

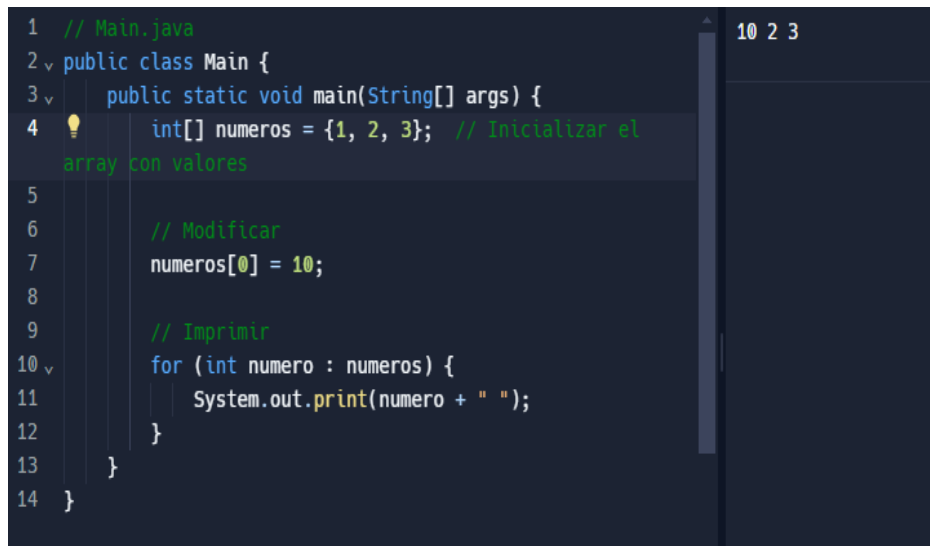
Estructura de datos

* Mutable o Inmutable,

LENGUAJE	MUTABLE	INMUTABLE
Python	Listas, Bytearray, memoryview, diccionarios, sets, clases definidas	Booleanos, complejos, enteros, float, frozenset, cadenas, tuplas, range, bytes
Golang	Slices, Maps, Pointers, channels	Int, float, bool, string, array, structs
Java	Array, ArrayList, LinkedList, HashMap, HashSet, StringBuilder, StringBuffer	String, integer, long, double, character, boolean
C++	Array, punteros, struct, close, vectores, streams	Int, float, double, char, const

Table 1: Tipos de datos en diferentes lenguajes de programación

Realizaremos una prueba muy simple de un arreglo en cada uno de los lenguajes, mirando como puede ser mutable o inmutable, también podemos ver las pequeñas diferencias en lenguajes.



```

1 // Main.java
2 public class Main {
3     public static void main(String[] args) {
4         int[] numeros = {1, 2, 3}; // Inicializar el
           array con valores
5
6         // Modificar
7         numeros[0] = 10;
8
9         // Imprimir
10        for (int numero : numeros) {
11            System.out.print(numero + " ");
12        }
13    }
14 }

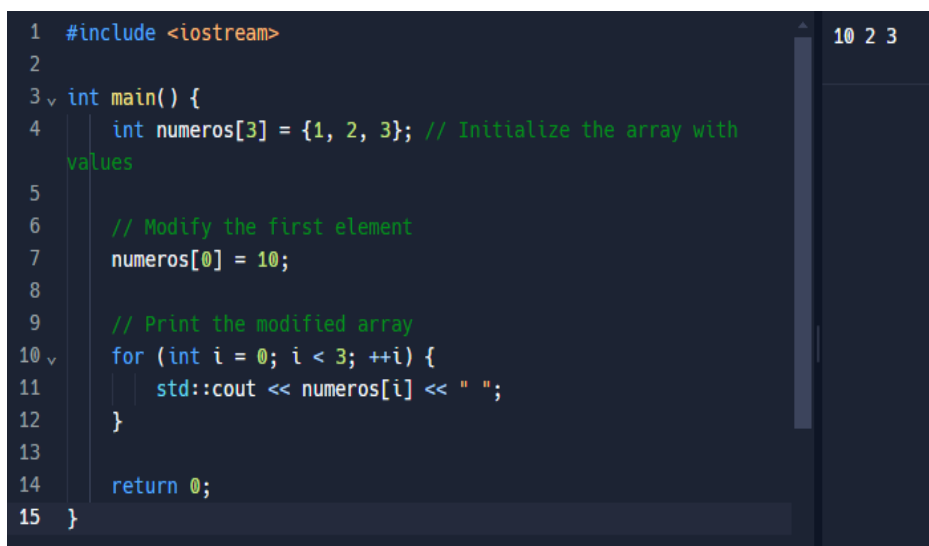
```

10 2 3

Figure 1: Java

Podemos ver que en este ejemplo en Java presenta un dato mutable debido que despues de que se crea el arreglo (Array) podemos cambiarle los valores, definiendo otro valor al espacio [0] y le ubicamos otro numero natural. Aunque la variable "numeros" que referencia el arreglo se declara como int[] (Que el int es inmutable), el array en

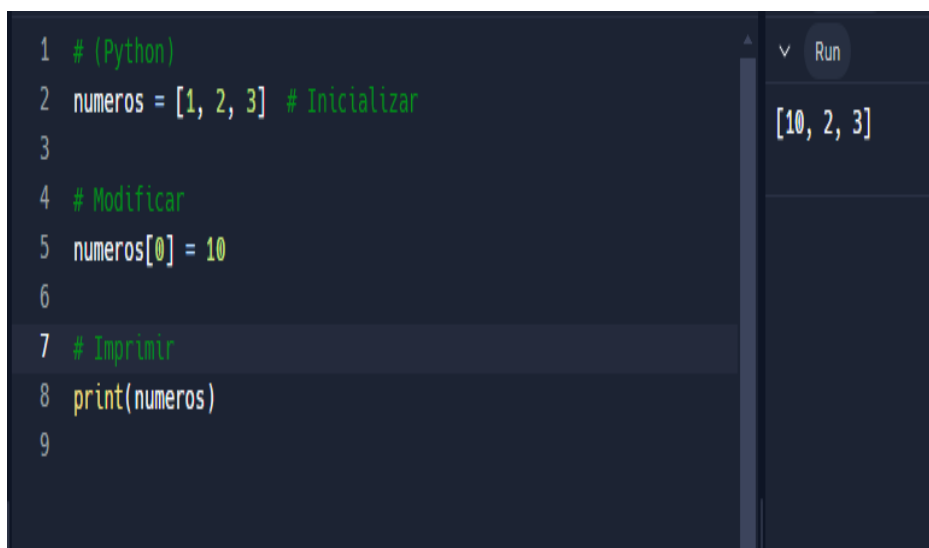
sí mismo es mutable. Solo la referencia a la ubicación del array en la memoria se almacena en numeros, pero los valores dentro del array se pueden modificar individualmente.



```
1 #include <iostream>
2
3 int main() {
4     int numeros[3] = {1, 2, 3}; // Initialize the array with
    values
5
6     // Modify the first element
7     numeros[0] = 10;
8
9     // Print the modified array
10    for (int i = 0; i < 3; ++i) {
11        std::cout << numeros[i] << " ";
12    }
13
14    return 0;
15 }
```

The screenshot shows a C++ code editor with a dark theme. The code defines an array 'numeros' of size 3, initializes it with {1, 2, 3}, changes the first element to 10, and prints the array. The output '10 2 3' is visible in the top right corner.

Figure 2: C++



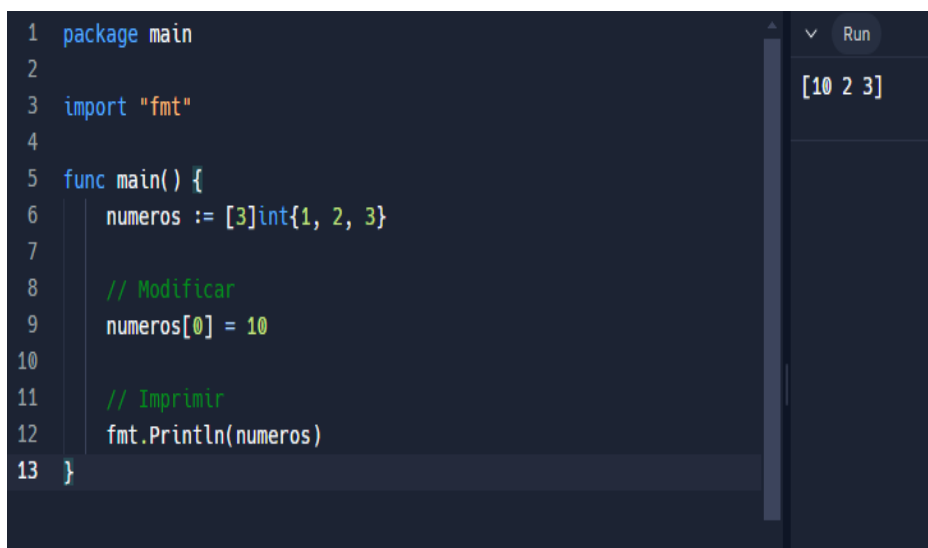
```
1 # (Python)
2 numeros = [1, 2, 3] # Inicializar
3
4 # Modificar
5 numeros[0] = 10
6
7 # Imprimir
8 print(numeros)
9
```

The screenshot shows a Python code editor with a dark theme. The code creates a list 'numeros' with [1, 2, 3], changes the first element to 10, and prints it. The output '[10, 2, 3]' is visible in the top right corner.

Figure 3: Python

En ambos casos de los ejemplos anteriores podemos ver arreglos mutables, ya que la inicializamos con numeros enteros y luego le cambiamos un espacio de memoria sustituyendo el valor del entero. En general, los arreglos

en muchos lenguajes de programación es mutable; Esto significa que puedes cambiar los valores de los elementos individuales del array después de su creación.



```

1 package main
2
3 import "fmt"
4
5 func main() {
6     numeros := [3]int{1, 2, 3}
7
8     // Modificar
9     numeros[0] = 10
10
11    // Imprimir
12    fmt.Println(numeros)
13 }

```

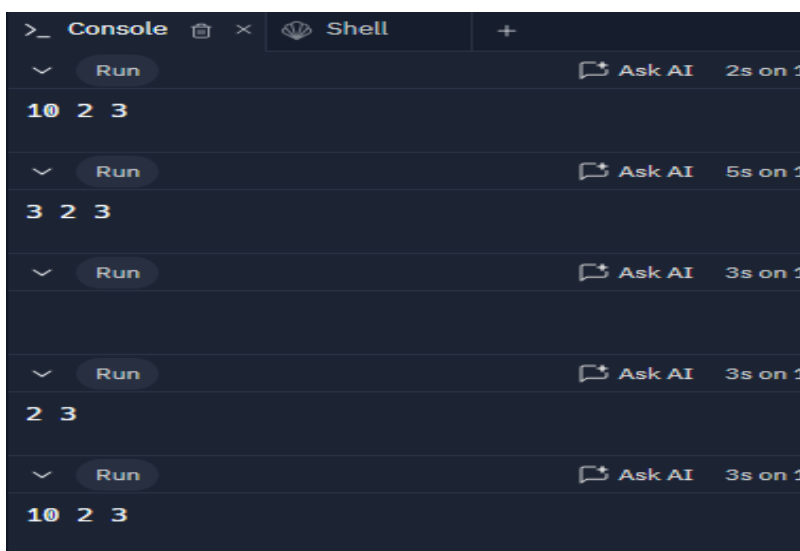
Run [10 2 3]

Figure 4: Go

COMPLEJIDAD COMPUTACIONAL

- La complejidad computacional depende de muchas cosas, como el computador en el que se corre y sus características. También depende de lo complejo que sea el código y cuántas veces recorrer o repite cada línea y cuanto le tarde en hacerlo, esto depende de la cantidad de datos que tenga la estructura que deba recorrer y el método que se utilice para hacerlo.

Hicimos una prueba en C++ donde cambiamos el dato de diferentes cosas como el número que modificamos y lo que deseamos que imprimamos o no. Estas pruebas nos dieron distintos tiempos de ejecución, en algunos es más pronunciado y en otros menor.



```

>_ Console Shell +
  Run Ask AI 2s on 1
10 2 3
  Run Ask AI 5s on 1
3 2 3
  Run Ask AI 3s on 1
2 3
  Run Ask AI 3s on 1
10 2 3

```

Figure 5: C++

- En java identificamos la cantidad de veces que repite el algoritmo y su variación si los adtos cambian, como poniendo una matriz de 4 en vez de 3, cambiando el dato al que modificamos, hasta logramos poner una letra n para ver cuanto tarda en copilar un error.

```

Run
10 2 3
Run
9 2 3
Run
10 2 3 4
Run
1 2 3 10
./src/main/java/Main.java:7: error: cannot find symbol
    numeros[3] = n;
                  ^
symbol:   variable n
location: class Main
1 error

```

Figure 6: Java

- Teniendo estas pruebas de escritorio vamos a calcular la complejidad algoritmica de los ejemplos que puso el tutor, ya que mis ejemplos son de tipo lineal, tienen el mismo tipo de complejidad algoritmica ya que tienen la misma longitud en el dato.

La [n] Es lo que puede variar, esto hace que nuestro algoritmo tenga una mayor o menor complejidad, eso es lo que debo identificar en cada uno de los ejemplos propuestos en el taller y debemos identificar el metodo que utilizamos, la forma en la que recorremos. Vemos que $O(1)$ que es un metodo constante, $O(n)$ es el lineal y $O(n^2)$ es un exponencial, el $O(\log n)$ es el logaritmico a medida de que aumenta disminuye el tiempo.

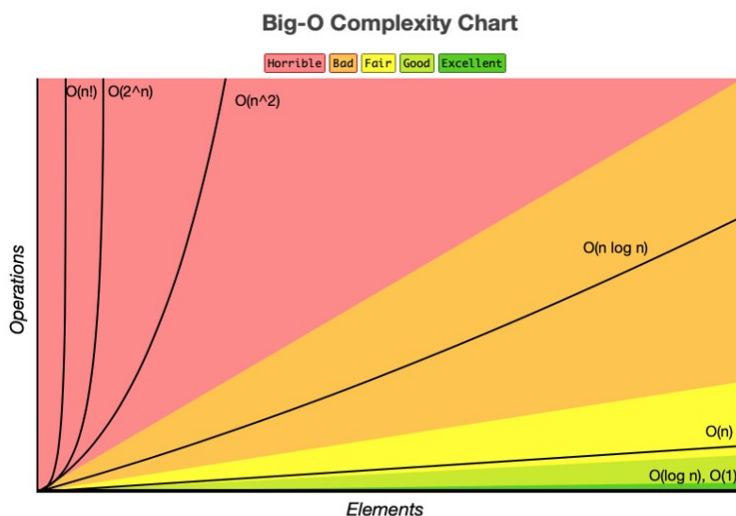


Figure 7: Complejidad, Tiempo

- Tenemos el primero ejemplo, que nos muestra un inicio de una lista que vale n , despues imprimimos, estas lineas de codigo son constantes, por ende suman 0 a la complejidad, despues tenemos un "condicional" que nos dice que si el archivo se llama igual que le nombre, entonces lista vale 2 e imprime, como la complejidad de la inicialización de lista vale $[0]$ entonces todo este algoritmo es constante. Big-O es $O(n)$.

Algoritmo 1

```
from memory_profiler import profile

@profile
def imprimir(lista):
    n = lista
    print(lista)

if "__main__" == __name__:
    lista = 2
    imprimir(lista)
```

Figure 8: Algoritmo 1

- En el segundo algoritmo tenemos una complejidad más avanzada ya que tenemos una lista vacia que se llama pares, otra lista que esta en un rango que empieza de (0) hasta (n), creamos 2 ciclos, uno adidado dentro del otro, donde por cada (i) genera unos pares con (j), los guarda en la lista vacia de (pares), despues imprime esa lista que estaba vacia pero ahora contiene toda la combinación de pares que se crearon en el for. entonces tiene una complejidad algoritmica de $O(n^2)$.

Algoritmo 2

```
from memory_profiler import profile

@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)
```

Figure 9: Algoritmo 2

- En el algoritmo 3 nos lleva al máximo el cpu y la memoria ram, generando un tope, despues hace una ciclo que se repite simultaneamente 5 veces para llamar a los metodos, haciendo que la memoria junto con el cpu (procesamiento) lleguen a su máximo poniendo un valor de (n) de 500000. Esto haria que su Big-O fuera de lineal $n(0)$, si el rango fuera del ciclo de 5 repeticiones fuera de (n) seria una Big-O de $O(n^2)$, porque seria un ciclo anidado a otro ciclo.

```
import random
import time
from memory_profiler import profile

@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)

@profile
def operacion_intensiva_cpu(n):
    """Operación que consume tiempo de CPU."""
    contador = 0
    for _ in range(n):
        contador += random.random()
    time.sleep(0.01) # Añade un pequeño retardo para simular procesamiento

def main():
    n = 500000
    for i in range(5):
        print(f"Pico {i+1}: Operación intensiva en memoria")
        operacion_intensiva_memoria(n)
        print(f"Pico {i+1}: Operación intensiva en CPU")
        operacion_intensiva_cpu(100)

if __name__ == "__main__":
    main()
```

Figure 10: Algoritmo 3

- En el algoritmo 4 tenemos un codigo que nos ayuda a encontrar un datos que necesitamos, primero hacemos un metodo que utiliza un ciclo condicional, para buscar el punto medio y compararlo con el dato que necesitamos y observamos cuanto nos toma encontrar el dato con estas operaciones logaritmicas. Entonces, su Big-O es de $O(\log n)$

Algoritmo 4

```
def busqueda(arr, elemento_buscado):
    izquierda, derecha = 0, len(arr) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        medio_valor = arr[medio]

        if medio_valor == elemento_buscado:
            return medio
        elif elemento_buscado < medio_valor:
            derecha = medio - 1
        else:
            izquierda = medio + 1
    return -1

indice = busqueda([1, 2, 3, 4, 5, 6, 7, 8, 9], 7)
```

Figure 11: Algoritmo 4

- Por ultimo, el algoritmo 5 nos muestra una serie de lineas que genera la serie de subconjuntos de un conjunto dado y lo agregan al conjunto que inicialmete esta vacio, por ende su complejidad es $O(2^n)$.

Algoritmo 5

```
def generar_subconjuntos(conjunto):
    subconjuntos = [[]] # Inicializa con el conjunto vacio
    for elemento in conjunto:
        nuevos_subconjuntos = []
        for subconjunto in subconjuntos:
            nuevo_subconjunto = subconjunto[:] # Crea una copia del subconjunto actual
            nuevo_subconjunto.append(elemento) # Agrega el elemento actual al nuevo subconjunto
            nuevos_subconjuntos.append(nuevo_subconjunto) # Agrega el nuevo subconjunto a la lista
        subconjuntos.extend(nuevos_subconjuntos) # Agrega todos los nuevos subconjuntos a la lista
    return subconjuntos

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

Figure 12: Algoritmo 5

PROFILING

En esta parte del taller haremos algo que se conoce como profiling, que tiene como objetivo principal observar su tiempo en memoria y en Cpu al ejecutar los codigos ya antes visto, como ya sabemos su complejidad podriamos hacernos una idea de la diferencia grafica de un ejemplo al otro.

1 PYTHON:

Utilizaremos colab para desarrolar este punto, previamente intalamos un paquete para que nos muestre lo que necesitamos ver.

1.1 Algoritmo 1

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el primero algoritmo.

```
PROGRAM: <cell line: 2>
0,020 <cell line: 2>
0,012 <module>
  27 frames hidden (memory_profiler, multiprocessing, <built-in>...)
0,011 wrapper
  33 frames hidden (memory_profiler, contextlib, debugpy...)
0,001 imprimir
  0,001 OutStream.write
    3 frames hidden (ipykernel, threading)
0,002 loads
0,001 stat

mprof: Sampling memory every 0.1s
running new process
2
Filename: /content/algoritmo1.py

Line #   Mem usage   Increment   Occurrences   Line Contents
=====
2        36.9 MiB    36.9 MiB         1   @profile
3                               1   def imprimir(lista):
4        36.9 MiB     0.0 MiB         1       n = lista
5        36.9 MiB     0.0 MiB         1       print(lista)
```

Figure 13:

1.2 Algoritmo 2

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el segundo algoritmo.

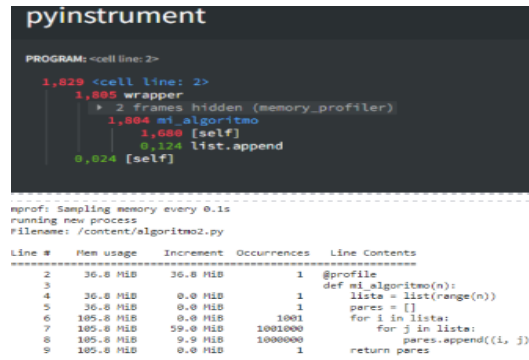


Figure 14:

1.3 Algoritmo 3

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el tercero algoritmo.

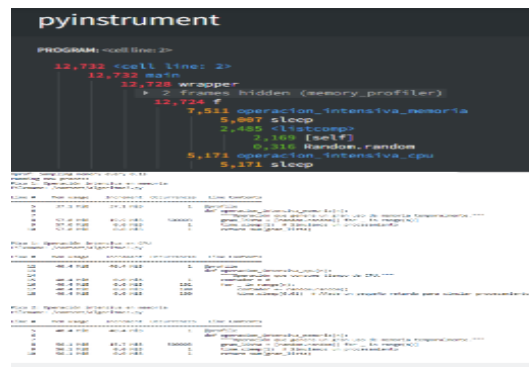


Figure 15:

1.4 Algoritmo 4

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el cuarto algoritmo.

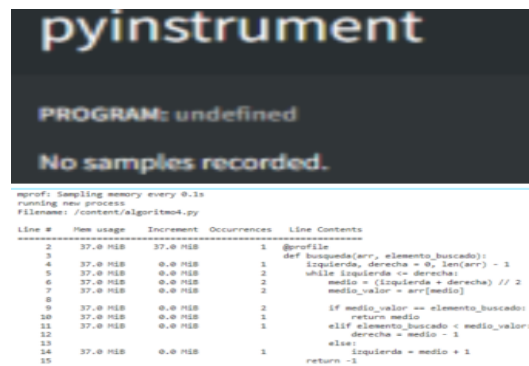


Figure 16:

1.5 Algoritmo 5

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el quinto algoritmo.

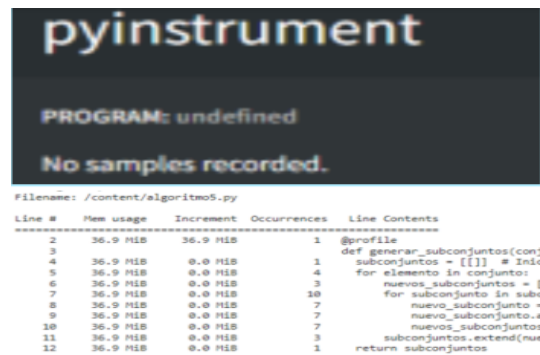


Figure 17:

1.6 Graficas

Con toda la información que recolectamos anteriormente, podemos crear una grafica que nos muestre visualmente como cambia la interacción en cada uno de los algoritmos y su complejidad contra el tiempo de ejecución. En colab hubo variaciones pero las graficas casi no se movieron.

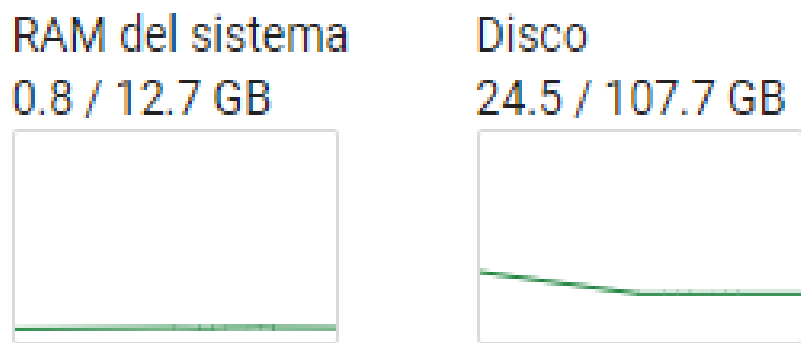


Figure 18: Grafica 1

2 JAVA:

Utilizaremos Net para hacer la segunda prueba y observar su variación tanto en Cpu como en memoria. Adaptamos el código que nos da el tutor a Java y pondremos los resultados que nos dio el profiling basandonos en lo enseñado en clase.

2.1 Algoritmo 1

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el primero algoritmo.

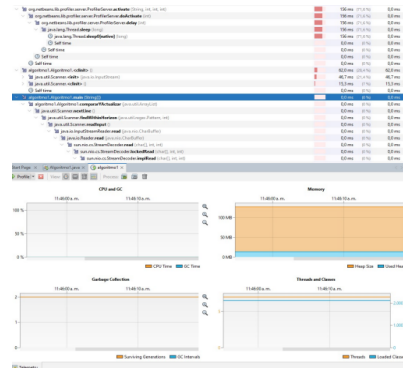


Figure 19:

2.2 Algoritmo 2

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el segundo algoritmo.

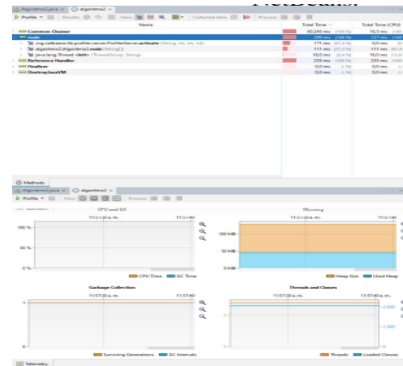


Figure 20:

2.3 Algoritmo 3

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el tercero algoritmo.

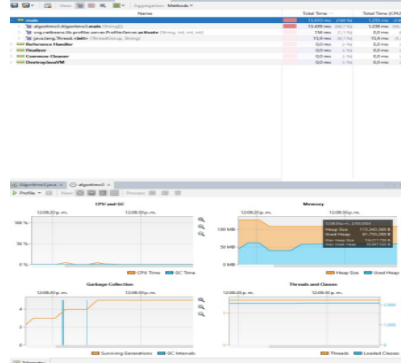


Figure 21:

2.4 Algoritmo 4

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el cuarto algoritmo.



Figure 22:

2.5 Algoritmo 5

En esta sección observaremos su variación en mermoria y en cpu al ejecutar el quinto algoritmo.



Figure 23:

2.6 Graficas

Con toda la información que recolectamos anteriormente, podemos crear una grafica que nos muestre visualmente como cambia la interacción en cada uno de los algoritmos y su complejidad contra el tiempo de ejecución.

Table 2: Comparación de complejidad en memoria

Algoritmo	Python)	Java)
Algoritmo 1	37	10
Algoritmo 2	37	49
Algoritmo 3	56.2	62
Algoritmo 4	37	11
Algoritmo 5	37	12

Aqui podemos ver una comparación del mismo codigo en 2 lenguajes distinto, donde a simple vista uno es más complejo que el otro. Hablando de mermoria utilizada para ejecutar el codigo podemos afirmar que Java utiliza más memoria en todos los algoritmos.

Table 3: Comparison en CPU (Timepo)

Algoritmo	Python	Java
Algoritmo 1	0.03 s	25.9ms
Algoritmo 2	1.8 s	1,8 ms
Algoritmo 3	2.7 s	2,7 ms
Algoritmo 4	0.0002 s	0.15 ms
Algoritmo 5	0.0003 s	0.3 ms

Camparamos el tiempo que se demora en ejecutar el codigo y como conclusión podemos decir Java es más rapido para ejecutarlos que python, pero en el algoritmo 3 son lentos los dos.