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

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!

Revision #1
Created 2022-03-11 05:55:07 UTC by gasick
Updated 2022-03-11 17:11:17 UTC by gasick