Análisis de Procesos Recursivos e Iterativos en Scala

Fundamentos de Programación Funcional y Concurrente

Ingeniería de Sistemas, Universidad del Valle

Esteban Samuel Cordoba Narvaez 2370976

Brigitte Vanesa Chavez Riascos 2510107

11 de septiembre de 2025

Resumen: *Este informe documenta la solución a dos problemas de programación recursiva en Scala: encontrar el máximo de una lista de enteros y resolver las Torres de Hanoi. A través de la implementación de funciones con procesos iterativos y de recursión en árbol, se analiza el comportamiento de cada algoritmo. El informe incluye una argumentación formal sobre la corrección del código, respaldada por un conjunto de pruebas diseñadas para validar las soluciones. Se concluye con una reflexión sobre los diferentes tipos de procesos computacionales y su impacto en la eficiencia del programa.*

1. Informe de Procesos

a. Máximo de una Lista

Ambas funciones, `maxIt` y `maxLin`, generan un proceso iterativo. Aunque la implementación utiliza la recursión, la forma en que el cálculo avanza es similar a un bucle.

Un proceso se considera **iterativo** cuando la información necesaria para el siguiente estado se encuentra completamente en los argumentos de la función, sin la necesidad de que la pila de llamadas crezca.

- **maxIt:** implementa explícitamente esa idea utilizando un **-acumulador-**. La función auxiliar loop toma la lista restante y el máximo encontrado hasta el momento. En cada llamada, el **-acumulador-** se actualiza y se pasa a la siguiente llamada. Al llegar al

caso base (lista vacía), el valor del -**acumulador-** es el resultado final, lo que evita que la pila se llene.

- **maxLin:** también genera un proceso iterativo, implementando **recursión de cola** (tail recursion), un tipo especial de recursión lineal en la que la llamada recursiva es la última operación en la función.

b. Torres de Hanoi

- **movsTorresHanoi:** Se genera un tipo de proceso recursivo, lineal (cola no optimizada): cada llamada depende de una única llamada recursiva con $n - 1$.

- **torresHanoi:** Esta función devuelve la lista de movimientos necesarios para resolver el problema con n discos.

  Este genera un proceso recursivo de tipo árbol binario, con complejidad O2^n porque se duplican las llamadas en cada nivel.

  El proceso que se genera es una doble recursión porque la función se llama a sí misma dos veces en cada paso para mover $n - 1$ discos a la torre auxiliar y luego a la torre destino.

2. Corrección

a. maxLin y maxIt

La corrección de estas funciones se demuestra por inducción estructural sobre la lista de entrada.

i. **Caso Base:** Para una lista de un solo elemento, la función es trivialmente correcta.

ii. **Paso Inductivo:** Si la función es correcta para una lista de tamaño n, entonces para una lista de tamaño $n + 1$, la función toma el primer elemento y lo compara con el máximo del subproblema de tamaño n (la cola de la lista), que por nuestra hipótesis inductiva es correcto. El valor retornado será el máximo de la lista completa.

b. movsTorresHanoi

La corrección de esta función se basa en la **inducción matemática** sobre el número de discos n.

i. **Caso Base:** Para $n = 0$, la función retorna 0, que es correcto.

ii. **Paso Inductivo:** Para resolver el problema con n discos, se requiere resolver dos subproblemas para $n - 1$ discos y un movimiento

adicional. Así, el número total de movimientos es $2 \times movsTorresHanoi(n-1) + 1$. La implementación recursiva de la función refleja esta relación de recurrencia, garantizando su corrección.

c. torresHanoi

i. **Caso Base:** Para $n = 0$, la función retorna una lista vacía de movimientos, que es la solución correcta.

ii. **Paso inductivo:** Para resolver el problema para n discos, la función descompone el problema en tres partes:

Una llamada recursiva para $n-1$ discos	Un movimiento individual	Otra llamada recursiva para $n-1$ discos

La lista de movimientos se **construye concatenando** los resultados de estas tres etapas, lo que es una representación fiel y correcta del algoritmo de las Torres de Hanoi

3. Casos de Prueba

Se diseñaron un conjunto de pruebas en el archivo **`Pruebas.sc`** para validar la corrección de las funciones en diferentes escenarios.

a. maxLin y maxIt

Se prueban con una lista estándar `(List(3, 20, 5, 4))`, una con números negativos `(List(-1, -5, -2))`, una lista de un solo elemento y una lista con el máximo al final. Los resultados obtenidos `(20, -1, 100, 10` respectivamente) coinciden con los valores esperados.

b. movsTorresHanoi

Se validan los primeros 5 casos de la serie (n de 1 a 5), para los cuales los resultados son conocidos `(1, 3, 7, 15, 31)`. Se incluye una prueba para 64 discos para verificar el manejo de números grandes con BigInt.

c. torresHanoi

Se verifican las secuencias de movimientos para $1, 2, y\ 3$ discos, asegurando que las listas de tuplas producidas por la función `(List((1, 3)), List((1, 2), (1, 3), (2, 3))`, etc.) coincidan con las soluciones esperadas.

Este conjunto de pruebas nos son suficientes para demostrar la corrección de las funciones en los casos clave, cubriendo los casos **base**, casos **típicos** y algunos casos **especiales**. Aunque una validación exhaustiva requeriría pruebas aleatorias y

una cobertura de código más profunda, los casos diseñados son adecuados para demostrar la validez de los algoritmos en el contexto de este taller.

4. Conclusiones

El desarrollo de esta práctica nos permitió comprender más a profundidad la diferencia entre una función recursiva y el proceso computacional que esta genera. Aunque `maxLin` y `maxIt` tienen una implementación similar en apariencia, su eficiencia radica en que ambas generan un **proceso iterativo optimizado**. Por otro lado, las funciones de las Torres de Hanoi son un ejemplo claro de un proceso recursivo en árbol, lo que permite entender por qué este tipo de problemas crecen de manera exponencial. La experiencia nos ha enseñado la importancia de no solo escribir **código funcional**, sino también de analizar el tipo de procesos que se generan para asegurar la **eficiencia** y evitar problemas de rendimiento en escenarios reales.