Taller 3-4

Andrés Salcedo Vera[1] y Eder David Barrero Castañeda[2]
Fundacion Universitaria Konrad Lorenz
[1]jaimea.salcedov@konradlorenz.com, [2]ederd.barreroc@konradlorenz.edu.co

***Resumen*—A continuación, se expondrán los resultados derivados de la ejecución de dos scripts: uno en Python y otro en Java. El propósito de este estudio es determinar tanto la complejidad temporal como la complejidad espacial en cada uno de los sistemas computacionales. Es relevante destacar que se incorporó el componente diferenciador de código, ya que cada uno de los participantes desarrolló su propia metodología para recopilar estas estadísticas. Este factor también influyó en el rendimiento y los resultados finales obtenidos.**

I. Introducción

Para el desarrollo de este taller, se proporcionaron dos scripts: uno en Python encargado de generar datos aleatorios y otro en Java dedicado a la limpieza de estos datos. El objetivo principal de esta actividad es determinar tanto la complejidad temporal como la complejidad espacial en cada uno de los ordenadores participantes, con el propósito de identificar y comprender el funcionamiento de características específicas dentro de nuestras máquinas.

Además, se llevará a cabo un análisis Big O con el fin de comprender por qué ciertos procesos son más rápidos o más lentos en determinadas situaciones, incluso cuando no se cuenta con las especificaciones más avanzadas en nuestro equipo. Este análisis nos permitirá valorar y aprovechar las características particulares de nuestros sistemas, demostrando que en ocasiones, tener "lo mejor"no es necesariamente la mejor opción.

II. Analisis python

El script de Python proporcionado presentaba un error en la lectura de datos, que requería una corrección individual en cada caso. El problema específico residía en la presencia de "generatedata(500)", ya que la función no requería ningún parámetro.

```
46 if __name__ == "__main__":
47     generated_data = generate_data(500)
48     save_to_csv(generated_data, 'dummy_data.csv')
49     print("Data generation and CSV creation complete.")
```

Figura 1. Script Entregado

```
def generate_data():
    data = []
    for _ in range(500**2):
        username = fake.user_name()
        birthdate = fake.date_of_birth(
            minimum_age=18,
            maximum_age=70).strftime('%Y-%m-%d')
        income = round(random.uniform(1000, 10000), 2)
        debt = round(random.uniform(0, 5000), 2)
        sex = random.choice(['Male', 'Female'])
        num_children = random.randint(0, 5)
        country = fake.country()
        data.append([
            username,
            birthdate,
            income,
            debt,
            sex,
            num_children,
            country])

    return data
```

Figura 2. Script Entregado

Para solucionar este problema, existían dos enfoques igualmente válidos. Uno de ellos consistía en eliminar el parámetro proporcionado a la función. El otro enfoque era modificar la función para que aceptara un parámetro y ajustar ese parámetro en el rango.

```
if __name__ == "__main__":
    start_time = time.time()
    memoria_inicio = psutil.virtual_memory().used
    generated_data = generate_data()
    save_to_csv(generated_data, 'dummy_data.csv')
    print("Data generation and CSV creation complete.")
    end_time = time.time()
    memoria_fin = psutil.virtual_memory().used
    difference_time = end_time - start_time
    print(difference_time)
    print(memoria_fin-memoria_inicio)
```

Figura 3. Solución 1

```python
def generate_data(cantidad):
    data = []
    for i in range(cantidad**2):
        username = fake.user_name()
        birthdate = fake.date_of_birth(
            minimum_age=18,
            maximum_age=70).strftime('%Y-%m-%d')
        income = round(random.uniform(1000, 10000), 2)
        debt = round(random.uniform(0, 5000), 2)
        sex = random.choice(['Male', 'Female'])
        num_children = random.randint(0, 5)
        country = fake.country()
        data.append([
            username,
            birthdate,
            income,
            debt,
            sex,
            num_children,
            country])

    return data
```

Figura 4. Solución 2

Ambas soluciones no añadían ni restaban valor al código en sí; más bien, representaban diferentes formas de abordar y resolver el problema.

II-A. Complegidad temporal, Complejidad espacial y Big O

En el script de Python, existen varias formas de medir el tiempo de ejecución. Una de ellas consiste en ejecutar todo el script en un cuaderno de Jupyter, aunque esta no es la opción más eficiente, proporciona una estimación del tiempo que lleva su ejecución. Otra alternativa es utilizar la biblioteca time en Python y agregarla al script para obtener un contador que registre el tiempo desde el inicio hasta el final de la ejecución.

La medición de la complejidad espacial se realizó de dos maneras distintas: una mediante la descarga de la biblioteca memory profiler para Python y la otra utilizando psutil. Se observaron notables diferencias en el rendimiento. La biblioteca memory profiler ofrece más información pero consume más recursos y tiempo debido a su mayor carga, mientras que psutil es ligera y no causa tantos problemas.

El análisis de la notación Big O depende de la función específica, ya que en el script de Python se encuentran dos funciones diferentes. En la Figura 4, se puede apreciar un bucle for, al que se le agregan funciones provenientes de las bibliotecas instaladas. En este caso particular, la complejidad sería O(n).

En cuanto a la función que utiliza la función open de la biblioteca, esta recorre línea por línea para escribir los datos. La complejidad en este caso puede aproximarse a O(log n), lo que significa que su impacto en el rendimiento es relativamente bajo y que se ejecuta de manera eficiente.

```python
@profile
def save_to_csv(data, filename):
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([
            'Username',
            'Birthdate',
            'Income',
            'Debt',
            'Sex',
            'Number of Children',
            'Country'])
        writer.writerows(data)
```

Figura 5. Segunda funcion

III. Analisis Java

El script en Java se encarga de leer el archivo CSV generado por el script de Python. Para comprender su complejidad espacial y temporal, debemos ejecutarlo en la IDE IntelliJ 2023 y utilizar un profiler para analizar tanto el tiempo de ejecución como el uso de memoria. En cuanto al código en sí, no se han identificado errores, por lo que no es necesario realizar ajustes adicionales, aparte de la lectura del documento.

III-A. Big O

El análisis Big O del código en Java se enfoca en el ciclo while que recorre cada una de las líneas del archivo CSV y realiza operaciones de limpieza de datos según sea necesario. Si este ciclo tiene una complejidad que depende del número de líneas del CSV y no del número de elementos por línea, entonces el resultado del análisis Big O sería más apropiadamente O(n), donde 'n' representa el número total de líneas en el archivo CSV.

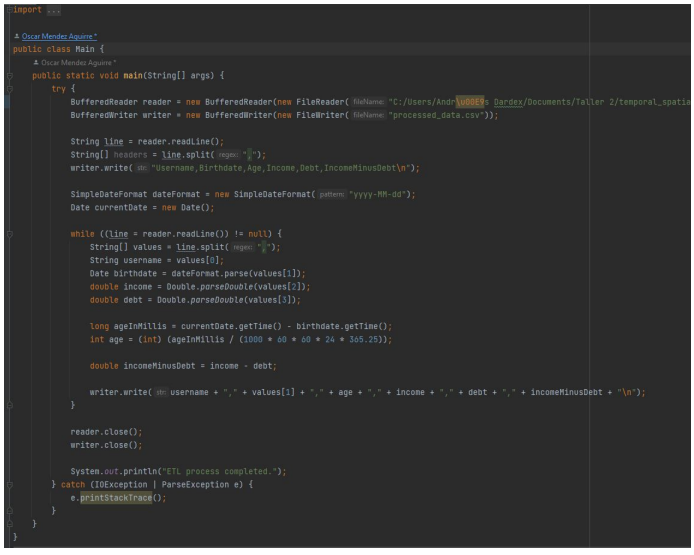

Figura 6. Script de java

Resultados

Uno de los computadores era el Asus tuf gaming f16 con las siguientes especificaciones

- CPU: intel core i5 10000
- Grafica: Ge force GTX 1660 ti
- RAM: 16 GB
- SSD: 512 GB
- Disco duro: 700 GB

El lenovo AMD con las siguientes especificaciones:

- CPU: Ryzen 5
- RAM: 8 GB
- SSD: 512 GB

y nos dio los siguientes resultados:

MÁQUINA	COMPLEJIDAD	PYTHON DEBUG 1	PYTHON DEBUG 2	PYTHON DEBUG 3	Promedio Python
Asus tuf gaming f15	TEMPORAL (seg)	254,06	254,44	248,99	252
	ESPACIAL (MB)	137,3	137,4	137,5	137
Lenovo AMD	TEMPORAL (seg)	46,26	47,46	45,82	47
	ESPACIAL (MB)	139,4	734,04	564,71	479

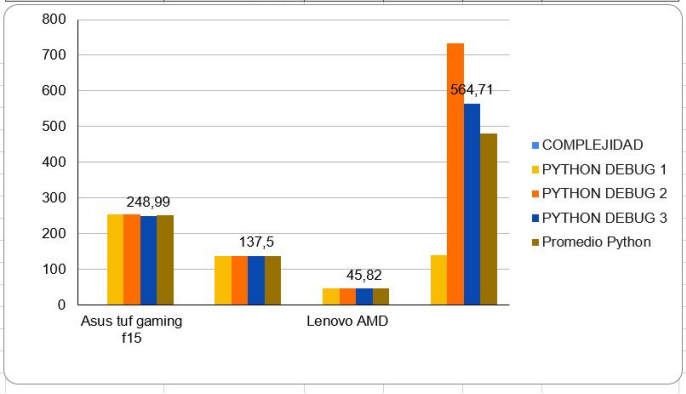

Figura 7. Complejidad Python

Como se puede observar, los resultados obtenidos por parte de Aus TUF Gaming en el script de Python muestran deficiencias en cuanto a la complejidad temporal. Esto se debe a que, como se mencionó previamente, el script se ejecutó de diferentes maneras y experimentó un procesamiento más complejo en este equipo en comparación con el Lenovo. En el Lenovo, por otro lado, se presentaron mayores problemas en la complejidad espacial, que podrían estar relacionados con las diferencias en el procesador y la memoria.

En el caso de Java, no se aprecia una diferencia significativa. Java demuestra una adaptabilidad más completa y eficiente en comparación con otros lenguajes, lo que puede dar lugar a resultados similares al ejecutar el código en ambos equipos, a pesar de que teóricamente uno de los equipos sea superior al otro.

MÁQUINA	COMPLEJIDAD	JAVA DEBUG 1	JAVA DEBUG 2	JAVA DEBUG 3	Promedio Java
Asus tuf gaming f15	TEMPORAL (seg)	1,2	1	1	1
	ESPACIAL (MB)	350,3	350,3	350,4	350
Lenovo AMD	TEMPORAL (seg)	2	2	2	2
	ESPACIAL (MB)	408,8	396,31	407,6	404

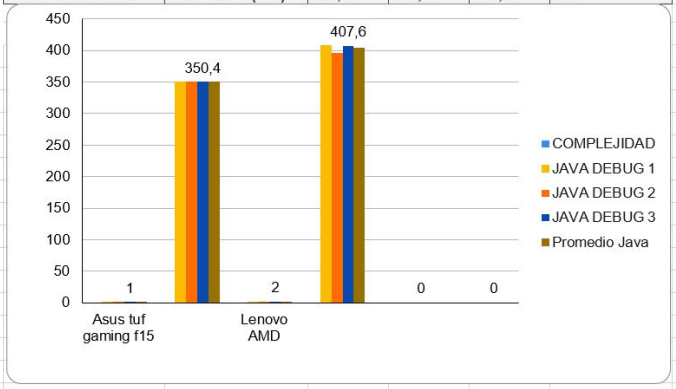

Figura 8. Complejidad Java

IV. Conclusiones

El desarrollo de este taller brinda una valiosa oportunidad para profundizar en la comprensión de la complejidad de un algoritmo en diversas máquinas. Contrario a la creencia común, no siempre se requieren las especificaciones más costosas o potentes para obtener los mejores resultados. La comprensión de las bases de las complejidades algorítmicas nos capacita para optimizar procesos, comprender las capacidades de los sistemas, reducir costos y esclarecer funciones que antes eran menos evidentes. Además, esta comprensión puede llevar a una mejor relación costo-beneficio que se ajuste a las necesidades del cliente, lo que es esencial en cualquier proyecto informático o de desarrollo de software.

V. Bibliografia

OpenAI. (2021). ChatGPT: A large language model.
Andés Salcedo Vera. (2023). Taller 3-4.