INFORME TALLER Nº 1 PFC

INTEGRANTES:

Jose Daniel Ceballos Osorio - 2266306

Jesus Estenillos Loaiza Serrano -

Sebastian Ceron Orozco - 2266148

Punto 1.

Informe de procesos.

Máximo de una lista de enteros

Para hacer este código nos enfocamos en realizarlo de varias formas para llevarlo a un caso de pruebas más extenuante, y estos son; el fastalgorithm el peasantalgorithm y por último el Splitalgorithm

Al ejecutar el fastalgorithm se crea una lista aleatoria de 1 a 100, para luego crear una nueva lista con el mismo rango fig(1).

Una vez creada la lista el for verifica que los rangos de las listas sean correctos, y si lo son procede a realizar la condición de prueba f(2).

- Si **la condición es verdadera** (el método multiply devuelve el mismo resultado que listA(i) * listB(i)), la prueba continúa con el siguiente número en el ciclo.

- Si **la condición es falsa** (el resultado del método multiply no coincide con la multiplicación estándar), el assert lanzará una excepción, lo que significa que la prueba ha fallado.

Pruebas.

En este caso tendremos como ejemplo dos listas las cuales cada uno tendrán números aleatorios desde el 1 hasta el 10.

Se crean las dos listas:

```
> fastAlgorithm = FastAlgorithm@1922
> listA = $colon$colon@1944 "List(9, 7, 1, 4, 8, 2, 10, 5, 3, 6)"
> listB = $colon$colon@1946 "List(5, 1, 10, 7, 4, 2, 6, 9, 8, 3)"
```

Fig1.

Se cumple la condición para entrar al ciclo y multiplica los índices A(0) y B(0)

```
Local
> fastAlgorithm$1 = FastAlgorithm@1922
> listA$1 = $colon$colon@1944 "List(9, 7, 1, 4, 8, 2, 10, 5, 3, 6)"
> listB$1 = $colon$colon@1946 "List(5, 1, 10, 7, 4, 2, 6, 9, 8, 3)"
  i = 0
```

Fig 2.

Se ejecuta los índices desde 0 hasta 9 en donde la ultima recursión suelta el numero mas grande de las listas, en este caso el 10.

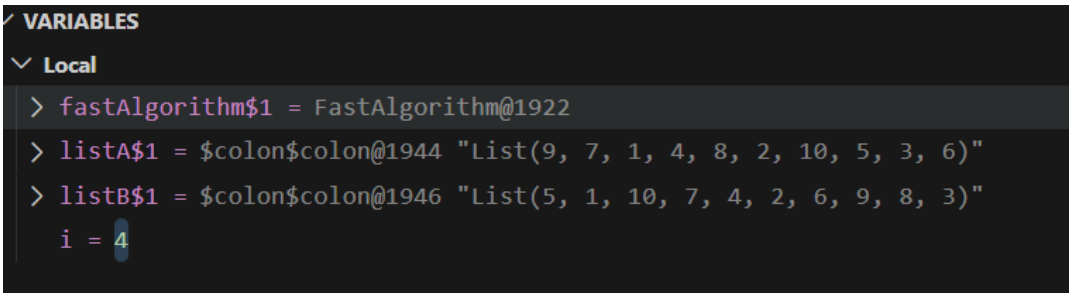


Fig 3.

Fig 4.

Conclusión.

Si el método multiply está implementado usando recursión, la recursión sería clave para lograr que este algoritmo sea eficiente y escalable, especialmente si se enfrenta a problemas de multiplicación grandes o complejos. La recursión permite dividir el problema en tareas más pequeñas, optimizando la ejecución y manteniendo la claridad en la implementación del código.

Punto 2.

Informe de procesos.

Torres de Hanoi.

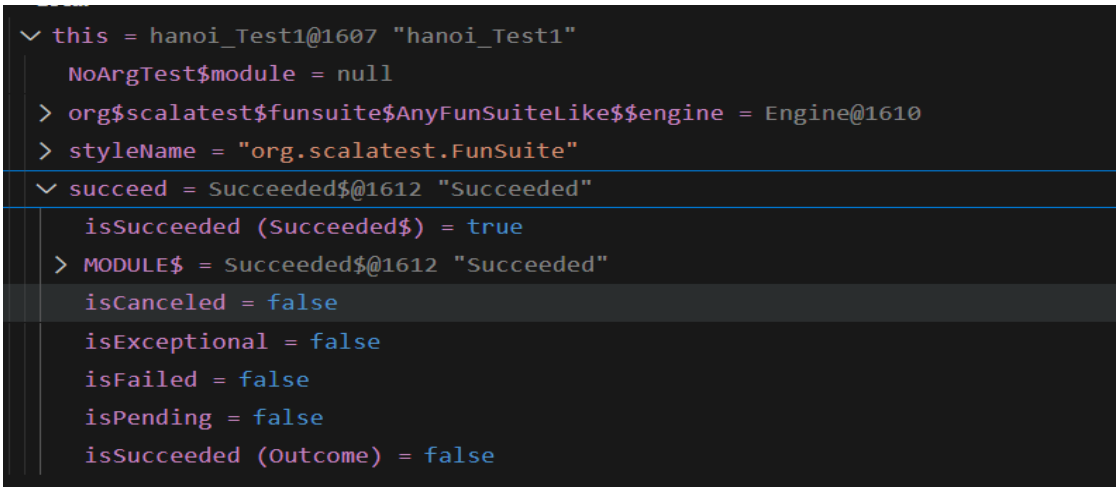
Este código define un objeto llamado torres_hanoi que resuelve el problema de las **Torres de Hanói** en Scala, proporcionando dos funciones principales:

1. **movsTorresHanoi(n: Int): BigInt**: Calcula el número mínimo de movimientos necesarios para resolver el problema con **n discos**.
2. **torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)]**: Devuelve una lista con los movimientos necesarios para mover los discos de un poste a otro, siguiendo las reglas de las Torres de Hanói.

Pruebas.

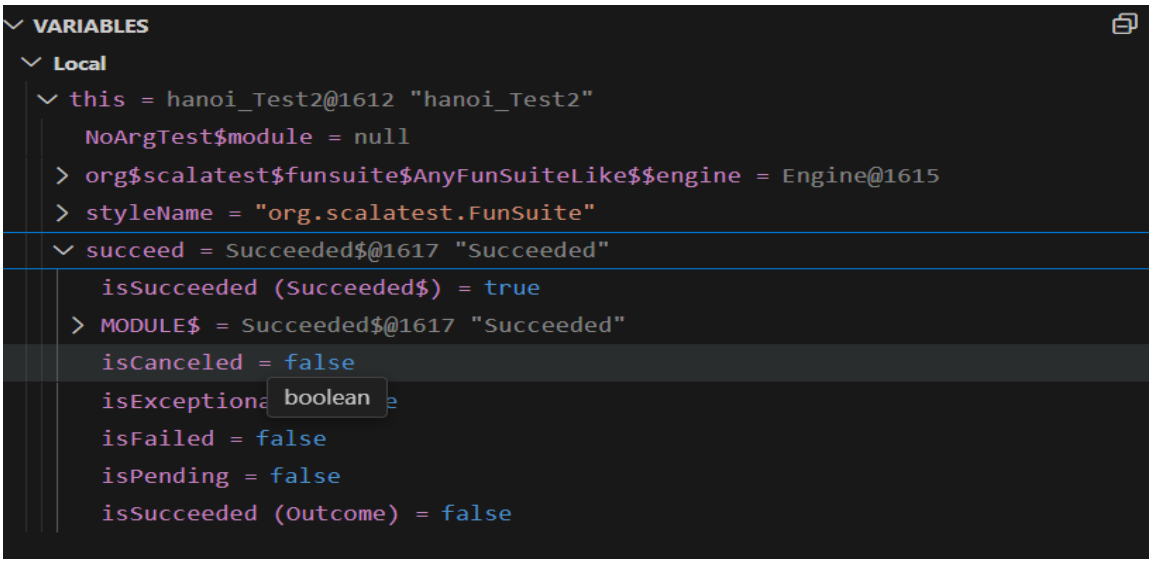
Test #1

Se verifica si el movimiento limite para un disco es igual a “1”.

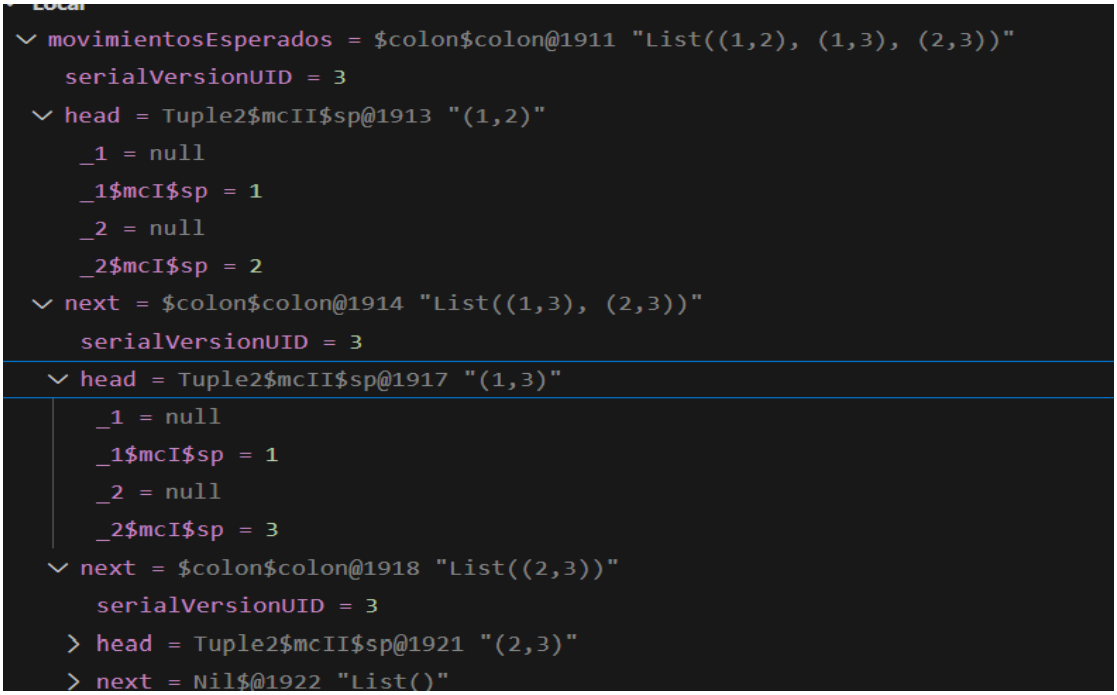


Test #2.

Se verifica si el número mínimo de movimientos para “4” discos es de “15”.

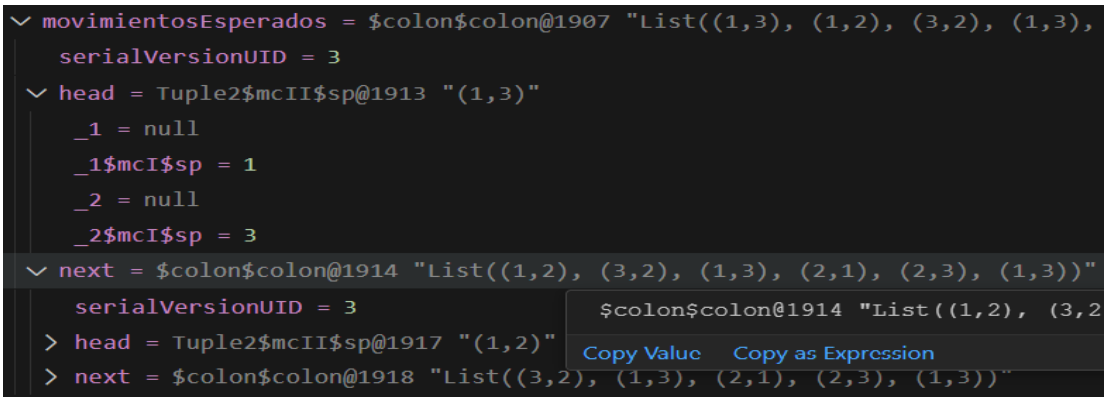


Test #3. para el test #2 se verifican el recorrido de las listas, moviendo las cabezas de cada lista entre ellas mismas hasta crear la lista esperada.



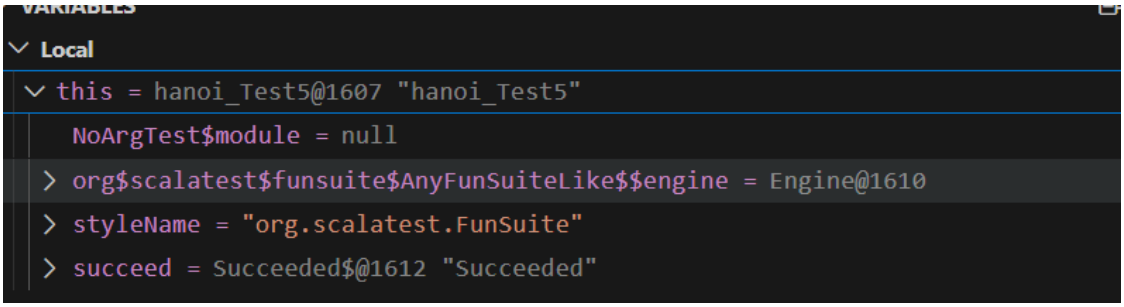
Test #4.

Secuencia de movimientos esperadas para 3 discos.



Test #5

Se verifica que para “0” discos no deben de haber movimientos.



Conclusión.

Este código resuelve el problema de las Torres de Hanói utilizando **recursión**. La recursión es clave porque permite descomponer el problema en subproblemas más pequeños, siguiendo el principio de **dividir y conquistar**. En cada paso, el problema de mover n discos se reduce a mover n-1 discos hasta que finalmente se llega al caso base (n == 0), en el que no hay más discos por mover.

Informe de Procesos

Jesús Loaiza-Jose Daniel Ceballos- Sebastian Ceron

September 16, 2024

1 Corrección de Funciones en Scala

1.1 Corrección de la función `maxLin` (Recursión lineal)

Implementación esquemática de `maxLin`:

```
def maxLin(lista: List[Int]): Int = lista match {
  case Nil          => throw new NoSuchElementException("Lista vacía")
  case x :: Nil     => x
  case x :: xs      => math.max(x, maxLin(xs))
}
```

Inducción estructural sobre la lista `L`:

- **Caso Base:** Si `L` tiene un único elemento, es decir, `L = List(x)`:
  - La función `maxLin(List(x))` devuelve `x`, que es evidentemente el máximo de una lista con un solo elemento.
  - Por lo tanto, la función es correcta en este caso.
- **Paso inductivo:** Supongamos que `maxLin` es correcta para una lista `L'` con `n` elementos, es decir, para una lista de la forma `L' = x2 :: x3 :: ... :: xn`. Esto significa que `maxLin(L')` devuelve correctamente el valor máximo de `L'`.
  - Ahora consideremos una lista `L = x1 :: L'`, con `n + 1` elementos.
  - La función `maxLin(L)` compara el primer elemento `x1` con el valor `maxLin(L')`, que por hipótesis inductiva es el máximo de la sublista `L'`. La función devuelve `math.max(x1, maxLin(L'))`.

- Como `maxLin(L')` devuelve el valor máximo de los elementos en `L'`, la llamada a `math.max(x1, maxLin(L'))` garantiza que el valor máximo de toda la lista `L` se devuelve correctamente.

Por lo tanto, por inducción, `maxLin` devuelve el valor máximo de cualquier lista no vacía de enteros positivos.

1.2 Corrección de la función `maxIt` (Recursión de cola)

Implementación esquemática de `maxIt`:

```
def maxIt(lista: List[Int]): Int = {
  def maxAux(actualMax: Int, listaRestante: List[Int]): Int = listaRestante matc
    case Nil       => actualMax
    case x :: xs   => maxAux(math.max(actualMax, x), xs)
  }
  lista match {
    case Nil       => throw new NoSuchElementException("Lista vacía")
    case x :: xs   => maxAux(x, xs)
  }
}
```

Inducción estructural sobre la lista `L`:

- **Caso Base:** Si `L = List(x)`, la lista contiene un solo elemento.
  - La función `maxIt` inicia llamando a `maxAux(x, Nil)`, lo cual devuelve `x`, ya que la lista restante está vacía.
  - Como `x` es el único elemento, es evidentemente el máximo de la lista.
  - Por lo tanto, la función es correcta para el caso base.
- **Paso inductivo:** Supongamos que `maxIt` es correcta para una lista de `n` elementos, es decir, que `maxAux(actualMax, L')` devuelve el valor máximo de la sublista `L'` para cualquier valor inicial de `actualMax`.
  - Ahora consideremos una lista `L = x1 :: L'` con `n + 1` elementos.
  - La función `maxIt(L)` llama a `maxAux(x1, L')`, y por hipótesis inductiva, `maxAux` devolverá el valor máximo de la lista `L'`, comparado con el valor inicial `x1`.

– Dado que `maxAux` compara cada elemento de la lista con el valor actual del máximo y lo actualiza si es necesario, al finalizar la iteración, devolverá el valor máximo de toda la lista `L`.

Por lo tanto, por inducción, `maxIt` devuelve el valor máximo de cualquier lista no vacía de enteros positivos.

2 Torres de Hanoi

2.1 Descripción del Problema

Este código define un objeto llamado `torres_hanoi` que resuelve el problema de las Torres de Hanói en Scala, proporcionando dos funciones principales:

- `movsTorresHanoi(n: Int): BigInt`: Calcula el número mínimo de movimientos necesarios para resolver el problema con n discos.
- `torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)]`: Devuelve una lista con los movimientos necesarios para mover los discos de un poste a otro, siguiendo las reglas de las Torres de Hanói.

2.2 Pruebas

Las siguientes pruebas verifican la implementación correcta del código en diferentes escenarios.

Test #1

Se verifica si el número mínimo de movimientos para un disco es igual a 1.

Test #2

Se verifica si el número mínimo de movimientos para 4 discos es igual a 15.

Test #3

Para el test #2, se verifican los movimientos realizados, comprobando que las listas de movimientos cumplen con el recorrido esperado.

Test #4

Se verifica la secuencia de movimientos esperada para 3 discos.

Test #5

Se verifica que para 0 discos no debe haber movimientos.

2.3 Corrección por Inducción Matemática

El número mínimo de movimientos para resolver las Torres de Hanoi con n discos sigue la fórmula:

$$T(n) = 2^n - 1$$

Vamos a demostrar que esta fórmula es correcta utilizando inducción matemática.

Paso 1: Base de Inducción

Para $n = 1$, el número mínimo de movimientos es:

$$T(1) = 2^1 - 1 = 1$$

Esto es correcto, ya que para un solo disco, se necesita exactamente un movimiento para trasladarlo de un poste a otro.

Paso 2: Hipótesis de Inducción

Supongamos que la fórmula es cierta para $n = k$, es decir:

$$T(k) = 2^k - 1$$

Paso 3: Paso Inductivo

Ahora debemos demostrar que la fórmula también es cierta para $n = k + 1$. Para resolver el problema con $k + 1$ discos, realizamos las siguientes operaciones:

1. Movemos los primeros k discos de la torre de origen a la torre auxiliar, lo cual requiere $T(k) = 2^k - 1$ movimientos.

2. Movemos el disco más grande directamente a la torre destino, lo cual requiere 1 movimiento.

3. Movemos los k discos de la torre auxiliar a la torre destino, lo cual requiere nuevamente $T(k) = 2^k - 1$ movimientos.

El número total de movimientos es:

$$T(k+1) = T(k) + 1 + T(k) = 2(2^k - 1) + 1 = 2^{k+1} - 1$$

Por lo tanto, la fórmula es válida para $n = k + 1$.

Conclusión

Hemos demostrado por inducción que el número mínimo de movimientos para resolver el problema de las Torres de Hanói con n discos es $T(n) = 2^n - 1$.