Informe taller 1

Docente

DELGADO SAAVEDRA CARLOS ANDRES

Estudiantes

Juan David García Arroyave-2359450

Sebastián Zacipa Martínez-2359695

Juan José Hincapié Tascón-2359493

Universidad del valle-Sede Tuluá ´

2024

Informe de procesos

Análisis del Programa 1: maxList

Definición del problema

La función maxLisn se encarga de encontrar el valor máximo de una lista de enteros usando recursividad.

Código de la función maxLint:

```scala
package taller

import scala.annotation.tailrec


class maxlist {
  def maxLin(l:List[Int] ): Int = {
    if(l.isEmpty)
      throw new IllegalArgumentException("la lista esta vacia")
    else if (l.tail.isEmpty)
      l.head
    else {
      val maxRest = maxLin(l.tail)
      if(l.head > maxRest ) l.head else maxRest
    }
  }
```

Descripción del proceso

Si la lista está vacía, la función advierte que está vacía.

Para el caso que la lista sea vacía, se retorna el primer elemento de esta

En el caso general, la función compara el primer elemento de la lista con el valor máximo del resto de la lista

Ejemplo de ejecución

Consideremos la lista List(3, 7, 2, 9, 5).

1. **Llamada inicial**: maxLin(List(3, 7, 2, 9, 5))

   Compara 3 con maxLin(List(7, 2, 9, 5))

2. **2 llamada**: maxLin(List(7, 2, 9, 5))

   Compara 7 con maxLin(List(2, 9, 5))

3. **3 llamada**: maxLin(List(2, 9, 5))

    Compara 2 con maxLin(List(9, 5))

4. **Cuarta llamada**: maxLin(List(9, 5))

    Compara 9 con maxLin(List(5))

5. **Quinta llamada**: maxLin(List(5))

    Como la lista tiene un solo elemento, retorna 5.

Una vez que se llega a la lista de un solo elemento, la función comienza a retornar resultados:

maxLin(List(5)) retorna 5.

maxLin(List(9, 5)) compara 9 con 5 y retorna 9.

maxLin(List(2, 9, 5)) compara 2 con 9 y retorna 9.

maxLin(List(7, 2, 9, 5)) compara 7 con 9 y retorna 9.

maxLin(List(3, 7, 2, 9, 5)) compara 3 con 9 y retorna 9.

Análisis del Programa 2: maxIt (Recursividad de Cola)

La función maxIt también se encarga de encontrar el valor máximo de una lista de enteros

```scala

def maxIt(l:List[Int]): Int = {
  if (l.isEmpty)
    throw new IllegalArgumentException("La lista no puede estar vacía")
  else {
    @tailrec
    def maxItAux(l:List[Int], max:Int): Int = {
      if(l.isEmpty)
        max
      else {
        val newMax = if(l.head > max) l.head else max
        maxItAux(l.tail, newMax)
      }
    }
    maxItAux(l.tail, l.head)
  }
}
```

Descripción del proceso

1. Si la lista no contiene ningún elemento, el programa despliega una excepción
2. Si la lista no está vacía, utiliza una función auxiliar, maxItAux, que recibe la lista restante y el valor máximo acumulado hasta el momento.

3. En cada llamada, la función compara el primer elemento de la lista con el valor máximo acumulado y actualiza el máximo si es necesario.

Ejemplo de ejecución

Usamos la misma lista de ejemplo List(3, 7, 2, 9, 5).

1. **Llamada inicial**: maxIt(List(3, 7, 2, 9, 5))

   Llama a maxItAux(List(7, 2, 9, 5), 3).

2. **Primera llamada a maxItAux**: maxItAux(List(7, 2, 9, 5), 3)

   Compara 7 con 3, actualiza newMax = 7, y llama a maxItAux(List(2, 9, 5), 7).

3. **Segunda llamada a maxItAux**: maxItAux(List(2, 9, 5), 7)

   Compara 2 con 7, newMax sigue siendo 7, y llama a maxItAux(List(9, 5), 7).

4. **Tercera llamada a maxItAux**: maxItAux(List(9, 5), 7)

   Compara 9 con 7, actualiza newMax = 9, y llama a maxItAux(List(5), 9).

5. **Cuarta llamada a maxItAux**: maxItAux(List(5), 9)

   Compara 5 con 9, newMax sigue siendo 9, y llama a maxItAux(List(), 9).

6. **Quinta llamada a maxItAux**: maxItAux(List(), 9)

   Como la lista está vacía, retorna 9.

Informe de Procesos: Resolución del Problema de las Torres de Hanoi

Análisis del Programa 1: movsTorresHanoi

Definición del Problema

La función movsTorresHanaoi a través de recursión lineal obtiene el numero de movimientos necesarios para mover n cantidad de discos.

```
class Hanoi {
    def movsTorresHanoi(n : Int) : BigInt = {
        if(n <= 0)
            throw new IllegalArgumentException("el numero de discos debe de ser mayor a cero")
        else if (n == 1)
            1
        else 2 * movsTorresHanoi(n-1) + 1


    }
```

Descripción del código

Si la cantidad de discos es menor que 0, es inconsistente porque requiere al menos un disco para realizar un movimiento.

Si n es igual obtendremos que la cantidad de movimientos será 1. Actúa como caso base

Si n > 1 se hace un llamado recursivo con n-1, a su vez, duplicando el valor y sumándole uno

Ejemplo de Ejecución

Para 3 discos:

Llamada inicial: movsTorresHanoi(3):

    2 * movsTorresHanoi(2) + 1

Segunda llamada: movsTorresHanoi(2):

    2 * movsTorresHanoi(1) + 1

Tercera llamada: movsTorresHanoi(1):

    Obtenemos 1

    2 * movsTorresHanoi(1) + 1 → 2(1) + 1 = 3

    2 * movsTorresHanoi(2) +1 → 2(3) + 1 = 7

Análisis 2: torresHanoi

Definición del problema

La función torresHanoi resuelve el problema de mover n discos desde la torre t1 hacia la torre t3, utilizando la torre t2 como torre auxiliar.

```scala
def torresHanoi( n : Int , t1 : Int , t2 : Int , t3 : Int) : List[(Int , Int)] = {
    if(n <= 0)
        Nil
    else if(n == 1)
        List((t1,t3))
    else {
        val mov1 = torresHanoi(n-1, t1, t3, t2)
        val movn = List((t1,t3))
        val mov2 = torresHanoi(n-1,t2, t1, t3)
        mov1 ++ movn ++ mov2
    }
}
```

Descripción del Proceso

1. Si n es menor o igual a 0, la función retorna una lista vacía, ya que no hay discos que mover.
2. Si n es 1, se retorna una lista con un solo movimiento: mover el disco de la torre t1 a la torre t3.
3. Para n>1 , el proceso recursivo genera tres listas de movimientos:

    mov1: los movimientos para trasladar n-1 discos de t1 a t2 usando t3 como auxiliar.

    movn: el movimiento del disco más grande de t1 a t3.

    mov2: los movimientos para trasladar n-1 discos de t2 a t3 usando t1 como auxiliar.

Ejemplo de Ejecución

Para n = 3, t1 = 1, t2 = 2 y t3 = 3:

1. **Llamada inicial**: torresHanoi(3, 1, 2, 3)
    - Llama a torresHanoi(2, 1, 3, 2) para mover 2 discos de la torre 1 a la torre 2.
2. **Segunda llamada**: torresHanoi(2, 1, 3, 2)
    - Llama a torresHanoi(1, 1, 2, 3) para mover un disco de la torre 1 a la torre 3.
3. **Tercera llamada**: torresHanoi(1, 1, 2, 3)
    - Retorna List((1, 3)).
4. **Cuarta llamada**: Después de mover los 2 discos, llama a torresHanoi(1, 2, 1, 3) para mover el disco más grande de la torre 2 a la torre 3.

El resultado final es:

[(1, 3), (1, 2), (3, 2), (1, 3), (2, 1), (2, 3), (1, 3)]

Informe de Corrección

Sea f: List[N] → N la función que calcula el máximo de una lista de enteros no vacía.

Demostremos que ∀l List[N]: maxLin(l) = f(l)

Caso base: l = List(a1)

maxLin(List(a1)) → if List(a1).tail.isEmpty then List(a1).head else...→ a1

por otro lado, f(List(a1)) = a1. Entonces, maxLin(List(a1)) = f(List(a1))

Caso de inducción l = List(a1, a2, ..., ak+1)

Supongamos que maxLin(List(a1, a2, ..., ak)) = f(List(a1, a2, ..., ak)

Queremos demostrar que maxLin(List(a1, a2, ..., ak) = f(List(a1, a2, ..., ak + 1)

maxLin(List(a1, a2,...., ak+1)) → math.max(maxLin(List(a2, ..., ak +1)), a1)

Usando la hipótesis de inducción

math.max(maxLin(List(a2,...., ak+1)), a1) = math.max(f(List(a1,a2, ...ak)), ak + 1) = f(List(a1,a2, ..., ak+1))

Por lo tanto, maxLin(List(a1, a2,....., ak+1)) = f(List(a1, a2,.....ak+1)).

Concluimos por inducción que ∀l E List[N] : maxLin(l) = f(l)

Análisis del Programa 2: maxIt

Sea f: List[N] → N

Este programa implementa el siguiente proceso iterativo:

Un estado s = (max, l) donde l = List(ai, ai+1, ..., ak) es una cola de L

El estado inicial s0 = (L.head, L.tail) = (a1, List(a2,....,ak))

El estado s = (max, l) es final si l es vacía

La invariante de ciclo es Inv(max, l) ≡ l = List(ai, ai+1,...., ak) ∧ max = f(List(a1, a2, ..., ai -1)).

La transformación de estado es transformar (max, l) = (nmax, l.tail) donde nmax = math.nax(max, l.head)

Demostración de los puntos

1. Inv(s0) El estado inicial cumple la condición invariante.
   s0 = (a1, List(a2,..., ak)) → a1 = f(List(a1)).

2. (Si ≠ sf ∧ Inv(si)) → Inv(transformar(si)):
   Si Inv(si) es verdadera y si ≠ , entonces Inv(transformar(si)) es verdadera.
   (max, l) → (nmax, l.tail), donde nmax = math.max(max, l.head) y nmax sigue siendo el máximo de List(a1,...., ai).
3. Inv(sf) → respuestas(sf) = f(a):
   En el estado final (max, List()), max = f(List(a1,..., ak)
4. En cada paso, la lista l se reduce, acercándose a ser vacía. Después de k iteraciones, l = List().

   Esto implica que maxIt(l) = maxIt(iter(L.head, L.tail)) = f(L)
   Concluimos que maxIt(l) = f(l)

Análisis 2: torresHanoi

La función f(n) se define como:

f(1) = 1 (un solo disco requiere un movimiento)

Para n > 1, f(n) = 2 * f(n-1) + 1

Demostrar que:

∀n E N^+ : Pf(n) == f(n)

1. Caso base n = 1

   Para n = 1, la función devuelve q, que es el número mínimo de movimientos necesarios para trasladar un disco. Así mismo, f(1) = 1 por definición obtenemos, Pf(1) == f(1)

2. n = k + 1 con k ≥ 1
   supongamos que Pf(k) == f(k). Demostrar que f(k+1) == Pf(k+1)

   Según la función recursiva, tenemos Pf(k +1) = 2 * Pf(k) +1
   Sustituimos Pf(k) por f(k), lo que nos da:
   Pf(k+1) = 2 * f(k) +1 = f(k +1)

   Se puede concluir que Pf(k +1) == f(k +1)
   Pf(n) == f(n) para todo n E N^+

Análisis del Programa 2: Secuencia de movimientos

Caso base:
Para n=1, la función devuelve una lista con un único movimiento, que es mover el disco de la torre t1 a la torre t3 . Este es el resultado esperado, ya que solo hay un disco que mover.

Paso inductivo:
Asumimos que la función es correcta para n=kn = kn=k, es decir, genera la secuencia correcta de movimientos para mover k discos de t1 a t3 utilizando t2como torre auxiliar. Ahora probamos que la función es correcta para n=k+1.

Para n=k+1, la función divide el problema en tres partes:

1. Mueve k discos de t1 a t2 utilizando t3 como auxiliar. Esto está cubierto por la hipótesis inductiva.
2. Mueve el disco más grande de t1 a t3. Este movimiento es correcto.
3. Mueve k discos de t2 a t3 utilizando t1 como auxiliar. Esto también está cubierto por la hipótesis inductiva.

Dado que cada parte del proceso es correcta por la hipótesis inductiva, la función es correcta para n=k+1. Por inducción, la función es correcta para todos los n≥1.

Conclusiones

El código ofrece soluciones recursivas para obtener la cantidad de movimientos y secuencia de movimientos para las torres, lo cual es una solución útil y eficiente para resolución de este problema, a través de este algoritmo sería posible responder a la pregunta ¿se podría calcular cuándo será el fin del mundo?

El código implementa de manera efectiva dos funciones de manera recursiva para calcular el valor máximo de una lista.