:
Comparar empíricamente el rendimiento de dos algoritmos de búsqueda (lineal y binaria) evaluando su tiempo de ejecución sobre listas de distintos tamaños, con el fin de validar sus complejidades teóricas.

La búsqueda binaria, de complejidad **O(log n)**, es considerablemente más eficiente que la búsqueda lineal (**O(n)**) para listas grandes, ya que reduce a la mitad el espacio de búsqueda en cada iteración. No obstante, **requiere que la lista esté previamente ordenada**, lo cual puede implicar un costo adicional no considerado en este análisis

Metodología:

1. Se implementaron dos funciones en Python:
    - **Búsqueda Lineal** O(n): recorre secuencialmente cada elemento de la lista hasta encontrar el objetivo.
    - **Búsqueda Binaria** O(log n): Divide repetidamente la lista ordenada hasta hallar el elemento buscado.
2. Se generaron listas ordenadas de tamaños variables (1k, 10k y 100k elementos) y se midieron los tiempos de ejecución con timeit.
3. Los resultados se visualizaron en una tabla y gráfico (usando matplotlib), demostrando cómo el tiempo de la búsqueda lineal crece linealmente, mientras que el de la binaria permanece casi constante.



```python
import timeit
import random
import matplotlib.pyplot as plt

# Búsqueda Lineal (O(n))
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

# Búsqueda Binaria (O(log n))
def busqueda_binaria(lista, objetivo):
    izquierda = 0
    derecha = len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

# Generar lista ordenada y objetivo
tamanos = [1000, 10000, 100000]
resultados_lineal = []
resultados_binaria = []

for tamano in tamanos:
    lista = sorted(random.sample(range(tamano * 10), tamano))
    objetivo = random.choice(lista)

    # Medir tiempo de búsqueda lineal
    tiempo_lineal = timeit.timeit(lambda: busqueda_lineal(lista, objetivo), number=1000)
    resultados_lineal.append(tiempo_lineal)

    # Medir tiempo de búsqueda binaria
    tiempo_binaria = timeit.timeit(lambda: busqueda_binaria(lista, objetivo), number=1000)
    resultados_binaria.append(tiempo_binaria)
```

Resultados

Como se observa en el gráfico, el tiempo de ejecución de la búsqueda lineal crece linealmente con el tamaño de la lista, validando su complejidad O(n). En contraste, la búsqueda binaria mantiene un tiempo casi constante, lo que refleja su complejidad logarítmica O(log n).

Gráfico:

- **Eje X: Tamaño de la lista.**
- **Eje Y: Tiempo de ejecución promedio (segundos).**
- **Líneas: Azul (Búsqueda lineal), Naranja (Búsqueda binaria).**

Conclusión:
El análisis empírico confirmó la superioridad de la búsqueda binaria en datasets grandes, validando su complejidad teórica O(log n). Sin embargo, se destaca que este algoritmo requiere una lista previamente ordenada, lo que implica un costo adicional no considerado en esta medición.

Caso 2: Aplicación de la Notación Big O en Funciones Básicas

Objetivo:
Demostrar cómo la **complejidad algorítmica** afecta el rendimiento práctico, contrastando una operación lineal **O(n)** con una recursiva ineficiente de complejidad exponencial **$O(2^n)$**. Este ejemplo destaca por qué es fundamental analizar la eficiencia de los algoritmos, especialmente los **recursivos no optimizados**.

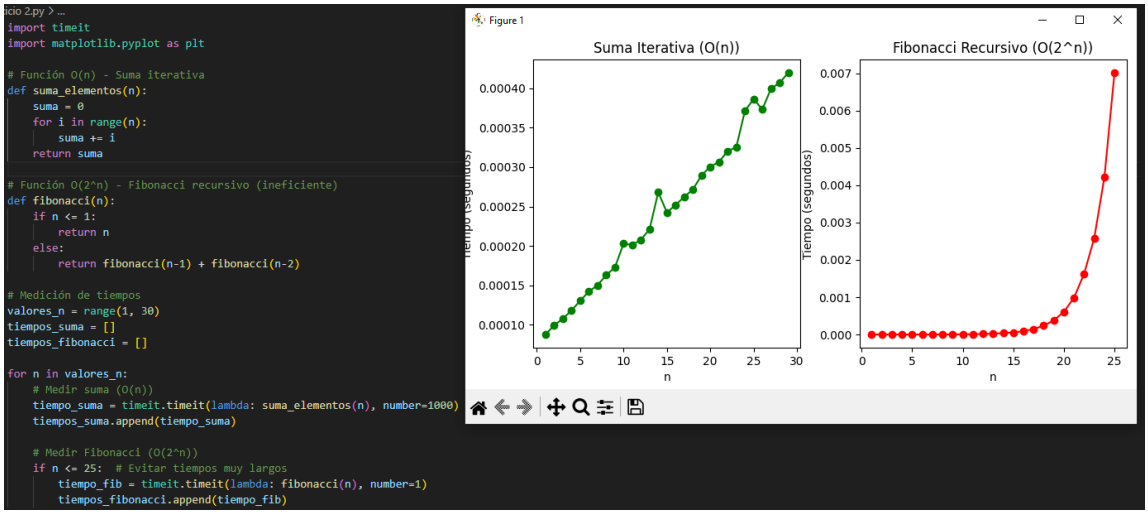
Nota importante: los algoritmos recursivos pueden parecer simples y elegantes, pero sin técnicas de optimización como *memorización*, pueden volverse ineficientes y costosos en tiempo y recursos.

Metodología:

1. Se implementaron dos funciones:
    - Suma iterativa O(n): acumula la suma de los primeros *n* números naturales usando un bucle for.
    - Fibonacci recursivo $O(2^n)$: calcula el n-ésimo número de Fibonacci mediante una implementación **recursiva ingenua, generando múltiples llamadas redundantes.**
2. Se midieron los tiempos de ejecución usando timeit para valores de *n* entre **1 y 29** en el caso de la suma, y hasta **n = 24** en el caso de Fibonacci (para evitar tiempos excesivos), observando cómo el tiempo de Fibonacci se dispara exponencialmente frente al crecimiento lineal de la suma.

3. Se destacó visualmente la diferencia mediante gráficos

    - Panel izquierdo: crecimiento **lineal** de la suma.
    - Panel derecho: crecimiento **exponencial** del Fibonacci recursivo.



Resultados

- En el gráfico, se observa cómo el **tiempo de ejecución del algoritmo de suma crece de forma constante y predecible**, como se espera de un algoritmo O(n).
- En contraste, **la función Fibonacci se vuelve cada vez más lenta de forma abrupta**, duplicando casi el tiempo de ejecución con cada incremento de *n*. Esto valida empíricamente su complejidad $O(2^n)$.

El gráfico refleja claramente el **desequilibrio de eficiencia** entre ambas estrategias, incluso con entradas pequeñas.

Conclusión:

El caso ilustra por qué **no basta con que un algoritmo funcione correctamente**, sino que también debe ser **eficiente**.

Las implementaciones recursivas deben analizarse cuidadosamente y optimizarse cuando sea necesario ya que pueden crecer muy rápidamente en costos operacionales.

La **elección del algoritmo correcto** puede ser la diferencia entre una solución práctica y una ineficiente, especialmente en escenarios de crecimiento exponencial.

Bibliografia:

Video 1 - Introducción al análisis de algoritmos
Video 2 - Análisis empirico de sumN
Video 3 - Análisis empirico de sumGaus
Introducción al Análisis de Algoritmos por.Ariel Enferrel
Análisis Empirico
Video 1 - Introducción al análisis teorico de algoritmos
Video 2 - Analisis Teorico de las estructuras de control
Video 3 - Análisis teorico de sumN y sumGaus
Video 4 - Primer ejemplo de analisis teorico
Video 5 - Segundo ejemplo analisis teorico
Análisis Teórico de Algoritmo por Ariel Enferrel
Análisis Teorico y BIG-O
Video 1 - Introducción a BIG-O
Video 2 - Tipos de ordenes de BIG-O
Video 3 -Ejemplo de como obtener la BIG-O
Notacion BIG-O
ChatGPT
GoogleGemini