CS 5035 (Fall 2016)

### Project 6. Modules (first attempt by Oct 17)

Based on chapter [7 of LYH](http://learnyouahaskell.com/modules). [Videos](https://sites.google.com/a/lclark.edu/drake/courses/pls/lesson-6-haskell-modules).

Write code to solve problem 4 from [here](https://www.shiftedup.com/2015/05/07/five-programming-problems-every-software-engineer-should-be-able-to-solve-in-less-than-1-hour).

Given a list of non-negative integers, arrange and concatenate them so that they form the largest possible number. For example, given [50, 2, 1, 9], the largest possible number is 95021. Or, given [5, 54, 56] the possible number is 56554. (Getting this case right is the key.)

Your code should look something like the following.

import Data.List

import Data.Ord

largestNumber :: [Int] -> Integer

largestNumber = read . concat . map show . sortBy myCompare

myCompare :: Int -> Int -> Ordering

-- The following inverts the usual ordering. (This is discussed below.)

myCompare = comparing Down

------------------------------------

> largestNumber [1, 2, 3, 4, 5, 4, 3, 2, 1]

544332211

For this input, myCompare produces the right answer. But in general this myCompare function doesn’t do the job.

> largestNumber [50, 2, 1, 9]

50921 -- should be 95021

> largestNumber [5, 54, 56]

56545 -- should be 56554

So your primary challenge is to write a better myCompare function. *(Don’t solve this problem simply by trying all permutations!)*

The rest of this discussion talks about how Haskell does sorting.

Use sortBy (from Data.List) in your solution. sortBy has this type.

sortBy :: (a -> a -> Ordering) -> [a] -> [a]

The first argument is the comparison function sortBy uses to compare elements. In our case its type is.

Int -> Int -> Ordering

What comparison function should sortBy use in this problem?

sortBy is designed to order elements from smallest to largest. If compare x y = LT then x is placed to the left of y. In this problem that is counter-intuitive. We want the *larger* element to the left. Suppose your comparison function is called myCompare. It will have to find the following.

For the first example: [50, 2, 1, 9], the correct ordering is: [9, 50, 2, 1].

myCompare 9 50 = myCompare 9 2 = myCompare 9 1 = LT  
myCompare 50 2 = myCompare 50 1 = LT  
myCompare 2 1 = LT

#### In the second example: [5, 54, 56], the correct ordering is [56, 5, 54].

myCompare 56 5 = myCompare 56 54 = LT  
myCompare 5 54 = LT

The answer to this problem requires a bit of ingenuity. It is not complex, but it may not be immediately obvious. How do you get myCompare to do the right thing?

The first half of [this page](https://ro-che.info/articles/2016-04-02-descending-sort-haskell) may be useful.

**Here’s a mini-tutorial on sorting**

I want you to read and understand it. (I may ask you about it.) It is important because it illustrates how one can compute by manipulating functions and not just by performing low level operations. It’s good for your brain to work through this.

1. The Ordering class consists of the three values LT, EQ, and GT. This may seem like a strange pace to start, but start here.
2. A type belongs to the Ord class if any pair of values can be compared to get an Ordering value. For example, Int, Integer, and Char all belong to the Ord class. Even Bool belongs to Ord. In particular, False is less than True.
3. The compare function maps pairs of values of an Ord type to an Ordering value.

> :t compare

compare :: Ord a => a -> a -> Ordering

> compare True False

GT

> compare 2 3

LT

1. The function sort sorts elements according to their ordering.

> :t sort

sort :: Ord a => [a] -> [a]

> sort [4, 2, 7, 5]

[2,4,5,7]

1. The function sortBy sorts elements according to an ordering defined by a user function.

> :t sortBy

sortBy :: (a -> a -> Ordering) -> [a] -> [a]

-- In this example we will reverse the natural ordering.

> sortBy (\a b -> compare b a) [4, 2, 7, 5]

[7,5,4,2]

1. The function comparing (from Data.Ord) allows you to select (or compute) a feature of elements and compare on that feature.

> :t comparing

comparing :: Ord a => (b -> a) -> b -> b -> Ordering

-- We can compare two elements by looking at their squares,   
-- (-5)2 is greater than 42.

> comparing (^2) (-5) 4

GT

-- It is defined this way.

[comparing](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html#comparing) [p](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html" \l "local-1627439095) [x](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html" \l "local-1627439096) [y](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html" \l "local-1627439097) = compare ([p](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html#local-1627439095) [x](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html#local-1627439096)) ([p](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html#local-1627439095) [y](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html#local-1627439097))

-- So:

> sortBy (comparing (^2)) [4, -2, -7, 5]

[-2,4,5,-7]

-- That’s the same as

> sortBy (\a b -> compare (a^2) (b^2)) [4, -2, -7, 5]

[-2,4,5,-7]

1. The constructor Down (also from Data.Ord) takes an element and simply wraps it in Down.

> Down 2

Down 2

1. Down is useful because the function compare recognizes things wrapped in Down and reverses the usual order of comparison.

compare ([Down](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html#Down) [x](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html" \l "local-1627439099)) ([Down](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html#Down) [y](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html" \l "local-1627439100)) = compare y [x](https://hackage.haskell.org/package/base-4.9.0.0/docs/src/Data.Ord.html#local-1627439099)

> sortBy (comparing Down) [4, -2, -7, 5]

[5,4,-2,-7]

-- That’s the same as

> sortBy (\a b -> compare (Down a) (Down b)) [4, -2, -7, 5]

[5,4,-2,-7]

-- Which is the same as

> sortBy (\a b -> compare b a) [4, -2, -7, 5]

[5,4,-2,-7]

This may seem a bit too much. But here’s an example from LYAH for comparing lists on their lengths.

> xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]

-- You must import Data.Function before using (on).

-- See below for (on).

> sortBy (compare `on` length) xs

[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]

-- This is the same as

> sortBy (comparing length) xs

[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]

-- Which is the same as

> sortBy (\a b -> compare (length a) (length b)) xs

[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]

Here is how on is defined. Apply the first function to the result of applying the second function to the last two arguments.

on :: (b -> b -> c) -> (a -> b) -> a -> a -> c

And here is an example: (+) `on` (^2) = \x y -> x^2 + y^2

> **import** Data.Function

> ((+) `on` (^2)) 3 4 -- = 3^2 + 4^2

25

We can define our own myOn as follows.

> myOn g f = \x y -> g (f x) (f y)

> myOn (+) (^2) 3 4

25

> ((+) `myOn` (^2)) 3 4

25