vector: Efficient Packed-Memory Data Representations

The de facto standard package in the Haskell ecosystem for integer-indexed array data is the vector package. This corresponds at a high level to arrays in C, or the vector class in C++'s STL. However, the vector package offers quite a bit of functionality not familiar to those used to the options in imperative and mutable languages.

While the interface for vector is relatively straightforward, the abundance of different modules can be daunting. This article will start off with an overview of terminology to guide you, and then step through a number of concrete examples of using the package.

Tutorial exercise

To help motivate learning, keep in mind this exercise while reading through the content below, and try to implement a solution. Use mutable vectors to write a program that will deal you a random hand of poker. Bonus: use an unboxed vector. Double bonus: minimize the memory representation.

Example

Since we're about to jump into a few sections of descriptive text, let's kick this off with a concrete example to whet your appetite. We're going to count the frequency of different bytes that appear on standard input, and then display this content.

Note that this example is purposely written in a very generic form. We'll build up to handling this form throughout this article.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE FlexibleContexts #-}
import           Control.Monad.Primitive     (PrimMonad, PrimState)
import qualified Data.ByteString.Lazy        as L
import qualified Data.Vector.Generic.Mutable as M
import qualified Data.Vector.Unboxed         as U
import           Data.Word                   (Word8)

main :: IO ()
main = do
    -- Get all of the contents from stdin
    lbs <- L.getContents

    -- Create a new 256-size mutable vector
    -- Fill the vector with zeros
    mutable <- M.replicate 256 0

    -- Add all of the bytes from stdin
    addBytes mutable lbs

    -- Freeze to get an immutable version
    vector <- U.unsafeFreeze mutable

    -- Print the frequency of each byte
    -- In newer vectors: we can use imapM_
    U.zipWithM_ printFreq (U.enumFromTo 0 255) vector

addBytes :: (PrimMonad m, M.MVector v Int)
         => v (PrimState m) Int
         -> L.ByteString
         -> m ()
addBytes v lbs = mapM_ (addByte v) (L.unpack lbs)

addByte :: (PrimMonad m, M.MVector v Int)
        => v (PrimState m) Int
        -> Word8
        -> m ()
addByte v w = do
    -- Read out the old count value
    oldCount <- M.read v index
    -- Write back the updated count value
    M.write v index (oldCount + 1)
  where
    -- Indices in vectors are always Ints. Our bytes come in as Word8, so we
    -- need to convert them.
    index :: Int
    index = fromIntegral w

printFreq :: Int -> Int -> IO ()
printFreq index count = putStrLn $ concat
    [ "Frequency of byte "
    , show index
```

```
    , ": "
    , show count
    ]
```

Terminology

There are two different varieties of vectors: immutable and mutable. Immutable vectors (such as provided by the `Data.Vector` module) are essentially swappable with normal lists in Haskell, though with drastically different performance characteristics (discussed below). Th high-level API is similar to lists, it implements common typeclasses like `Functor` and `Foldable`, and plays quite nicely with parallel cod

By contrast, mutable vectors are much closer to C-style arrays. Operations working on these values must live in the `IO` or `ST` monads (se `PrimMonad` below for more details). Concurrent access from multiple threads has all of the normal concerns of shared mutable state. An perhaps most importantly for usage: mutable vectors can be *much* more efficient for certain use cases.

However, that's not the only dimension of choice you get in the vector package. vector itself defines three flavors: boxed (`Data.Vector`/`Data.Vector.Mutable`), storable (`Data.Vector.Storable` and `Data.Vector.Storable.Mutable`), and unboxed (`Data.Vector.Unboxed` and `Data.Vector.Unboxed.Mutable`). (There's also technically primitive vectors, but in practice you should always prefer unboxed vectors; see the module documentation for more information on the distinction here.)

All vectors a *spine strict*. Boxed vectors are value lazy, while storable and unboxed vectors are also value strict. We'll cover these points with examples below.

And our final point: in addition to having these three flavors, the vector package provides a typeclass-based interface which allows you to write code that works in any of these three (plus other vector types that may be defined in other packages, like hybrid-vectors). These interfaces are in `Data.Vector.Generic` and `Data.Vector.Generic.Mutable`. When using these interfaces, you must still eventually choose a concrete representation, but your helper code can be agnostic to what it is.

What's nice is that - with small differences - all four mutable modules have the same interface, and all four immutable modules have the same interface. This means you can focus on learning one type of vector, and almost for free have that knowledge apply to other types as well. It then just becomes a question of choosing the representation that best fits your use case, which we'll get to shortly.

Efficiency

Standard lists in Haskell are immutable, singly-linked lists. Every time you add another value to the front of the list, it has to allocate another heap object for that cell, create a pointer to the head of the original list, and create a pointer to the value in the current cell. This takes up a lot of memory for holding pointers, and makes it inefficient to index or traverse the list (indexing to position N requires N pointer dereferences).

In contrast, vectors are stored in a contiguous set of memory locations, meaning random access is a constant time operation, and the memory overhead per additional item in the vector is much smaller (depending on the type of vector, which we'll cover in a moment). However, compared to lists, prepending an item to a vector is relatively expensive: it requires creating a new buffer in memory, copying th old values, and then adding the new value.

There are other data structures that can be considered for list-like data, such as `Seq` from containers, or in some cases a `Set`, `IntMap`, o `Map`. Figuring out the best choice for each use case can only be reliably determined via profiling and benchmarking. As a general rule though, a densely populated collection requiring integral or random access to the values will be best served by a vector.

Now let's talk about some of the other things that make vector so efficient.

Boxed, storable and unboxed

Boxed vectors hold normal Haskell values. These can be *any* values at all, and are stored on the heap with pointers kept in the vector. Th advantage is that this works for all datatypes, but the extra memory overhead for the pointers and the indirection of needing to dereference those pointers makes them (relative to the next two types) inefficient.

Storable and unboxed vectors both store their data in a byte array, avoiding pointer indirection. This is more memory efficient and allows better usage of caches. The distinction between storable and unboxed vectors is subtle:

- Storable vectors require data which is an instance of the `Storable` type class. This data is stored in `malloc`ed memory, which is *pinned* (the garbage collector can't move it around). This can lead to memory fragmentation, but allows the data to be shared over the C FFI.
- Unboxed vectors require data which is an instance of the `Prim` type class. This data is stored in GC-managed *unpinned* memory, which helps avoid memory fragmentation. However, this data cannot be shared over the C FFI.

Both the `Storable` and `Prim` typeclasses provide a way to store a value as bytes, and to load bytes into a value. The distinction is what type of bytearray is used.

As usual, the only true measure of performance will be benchmarking. However, as a general guideline:

- If you don't need to pass values to a C FFI, and you have a `Prim` instance, use unboxed vectors.
- If you have a `Storable` instance, use a storable vector.
- Otherwise, use a boxed vector.

There are also other issues to consider, such as the fact that boxed vectors are instances of `Functor` while storable and unboxed vector are not.

Stream fusion

Take a guess how much memory the following program will take to run:

```
import qualified Data.Vector.Unboxed as V

main :: IO ()
main = print $ V.sum $ V.enumFromTo 1 (10^9 :: Int)
```

A valid guess may be `10^9 * sizeof int` bytes. However, when compiled with optimizations (`-O2`) on my system, it allocates a total o only 52kb! How it is possible to create a one billion integer array without using up 4-8GB of memory?

The vector package has a powerful technique: stream fusion. Using GHC rewrite rules, it's able to find many cases where creating a vecto is unnecessary, and instead create a tight inner loop. In our case, GHC will end up generating code that can avoid touching system memory, and instead work on just the registers, yielding not only a tiny memory footprint, but performance close to a for-loop in C. This is one of the beauties of this library: you can write high-level code, and optimizations can churn out something much more CPU-friendly.

Slicing

Above we discussed the problem of appending values to the front of a vector. However, one place where vector shines is with *slicing*, or taking a subset of the vector. When dealing with immutable vectors, slicing is a safe operation, with slices being sharable with multiple threads. Slicing also works with mutable vectors, but as usual you need to be a bit more careful.

Replacing lists

Enough talk! Let's start using vector. Assuming you're familiar with the list API, this should look rather boring.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.Vector as V

main :: IO ()
main = do
    let list = [1..10] :: [Int]
        vector = V.fromList list :: V.Vector Int
        vector2 = V.enumFromTo 1 10 :: V.Vector Int
    print $ vector == vector2 -- True
    print $ list == V.toList vector -- also True
    print $ V.filter odd vector -- 1,3,5,7,9
    print $ V.map (* 2) vector -- 2,4,6,...,20
    print $ V.zip vector vector -- (1,1),(2,2),...(10,10)
    print $ V.zipWith (*) vector vector -- (1,4,9,16,...,100)
    print $ V.reverse vector -- 10,9,...,1
    print $ V.takeWhile (< 6) vector -- 1,2,3,4,5
    print $ V.takeWhile odd vector -- 1
    print $ V.takeWhile even vector -- []
    print $ V.dropWhile (< 6) vector -- 6,7,8,9,10
    print $ V.head vector -- 1
    print $ V.tail vector -- 2,3,4,...,10
    print $ V.head $ V.takeWhile even vector -- exception!
```

Hopefully there's nothing too surprising about this. Most `Prelude` functions that apply to lists have a corresponding vector function. If you know what a function does in `Prelude`, you probably know what it does in `Data.Vector`. This is the simplest usage of the vector package: import `Data.Vector` qualified, convert to/from lists with `V.fromList` and `V.toList`, and then prefix your function calls witl `V.`.

- Exercise 1: Try out some other functions available in the [`Data.Vector` module]. In particular, try some of the fold functions, which \ haven't covered here.
- Exercise 2: Try using the `Functor`, `Foldable`, and `Traversable` versions of functions with a vector
- Exercise 3: Use an unboxed (or storable) vector instead of the boxed vectors we were using above. What code did you have to change from the original example? Do all of your examples from exercise 2 still work?

There are also a number of functions in the `Data.Vector` module with no corresponding function in `Prelude`. Many of these are relate to mutable vectors (which we'll cover shortly). Others are present to provide more efficient means of manipulating a vector, based on the special in-memory representation.

Mutable vectors

I want to test how fair the `System.Random` number generator is at generating numbers between 0 and 9, inclusive. I want to generate 1,000,000 random values, count the frequency of each result, and then print how often each value appeared. Let's first implement this using immutable vectors:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import           Data.Vector.Unboxed ((!), (//))
import qualified Data.Vector.Unboxed as V
import           System.Random       (randomRIO)

main :: IO ()
main = do
    let v0 = V.replicate 10 (0 :: Int)

        loop v 0 = return v
        loop v rest = do
            i <- randomRIO (0, 9)
            let oldCount = v ! i
                v' = v // [(i, oldCount + 1)]
            loop v' (rest - 1)

    vector <- loop v0 (10^6)
    print vector
```

We've introduced the `!` operator for indexing, and the `//` operator for updating. Other than that, this is fairly straightforward code. When ran this on my system, it had 48MB maximum memory residency, and took 1.968s to complete. Surely we can do better.

This problem is inherently better as a mutable state one: instead of generating a new immutable `Vector` for each random number generated, we'd like to simply increment a piece of memory. Let's rewrite this to use a mutable, unboxed vector:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import           Control.Monad               (replicateM_)
import           Data.Vector.Unboxed         (freeze)
import qualified Data.Vector.Unboxed.Mutable as V
import           System.Random               (randomRIO)

main :: IO ()
main = do
    vector <- V.replicate 10 (0 :: Int)

    replicateM_ (10^6) $ do
        i <- randomRIO (0, 9)
        oldCount <- V.read vector i
        V.write vector i (oldCount + 1)

    ivector <- freeze vector
    print ivector
```

Once again, we use `replicate` to create a size-10 vector filled with 0. But now we've created a mutable vector (note the change in import). We then use `replicateM_` to perform the inner action 1,000,000 times, namely: generate a random index, read the old value at that index, increment it, and write it back.

After we're finished, we *freeze* the vector (more on that in the next section) and print it. The resulting distribution of values is the same (c close - we are dealing with random numbers here) as the previous calculation using an immutable vector. But instead of 48MB and 1.968s this program has a maximum residency of 44KB and runs in 0.247s! That's a significant improvement!

If we feel like being even more adventurous, we can replace our `read` and `write` calls with `unsafeRead` and `unsafeWrite`. That will disable some bounds checks before reading and writing. This can be a nice performance boost in very tight loops, but has the potential t segfault your program, so caveat emptor! For example, try replacing `replicate 10` with `replicate 9`, change the `read` for an `unsafeRead`, and run your program. You'll see something like:

```
internal error: evacuate: strange closure type -1944718914
    (GHC version 7.10.2 for x86_64_unknown_linux)
    Please report this as a GHC bug:  https://www.haskell.org/ghc/reportabug
Aborted (core dumped)
```

The same logic applies to the other `unsafe` functions in vector. The nomenclature means: `unsafe` may segfault your whole process, wh not-marked-unsafe may just throw an impure exception (also not great, but certainly better than a segfault).

And if you were curious: on my system using `unsafeRead` and `unsafeWrite` speeds the program up marginally, from 0.247s to 0.233s. our example, most of our time is spent on generating the random numbers, so taking off the safety checks does not have a significant impact.

Freezing and thawing

We used the `freeze` function above. The behavior of this may not be immediately obvious. When you freeze a mutable vector, what happens is:

1. A new mutable vector of the same size is created
2. Each value in the original mutable vector is copied to the new mutable vector
3. A new immutable vector is created out of the memory space used by the new mutable vector

Why not just freeze it in place? Two reasons, actually:

1. It has the potential to break referential transparency. Consider this code:

   ```
   #!/usr/bin/env stack
   -- stack --resolver lts-12.21 script
   import           Data.Vector.Unboxed         (freeze)
   import qualified Data.Vector.Unboxed.Mutable as V

   main :: IO ()
   main = do
       vector <- V.replicate 1 (0 :: Int)
       V.write vector 0 1
       ivector <- freeze vector
       print ivector
       V.write vector 0 2
       print ivector
   ```

   If we froze the vector in-place in the call to `freeze`, then the second `write` call would modify our `ivector` value, meaning that th first and second call to `print ivector` would have different results!

2. When you freeze a mutable vector, its memory is marked differently for garbage collection purposes. Later attempts to write to tha same memory can lead to a segfault.

However, if you really want to avoid that extra buffer copy, and are certain it's safe, you can use `unsafeFreeze`. And in fact, our random number example above is a case where `freeze` can be safely replaced by `unsafeFreeze`, since after the freeze, the original mutable vector is never used again.

- Exercise 1: Go ahead and make that swap and confirm that your program works as expected.
- Exercise 2: In the program just above (with `V.replicate 1 (0 :: Int)`), replace `freeze` with `unsafeFreeze`. What result do you see?

The opposite of `freeze` is `thaw`. Similar to `freeze`, `thaw` will copy to a new mutable vector instead of exposing the current memory buffer. And also, like `freeze`, there's an `unsafeThaw` that turns off the safety measures. Like everything `unsafe`: caveat emptor!

(We'll cover some functions like `create` that provide safe wrappers around `unsafeFreeze` and `unsafeThaw` later.)

PrimMonad

If you look at the mutation functions we used above like `read` and `write`, you can tell that they were looking in the `IO` monad. However, vector is more generic than that, and will allow your mutations to live in any *primitive monad*, meaning: `IO`, strict `ST s`, and transformers sitting on top of those two. The type class controlling this is `PrimMonad`.

You can get more information on `PrimMonad` in the Primitive Haskell article. Without diving into details: every primitive monad also has a associated primitive state token type, which is captured with `PrimState`. As a result, the type signatures for `read` and `write` (for boxe vectors) look like:

```
read :: PrimMonad m => MVector (PrimState m) a -> Int -> m a
write :: PrimMonad m => MVector (PrimState m) a -> Int -> a -> m ()
```

Every mutable vector takes two type parameters: the state token of the monad it lives in, and the type of value it holds. These gymnastic may seem overkill now, but are necessary for making mutable vectors both versatile in multiple monads, and type safe.

modify and the ST monad

Let's check out a particularly complicated type signature (for unboxed vectors):

```
modify :: Unbox a => (forall s. MVector s a -> ST s ()) -> Vector a -> Vector a
```

What this function does is:

1. Creates a new mutable buffer the same length as the original vector
2. Copies the values from the original vector into the new mutable vector
3. Runs the provided `ST` action on the provided mutable vector
4. Unsafely freezes the mutable vector and returns it.

What's great about this function is that it does the minimal amount of buffer copying to be safe, and that it can be used from pure code (since all side-effects are captured inside the `ST` action you provide).

- Exercise 1: Steps 1 and 2 should look pretty similar to a function we discussed above. Can you figure out which one it is?
- Exercise 2: Implement `modify` yourself using functions we've discussed and `runST` from `Control.Monad.ST`.

Let's use our new function to implement a Fisher-Yates shuffle. If we start with a vector of size 20, we'll generate a random number between 0 and 19. Then we'll swap position 19 with that generated random number. Then we'll loop, but this time with a random number between 0 and 18 and swapping with position 18. We continue until we get down to 0.

```
import           Control.Monad.Primitive     (PrimMonad, PrimState)
import qualified Data.Vector.Unboxed         as V
import qualified Data.Vector.Unboxed.Mutable as M
import           System.Random               (StdGen, getStdGen, randomR)

shuffleM :: (PrimMonad m, V.Unbox a)
         => StdGen
         -> Int -- ^ count to shuffle
         -> M.MVector (PrimState m) a
         -> m ()
shuffleM _ i _ | i <= 1 = return ()
shuffleM gen i v = do
    M.swap v i' index
    shuffleM gen' i' v
  where
    (index, gen') = randomR (0, i') gen
    i' = i - 1

shuffle :: V.Unbox a
        => StdGen
        -> V.Vector a
        -> V.Vector a
shuffle gen vector = V.modify (shuffleM gen (V.length vector)) vector

main :: IO ()
main = do
    gen <- getStdGen
    print $ shuffle gen $ V.enumFromTo 1 (20 :: Int)
```

Notice how `shuffleM` is a mutable, side-effecting function. However, `shuffle` itself is pure.

Generic

After everything else we've dealt with, `Generic` is a relatively easy addition. We introduce two new typeclasses:

```
class MVector v a
class MVector (Mutable v) a => Vector v a
```

Said in English: an instance `MVector v a` is a mutable vector of type `v` that can hold values of type `a`. The `Vector v a` is the immutab counterpart to some mutable vector. You can find the mutable version with `Mutable v`.

One important thing to keep in mind is *kinds*. The kind of the `v` is `MVector v a` is `* -> * -> *`, since it takes parameters for both the state token and the value it holds. With the immutable `Vector v a`, the `v` is of kind `* -> *`. Was that a little abstract? No problem, som type signatures should help:

```
length :: MVector v a => v s a -> Int
length :: Vector v a => v a -> Int

read :: (PrimMonad m, MVector v a) => v (PrimState m) a -> Int -> m a
```

It takes a bit of time to get used to these generic classes, but once you do it's fairly easy to use them. The best advice is to practice! And such:

- Exercise: modify the `shuffle` program above to work on a generic vector instead of specifically on an unboxed vector.

The final trick when working with generic vectors is that, ultimately, you will need to provide a concrete type. If you forget to do so, you'll end up with error messages that look like the following:

```
stream.hs:28:13:
    No instance for (V.Vector v0 Int) arising from a use of ‘shuffle’
    In the expression: shuffle gen
    In the second argument of ‘($)’, namely
      ‘shuffle gen $ V.enumFromTo 1 (20 :: Int)’
    In a stmt of a 'do' block:
      print $ shuffle gen $ V.enumFromTo 1 (20 :: Int)
```

As an example:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE FlexibleContexts #-}
import qualified Data.Vector as VB
import qualified Data.Vector.Storable as VS
import qualified Data.Vector.Unboxed as VU
import qualified Data.Vector.Generic as V

myFunc :: V.Vector v Int => v Int -> IO ()
myFunc = V.mapM_ print . V.map (* 2) . V.filter odd

main :: IO ()
main = do
  myFunc $ VB.enumFromTo 1 10
  myFunc $ VS.enumFromTo 1 10
  myFunc $ VU.enumFromTo 1 10
```

vector-algorithms

A package of note is vector-algorithms, which provides some algorithms (mostly sort) on mutable vectors. For example, let's generate 10(random numbers and then sort them.

```
import           Data.Vector.Algorithms.Merge (sort)
import qualified Data.Vector.Generic.Mutable  as M
import qualified Data.Vector.Unboxed          as V
import           System.Random                (randomRIO)

main :: IO ()
main = do
    vector <- M.replicateM 100 $ randomRIO (0, 999 :: Int)
    sort vector
    V.unsafeFreeze vector >>= print
```

- Exercise 1: write a helper function `sortImmutable` that uses `modify` and `sort` from vector-algorithms to sort an immutable vect safely
- Exercise 2: rewrite the main function above to use `sortImmutable` and only the immutable vector API

- Exercise 3: is your new version more efficient, less efficient, or the same? Explain.

mwc-random

Another library to mention now is mwc-random, a random number generation library built on top of vector and primitive. Its API can be a bit daunting initially, but given your newfound understanding of the vector package, the API might make a lot more sense now. It provide a `Gen s` type, where `s` is some state token. You can then use `uniform` and `uniformR` to get random numbers out of that generator.

As a final example, here's how we can shuffle the numbers 1-20 using mwc-random.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import           Control.Monad.ST            (ST)
import qualified Data.Vector.Unboxed         as V
import qualified Data.Vector.Unboxed.Mutable as M
import           System.Random.MWC           (Gen, uniformR, withSystemRandom)

shuffleM :: V.Unbox a
         => Gen s
         -> Int -- ^ count to shuffle
         -> M.MVector s a
         -> ST s ()
shuffleM _ i _ | i <= 1 = return ()
shuffleM gen i v = do
    index <- uniformR (0, i') gen
    M.swap v i' index
    shuffleM gen i' v
  where
    i' = i - 1

main :: IO ()
main = do
    vector <- withSystemRandom $ \gen -> do
        vector <- V.unsafeThaw $ V.enumFromTo 1 (20 :: Int)
        shuffleM gen (M.length vector) vector
        V.unsafeFreeze vector
    print vector
```

Strictness

Guess the output:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE FlexibleContexts #-}
import qualified Data.Vector as VB
import qualified Data.Vector.Storable as VS
import qualified Data.Vector.Unboxed as VU
import UnliftIO.Exception (pureTry)

main :: IO ()
main = do
  print $ pureTry $ VB.head $ VB.fromList (():undefined)
  print $ pureTry $ VS.head $ VS.fromList (():undefined)
  print $ pureTry $ VU.head $ VU.fromList (():undefined)

  print $ pureTry $ VB.head $ VB.fromList [(), undefined]
  print $ pureTry $ VS.head $ VS.fromList [(), undefined]
  print $ pureTry $ VU.head $ VU.fromList [(), undefined]
```

- Boxed: spine strict
- Storable and unboxed: value strict

Question Why does this difference exist?

vector-algorithms

Exercise Fill a vector with 100 random integers between 1 and 10000 and sort it. Use vector-algorithms.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.Vector.Unboxed as V
import Data.Vector.Algorithms.Insertion (sort)
import System.Random (randomRIO)

main :: IO ()
main = do
  v <- V.replicateM 100 $ randomRIO (1, 10000 :: Int)
  print $ V.modify sort v
```

Or

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.Vector.Unboxed as V
import qualified Data.Vector.Unboxed.Mutable as VM
import Data.Vector.Algorithms.Insertion (sort)
import System.Random

main :: IO ()
main = do
  gen0 <- getStdGen
  print $ V.create $ do
    mv <- VM.new 100
    let loop gen idx
          | idx >= 100 = return ()
          | otherwise = do
              let (x, gen') = randomR (1, 10000) gen
              VM.write mv idx (x :: Int)
              loop gen' (idx + 1)
    loop gen0 0
    sort mv
    return mv
```

Or

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.Vector.Unboxed as V
import qualified Data.Vector.Unboxed.Mutable as VM
import Data.Vector.Algorithms.Insertion (sort)
import System.Random
import Data.Foldable (forM_)

main :: IO ()
main = do
  mv <- VM.new 100
  forM_ [0..99] $ \idx -> do
    x <- randomRIO (1, 10000)
    VM.write mv idx (x :: Int)
  sort mv
  v <- V.unsafeFreeze mv
  print v
```

Recommendations

There's some confusion about which data structure to use among the different kinds of vectors and lists. I typically advise using vectors over list. If so, why are lists so ubiquitous?

- The Prelude encourages them
- They're in base
- There's built-in syntax for them
- Sometimes they *are* better than vectors, such as using them as a control structure instead of for data storage

Here's a checklist I follow for choosing a data structure:

- Unless you have a good reason to do otherwise: use an immutable structure
- If unboxed is possible, use it
- Otherwise, if storable is possible, use it
- Otherwise, use boxed
- Generic algorithm? Use `Generic`
- Polymorphic container? Stick with boxed

Exercises

Test the randomness of `System.Random`: use `randomRIO (0, 9)` repeatedly to generate a random values and see if the distribution i close to uniform. First use immutable vectors:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import           Data.Vector.Unboxed ((!), (//))
import qualified Data.Vector.Unboxed as V
import           System.Random       (randomRIO)

main :: IO ()
main = do
    let v0 = V.replicate 10 (0 :: Int)

        loop v 0 = return v
        loop v rest = do
            i <- randomRIO (0, 9)
            let oldCount = v ! i
                v' = v // [(i, oldCount + 1)]
            loop v' (rest - 1)

    vector <- loop v0 (10^6)
    print vector
```

Question Is this efficient?

Now use mutable vectors:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import           Control.Monad               (replicateM_)
import           Data.Vector.Unboxed         (freeze)
import qualified Data.Vector.Unboxed.Mutable as V
import           System.Random               (randomRIO)

main :: IO ()
main = do
    vector <- V.replicate 10 (0 :: Int)

    replicateM_ (10^6) $ do
        i <- randomRIO (0, 9)
        oldCount <- V.read vector i
        V.write vector i (oldCount + 1)

    ivector <- freeze vector
    print ivector
```

Calculate the frequency of each byte (0-255) for the content coming from standard input.

```haskell
{-# LANGUAGE FlexibleContexts #-}
import           Control.Monad.Primitive     (PrimMonad, PrimState)
import qualified Data.ByteString.Lazy        as L
import qualified Data.Vector.Generic.Mutable as M
import qualified Data.Vector.Unboxed         as U
import           Data.Word                   (Word8)

main :: IO ()
main = do
    -- Get all of the contents from stdin
    lbs <- L.getContents

    -- Create a new 256-size mutable vector
    -- Fill the vector with zeros
    mutable <- M.replicate 256 0

    -- Add all of the bytes from stdin
    addBytes mutable lbs

    -- Freeze to get an immutable version
    vector <- U.unsafeFreeze mutable

    -- Print the frequency of each byte
    -- In newer vectors: we can use imapM_
    U.zipWithM_ printFreq (U.enumFromTo 0 255) vector

addBytes :: (PrimMonad m, M.MVector v Int)
         => v (PrimState m) Int
         -> L.ByteString
         -> m ()
addBytes v lbs = mapM_ (addByte v) (L.unpack lbs)

addByte :: (PrimMonad m, M.MVector v Int)
        => v (PrimState m) Int
        -> Word8
        -> m ()
addByte v w = do
    -- Read out the old count value
    oldCount <- M.read v index
    -- Write back the updated count value
    M.write v index (oldCount + 1)
  where
    -- Indices in vectors are always Ints. Our bytes come in as Word8, so we
    -- need to convert them.
    index :: Int
    index = fromIntegral w

printFreq :: Int -> Int -> IO ()
printFreq index count = putStrLn $ concat
    [ "Frequency of byte "
    , show index
    , ": "
    , show count
```

```
    ]
```

Tutorial exercise above

Now try taking a crack at the tutorial exercise we mentioned at the top. Some advice:

- Use `mwc-random` package
    - Not a recommendation for random packages, just a good way to practice vectors
- May want to consider: vector-th-unbox
- Note: that won't provide the tightest representation!
- Hard core: write an `Unbox` instance by hand
- Less hard core (what I'd probably do): can you use `GeneralizedNewtypeDeriving`?