Random numbers in Haskell

Computers are *deterministic*: Given a certain state of things, they produce a consistent result.

To have the appearance of "randomness", computers often implement what is known as a **pseudo-random number generator**. This is a *deterministic* function that given one value produces a new value, in such a way that those values appear to be unrelated to each other, and uniformly distributed across the range of values (so that each number is equally likely to show up).

BUT this is still a deterministic function: We provide it with an initial number "seed", and it will produce a consistent set of "random-looking" numbers, but always the same set for the same seed. This helps us write tests for code that uses the random number generator.

To use this in practice, we need to create an initial "seed" that is sufficiently random. This is often done by for example looking at the computer's exact clock time, or something like that.

In Haskell the System.Random module has various definitions related to random number generations. In particular, it has the type StdGen for "standard generator". We can create one with an initial seed like so:

```
import System.Random

let gen = mkStdGen 5      -- You can pick any number
```

To use such a generator, we can ask it to give us one or more values from a particular range, with something like this:

```
take 50 $ randomRs (0, 10) gen
```

Try this out and notice that each time you run this function you get the same set of fairly random numbers. If you change the definition of gen to use a different initial seed, you'll get a different set of numbers:

```
take 50 $ randomRs (0, 10) $ mkStdGen 2
```

We typically use the generator in a slightly different way, with the methods provided by the Random class. These methods typically return both a result *and* a new generator:

```
-- randomR :: RandomGen g => (a, a) -> g -> (a, g)
let (a, gen2) = randomR (0, 100) gen    --  a is in range [0..100]
let (b,    _) = randomR (0, 100) gen    --  b will be the same as a
let (b, gen3) = randomR (0, 100) gen2   --  b will be possibly different
```

So to keep generating random numbers, we need to keep updating our generator through the process. As an example, let's write a function that picks a specific set of random numbers from a given range, and puts them in a list:

```
getMany :: RandomGen g =>  (a, a) -> Int -> g -> ([a], g)
let getMany range n gen
        | n == 0        = ([], gen)
        | otherwise     = (x:xs, gen'')
```

```
    where (x, gen')    = randomR range gen
          (xs, gen'')  = getMany range (n-1) gen'
```

Practice: Run this on the generators gen and gen2 we defined earlier and notice the relation between the two.

Practice:

1. Write a function that is given a generator and returns a lowercase letter at random. The function randomR can actually accept characters rather than integers as the delimiters.
2. Write a function that is given a generator and a length and returns a random string of lowercase letters of that length.
3. Write a function that is given a generator and returns a random string of lowercase letters, of a random length between 0 and 50.

In order to produce truly random numbers, we have to do it within the IO system. There is in fact a function that will give us the standard generator in an IO setup:

```
getStdGen :: IO StdGen
```

We would normally have to worry about updating that generator, and there is a provided method for us to do that. So we could do something like:

```
getManyIO :: (a, a) -> Int -> IO [a]
getManyIO range n = do
    gen <- getStdGen                          -- Get the current generator
    let (xs, gen') = getMany range n gen      -- Generate our values
    setStdGen gen'                            -- Update the generator
    return xs                                 -- Return our values
```

In fact there is something better: There is a method that takes as input the kind of function we created via getMany and makes it work with the IO standard generator, essentially automatically doing the above steps:

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

For example we can implement a version of getMany that works with this generator, as follows:

```
getManyIO :: (a, a) -> Int -> IO [a]
getManyIO range n = getStdRandom $ getMany range n
```

Try it out with:

```
-- Each gives a different set of values!
getManyIO (0, 6) 10
getManyIO (0, 6) 10
getManyIO (0, 6) 10
```

In this way we have both a testable version (without the IO) as well as a real one (with the IO).

Shuffling a list

We will now discuss how to implement a list shuffle in Haskell. One key output from such a shuffle is that any of the possible arrangements should all be equally likely. For example if we were shuffling the list [1,2,3] we would want to have an equal chance of getting any of the 6 possible orderings.

A somewhat inefficient idea, which will be good enough for what we want to do, is this:

- In order to shuffle a list with n elements, we will start by generating n random numbers r1, r2, ..., rn, with some very specific properties.
- The first number, r1, is in the range from 0 to n−1. It is supposed to tell us which of the elements will be chosen first. For example if we were shuffling the list of characters "ABC" and r1=1 then the first element in the resulting shuffled list would be the 'B'.
- The second number, r2 is in the range from 0 to n−2. It is supposed to tell us which of the *remaining* elements will be chosen next.
- We continue in this way, each subsequent number telling us which of the remaining elements to choose next. Notice that the very last number, rn has no choice: It must be 0 as at that point there is only one element left to choose.

As an example of this, consider the list of characters "ABC". Then these are the possible numbers and corresponding orders:

```
[0, 0, 0]     --->    "ABC"          keeps order
[0, 1, 0]     --->    "ACB"
[1, 0, 0]     --->    "BAC"
[1, 1, 0]     --->    "BCA"
[2, 0, 0]     --->    "CAB"
[2, 1, 0]     --->    "CBA"          reverses
```

Therefore we can roughly break down the problem of shuffling a list into a few functions, which you will be asked to implement in your assignment:

```
getRs :: RandomGen g => Int -> g -> ([Int], g)
-- Example call: getRs 3 gen   returns one of the six arrangements
--                             above, plus a new generator

pluck :: Int -> [a] -> (a, [a])
-- Example call: pluck 1 "ABC" = ('B', "AC")

shuffle :: [Int] -> [a] -> [a]
-- Example call: shuffle [1,1,0] "ABC" = "BCA"

shuffleGen :: RandomGen g => [a] -> g -> ([a], g)

shuffleIO :: [a] -> IO [a]
```

Some details on these functions:

- getRs produces the list of r1, r2, ..., rn that we described earlier. It will use randomR as well as a recursive call. Make sure to update the generator through each step, and return the newest version along with the list of numbers.
- pluck is simply a list-manipulation function: It is given an index into the list, and is supposed to select the element at that index, remove it from the list, and return both the element and the updated list. You won't have to worry about the case where the index is not valid for the list (at least not unless you didn't implement getRs correctly).
- shuffle is given this list of integers rs and the list elements xs, and returns the shuffled list. It will require a recursive call using pluck along the way.
- shuffleGen combines getRs and shuffle, to get the r1,r2 ,..., rn from getRs and then use them with shuffle. Make sure that you return the updated generator that getRs gives you.
- shuffleIO uses getStdRandom together with shuffleGen to produce a shuffle using the IO generator, similarly to what our getManyIO did a few steps above.