TALLER I: INFORME DEL TALLER
FUNDAMENTOS DE PROGRAMACIÓN FUNCIONAL Y CONCURRENTE

JUAN DIEGO ESCOBAR TRIVIÑO - 2359393
DIEGO LENIS DELGADO -
GABRIEL URAZA GARCIA - 2359594
PROGRAMA ACADÉMICO - 3743

CARLOS ANDRES DELGADO
DOCENTE

UNIVERSIDAD DEL VALLE
16/09/2024
TULUÁ VALLE DEL CAUCA

TABLA DE CONTENIDO

INFORME DE PROCESOS:

Proceso de la función maxLin

Ejemplo con : val lista = List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)

Pila de llamadas:

```
maxLin(List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5))
  -> maxLin(List(1, 4, 1, 5, 9, 2, 6, 5, 3, 5))
    -> maxLin(List(4, 1, 5, 9, 2, 6, 5, 3, 5))
      -> maxLin(List(1, 5, 9, 2, 6, 5, 3, 5))
        -> maxLin(List(5, 9, 2, 6, 5, 3, 5))
          -> maxLin(List(9, 2, 6, 5, 3, 5))
            -> maxLin(List(2, 6, 5, 3, 5))
              -> maxLin(List(6, 5, 3, 5))
                -> maxLin(List(5, 3, 5))
                  -> maxLin(List(3, 5))
                    -> maxLin(List(5))
                      -> maxLin(Nil)
```

- maxLin(Nil) retorna una excepción porque la lista está vacía.
- Luego, se resuelve el caso base para maxLin(List(5)) como 5.
- La pila se deshace comparando el máximo entre 5 y 3, luego 5 y 5, y así sucesivamente hasta obtener el máximo de toda la lista.

```
> Task :app:run
Máximo calculado con recursión lineal: 9
```

Proceso Función maxIt

Ejemplo con : val lista = List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)

```
maxIt(List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5))
  -> maxItAux(List(1, 4, 1, 5, 9, 2, 6, 5, 3, 5), 3)
    -> maxItAux(List(4, 1, 5, 9, 2, 6, 5, 3, 5), 3)
      -> maxItAux(List(1, 5, 9, 2, 6, 5, 3, 5), 4)
        -> maxItAux(List(5, 9, 2, 6, 5, 3, 5), 5)
          -> maxItAux(List(9, 2, 6, 5, 3, 5), 9)
            -> maxItAux(List(2, 6, 5, 3, 5), 9)
              -> maxItAux(List(6, 5, 3, 5), 9)
                -> maxItAux(List(5, 3, 5), 9)
                  -> maxItAux(List(3, 5), 9)
                    -> maxItAux(List(5), 9)
                      -> maxItAux(Nil, 9)
```

- La función maxItAux recorre la lista de manera iterativa, manteniendo el máximo actual en cada llamada recursiva.
- No hay acumulación de llamadas pendientes, ya que se trata de una recursión de cola.

Resultado:

```
Máximo calculado con recursión lineal: 9
Máximo calculado con recursión de cola: 9
```

Proceso Función Buscar Elemento

Ejemplo 1: valor de búsqueda (5)

Ejemplo con : val lista = List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)

```
val existeElemento1 = buscarLista.buscarElemento(lista, 5)
```

Comienza el proceso

```
buscarElemento(List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5), 5)
  -> buscarElemento(List(1, 4, 1, 5, 9, 2, 6, 5, 3, 5), 5)
    -> buscarElemento(List(4, 1, 5, 9, 2, 6, 5, 3, 5), 5)
      -> buscarElemento(List(1, 5, 9, 2, 6, 5, 3, 5), 5)
        -> buscarElemento(List(5, 9, 2, 6, 5, 3, 5), 5)
```

en la última llamada encuentra el elemento ‘5’, entonces retorna ‘true’

Resultado:

```
¿El elemento 5 está en la lista? true
```

Ejemplo 2: valor de búsqueda (10)

Ejemplo con : val lista = List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)

```
val existeElemento2 = buscarLista.buscarElemento(lista, 10)
```

Comienza el proceso

```
buscarElemento(List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5), 10)
  -> buscarElemento(List(1, 4, 1, 5, 9, 2, 6, 5, 3, 5), 10)
    -> buscarElemento(List(4, 1, 5, 9, 2, 6, 5, 3, 5), 10)
      -> buscarElemento(List(1, 5, 9, 2, 6, 5, 3, 5), 10)
        -> buscarElemento(List(5, 9, 2, 6, 5, 3, 5), 10)
          -> buscarElemento(List(9, 2, 6, 5, 3, 5), 10)
            -> buscarElemento(List(2, 6, 5, 3, 5), 10)
              -> buscarElemento(List(6, 5, 3, 5), 10)
                -> buscarElemento(List(5, 3, 5), 10)
                  -> buscarElemento(List(3, 5), 10)
                    -> buscarElemento(List(5), 10)
                      -> buscarElemento(Nil, 10)
```

No encuentra el elemento 10 en ninguna parte de la lista, retorna un false

Resultado:

```
¿El elemento 5 está en la lista? true
¿El elemento 10 está en la lista? false
```

Ejemplo 3: valor de búsqueda (3)

Ejemplo con : val lista = List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)

```
val existeElemento3 = buscarLista.buscarElemento(lista, 3)
```

Comienza el proceso

```
buscarElemento(List(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5), 3)
```

en esta llamada se encuentra el 3 inmediatamente, retorna true

Resultado:

```
¿El elemento 5 está en la lista? true
¿El elemento 10 está en la lista? false
¿El elemento 3 está en la lista? true
```

Proceso Función Torres de Hanoi

Ejemplo 1: con 2 discos

1. Llamada inicial con 2 discos

```
val movimientos2Discos = hanoi.torresHanoi(2, 1, 2, 3)
```

   Se inicia la resolución del problema moviendo 2 discos de la torre 1 a la torre 3 usando la torre 2 como intermediaria

2. Luego se llama con 2 discos de la torre 1, usando la torre 2 como auxiliar

```
torresHanoi(2, 1, 2, 3)
```

3. Luego se mueve un disco de la torre 1 a la torre 2 usando la torre 3 como auxiliar

```
torresHanoi(1, 1, 3, 2)
```

4. Se mueve el disco de la torre 1 a la torre 2

```
List((1, 2))
```

5. Se mueve el disco 2 de la torre 1 a la torre 3

```
List((1, 3))
```

6. Mueve un disco de la torre 2 a la torre 3 usando la torre 1 como auxiliar

```
torresHanoi(1, 2, 1, 3)
```

7. Mueve el disco de la torre 2 a la torre 3

```
List((2, 3))
```

8. Solución completa para mover los 2 discos de la torre 1 a la torre 3

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

Resultado final por consola:

```
Movimientos para resolver las Torres de Hanoi con 2 discos:
Mover disco de torre 1 a torre 2
Mover disco de torre 1 a torre 3
Mover disco de torre 2 a torre 3
```

Ejemplo 2 : con 3 discos

```
val movimientos3Discos = hanoi.torresHanoi(3, 1, 2, 3)// Se inicia el proceso
torresHanoi(3, 1, 2, 3) // Se mueve 2 discos de la torre 1 a la torre 2
torresHanoi(2, 1, 3, 2) // Se mueve 1 disco de la torre 1 a la torre 3
torresHanoi(1, 1, 2, 3) // Se mueve el disco 1 de la torre 1 a la torre 2
List((1, 2)) // Se mueve el disco 1 de la torre 1 a la torre 2
List((1, 3)) // Se mueve el disco 2 de la torre 1 a la torre 3
torresHanoi(1, 2, 1, 3) // Se mueve el disco 1 de la torre 2 a la torre 3
List((2, 3)) // Se mueve el disco 1 de la torre 2 a la torre 3
List((1, 3)) // Se mueve el disco 3 de la torre 1 a la torre 3
torresHanoi(2, 2, 1, 3) // Se mueve 2 discos de la torre 2 a la torre 3
List((1, 2), (1, 3), (2, 3), (1, 3), (2, 1), (2, 3), (1, 3)) // se necesitarion 7 movimientos en total
```

Resultado final de consola:

```
Movimientos para resolver las Torres de Hanoi con 3 discos:
Mover disco de torre 1 a torre 3
Mover disco de torre 1 a torre 2
Mover disco de torre 3 a torre 2
Mover disco de torre 1 a torre 3
Mover disco de torre 2 a torre 1
Mover disco de torre 2 a torre 3
Mover disco de torre 1 a torre 3
```

Ejemplo 3: con 4 discos

```
val movimientos4Discos = hanoi.torresHanoi(4, 1, 2, 3) // -> Llamada a la funcion para iniciar
torresHanoi(4, 1, 2, 3) //-> esto mueve 3 discos de la tore 1 a la 2
torresHanoi(3, 1, 3, 2) //-> luego se mueven 2 discos de la torre 1 a la torre 3
torresHanoi(2, 1, 2, 3) //-> esto mueve 1 disco de la torre 1 a la torre 2
torresHanoi(1, 1, 3, 2) //-> despues el disco 1 de la torre 1 se mueve a  la torre 2
List((1, 2)) //-> se mueve el disco 1 de la torre 1 a la torre 2
List((1, 3)) //-> despues se mueven el disco 2 de la torre 1 a la torre 3
torresHanoi(1, 2, 1, 3) //-> mueve el disco de la torre 2 a la torre 3
List((2, 3)) //-> mueve el disco 1 de la torre 2 a la torre 3
List((1, 2), (1, 3), (2, 3)) //-> ahi son los movimientos para mover 2 discos de la torre 1 a la torre 3
List((1, 2)) //-> mueve el disco 3 de la torre 1 a la torre 2
torresHanoi(2, 3, 1, 2) //-> de mueven 2 discos de la torre 3 a la torre 2
torresHanoi(1, 3, 2, 1) //-> se mueve el disco 1 de la torre 3 a la torre 1
List((3, 1)) //-> se mueve el disco 1 de la torre 3 a la torre 1
List((3, 2)) //-> muevee el disco 2 de la torre 3 a la torre 2
torresHanoi(1, 1, 3, 2) //-> se mueveueve el disco 1 de la torre 1 a la torre 2
List((1, 2)) //-> mueve el disco 1 de la torre 1 a la torre 2
List((1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2))
//-> movimientos para mover 3 discos de la torre 1 a la torre 2
List((1, 3)) //-> muueve el disco 4 de la torre 1 a la torre 3
torresHanoi(3, 2, 1, 3) //-> se mueven 3 discos de la torre 2 a la torre 3 u
torresHanoi(2, 2, 3, 1) //-> se mueven  2 discos de la torre 2 a la torre 1
torresHanoi(1, 2, 1, 3) //-> mueve el disco 1 de la torre 2 a la torre 3
List((2, 3)) //-> mueve el disco 1 de la torre 2 a la torre 3
List((2, 1)) //-> mueve el disco 2 de la torre 2 a la torre 1
torresHanoi(1, 3, 2, 1) //-> mueve el disco 1 de la torre 3 a la torre 1
List((3, 1)) //-> mueve el disco 1 de la torre 3 a la torre 1
List((2, 3), (2, 1), (3, 1)) //-> movimientos para poder mover 2 discos de la torre 2 a la torre 1
List((2, 3)) //-> mueve el disco 3 de la torre 2 a la torre 3
torresHanoi(2, 1, 2, 3) //-> mueve 2 discos de la torre 1 a la torre 3
torresHanoi(1, 1, 3, 2) //-> mueve el disco 1 de la torre 1 a la torre 2
List((1, 2)) //-> mueve el disco 1 de la torre 1 a la torre 2
List((1, 3)) //-> mueve el disco 2 de la torre 1 a la torre 3
torresHanoi(1, 2, 1, 3) //-> mueve el disco 1 de la torre 2 a la torre 3
List((2, 3)) //-> mueve el disco 1 de la torre 2 a la torre 3
List((1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2), (1, 3), (2, 1), (2, 3), (1, 3), (2, 3), (1, 2), (1, 3), (2, 3))
//-> En total se requirieron 15 movimientos para poder acomodar los discos
```

Resultado final por consola:

```
Movimientos para resolver las Torres de Hanoi con 4 discos:
Mover disco de torre 1 a torre 2
Mover disco de torre 1 a torre 3
Mover disco de torre 2 a torre 3
Mover disco de torre 1 a torre 2
Mover disco de torre 3 a torre 1
Mover disco de torre 3 a torre 2
Mover disco de torre 1 a torre 2
Mover disco de torre 1 a torre 3
Mover disco de torre 2 a torre 3
Mover disco de torre 2 a torre 1
Mover disco de torre 3 a torre 1
Mover disco de torre 2 a torre 3
Mover disco de torre 1 a torre 2
Mover disco de torre 1 a torre 3
Mover disco de torre 2 a torre 3
```

INFORME DE CORRECCIÓN

Argumentación sobre corrección:

Método de clase ListTail - maxLin

```
  def maxLin(l: List[Int]): Int = {
    //si la lista esta vacia lanza una excepcion
    if (l.isEmpty) 0
    else if (l.tail.isEmpty) l.head//si la lista tiene un solo
elemento retorna ese elemento
    else {
      //si la lista tiene mas de un elemento
      val maxTail = maxLin(l.tail)
      //retorna el maximo entre la cabeza de la lista y el maximo de
la cola
      if (l.head > maxTail) l.head else maxTail
    }
  }
```

Explicación de procesos:

1 - El método recibe una lista de “n” elementos, que son números enteros.

2 - En principio verifica que no sea un alista vacía, evitando realizar procesos innecesarios (se ahora buscar elementos que NO va a encontrar)

3 - Realiza una comparación entre el primer elemento de la lista y la cola de la misma, si no hay cola, existe un solo elemento y no puede haber un elemento de mayor valor que ese.

4 - El otro caso que se puede dar es que la lista contenga más elementos, en ese caso la función es llamada de nuevo sobre la cola de la lista original, esto inicia la recursión lineal que realiza el mismo proceso para el resto de la lista.

5 - El último proceso de este método compara el primer elemento de la lista con el valor *maxTail,* que es diferente para cada iteración de la recursión.

En ultima instancia e independientemente de la longitud de la lista o su contenido, el último llamado a la función se resuelve ya que este llamado se realizará sobre una lista vacía, al compararse con el llamado anterior se retorna l.head, lo que retorna un valor que puede compararse en la llamada anterior a la función, hasta llegar al primer llamado, entregando como resultado el número de mayor valor.

A medida que estos llamados se resuelven se asegura:

- todo elemento es comparado
- se obtiene el elemento de mayor valor en la lista
- no se hacen comparaciones innecesarias
- la posición en la que se encuentra el elemento que se busca no afecta el resultado

Método de clase ListTail - maxIt

```
def maxIt(l: List[Int]): Int = {
    if (l.isEmpty)0


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

```
}
```

Explicación de procesos.

La función recibe una lista de enteros y emplea otra función cómo auxiliar

1- anotación tailrec para indicar al compilador que se ejecute como iteración.

2 - La función auxiliar recibe una lista y un valor entero.

3 - En primer lugar se verifica que la lista no esté vacía, si lo está retorna 0

4 - si la lista contiene algún elemento , la misma función auxiliar es llamada de nuevo para los elementos de la cola, el segundo argumento de la llamada a la función es la cabeza en caso de ser mayor a valor maximo almacenado, de lo contrario este valor es el valor max

5 - El primer llamado a la función auxiliar se realiza con la lista y la cabeza de la lista cómo primer argumento

6 - Siempre que se hagan estos llamados, se almacenará un nuevo valor como “max” este se obtiene de la cabeza de la lista y después se compara con el llamado de la función para el resto de la lista, esto se repite hasta el final de la recursión y de la lista hasta haber comparado cada elemento.

A medida que estos llamados se resuelven se asegura:

- todo elemento es comparado
- se obtiene el elemento de mayor valor en la lista
- no se hacen comparaciones innecesarias
- la posición en la que se encuentra el elemento que se busca no afecta el resultado
- el proceso se optimiza empleando una recursión de cola.

Metodo de clase Hanoi - movsTorresHanoi

```
def movsTorresHanoi (n: Int): BigInt = {
        if (n == 0) 0
        else if (n == 1) 1
```

```
        else 2 * movsTorresHanoi(n - 1) + 1

    }
```

1 - La función recibe n cómo único argumento, que representa el numero de discos que se intentan llevar de una vara a otra.

2 - si se intenta ejecutar con 0 como parametro, es decir sin ningún disco, retorna el valor 0

3 - si hay exactamente un disco, solo toma un movimiento

4 - de haber más de un disco, se llama de nuevo la función para n-1, que es igual a repetir el procedimiento para todos los discos, menos el que ya se movió, este valor se multiplica por dos ya que hay que realizar un movimiento nuevamente para pasar de la vara auxiliar a la última, se suma uno para contar el movimiento que se realiza en cada llamada

5 - al resolverse la pila de llamados se suman los resultados en cad iteración y este resultado es igual a 2 + 2 + 2 …….., excepto para el último, que retorna 1 al "solo quedar 1 disco" (se llama para n-1 hasta que n = 1)

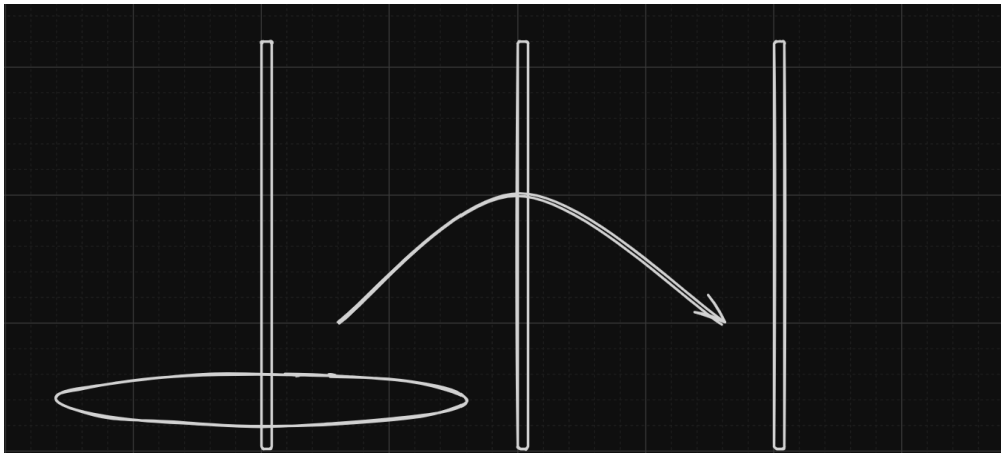
El resultado que se obtiene es igual a la fórmula (2^n) - 1, que determina el número mínimo de movimientos para resolver las torres usando el mínimo de movimientos.

Inducción-Movimientos Hanoi

$(2^n - 1) > 0$

#Paso_base (n debe ser mayor o igual a 1)

$(2^1 - 1) = 1$



CONCLUSIONES:

Durante el desarrollo del taller, se implementaron aplicaciones de cola, recursión lineal y recursión simple, en los primeros ejercicios del taller, en la cual es la clase ListTail específicamente en su primera función MaxLin sé usó un manejo de excepciones, además de verificaciones de listas vacías y diferentes comparaciones entre la cabeza de la lista y su cola en comparación de la otra función donde se procuró utilizar un modelo de comparación de una recursión lineal.

De este primer punto se lograron los indicadores de funcionalidad propuestos por el docente.

Al estudiar el algoritmo para resolver el problema de las torres de hanoi, se puede alcanzar una mejor comprensión sobre el concepto de recursión, ya que el algoritmo obliga a pensar con una mejor lógica para resolver problemas más complejos por medio de la recursión ya que el problema principal comienza a descomponerse en problemas más pequeños.