Exercises 2

Please, write your solutions on a new Haskell file. Rename it to Ex2.hs and include a comment at the beginning of the file filling in the following information:

```
-------------------------------------------------------------------------------
-- Data Structures. ETSI Informática. UMA
--
-- (please, fill in the following form)
-- Degree: Grado en Ingeniería .......................... [Informática | del Software | de Computadores].
-- Student: SURNAME, NAME
-- Date:  DAY | MONTH | YEAR
--
-- Exercises 2. I solved the following exercises: .........
--
-------------------------------------------------------------------------------

import Test.QuickCheck
```

1. Define a function `allDifferent :: Eq a => [a] -> Bool` to test whether all elements in a list are different. For instance:

```
allDifferent [1,7,3] => True            allDifferent [1,7,3,1] => False
```

2. Predefined function `replicate :: Int -> a -> [a]` takes a natural number n and a value x and returns a list with n repetitions of x.

a) Define your own version of this function (call it `replicate'`) by using list comprenhensions:

```
replicate' 3 0 => [0,0,0]              replicate' 4 'a' => "aaaa"
```

b) Read and understand the following property on `replicate'`:

```
p_replicate' n x =  n >= 0 && n <= 1000 ==>
                         length (filter (==x) xs) == n
                         && length (filter (/=x) xs) == 0
                           where xs = replicate' n x
```

c) Test this property using *QuickCheck*.

3. Define using list comprenhensions a function called `divisors` returning all naturals divisors of a natural number:

```
divisors 10 => [1,2,5,10]
```

Define another function returning positive and negative divisors for an integer number:

```
divisors' (-10) => [-10,-5,-2,-1,1,2,5,10]
```

4. Greatest common divisor of two numbers *x* and *y*, denoted by *gcdiv x y*, is the maximum from the set comprising common divisors for *x* and *y*. Recall that greatest common divisor of two numbers is only defined if *x* and *y* are not simultaneously zero.

a) Define, using a list comprenhension, a function `gcdiv` returning the greatest common divisor of two numbers. For instance:

```
gcdiv 30 75 => 15
```

You will have to return the maximum element of the list that includes common divisors for both numbers using the following predefined function:

```
maximum :: (Ord a) => [a] -> a
```

b) Define and test using *QuickCheck* the following property: for *x*,*y*,*z* > 0, *gcdiv* of *z**x* and *z**y* is equal to *z* multiplied by *gcdiv* of *x* and *y*.

c) By using this property that relates *gcdiv* and least common multiple (*lcmul*)

gcdiv x y · lcmul x y = x · y

define a function to compute *lcmul* of two numbers:

```
lcmul 9 15 => 45                lcmul 30 75 => 150
```

Remark: Euclides' algorithm is more efficient than the one you have developed in this exercise.

5. Prime numbers

a) A prime number is a natural number with exactly two different positive divisors: *1* and *p*; hence, 1 is not a prime number. Define a function `isPrime` to check whether a number is prime:

```
isPrime 7 => True                isPrime 10 => False
```

b) Define, using list comprenhensions, a function `primesUpto` returning a list with all primes numbers whose value is less than or equal to function argument:

```
primesUpto 50 => [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

c) Provide a different definition (call it `primesUpto'`) by using predefined function `filter` instead of a list comprenhension.

d) Test that both functions return same result with *QuickCheck* by using the following property:

```
p1_primes x = primesUpto x == primesUpto' x
```

Remark: *Eratosthenes Sieve* is a more efficient algorithm for this problem.

6. Predefined function `zip` takes two lists and returns a list of corresponding pairs. If one input list is shorter, excess elements of the longer list are discarded:

```
zip [1,2,3] [True,True,False] => [(1,True),(2,True),(3,False)]
zip [1,2] [True,True,False] => [(1,True),(2,True)]
```

Predefined function `take` takes a natural number `n` and a list `xs` and returns a list with xs first n elements:

```
take 2 [7,3,1,2] => [7,3]
```

a) Fill in the following definition for this function (note that you cannot use the same name as the predefined function):

```
take' :: Int -> [a] -> [a]
take' n xs = [ ???  | (p,x) <- zip [0.. ??? ] xs, ???  ]
```

so that `take'` computes same values as predefined function `take`:

```
take' 3 [0,1,2,3,4,5] => [0,1,2]
take' 0 [0,1,2,3,4,5] => []
take' 5 [0,1,2]       => [0,1,2]
```

b) Predefined function `drop` takes a natural number n and a list xs and returns a list like xs but without its first n elements. Fill in the following definition:

```
drop' :: Int -> [a] -> [a]
drop' n xs = [ ??? | (p,x) <- zip [ ??? ] xs,  ??? ]
```

so that `drop'` computes same values as predefined function `drop`:

```
drop' 3 [0,1,2,3,4,5] => [3,4,5]
drop' 0 [0,1,2,3,4,5] => [0,1,2,3,4,5]
drop' 5 [0,1,2]       => []
```

c) Define and test using *QuickCheck* the following property: for n≥0 and any list `xs`, the concatenation of `take' n xs` and `drop' n xs` equals `xs`.

7. Predefined function

```
concat :: [[a]] -> [a]
```

takes a list of list and return their concatenation.

a) Define your own version of this function (name it `concat'`) by using `foldr`:

```
concat' [ [1,2,3], [5,6], [8,0,1,2] ] => [1,2,3,5,6,8,0,1,2]
```

Hint: notice that the result corresponds to the evaluation of this expression:

```
[1,2,3] ++ ( [5,6] ++ ( [8,0,1,2] ++ [] ) )
```

where (++) concatenates two lists.

b) Define a new version of `concat` (call it `concat''`) by using lists comprenhensions. Use two generators, so that the first one extracts each list and the second one extracts each element from each list returned by the first one.

8. Predefined functions `and`, `tail` and `zip` have been already introduced. Read and understand what the following function computes:

```
unknown :: (Ord a) => [a] -> Bool
unknown xs = and [ x<=y | (x,y) <- zip xs (tail xs) ]
```

9. Predefined function

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

returns the longest prefix with elements in a list (2nd argument) fulfilling a predicate (1st argument):

```
takeWhile even [2,4,6,8,11,13,16,20] => [2,4,6,8]
```

because all elements in the list before 11 are even numbers. Another example is this:

```
takeWhile (<5) [2,4,6,1] => [2,4]
```

because 6 is the first element in the list greater than or equal to 5. For same arguments, function `dropWhile` eliminates the preffix that `takeWhile` returns:

```
dropWhile even [2,4,6,8,11,13,16,20] => [11,13,16,20]
dropWhile (<5) [2,4,6,1] => [6,1]
```

a) By using these functions, define a function `insert` taking an element *x* and an already ascending sorted list *xs* (assume that this precondition holds), and returning the sorted list obtained by inserting *x* in its corresponding position into *xs*. For instance:

```
insert 5 [1,2,4,7,8,11] => [1,2,4,5,7,8,11]
insert 2 [1,2,4,7,8,11] => [1,2,2,4,7,8,11]
insert 0 [1,2,4,7,8,11] => [0,1,2,4,7,8,11]
insert 20 [1,2,4,7,8,11] => [1,2,4,7,8,11,20]
```

b) Understand and test using *QuickCheck* the following property for `insert`:

```
p1_insert x xs = unknown xs ==> unknown (insert x xs)
```

where unknown is the function defined in previous exercise.

c) We can use function `insert` to sort a list in ascending order. For instance, in order to sort list `[9,3,7]`, we can evaluate the following expression:

```
9 `insert` (3 `insert` (7 `insert` []))
```

Explain why this algorithm sorts a list

Remark: this sorting algorithm is known as insertion sort.

d) By using functions `foldr` and `insert`, define a function `isort` taking a list of values and returning that list sorted in ascending order. For instance:

```
isort [9,3,7] => [3,7,9]      isort "abracadabra"=> "aaaaabbcdrr"
```

e) Define and test using *QuickCheck* the following property: for any list `xs`, `isort xs` is a sorted list.

f) Estimate efficiency for `isort`.

10. Predefined higher order function

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

takes a function `f` and a value `x`, and returns an infinite list as follows:

```
iterate f x => [x, f x, f (f x), f (f (f x))), f (f (f (f x)))), ... ]
```

i.e., first element is `x` and remaining ones are obtained by applying function `f` to previous one. By using `iterate`, we can define arithmetic sequences if function `f` adds a fixed value. For instance:

```
iterate (+1) 0 => [ 0, 1, 2, 3, 4 ...
iterate (+2) 1 => [ 1, 3, 5, 7, 9 ...
```

a) In mathematics, a geometric progression, also known as a geometric sequence, is a sequence of numbers where each term after the first one is found by multiplying the previous one by a fixed non-zero number called the common ratio. Define using `iterate` a function named `geometric` taking an initial value and a common ratio and returning corresponding geometric sequence:

```
geometric 1 2  => [ 1, 2, 4, 8, 16 ...
geometric 10 3 => [ 10, 30, 90, 270,...
```

b) What does the following property check?

```
p1_geometric x r = x>0 && r>0 ==>
                        and [ div z y == r | (y,z) <- zip xs (tail xs) ]
                             where xs = take 100 (geometric x r)
```

c) By using `iterate`, define a function `multiplesOf` returning a list with the multiples of its arguments. For instance:

```
multiplesOf 2 => [ 0, 2, 4, 6, 8, 10 ...
multiplesOf 3  => [ 0, 3, 6, 9, 12, 15 ...
```

d) By using `iterate`, define a function `powersOf` returning a list with powers of its arguments. For instance:

```
powersOf 2 => [ 1, 2, 4, 8, 16, 32 ...
powersOf 3 => [ 1, 3, 9, 27, 81, ...
```

11. Recall that predefined concatenation operator is recursively defined as follows:

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
```

a) Define a property p_rightUnit to test using *QuickCheck* that empty list is right unit for concatenation:

```
xs ++ [] = xs
```

Test this property on lists of integers.

b) By using the definition of `(++)`, prove by induction on lists that empty list is right unit for concatenation. For this purpose, prove the following base and inductive cases:

(Base case) `[] ++ [] = []`

(Induction step) If `xs ++ [] = xs` then `(x:xs) ++ [] = (x:xs)`

12. In this exercise we will study efficiency, the associative property and associativity of the operator for list concatenation.

a) By using recursive definition in previous exercise, calculate step by step the result of expression `[1,2,3]++[4,5,6,7,8,9]`.

b) Let `lx` be the length of list `xs`, and let `ly` be the length of `ys`. How many evaluation steps does it take to compute `xs++ys`?

c) Define a property `p_associative` to test with *QuickCheck* that concatenation operator is associative.

d) Prove property:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

using induction on lists:

(Base case) `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

(Inductive step) If `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
then `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

e) Although we have proved in previous section that results for `(xs ++ ys) ++ zs` and `xs ++ (ys ++ zs)` coincide, one of those computations is more efficient. Let `lx`, `ly` and `lz` be the lengths of lists `xs`, `ys` and `zs`. Using solution to section b), find out how many computation steps does it take to calculate `(xs ++ ys) ++ zs` and `xs ++ (ys ++ zs)`.

f) Do you now understand why `(++)` is predefined as a right associative operator?

13. Function `nub` on `List` library removes repeated elements in a list. For instance:

```
nub [1,3,1,2,7,2,9] => [1,3,2,7,9]
```

a) Define your own version of this function (call it `nub'` and overload it for types with equality).

b) Test your function on list of integers using *QuickCheck* and this property:

```
p_nub' xs = nub xs == nub' xs
```

Don't forget to import `List` library.

c) Let us consider the following function to test correctness of `nub'`:

```
p_necessary xs = allDifferent (nub' xs)
```

Why is this property incomplete?, i.e, why is it a necessary condition but not a sufficient one to test `nub'` correctness.

d) Predefined function `all` tests whether a predicate holds for all elements in a list:

```
all (>10)   [100,50,20]  => True
all (=='0') "0000"       => True
all even    [1,2,3,4]    => False
```

Predefined function `elem` tests whether a value is an element of a list:

```
5   `elem` [1,2,5,9] => True
'1' `elem` "0000"    => False
```

Let us consider the following (non-predefined) function:

```
allIn :: (Eq a) => [a] -> [a] -> Bool
ys `allIn` xs = all (`elem` xs) ys
```

to test whether all elements in list `ys` are elements of list `xs`. For instance:

```
"011001"  `allIn` "01" => True
"01A1001" `allIn` "01" => False
```

Define by using function `allIn` a property `p_noRepetition` to properly test correctness for function `nub'`.

14. Define a function `bin` taking a natural number n and returning a list of strings corresponding to all binary numbers of length n:

```
bin 0 => [""]
bin 1 => ["0","1"]
bin 2 => ["00","01","10","11"]
```

15. Let xs be a list of different values. Variations with repetition for list xs with n element drawn in groups of size m are all list with length m that can be obtained by combining elements on list xs, (elements in a variation can be repeated).

a) Define a recursive function `varRep` taking a natural number m and a list xs and returning corresponding:

```
varRep 0 "abc" => [""]
varRep 1 "abc" => ["a","b","c"]
varRep 2 "abc" => ["aa","ab","ac","ba","bb","bc","ca","cb","cc"]
```

Hint: solution should be similar to function bin in previous exercise, but you should use a generator in a list comprenhension to select each of the n elements.

b) There exists n^m variations with repetition for n elements drawn in groups of size m. Understand and test for lists of characters and integers using *QuickCheck* the following property:

```
p_varRep m xs = m>=0 && m<=5 && n<=5 && allDifferent xs ==>
                    len vss == n^m
                    && allDifferent vss
                    && all (`allIn` xs) vss
      where
       vss = varRep m xs
       n = len xs
       len :: [a] -> Integer
       len xs = fromIntegral (length xs)
```

16. Using accumulator parameters, define a recursive function `facts :: [Integer]` returning an infinite list with the factorials for each natural number, beginning with 0!. Use two accumulators, so that each recursive invocation will only need to do an addition and a multiplication. For instance:

```
take 10 facts => [1,1,2,6,24,120,720,5040,40320,362880]
```

Hint: notice that by keeping values for n! and (n+1), values for (n+1)! and (n+2) can be obtained by doing just an addition and a multiplication, because (n+1)! = (n+1) · n! .