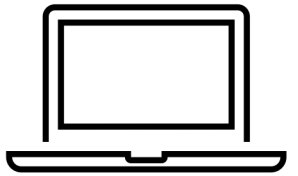
Week 3 exercises

Once again, this includes some exercises presented in class and additional exercises to work on in your own time.

Some problems are considerably more challenging that others.

Give it a go!



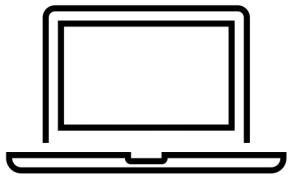
In the lecture this morning, we looked at a new definition for mergesort. (It is available on Blackboard).

1. What function would you pass as an argument to this function in order to sort a list of (name, grade) pairs by grade order, from lowest to highest? (Can you use a library function, or do you need to write your own function? If you need to write your own function, write it.)

   For example,

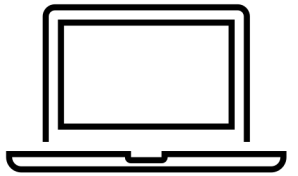
   - `> mergesort my_function [(“Sam”, 46), (“Bob”, 22), (“Alice”, 65), (“George”, 87), (“Jason”, 77)]`
   - `[(“Bob”, 22), (“Sam”, 46), (“Alice”, 65), (“Jason”, 77), (“George”, 87)]`

Give it a go!



2. Redefine map f and filter p using foldr.

Give it a go!



3. Define a function

```
scale :: Float -> Shape -> Shape
```

that takes a scaling factor and a shape, and returns the shape with the scaling factor applied. For example:

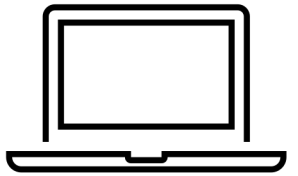
```
> scale 2.0 (Circle 1)
Circle 2.0
> scale 2.0 (Rect 1 2)
Rect 2.0 4.0
```

Note that in order to test your code, you will have to add “deriving Show” to the end of the type definition:

```
data Shape = Circle Float | Rect Float Float deriving Show
```

Don’t worry about why this is now; we will come to it in later weeks.

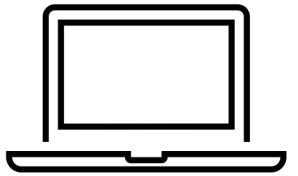
Give it a go!



4. Rewrite the following expressions as list comprehensions:

```
a) map (+3) xs
b) filter (>7) xs
c) concat (map (\x -> map (\y -> (x,y)) ys) xs)
d) filter (>3) (map (\(x,y) -> x+y) xys)
```

Give it a go!

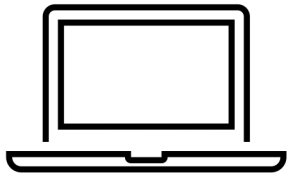


5. What does this function do:

```
mystery xs = foldr (++) [] (map sing xs)
                    where
                    sing x = [x]
```

Hint: Work through what it does for a simple list, e.g. [1,2,3]

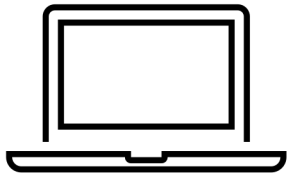
Give it a go!



6. Implement your own version of maximum using one of the fold functions.

(maximum takes a list as an argument and returns the maximum value in a list)

Give it a go!

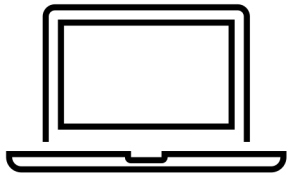


7. Without looking at the definitions from the standard prelude, define the higher-order library function curry that converts a function on pairs into a curried function, and, conversely, the function uncurry that converts a curried function with two arguments into a function on pairs.

   Hint: start with the type definitions.

(The solution is simple, but it might take some effort to get there…)

Give it a go!

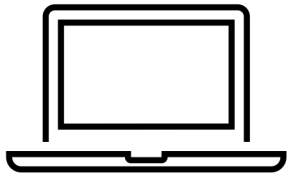


8. [Hutton, ex 7.9]
   Define a function `altMap :: (a->b) -> (a->b) -> [a] -> [b]` that alternately applies its two argument functions to successive elements in a list, in turn about order. For example,

```
> altMap (+10) (+100) [0,1,2,3,4]
[10,101,12,103,14]
```

9. [Hutton, ex 7.10] Using altMap, define a function luhn :: [Int] -> Bool that implements the *Luhn algorithm* from the Week 2 Exercises for bank card numbers of any length. Test your new function using your own bank card.

Give it a go!



10. A bag is a collection of items. Each item may occur one or more times in the bag. All the items are of the same type. The order of the items is unimportant. e.g.,
    - over a 40-game season a football team might score 5 goals in 2 games, 4 goals in 1 game, 3 goals in 4 games, 2 goals in 10 games, 1 goal in 12 games and 0 goals in 11 games;
    - a student's assessment profile might comprise 3 A grades, 4 B grades and 2 C grades.

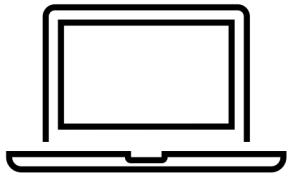
A bag may be implemented by a list (e.g. ['A','A','A','B','B','B,'B','C,'C'] for the student grades) but this is clearly inefficient.

In this task you will implement and test a Haskell module Bags.hs to handle bags. **Bags.hs** should contain

a) A definition for a polymorphic Haskell datatype **Bag** to represent bags efficiently.

b) The following functions
    - **i.** **listToBag** takes a list of items (such as ['A','A','A','B','B','B,'B','C,'C'] above) and returns a bag containing exactly those items; the number of occurrences of an item in the list and in the resulting bag is the same.
    - **ii.** **bagEqual** takes two bags and returns True if the two bags are equal (i.e., contain the same items and the same number of occurrences of each) and False otherwise.
    - **iii.** **bagInsert** takes an item and a bag and returns the bag with the item inserted; bag insertion either adds a single occurrence of a new item to the bag or increases the number of occurrences of an existing item by one.
    - **iv.** **bagSum** takes two bags and returns their bag sum; the sum of bags X and Y is a bag which contains all items that occur in X and Y; the number of occurrences of an item is the sum of the number of occurrences in X and Y.
    - **v.** **bagIntersection** takes two bags and returns their bag intersection; the intersection of bags X and Y is a bag which contains all items that occur in both X and Y; the number of occurrences of an item in the intersection is the number in X or in Y, whichever is less.

Give it a go!



11. Using `'.'`, `foldr`, `map` and the identity function `id`, write a function pipeline which given a list of functions, each of type `a -> a` will form a *pipeline* function of type `[a] -> [a]`. In such a pipeline, each function in the original function list is applied in turn to each element of the input (assume the functions are applied from right to left in this case). You can imagine this as being like a conveyor belt system in a factory where goods are assembled in a fixed number of processing steps as they pass down a conveyor belt. Each process performs a part of the assembly and passes the (partially completed) goods on to the next process.
    Test your function by forming a pipeline from the function list
    `[(+ 1), (* 2), pred]`
    with the resulting pipeline being applied to the input list `[1, 2, 3]`.
    Hint: Notice that if `f :: a -> a` then `(map f)` is a *function* of type `[a] -> [a]`.