

INFORME DE PROCESOS

❖ **Máximo de una lista de enteros con recursión lineal y recursión de cola.**

Se presenta una clase que contiene dos métodos: `maxList` y `minList`. El primer método sirve para calcular el elemento mayor de una lista de enteros, y el segundo calcula el elemento menor.

Función `maxIt` con recursión de cola

Demostración de la funcionalidad con la depuración en Visual Studio Code del test

```
def maxIt( l : List [ Int ] ) : Int = {  
  
    def maxItAux( l : List [ Int ] , max : Int ) : Int = {  
        if ( l.isEmpty ) max  
        else if ( l.head > max ) maxItAux( l.tail , l.head )  
        else maxItAux( l.tail , max )  
    }  
  
    if ( l.isEmpty ) 0  
    else maxItAux( l , Int.MinValue )  
  
}
```

Para esta función nos apoyamos de otra función auxiliar que nos permite llevar a cabo una recursión de cola.

Inicialmente se usó una lista `l = (99, 8, 25, 30, 90, 101, 110)`, la función verifica que no sea una lista vacía, seguido de eso llama la función auxiliar pasándole la lista y un valor definido `minValue`.

```
✓ Local  
✓ l = $colon$colon@1939 "List(99, 8, 25, 30, 90, 101, 110)"  
    serialVersionUID = 3  
    > head = Integer@1946 "99"  
    > next = $colon$colon@1947 "List(8, 25, 30, 90, 101, 110)"  
> this = MaxMinList@1940
```

```
✓ Local
✓ l = $colon$colon@1939 "List(99, 8, 25, 30, 90, 101, 110)"
  serialVersionUID = 3
  > head = Integer@1946 "99"
  > next = $colon$colon@1947 "List(8, 25, 30, 90, 101, 110)"
  max = -2147483648
  > this = MaxMinList@1940
```

Luego, la función auxiliar verifica si l.head que es igual al primer elemento de la lista (99) es mayor que la variable max, como si es el caso vuelve a invocar la función con l.tail como lista (que devuelve la lista sin el primer elemento), y manda la variable max como el valor de l.head.

```
✓ Local
✓ l = $colon$colon@1947 "List(8, 25, 30, 90, 101, 110)"
  serialVersionUID = 3
  > head = Integer@1956 "8"
  > next = $colon$colon@1957 "List(25, 30, 90, 101, 110)"
  max = 99
  > this = MaxMinList@1940
```

Ahora, hace las mismas verificaciones, como no es una lista vacía cuestiona si ahora l.head(8) es mayor a la variable max(99), como no es el caso pasa a la última condición, donde nuevamente invoca la función con l.tail como lista (que excluye el primer elemento (8) de la lista) y el mismo valor max que tenía (99).

```
✓ Local
✓ l = $colon$colon@1957 "List(25, 30, 90, 101, 110)"
  serialVersionUID = 3
  > head = Integer@1985 "25"
  > next = $colon$colon@1986 "List(30, 90, 101, 110)"
  max = 99
  > this = MaxMinList@1940
```

En este caso l.head (25) tampoco es mayor que el valor de max(99), por lo cual entra en la condición donde se pasa nuevamente la lista sin el primer elemento y el mismo valor max.

```
✓ Local
  ✓ l = $colon$colon@1986 "List(30, 90, 101, 110)"
    serialVersionUID = 3
    > head = Integer@1996 "30"
    > next = $colon$colon@1997 "List(90, 101, 110)"
    max = 99
    > this = MaxMinList@1940
```

Pasa el mismo caso, el primer elemento(30) no es mayor que max (99) y vuelve a entrar en la condición ya mencionada.

```
✓ Local
  ✓ l = $colon$colon@1997 "List(90, 101, 110)"
    serialVersionUID = 3
    > head = Integer@2007 "90"
    > next = $colon$colon@2008 "List(101, 110)"
    max = 99
    > this = MaxMinList@1940
```

En este caso se cumple la misma condición mencionada anteriormente, 90 no es mayor que max (99), vuelve a invocar la función sin el primer elemento (90) y con el mismo valor max.

```
✓ Local
  ✓ l = $colon$colon@2008 "List(101, 110)"
    serialVersionUID = 3
    > head = Integer@2018 "101"
    > next = $colon$colon@2019 "List(110)"
    max = 99
    > this = MaxMinList@1940
```

Ahora, vemos como l.head (101) si es mayor que el valor almacenado en max (99), por lo cual entra en la condición y ahora invoca la función con la l.tail y como valor max envía l.head (101).

```
✓ Local
  ✓ l = $colon$colon@2019 "List(110)"
    serialVersionUID = 3
  > head = Integer@2039 "110"
  > next = Nil$@2040 "List()"
  max = 101
  > this = MaxMinList@1940
```

Ahora, l.head (110 que es el elemento restante) también es mayor al valor max (101), por lo que nuevamente invoca la función con l.tail y como valor max 110.

```
✓ Local
  ✓ l = Nil$@2040 "List()"
  > EmptyUnzip = Tuple2@2047 "(List(),List())"
  > MODULE$ = Nil$@2040 "List()"
  serialVersionUID = 3
  max = 110
  > this = MaxMinList@1940
```

Ahora la lista se encuentra vacía, ya que recorrimos todos los elementos, finalmente, entra en la condición donde l.isEmpty es verdadera y retorna el valor almacenado en la variable max que es 110, este valor es el mayor de todos en la lista dada, y así termina la prueba de esta función.

Función maxLin con recursión lineal

Demostración de la funcionalidad con la depuración en Visual Studio Code del test

Inicialmente, tenemos una función que recibe una lista dada, y por medio de un patrón match se descompone la lista en head y tail, para así analizar qué tipo de lista se está manejando.

En el caso base verifica si esta es una lista vacía, generando una excepción en dicho caso, en el segundo caso, verifica si el elemento es único en la lista retornando el mismo, y en el último caso, si la lista tiene más de un elemento se creará una variable llamada `maxTail` que recibirá el valor de la invocación nuevamente de la función, pasándole como parámetro

Vanessa Alexandra Durán Mona - 2359394

Juan Damián Cuervo Buitrago - 2359413

Juan Sebastián Rodas Ramírez - 2359681

la lista sin el primer elemento, después, verifica la cabeza de la lista es mayor que la variable maxTail, si es así retorna la cabeza, sino, retorna la variable.

```
def maxLin( l : List [ Int ] ) : Int = {  
  l match {  
    case Nil => throw new NoSuchElementException("La lista está vacía")  
    case head :: Nil => head  
    case head :: tail =>  
      val maxTail = maxLin(tail)  
      if (head > maxTail) head else maxTail  
  }  
}
```

Veamos ahora un caso con l = (4,0,1,5,6)

Local

```
> l = $colon$colon@1933 "List(4, 0, 1, 5, 6)"  
  serialVersionUID = 3  
> head = Integer@1943 "4"  
> next = $colon$colon@1944 "List(0, 1, 5, 6)"  
> this = MaxMinList@1934
```

Local

```
> l = $colon$colon@1933 "List(4, 0, 1, 5, 6)"  
  serialVersionUID = 3  
> head = Integer@1943 "4"  
> next = $colon$colon@1944 "List(0, 1, 5, 6)"  
  head = 4  
> tail = $colon$colon@1944 "List(0, 1, 5, 6)"  
> this = MaxMinList@1934
```

Al entrar en el match hace las verificaciones, como es una lista con más de un elemento, se crea una variable con el valor del retorno de la función, con l.tail, por eso se muestra en la lista de variables.

```
Local
  1 = $colon$colon@1944 "List(0, 1, 5, 6)"
    serialVersionUID = 3
  > head = Integer@1953 "0"
  > next = $colon$colon@1954 "List(1, 5, 6)"
  > this = MaxMinList@1934
```

Ahora se evalúa la expresión para calcular el valor de la variable maxTail, volviendo a pasar por los casos.

```
Local
  1 = $colon$colon@1944 "List(0, 1, 5, 6)"
    serialVersionUID = 3
  > head = Integer@1953 "0"
  > next = $colon$colon@1954 "List(1, 5, 6)"
  head = 0
  > tail = $colon$colon@1954 "List(1, 5, 6)"
  > this = MaxMinList@1934
```

```
Local
  1 = $colon$colon@1954 "List(1, 5, 6)"
    serialVersionUID = 3
  > head = Integer@1974 "1"
  > next = $colon$colon@1975 "List(5, 6)"
  > this = MaxMinList@1934
```

Vuelve a calcularse ahora el maxTail con el l.tail de la lista dada anteriormente.

Vanessa Alexandra Durán Moná - 2359394
Juan Damián Cuervo Buitrago - 2359413
Juan Sebastián Rodas Ramírez - 2359681

✓ Local

```
✓ l = $colon$colon@1975 "List(5, 6)"  
  serialVersionUID = 3  
> head = Integer@1987 "5"  
> next = $colon$colon@1988 "List(6)"  
> this = MaxMinList@1934
```

✓ Local

```
✓ l = $colon$colon@1975 "List(5, 6)"  
  serialVersionUID = 3  
> head = Integer@1987 "5"  
> next = $colon$colon@1988 "List(6)"  
  head = 5  
> tail = $colon$colon@1988 "List(6)"  
> this = MaxMinList@1934
```

Repite el mismo proceso anterior

✓ Local

```
✓ l = $colon$colon@1988 "List(6)"  
  serialVersionUID = 3  
> head = Integer@2005 "6"  
> next = Nil$@2006 "List()"   
> this = MaxMinList@1934
```

Ahora, solo queda un elemento, por ende, retornará l.head, y empieza hacer las comparaciones.

✓ Local

```
✓ l = $colon$colon@1975 "List(5, 6)"  
  serialVersionUID = 3  
> head = Integer@1987 "5"  
> next = $colon$colon@1988 "List(6)"  
  head = 5  
> tail = $colon$colon@1988 "List(6)"  
  maxTail = 6  
> this = MaxMinList@1934
```

maxTail ahora vale 6, y la cabeza de la lista 5, como 5 no es mayor, se conserva el mismo valor maxTail.

✓ Local

```
✓ l = $colon$colon@1954 "List(1, 5, 6)"  
  serialVersionUID = 3  
> head = Integer@1974 "1"  
> next = $colon$colon@1975 "List(5, 6)"  
  head = 1  
> tail = $colon$colon@1975 "List(5, 6)"  
  maxTail = 6  
> this = MaxMinList@1934
```

Ahora l.head vale 1, y este no es mayor que maxTail, por ende se conserva el valor de esta variable.


```
✓ Local
  ✓ l = $colon$colon@1944 "List(0, 1, 5, 6)"
    serialVersionUID = 3
  > head = Integer@1953 "0"
  > next = $colon$colon@1954 "List(1, 5, 6)"
    head = 0
  > tail = $colon$colon@1954 "List(1, 5, 6)"
    maxTail = 6
  > this = MaxMinList@1934
```

En este caso l.head vale 0, que no es menor a 6.

```
✓ Local
  ✓ l = $colon$colon@1933 "List(4, 0, 1, 5, 6)"
    serialVersionUID = 3
  > head = Integer@1943 "4"
  > next = $colon$colon@1944 "List(0, 1, 5, 6)"
    head = 4
  > tail = $colon$colon@1944 "List(0, 1, 5, 6)"
    maxTail = 6
  > this = MaxMinList@1934
```

Para este caso l.head tampoco es mayor que maxTail, el valor de la misma se conserva. Terminando así la función y retornando el valor maxTail que corresponde al valor máximo de una lista dada.

❖ Torres de Hanoi

Se presenta una clase que contiene dos métodos: **movsTorresHanoi** y **torresHanoi**. El primero es un método para calcular el número de movimientos necesarios para resolver el problema de las torres de Hanoi, y el segundo pasa **n** discos de la primera torre a la tercera.

movsTorresHanoi:

```
def movsTorresHanoi(n: Int) : BigInt = {  
    if (n == 1) 1  
    else 2 * movsTorresHanoi(n-1) + 1  
}
```

En este caso, la función toma un **Int n** que representa el número de discos a pasar y devuelve un **BigInt**, esto se debe a que el número de movimientos crece exponencialmente con **n**.

Caso base: **if(n == 1)** la función devuelve **1**. Es porque si solo hay un disco, solo se necesita un movimiento para moverlo a la tercera torre.

Caso recursivo: **else * movsTorresHanoi(n-1) + 1**; si **n** es mayor que **1**, la función calcula el número de movimientos de manera recursiva.

Ejemplo con n=3:

Paso 1: movsTorresHanoi(3)

1. La función entra con $n=3$.
2. Como $n \neq 1$, se evalúa: $2 \times \text{movsTorresHanoi}(3-1) + 1$

Paso 2: movsTorresHanoi(2)

1. La función se evalúa con $n = 2$.
2. Como $n \neq 1$, se evalúa: $2 \times \text{movsTorresHanoi}(2-1) + 1$

Paso 3: movsTorresHanoi(1)

1. La función se evalúa con $n=1$, por lo que es el caso base y devuelve 1.

Paso 4: Se vuelve a movsTorresHanoi(2)

1. Ya se sabe que $\text{movsTorresHanoi}(1) = 1$, por lo que la evaluación de $\text{movsTorresHanoi}(2)$ continúa: $2 \times 1 + 1 = 3$
2. La función devuelve 3.

Paso 5: Se vuelve a movsTorresHanoi(3)

1. Se sabe que $\text{movsTorresHanoi}(2) = 3$, por lo que la evaluación de $\text{movsTorresHanoi}(3)$ es: $2 \times 3 + 1 = 7$. -> La función devuelve 7.

Razonamiento para la Creación de la Función *movsTorresHanoi*

Para mover **n** discos de una torre a otra usando una torre auxiliar, se tienen que seguir tres pasos básicos:

1. Mover **n-1** discos más pequeños (de la torre inicial a la torre auxiliar): esto requiere resolver el mismo problema, pero con **n-1** discos. El número de movimientos necesarios para esta operación es el mismo que para resolver el problema con **n-1** discos.
2. Mover el disco más grande (el disco **n**) de la torre inicial a la torre final: este es un solo movimiento, ya que solo se mueve un disco.
3. Mover los **n-1** discos más pequeños de la torre auxiliar a la torre final: Después de mover nuevamente los **n-1** discos de la torre auxiliar a la torre final. Esta es exactamente la misma operación que el paso 1, pero con las torres inicial y final invertidas.

Por tanto, $H(n) = H(n-1) + 1 + H(n-1) \Rightarrow H(n) = 2 \times H(n-1) + 1$

Ejemplo con $n=3$

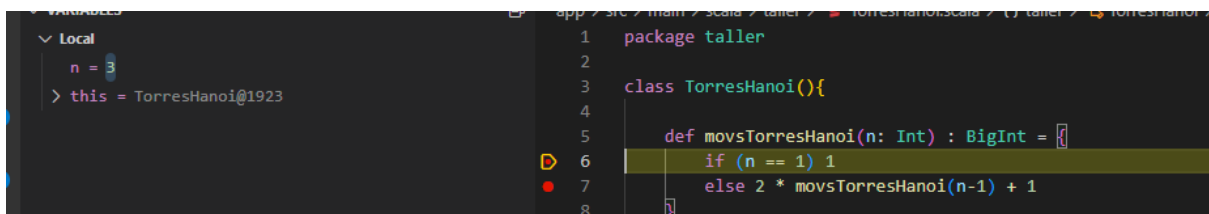
1. Mover los 2 discos más pequeños (los $n-1$) a la torre auxiliar. Esto toma $2^2 - 1 = 3$ movimientos.
2. Mover el disco más grande a la torre final. Esto toma 1 movimiento.
3. Mover los 2 discos de la torre auxiliar a la torre final. Esto toma otros 3 movimientos.

Por tanto, $3 + 1 + 3 = 7$ que es lo mismo que $2 \times 3 + 1 = 7$

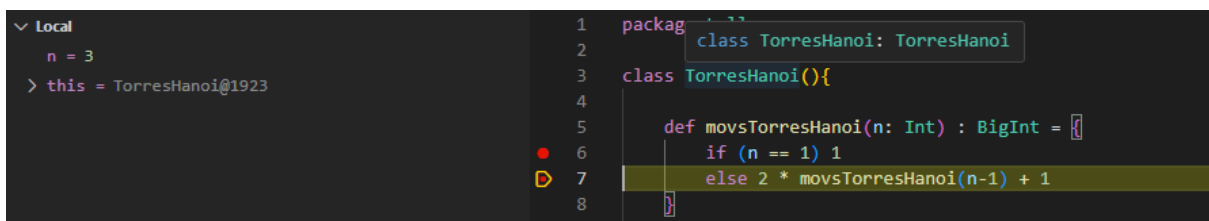
En resumen, el doble de movimientos en la función se debe a que, en el proceso, se tiene que mover los $n-1$ discos dos veces: una vez al inicio (a la torre auxiliar) y otra al final (a la torre final) y el 1 es el movimiento del disco más grande.

Demostración de la funcionalidad con la depuración en Visual Studio Code del test con $n=3$

```
// Test 2: Caso n=3
test("movsTorresHanoi deberia retornar 7 cuando n=3"){
  assert(torresHanoi.movsTorresHanoi(3) == 7)
}
```

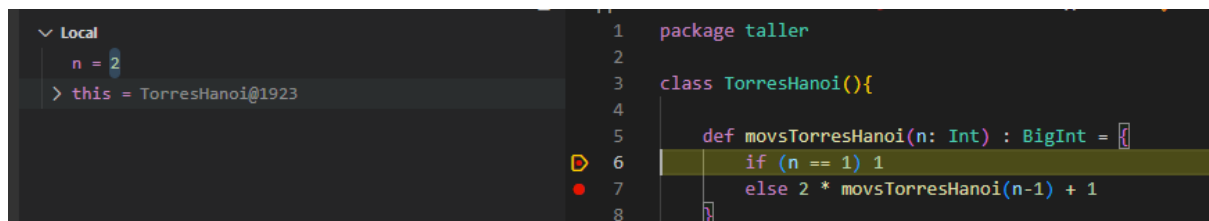


Aquí se observa el momento en que el programa entra en la función con el valor $n=3$. En el panel de variables, se puede visualizar que n tiene el valor de 3, y el programa está listo para evaluar la condición `if(n==1)`



Se observa que no entró a la primera condición dado que $n=3$ por lo que pasa a la segunda condición

Vanessa Alexandra Durán Mona - 2359394
Juan Damián Cuervo Buitrago - 2359413
Juan Sebastián Rodas Ramírez - 2359681

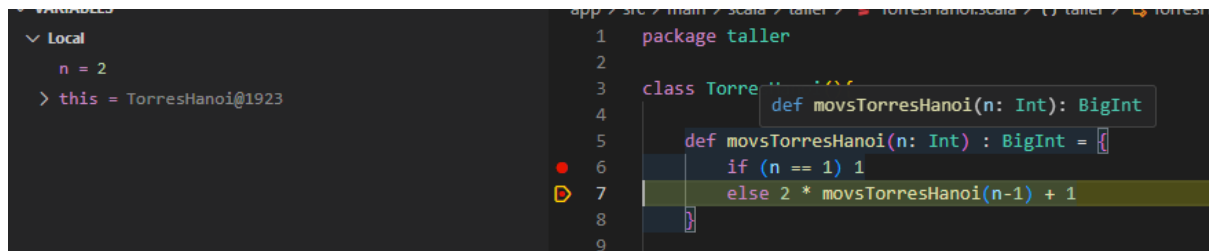


```
1 package taller
2
3 class TorresHanoi(){
4
5     def movsTorresHanoi(n: Int) : BigInt = {
6         if (n == 1) 1
7         else 2 * movsTorresHanoi(n-1) + 1
8     }
```

Local

- n = 2
- > this = TorresHanoi@1923

Se evaluó el caso recursivo. La depuración ha avanzado y ahora se muestra en el panel de variables que $n=2$. Nuevamente se vuelve a comparar si $n=1$, pero no entra en esta condición.

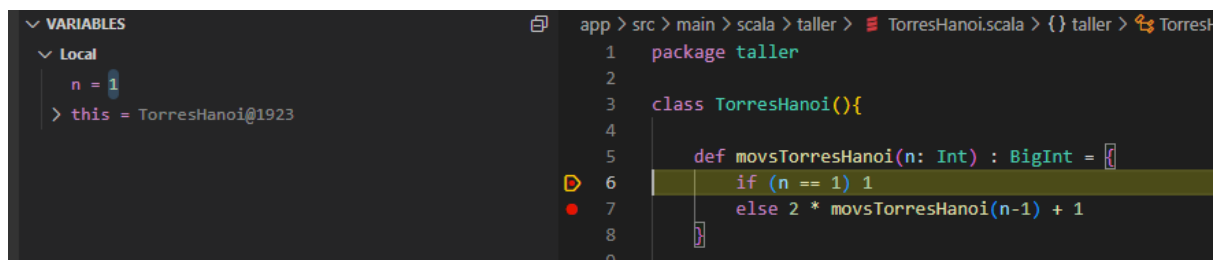


```
1 package taller
2
3 class TorresHanoi(){
4
5     def movsTorresHanoi(n: Int) : BigInt = {
6         if (n == 1) 1
7         else 2 * movsTorresHanoi(n-1) + 1
8     }
```

Local

- n = 2
- > this = TorresHanoi@1923

Se observa que no entró a la primera condición dado que $n=2$ por lo que pasa a la segunda condición



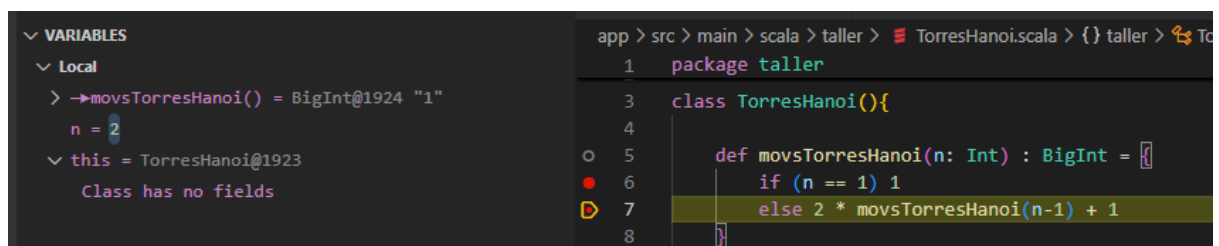
```
1 package taller
2
3 class TorresHanoi(){
4
5     def movsTorresHanoi(n: Int) : BigInt = {
6         if (n == 1) 1
7         else 2 * movsTorresHanoi(n-1) + 1
8     }
```

VARIABLES

Local

- n = 1
- > this = TorresHanoi@1923

Dado que en el paso anterior se entró en la segunda condición, ahora $n=1$, lo que supone el caso base y la función devolverá 1.



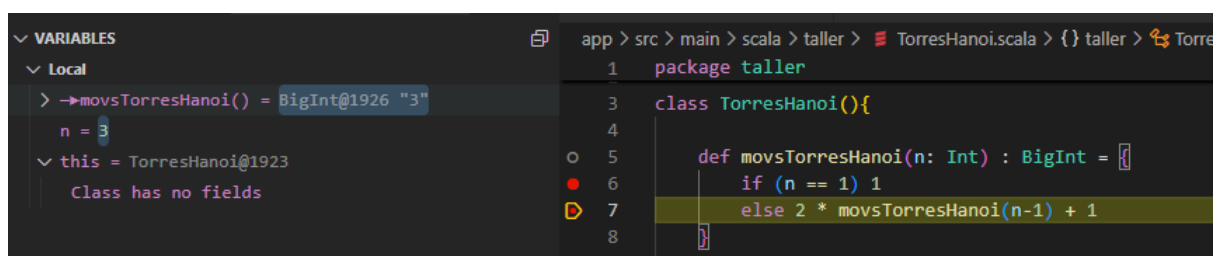
```
1 package taller
2
3 class TorresHanoi(){
4
5     def movsTorresHanoi(n: Int) : BigInt = {
6         if (n == 1) 1
7         else 2 * movsTorresHanoi(n-1) + 1
8     }
```

VARIABLES

Local

- > ->movsTorresHanoi() = BigInt@1924 "1"
- n = 2
- this = TorresHanoi@1923
- Class has no fields

Ahora n toma el valor de 2 y como ya se sabe que la cuando n vale 1 la función retorna 1, entonces se hace directamente la evaluación.



```
1 package taller
2
3 class TorresHanoi(){
4
5     def movsTorresHanoi(n: Int) : BigInt = {
6         if (n == 1) 1
7         else 2 * movsTorresHanoi(n-1) + 1
8     }
```

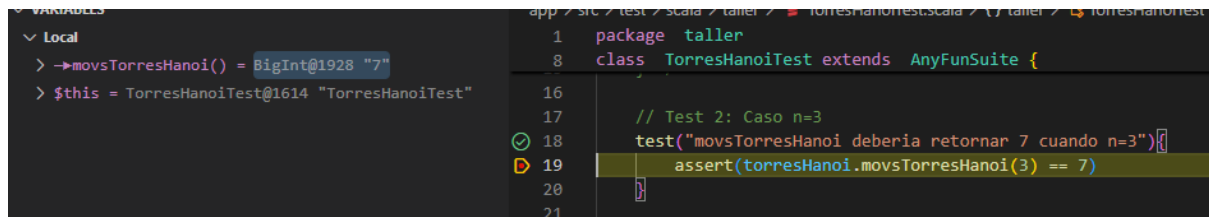
VARIABLES

Local

- > ->movsTorresHanoi() = BigInt@1926 "3"
- n = 3
- this = TorresHanoi@1923
- Class has no fields

Vanessa Alexandra Durán Mona - 2359394
Juan Damián Cuervo Buitrago - 2359413
Juan Sebastián Rodas Ramírez - 2359681

Al haber evaluado la función con $n=2$ la función retornó 3, por lo que ahora $n=3$, y se pasa a evaluar con este valor.



```
app / src / test / scala / taller / TorresHanoiTest.scala / 17 taller / TorresHanoiTest
1 package taller
8 class TorresHanoiTest extends AnyFunSuite {
16
17 // Test 2: Caso n=3
18 test("movsTorresHanoi debería retornar 7 cuando n=3") {
19     assert(torresHanoi.movsTorresHanoi(3) == 7)
20 }
21
```

Después de haber evaluado la función con $n=3$, esta retorna 7, que efectivamente es el valor esperado.

torresHanoi:

```
def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
    if (n == 1) List((t1, t3))
    else {
        val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
        val moverDMayor = List((t1, t3))
        val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
        moverDMenor ++ moverDMayor ++ moverDMenor2
    }
}
```

Parámetros de la función:

n: Número de discos

t1: Poste inicial (posición inicial de los discos)

t2: Poste auxiliar (se usa como ayuda en los movimientos)

t3: Poste destino (posición final de los discos)

1. Caso base (**$n=1$**): En caso de haber un solo disco, solo se mueve directamente del poste **t1** al **t3**. El resultado es una lista con solo un movimiento: **List((t1,t3))**.
2. Caso recursivo (**$n>1$**):
 - La función mueve los **n-1** discos más pequeños del **t1** al **t2**, usando **t3** como auxiliar. Esto se realiza con la llamada recursiva **torresHanoi(n-1,t1,t3,t2)**.
 - Luego, mueve el disco más grande de **t1** a **t3** con el movimiento **List((t1,t3))**.
 - Por último, mueve los **n-1** discos que están en **t2** hacia **t3**, usando ahora **t1** como auxiliar, con la llamada recursiva **torresHanoi(n-1,t2,t1,t3)**.

Ejemplo para **n=3**:

Paso 1: **torresHanoi(3,1,2,3)**

Primero, mueve los 2 discos superiores del **t1(1)** a **t2(2)**:

Paso 2: **torresHanoi(2,1,3,2)**

1. Mueve el disco superior de **t1(1)** a **t3(3)**.
2. Mueve el disco del medio de **t1(1)** a **t2(2)**.
3. Mueve el disco de **t3(3)** a **t2(2)**.
4. Dando como resultado **List((1,3), (1,2), (3,2))**
5. Luego, mueve el disco más grande (el 3) de **t1(1)** a **t3(3)** (**List((1,3))**).

Paso 3: **torresHanoi(2,2,1,3)**

1. Mueve el disco superior de **t2(2)** a **t1(1)**.
2. Mueve el disco del medio de **t2(2)** a **t(3)**.
3. Mueve el disco de **t1(1)** a **t3(3)**.
4. Dando como resultado **List((2,1),(2,3),(1,3))**

Resultado final:

Combinado todos los movimientos hechos, el resultado total para **n=3** sería:

List((1, 3), (1, 2), (3, 2), (1, 3), (2, 1), (2, 3), (1, 3))

Que visto de manera general nos da los siguientes pasos:

1. Mueve los **n-1** discos más pequeños de **t1 a t2**
2. Mueve el disco más grande de **t1 a t3**
3. Mueve los **n-1** discos más pequeños de **t2 a t3**
- 4.

Demostración de la funcionalidad con la depuración en Visual Studio Code del test n=3

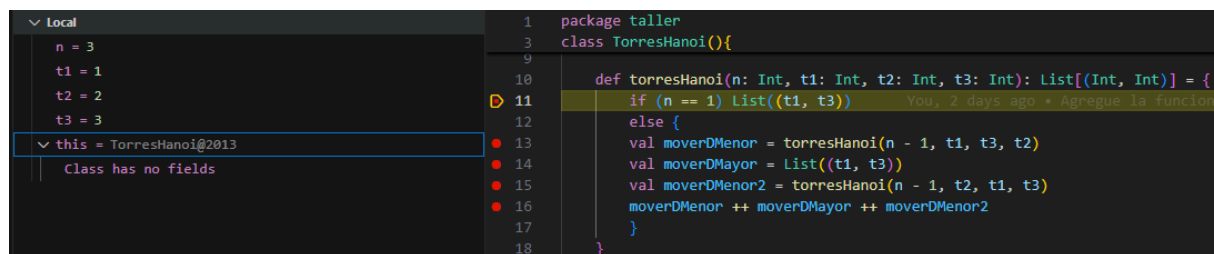
Funcion:

```
def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {  
  if (n == 1) List((t1, t3))  
  else {  
    val moverDMenor = torresHanoi(n - 1, t1, t3, t2)  
    val moverDMayor = List((t1, t3))  
    val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)  
    moverDMenor ++ moverDMayor ++ moverDMenor2  
  }  
}
```

Test:

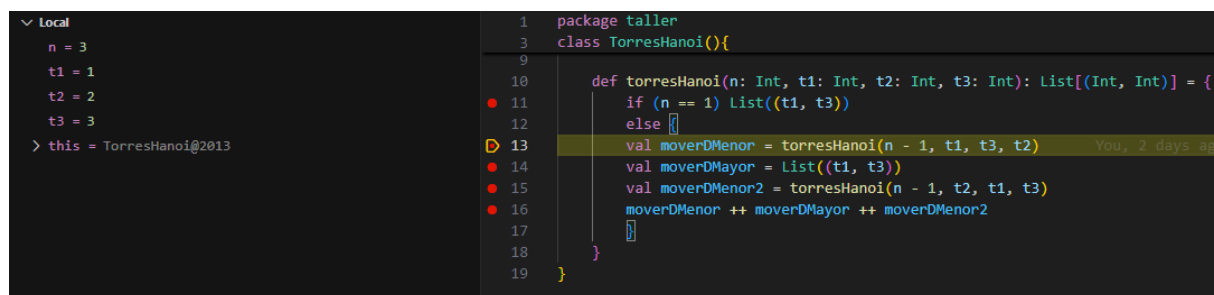
Vanessa Alexandra Durán Mona - 2359394
Juan Damián Cuervo Buitrago - 2359413
Juan Sebastián Rodas Ramírez - 2359681

```
// Test 1: Caso n=3
test("torresHanoi deberia retornar los movimientos correctos cuando n=3") {
  assert(torresHanoi.torresHanoi(3, 1, 2, 3) == List(
    (1,3), (1,2), (3,2),
    (1,3), (2,1), (2,3),
    (1,3)))
}
```



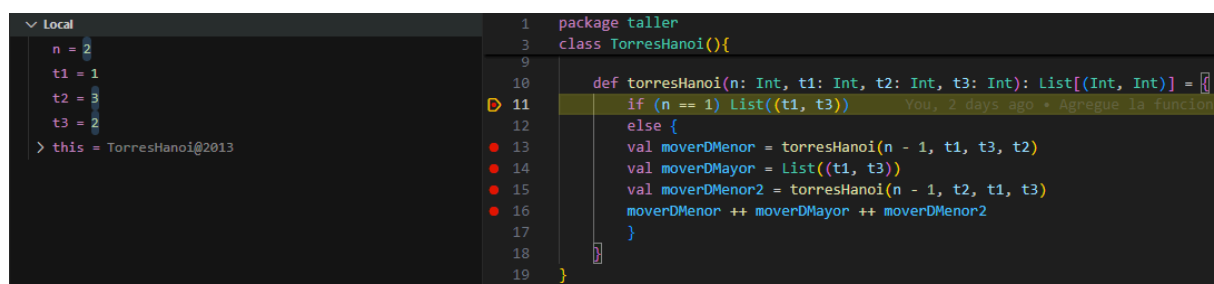
```
1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 1. En este caso se observa el momento en que el test entra en la función con $n=3$ y se está por evaluar en $\text{if}(n==1)$.



```
1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 2. Acá podemos ver que el programa rechaza la primera condición al ser $n \neq 1$ y se prepara para hacer los respectivos cambios impuestos en el `val moverDMenor`.



```
1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 3. El programa ejecutó lo dado en el `val moverDMenor`, restándole 1 a n ($n-1$) y cambiando la posición en el programa de los pilares $t2$ y $t3$, siendo ahora $t2=3$ y $t3=2$. Además, se alista para evaluar la primera condición.

Vanessa Alexandra Durán Mona - 2359394

Juan Damián Cuervo Buitrago - 2359413

Juan Sebastián Rodas Ramírez - 2359681

```
Local
n = 2
t1 = 1
t2 = 3
t3 = 2
> this = TorresHanoi@2013

1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
19 }
```

Paso 4. Al ser de nuevo n diferente de 1, el programa salta a evaluar el else y se prepara para ejecutar el primer val.

```
Local
n = 1
t1 = 1
t2 = 2
t3 = 3
> this = TorresHanoi@2013

1 package taller
3 class TorresHanoi(){
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 5. El programa ejecutó lo dado en el val moverDMenor, restándole 1 a n (n-1) y cambiando la posición en el programa de los pilares t2 y t3, siendo ahora de nuevo t2=2 y t3=3. Además, se alista para evaluar la primera condición

```
VARIABLES
Local
> torresHanoi() = $colon$colon@2018 "List((1,3))"
n = 2
t1 = 1
t2 = 3
t3 = 2
> this = TorresHanoi@2013

app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi
1 package taller
3 class TorresHanoi(){
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
19 }
```

Paso 6. Al ser n=1, el programa añade (1,3) como parte del conjunto de movimientos en la función. Posteriormente, podemos ver que los valores de n, t1, t2, t3 vuelve a ser los del Paso 4 y se encuentran preparados para evaluar el mismo val (val moverDMenor), teniendo ya un valor en la función, el cual debería asignarse al val moverDMenor.

```
VARIABLES
Local
n = 2
t1 = 1
t2 = 3
t3 = 2
moverDMenor = $colon$colon@2018 "List((1,3))"
serialVersionUID = 3
> head = Tuple2$mcII$sp@2021 "(1,3)"
> next = Nil$@2022 "List()"
> this = TorresHanoi@2013

app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi
1 package taller
3 class TorresHanoi(){
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
19 }
```


Vanessa Alexandra Durán Mona - 2359394

Juan Damián Cuervo Buitrago - 2359413

Juan Sebastián Rodas Ramírez - 2359681

Paso 7. Se guarda el primer movimiento que es el del disco de menor diámetro de t1 a t3. Y se prepara para ejecutar el siguiente val, ya que moverDMenor ya tiene un valor asignado.

```
Local
n = 2
t1 = 1
t2 = 3
t3 = 2
> moverDMenor = $colon$colon@2018 "List((1,3))"
> moverDMayor = $colon$colon@2025 "List((1,2))"
> this = TorresHanoi@2013

1 package taller
3 class TorresHanoi(){
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 8. Al ejecutar el val moverDMayor, toma el valor indicado, teniendo en cuenta que como vemos t3=t2 y, por tanto, tiene el valor de 2, por esto el moverDMayor=List((1,2)). Ahora se ejecutara val moverDMenor2, ya que moverDMayor ya tiene un valor asignado.

```
Local
n = 1
t1 = 3
t2 = 1
t3 = 2
> this = TorresHanoi@2013

1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 9. Se cambiaron los valores respectivos como indicaba val moverDMenor2, siendo t1=3 y t2=1. Como n es igual a 1 debería entrar en el if y guardar los valores en una lista.

```
> torresHanoi() = $colon$colon@2033 "List((3,2))"
Local
n = 2
t1 = 1
t2 = 3
t3 = 2
> moverDMenor = $colon$colon@2018 "List((1,3))"
> moverDMayor = $colon$colon@2025 "List((1,2))"
> this = TorresHanoi@2013

3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 10. Se crea el movimiento (3,2) y se prepara para asignarse al val moverDMenor2. Además, se modificaron los valores siendo los mismos que en el Paso 4.

```
VARIABLES
Local
n = 2
t1 = 1
t2 = 3
t3 = 2
> moverDMenor = $colon$colon@2018 "List((1,3))"
> moverDMayor = $colon$colon@2025 "List((1,2))"
> moverDMenor2 = $colon$colon@2033 "List((3,2))"
> this = TorresHanoi@2013

app > src > main > scala > taller > torresHanoi.scala > {} taller > torresHanoi > torresHanoi

1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 11. Se asigna el valor de la lista en moverDMenor2 y se prepara para juntar todas las listas asignadas hasta el momento (moverDMenor, moverDMayor, moverDMenor2) y asignarlas a la función.

Vanessa Alexandra Durán Mona - 2359394

Juan Damián Cuervo Buitrago - 2359413

Juan Sebastián Rodas Ramírez - 2359681

```
app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi

1 package taller
2 class TorresHanoi(){
3
4     def movsTorresHanoi(n: Int) : BigInt = {
5
6     }
7
8
9
10    def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11        if (n == 1) List((t1, t3))
12        else {
13            val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14            val moverDMayor = List((t1, t3))
15            val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16            moverDMenor ++ moverDMayor ++ moverDMenor2
17        }
18    }
19 }
```

Paso 12. Se juntaron todos los movimientos hechos y se guardaron en la función torresHanoi. Y el programa se prepara para ejecutar el val moverDMenor y guardar la lista de movimientos en este.

```
app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi

1 package taller
2 class TorresHanoi(){
3
4     def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
5         if (n == 1) List((t1, t3))
6         else {
7             val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
8             val moverDMayor = List((t1, t3))
9             val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
10            moverDMenor ++ moverDMayor ++ moverDMenor2
11        }
12    }
13 }
```

Paso 13. Guarda la lista de movimientos en el valor y se prepara para ejecutar val moverDMayor, ya que moverDMenor ya tiene un valor asignado

```
app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi

1 package taller
2 class TorresHanoi(){
3
4     def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
5         if (n == 1) List((t1, t3))
6         else {
7             val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
8             val moverDMayor = List((t1, t3))
9             val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
10            moverDMenor ++ moverDMayor ++ moverDMenor2
11        }
12    }
13 }
```

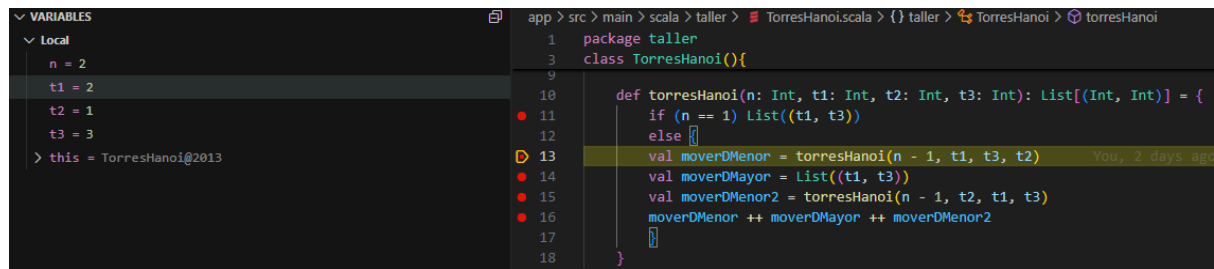
Paso 14. Guardo el movimiento (1,3) en el val moverDMayor y ejecuta el val moverDMenor2, debido a que moverDMayor ya tiene un valor asignado.

```
app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi

1 package taller
2 class TorresHanoi(){
3
4     def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
5         if (n == 1) List((t1, t3))
6         else {
7             val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
8             val moverDMayor = List((t1, t3))
9             val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
10            moverDMenor ++ moverDMayor ++ moverDMenor2
11        }
12    }
13 }
```

Paso 15. Cambia los valores como lo indicaba el val, siendo n=2 (3-1) y cambiando los valores de t1 y t2, siendo t1=2 y t2=1. Y evalúa la primera condición.

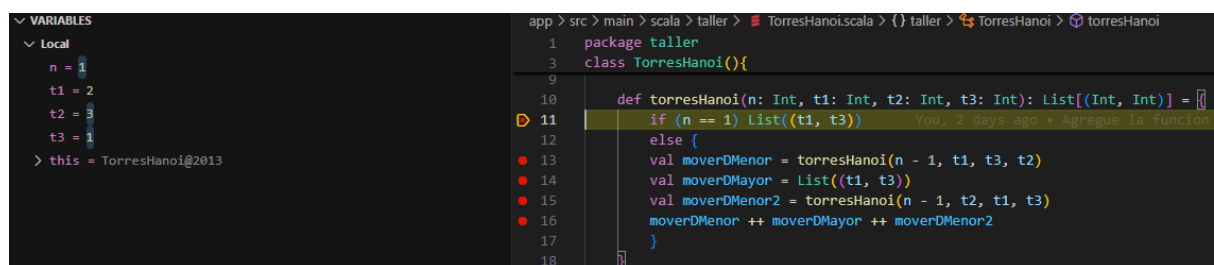
Vanessa Alexandra Durán Mona - 2359394
Juan Damián Cuervo Buitrago - 2359413
Juan Sebastián Rodas Ramírez - 2359681



```
app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi

1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

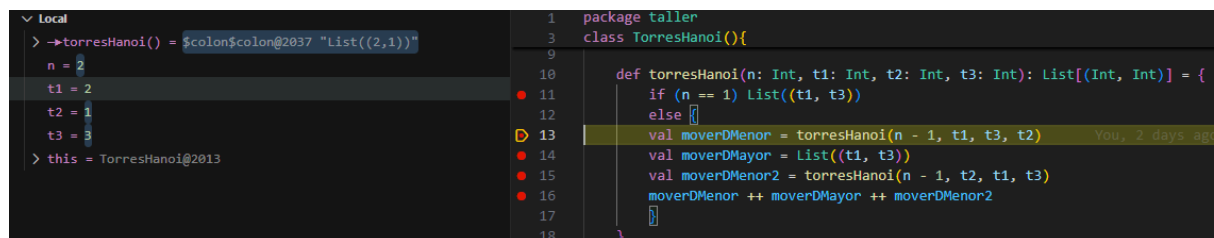
Paso 16. No cumple la condición y, por tanto, ejecuta una nueva función torresHanoi, en la cual ejecuta el val moverDMenor.



```
app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi

1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

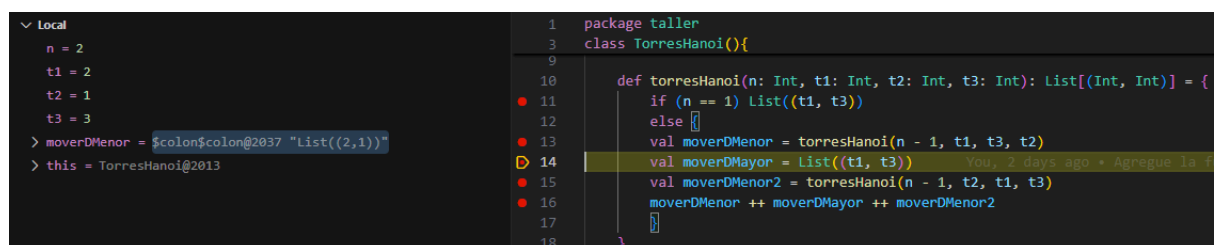
Paso 17. Cambia los valores como indicaba, siendo los cambios n=1, t2=3 y t3=1. Y evalúa la primera condición.



```
app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi

1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 18. Al cumplir la condición, guarda en la función List((2,1)) y se prepara para guardar el movimiento en moverDMenor.



```
app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi

1 package taller
3 class TorresHanoi(){
9
10 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
11   if (n == 1) List((t1, t3))
12   else {
13     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
14     val moverDMayor = List((t1, t3))
15     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
16     moverDMenor ++ moverDMayor ++ moverDMenor2
17   }
18 }
```

Paso 19. Guardo el movimiento List((2,1)) en moverDmenor y se prepara para crear y guardar el moverDmayor, de acuerdo a lo ordenado en la línea.

Vanessa Alexandra Durán Mona - 2359394

Juan Damián Cuervo Buitrago - 2359413

Juan Sebastián Rodas Ramírez - 2359681

The screenshot shows the REPL interface. On the left, the 'VARIABLES' pane lists local variables: `n = 2`, `t1 = 2`, `t2 = 1`, and `t3 = 3`. Below these, there are three REPL commands: `> moverDMenor = $colon$colon@2037 "List((2,1))"`, `> moverDMayor = $colon$colon@2040 "List((2,3))"`, and `> this = TorresHanoi@2013`. On the right, the Scala code for the `TorresHanoi` class is shown. The `torresHanoi` function is defined with parameters `(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)]`. The function body starts with a conditional: `if (n == 1) List((t1, t3))`. The rest of the function body is currently empty, with line numbers 10 through 18 visible.

Paso 20. Creo y guardo el movimiento `List(2,3)` en `moverDMayor` y se prepara para ejecutar lo dado en `moverDmenor2`.

The screenshot shows the REPL interface. On the left, the 'VARIABLES' pane lists local variables: `n = 1`, `t1 = 1`, `t2 = 2`, and `t3 = 3`. Below these, there is one REPL command: `> this = TorresHanoi@2013`. On the right, the Scala code for the `TorresHanoi` class is shown. The `torresHanoi` function is defined with parameters `(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)]`. The function body is now complete: `if (n == 1) List((t1, t3))`, followed by an `else` block containing `val moverDMenor = torresHanoi(n - 1, t1, t3, t2)`, `val moverDMayor = List((t1, t3))`, `val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)`, and `moverDMenor ++ moverDMayor ++ moverDMenor2`. The function ends with a closing brace. Line numbers 10 through 19 are visible.

Paso 21. Cambio los valores de acuerdo a lo ordenado, cambiado `n=1`, `t1=1` y `t2=2`, además ejecuta la primera condición, la cual debería ser verdadera.

The screenshot shows the REPL interface. On the left, the 'VARIABLES' pane lists local variables: `n = 2`, `t1 = 2`, `t2 = 1`, and `t3 = 3`. Below these, there are four REPL commands: `> ->torresHanoi() = $colon$colon@2043 "List((1,3))"`, `> moverDMenor = $colon$colon@2037 "List((2,1))"`, `> moverDMayor = $colon$colon@2040 "List((2,3))"`, and `> this = TorresHanoi@2013`. On the right, the Scala code for the `TorresHanoi` class is shown. The `torresHanoi` function is defined with parameters `(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)]`. The function body is the same as in the previous screenshot. Line numbers 10 through 18 are visible.

Paso 22. Guarda el movimiento `(1,3)` en la función y se prepara para guardarlo como una lista en `moverDMenor2`, ya que los otros dos val ya tienen valores asignados.

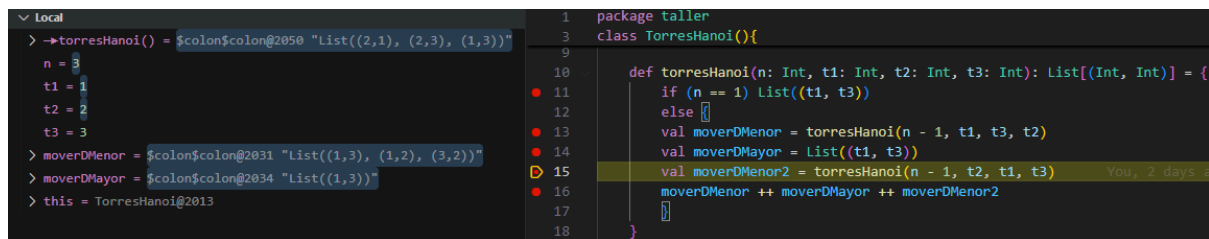
The screenshot shows the REPL interface. On the left, the 'VARIABLES' pane lists local variables: `n = 2`, `t1 = 2`, `t2 = 1`, and `t3 = 3`. Below these, there are four REPL commands: `> moverDMenor = $colon$colon@2037 "List((2,1))"`, `> moverDMayor = $colon$colon@2040 "List((2,3))"`, `> moverDMenor2 = $colon$colon@2043 "List((1,3))"`, and `> this = TorresHanoi@2013`. On the right, the Scala code for the `TorresHanoi` class is shown. The `torresHanoi` function is defined with parameters `(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)]`. The function body is the same as in the previous screenshot. Line numbers 10 through 18 are visible.

Paso 23. Guarda el movimiento en `moverDMenor2` y junta los tres vals (`moverDMayor`, `moverDMenor` y `moverDMenor2`) para guardarlos en la función `torresHanoi`.

Vanessa Alexandra Durán Mona - 2359394

Juan Damián Cuervo Buitrago - 2359413

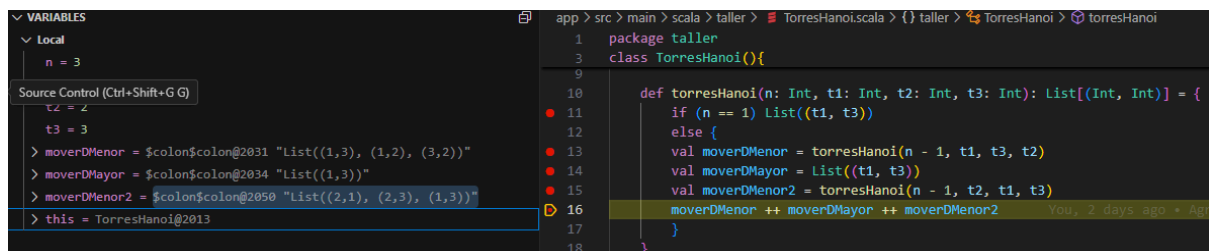
Juan Sebastián Rodas Ramírez - 2359681



```
Local
> ->torresHanoi() = $colon$colon@2050 "List((2,1), (2,3), (1,3))"
n = 3
t1 = 1
t2 = 2
t3 = 3
> moverDMenor = $colon$colon@2031 "List((1,3), (1,2), (3,2))"
> moverDMayor = $colon$colon@2034 "List((1,3))"
> this = TorresHanoi@2013

1 package taller
3 class TorresHanoi(){
9
10
11 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
12   if (n == 1) List((t1, t3))
13   else {
14     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
15     val moverDMayor = List((t1, t3))
16     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
17     moverDMenor ++ moverDMayor ++ moverDMenor2
18   }
19 }
```

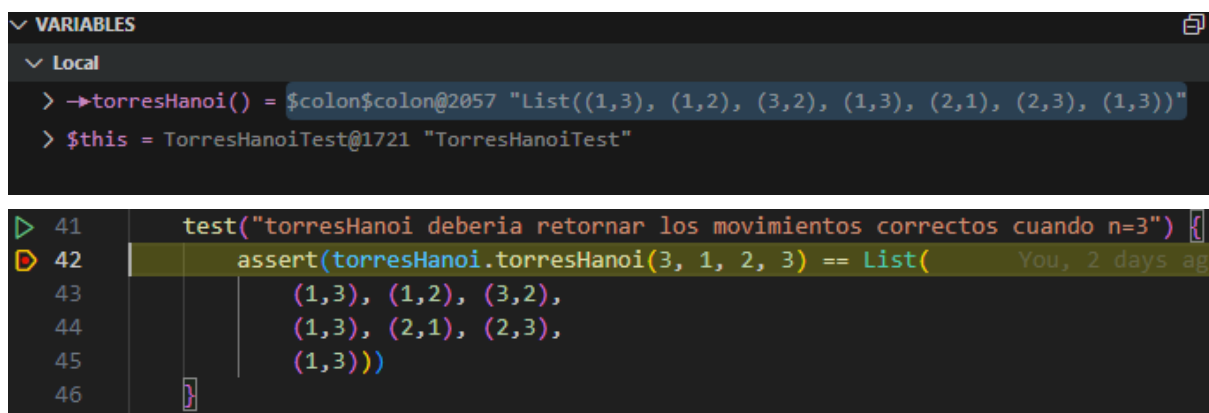
Paso 24. Regresa a la función torresHanoi anterior donde estaban guardadas los otros movimientos creados del Paso 1 al Paso 14 para guardar los nuevos valores ejecutados del Paso 16 al Paso 24 en otra función, guardando estos datos en el val vacío moverDMenor2.



```
VARIABLES
Local
n = 3
Source Control (Ctrl+Shift+G)
t2 = 2
t3 = 3
> moverDMenor = $colon$colon@2031 "List((1,3), (1,2), (3,2))"
> moverDMayor = $colon$colon@2034 "List((1,3))"
> moverDMenor2 = $colon$colon@2050 "List((2,1), (2,3), (1,3))"
> this = TorresHanoi@2013

app > src > main > scala > taller > TorresHanoi.scala > {} taller > TorresHanoi > torresHanoi
1 package taller
3 class TorresHanoi(){
9
10
11 def torresHanoi(n: Int, t1: Int, t2: Int, t3: Int): List[(Int, Int)] = {
12   if (n == 1) List((t1, t3))
13   else {
14     val moverDMenor = torresHanoi(n - 1, t1, t3, t2)
15     val moverDMayor = List((t1, t3))
16     val moverDMenor2 = torresHanoi(n - 1, t2, t1, t3)
17     moverDMenor ++ moverDMayor ++ moverDMenor2
18   }
19 }
```

Paso 25. Guarda los movimientos en moverDMenor2 y se prepara para juntar los datos y guárdalos en la función torresHanoi.



```
VARIABLES
Local
> ->torresHanoi() = $colon$colon@2057 "List((1,3), (1,2), (3,2), (1,3), (2,1), (2,3), (1,3))"
> $this = TorresHanoiTest@1721 "TorresHanoiTest"

41 test("torresHanoi deberia retornar los movimientos correctos cuando n=3") {
42   assert(torresHanoi.torresHanoi(3, 1, 2, 3) == List(
43     (1,3), (1,2), (3,2),
44     (1,3), (2,1), (2,3),
45     (1,3)))
46 }
```

Paso 26. Habiendo evaluado la función n=3, dando como resultado List ((1,3), (1,2), (3,2), (1,3), (2,1), (2,3), (1,2)), que efectivamente es el resultado esperado

ARGUMENTACIÓN SOBRE LA CORRECCIÓN

- Demostración por inducción estructural de la función maxIt.

- Caso *base*: Lista vacía. Cuando la lista está vacía, **maxIt** lanza una excepción NoSuchElementException, lo cual es correcto, ya que no existe máximo en una lista vacía. En la función auxiliar **maxItAux**, la lista vacía detiene la recursión y regresa el valor máximo acumulado hasta ese punto. El caso base se cumple, ya que si se llega a una lista vacía, **maxItAux** devuelve correctamente el valor acumulado de max.

- *Paso Inductivo*: Lista no vacía. Se asume que la función funciona correctamente para una lista de longitud n . Ahora, se prueba que la función sigue siendo correcta para una lista de longitud $n + 1$.

Suponemos que la lista no vacía es de la forma $l = x :: xs$, donde x es la cabeza de la lista y xs es la cola.

La función se comporta de la siguiente manera:

1. Se compara x con el valor máximo actual.
2. Se llama recursivamente a **maxItAux(xs, max)** actualizando el valor mínimo si $x > \text{min}$.

- *Hipótesis de inducción*: se asume que **maxItAux(xs, max)** devuelve el valor máximo correcto para la lista xs . Ahora se necesita probar que **maxItAux(x :: xs, max)** también devuelve el valor máximo correcto.

Existen dos casos que se consideran:

1. si $x > \text{max}$, entonces el nuevo valor máximo será x . La función recursiva es llamada con **maxItAux(xs, x)**, y por hipótesis inductiva, esto devuelve el valor máximo correcto para xs , que junto con x será el valor máximo correcto para la lista completa $x::xs$.
2. si $x \leq \text{max}$, la recursión continua con **maxItAux(xs, max)** sin cambiar el valor máximo actual. Por hipótesis inductiva, **maxItAux(xs, max)** devolverá el valor máximo correcto para xs , y como x no es mayor que el valor actual de max , este seguirá siendo el máximo correcto para toda la lista $x::xs$.

• Demostración por inducción estructural de la función **maxLin**.

- *Caso base*: Lista vacía. Cuando la lista está vacía, **maxLin** lanza la excepción, **NoSuchElementException**, siendo lo correcto, ya que no existe un máximo en una lista vacía. Por lo tanto, el caso base se cumple correctamente.

- *Paso Inductivo*: Lista no vacía. Se supone que la función **maxLin** funciona correctamente para una lista de tamaño n . Entonces, se prueba que sigue siendo correcta para una lista de tamaño $n+1$.

- *Hipótesis de inducción*: Se supone que la función devuelve el valor máximo correcto para una lista tail de tamaño n , ósea, **maxLin(tail)** devuelve el máximo de la lista **tail**.

Ahora se prueba que **maxLin(head::tail)** devuelve el máximo correcto para la lista **head::tail**, de tamaño $n+1$.

Entonces la función realiza los siguientes pasos:

1. Compara el valor de la cabeza (**head**) con el valor máximo de la cola (**maxLin(tail)**).

2. Si **head > maxLin(tail)**, entonces el nuevo valor máximo es la cabeza (**head**). De lo contrario, el valor máximo sigue siendo el mismo (**maxLin(tail)**).

Hay dos casos que considerar:

1. Si **head > maxLin(tail)**, entonces la función devuelve a la cabeza (**head**), que es el valor máximo correcto para toda la lista **head::tail**, dado que ya sabemos que **maxLin(tail)** devuelve el valor máximo de tail por hipótesis inductiva.
2. Si **head <= maxLin(tail)**, entonces la función devuelve **maxLin(tail)**, que por hipótesis inductiva es el máximo de la lista **tail**, lo que garantiza que es también el valor máximo corrector para la lista completa **head::tail**.

- **Demostración por inducción matemática de movsTorresHanoi.**

- *Caso base: n = 1.* En este caso, la función **movsTorresHanoi** debe devolver 1.
movsTorresHanoi(1) = 1

- *Hipótesis de inducción:* Suponemos que la función **movsTorresHanoi** es correcta para todo **n** menor o igual a **k**, es decir:

$$\text{movsTorresHanoi}(n) = 2^n - 1, \text{ para todo } n \leq k$$

- *Paso inductivo:* se debe demostrar que si funciona para $n \leq k$, debe funcionar para $n = k+1$. Entonces:

$$\text{movsTorresHanoi}(k+1) = 2^{k+1} - 1$$

Utilizando la definición de la función **movsTorresHanoi**:

$$\text{movsTorresHanoi}(k+1) = 2 * \text{movsTorresHanoi}(k) + 1$$

Por tanto, sustituyendo en la hipótesis de inducción:

$$\begin{aligned} \text{movsTorresHanoi}(k+1) &= 2 * (2^k - 1) + 1 \\ &= 2 * 2^k - 2 * 1 + 1 \\ &= 2^{k+1} - 2 + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

- **Demostración por inducción estructural de torresHanoi.**

La función torresHanoi(n: Int, t1: Int, t2: Int, t3: Int) se comporta de la siguiente manera:

- Caso base: Si $n=1$, el único movimiento es mover el disco de t1 a t3.
- Caso recursivo: Si $n>1$ la función realiza tres pasos:
 - Mueve los $n-1$ discos de t1 a t2, utilizando t3 como torre auxiliar.
 - Mueve el disco más grande (el disco n) de t1 a t3. Por eso en el paso anterior le restamos el mismo.
 - Mueve los $n-1$ discos de t2 a t3, utilizando t1 como torre auxiliar.
- caso base: **n = 1**. En este caso, la función torresHanoi debe retornar una lista con una única tupla que sería (t1, t3), indicando que el disco se mueve directamente de la torre t1 a la torre t3.
- Hipótesis de inducción: Suponiendo que la función torresHanoi es correcta para $n = k$, esta deberá retornar una lista con $(2^k - 1)$ tuplas, que indican los movimientos necesarios para trasladar k discos de t1 a t3.
- paso inductivo: se debe demostrar que si funciona para $n \leq k$, debe funcionar para $n = k+1$. Entonces:
 1. Primer movimiento recursivo:
val moverDMenor = torresHanoi((k+1) - 1, t1, t3, t2)
Este paso mueve los k discos de t1 a t2 usando t3 como intermediaria.
 2. Mover el disco más grande:
val moverDMayor = List((t1, t3))
Este paso mueve el disco k+1 de t1 a t3, este paso es correcto para cualquier $k>0$, ya que no hay otro disco más grande.
 3. Segundo movimiento recursivo:
val moverDMenor2 = torresHanoi((k+1) - 1, t2, t1, t3)
Este paso mueve los k discos de t2 a t3 usando t1 como intermediaria.
 4. Combinación de resultados:
Al terminar todo este proceso la función devuelve:
moverDMenor ++ moverDMayor ++ moverDMenor2

Combinando los movimientos en tres partes:

1. Los movimientos necesarios para trasladar los k discos pequeños de t1 a t2.
2. El movimiento del disco más grande de t1 a t3.
3. Los movimientos necesarios para trasladar los k discos pequeños de t2 a t3.

Por inducción estructural, se ha demostrado que la función `torresHanoi(n, t1, t2, t3)` es correcta para cualquier número de discos n . Cada llamada recursiva sigue una estructura lógica bien definida que se apoya en la validez de casos más pequeños (gracias a la hipótesis de inducción).

CONCLUSIONES

- La recursión se vuelve un elemento esencial para resolver problemas que requieran una descomposición estructural, es decir, cuando un problema complejo puede desglosarse en subproblemas más pequeños y simples, demostrando así, como la recursión es una herramienta poderosa para abordar problemas complejos en la programación.
- Se reconocen las diferencias en cuanto a eficiencia de los algoritmos para hallar el máximo de una lista. En la función `maxIt` se implementó la recursión de cola y en la función `maxIn` la recursión lineal, aunque ambos cumplen la misma función, la realidad es que la optimización puede llevarse a cabo en el algoritmo que usa recursión de cola, consumiendo a su vez menos espacio en memoria, mientras que en la función `MaxLin` no permite la optimización, por lo que puede desencadenar en un consumo de memoria mayor a medida que crece la lista.
- Al abordar el problema de las Torres de Hanoi, la función permite calcular de manera eficiente el número de movimientos necesarios para cualquier cantidad de discos, evidenciando el crecimiento exponencial de la complejidad con el incremento de discos. Este tipo de problemas, al ser desglosados en subproblemas más simples y resueltos mediante la recursión, demuestra la potencia y versatilidad de este paradigma para afrontar desafíos complejos en programación.