1 Task I.1 — Hamming Numbers

1.1 Solutions

For this problem I developed 2 solutions, one using a naive approach checking which numbers are hamming numbers and the other generating the list of hamming numbers based upon previous values in the list.

1.1.1 Naive Approach

The Naive approach is defined in the following way:

```
import Data.Numbers.Primes
hamming' :: Integral a => [a]
hamming' = filter (\n -> all (`elem` [2,3,5]) $ primeFactors n) [1..]
```

The type of hamming' is a list elements of the type class Integral. Strictly it is of type Integer but the use of type classes allows for easy readability and expandability. hamming' is an infinite list of Integrals.

hamming' generates an infinite list of all integers from 1 onwards and then filters the list based upon a wether the number is a hamming number. This filter is done by checking each number in the list against a condition which expresses if a number is a hamming number. This condtion is expressed in the lambda function.

The test for hamming numbers exploits the fact that hamming numbers are created by multiplying another hamming number by 2, 3 or 5. This means that any hamming number can be represented as $2^i 3^j 5^k \; where \; x, y, z \in Z$. As 2, 3 and 5 are also prime numbers it means that this representation this is also the prime factorisation of the hamming number. Therefore any hamming number will have only 2, 3 and 5 as prime factors.

To exploit this the condition to filter on finds the prime factors of the number (represented as the list of prime factors) and checks that the only elements in the prime factor list are either 2, 3 or 5. If there is any prime factors that aren't 2, 3 or 5 then the number isn't added to the list and the next value is tested.

The problem with this approach is that it tests all the integers systematically meaning that there is a lot of numbers to consider. primeFactors also takes a long time to run as it has to systematically find the prime factors. This gets computational expensive for large numbers.

1.1.2 Main Approach

The main approach is defined in the following way:

```
hamming :: Integral a => [a]
hamming = 1 : (merge hamming5 $ merge hamming2 hamming3)
          where hamming2 = map (2*) hamming
                hamming3 = map (3*) hamming
                hamming5 = map (5*) hamming
```

The type of hamming is the same as hamming' as they will both produce the same infinite list. This being a list of Integrals.

hamming works by adding a modified version of hamming to the end of itself. 3 different modifications to the list are made, multiplying the list by 2, 3 and 5 represented by hamming2, hamming3 and hamming5. To achieve this we map the multiplication of the respective number across the list.

This will create 3 infinite lists that need to be combined into 1 list. To do so the merge function is used to join 2 such lists together. To merge all 3 lists all that is needed to be done is 2 lists are merged together and the result of that is merged with the final list. As the merge function removes duplicates we don't need to worry about duplicates in the list.

1.2 Cyclic Graphs

2 Task I.2 — SpreadSheet Evaluator

2.1 Solution

The following lines were added to evalCell to extend it to allow evaluation of sum and average expressions:

```
evalCell :: Sheet Double -> Exp -> Double
evalCell s (Sum r1 r2) = sum [ evalCell s (Ref r) | r <- range (r1,r2) ]
evalCell s (Avg r1 r2) = evalCell s (Sum r1 r2)
                         / fromIntegral(length $ range (r1,r2))
```

To evaluate the sum between 2 cells a list of the cell values is created and then sum is used to add them all together. To get all of the cell values to sum together, range is used to get a list of the cell references needed to be added together. For each cell in that range evalCell is called upon it to get the cell value.

Average works in a similar way using the sum function previously defined to get the sum between the 2 cell references. This value is then divided by the number of values added together, which can be found out by getting the length of the range between the 2 cells. fromIntegral is used as / needs both values to be a Double but length will return an Integer.

2.2 Weakness in the Evaluator

The weakness in this evaluator is that if there is any cells that reference each other otherwise known as a circular reference. If this occurs then the evaluator will hang as it tries to calculate an infinite loop. One way to fix this would be to check for circular references before evaluating a sheet and if there is a circular reference throw an error. This can be done by representing the sheet as a graph with each cell being a node and dependencies on other cells being edges. Once this graph is generated you can perform cycle detection upon it to find circular references.

Another approach that Microsoft Excel can use to compute circular references is to iterate over the sheet a number of times. This can be necessary for some iterative functions to run. You can calculate the value of the cell depending on the previous values and iterate up until a fixed number of iterations or until the values in the cells don't change.

3 Task I.3 — Skew Binary Random Access Trees

3.1 Solution

The following lines were added to implment drop for Skew Binary Random Access Trees:

```
drop :: Int -> RList a -> RList a
drop i ((w, t) : wts) | i < w     = dropTree i w t ++ wts
                      | otherwise = drop (i - w) wts

dropTree :: Int -> Int -> Tree a -> RList a
dropTree i _ (Leaf x) | i /= 0    = []
                      | otherwise = [(1, Leaf x)]
dropTree i w (Node t1 x t2) | i == 0    = [(w, (Node t1 x t2))]
                            | i <= w'   = dropTree (i - 1) w' t1 ++ [(w', t2)]
                            | otherwise = dropTree (i - w' - 1) w' t2
                              where
                                  w' = w `div` 2
```

drop compares the fi

3.2 Time Complexity

The solution has the desired time complexity of $O(log\ n)$ as

4 Task I.4 — Interval Arithmetic