Análisis de la eficiencia algorítmica en Python

Análisis de la eficiencia algorítmica en Python

• Alumno: Julián Diaz Andre (julianandre004@gmail.com)

• Materia: Programación I

• Profesor/a: Julieta Agustina Trapé

•Tutor: Miguel Barrera Oltra

• Fecha de Entrega: 20/06/2025

1. Introducción

Este trabajo aborda el análisis de algoritmos desde la perspectiva de la eficiencia, evaluando el rendimiento en tiempo y uso de memoria. El objetivo es comprender cómo la complejidad algorítmica afecta directamente el comportamiento de los programas a medida que crece la cantidad de datos. Se eligió este tema por su relevancia en el desarrollo de software eficiente y escalable, algo fundamental en programación moderna.

2. Marco Teórico

La notación Big O es una herramienta utilizada para describir el comportamiento asintótico de un algoritmo. Expresa cómo crece el tiempo de ejecución o el uso de memoria en función del tamaño de los datos de entrada. Algunas complejidades comunes son:
- O(1): constante
- O(log n): logarítmica
- O(n): lineal
- $O(n^2)$: cuadrática
- $O(2^n)$: exponencial

Big O no mide tiempos exactos, sino cómo escala un algoritmo. Es útil para comparar alternativas antes de implementarlas. En Python, se puede medir el tiempo con “time” y la memoria con “psutil”.

3. Caso Práctico

Se desarrolló un caso real basado en el cálculo del índice de masa corporal (IMC). A partir de la fórmula IMC = peso / altura², se implementaron cinco versiones del cálculo que representaran diferentes complejidades: O(1), O(n), $O(n^2)$, O(log n) y $O(2^n)$. Se usaron estructuras como listas, bucles anidados y recursividad, simulando diferentes cargas de procesamiento. Se midieron tiempos y memoria para comparar eficiencia algorítmica.

También se desarrolló un segundo bloque de pruebas teóricas con estructuras clásicas para observar diferencias en escalabilidad.

Caso real.

```
#Paquetes utilizados
import time
import psutil
import os

# Función para mostrar uso de memoria actual (en MB)
def memoria_actual():
    process = psutil.Process(os.getpid())
    memoria_mb = process.memory_info().rss / 1024 ** 2
    return memoria_mb

# Función para medir tiempo y memoria de un bloque (función pasada como
parámetro)
def medir(func, *args, **kwargs):
    mem_inicio = memoria_actual()
    t_inicio = time.time()

    resultado = func(*args, **kwargs)

    t_final = time.time()
    mem_final = memoria_actual()

    tiempo = t_final - t_inicio
    memoria = mem_final - mem_inicio  # memoria consumida durante la
ejecución

    return resultado, tiempo, memoria

# Funciones general
def calculos_imc(peso, altura):
    if altura == 0:
        return 0
    return round(peso / (altura ** 2), 2)

#funcion para caso logaritmico
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
```

```
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1

#funcion para caso exponencial
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Casos a medir como funciones para facilitar la medición

def caso1_constante():
    peso = 97
    altura = 1.89
    return calculos_imc(peso, altura)

def caso2_lineal():
    personas_imc = [
        (89, 1.74),
        (67, 1.65),
        (103, 1.93)
    ]
    resultados = []
    for peso, altura in personas_imc:
        resultados.append(calculos_imc(peso, altura))
    return resultados

def caso3_cuadratica():
    personas_imc = [
        (89, 1.74),
        (67, 1.65),
        (103, 1.93)
    ]
    imcs = [calculos_imc(p, a) for p, a in personas_imc]
    resultados = []
    for i in range(len(imcs)):
        for j in range(len(imcs)):
            resultados.append(imcs[i] * imcs[j])
    return resultados
```

```
def caso4_logaritmica():
    datos_ordenados = list(range(0, 1000000, 2))
    return busqueda_binaria(datos_ordenados, 882)

def caso5_exponencial():
    return fibonacci(20)

# --- Medición ---

res1, tiempo1, mem1 = medir(caso1_constante)
res2, tiempo2, mem2 = medir(caso2_lineal)
res3, tiempo3, mem3 = medir(caso3_cuadratica)
res4, tiempo4, mem4 = medir(caso4_logaritmica)
res5, tiempo5, mem5 = medir(caso5_exponencial)

# --- Resultados ---

print(f"Caso 1 (Constante O(1))       - Tiempo: {tiempo1:.8f}s -
Memoria: {mem1:.4f} MB")
print(f"Caso 2 (Lineal O(n))          - Tiempo: {tiempo2:.8f}s -
Memoria: {mem2:.4f} MB")
print(f"Caso 3 (Cuadrática O(n²))     - Tiempo: {tiempo3:.8f}s -
Memoria: {mem3:.4f} MB")
print(f"Caso 4 (Logarítmica O(log n)) - Tiempo: {tiempo4:.8f}s -
Memoria: {mem4:.4f} MB")
print(f"Caso 5 (Exponencial O(2^n))   - Tiempo: {tiempo5:.8f}s -
Memoria: {mem5:.4f} MB")
```

Caso teorico

```
#Modulos
import time
import psutil
import os

# Función para obtener memoria usada (en MB)
def memoria_actual():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss / 1024**2

# Función para medir tiempo y memoria usando psutil
```

```python
def medir(func, *args, **kwargs):
    mem_inicio = memoria_actual()
    t_inicio = time.time()

    resultado = func(*args, **kwargs)

    t_fin = time.time()
    mem_fin = memoria_actual()

    tiempo = t_fin - t_inicio
    memoria = mem_fin - mem_inicio  # Memoria consumida en MB

    return resultado, tiempo, memoria

# --- Funciones para cada caso de complejidad ---
#Caso 1
def constante(lista):
    return lista[0]

#Caso 2
def lineal(lista):
    total = 0
    for x in lista:
        total += x
    return total

#Caso 3
def cuadratica(lista):
    contador = 0
    for i in lista:
        for j in lista:
            contador += 1
    return contador

#Caso 4
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
```

```
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1

#Caso 5
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# --- Ejecutar y medir ---

lista_corta = list(range(1000))         # Para O(n²)
lista_larga = list(range(1_000_000))    # Para O(n) y O(log n)

res1, tiempo1, memoria1 = medir(constante, lista_larga)
res2, tiempo2, memoria2 = medir(lineal, lista_larga)
res3, tiempo3, memoria3 = medir(cuadratica, lista_corta)
res4, tiempo4, memoria4 = medir(busqueda_binaria, lista_larga, 987654)
res5, tiempo5, memoria5 = medir(fibonacci, 30)

print(f"O(1) - Constante: {tiempo1:.8f} s - Memoria consumida:
{memoria1:.6f} MB")
print(f"O(n) - Lineal: {tiempo2:.8f} s - Memoria consumida:
{memoria2:.6f} MB")
print(f"O(n²) - Cuadrática: {tiempo3:.8f} s - Memoria consumida:
{memoria3:.6f} MB")
print(f"O(log n) - Logarítmica: {tiempo4:.8f} s - Memoria consumida:
{memoria4:.6f} MB")
print(f"O(2^n) - Exponencial: {tiempo5:.8f} s - Memoria consumida:
{memoria5:.6f} MB")
```

4. Metodología Utilizada

Primero se investigaron conceptos teóricos de complejidad y medición de rendimiento. Luego se implementaron los algoritmos en Python, utilizando “time.time()” para medir tiempo y “psutil” para medir memoria.

5. Resultados Obtenidos

En el caso real, los tiempos fueron muy bajos debido al pequeño tamaño de los datos, pero el algoritmo exponencial se destacó por su lentitud relativa. En el caso teórico, las diferencias fueron más claras: O(1), O(n) y O(log n) fueron muy rápidos incluso con listas grandes, mientras que $O(n^2)$ y $O(2^n)$ mostraron un crecimiento notable del tiempo. En cuanto a la memoria, solo el logarítmico (por el tamaño de la lista) y el exponencial (por la recursividad) mostraron un consumo mayor.

6. Conclusiones

Este trabajo permitió aplicar conceptos teóricos de eficiencia a casos concretos, observando cómo la estructura de un algoritmo afecta su rendimiento. Se comprobó que la notación Big O tiene impacto práctico, especialmente al escalar los datos. Se reforzó la importancia de elegir algoritmos adecuados desde el inicio, para evitar cuellos de botella y garantizar soluciones eficientes y sostenibles en el tiempo.

7. Bibliografía

- Python Software Foundation. (2024). Python 3 Documentation. https://docs.python.org/3/

-Big O notation - wikipedia
https://en.wikipedia.org/wiki/Big_O_notation

-What is Big O Notation Explained: Space and Time Complexity.
https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/

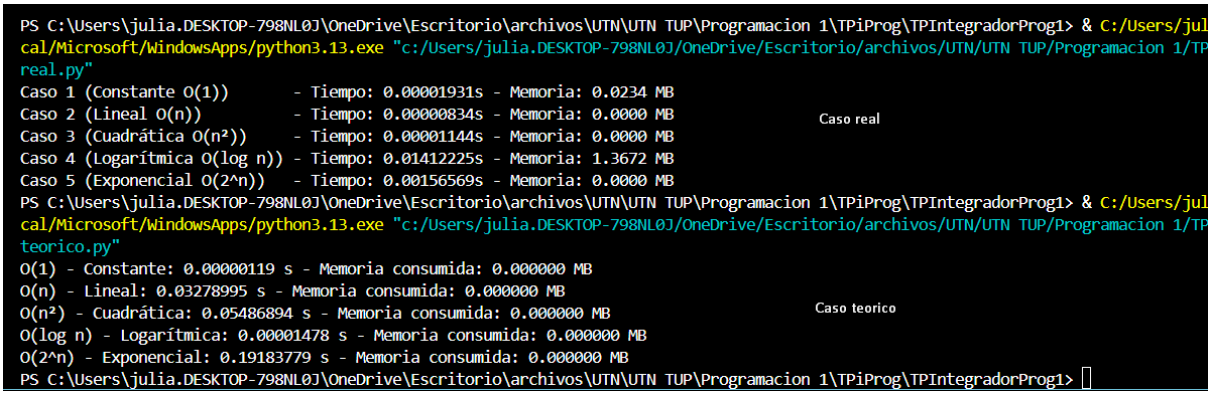
-Os documentación.
https://docs.python.org/3/library/os.html

-Psutil documentación.
https://psutil.readthedocs.io/

-Time documentación
https://docs.python.org/3/library/time.html

8. Anexo

Captura de los resultados



Link al repositorio de github
https://github.com/JulianDA90/TPIntegradorProg1

9. Video Explicativo

Video del TP
https://youtu.be/gtrTU8TIZ00