Taller 1 Programación

Nombres:

Yulieth Tatiana Rengifo Rengifo – 2359748

Pedro José López Quiroz - 2359423

Juan José Valencia Jimenez - 2359567

Docente:

Carlos Andres Delgado

Universidad del Valle

Sede Tuluá

Informe de procesos Maximo Lista

1. **Función “maxLin”: proceso recursivo**

La función “maxLin” calcula el valor máximo de una lista usando recursión. En cada llamada, compara el primer elemento de la lista con el valor máximo del resto de la lista.

Ejemplo:

val lista = List(3, 6, 2)

Pila de llamadas:

1. maxLin(List(3, 6, 2))

   - Compara “3” con el resultado de “maxLin(List(6, 2))”

2. maxLin(List(6, 2))

   - Compara “6” con el resultado de “maxLin(List(2))”

3. maxLin(List(2))

   - Caso base: la lista tiene un solo elemento, retorna “2”.

Desenrollo de la pila:

3. maxLin(List(2)) = 2

2. maxLin(List(6, 2)) = max(6, 2) = 6

1. maxLin(List(3, 6, 2)) = max(3, 6) = 6

Resultado final: “6”

2. **Función “maxIt”: proceso iterativo (recursión de cola)**

La función “maxIt” encuentra el valor máximo de una lista usando recursión de cola. La función auxiliar “maxItAux” recibe el valor máximo hasta el momento y recorre el resto de la lista actualizándolo.

Ejemplo:

val lista = List(3, 6, 2)

Pila de llamadas:

1. maxIt(List(3, 6, 2)) llama a maxItAux(List(6, 2), 3)

2. maxItAux(List(6, 2), 3) llama a maxItAux(List(2), max(6, 3)) = 6

3. maxItAux(List(2), 6) llama a maxItAux(Nil, max(2, 6)) = 6

4. maxItAux(Nil, 6) retorna 6 ya que la lista está vacía.

Resultado final: “6”

2. Informe de corrección:

Argumentación sobre la corrección

1. **Función “maxLin”**

Demostraremos la corrección de la función usando **inducción estructural** sobre la lista “l”.

- Caso base: Si la lista l tiene un solo elemento, la función retorna ese elemento, que es el máximo de la lista.

- Paso inductivo: Supongamos que “maxLin” es correcta para una lista de longitud “n” (es decir, encuentra correctamente el valor máximo de una lista de longitud “n”). Entonces, para una lista de longitud “n+1”, la función compara el primer elemento con el máximo del resto de la lista (de longitud “n”), que por hipótesis inductiva es correcto. De esta forma, la función retorna el mayor entre ambos, lo que asegura que el valor retornado es el máximo de toda la lista.

2. **Función “maxIt”**

La corrección de “maxIt” se puede demostrar utilizando inducción sobre el tamaño de la lista l.

- Caso base: Si la lista es vacía, la función lanza una excepción, lo cual es el comportamiento esperado según la definición del problema, ya que no hay un valor máximo en una lista vacía. Si la lista tiene un solo elemento, “maxItAux” retorna ese elemento, que es el máximo de la lista.

- Paso inductivo: Supongamos que la función es correcta para una lista de longitud “n”. Para una lista de longitud “n+1”, la función “maxItAux” compara el valor máximo encontrado hasta el momento con el primer elemento de la lista y recorre el resto de la lista de manera recursiva. Como “maxItAux” mantiene el valor máximo acumulado de forma eficiente, garantiza que el resultado final es el valor máximo de la lista.

3. Casos de prueba

1. Función “maxLin”

- Prueba 1: Lista con todos los elementos iguales.

```
test("MaximoLista = 5") {
    assert(objMaximoList.maxLin(List(5,5,5,5, 5)) == 5)
}
```

- Prueba 2: Lista con el máximo al final.

```
test("MaximoLista = 10") {
    assert(objMaximoList.maxLin(List(1, 2, 3, 4, 5, 5,10,10)) == 10)
}
```

- Prueba 3: Lista con un solo elemento.

```
test("MaximoLista = 15") {
    assert(objMaximoList.maxLin(List(15)) == 15)
}
```

- Prueba 4: Lista vacía.

```
test("Maximo Lista Vacia") {
    assertThrows[IllegalArgumentException] {
        objMaximoList.maxLin(List())
    }
}
```

2. Función “maxIt”

- Prueba 1: Lista con elementos ascendentes.

```
test("MaximoLista RecursivaCola = 5") {
    assert(objMaximoList.maxIt(List(1,2,3,4,5)) == 5)
}
```

- Prueba 2: Lista con el máximo duplicado.

```
test("MaximoLista RecursivaCola = 10") {
    assert(objMaximoList.maxIt(List(1, 2, 3, 4, 5, 5,10,10)) == 10)
}
```

- Prueba 3: Lista con un solo elemento.

```
test("MaximoLista RecursivaCola = 12") {
    assert(objMaximoList.maxIt(List(12)) == 12)
}
```

- Prueba 4: Lista vacía.

```
test("Maximo Lista Vacia RecursivaCola") {
    assertThrows[IllegalArgumentException] {
        objMaximoList.maxIt(List())
    }
}
```

Conclusión

Las funciones “maxLin” y “maxIt” cumplen correctamente con sus propósitos según los resultados de las pruebas realizadas. La función “maxLin” sigue un enfoque recursivo tradicional, donde la pila de llamadas se construye hasta alcanzar el caso base. En

cambio, “maxIt” usa recursión de cola, lo que permite optimizar el uso de la pila y manejar listas más largas de manera más eficiente. Ambos algoritmos han sido verificados mediante pruebas de unidad, mostrando que el código es correcto bajo diferentes escenarios.

Informe de procesos Torres de Hanoi

En esta sección, describimos cómo se generan las pilas de llamadas en las funciones recursivas para los problemas de las Torres de Hanoi y cómo estas se van resolviendo a medida que se despliega el proceso.

1. **Función “movsTorresHanoi”: proceso recursivo**

La función “movsTorresHanoi(n: Int)” calcula el número mínimo de movimientos para resolver el problema de las Torres de Hanoi con “n” discos, utilizando la fórmula \(2^n - 1\). Esta función no realiza una pila de llamadas compleja, ya que simplemente aplica la fórmula para el cálculo, pero verificamos el caso base con “n <= 0”, lanzando una excepción en ese caso.

Función “torresHanoi”: proceso recursivo

La función “torresHanoi(n: Int, t1: Int, t2: Int, t3: Int)” genera una lista con los movimientos necesarios para resolver el problema de las Torres de Hanoi con “n” discos, utilizando tres torres, representadas por “t1”, “t2”, y “t3”.

Ejemplo: “torresHanoi(2, 1, 2, 3)”

Veamos cómo se genera y despliega la pila de llamadas en este caso:

1. Llamada inicial: torresHanoi(2, 1, 2, 3)

- Paso 1: torresHanoi(1, 1, 3, 2) → Mover un disco de la torre 1 a la 2 usando la 3 como intermediaria.

- Paso 2: Mover el disco más grande de la torre 1 a la torre 3.

- Paso 3: torresHanoi(1, 2, 1, 3) → Mover el disco restante de la torre 2 a la torre 3 usando la 1 como intermediaria.

2. Desglose de la pila:

- Primera llamada: “torresHanoi(1, 1, 3, 2)”

  - Paso 1: torresHanoi(0, 1, 2, 3) → Retorna List().

  - Paso 2: Mover el disco 1 de la torre 1 a la torre 2 → List((1, 2)).

  - Paso 3: torresHanoi(0, 3, 1, 2) → Retorna List().

- Resultado de la primera llamada: List((1, 2)).

- Segunda llamada: Mover el disco grande de la torre 1 a la torre 3 →List((1, 3)).

- Tercera llamada: torresHanoi(1, 2, 1, 3)

  - Paso 1: torresHanoi(0, 2, 3, 1) → Retorna List().

  - Paso 2: Mover el disco 1 de la torre 2 a la torre 3 → List((2, 3)).

  - Paso 3: torresHanoi(0, 1, 3, 2) → Retorna List().

- Resultado de la tercera llamada: List((2, 3)).

3. Resultado final: Al combinar todos los pasos, obtenemos:

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

2. Informe de corrección

Argumentación sobre la corrección

1. Función “movsTorresHanoi”:

Esta función aplica directamente la fórmula \(2^n - 1\) para calcular el número de movimientos necesarios para resolver el problema de las Torres de Hanoi con `n` discos. La corrección de la función está garantizada por la validez matemática de esta fórmula, que ha sido probada formalmente. Adicionalmente, la función contiene una condición para lanzar una excepción si “n” es menor o igual a 0, asegurando que no se calculen valores inválidos.

2. Función “torresHanoi”:

La corrección de esta función se puede demostrar mediante **inducción estructural** sobre el número de discos “n”.

- Caso base: Si “n == 0”, la función retorna una lista vacía, lo cual es correcto ya que no hay discos para mover.

- Paso inductivo: Supongamos que la función es correcta para un valor “n”. Para el caso “n+1”, la función sigue estos pasos:

1. Mueve “n” discos de la torre “t1” a la torre “t2” usando la torre “t3” como intermediaria.

2. Mueve el disco más grande directamente de la torre “t1” a la torre “t3”.

3. Mueve los “n” discos restantes de la torre “t2” a la torre “t3” usando la torre “t1” como intermediaria.

Dado que la función resuelve correctamente el problema para “n” discos (hipótesis inductiva) y los pasos descritos siguen la estrategia correcta para resolver el problema con “n+1” discos, la función es correcta por inducción.

Casos de prueba

1. Función “movsTorresHanoi”

- Prueba 1: Un solo disco.

```
test("Torres de Hanoi con 1 disco") {
    assert(objTorre.torresHanoi(1, 1, 2, 3) == List((1, 3)), "Error en test 1 disco")
}
```

- Prueba 2: Dos discos.

```
test("Torres de Hanoi con 2 discos") {
    assert(objTorre.torresHanoi(2, 1, 2, 3) == List((1, 2), (1, 3), (2, 3)), "Error en test 2 discos")
}
```

- Prueba 3: Tres discos.

```
test("Torres de Hanoi con 3 discos") {
    assert(objTorre.torresHanoi(3, 1, 2, 3) == List((1, 3), (1, 2), (3, 2), (1, 3), (2, 1),
    (2, 3), (1, 3)), "Error en test 3 discos")
}
```

- Prueba 4: Cero discos (debe lanzar excepción).

```
test("Torres de Hanoi sin discos") {
    assert(objTorre.torresHanoi(0, 1, 2, 3) == List(), "Error en test sin discos")
}
```

2. Función “torresHanoi”

- Prueba 1: Un disco.

```
test("Torres de Hanoi Movimientos con 1 disco") {
    assert(objTorre.movsTorresHanoi(1) == 1, "Error en test 1 disco")
}
```

- Prueba 2: Dos discos.

```
test("Torres de Hanoi Movimientos con 2 discos") {
    assert(objTorre.movsTorresHanoi(2) == 3, "Error en test 2 discos")
}
```

- Prueba 3: Tres discos.

```
test("Toores de Hanoi Movimientos con 3 discos") {
    assert(objTorre.movsTorresHanoi(3) == 7, "Error en test 3 discos")
}
```

- Prueba 4: Cero discos.

```
test("Torres de Hanoi Movimientos sin discos") {
    assert(objTorre.movsTorresHanoi(0) == 0, "Error en test sin discos")
}
```

Conclusión

El informe de procesos y la corrección muestran que las funciones “movsTorresHanoi” y “torresHanoi” son correctas y eficientes. Ambas funciones pasan las pruebas unitarias correspondientes, verificando su comportamiento con varios casos, incluidos los bordes (listas vacías, valores negativos, etc.). Esto asegura que el código es robusto y cumple con los requisitos del problema de las Torres de Hanoi.