Twitter waterflow problem and loeb

By Chris Done

The Waterflow Problem

I recently saw *I Failed a Twitter Interview* which features the following cute problem. Consider the following picture:

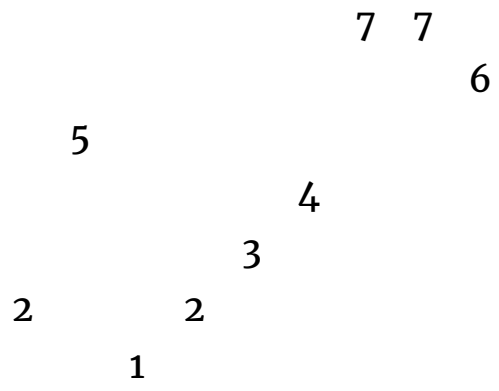
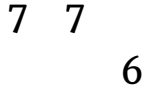


Fig. 1

In *Fig. 1*, we have walls of different heights. Such pictures are represented by an array of integers, where the value at each index is the height of the wall. *Fig. 1* is represented with an array as `[2,5,1,2,3,4,7,7,6]`.

Now imagine it rains. How much water is going to be accumulated in puddles between walls? For example, if it rains in *Fig 1*, the following puddle will be formed:



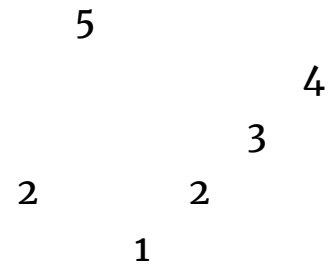


Fig. 2

No puddles are formed at edges of the wall, water is considered to simply run off the edge.

We count volume in square blocks of 1×1. Thus, we are left with a puddle between column 1 and column 6 and the volume is 10.

Write a program to return the volume for any array.

My Reaction

I thought, this looks like a spreadsheet problem, and closed the page, to get on with my work. Last thing I need right now is nerd sniping.

A week or so later I saw *A functional solution to Twitter's waterflow problem* which presented a rather concise and beautiful approach to solving the problem. I present it here, in the style that I prefer:

```
water :: [Int] -> Int
water h =
  sum (zipWith (-)
               (zipWith min
                        (scanl1 max h)
                        (scanr1 max h))
               h)
```

This is the second fastest algorithm in this page, clocking in at a mean

2.624748 ms for a random list of 10000 elements. See Benchmarks for more details.

An efficient algorithm can be achieved with a trivial rewrite of Michael Kozakov's Java solution is:

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE ViewPatterns #-}

import qualified Data.Vector as V
import Data.Vector ((!),Vector)

water :: Vector Int -> Int
water land = go 0 0 (V.length land - 1) 0 0 where
  go !volume !left !right
     (extend left -> leftMax)
     (extend right -> rightMax)
    | left < right =
      if leftMax >= rightMax
         then go (volume + rightMax - land!right)
                 left (right - 1) leftMax rightMax
         else go (volume + leftMax - land!left)
                 (left +  1) right leftMax rightMax
    | otherwise = volume
  extend i d = if land!i > d then land!i else d
```

This is expectedly the fastest algorithm in this page, clocking in at a mean of 128.2953 us for a random vector of 10000 elements.

But I still thought my spreadsheet idea was feasible.

My approach

In a similar way to Philip Nilsson, I can define the problem as it comes intuitively to me. As I saw it in my head, the problem can be broken down into "what is the volume that a given column will hold?" That can

be written like this:

$$volume_0 = 0$$

$$volume_{|S|-1} = 0$$

$$volume_i = \min(left_{i-1}, right_{i+1}) - height_i$$

Where *left* and *right* are the peak heights to the left or right:

$$left_0 = height_0$$

$$left_i = \max(height_i, left_{i-1})$$

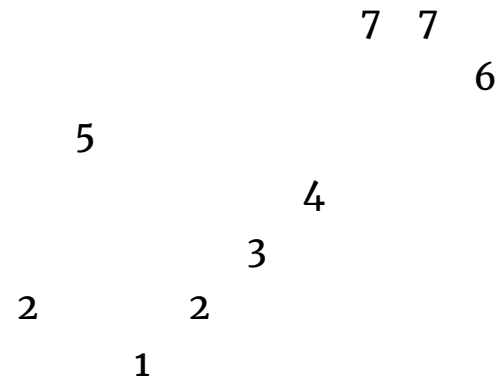
$$right_{|S|-1} = height_{|S|-1}$$

$$right_i = \max(height_i, right_{i+1})$$

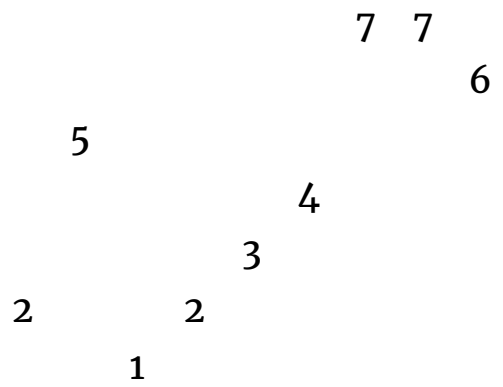
That's all.

A visual example

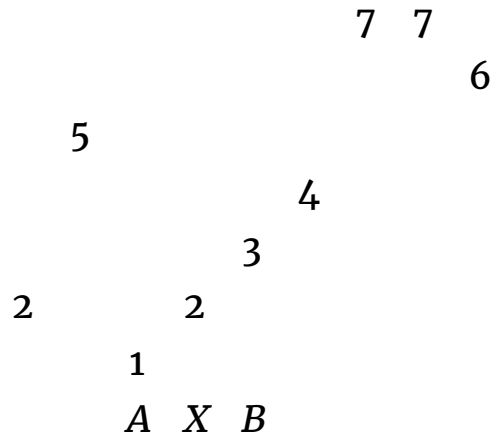
An example of `i` is:



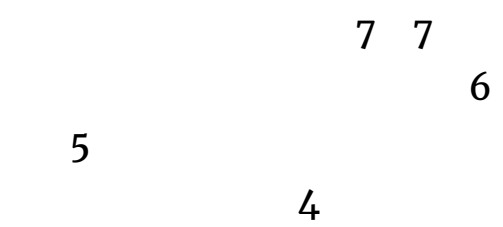
We spread out in both directions to find the “peak” of the columns:

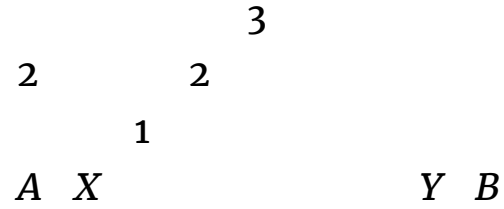


How do we do that? We simply define the volume of a column to be in terms of our immediate neighbors to the left and to the right:



X is defined in terms of A and B. A and B are, in turn, are defined in terms of their immediate neighbors. Until we reach the ends:



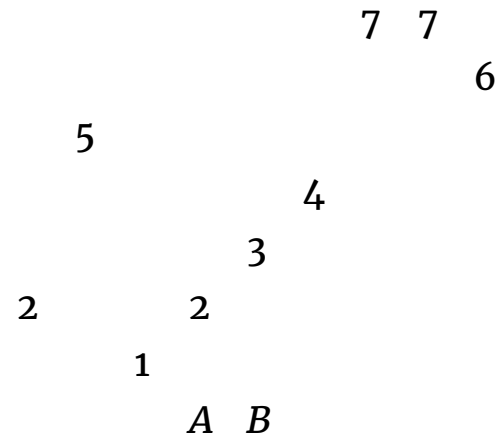


The ends of the wall are the only ones who only have *one side* defined in terms of their single neighbor, which makes complete sense. Their volume is always `0`. It's impossible to have a puddle on the edge. A's "right" will be defined in terms of X, and B's "left" will be defined in terms of Y.

But how does this approach avoid infinite cycles? Easy. Each column in the spreadsheet contains three values:

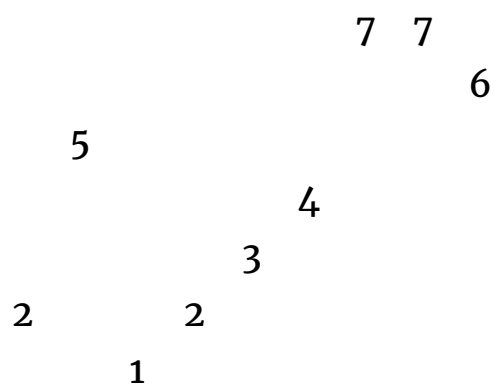
1. The peak to the left.
2. The peak to the right.
3. My volume.

A and B below depend upon eachother, but for different slots. A depends on the value of B's "right" peak value, and B depends on the value of A's "left" value:

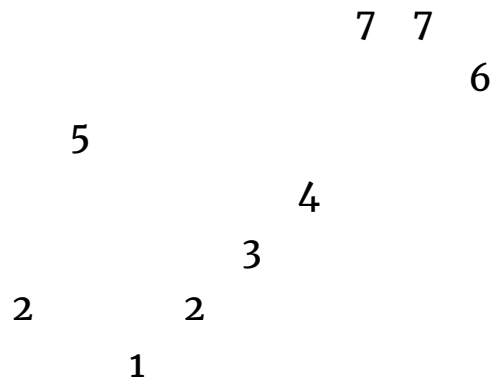


The *height* of the column's peak will be the smallest of the two peaks on

either side:



And then the *volume* of the column is simply the height of the peak minus the column's height:



Enter loeb

I first heard about `loeb` from Dan Piponi's From Löb's Theorem to Spreadsheet Evaluation some years back, and ever since I've been wanting to use it for a real problem. It lets you easily define a spreadsheet generator by mapping over a functor containing functions. To each function in the container, the container itself is passed to that function.

Here's `loeb`:

```haskell
loeb :: Functor f => f (f b -> b) -> f b
loeb x = fmap (\f -> f (loeb x)) x
```

(Note, there is a more efficient version here.)

So as described in the elaboration of how I saw the problem in my head, the solution takes the vector of numbers, generates a spreadsheet of triples, defined in terms of their neighbors—exept edges—and then simply makes a sum total of the third value, the volumes.

```haskell
import Control.Lens
import qualified Data.Vector as V
import Data.Vector ((!),Vector)

water :: Vector Int -> Int
water = V.sum . V.map (view _3) . loeb . V.imap cell where
  cell i x xs
    | i == 0                = edge _2
    | i == V.length xs - 1 = edge _1
    | otherwise             = col i x xs
    where edge ln = set l (view l (col i x xs)) (x,x,0)
            where l r = cloneLens ln r
  col i x xs = (l,r,min l r - x)
    where l = neighbor _1 (-)
          r = neighbor _2 (+)
          neighbor l o = max x (view l (xs ! (i `o` 1)))
```

It's not the most efficient algorithm—it relies on laziness in an almost perverse way, but I like that I was able to express exactly what occured to me. And `loeb` is suave. It clocks in at a mean of 3.512758 ms for a vector of 10000 random elements. That's not too bad, compared to the `scanr`/`scanl`.

This is was also my first use of lens, so that was fun. The `cloneLens` are required because you can't pass in an arbitrary lens and then use it both

as a setter and a getter, the type becomes fixed on one or the other, making it not really a *lens* anymore. I find that pretty disappointing. But otherwise the lenses made the code simpler.

Update with comonads & pointed lists

Michael Zuser pointed out another cool insight from *Comonads and reading from the future* (Dan Piponi's blog is a treasure trove!) that while `loeb` lets you look at the *whole* container, giving you *absolute* references, the equivalent corecursive fix (below `wfix`) on a comonad gives you *relative* references. Michael demonstrates below using Jeff Wheeler's pointed list library and Edward Kmett's comonad library:

```
import Control.Comonad
import Control.Lens
import Data.List.PointedList (PointedList)
import qualified Data.List.PointedList as PE
import Data.Maybe

instance Comonad PointedList where
    extend  = PE.contextMap
    extract = PE._focus

water :: [Int] -> Int
water = view _2 . wfix . fmap go . fromMaybe (PE.singleton 0) . PE.fromList
  where
    go height context = (lMax, total, rMax)
      where
        get f = maybe (height, 0, height) PE._focus $ f context
        (prevLMax,          _,         _) = get PE.previous
        (_        , prevTotal, prevRMax) = get PE.next
        lMax = max height prevLMax
        rMax = max height prevRMax
        total = prevTotal + min lMax rMax - height
```

I think if I'd've heard of this before, this solution would've come to mind

instead, it seems entirely natural!

Sadly, this is the slowest algorithm on the page. I'm not sure how to optimize it to be better.

Update on lens

Russell O'Connor gave me some hints for reducing the lens verbiage. First, eta-reducing the locally defined lens `l` in my code removes the need for the `NoMonomorphismRestriction` extension, so I've removed that. Second, a rank-N type can also be used, but then the type signature is rather large and I'm unable to reduce it presently without reading more of the lens library.

Update on loeb

On re-reading the comments for *From Löb's Theorem to Spreadsheet Evaluation* I noticed Edward Kmett pointed out we can get better laziness sharing with the following definition of loeb:

```
loeb :: Functor f => f (f b -> b) -> f b
loeb x = xs
  where xs = fmap ($ xs) x
```

For my particular `water` function, this doesn't make much of a difference, but there is a difference. See the next section.

Time travelling solution

Michael Zuser demonstrated a time travelling solution based on the Tardis monad:

```
import Control.Monad
```

```haskell
import Control.Monad.Tardis

water :: [Int] -> Int
water = flip evalTardis (minBound, minBound) . foldM go 0
  where
    go total height = do
        modifyForwards $ max height
        leftmax <- getPast
        rightmax <- getFuture
        modifyBackwards $ max height
        return $ total + min leftmax rightmax - height
```

Fastest

Sami Hangaslammi submitted this fast version (clocks in at 33.80864 us):

```haskell
import Data.Array

waterVolume :: Array Int Int -> Int
waterVolume arr = go 0 minB maxB where
  (minB,maxB) = bounds arr

  go !acc lpos rpos
    | lpos >= rpos             = acc
    | leftHeight < rightHeight =
      segment leftHeight    1  acc lpos contLeft
    | otherwise                =
      segment rightHeight (-1) acc rpos contRight
    where
        leftHeight          = arr ! lpos
        rightHeight         = arr ! rpos
        contLeft  acc' pos' = go acc' pos' rpos
        contRight acc' pos' = go acc' lpos pos'

  segment limit move !acc' !pos cont
    | delta <= 0 = cont acc' pos'
    | otherwise  = segment limit move (acc' + delta) pos' cont
```

```
    where
        delta = limit - arr ! pos'
        pos'  = pos + move
```

Changing the data structure to a vector brings this down to 26.79492 us.

Benchmarks

Here are benchmarks for all versions presented in this page.

```
benchmarking water/10000
mean: 2.624748 ms, lb 2.619731 ms, ub 2.630364 ms, ci 0.950
std dev: 85.45235 us, lb 78.33856 us, ub 103.2333 us, ci 0.950
found 54 outliers among 1000 samples (5.4%)
  51 (5.1%) high mild
  3 (0.3%) high severe
variance introduced by outliers: 79.838%
variance is severely inflated by outliers

benchmarking water_loeb/10000
mean: 3.512758 ms, lb 3.508533 ms, ub 3.517332 ms, ci 0.950
std dev: 70.77688 us, lb 65.88087 us, ub 76.97691 us, ci 0.950
found 38 outliers among 1000 samples (3.8%)
  24 (2.4%) high mild
  14 (1.4%) high severe
variance introduced by outliers: 59.949%
variance is severely inflated by outliers

benchmarking water_loeb'/10000
mean: 3.511872 ms, lb 3.507492 ms, ub 3.516778 ms, ci 0.950
std dev: 74.34813 us, lb 67.99334 us, ub 84.56183 us, ci 0.950
found 43 outliers among 1000 samples (4.3%)
  34 (3.4%) high mild
  9 (0.9%) high severe
variance introduced by outliers: 62.285%
variance is severely inflated by outliers

benchmarking water_onepass/10000
```

```
mean: 128.2953 us, lb 128.1194 us, ub 128.4774 us, ci 0.950
std dev: 2.864153 us, lb 2.713756 us, ub 3.043611 us, ci 0.950
found 18 outliers among 1000 samples (1.8%)
  17 (1.7%) high mild
  1 (0.1%) high severe
variance introduced by outliers: 64.826%
variance is severely inflated by outliers

benchmarking water_array/10000
mean: 33.80864 us, lb 33.76067 us, ub 33.86597 us, ci 0.950
std dev: 844.9932 ns, lb 731.3158 ns, ub 1.218807 us, ci 0.950
found 29 outliers among 1000 samples (2.9%)
  27 (2.7%) high mild
  2 (0.2%) high severe
variance introduced by outliers: 69.836%
variance is severely inflated by outliers

benchmarking water_vector/10000
mean: 26.79492 us, lb 26.75906 us, ub 26.83274 us, ci 0.950
std dev: 595.1865 ns, lb 559.8929 ns, ub 652.5076 ns, ci 0.950
found 21 outliers among 1000 samples (2.1%)
  18 (1.8%) high mild
  3 (0.3%) high severe
variance introduced by outliers: 64.525%
variance is severely inflated by outliers

benchmarking water_comonad/10000
collecting 1000 samples, 1 iterations each, in estimated 28315.44 s

benchmarking water_tardis/10000
collecting 1000 samples, 1 iterations each, in estimated 28.98788 s
```

I never bothered waiting around for the comonad or the time traveller ones to complete. They're slow, let's say that.

Here's the source code to the benchmarks, let me know if I made a mistake in the benchmark setup.

