

Если вы видите что-то необычное, просто сообщите мне.

# Profiling and Benchmarking

I've said it before, but I'll say it again. As much as we'd like to think it's the case, our Haskell code doesn't work just because it compiles. In part 1 of this testing series, we saw how to construct basic test suites to make sure our code functions properly. But even if it passes our test suites, this doesn't mean it works as well as it could either. Sometimes we'll realize that the code we wrote isn't quite performant enough, so we'll have to make improvements.

But improving our code can sometimes feel like taking shots in the dark. You'll spend a great deal of time tweaking a certain piece. Then you'll find you haven't actually made much of a dent in the total run time of the application. Certain operations generally take longer, like database calls, network operations, and IO. So you can often have a decent idea of where to start. But it always helps to be sure. This is where benchmarking and profiling come in. We're going to take a specific problem and learn how we can use some Haskell tools to zero in on the problem point. In part 3 of this series, we'll see how we can fix some of the problems that we identify with some advanced data structures!

As a note, the tools we'll use require you to be organizing your code using Stack or Cabal. If you've never used either of these before, you should check out our Stack Mini Course! It'll teach you the basics of creating a project with Stack. You'll also learn the primary commands to use with Stack. It's free, so check it out!

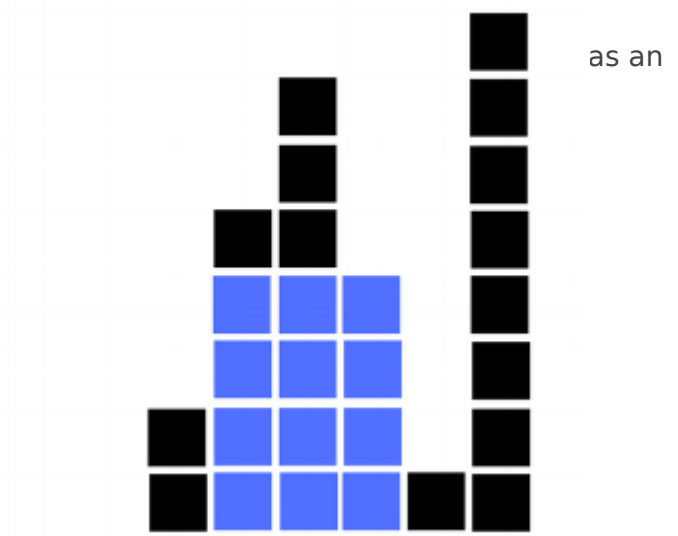
You can also follow along with this code by heading to the Github repository for this series! The bulk of the code for this part lives in the Fences module and the Benchmark file that we'll design.

## THE PROBLEM

Our overarching problem for this article will be the "largest rectangle" problem. You can actually try to solve this problem yourself on Hackerrank under the name "John and Fences". Imagine we have a series of vertical bars with varying heights placed next to each other. We want to find the

area of the largest rectangle that we can draw over these bars that doesn't include any empty space. Here's a visualization of one such problem and solution.

In this example, we have posts with heights [2,5,7,4] as an



area of 12, as we see with the highlighted squares.

# Fence Problem.png

This problem is pretty neat and clean to solve with Haskell, as it lends itself to a recursive solution. First let's define a couple newtypes to illustrate our concepts for this problem. We'll use a compiler extension to derive the Num typeclass on our index type, as this will be useful later.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

...

newtype FenceValues = FenceValues { unFenceValues :: [Int] }
newtype FenceIndex = FenceIndex { unFenceIndex :: Int }
    deriving (Eq, Num, Ord)
-- Left Index is inclusive, right index is non-inclusive
newtype FenceInterval = FenceInterval { unFenceInterval :: (FenceIndex, FenceIndex) }
newtype FenceSolution = FenceSolution { unFenceSolution :: Int }
    deriving (Eq, Show, Ord)
```

Next, we'll define our primary function. It will take our FenceValues, a list of integers, and return our solution.

```
largestRectangle :: FenceValues -> FenceSolution
largestRectangle values = ...
```

It in turn will call our recursive helper function. This function will calculate the largest rectangle over a specific interval. We can solve it recursively by using smaller and smaller intervals. We'll start by calling it on the interval of the whole list.

```
largestRectangle :: FenceValues -> FenceSolution
largestRectangle values = largestRectangleAtIndices values
  (FenceInterval (FenceIndex 0, FenceIndex (length (unFenceValues values))))

largestRectangleAtIndices :: FenceValues -> FenceInterval -> FenceSolution
largestRectangleAtIndices = ...
```

Now, to break this into recursive cases, we need some more information first. What we need is the index *i* of the minimum height in this interval. One option is that we could make a rectangle spanning the whole interval with this height.

Any other "largest rectangle" won't use this particular index. So we can then divide our problem into two more cases. In the first, we'll find the largest rectangle on the interval to the left. In the second, we'll look to the right.

As your might realize, these two cases simply involve making recursive calls! Then we can easily compare their results. The only thing we need to add is a base case. Here are all these cases represented in code:

```
largestRectangleAtIndices :: FenceValues -> FenceInterval -> FenceSolution
largestRectangleAtIndices
  values
  interval@(FenceInterval (leftIndex, rightIndex)) =
  -- Base Case: Checks if left + 1 >= right
  if isBaseInterval interval
  then FenceSolution (valueAtIndex values leftIndex)
  -- Compare three cases
  else max (max middleCase leftCase) rightCase
  where
  -- Find the minimum height and its index
  (minIndex, minValue) = minimumHeightIndexValue values interval
```

```

-- Case 1: Use the minimum index
middleCase = FenceSolution $ (intervalSize interval) * minValue
-- Recursive call #1
leftCase = largestRectangleAtIndices values (FenceInterval (leftIndex, minIndex))
-- Guard against case where there is no "right" interval
rightCase = if minIndex + 1 == rightIndex
  then FenceSolution (minBound :: Int) -- Supply a "fake" solution that we'll ignore
  -- Recursive call #2
  else largestRectangleAtIndices values (FenceInterval (minIndex + 1, rightIndex))

```

And just like that, we're actually almost finished. The only sticking point here is a few helper functions. Three of these are simple:

```

valueAtIndex :: FenceValues -> FenceIndex -> Int
valueAtIndex values index = (unFenceValues values) !! (unFenceIndex index)

isBaseInterval :: FenceInterval -> Bool
isBaseInterval (FenceInterval (FenceIndex left, FenceIndex right)) = left + 1 >= right

intervalSize :: FenceInterval -> Int
intervalSize (FenceInterval (FenceIndex left, FenceIndex right)) = right - left

```

Now we have to determine the minimum on this interval. Let's do this in the most naive way, by scanning the whole interval with a fold.

```

minimumHeightIndexValue :: FenceValues -> FenceInterval -> (FenceIndex, Int)
minimumHeightIndexValue values (FenceInterval (FenceIndex left, FenceIndex right)) =
  foldl minTuple (FenceIndex (-1), maxBound :: Int) valsInInterval
  where
    valsInInterval :: [(FenceIndex, Int)]
    valsInInterval = drop left (take right (zip (FenceIndex <$> [0..]) (unFenceValues values)))
    minTuple :: (FenceIndex, Int) -> (FenceIndex, Int) -> (FenceIndex, Int)
    minTuple old@(_, heightOld) new@(_, heightNew) =
      if heightNew < heightOld then new else old

```

And now we're done! As an exercise you can head to this unit test module and write some HUnit tests for this function. Write a few basic tests at first, and then incorporate a test case for the `input10000` and `output10000` expressions in the file. Run the tests with this command:

# BENCHMARKING OUR CODE

Now, this is a neat little algorithmic solution, but we want to know if our code is efficient. We need to know if it will scale to larger input values. If you incorporated the size-10000 example into your unit tests, you may have found that the test suite is suddenly quite a bit slower.

We can find the answer to these performance questions by writing benchmarks. Benchmarks are a feature we can use in conjunction with Cabal and Stack. They work a lot like test suites. But instead of proving the correctness of our code, they'll show us how fast our code runs under various circumstances. We'll use the Criterion library to do this. We'll start by adding a section in our .cabal file for this benchmark:

```
benchmark fences-benchmark
  type:          exitcode-stdio-1.0
  hs-source-dirs: benchmark
  main-is:       FencesBenchmark.hs
  build-depends: base
                , Testing
                , criterion
                , random
  default-language: Haskell2010
```

Now we'll look at our FencesBenchmark file, make it a Main module and add a main function. We'll start by generating 6 lists, increasing in size by a factor of 10 each time.

```
module Main where

import Criterion
import Criterion.Main (defaultMain)
import System.Random

import Fences

main :: IO ()
```

```

main = do
  [l1, l2, l3, l4, l5, l6] <- mapM
    randomList [1, 10, 100, 1000, 10000, 100000]
  ...

-- Generate a list of a particular size
randomList :: Int -> IO FenceValues
randomList n = FenceValues <$> (sequence $ replicate n (randomRIO (1, 10000 :: Int)))

```

Now the syntax for the Criterion library is a lot like HUnit in many respects. It has a defaultMain function. The Benchmark type is a lot like the TestTree type. We can create a single Benchmark using the bench expression, and combine a group of them with bGroup:

```

main :: IO ()
main = do
  [l1, l2, l3, l4, l5, l6] <- mapM
    randomList [1, 10, 100, 1000, 10000, 100000]
  defaultMain
    [ bgroup "fences tests"
      [ bench "Size 1 Test" $ ...
        , bench "Size 10 Test" $ ...
      ]
    ]

```

The difference is that instead of filling in each case with a test predicate assertion, we can fill it in with a Benchmarkable element. We create these by taking a code expression we want to benchmark (like a call to largestRectangle) and passing it to the whnf function.

```

main :: IO ()
main = do
  [l1, l2, l3, l4, l5, l6] <- mapM
    randomList [1, 10, 100, 1000, 10000, 100000]
  defaultMain
    [ bgroup "fences tests"
      [ bench "Size 1 Test" $ whnf largestRectangle l1
        , bench "Size 10 Test" $ whnf largestRectangle l2
        , bench "Size 100 Test" $ whnf largestRectangle l3
        , bench "Size 1000 Test" $ whnf largestRectangle l4
        , bench "Size 10000 Test" $ whnf largestRectangle l5
      ]
    ]

```

```
, bench "Size 100000 Test" $ whnf largestRectangle l6  
]  
]
```

That's all there is to it really! We're ready to run our benchmark now. We'd normally run all our benchmarks with `stack bench` (or `cabal bench` if you're not using Stack). And you can run an individual benchmark set similar to an individual test set:

```
>> stack build Testing:bench:fences-benchmark
```

But we can also compile our code with the `--profile` flag. This will automatically create a profiling report with more information about our code. Note using profiling requires re-compiling ALL the dependencies to use profiling as well. So you don't want to switch back and forth a lot.

```
>> stack build Testing:bench:fences-benchmark --profile  
Benchmark fences-benchmark: RUNNING...  
benchmarking fences tests/Size 1 Test  
time          47.79 ns (47.48 ns .. 48.10 ns)  
              1.000 R² (0.999 R² .. 1.000 R²)  
mean          47.78 ns (47.48 ns .. 48.24 ns)  
std dev       1.163 ns (817.2 ps .. 1.841 ns)  
variance introduced by outliers: 37% (moderately inflated)  
  
benchmarking fences tests/Size 10 Test  
time          3.324 µs (3.297 µs .. 3.356 µs)  
              0.999 R² (0.999 R² .. 1.000 R²)  
mean          3.340 µs (3.312 µs .. 3.368 µs)  
std dev       98.52 ns (79.65 ns .. 127.2 ns)  
variance introduced by outliers: 38% (moderately inflated)  
  
benchmarking fences tests/Size 100 Test  
time          107.3 µs (106.3 µs .. 108.2 µs)  
              0.999 R² (0.999 R² .. 0.999 R²)  
mean          107.2 µs (106.3 µs .. 108.4 µs)  
std dev       3.379 µs (2.692 µs .. 4.667 µs)  
variance introduced by outliers: 30% (moderately inflated)  
  
benchmarking fences tests/Size 1000 Test  
time          8.724 ms (8.596 ms .. 8.865 ms)
```

```

0.998 R² (0.997 R² .. 0.999 R²)
mean      8.638 ms (8.560 ms .. 8.723 ms)
std dev   228.8 µs (193.6 µs .. 272.8 µs)

benchmarking fences tests/Size 10000 Test
time      909.2 ms (899.3 ms .. 914.1 ms)
          1.000 R² (1.000 R² .. 1.000 R²)
mean      915.1 ms (914.6 ms .. 915.8 ms)
std dev   620.1 µs (136.0 as .. 664.8 µs)
variance introduced by outliers: 19% (moderately inflated)

benchmarking fences tests/Size 100000 Test
time      103.9 s (91.11 s .. 117.3 s)
          0.997 R² (0.997 R² .. 1.000 R²)
mean      107.3 s (103.7 s .. 109.4 s)
std dev   3.258 s (0.0 s .. 3.702 s)
variance introduced by outliers: 19% (moderately inflated)

Benchmark fences-benchmark: FINISH

```

So when we run this, we'll find something...troubling. It takes a looong time to run the final benchmark on size 100000. On average, this case takes over 100 seconds...more than a minute and a half! We can further take note of how the average run time increases based on the size of the case. Let's pare down the data a little bit:

```

Size 1: 47.78 ns
Size 10: 3.340 µs (increased ~70x)
Size 100: 107.2 µs (increased ~32x)
Size 1000: 8.638 ms (increased ~81x)
Size 10000: 915.1 ms (increased ~106x)
Size 100000: 107.3 s (increased ~117x)

```

Each time we increase the size of the problem by a factor of 10, the time spent increased by a factor closer to 100! This suggests our run time is  $O(n^2)$  (check out this guide if you are unfamiliar with Big-O notation). We'd like to do better.

# DETERMINING THE PROBLEM



So we want to figure out why our code isn't performing very well. Luckily, we already profiled our benchmark!. This outputs a specific file that we can look at, called fences-benchmark.prof. It has some very interesting results:

COST CENTRE	MODULE SRC	%time	%alloc
minimumHeightIndexValue.valsInInterval	Lib src/Lib.hs:45:5-95	69.8	99.7
valueAtIndex	Lib src/Lib.hs:51:1-74	29.3	0.0

We see that we have two big culprits taking a lot of time. First, there is our function that determines the minimum between a specific interval. The report is even more specific, calling out the specific offending part of the function. We spend a lot of time getting the different values for a specific interval. In second place, we have valueAtIndex. This means we also spend a lot of time getting values out of our list.

First let's be glad we've factored our code well. If we had written our entire solution in one big function, we wouldn't have any leads here. This makes it much easier for us to analyze the problem. When examining the code, we see why both of these functions could produce  $O(n^2)$  behavior.

Due to the number of recursive calls we make, we'll call each of these functions  $O(n)$  times. Then when we call valueAtIndex, we use the (!!) operator on our linked list. This takes  $O(n)$  time. Scanning the whole interval for the minimum height has the same effect. In the worst case, we have to look at every element in the list! I'm hand waving a bit here, but that is the basic result. When we call these  $O(n)$  pieces  $O(n)$  times, we get  $O(n^2)$  time total.

## CLIFF HANGER ENDING

We can actually solve this problem in  $O(n \log n)$  time, a dramatic improvement over the current  $O(n^2)$ . But we'll have to improve our data structures to accomplish this. First, we'll store our values so that we can go from the index to the element in sub-linear time. This is easy. Second, we have to determine the index containing the minimum element within an arbitrary interval. This is a bit trickier to do in sub-linear time. We'll need a more advanced data structure. To see how this all works, you'll need to check out part 3, the grand finale of this series!

As a reminder, you should take a look at our mini-course on Stack. It'll teach you the basics of laying out a project and running commands on it using the Stack tool. You should enroll in the Monday Morning Haskell Academy to sign up! Once you know about Stack, it'll be a lot easier to try this problem out for yourself!

In addition to Stack, recursion also featured pretty heavily in our solution here. If you've done any amount of functional programming you've seen recursion in action. But if you want to solidify your knowledge, you should download our Recursion Workbook! It has two chapters worth of content on recursion and it has 10 practice problems you can work through! It also has a full test suite already, so you can use incremental test driven development!

---

Revision #1

Created 11 March 2022 05:51:45 by gasick

Updated 11 March 2022 17:11:17 by gasick