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

Testing in Haskell

В Haskell, мы бы предпочли, чтобы показателем было не только то что наш код собирается, но и то что он правильно работает. В Haskell это проще чем в других языках. Но конечно, всегда есть что по делать. Важная часть любого языка - написание тестов. В этой части рассмотрим несколько распространенных тестовых библиотек. А так же обсудим базовые парадигмы вокруг интеграционных тестов в процессе разработки.

TDD и базовые библиотеки

Сколько раз вы встречали зависимость от ошибок в вашем коде? Это может быть очень обидно для разработчика програмного обеспечеия. Вы отправили код в уверенности, что он отлично работает. Но теперь оказалось, что сломано, что-то другое. Даже хуже, вы обнаружили, что несмотря на то что ваш код правилно работает, он делает это очень медленно. Ваша система начинает ломаться при увеличении нагрузки, оставляя плохое впечаление пользователям.

Лучший способ избежать этих проблем это иметь автоматический код, который проверят состояние и производительность ваших программ. В этой части над тестированием в Haskell, мы посмотрим, что библиотеки могут использовать для тестирования и профилирования нашего кода. Первая статья пройдет по общей идее стоящей за TDD, и некоторыми базовыми библиотеками The best way to avoid these issues is to have automated code that verifies test conditions and the performance of your program. In this series on Testing with Haskell, we'll see what libraries we can use to test and profile our code. This first part goes over the general ideas behind test driven development (TDD) and some of the basic libraries we can use to make it work in Haskell. We'll also quickly examine why Haskell is a good fit for TDD.

If you're already familiar with libraries like HUnit and HSpec, you can move onto part 2 of this series, where we discuss how to identify performance issues using profiling.

To use testing properly, you'll need to have some understanding of how we organize projects in Haskell. I recommend you learn how to use Stack to organize your Haskell code. Learn how by taking our free Stack mini-course!

You can follow along with this code on the companion Github Repository for this series! In a few spots we'll reference specific files you can look at, so keep your eyes peeled!

FUNCTIONAL TESTING ADVANTAGES

Testing works best when we are testing specific functions. We pass input, we get output, and we expect the output to match our expectations. In Haskell, this is a approach is a natural fit. Functions are first class citizens. And our programs are largely defined by the composition of functions. Thus our code is by default broken down into our testable units.

Compare this to an object oriented language, like Java. We can test the static methods of a class easily enough. These often aren't so different from pure functions. But now consider calling a method on an object, especially a void method. Since the method has no return value, its effects are all internal. And often, we will have no way of checking the internal effects, since the fields could be private.

We'll also likely want to try checking certain edge cases. But this might involve constructing objects with arbitrary state. Again, we'll run into difficulties with private fields.

In Haskell, all our functions have return values, rather than depending on effects. This makes it easy for us to check their true results. Pure functions also give us another big win. Our functions generally have no side effects and do not depend on global state. Thus we don't have to worry about as many pathological cases that could impact our system.

TEST DRIVEN DEVELOPMENT

So now that we know why we're somewhat confident about our testing, let's explore the process of writing tests. The first step is to define the public API for a particular module. To do this, we define a particular function we're going to expose, and the types that it will take as input as output. Then we can stub it out as undefined, as suggested in this article on Compile Driven Learning. This makes it so that our code that calls it will still compile.

Now the great temptation for much all developers is to jump in and write the function. After all, it's a new function, and you should be excited about it!

But you'll be much better off in the long run if you first take the time to define your test cases. You should first define specific sets of inputs to your function. Then you should match those with the expected output of those parameters. We'll go over the details of this in the next section. Then you'll write your tests in the test suite, and you should be able to compile and run the tests. Since your function is still undefined, they'll all fail. But now you can implement the function incrementally.

Your next goal is to get the function to run to completion. Whenever you find a value you aren't sure how to fill in, try to come up with a base value. Once it runs to completion, the tests will tell you about incorrect values, instead of errors. Then you can gradually get more and more things right. Perhaps some of your tests will check out, but you missed a particular corner case. The tests will let you know about it.

WRITING OUR TEST SUITE

Suppose to start out, we're writing a function that will take three inputs. It should multiply the first two, and subtract the third. We'll start out by making it undefined. You can see this function in this module in the "library" of our Haskell project:

```
simpleMathFunction :: Int -> Int -> Int -> Int
simpleMathFunction a b c = undefined
```

Now let's write a test suite that will evaluate this function! To do this we'll go into the .cabal file for our project and add a test-suite section that looks like this:

```
test-suite unit-test
  type: exitcode-stdio-1.0
  main-is: UnitTest.hs
  other-modules:
      Paths_Testing
  hs-source-dirs:
      test
```

```
  ghc-options: -threaded -rtsopts -with-rtsopts=-N
  build-depends:
      Testing
    , base >=4.7 && <5
    , tasty
    , tasty-hunit
  default-language: Haskell2010
```

A test suite is like an executable. So it has a "Main" module specified by the main-is file, and you should specify the directory it lives in. Many of the other properties are pretty standardized. But the build-depends section will change depending on the test library you decide to use. In our case, we're going to test our code using the HUnit library combined with the Tasty framework.

USING HUNIT

We start out our test suite the same way we start out an executable, by creating a main function of type IO ():

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

main :: IO ()
main = ...
```

Most testing libraries have some kind of a "default" main function you can use that will provide most of their functionality. In the case of HUnit, we'll use defaultMain and then provide a TestTree expression:

```
main :: IO ()
main = ...

simpleMathTests :: TestTree
simpleMathTests = ...
```

We construct a "tree" in two ways. The first is to use an individual case with testCase. This function takes name to identify the case, and then a "predicate assertion".

```
simpleMathTests :: TestTree
simpleMathTests = testCase "Small Numbers" $
  ... -- (predicate assertion)
```

Ultimately, an assertion is just an IO action. But there are some special combinators we can use to make statements about the function of our code. The most common of these in HUnit are (@?=) and (@=?). These take two expressions and assert that they are equal. One of these should be the "actual" value we get from running our code, and the other should be the "expected" value. Here's our complete test case:

```
simpleMathTests :: TestTree
simpleMathTests = testCase "Small Numbers" $
  simpleMathFunction 3 4 5 @?= 7
```

The @=? operator works the same way, except you should reverse the "actual" and "expected" sides.

The other way to build a TestTree is to use testGroup. This simply takes a name for this layer of the tree, and then a list of TestTree elements. We can then use testCase for those specific elements.

```
simpleMathTests :: TestTree
simpleMathTests = testGroup "Simple Math Tests"
  [ testCase "Small Numbers" $
      simpleMathFunction 3 4 5 @?= 7
  ]
```

If you go to this file in the repository, you can add additional test cases to this list and run them!

RUNNING OUR TESTS

Our basic test suite is now complete! We can run this suite from our project directory by using the following command:

```
stack build Testing:test:unit-test
```

We can also use stack test to run all the different test suites we have. With our undefined function, we'll get this output:

```
Simple Math Tests
  Small Numbers:  FAIL
    Exception: Prelude.undefined
```

So as expected, our test cases fail, so we know how we can go about improving our code. So let's implement this function:

```
simpleMathFunction :: Int -> Int -> Int -> Int
simpleMathFunction a b c = a * b - c
```

And now everything succeeds!

```
Simple Math Tests
  Small Numbers:  OK


All 1 test passed (0.00s)
```

BEHAVIOR DRIVEN DEVELOPMENT

As you work on bigger projects, you'll find you aren't just interacting with other engineers on your team. There are often less technical stakeholders like project managers and QA testers. These folks are less concerned with the internal details of the code, but are focused more on its broader behavior. In these cases, you may want to adopt "behavior driven development." This is like test driven development, but with a different flavor. In this framework, you describe your code and its expected effects via a set of behaviors. Ideally, these are abstract enough that less technical people can understand them.

You as the engineer then want to be able to translate these behaviors into code. Luckily, Haskell is an immensely expressive language. You can often define your functions in such a way that they can almost read like English.

HSPEC

In Haskell, you can implement behavior driven development with the Hspec library. With this library, you describe your functions in a particularly expressive way. All your test specifications will belong to a Spec monad.

In this monad, you can use composable functions to describe the test cases. You will generally begin a description of a test case with the "describe" function. This takes a string describing the general overview of the test case.

```
simpleMathSpec :: Spec
simpleMathSpec = describe "Tests of our simple math function" $ do
  ...
```

You can then modify it by adding a different "context" for each individual case. The context function also takes a string. However, the idiomatic usage of context is that your string should begin with the words "when" or "with".

```
simpleMathSpec :: Spec
simpleMathSpec = describe "Tests of our simple math function" $ do
  context "when the numbers are small" $
    ...
  context "when the numbers are big" $
    ...
```

Now you'll describe each the actual test cases. You'll use the function "it", and then a comparison. The combinators in the Hspec framework are functions with descriptive names like shouldBe. So your case will start with a sentence-like description and context of the case. The the case finishes "it should have a certain result": x "should be" y. Here's what it looks like in practice:

```
simpleMathSpec :: Spec
simpleMathSpec = describe "Tests of our simple math function" $ do
```

```
  context "when the numbers are small" $
    it "Should match the our expected value" $
      simpleMathFunction 3 4 5 `shouldBe` 7
  context "when the numbers are big" $
    it "Should match the our expected value" $
      simpleMathFunction 22 12 64 `shouldBe` 200
```

It's also possible to omit the context completely:

```
simpleMathSpec :: Spec
simpleMathSpec = describe "Tests of our simple math function" $ do
  it "Should match the our expected value" $
    simpleMathFunction 3 4 5 `shouldBe` 7
  it "Should match the our expected value" $
    simpleMathFunction 22 12 64 `shouldBe` 200
```

Now to incorporate this into your main function, all you need to do is use hspec together with your Spec!

```
main :: IO ()
main = hspec simpleMathSpec
```

Note that Spec is a monad, so multiple tests are combined with "do" syntax. You can explore this library more and try writing your own test cases in this file in the repository!

At the end, you'll get neatly formatted output with descriptions of the different test cases. By writing expressive function names and adding your own combinators, you can make your test code even more self documenting.

```
Tests of our simple math function
  when the numbers are small
    Should match the our expected value
  when the numbers are big
    Should match the our expected value

Finished in 0.0002 seconds
2 examples, 0 failures
```

CONCLUSION

This concludes our introduction to testing in Haskell. We went through a brief description of the general practices of test-driven development. We saw why it's even more powerful in a functional, typed language like Haskell. We went over some of the basic testing mechanisms you'll find in the HUnit library. We then described the process of "behavior driven development", and how it differs from normal TDD. We concluded by showing how the HSpec library brings BDD to life in Haskell.

But testing correctness is only half the story! We also need to be sure that our code is performant enough. In part 2 of this series, we'll discuss how we can use the Criterion library to identify performance issues in our system.

If you want to see TDD in action and learn about a cool functional paradigm along the way, you should check out our Recursion Workbook. It has 10 practice problems complete with tests, so you can walk through the process of incrementally improving your code and finally seeing the tests pass!

If you want to learn the basics of writing your own test suites, you need to understand how Haskell code is organized! Take our quick and free Stack mini-course to learn how to use the Stack tool for this!

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.

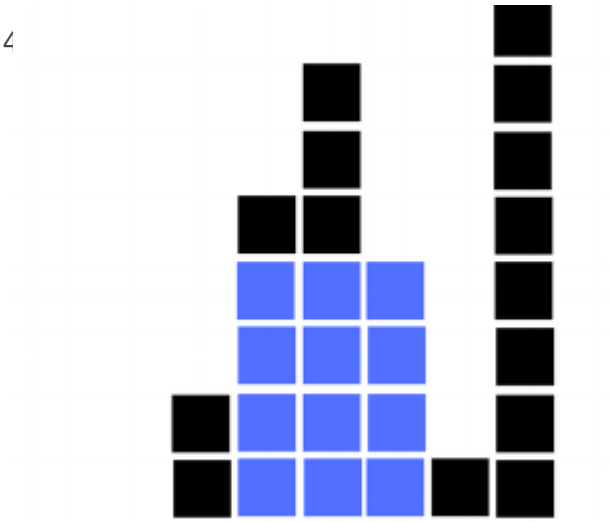
But improving our code can sometimes feel like taking shots in the dark. You'll spend a great deal of time tweaking a certain piece. Then you'll find you haven't actually made much of a dent in the total run time of the application. Certain operations generally take longer, like database calls, network 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:

```
>> stack build Testing:test:fences-tests
```

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 shold 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!

Improving Performance with Data Structures

Welcome to the third and final part of our Haskell testing series! In part 2, we wrote a solution to the "largest rectangle" problem. We implemented benchmarks to determine how well our code performs on certain inputs. First we used the Criterion library to get some measurements for our code. Then we were able to look at those measurements in some spiffy output. We also profiled our code to try to determine what part was slowing us down.

The profiling output highlighted two functions that were taking an awful lot of time. When we analyzed them, we found they were very inefficient. In this article, we'll resolve those problems and improve our code in a couple different ways. First, we'll use an array rather than a list to make our value accesses faster. Then, we'll add a cool data structure called a segment tree. This will help us to quickly get the smallest height value over a particular interval.

The code examples in this article series make good use of the Stack tool. If you've never used Stack before, you should check out our FREE Stack mini-course. It'll walk you through the basics of organizing your code, getting dependencies, and running commands.

Hopefully you've been following the Github Repository for this series! The improved code for this article can be found in the FencesFast module! You can also try re-doing some of these examples for yourself in a Test-Driven-Development style by working in this practice module with these unit tests!

WHAT WENT WRONG?

So first let's take some time to remind ourselves why our solution was inefficient. Both our minimum height function and our "value at index" function ran in O(n) time. This means each of them could scan the entire list in the worst case. Next we observed that both of these functions will get called O(n) times. Thus our total algorithm will be O(n^2) time. The time benchmarks we took

backed up this theory. Increasing our input size by a factor of 10 would often result in the solution taking 100 times longer.

The data structures we mentioned in the intro will help us get the values we need without doing a full scan. We'll start with the easier step, substituting an array for our list of values.

ARRAYS

Linked lists are very common when we're solving functional programming problems. They have some nice properties, and work very well with recursion. However, they do not allow fast access by index. For these situations, we need to use arrays. Arrays aren't as common in Haskell as other languages, and there are a few differences.

First, Haskell arrays have two type parameters. When you make an array in Java, you say whether it's an int array (int[]) or a string array (String[]), or whatever other type. So this is only a single parameter. Whenever we want to index into the array, we always use integers.

In Haskell, we get to choose both the type that the array stores AND the type that indexes the array. Now, the indexing type has to belong to the index (Ix) typeclass. And in this case we'll be using Int anyways. But it's cool to know that you have more flexibility. For instance, consider representing a matrix. In Java, we have to use an "array of arrays". This involves a lot of awkward syntax. In Haskell, we can instead use a single array indexed by tuples of integers! Accessing a Matrix with index (2, 1) feels a bit more natural than matrix[2][1]. We could also do something like index from 1 instead of 0 if the situation called for it.

So for our problem, we'll use Array Int Int for our inner fence values instead of a normal list. We'll only need to make a few code changes though! First, we'll import a couple modules and change our type to use the array:

```
import Data.Array
import Data.Ix (range)

...

newtype FenceValues = FenceValues { unFenceValues :: Array Int Int }
```

Next, instead of using (!!) to access by index, we'll use the specialized array index (!) operator to access them.

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

Finally, let's improve our minimumHeight function. We'll now use the range function on our array instead of resorting to drop and take. Note we now use right - 1 since we want to exclude the right endpoint of the interval.

```
where
  valsInInterval :: [(FenceIndex, Int)]
  valsInInterval = zip
    (FenceIndex <$> intervalRange)
    (map ((unFenceValues values) !) intervalRange)
    where
      intervalRange = range (left, right - 1)
```

We'll also have to change our benchmarking code to produce arrays instead of lists. (You can see these updates in the Fast benchmark:

```
import Data.Array (listArray)

...

randomList :: Int -> IO FenceValues
randomList n = FenceValues . mkListArray <$>
  (sequence $ replicate n (randomRIO (1, 10000 :: Int)))
  where
    mkListArray vals = listArray (0, (length vals) - 1) vals
```

Both our library and our benchmark now need to use array in their build-depends section of the Cabal file. We need to make sure we add this! Once we have, we can benchmark our code again, and we'll find it's already sped up quite a bit!

```
>> stack build Testing:bench:fences-fast-benchmark --profile
Running 1 benchmarks...
Benchmark fences-fast-benchmark: RUNNING...
benchmarking fences tests/Size 1 Test
```

```
time                 49.33 ns   (48.98 ns .. 49.71 ns)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 49.46 ns   (49.16 ns .. 49.86 ns)
std dev              1.105 ns   (861.0 ps .. 1.638 ns)
variance introduced by outliers: 33% (moderately inflated)

benchmarking fences tests/Size 10 Test
time                 4.541 µs   (4.484 µs .. 4.594 µs)
                     0.999 R²   (0.998 R² .. 1.000 R²)
mean                 4.496 µs   (4.456 µs .. 4.531 µs)
std dev              132.0 ns   (109.6 ns .. 164.3 ns)
variance introduced by outliers: 36% (moderately inflated)

benchmarking fences tests/Size 100 Test
time                 79.81 µs   (79.21 µs .. 80.45 µs)
                     0.999 R²   (0.999 R² .. 1.000 R²)
mean                 79.51 µs   (78.93 µs .. 80.39 µs)
std dev              2.396 µs   (1.853 µs .. 3.449 µs)
variance introduced by outliers: 29% (moderately inflated)

benchmarking fences tests/Size 1000 Test
time                 1.187 ms   (1.158 ms .. 1.224 ms)
                     0.995 R²   (0.992 R² .. 0.998 R²)
mean                 1.170 ms   (1.155 ms .. 1.191 ms)
std dev              56.61 µs   (48.02 µs .. 70.28 µs)
variance introduced by outliers: 37% (moderately inflated)

benchmarking fences tests/Size 10000 Test
time                 15.03 ms   (14.71 ms .. 15.32 ms)
                     0.997 R²   (0.994 R² .. 0.999 R²)
mean                 15.71 ms   (15.44 ms .. 16.03 ms)
std dev              729.7 µs   (569.3 µs .. 965.4 µs)
variance introduced by outliers: 16% (moderately inflated)

benchmarking fences tests/Size 100000 Test
time                 191.4 ms   (189.2 ms .. 193.9 ms)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 189.3 ms   (188.2 ms .. 190.5 ms)
std dev              1.471 ms   (828.0 µs .. 1.931 ms)
variance introduced by outliers: 14% (moderately inflated)
```

```
Benchmark fences-fast-benchmark: FINISH
```

Here's what the multiplicative factors are:

```
Size 1: 49.33 ns
Size 10: 4.451 µs (increased ~90x)
Size 100: 79.81 µs (increased ~18x)
Size 1000: 1.187 ms (increased ~15x)
Size 10000: 15.03 ms (increased ~13x)
Size 100000: 191.4 ms (increased ~13x)
```

For the later cases, increasing size by a factor 10 seems to only increase the time by a factor of 13-15. We could be forgiven for thinking we have achieved O(n log n) time already!

DIFFERENT TEST CASES

But something still doesn't sit right. We have to remember that the theory doesn't quite justify our excitement here. In fact our old code was so bad that the NORMAL case was O(n^2). Now it seems like we may have gotten O(n log n) for the average case. But we want to prepare for the worst case if we can. In this situation, our code will not be so performant when the lists of input heights is sorted!

```
main :: IO ()
main = do
  [l1, l2, l3, l4, l5, l6] <- mapM
    randomList [1, 10, 100, 1000, 10000, 100000]
  let l7 = sortedList
  defaultMain
    [ bgroup "fences tests"
       ...
      , bench "Size 100000 Test" $ whnf largestRectangle l6
      , bench "Size 100000 Test (sorted)" $ whnf largestRectangle l7
      ]
    ]

...
```

```
sortedList :: FenceValues
sortedList = FenceValues $ listArray (0, 99999) [1..100000]
```

We'll once again find that this last case takes a looooong time, and we'll see a big spike in run time.

```
>> stack build Testing:bench:fences-fast-benchmark --profile
Running 1 benchmarks...
Benchmark fences-fast-benchmark: RUNNING...

...

benchmarking fences tests/Size 100000 Test (sorted)
time                 378.1 s    (355.0 s .. 388.3 s)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 384.5 s    (379.3 s .. 387.2 s)
std dev              4.532 s    (0.0 s .. 4.670 s)
variance introduced by outliers: 19% (moderately inflated)

Benchmark fences-fast-benchmark: FINISH
```

It averages more than 6 minutes per case! But this time, we'll see the profiling output has changed. It only calls out various portions of minimumHeightIndexValue! We no longer spend a lot of time in valueAtIndex.

```
COST CENTRE                                          %time %alloc

minimumHeightIndexValue.valsInInterval               65.0   67.7
minimumHeightIndexValue                              22.4    0.0
minimumHeightIndexValue.valsInInterval.intervalRange 12.4   32.2
```

So now we have to solve this new problem by improving our calculation of the minimum.

SEGMENT TREES

Our current approach still requires us to look at every element in our interval. Even though some of our intervals will be small, there will be a lot of these smaller calls, so the total time is still O(n^2).

We need a way to find the smallest item and value on a given interval without resorting to a linear scan.

One idea would be to develop an exhaustive list of all the answers to this question right at the start. We could make a mapping from all possible intervals to the smallest index and value in the interval. But this won't help us in the end. There are still n^2 possible intervals. So creating this data structure will still mean that our code takes O(n^2) time.

But we're on the right track with the idea of doing some of the work before hand. We'll have to use a data structure that's not an exhaustive listing though. Enter segment trees.

A segment tree has the same structure as a binary search tree. Instead of storing a single value though, each node corresponds to an interval. Each node will store its interval, the smallest value over that interval, and the index of that value.

The top node on the tree will refer to the interval of the whole array. It'll store the pair for the smallest value and index overall. Then it will have two children nodes. The left one will have the minimum pair over the first half of the tree, and the right one will have the second half. The next layer will break it up into quarters, and so on.

As an example, let's consider how we would determine the minimum pair starting from the first quarter point and ending at the third quarter point. We'll do this using recursion. First, we'll ask the left subtree for the minimum pair on the interval from the quarter point to the half point. Then we'll query the right tree for the smallest pair from the half point to the three-quarters point. Then we can take the smallest of those and return it. I won't go into all the theory here, but it turns out that even in the worst case this operation takes O(log n) time.

DESIGNING OUR SEGMENT TREE

There is a library called Data.SegmentTree on hackage. But our code is short and specialized enough that we can do this from scratch. We'll compose our tree from SegmentTreeNodes. Each node is either empty, or it contains six fields. The first two refer to the interval the node spans. The

next will be the minimum value and the index of that value over the interval. And then we'll have fields for each of the children nodes of this node:

```
data SegmentTreeNode = ValueNode
  { fromIndex :: FenceIndex
  , toIndex :: FenceIndex
  , value :: Int
  , minIndex :: FenceIndex
  , leftChild :: SegmentTreeNode
  , rightChild :: SegmentTreeNode
  }
  | EmptyNode
```

We could make this Segment Tree type a lot more generic so that it isn't restricted to our fence problem. I would encourage you to take this code and try that as an exercise!

BUILDING THE SEGMENT TREE

Now we'll add our preprocessing step where we'll actually build the tree itself. This will use the same interval/tail pattern we saw before. In the base case, the interval's span is only 1, so we make a node containing that value with empty sub-children. We'll also add a catchall that returns an EmptyNode:

```
buildSegmentTree :: Array Int Int -> SegmentTreeNode
buildSegmentTree ints = buildSegmentTreeTail
  ints
  (FenceInterval ((FenceIndex 0), (FenceIndex (length (elems ints)))))

buildSegmentTreeTail :: Array Int Int -> FenceInterval -> SegmentTreeNode
buildSegmentTreeTail array
  (FenceInterval (wrappedFromIndex@(FenceIndex fromIndex), wrappedToIndex@(FenceIndex
toIndex)))
  | fromIndex + 1 == toIndex = ValueNode
      { fromIndex = wrappedFromIndex
```

```
      , toIndex = wrappedToIndex
      , value = array ! fromIndex
      , minIndex = wrappedFromIndex
      , leftChild = EmptyNode
      , rightChild = EmptyNode
      }
  | ... TODO
  | otherwise = EmptyNode
```

Now our middle case will be the standard case. First we'll divide our interval in half and make two recursive calls.

```
where
  average = (fromIndex + toIndex) `quot` 2
  -- Recursive Calls
  leftChild = buildSegmentTreeTail
    array (FenceInterval (wrappedFromIndex, (FenceIndex average)))
  rightChild = buildSegmentTreeTail
    array (FenceInterval ((FenceIndex average), wrappedToIndex))
```

Next we'll write a function that'll extract the minimum value and index, but handle the empty node case. This provided maxBound as the "minimum" so comparisons will always favor the non-empty nodes:

```
-- Get minimum val and index, but account for empty case.
valFromNode :: SegmentTreeNode -> (Int, FenceIndex)
valFromNode EmptyNode = (maxBound :: Int, FenceIndex (-1))
valFromNode n@ValueNode{} = (value n, minIndex n)
```

Now, back in buildSegmentTreeTail, we'll compare the three cases for the minimum. It'll likely be the values from the left or the right. Otherwise it's the current value.

```
where
  ...
  leftCase = valFromNode leftChild
  rightCase = valFromNode rightChild
  currentCase = (array ! fromIndex, wrappedFromIndex)
  (newValue, newIndex) = min (min leftCase rightCase) currentCase
```

Finally we'll complete our definition by filling in the missing variables in the middle/normal case. You can look at the complete definition here:

```
buildSegmentTreeTail :: Array Int Int -> FenceInterval -> SegmentTreeNode
buildSegmentTreeTail array
  (FenceInterval (wrappedFromIndex@(FenceIndex fromIndex), wrappedToIndex@(FenceIndex
toIndex)))
  | ... -- base case
  | fromIndex < toIndex = ValueNode
    { fromIndex = wrappedFromIndex
    , toIndex = wrappedToIndex
    , value = newValue
    , minIndex = newIndex
    , leftChild = leftChild
    , rightChild = rightChild
    }
  | otherwise = EmptyNode
    where
      average = ...
      leftChild = ...
      rightChild = ...

      leftCase = valFromNode leftChild
      rightCase = valFromNode rightChild
      currentCase = (array ! fromIndex, wrappedFromIndex)
      (newValue, newIndex) = min (min leftCase rightCase) currentCase

valFromNode :: SegmentTreeNode -> (Int, FenceIndex)
valFromNode = ...
```

FINDING THE MINIMUM

Now let's write the critical function of finding the minimum over the given interval. This will be like our slower version, but we'll add our tree as another parameter. Then we'll handle the EmptyNode case in the same way as above. Then we can unwrap our values for the full case:

```
minimumHeightIndexValue :: FenceValues -> SegmentTreeNode -> FenceInterval -> (FenceIndex,
Int)
minimumHeightIndexValue values tree
  originalInterval@(FenceInterval (FenceIndex left, FenceIndex right)) =
  case tree of
    EmptyNode -> (maxBound :: Int, -1)
    ValueNode
      { fromIndex = FenceIndex nFromIndex
      , toIndex = FenceIndex nToIndex
      , value = nValue
      , minIndex = nMinIndex
      , leftChild = nLeftChild
      , rightChild = nRightChild} -> ...
```

The first case we'll handle is that the current node exactly matches the interval we are passed. Obviously we can simply supply the value and index here:

```
case tree of
  ValueNode
    { fromIndex = FenceIndex nFromIndex
    , toIndex = FenceIndex nToIndex
    , value = nValue
    , minIndex = nMinIndex
    , ... } -> if left == nFromIndex && right == nToIndex
      then (nMinIndex, nValue)
      else ...
```

Next we'll observe two cases that will need only one recursive call. If the right index is below the midway point, we recursively call to the left sub-child. And if the left index is above the midway point, we'll call on the right side (we'll calculate the average later).

```
case tree of
  ValueNode
    { ... } -> if left == nFromIndex && right == nToIndex
      then (nMinIndex, nValue)
      else if right < average
        then minimumHeightIndexValue values nLeftChild originalInterval
        else if left >= average
          then minimumHeightIndexValue values nRightChild originalInterval
```

```
          else ...
        where
          average = (nFromIndex + nToIndex) `quot` 2
```

Finally we have the tricky part. If the interval does cross the halfway mark, we'll have to divide it into two sub-intervals. Then we'll make two recursive calls, and get their solutions. Finally, we'll compare the two solutions and take the smaller one. This requires the definition of one more helper function minTuple, to compare indices by their corresponding heights.

```
case tree of
  ValueNode
    { ... } -> if left == nFromIndex && right == nToIndex
      then (nMinIndex, nValue)
      else if right < average
        then ... -- left recursive case
        else if left >= average
          then ... -- right recursive case
          else minTuple leftResult rightResult
        where
          average = (nFromIndex + nToIndex) `quot` 2
          leftResult = minimumHeightIndexValue values nLeftChild
            (FenceInterval (FenceIndex left, FenceIndex average))
          rightResult = minimumHeightIndexValue values nRightChild
            (FenceInterval (FenceIndex average, FenceIndex right))
          minTuple :: (FenceIndex, Int) -> (FenceIndex, Int) -> (FenceIndex, Int)
          minTuple old@(_, heightOld) new@(_, heightNew) =
            if heightNew < heightOld then new else old
```

Again, you can see the complete function here.

TOUCHING UP THE REST

Once we've accomplished this, the rest is pretty straightforward. First, we'll build our segment tree at the beginning and pass it as a parameter to our function. Then we'll plug in our new minimum function in place of the old one. We'll make sure to add the tree to each recursive call as well.

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

...
-- Notice the extra parameter
largestRectangleAtIndices :: FenceValues -> SegmentTreeNode -> FenceInterval -> FenceSolution
largestRectangleAtIndices
  values
  tree
...
      where
      ...
      -- And down here add it to each call
      (minIndex, minValue) = minimumHeightIndexValue values tree interval
      leftCase = largestRectangleAtIndices values tree (FenceInterval (leftIndex, minIndex))
      rightCase = if minIndex + 1 == rightIndex
        then FenceSolution (maxBound :: Int)
        else largestRectangleAtIndices values tree (FenceInterval (minIndex + 1, rightIndex))
```

And now we can run our benchmark again. This time, we'll see that our code runs a great deal faster on both large cases! Success!

```
benchmarking fences tests/Size 100000 Test
time                 179.1 ms   (173.5 ms .. 185.9 ms)
                     0.999 R²   (0.998 R² .. 1.000 R²)
mean                 184.1 ms   (182.7 ms .. 186.1 ms)
std dev              2.218 ms   (1.197 ms .. 3.342 ms)
variance introduced by outliers: 14% (moderately inflated)

benchmarking fences tests/Size 100000 Test (sorted)
time                 238.4 ms   (227.2 ms .. 265.1 ms)
                     0.998 R²   (0.989 R² .. 1.000 R²)
mean                 243.5 ms   (237.0 ms .. 251.8 ms)
std dev              8.691 ms   (2.681 ms .. 11.83 ms)
variance introduced by outliers: 16% (moderately inflated)
```

CONCLUSION

So in this series, we learned a whole lot. In part 1, we talked about the basic ideas behind test driven development and some Haskell unit testing libraries. In part 2, we then covered how to create benchmarks for our code using Cabal/Stack. When we ran those benchmarks, we found results took longer than we would like. We then used profiling to determine what the problematic functions were.

To solve the problems we found, we dove head-first into some data structures knowledge. We saw first hand how changing the underlying data structures of our program could improve our performance. We also learned about arrays, which are somewhat overlooked in Haskell. Then we built a segment tree from scratch and used its API to enable our program's improvements.

If you want some extra practice with Test Driven Development, benchmarks, and our Fence example, you can take a look at our practice module and corresponding practice test suite. You can solve the most important parts of the algorithm using TDD and writing your test cases as you go along!

This problem involved many different uses of recursion. If you want to become a better functional programmer, you'd better learn recursion. If you want a better grasp of this fundamental concept, you should check out our free Recursion Workbook. It has two chapters of useful information as well as 10 practice problems!

Finally, be sure to check out our Stack mini-course. Once you've mastered the Stack tool, you'll be well on your way to making Haskell projects like a Pro!