Triangle Problem

Ryan Orendorff

The triangle problem is as follows.

Given a triangle t, find a maximum path through t.

In other words, find the path, starting from the tip of the triangle t down to the bottom, where the sum of the nodes along the way is the maximum of any possible path taken. This is a tad confusing, so let's consider calculate a discrete example. Let's look at the following triangle.

```haskell
type Triangle = [[Int]]

readTriangle :: String → Triangle
readTriangle = map (map read . words) . lines

— The string matches the file format the triangles come in.
t :: Triangle
t = readTriangle "5\n\
                \ 9 6\n\
                \ 4 6 8\n\
                \ 0 7 1 5"
{- t =    5
         9 6
        4 6 8
       0 7 1 5
-}
```

When we traverse this triangle, we are only allowed to move down. This means that we can take the following path with a sum of 25.

```
   5
  /
  9 6
 /
 4 6 8
```

```
  \
0 7 1 5
```

but not the following.

```
   5
  /
  9 6
 /
 4-6 8
    \
0 7 1 5
```

For this triangle, the path can be easily done by hand, resulting in the following path with a total sum of 27.

```
   5
  /
  9 6
   \
 4 6 8
  /
0 7 1 5
```

How to solve the triangle problem

Warning: if you don't want to know a way to solve this yet, stop reading!

If we attempt to brute force the solution by starting at the tip of the triangle and enumerating every possible path, we find that there are quite a few to check. Below is the triangle converted into a binary tree representing all of the paths that can be taken.

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
type TriangleTree = Tree Int

— Modified show from
— http://yannesposito.com/Scratch/en/blog/Haskell-the-Hard-Way/#trees
— to match the symmetric balanced binary trees that we are using.
instance (Show a) ⇒ Show (Tree a) where
  show tree = ”< ” ++ replace '\n' ”\n: ” (treeshow ”” tree)
    where
    treeshow _ (Leaf l) = show l
    treeshow pref (Node x left right) =
                  pshow pref x ++ "\n" ++
                  showSon pref "|—" "|  " left ++ "\n" ++
                  showSon pref "`—" "   " right

    — shows a tree using some prefixes to make it nice
    showSon pref before next subtree =
                  pref ++ before ++ treeshow (pref ++ next) subtree

    — pshow replaces "\n" by "\n"++pref
    pshow pref x = replace '\n' ('\n':pref) (show x)

    — replaces one char by another string
    replace c new =
      concatMap (change c new)
      where
          change c' new' x
              | x == c' = new'
              | otherwise = [x] — "x"


treeFromTriangle :: Triangle -> TriangleTree
treeFromTriangle = head . treeFromTriangle'
    where
        treeFromTriangle' [l] = map Leaf l
        treeFromTriangle' (roots:trees) = zipWith constructTree roots pairs
            where
                constructTree root (ll, lr) = Node root ll lr
                subtrees = treeFromTriangle' trees
                pairs = zip subtrees (tail subtrees)
```

```
treeFromTriangle t
{-
< 5
: |—9
: |  |—4
: |  |  |—0
: |  |  `—7
: |  `—6
: |     |—7
: |     `—1
: `—6
:    |—6
:    |  |—7
:    |  `—1
:    `—8
:       |—1
:       `—5
-}
```

The total number of paths in the above tree is equal to the number of leaves in the tree, which is 8. In fact, for any triangle, this search method will take $\Omega\left(2^{b-1}\right)$, where b is the height of the triangle. This bound is clearly too large in order to perform a brute force of triangles of any moderate size. However, for grins let's define a function to perform this brute search.

```haskell
-- Brute searches through a triangle for the maximum path
brute :: TriangleTree -> Int
brute (Leaf l) = l
brute (Node root nl nr) = root + max (brute nl) (brute nr)
```

Running this function does indeed return the maximum path is 27, as we had calculated by hand.

Since the brute force method is quite inefficient, let's instead try to think about the simplest cases. The first is a triangle of height 1, where the maximum path is just the value of the top of the triangle.

That was kinda boring. Let's look at triangles of height 2. Here is a random triangle.

```
 5

7 3
```

It is pretty clear how to find the maximum path in this triangle: simply figure out which of the left or right side is larger, and return that value added to the root value. In this case, the maximum path is on the left, so our final maximum path sum is 12.

```
 5
/
7 3
```

Here is another random simple triangle, where we know the maximum path is 11.

```
 3
  \
3 8
```

Now onto a slightly larger example.

```
  9

 5 3

7 3 8
```

Hmm, some parts of this triangle look familiar. If we break down the triangle into subtriangles.

```
      9
   /     \
  .       .
 /5\     /3\
/7 3\   /3 8\
-----   -----
```

We already know the maximum paths of these subtriangles, so we don't need to recompute them. We can just replace the triangles with their maximum path values.

```
  9

12 11
```

The result is a simple triangle, of which we can calculate the maximum path (21).

```
  9
 /
12 11
```

Now if we trace the result out by hand, we should be able to check that 21 is the maximum path.

```
  9
 /
 5 3
/
7 3 8
```

If we recursively apply the prior procedure, starting from the bottom of the triangle and working upwards, we get an algorithm that works a triangle of any height.

```
maximumPath :: Triangle -> Int
maximumPath = maximumPath' . reverse
    where
        maximumPath' [[root]] = root
        maximumPath' (base:subbase:rest) = maximumPath' $ newbase:rest
            where
                maxheight2 root (l, r) = root + max l r

                leaves = zip base (tail base)
                newbase = zipWith maxheight2 subbase leaves
```

The basic idea behind this algorithm is this; given a triangle t, we generate a new triangle t' with the following properties:

- The height of t' is one less than t,
- The maximum path through t' and the maximum path through t (minus the last link, which can't exist because the height is one less) are the same, and
- the maximum path value is the same for t and t'.

Since we generate a triangle that is one smaller but has the same maximum path, we can keep reducing the triangle size until it becomes easy to read off the answer. The base case here is a triangle with a height of one, in which case the maximum path is just the node.

So what is the running time of this algorithm? Well the running time to solve each height 2 triangle is $O(1)$, and the number of subtriangles is $(b(b-1))/2$, for a total running time of $O(b^2)$.

This running time is definitely better than the brute force method. But is it the best we can do? Well the total number of points in the triangle is $(b(b+1))/2$, so you can't really do better than to touch every point a constant number of times (in this case at most twice).

Extensions

There is also an interesting point to note here. This problem is equally valid for any monoid that is also in the ordered class. Therefore the maximumPath algorithm could easily work with say a maximum product by using the mappend (`<>`) operator instead of (`+`).

```
import Data.Monoid (Monoid, (<>), Product(Product), Sum(Sum))

```

```haskell
maximumPathMonoid :: (Ord m, Monoid m) => [[m]] -> m
maximumPathMonoid = maximumPath' . reverse
    where
        maximumPath' [[root]] = root
        maximumPath' (base:subbase:rest) = maximumPath' $ newbase:rest
            where
                -- (+) to (<>) is the only change.
                maxheight2 root (l, r) = root <> max l r

                leaves = zip base (tail base)
                newbase = zipWith maxheight2 subbase leaves

-- Use Integer since products quickly grow in size.
productT :: Triangle -> [[Product Integer]]
productT = map (map $ Product . toInteger)

sumT :: Triangle -> [[Sum Int]]
sumT = map (map Sum)
```

```haskell
maximumPathMonoid $ sumT t     -- Sum {getSum = 27}
maximumPathMonoid $ productT t -- Product {getProduct = 1890}
```

(There is one algebraic flaw in this argument, do you know what it is?)

Also of interest is finding the route associated with this maximum path. The simplest way to do this is to perform the calculation inside the writer monad, leaving a log that is a list of the actions taken thus far when choosing a path. This can also be done manually by modifying the `maximumPath'` function to pass a pair around where one of the elements is a log.

Another extension is that the algorithm above can be used to find the minimum path by changing only *two* characters! In a similar manner, the function could take in an input that would take either the maximum or minimum path, making the path function more generic.