



INFORME TALLER 1: FUNCIONES Y PROCESOS

Santiago Avalo Monsalve - 2359442
Daniel Gómez Cano - 2359396
Edward Stevens Pinto - 2359431

Trabajo presentado a:
Carlos Andrés Delgado S.

Universidad del Valle
Fundamentos de Programación Funcional y Concurrente
Ingeniería de Sistemas
Tuluá - Valle del Cauca
2024

1. Informe de procesos

1.1. Método maxIt

```
● 11  def maxIt(l: List[Int]): Int = {  
● 12      require(!l.isEmpty, "La lista no puede estar vacía")  
● 13      require(l.forall(_ >= 0), "La lista no puede contener números negativos")  
● 14      @tailrec  
● 15      def maxIt_nAux ( lista: List[Int] , max: Int ): Int = {  
● 16          if(lista.isEmpty) max  
● 17          else if (lista.head > max) maxIt_nAux(lista.tail, lista.head)  
● 18          else maxIt_nAux(lista.tail, max)  
● 19      }  
● 20      maxIt_nAux(l.tail, l.head)  
● 21  }  
● 22  
● 23  }  
● 24  
● 25 }
```

Línea 12 y 13, en estas dos líneas se comprueba que el valor de “l” ingresado sea una lista no vacía y que todos sus valores sean mayores o iguales a 0, o sea que no sean negativos.

Lista ingresada para el proceso:

```
> l = $colon$colon@653 "List(1, 2, 0, 5, 4)"
```

Después de comprobar que no es una lista vacía, comprueba que cada valor no sea negativo:

```
x$1 = 1 x$1 = 2 x$1 = 0 x$1 = 5 x$1 = 4
```

Y luego ahora si empieza a llamar recursivamente:

Llamado 1:

```
✓ lista = $colon$colon@667 "List(2, 0, 5, 4)"  
    serialVersionUID = 3  
> head = Integer@669 "2"  
> next = $colon$colon@670 "List(0, 5, 4)"  
    max = 1  
> this = BuscarLista@654
```

Llamado 2:

```
✓ lista = $colon$colon@670 "List(0, 5, 4)"  
    serialVersionUID = 3  
> head = Integer@679 "0"  
> next = $colon$colon@680 "List(5, 4)"  
    max = 2  
> this = BuscarLista@654
```

Llamado 3:

```
✓ lista = $colon$colon@680 "List(5, 4)"  
    serialVersionUID = 3  
> head = Integer@690 "5"  
> next = $colon$colon@691 "List(4)"  
    max = 2  
> this = BuscarLista@654
```

Llamado 4:

```
  ✓ lista = $colon$colon@691 "List(4)"  
    serialVersionUID = 3  
    > head = Integer@698 "4"  
    > next = Nil$@699 "List()"  
    max = 5  
  > this = BuscarLista@654
```

Llamado 5:

```
  ✓ lista = Nil$@699 "List()"  
    > EmptyUnzip = Tuple2@709 "(List(),List())"  
    > MODULE$ = Nil$@699 "List()"  
    serialVersionUID = 3  
    max = 5  
  > this = BuscarLista@654
```

Se generó un llamado por cada vez que era llamado el método de forma recursiva con los valores nuevos.

Call Stack:

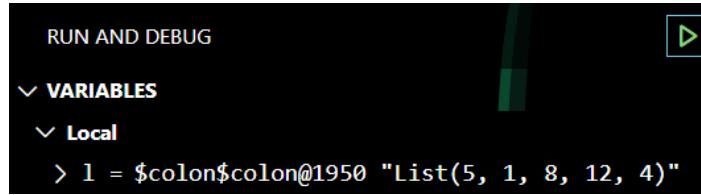
BuscarLista.maxLi_nAux\$1(List, int): int	BuscarLista.scala	23:1
BuscarLista.maxLin(List): int	BuscarLista.scala	23:1
App\$.pruebas(): void	App.scala	14:1
App\$.main(String[]): void	App.scala	9:1
App.main(String[]): void	Unknown Source	-1:1

1.2 Método maxLin

```
30 |   //metodo para hallar el mayor valor en una lista no vacia de enteros positivos  
31 |   //con Recursión Lineal ;)  
32 class MaxLin {  
33 |  
34 |   def maxLin(l: List[Int]): Int = {  
35 |     if (l.isEmpty) throw new NoSuchElementException("La lista no puede estar vacia")  
36 |     else if (l.tail.isEmpty) l.head //Si la lista tiene un elemento, ese es el maximo  
37 |     else l.head max maxLin(l.tail) //Compara el primer elemento con el maximo del resto  
38 |   }  
39 }
```

La función **no almacena el máximo** en una variable desde el principio, sino que lo va encontrando a medida que recorre toda la lista. En cada paso de la recursión, se **compara** el valor actual (`l.head`) con el máximo encontrado hasta ese momento en el resto de la lista (`maxLin(l.tail)`). Una vez que la función llega al caso base (una lista con un solo elemento), ese elemento se propaga hacia arriba en las llamadas recursivas, siempre comparándose con los valores anteriores hasta encontrar el mayor.

Lista empleada para el proceso



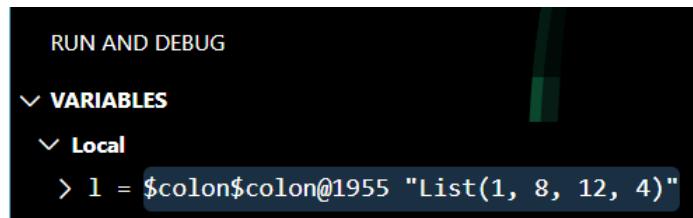
```
RUN AND DEBUG

▼ VARIABLES
  ▼ Local
    > l = $colon$colon@1950 "List(5, 1, 8, 12, 4)"
```

Comprueba que no sea una lista vacía, y luego evalúa el caso base, donde la lista tiene un solo elemento. Una vez que se determina que la lista tiene más de un elemento, continúa con la recursión.

La función empieza con el primer elemento: “5”. Luego compara “5” con el máximo del resto de la lista “List(5, 1, 8, 12, 4)”.

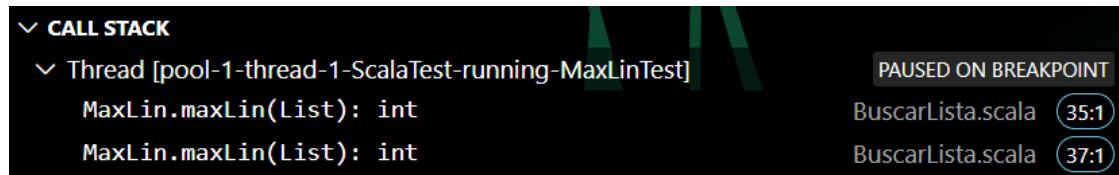
El máximo del resto de la lista “List(5, 1, 8, 12, 4)” es calculado recursivamente:



```
RUN AND DEBUG

▼ VARIABLES
  ▼ Local
    > l = $colon$colon@1955 "List(1, 8, 12, 4)"
```

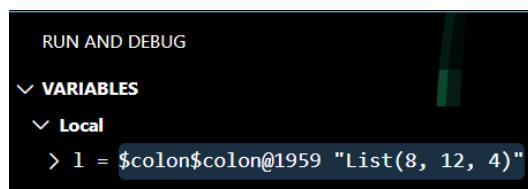
En el **Call Stack** se genera el resultado del proceso anterior, para posteriormente regresar en la recursión.



```
▼ CALL STACK
  ▼ Thread [pool-1-thread-1-ScalaTest-running-MaxLinTest]
    MaxLin.maxLin(List): int
    MaxLin.maxLin(List): int
```

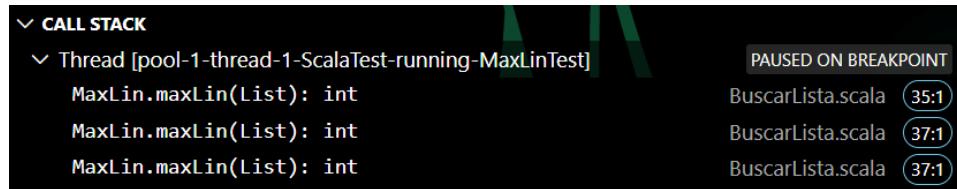
PAUSED ON BREAKPOINT
BuscarLista.scala 35:1
BuscarLista.scala 37:1

Se hace nuevamente el proceso.



```
RUN AND DEBUG

▼ VARIABLES
  ▼ Local
    > l = $colon$colon@1959 "List(8, 12, 4)"
```



```
▼ CALL STACK
  ▼ Thread [pool-1-thread-1-ScalaTest-running-MaxLinTest]
    MaxLin.maxLin(List): int
    MaxLin.maxLin(List): int
    MaxLin.maxLin(List): int
```

PAUSED ON BREAKPOINT
BuscarLista.scala 35:1
BuscarLista.scala 37:1
BuscarLista.scala 37:1

A medida que ocurren los llamados, la lista a comparar se va reduciendo, mientras que el call stack va aumentando.

RUN AND DEBUG

▼ VARIABLES

 ▼ Local

 > l = \$colon\$colon@1963 "List(12, 4)"

▼ CALL STACK

 ▼ Thread [pool-1-thread-1-ScalaTest-running-MaxLinTest]

 MaxLin.maxLin(List): int
 MaxLin.maxLin(List): int
 MaxLin.maxLin(List): int
 MaxLin.maxLin(List): int

PAUSED ON BREAKPOINT

 BuscarLista.scala (35:1)
 BuscarLista.scala (37:1)
 BuscarLista.scala (37:1)
 BuscarLista.scala (37:1)

Siguiente llamado, donde la lista solo tendrá un solo elemento

RUN AND DEBUG

▼ VARIABLES

 ▼ Local

 > l = \$colon\$colon@1967 "List(4)"

▼ CALL STACK

 ▼ Thread [pool-1-thread-1-ScalaTest-running-MaxLinTest]

 MaxLin.maxLin(List): int
 MaxLin.maxLin(List): int
 MaxLin.maxLin(List): int
 MaxLin.maxLin(List): int
 MaxLin.maxLin(List): int

PAUSED ON BREAKPOINT

 BuscarLista.scala (35:1)
 BuscarLista.scala (37:1)
 BuscarLista.scala (37:1)
 BuscarLista.scala (37:1)
 BuscarLista.scala (37:1)

En este punto, solo queda un elemento: “4”. Entonces, la recursión retorna “4” y este se propaga hacia arriba en las llamadas recursivas.

Compara “12” con “4”, y se retorna “12”, porque este es el mayor.

Compara “12” con “8”, y se retorna “12”, porque este es el mayor.

Así sucesivamente, la función propaga el valor máximo a través de las comparaciones del resto de la lista, retornando “12”, ya que este es el mayor de la lista.

1.3 Método movsTorresHanoi

```
● 5  def movsTorresHanoi (n_discos : Int ): BigInt = {  
● 6    //formula que calcula 2 elevado al numero de discos - 1  
● 7    //Resultado igual al numero de movimientos  
● 8    BigInt(2).pow(n_discos) - 1  
● 9  }
```

La línea 8 calcula y devuelve la cantidad de movimientos al elevar el 2 al número de discos y restarle 1.

Valor ingresado para el proceso:

Llamado 1:

```
✓ VARIABLES
  ✓ Local
    n_discos = 3
    > this = Torre_hanoi@592
```

Llamado 2:

```
✓ VARIABLES
  ✓ Local
    > →movsTorresHanoi() = BigInt@593 "7"
    args = String[0]@594
    > hanoi = Torre_hanoi@592
    > this = App$@595
```

Se generaron 2 llamados esenciales los cuales determinan el valor de “n” y el valor del resultado por la fórmula.

1.4 Método listaTorresHanoi

```
● 12 def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
● 13   if (n == 0) {
● 14     List() //caso base
● 15   } else {
● 16
● 17     //Movimientos realizados para formar la torre, sin el ultimo, disco en el centro
● 18     val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
● 19
● 20     //Movimiento principal del disco mas grande a la ultima torre
● 21     val movimientoPrincipal = List((t1, t3))
● 22
● 23     //Movimientos realizados para llevar la torre del centro hasta la torre final
● 24     val movimientos_posteriores= listaTorresHanoi(n - 1, t2, t1, t3)
● 25
● 26     // Combina todos los movimientos
● 27     movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
● 28
● 29   }
● 30 }
```

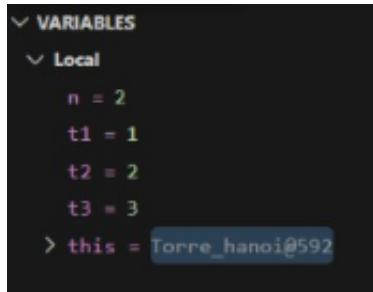
La línea 13 es el caso base cuando no hay más discos (n).

La línea 18 es el caso cuando toca mover toda la torre menos el disco más grande al palo diferente al que tiene que ir el disco más grande. De manera recursiva para las demás torres que se van formando, cambiando sus palos (t) según corresponda el caso,

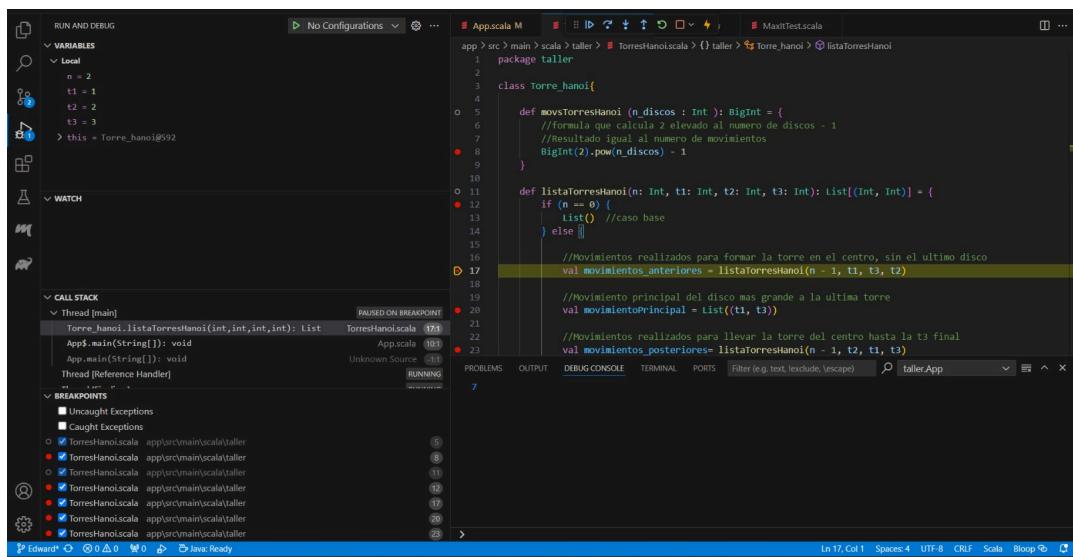
La línea 21 es el movimiento del disco mayor al palo final, de manera recursiva es el movimiento que genera cada disco según los llamados recursivos hechos por las líneas 18 y 27.

Línea 27 es el caso en el que toda la torre menos el disco más pesado se mueve del palo central al último palo, encima del disco con más peso. De manera recursiva para las demás torres que se van formando, cambiando sus palos (t) según corresponda el caso,

Valor ingresado para la prueba:



Llamado 1:



Llamado 2:

The screenshot shows the IntelliJ IDEA interface with the debugger open. The call stack window is visible, showing the following stack frames:

- Thread [main] (Paused on breakpoint)
- Torre_hanoi.listaTorresHanoi(int,int,int): List
- Torre_hanoi.listaTorresHanoi(int,int,int): List
- App\$.main(String[]): void
- App.main(String[])

The code editor shows the `Torre_hanoi` class definition. The current line of execution is at line 17, which defines the `listaTorresHanoi` function. The variable `n` is set to 1 in the local variables pane.

```

RUN AND DEBUG No Configurations ...
VARIABLES Local
    n = 1
    t1 = 1
    t2 = 3
    t3 = 2
    > this = Torre_hanoi@592
WATCH
CALL STACK Thread [main] PAUSED ON BREAKPOINT
    Torre_hanoi.listaTorresHanoi(int,int,int): List
    Torre_hanoi.listaTorresHanoi(int,int,int): List
    App$.main(String[]): void
    App.main(String[])
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter (e.g. text, exclude, escape) tallerApp
Ln 17, Col 1 Spaces: 4 UTF-8 CRLF Scala Bloop

```

Llamado 3:

The screenshot shows the IntelliJ IDEA interface with the debugger open. The call stack window is visible, showing the following stack frames:

- Thread [main] (Paused on breakpoint)
- Torre_hanoi.listaTorresHanoi(int,int,int): List
- Torre_hanoi.listaTorresHanoi(int,int,int): List
- Torre_hanoi.listaTorresHanoi(int,int,int): List
- App\$.main(String[]): void
- App.main(String[])

The code editor shows the `Torre_hanoi` class definition. The current line of execution is at line 17, which defines the `listaTorresHanoi` function. The variable `n` is set to 2 in the local variables pane.

```

RUN AND DEBUG No Configurations ...
VARIABLES Local
    n = 2
    t1 = 1
    t2 = 2
    t3 = 3
    > this = Torre_hanoi@592
WATCH
CALL STACK Thread [main] PAUSED ON BREAKPOINT
    Torre_hanoi.listaTorresHanoi(int,int,int): List
    Torre_hanoi.listaTorresHanoi(int,int,int): List
    Torre_hanoi.listaTorresHanoi(int,int,int): List
    App$.main(String[]): void
    App.main(String[])
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter (e.g. text, exclude, escape) tallerApp
Ln 12, Col 1 Spaces: 4 UTF-8 CRLF Scala Bloop

```

En estos 3 primeros llamados se nos muestra la ramificación de los casos menores cambiando en cada caso la torre del medio y la del final, para devolver un solo movimiento, que sería el del disco más alto.

Llamado 4: Primer movimiento

```

    app > src > main > scala > taller > TorresHanoi.scala > {} taller > Torre_hanoi > listaTorresHanoi
    1 package taller
    2
    3 class Torre_hanoi{
    4
    5     def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
    6         if (n == 0) {
    7             List() //caso base
    8         } else {
    9
    10            //Movimientos realizados para formar la torre en el centro, sin el ultimo disco
    11            val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
    12
    13            //Movimiento principal del disco mas grande a la ultima torre
    14            val movimientoPrincipal = list((t1, t3))
    15
    16            //Movimientos realizados para llevar la torre del centro hasta la t3 final
    17            val movimientos_posteriores = listaTorresHanoi(n - 1, t2, t1, t3)
    18
    19            // Combina todos los movimientos
    20            movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
    21
    22        }
    23
    24    }
    25
    26
    27
    28
    29
    30    }

```

Llamado 5: Fin de la primera ramificación.

```

    app > src > main > scala > taller > TorresHanoi.scala > {} taller > Torre_hanoi > listaTorresHanoi
    1 package taller
    2
    3 class Torre_hanoi{
    4
    5     def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
    6         if (n == 0) {
    7             List() //caso base
    8         } else {
    9
    10            //Movimientos realizados para formar la torre en el centro, sin el ultimo disco
    11            val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
    12
    13            //Movimiento principal del disco mas grande a la ultima torre
    14            val movimientoPrincipal = list((t1, t3))
    15
    16            //Movimientos realizados para llevar la torre del centro hasta la t3 final
    17            val movimientos_posteriores = listaTorresHanoi(n - 1, t2, t1, t3)
    18
    19            // Combina todos los movimientos
    20            movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
    21
    22        }
    23
    24    }
    25
    26
    27
    28
    29
    30    }

```

Llamado 6: Movimiento del disco mayor.

```

    app > src > main > scala > taller > TorresHanoi.scala > {} taller > Torre_hanoi > listaTorresHanoi
    1 package taller
    2
    3 class Torre_hanoi{
    4
    5     def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
    6         if (n == 0) {
    7             List() //caso base
    8         } else {
    9
    10            //Movimientos realizados para formar la torre en el centro, sin el ultimo disco
    11            val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
    12
    13            //Movimiento principal del disco mas grande a la ultima torre
    14            val movimientoPrincipal = list((t1, t3))
    15
    16            //Movimientos realizados para llevar la torre del centro hasta la t3 final
    17            val movimientos_posteriores = listaTorresHanoi(n - 1, t2, t1, t3)
    18
    19            // Combina todos los movimientos
    20            movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
    21
    22        }
    23
    24    }
    25
    26
    27
    28
    29
    30    }

```

Llamado 7: Inicio de la segunda ramificación.

The screenshot shows the Eclipse IDE interface with the code editor open to the `TorreHanoi` class. The code implements the Tower of Hanoi problem using recursion. A breakpoint is set at line 23, which is part of the `listaTorreHanoi` method. The code is paused at this point, as indicated by the "PAUSED ON BREAKPOINT" status in the call stack. The call stack shows the current thread and the recursive call stack. The variables view shows local variables `n`, `t1`, `t2`, and `t3` with their respective values: `n = 2`, `t1 = 1`, `t2 = 2`, and `t3 = 3`. The watch view is empty. The breakpoints view shows several breakpoints set across the code, with some being triggered (indicated by a red dot). The status bar at the bottom indicates "Edward" and "Java Ready".

```
1 package taller
2
3 class Torre_hanoi{
4     def listaTorreHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
5         //Movimientos realizados para formar la torre en el centro, sin el ultimo disco
6         val movimientos_anteriores = listaTorreHanoi(n - 1, t1, t3, t2)
7
8         //Movimiento principal del disco mas grande a la ultima torre
9         val movimientoPrincipal = List((t1, t3))
10
11         //Movimientos realizados para llevar la torre del centro hasta la t3 final
12         val movimientos_posteriores = listaTorreHanoi(n - 1, t2, t1, t3)
13
14         // Combina todos los movimientos
15         movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
16     }
17
18
19
20
21
22
23
24
25
26
27
28
29
30 }
```

Llamado 8:

The screenshot shows the Eclipse IDE interface with the code editor open to the `TorreHanoi` class. The code is now at line 12 of the `listaTorreHanoi` method. The code is paused at this point, as indicated by the "PAUSED ON BREAKPOINT" status in the call stack. The call stack shows the current thread and the recursive call stack. The variables view shows local variables `n`, `t1`, `t2`, and `t3` with their respective values: `n = 0`, `t1 = 2`, `t2 = 3`, and `t3 = 1`. The watch view is empty. The breakpoints view shows several breakpoints set across the code, with some being triggered (indicated by a red dot). The status bar at the bottom indicates "Edward" and "Java Ready".

```
1 package taller
2
3 class Torre_hanoi{
4     def listaTorreHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
5         //formula que calcula 2 elevado al numero de discos - 1
6         //resultado igual al numero de movimientos
7         BigInt(2).pow(n_discos) - 1
8     }
9
10
11     def listaTorreHanoi(n: Int): List[(Int, Int)] = {
12         if (n == 0) {
13             List() //caso base
14         } else {
15             //Movimientos realizados para formar la torre en el centro, sin el ultimo disco
16             val movimientos_anteriores = listaTorreHanoi(n - 1, t1, t3, t2)
17
18             //Movimiento principal del disco mas grande a la ultima torre
19             val movimientoPrincipal = List((t1, t3))
20
21             //Movimientos realizados para llevar la torre del centro hasta la t3 final
22             val movimientos_posteriores = listaTorreHanoi(n - 1, t2, t1, t3)
23
24             // Combina todos los movimientos
25             movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
26         }
27     }
28
29
30 }
```

Llamado 9:

```

RUN AND DEBUG No Configurations ... App.scala M MaxTestscala
app > src > main > scala > taller > TorresHanoiscala > () taller > Torre_hanoi > listaTorresHanoi
1 package taller
2 class Torre_hanoi{
3
4   def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
5     if (n == 0) {
6       List()
7     } else {
8
9      //Movimientos realizados para formar la torre en el centro, sin el ultimo disco
10     val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
11
12     //Movimiento principal del disco mas grande a la ultima torre
13     val movimientoPrincipal = List((t1, t3))
14
15     //Movimientos realizados para llevar la torre del centro hasta la t3 final
16     val movimientos_posteriores = listaTorresHanoi(n - 1, t2, t1, t3)
17
18     // Combina todos los movimientos
19     movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
20   }
21
22
23
24
25
26
27
28
29
30 }

```

Llamado 10:

```

RUN AND DEBUG No Configurations ... App.scala M MaxTestscala
app > src > main > scala > taller > TorresHanoiscala > () taller > Torre_hanoi > listaTorresHanoi
1 package taller
2 class Torre_hanoi{
3
4   def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
5     if (n == 0) {
6       List()
7     } else {
8
9      //Movimientos realizados para formar la torre en el centro, sin el ultimo disco
10     val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
11
12     //Movimiento principal del disco mas grande a la ultima torre
13     val movimientoPrincipal = List((t1, t3))
14
15     //Movimientos realizados para llevar la torre del centro hasta la t3 final
16     val movimientos_posteriores = listaTorresHanoi(n - 1, t2, t1, t3)
17
18     // Combina todos los movimientos
19     movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
20   }
21
22
23
24
25
26
27
28
29
30 }

```

Llamado 11: Fin de la segunda ramificación.

```

RUN AND DEBUG No Configurations ... App.scala M MaxTestscala
app > src > main > scala > taller > TorresHanoiscala > () taller > Torre_hanoi > listaTorresHanoi
1 package taller
2 class Torre_hanoi{
3
4   def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
5     if (n == 0) {
6       List()
7     } else {
8
9      //Movimientos realizados para formar la torre en el centro, sin el ultimo disco
10     val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
11
12     //Movimiento principal del disco mas grande a la ultima torre
13     val movimientoPrincipal = List((t1, t3))
14
15     //Movimientos realizados para llevar la torre del centro hasta la t3 final
16     val movimientos_posteriores = listaTorresHanoi(n - 1, t2, t1, t3)
17
18     // Combina todos los movimientos
19     movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
20   }
21
22
23
24
25
26
27
28
29
30 }

```

Llamado 12: Unión de todas las ramas.

```
app > src > main > scala > taller > Torre_hanoi.scala > () taller > Torre_hanoi > listaTorresHanoi
1 package taller
2 class Torre_hanoi{
3     def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
4         if (n == 1) {
5             List((t1, t3))
6         } else {
7             val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
8             val movimientoPrincipal = (t1, t3)
9             val movimientos_posteriores = listaTorresHanoi(n - 1, t2, t1, t3)
10            // Movimientos realizados para formar la torre en el centro, sin el ultimo disco
11            def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
12                if (n == 1) {
13                    List((t1, t3))
14                } else {
15                    val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
16                    val movimientoPrincipal = (t1, t3)
17                    val movimientos_posteriores = listaTorresHanoi(n - 1, t2, t1, t3)
18                    // Movimiento principal del disco mas grande a la ultima torre
19                    // Movimientos realizados para llevar la torre del centro hasta la t3 final
20                    // Combina todos los movimientos
21                    movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
22                }
23            }
24        }
25    }
26    // Combinacion de los tres tipos de movimientos
27    movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
28 }
29 }
```

Llamado 13 (final): Mostrar la lista.

```
app > src > main > scala > taller > App.scala > () taller > () App > main
1 /*
2  * This Scala source file was generated by the Gradle 'init' task.
3  */
4 package taller
5
6 run | debug
7 object App {
8     def main(args: Array[String]): Unit = {
9         val hanoi = new Torre_hanoi
10        println(hanoi.mvsTorresHanoi(3))
11        println(hanoi.listaTorresHanoi(2, 1, 2, 3))
12    }
13
14    def greeting(): String = "Hello, world!"
15 }
```

Se realizaron un total de 13 llamados, algunos llamados no fueron colocados pues eran redundantes (mostraban lo mismo del llamado anterior), así que fueron seleccionados los llamados que cambiaban los valores para que el informe de procesos no quedara tan largo.

2. Informe de corrección

2.1. Método maxIt

$f: List[Naturales] = Natural$, devuelve el valor máximo de una lista de enteros no negativos.

```
def maxIt(l: List[Int]): Int = {
    require(!l.isEmpty, "La lista no puede estar vacía")
    require(l.forall(_ >= 0), "La lista no puede contener números negativos")
    @tailrec
    def maxIt_nAux ( lista: List[Int] , max: Int ): Int = {
        if(lista.isEmpty) max
        else if (lista.head > max) maxIt_nAux(lista.tail, lista.head)
        else maxIt_nAux(lista.tail, max)
    }
    maxIt_nAux(l.tail, l.head)
}
```

P_f :

Para los casos en que la lista está vacía o tiene algún elemento negativo, arrojaría un error debido a que no se pueden implementar listas vacías o con algún elemento negativo.

$$P_f(List(a_1, a_2, \dots, a_n)) == f(List(a_1, a_2, \dots, a_n))$$

Caso base: $n = 1$ //Lista de un solo elemento

$$P_f(List(x_1)) = p_f(List(), x_1)$$

$$p_f(List(), x_1) = if(List().isEmpty) x_1$$

Como se puede esperar de una lista de un solo elemento, el máximo de esa lista es el elemento.

Caso de inducción: $n = k + 1; k > 0$

Demostrar:

$$P_f(List(a_1, a_2, \dots, a_n)) == f(List(a_1, a_2, \dots, a_n)) \rightarrow$$

$$P_f(List(b_1, b_2, \dots, b_{n+1})) == f(List(b_1, b_2, \dots, b_{n+1}))$$

$$P_f(L) = p_f(L_n, a_1)$$

$$p_f(L_{n-1}, a_1) = if(L_{n-1}.isEmpty) a_1 else if(L_{n-1}.head > a_1) p_f(L_{n-2}, b_1) else p_f(L_{n-2}, a_1)$$

A partir de este punto tomamos 3 posibles rutas en nuestro código.

- $\text{if}(L_{n-1} \cdot \text{isEmpty}) a_1$, en este caso se cumpliría el caso base antes visto.
- $\text{else if}(L_{n-1} \cdot \text{head} > a_1) p_f(L_{n-2}, b_1); b_1 > a_1; b_1 = f(List(a_1, a_2, \dots, a_{n+1}))$
- $\text{else } p_f(L_{n-2}, a_1); a_1 > b_1; a_1 = f(List(a_1, a_2, \dots, a_{n+1}))$

La primera ruta es el caso base, y las otras dos son las comparaciones entre si el nuevo a_n es menor o mayor al otro reemplazandolo en el caso de ser mayor, por tanto podemos decir que:

$$P_f(L) == f(L)$$

2.2 Método MaxLin

$f: List[Naturales] = Natural$, devuelve el valor máximo de una lista de enteros no negativos.

$$P_f:$$

```
def maxLin(l: List[Int]): Int = {
  if (l.isEmpty) throw new NoSuchElementException("La lista no puede estar vacía")
  else if (l.tail.isEmpty) l.head // Si la lista tiene un solo elemento, ese es el máximo
  else l.head max maxLin(l.tail) //Compara el primer elemento con el máximo del resto
}
```

Para los casos en que la lista está vacía, arrojaría un error debido a que no se pueden implementar listas vacías.

$$P_f(List(a_1, a_2, \dots, a_n)) == f(List(a_1, a_2, \dots, a_n))$$

Caso base: $n = 1$ //Lista de un solo elemento

$$P_f(List(x_1)) = \text{if}(List().isEmpty) x_1 \text{ else } x_1 \max(P_f(List()))$$

Si la lista tiene un elemento, este sería su elemento más grande de la lista.

Caso de inducción: $n = k + 1; k > 0$

Demostrar:

$$P_f(List(a_1, a_2, \dots, a_n)) == f(List(a_1, a_2, \dots, a_n)) \rightarrow$$

$$P_f(List(b_1, b_2, \dots, b_{n+1})) == f(List(b_1, b_2, \dots, b_{n+1}))$$

$$P_f(L_n) = \text{if}(L_n.isEmpty) \text{ NoSuchElementException } \text{ else if}(L_{n-1}.isEmpty) a_1 \text{ else } a_1 \max(P_f(L_{n-1})) \\ a_1 \max(a_2 \max(a_3 \max(\dots)))$$

De esta reducción solo sacamos un camino posible y es el preguntar de manera constante cuál elemento es mayor e ir escalando como una fórmula matemática. Por tanto podemos decir que:

$$P_f(L) == f(L)$$

2.3 Método movsTorresHanoi

$f: Natural = Natural$; devuelve un número natural igual a 2 elevado al número de discos menos 1

P_f :

```
def movsTorresHanoi (n_discos : Int ): BigInt = {
    //formula que calcula 2 elevado al numero de discos - 1
    //Resultado igual al numero de movimientos
    require(n_discos >= 0)
    if(n_discos == 0) 0
    else BigInt(2).pow(n_discos) - 1
}
```

Caso base: $n = 0$

$$P_f(0) = \text{if}(0 == 0) 0 \text{ else } BigInt(2).pow(0) - 1$$

En el caso de ser 0 discos no tendría solución por lo que no tendría movimientos

Inducción:

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

$$P_f(n) = 2.pow(n) - 1$$

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

Por tanto podemos decir que:

$$P_f(n) == f(n)$$

2.4 Método listaTorresHanoi

$f: Natural = Lista(tupla(P_i, P_f))$, recibe un número natural de discos y devuelve una lista con tuplas de 2 valores, siendo la posición inicial y la posición final.

P_f :

```

• 12  def listaTorresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
• 13    if (n == 0) {
• 14      List() //caso base
• 15    } else {
• 16
• 17      //Movimientos realizados para formar la torre, sin el ultimo, disco en el centro
• 18      val movimientos_anteriores = listaTorresHanoi(n - 1, t1, t3, t2)
• 19
• 20      //Movimiento principal del disco mas grande a la ultima torre
• 21      val movimientoPrincipal = List((t1, t3))
• 22
• 23      //Movimientos realizados para llevar la torre del centro hasta la torre final
• 24      val movimientos_posteriores= listaTorresHanoi(n - 1, t2, t1, t3)
• 25
• 26      // Combina todos los movimientos
• 27      movimientos_anteriores ++ movimientoPrincipal ++ movimientos_posteriores
• 28
• 29
• 30    }
}

```

Caso Base: $n = 0$

$$P_f(0, t_1, t_2, t_3) = \text{if } (0 == 0) \text{List}()$$

En el caso de no tener discos que mover devolverá una lista vacía.

Caso Base: $n = 1$

$$P_f(1, t_1, t_2, t_3) = \text{if } (1 == 0) \text{List}() \text{ else } P_f(0, t_1, t_3, t_2) ++ \text{List}((t_1, t_3)) ++ P_f(0, t_2, t_1, t_3)$$

$$P_f(1, t_1, t_2, t_3) = \text{List}() ++ \text{List}((t_1, t_3)) ++ \text{List}()$$

$$P_f(1, t_1, t_2, t_3) = \text{List}((t_1, t_3))$$

En el caso de tener un solo disco, devolverá una lista con una tupla (1 solo movimiento), el movimiento de la torre de inicio y la torre de fin.

Caso inductivo: $n = k + 1; k > 0$

$$P_f(n, t_1, t_2, t_3) = f(n) \rightarrow P_f(n + 1, t_1, t_2, t_3) = f(n + 1)$$

$$P_f(n + 1, t_1, t_2, t_3) = \text{if } (n + 1 == 0) \text{List}() \text{ else } P_f(n, t_1, t_3, t_2) ++ \text{List}((t_1, t_3)) ++ P_f(n, t_2, t_1, t_3)$$

En este punto el código se empieza a ramificar hasta que no sea igual a 0 y devuelva la lista vacía.

Para el caso $P_f(n + 1)$, devuelve la lista con la tupla (t_1, t_3) sumado a los dos movimientos de $P_f(n)$, podemos determinar que $P_f(n)$ es la sub torre que se genera en la t2 (torre de el medio o torre de apoyo) pues para cada paso de $P_f(n)$ Se genera un movimiento de entrada desde t1 hasta t2 y un movimiento de salida de t2 a t3.

Entonces podemos decir que:

$$P_f(n + 1) == f(n + 1)$$

3. Conclusiones

3.1 Optimización y eficiencia mediante recursión de cola:

La implementación de las funciones `maxLin` y `maxIt` utilizando recursión de cola nos permitió obtener una solución eficiente para encontrar el máximo de una lista de enteros positivos no vacía. Aunque ambas funciones cumplen el mismo propósito, `maxIt` destaca por hacer uso de un acumulador, lo que garantiza un comportamiento más eficiente en términos de memoria. La utilización de la anotación `@tailrec` asegura que el compilador de Scala optimice la recursión y prevenga el desbordamiento de la pila, demostrando así la importancia de esta técnica para optimizar funciones recursivas.

3.2. Validación y robustez:

Esto no estaba directamente explícito en el trabajo, pero se decidió incluir validaciones en ambas funciones de cálculo del máximo, tanto para verificar que la lista no esté vacía como para asegurar que todos los elementos sean no negativos. Esto no solo mejora la robustez del código sino que también previene posibles errores en la ejecución. Las excepciones lanzadas proporcionan retroalimentación clara en casos de entradas inválidas, haciendo las funciones más seguras y predecibles.

3.3 Resolución de las Torres de Hanoi:

En el segundo ejercicio, las funciones “movsTorresHanoi” y “listaTorresHanoi” lograron resolver el clásico problema de las Torres de Hanoi utilizando recursión. La función “movsTorresHanoi” destacó por su eficiencia al implementar una fórmula matemática para calcular el número mínimo de movimientos necesarios, demostrando la capacidad de resolver problemas complejos de forma compacta. Mientras tanto, “listaTorresHanoi” mostró cómo descomponer el problema en subproblemas más pequeños y generar una lista de movimientos para completar la tarea, aplicando correctamente la técnica divide y vencerás.

3.4. Rigor matemático y pruebas exhaustivas:

Durante el desarrollo del informe de corrección, las pruebas exhaustivas y el análisis matemático de las funciones confirmaron la corrección de las mismas. Se crearon múltiples tests que validaron tanto el comportamiento en casos límite como la consistencia de los resultados esperados. Esto no solo refuerza la calidad de las soluciones propuestas, sino que también permite entender cómo los conceptos matemáticos subyacentes a los problemas tratados se traducen de manera directa a soluciones algorítmicas.

3.5. Importancia del diseño modular:

En todo el trabajo se destacó la importancia de dividir los problemas en partes más pequeñas y manejables. Tanto en el cálculo del máximo de una lista como en la resolución de las Torres de Hanoi, descomponer los problemas en subfunciones y subcasos facilitó la comprensión y la implementación de las soluciones. Esta metodología permitió crear soluciones eficientes y escalables que pueden ser fácilmente adaptadas o extendidas en futuras aplicaciones.