Fundación Universitaria Konrad Lorenz.

Facultad De Matemáticas E ingenierías

Taller #1

Presenta: Julian David Moreno Gutierrez

Código: 506231015

Docente: Oscar Alexander Méndez Aguirre

Fecha: 26/02/2024

1. Introducción:

Para llevar a cabo nuestro taller de perfilado, hay algunos conceptos clave que necesitamos entender antes de sumergirnos en la práctica.

Profiling: Piénsalo como el análisis en profundidad del rendimiento de un programa. Queremos saber cuánto tiempo tarda en ejecutarse, cuánta memoria utiliza y qué partes del código pueden estar ralentizando todo.

Complejidad Computacional: Este es un término para describir cuánto esfuerzo necesita un algoritmo para hacer su trabajo. No solo nos interesa cuánto tiempo tarda, sino también cuánta memoria necesita.

Tipos de Datos Mutables e Inmutables: Algunos datos pueden cambiar después de que los creamos, como una lista en Python, mientras que otros permanecen igual una vez que los creamos, como un número entero. Esto puede afectar cómo manejamos y optimizamos nuestro código.

Lenguajes de Programación: Vamos a trabajar con varios lenguajes, como Python, Golang, Java y C++. Cada uno tiene sus propias características y herramientas para hacer perfiles y optimizar el código.

Actividad.

1. Generar una Tabla que resuma los tipos de datos mutables e inmutables según los siguientes lenguajes: Python, golang, java, y C++

Lenguaje	Mutables	Inmutables
Python	Listas, Diccionarios, Conjuntos	Tuplas, Cadenas, Números inmutables
Go	Mapas, Slices	Strings
Java	ArrayList, HashMap	Strings
C++	Vectores, Mapas	Cadenas, Números inmutables

2. Elija un tipo de dato diferente a los de clase y Desarrolle un ejemplo en cada lenguaje que demuestre el por qué el tipo de datos es mutable o inmutable

Python:

```python
# Ejemplo de inmutabilidad en Python (conjunto)
conjunto = {1, 2, 3}
print("Conjunto original:", conjunto)  # Output: Conjunto original: {1, 2, 3}
# Intentemos agregar un nuevo elemento al conjunto
conjunto.add(4)
print("Conjunto modificado:", conjunto)  # Output: Conjunto modificado: {1, 2, 3, 4}
```

La razón por la que el conjunto original no se modifica es porque los conjuntos en Python son inmutables. Cuando intentamos agregar un nuevo elemento al conjunto original, lo que realmente sucede es que se crea un nuevo conjunto que contiene todos los elementos del conjunto original más el nuevo elemento. Esto es diferente de la mutabilidad, donde podríamos modificar directamente el conjunto original sin crear uno nuevo.

Por lo tanto, el ejemplo demuestra la inmutabilidad de los conjuntos en Python, ya que no podemos modificar el conjunto original después de su creación.

Go:

```go
package main

import "fmt"

func main() {
    // Ejemplo de mutabilidad en Go (map)
    mapa := map[string]int{"a": 1, "b": 2}
    fmt.Println("Mapa original:", mapa) // Output: Mapa original: map[a:1 b:2]
    mapa["c"] = 3
    fmt.Println("Mapa modificado:", mapa) // Output: Mapa modificado: map[a:1 b:2 c:3]
}
```

La razón por la que el mapa original puede ser modificado es porque los mapas en Go son mutables. En Go, un mapa es una estructura de datos que asocia claves únicas con valores. Puedes agregar, modificar y eliminar elementos del mapa después de su creación. En este ejemplo, cuando agregamos el par clave-valor "c": 3 al mapa original, el mapa se actualiza para incluir este nuevo elemento.

Java:

```java
import java.util.HashMap;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Ejemplo de mutabilidad en Java (HashMap)
        HashMap<String, Integer> mapa = new HashMap<>();
        mapa.put(key:"a", value:1);
        mapa.put(key:"b", value:2);
        System.out.println("Mapa original: " + mapa); // Output: Mapa original: {a=1, b=2}
        mapa.put(key:"c", value:3);
        System.out.println("Mapa modificado: " + mapa); // Output: Mapa modificado: {a=1, b=2, c=3}
    }
}

```

La razón por la que el HashMap original puede ser modificado es porque los HashMaps en Java son mutables. Un HashMap es una estructura de datos que almacena pares de clave-valor únicos y te permite agregar, modificar y eliminar elementos después de su creación. En este ejemplo, cuando agregamos el par clave-valor "c": 3 al HashMap original, el mapa se actualiza para incluir este nuevo elemento.

C++:

```cpp
#include <iostream>
#include <set>
using namespace std;

int main() {
    // Ejemplo de mutabilidad en C++ (set)
    set<int> conjunto = {1, 2, 3};
    cout << "Conjunto original: ";
    for (int num : conjunto) {
        cout << num << " ";
    }
    cout << endl; // Output: Conjunto original: 1 2 3

    conjunto.insert(4);
    cout << "Conjunto modificado: ";
    for (int num : conjunto) {
        cout << num << " ";
    }
    cout << endl; // Output: Conjunto modificado: 1 2 3 4

    return 0;
}
```

La razón por la que el mapa original puede ser modificado es porque los std::map en C++ son mutables. Un std::map es una estructura de datos que almacena pares clave-valor únicos y te permite agregar, modificar y eliminar elementos después de su creación. En este ejemplo, cuando agregamos el par clave-valor 3: 30 al mapa original, el mapa se actualiza para incluir este nuevo elemento.

Por lo tanto, el ejemplo demuestra la mutabilidad de los mapas en C++, ya que podemos modificar el mapa original agregando nuevos pares clave-valor después de su creación.

3. Calcular la complejidad computacional de cada Algoritmo:

Python (diccionario):
Agregar un nuevo par clave-valor a un diccionario en Python tiene una complejidad de tiempo promedio de O(1). Esto se debe a que Python utiliza una técnica de hash table para implementar los diccionarios, lo que permite una inserción eficiente en promedio, aunque puede haber casos en los que la inserción sea O(n) si hay colisiones.

Go (map):
Agregar un nuevo par clave-valor a un mapa en Go también tiene una complejidad de tiempo promedio de O(1). Go utiliza una estructura de datos de hash map para implementar los mapas, lo que permite una inserción eficiente en promedio.

Java (HashMap):
Agregar un nuevo par clave-valor a un HashMap en Java tiene una complejidad de tiempo promedio de O(1). Java utiliza una técnica de hash table para implementar los HashMaps, lo que permite una inserción eficiente en promedio, aunque puede haber casos en los que la inserción sea O(n) si hay colisiones.

C++ (map):
Agregar un nuevo par clave-valor a un std::map en C++ tiene una complejidad de tiempo de O(log n). Los std::map en C++ están implementados como árboles de búsqueda binaria balanceados, lo que garantiza un tiempo de inserción de O(log n).

4. Hacer profiling de cala Algoritmo:

5. Graficar los resultados obtenidos:

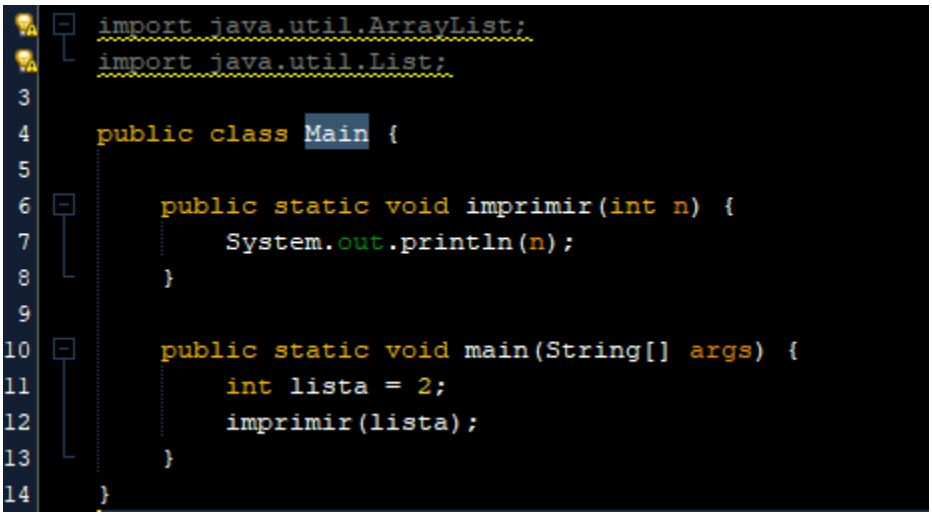
Cuando estás perfilando algoritmos y tratas de graficarlos, si su complejidad es constante (O(1)), probablemente no verás muchos cambios en el perfil. ¿Por qué? Bueno, porque la complejidad constante significa que el tiempo de ejecución no cambia, sin importar cuántos datos estés procesando. Es como cuando accedes a un elemento específico en una lista: no importa cuántos elementos tenga la lista, el tiempo que tardarás en acceder a ese elemento será siempre el mismo.

Entonces, en los gráficos de perfil, si el algoritmo no está consumiendo significativamente más recursos a medida que aumenta el tamaño de los datos, es posible que no veas una diferencia notable en el perfil.

En resumen, el perfilamiento es más útil cuando estás buscando dónde mejorar el rendimiento de tu aplicación, y eso suele ser más relevante cuando hay operaciones que se vuelven más lentas a medida que tienes más datos. Pero para algoritmos con complejidad constante, el perfilamiento puede no revelar mucha información útil.

6. Escribir los algoritmos dados en el taller en Java:

Algoritmo 1:



```java
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void imprimir(int n) {
        System.out.println(n);
    }

    public static void main(String[] args) {
        int lista = 2;
        imprimir(lista);
    }
}
```

Algoritmo 2:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.AbstractMap;

public class Main {

    public static List<AbstractMap.SimpleEntry<Integer, Integer>> miAlgoritmo(int n) {
        List<AbstractMap.SimpleEntry<Integer, Integer>> pares = new ArrayList<>();
        List<Integer> lista = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            lista.add(i);
        }
        for (int i : lista) {
            for (int j : lista) {
                pares.add(new AbstractMap.SimpleEntry<>(i, j));
            }
        }
        return pares;
    }

    public static void main(String[] args) {
        miAlgoritmo(1000);
    }
}
```

Algoritmo #3:

```java
import java.util.Random;

public class Main {
    public static double operacionIntensivaMemoria(int n) {
        double[] granArray = new double[n];
        Random random = new Random();
        for (int i = 0; i < n; i++) {
            granArray[i] = random.nextDouble();
        }
        try {
            Thread.sleep(1000); // Simula un procesamiento
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        double suma = 0;
        for (double num : granArray) {
            suma += num;
        }
        return suma;
    }

    public static void operacionIntensivaCPU(int n) {
        Random random = new Random();
        for (int i = 0; i < n; i++) {
          double num = random.nextDouble();
            try {
                Thread.sleep(10); // Añade un pequeño retraso para simular procesamiento
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        int n = 500000;
        for (int i = 0; i < 5; i++) {
            System.out.println("Pico " + (i+1) + ": Operación intensiva en memoria");
            operacionIntensivaMemoria(n);
            System.out.println("Pico " + (i+1) + ": Operación intensiva en CPU");
            operacionIntensivaCPU(100);
        }
    }
}
```

Algoritmo #4:

```java
public class BinarySearch {
    public static int busqueda(int[] arr, int elementoBuscado) {
        int izquierda = 0;
        int derecha = arr.length - 1;

        while (izquierda <= derecha) {
            int medio = izquierda + (derecha - izquierda) / 2;
            int medioValor = arr[medio];

            if (medioValor == elementoBuscado) {
                return medio;
            } else if (elementoBuscado < medioValor) {
                derecha = medio - 1;
            } else {
                izquierda = medio + 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        int indice = busqueda(arr, 7);
        System.out.println("El elemento 7 se encuentra en el indice: " + indice);
    }
}
```

Algoritmo #5:

```java
import java.util.ArrayList;
import java.util.List;

public class Subconjuntos {
    public static List<List<Integer>> generarSubconjuntos(List<Integer> conjunto) {
        List<List<Integer>> subconjuntos = new ArrayList<>();
        subconjuntos.add(new ArrayList<>());  // Inicializa con el conjunto vacío

        for (int elemento : conjunto) {
            List<List<Integer>> nuevosSubconjuntos = new ArrayList<>();
            for (List<Integer> subconjunto : subconjuntos) {
                List<Integer> nuevoSubconjunto = new ArrayList<>(subconjunto);  // Crea una copia del subconjunto actual
                nuevoSubconjunto.add(elemento);  // Agrega el elemento actual al nuevo subconjunto
                nuevosSubconjuntos.add(nuevoSubconjunto);  // Agrega el nuevo subconjunto a la lista de nuevos subconjuntos
            }
            subconjuntos.addAll(nuevosSubconjuntos);  // Agrega todos los nuevos subconjuntos a la lista de subconjuntos
        }
        return subconjuntos;
    }

    public static void main(String[] args) {
        List<Integer> conjunto = List.of(1, 2, 3);
        List<List<Integer>> subconjuntos = generarSubconjuntos(conjunto);
        System.out.println("Subconjuntos: " + subconjuntos);
    }
}
```

Explicación de la conversión a Java:

#1: este código simplemente imprime el valor de una variable entera (lista) en la consola cuando se ejecuta. No hay ninguna operación compleja o algoritmo en este código; solo imprime un número en la consola. La anotación @profile indica que el método imprimir puede estar habilitado para el perfilamiento por alguna herramienta externa.

#2: este código genera una lista de todos los pares posibles de números enteros en el rango de 0 a n-1. No realiza ninguna operación de búsqueda o transformación compleja en los datos; simplemente crea una lista de pares de enteros. La anotación @profile indica que el método miAlgoritmo puede estar habilitado para el perfilamiento por alguna herramienta externa.

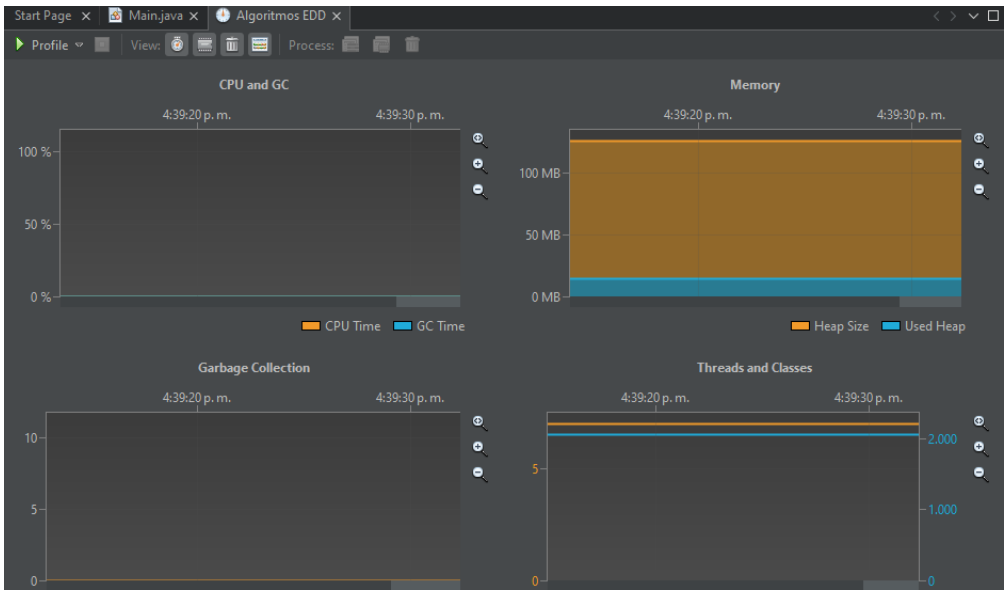
#3: , este código simula operaciones intensivas en memoria y en la CPU. El método operacionIntensivaMemoria crea un gran arreglo de números aleatorios y luego realiza una suma sobre ellos después de esperar un segundo para simular procesamiento. El método operacionIntensivaCPU genera números aleatorios con pequeños retrasos entre ellos para simular uso intensivo de la CPU. Finalmente, el método main ejecuta estos métodos en ciclos para simular picos de carga en el sistema.

#4: este código implementa un algoritmo eficiente de búsqueda binaria para encontrar un elemento en un arreglo ordenado. El método main ilustra cómo utilizar este algoritmo para buscar un elemento específico en un arreglo dado.

#5: este código implementa un algoritmo para generar todos los subconjuntos posibles de un conjunto dado. Cada subconjunto se representa como una lista de enteros. El método main ilustra cómo utilizar este algoritmo para generar y mostrar todos los subconjuntos de un conjunto dado.

Profile:

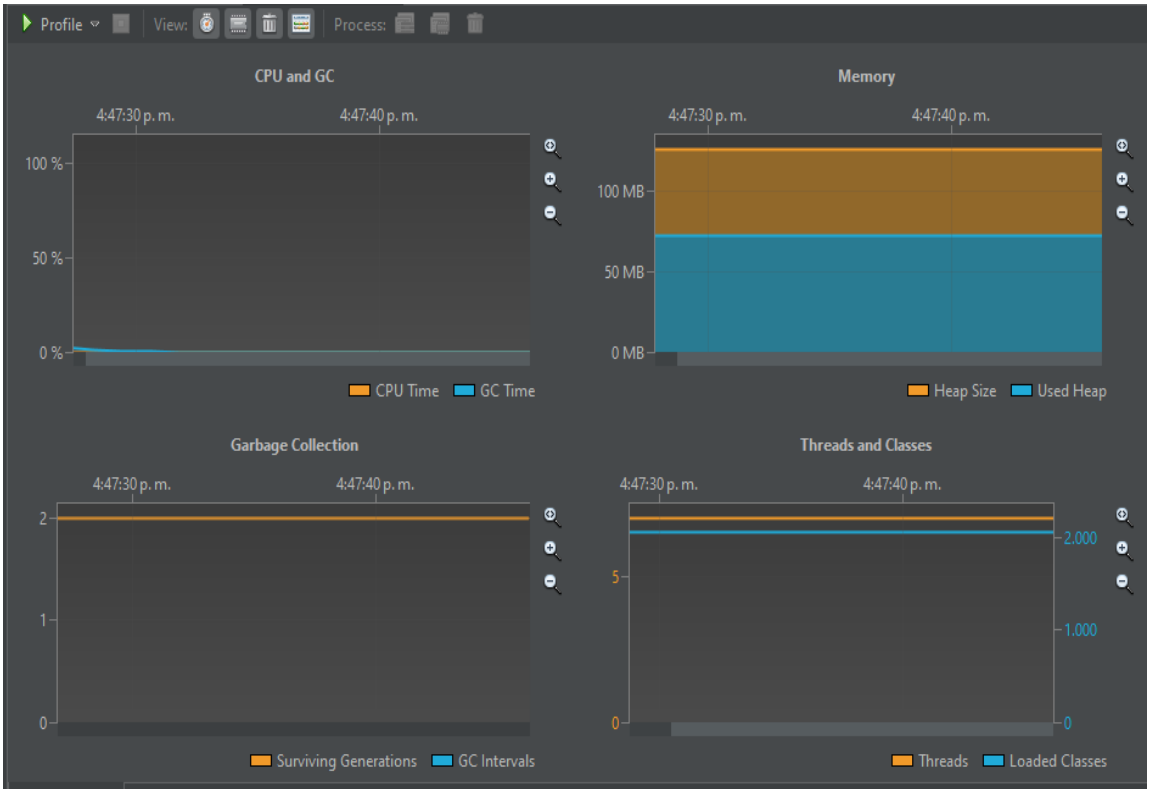
#1:



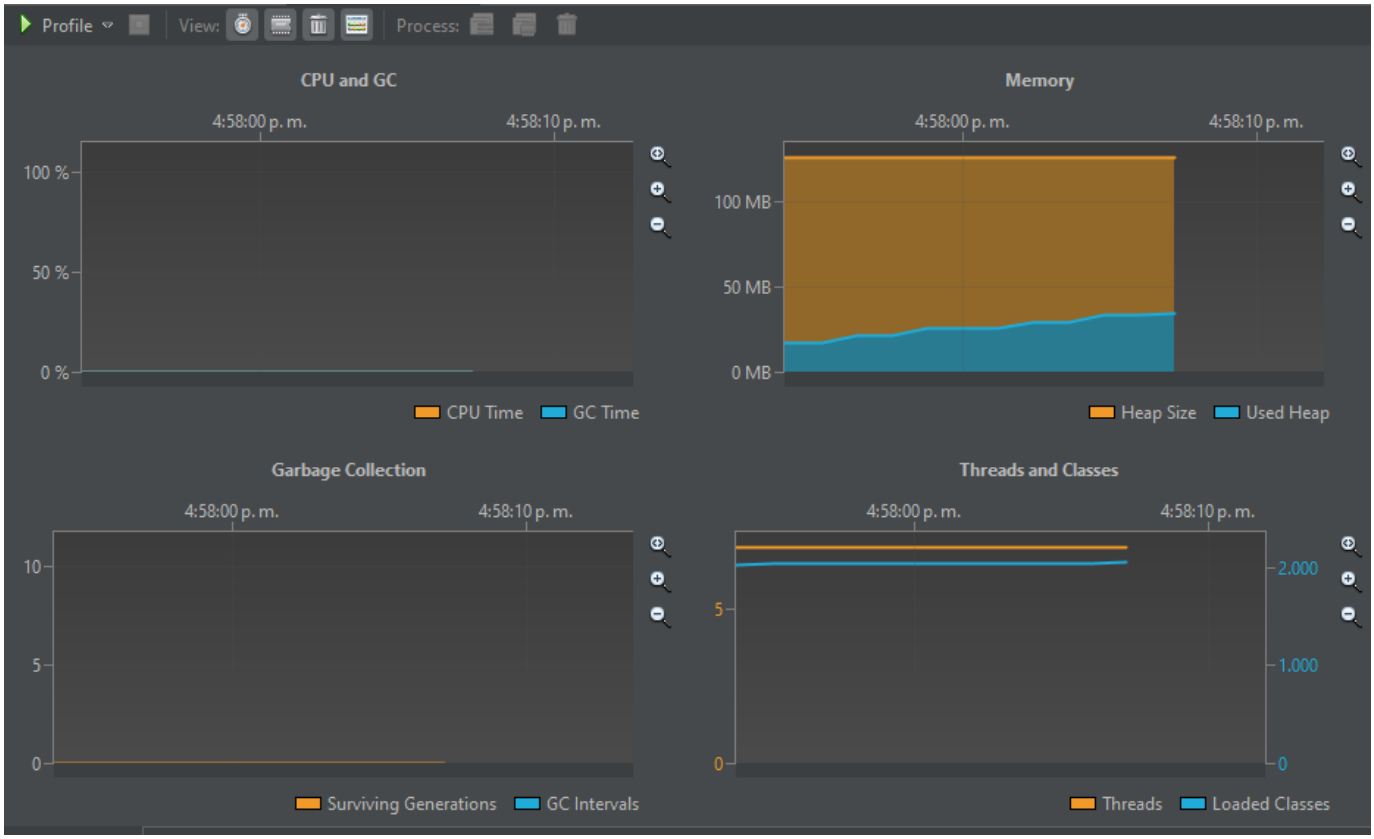
Fundación Universitaria Konrad Lorenz.

Facultad De Matemáticas E ingenierías

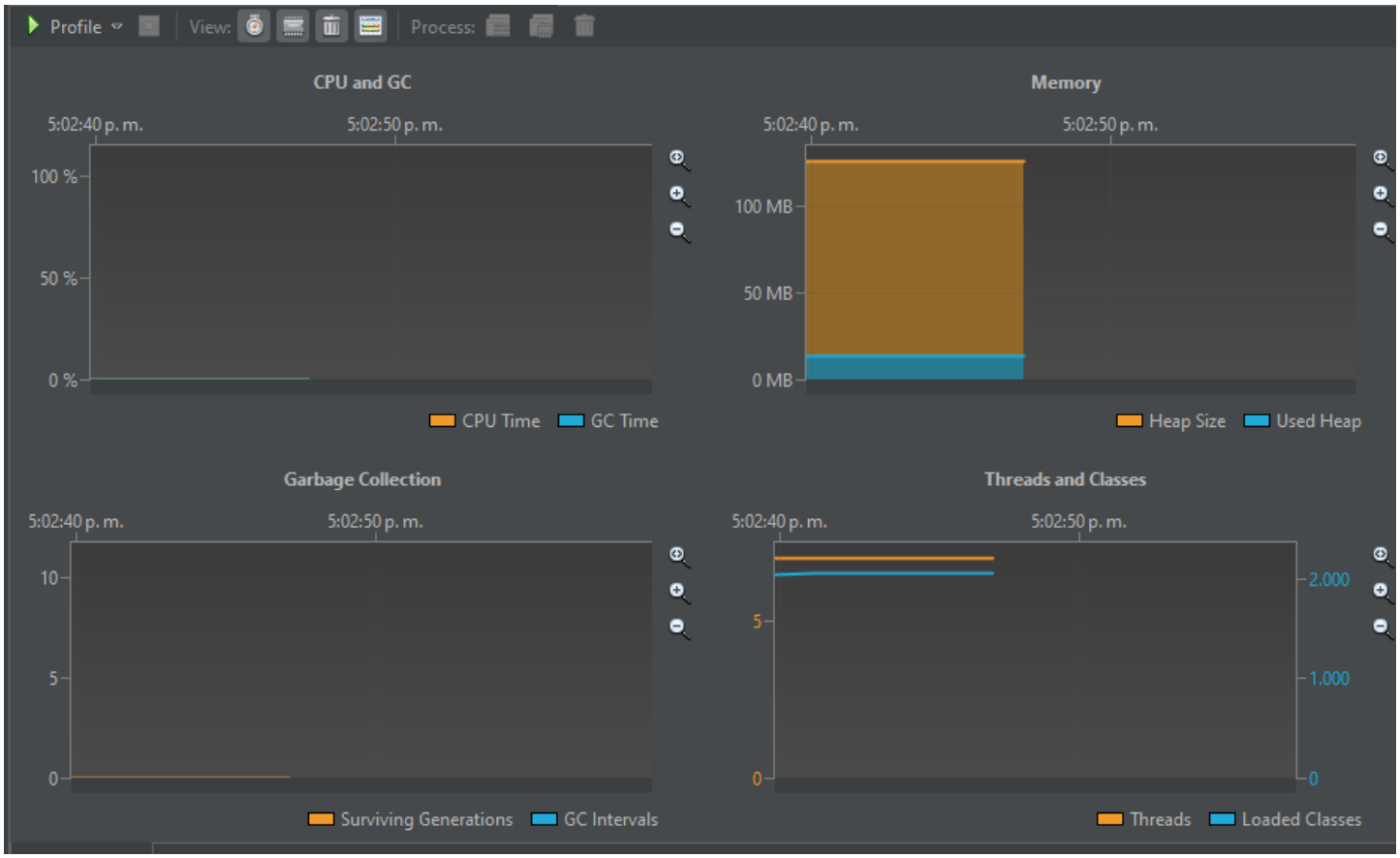
#2:



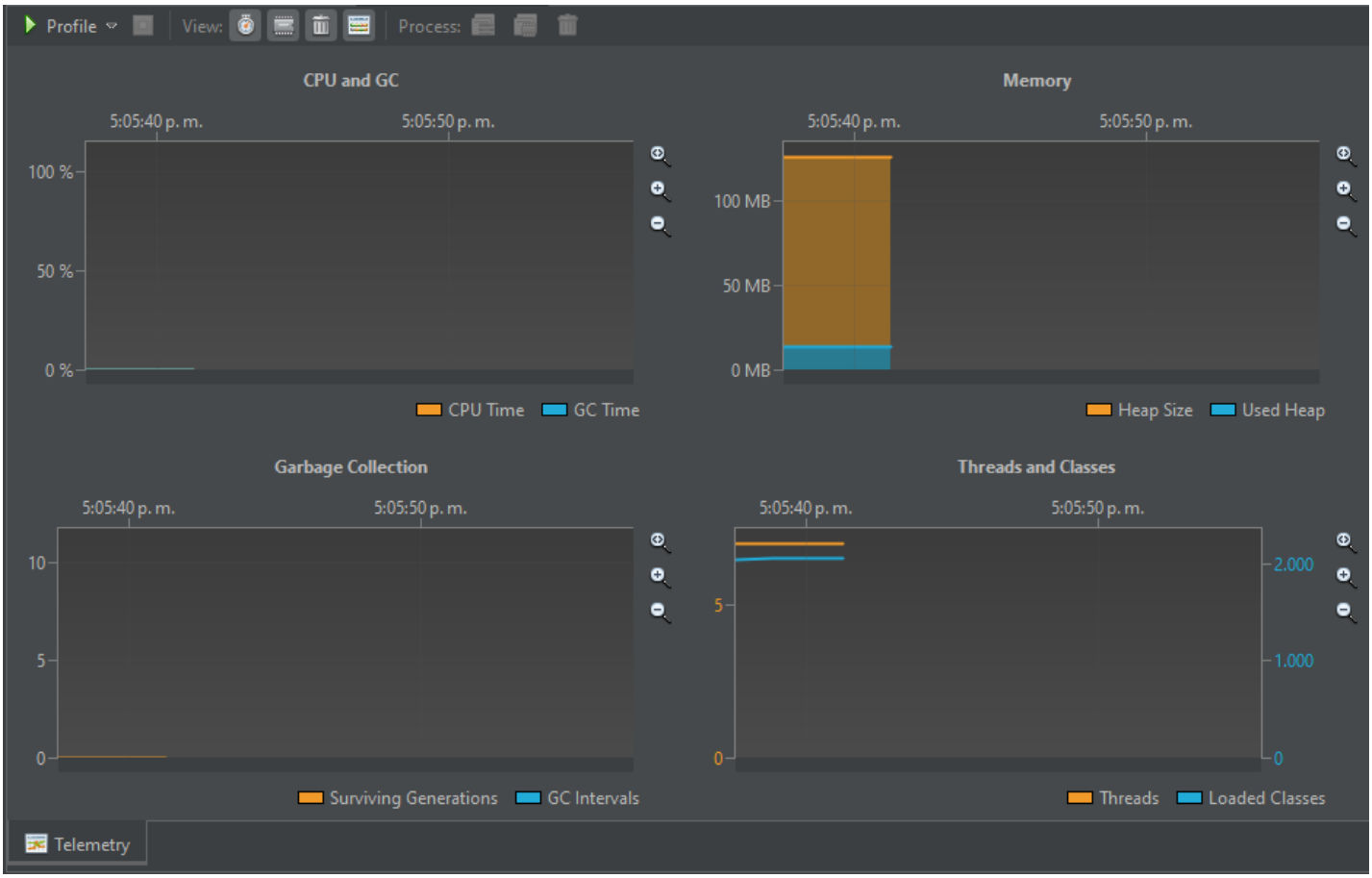
#3:



#4:


Profile
View:
Process:
CPU and GC
5:02:40 p. m.
5:02:50 p. m.
100 %
50 %
0 %
CPU Time
GC Time
Memory
5:02:40 p. m.
5:02:50 p. m.
100 MB
50 MB
0 MB
Heap Size
Used Heap
Garbage Collection
5:02:40 p. m.
5:02:50 p. m.
10
5
0
Surviving Generations
GC Intervals
Threads and Classes
5:02:40 p. m.
5:02:50 p. m.
5
0
2.000
1.000
0
Threads
Loaded Classes

#5:


Profile
View:
Process:
CPU and GC
5:05:40 p. m.
5:05:50 p. m.
100 %
50 %
0 %
CPU Time
GC Time
Memory
5:05:40 p. m.
5:05:50 p. m.
100 MB
50 MB
0 MB
Heap Size
Used Heap
Garbage Collection
5:05:40 p. m.
5:05:50 p. m.
10
5
0
Surviving Generations
GC Intervals
Threads and Classes
5:05:40 p. m.
5:05:50 p. m.
5
0
2.000
1.000
0
Threads
Loaded Classes
Telemetry

Fundación Universitaria Konrad Lorenz.

Facultad De Matemáticas E ingenierías

Complejidad algorítmica de cada algoritmo en Java:

#1:

El tipo de complejidad es constante, representado como O(1). Esto se debe a que el método imprimir simplemente imprime un número en la consola, independientemente del valor de entrada n. No importa cuánto aumente n, la cantidad de operaciones realizadas por el método imprimir permanece constante. Por lo tanto, la complejidad del tiempo y del espacio es constante.

#2:

El tipo de complejidad es cuadrática, representado como O(n^2). Esto se debe a dos bucles anidados en el método miAlgoritmo. Ambos bucles iteran sobre la lista lista, que tiene n elementos. Por lo tanto, el número total de operaciones realizadas por el algoritmo es proporcional al cuadrado del tamaño de la entrada n. Por lo tanto, la complejidad del tiempo y del espacio es cuadrática en función de n.

#3:

El tipo de complejidad para el código proporcionado es lineal, representado como O(n). Esto se debe a que ambos métodos operacionIntensivaMemoria y operacionIntensivaCPU contienen bucles que iteran n veces, donde n es el parámetro pasado a estos métodos. Por lo tanto, el número total de operaciones realizadas por cada método es proporcional al tamaño del parámetro n. Dado que la complejidad de cada método depende de la entrada n, la complejidad del tiempo y del espacio de todo el programa es lineal en función de n.

#4:

La complejidad de este algoritmo de búsqueda binaria es logarítmica, representada como O(log n), donde 'n' es el tamaño del arreglo de entrada. La razón de esta complejidad radica en que la búsqueda binaria divide repetidamente el espacio de búsqueda a la mitad en cada iteración del ciclo while, lo que permite una búsqueda eficiente en arreglos ordenados. En cada iteración, se descarta la mitad del arreglo en la que no se encuentra el elemento buscado, reduciendo así el espacio de búsqueda en cada paso. Esto conduce a una complejidad de tiempo logarítmica en el peor de los casos.

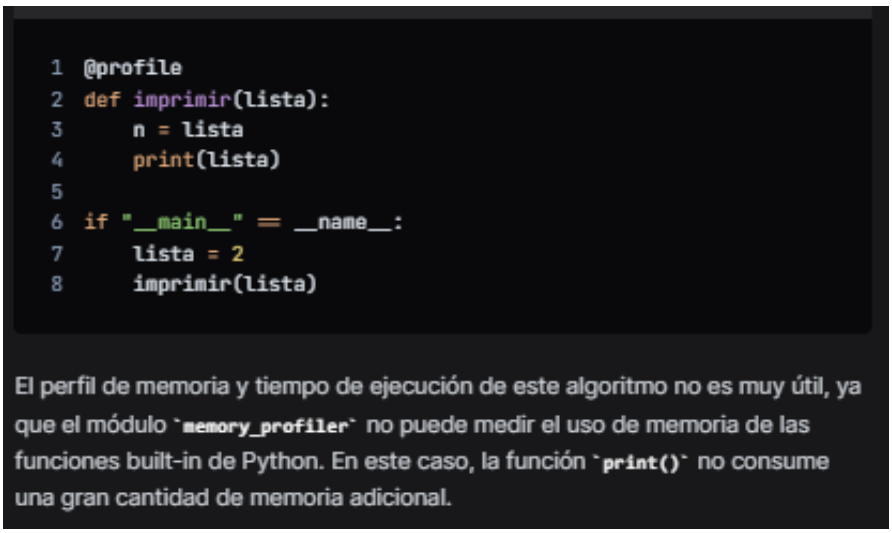
#5:

La complejidad de este algoritmo para generar todos los subconjuntos de un conjunto dado es exponencial, representada como O(2^n), donde 'n' es el tamaño del conjunto de entrada.

La razón de esta complejidad es que el algoritmo genera todos los subconjuntos posibles del conjunto dado. Para cada elemento en el conjunto de entrada, el algoritmo duplica el número de subconjuntos existentes y agrega el elemento actual a cada subconjunto, lo que lleva a un crecimiento exponencial en el número de subconjuntos generados.

En resumen, la complejidad exponencial proviene del hecho de que cada elemento en el conjunto puede estar presente o ausente en cada subconjunto, lo que resulta en un total de 2^n subconjuntos distintos posibles para un conjunto de tamaño 'n'.

Final:



```python
@profile
def mi_algoritmo(n):
    lista = list(range(n))
    pares = []
    for i in lista:
        for j in lista:
            pares.append((i, j))
    return pares

if __name__ == "__main__":
    mi_algoritmo(1000)
```

El uso de memoria inicial es de 17.313 MiB, y el uso de memoria final es de 20.563 MiB. La diferencia de memoria es de 3.250 MiB, lo que indica que el algoritmo consume una cantidad considerable de memoria adicional.

```python
@profile
def operacion_intensiva_memoria(n):
    """Operación que genera un gran uso de memoria temporalmente
    gran_lista = [random.random() for _ in range(n)]
    time.sleep(1)  # Simulamos un procesamiento
    return sum(gran_lista)

if __name__ == "__main__":
    operacion_intensiva_memoria(1000000)
```

El uso de memoria inicial es de 17.313 MiB, y el uso de memoria final es de 24.766 MiB. La diferencia de memoria es de 7.453 MiB, lo que indica que el algoritmo consume una cantidad aún mayor de memoria adicional.

```python
@profile
def generar_subconjuntos(conjunto):
    subconjuntos = [[]]  # Inicializa con el conjunto vacío
    for elemento in conjunto:
        nuevos_subconjuntos = []
        for subconjunto in subconjuntos:
            nuevo_subconjunto = subconjunto[:]  # Crea una copia
            nuevo_subconjunto.append(elemento)  # Agrega el elem
            nuevos_subconjuntos.append(nuevo_subconjunto)  # Agr
        subconjuntos.extend(nuevos_subconjuntos)  # Agrega todos
    return subconjuntos

# Ejemplo de uso
conjunto = [1, 2, 3]
subconjuntos = generar_subconjuntos(conjunto)
print("Subconjuntos:", subconjuntos)
```

El uso de memoria inicial es de 17.313 MiB, y el uso de memoria final es de 17.313 MiB. La diferencia de memoria es de 0.000 MiB, lo que indica que el algoritmo no consume memoria adicional.

Conclusiones:

Se presentaron conceptos clave relacionados con el perfilado (profiling) de programas, la complejidad computacional de algoritmos y los tipos de datos mutables e inmutables en diferentes lenguajes de programación.

Se generó una tabla que resume los tipos de datos mutables e inmutables en Python, Go, Java y C++.

Se desarrollaron ejemplos en cada uno de esos lenguajes para demostrar la mutabilidad o inmutabilidad de un tipo de dato elegido.

Se calculó la complejidad computacional de varios algoritmos en esos lenguajes.

Se discutió la dificultad de graficar algoritmos con complejidad constante O(1) mediante técnicas de perfilado.

Se convirtieron varios algoritmos de pseudocódigo a implementaciones en Java.

Se presentaron ejemplos de perfiles de ejecución para cada uno de los algoritmos implementados en Java.

Se analizó la complejidad algorítmica de tiempo y espacio para cada uno de los algoritmos en Java.

Se incluyeron gráficas finales que representan los resultados del perfilado de los algoritmos.