QUESTION 1
Question about [a] vs [[a]] vs Maybe [a] vs [Maybe a] - which is most appropriate when? Eg Matt Giuca's LMS posting:

Let's assume we're writing a "sports team signup sheet" program, where a Team consists of a number of Players. Naturally, we would define it like this:

type Team = [Player]

There is no immediate reason to have a Maybe type either inside or outside of the list. If you instead made it [Maybe Player], you would have to be explicitly checking each entry to see if it's Nothing or Just (as an aside, note that in Java you would always have to check for null -- it's a key advantage of Haskell/Mercury that null isn't allowed unless you explicitly declare something Maybe). If you wanted to print out the list, you would have to delete all the non-Nothing entries. And [], [Nothing] and [Nothing, Nothing, Nothing] would represent the same team -- so why allow this redundant data structure. Similarly, if you made it a Maybe [Player], now an "empty" team could be represented as "Nothing" or "Just []", so you would have special cases for the empty team all over the place.

But there are some reasons to combine a List and a Maybe. Consider that this program now allows a Team to be created, but it can't have Players in it until they have paid their signup fee. Now maybe it makes sense to define it as:

type Team = Maybe [Player]

A Team of Nothing does exist, but hasn't paid its signup fee, so the player list is more than just empty -- it isn't there at all. A Team of Just [] has paid their signup fee, but hasn't enrolled any Players yet. The important thing is that both "Nothing" and "Just []" have a distinct meaning, so the Maybe List type is justified (though there are probably better ways to represent this).

Alternatively, consider that the program now allows "undecided" signups -- a team can nominate that they intend to put a player in a particular place, but haven't decided on a person yet. Now maybe it makes sense to define:

type Team = [Maybe Player]

The empty Team, [], actually has no players. The Team [Nothing, Nothing, Nothing] has three player spots nominated, but have not appointed any specific people there yet. Still, length will tell us the planned team size. This is a useful representation.

The key is to consider, for every valid value of this data type, a) does it mean something sensible, and b) is it the only way to represent that meaning in this data type. (a) is highly desirable, (b) is less desirable but still good. Again, they aren't always possible.

QUESTION 2
A question that shows code that uses a variable instead of a constructor in a pattern, leading to a bug. Ask students to find the bug.

QUESTION 3
Heap implementation.

QUESTION 4

Write definitions (as simple as possible) of the Haskell functions
mysum and myproduct, which return the sum and product of a list of
numbers, respectively. These definitions have a similar structure;
what other definitions you have seen have the same structure? Note:
later we will consider a "higher order function" which allows you to
write definitions of such functions much more concisely.
XXX do not use: will be discussed in lectures

QUESTION 5
Write a function called filter_map that does the jobs of filter and map
at the same time. The type of filter_map should be

```
filter_map :: (a -> Maybe b) -> [a] -> [b]
```

XXX do not use: will be discussed in lectures

QUESTION 6
Use standard higher order functions and operator sections to write
single line definitions of sum, product, all_pos, some_not_pos and length
(see question in previous tutorial). What are the advantages and
disadvantages of such definitions.
XXX do not use: will be discussed in lectures

QUESTION 7
Foldr takes a list and "folds" it into a single value, but that value
could be any type, including a list. Can you implement map and filter
using foldr?

QUESTION 8
Consider the task of converting a list of single element lists into
a sorted list by repeatedly merging lists. This is a fold operation.
What is the algorithmic complexity if we use "foldr merge []"? What if
we use foldl instead of foldr? Is the result the same, and why or why not?
What is the complexity? What if we use balanced_fold (defined in lectures)?

QUESTION 9
There are two improvements we can make to the definition of the balanced_fold
function given in lectures.

The balanced_fold function given in lectures computes the length of not just
the original list, but of every one of the lists it is divided into,
even though the code that divides a list knows (or should know) how long
the resulting lists should be. The first improvement is therefore avoiding
the redundant length computations, and computing the length of just one list:
the original list.

The second improvement is to avoid the intermediate lists being created
at each level of recursion, when the list is split in two. An alternative
is for the first recursive call to return both the fold of the first
half (or n elements) of the list and the remainder of the list (which
is then used in the second recursive call). For example, using the scenario
from the previous question, doing a merge using balanced folds on the list
of lists [[4],[1],[6],[2],[8],[7],[3],[5]], the first recursive call could
return the pair ([1,2,4,6], [[8],[7],[3],[5]]), and then pass the list
of lists [[8],[7],[3],[5]] to the second call.

Write two versions of balanced_fold. The first should have the first of these
improvements, the second should have both.

You might want also want to code merge sort (for example) in this style,
and then generalise it to balanced_fold.

QUESTION 10

Consider the following tree data type:

```
>data Tree a = Empty | Node (Tree a) a (Tree a)
```

Define a higher order function

```
>map_tree :: (a->a) -> Tree a -> Tree a
```

which is the analogue of map for trees (rather than lists): it applies a function to each element of the tree and produces a new tree containing the results. The result tree is the same shape as the input tree, just as the result list in map is the same length as the input list.

QUESTION 11
Given the Tree type above, write functions which take a Tree and compute

(a) the height,
(b) the number of nodes,
(c) the concatenation of the elements in the nodes (assuming that the values in tree nodes are in fact lists)
(d) the sum of the elements (assuming that values in the tree are Nums)
(e) the product of the elements in the nodes (assuming they are Nums), and
(f) Just the maximum of the elements in the nodes, or, Nothing if the tree is empty Nothing.

Before you start, it may be helpful to review the tree sort question from the workshop for week 4.

These operations can be seen as folds on Trees. When you get tired of writing the same pattern for traversing the Tree, write a higher order function

```
>foldr_tree :: (a->b->a->a) -> a -> Tree b -> a
```

which can be used to define the other functions. The first argument is a function which is applied at each Node, after the subtrees have been folded. The second argument is returned for Empty trees.

QUESTION 12

Write a Mercury program that takes at least one integer command line arguments, and prints out an arithmetic expression whose value is the first command line argument using all and only the command line arguments following the first exactly once, and using any of the operations of addition, subtraction, multiplication, and division. Calculations may use parentheses as necessary. All calculations should be integer-valued, so all divisions must work out to an integer. Eg, 7/3 would not be allowed, but 6/3 would. For example,

```
        ./mathjeopardy 24 12 6 3 2
```

might print

```
        12+6+3*2
```

and

```
        ./mathjeopardy 91 8 27 36 4 9 18
```

might print

```
        8+27+4*(18-36/9)
```

For starters, print the expression with parentheses everywhere. When you get that working, try to print as few parentheses as necessary.

(I'm calling this mathjeopardy because it's a bit like the game show Jeopardy: you are given the answer and have to come up with the question.)

QUESTION 13
Implement a Haskell class called Appendlist which supports the following operations:

empty:	creates an empty list
cons:	adds an element to the front of a list
append:	appends two lists together
is_empty:	checks if a list is empty, returning a Bool
de_cons:	returns a pair containing the head (the first element) and the tail (the rest of the list), aborting if the list is empty

Implement two instances: one for (normal) lists and one for cords as defined in lectures:

```
>data Cord a = Nil | Leaf a | Branch (Cord a) (Cord a)
```

QUESTION 14
In a language such as C we sometimes use doubly linked lists, where each cell has pointers to the next cell and also to the previous cell. This allows us to move one element left or right in the list, insert or delete an element to the left or right of the "current position", or "cursor". By thinking of the current position as being *between* two elements (or to the left of the leftmost element or to the right of the rightmost element), we can naturally support empty lists (which are a problem if we need a "current element"). How can we support the following operations in constant time in Haskell?

dll_new:	creates a new (empty) doubly linked list
dll_empty:	tests if the list is empty
dll_empty_l:	tests there are no elements to the left of the cursor
dll_empty_r:	tests there are no elements to the right of the cursor
dll_l:	the element on the left
dll_r:	the element on the right
dll_move_l:	move the cursor one element left
dll_move_r:	move the cursor one element right
dll_add_l:	adds an element to the left of the cursor
dll_add_r:	adds an element to the right of the cursor
dll_remove_l:	removes the element to the left of the cursor
dll_remove_r:	removes the element to the right of the cursor

In Mercury we can have several modes for each predicate. Which of the operations above could potentially be combined into a single predicate by using multiple modes? Give the code (including the determinism in the declarations) and some sample calls which illustrate its use in different modes. Is it always a good idea to combine the different modes? Why or why not?

QUESTION 15
Define versions of map for the following data types:

```
>data Ltree a = LTLeaf a | LTBranch (Ltree a) (Ltree a)
>data Cord a = Nil | Leaf a | Branch (Cord a) (Cord a)
>data Mtree a = Mnode a [Mtree a]
```

QUESTION 16

In "foldr f b" we essentially replace [] by b and : by f. For example,

```
        foldr f (+) 0 [1, 2]
        = foldr f (+) 0 (1:2:[])
        = foldr f (+) 0 ((:) 1 ((:) 2 []))
        = ((+) 1 ((+) 2 0))
        = 1 + (2 + 0)
        = 3.
```

Similarly, foldr_tree f b (from the previous workshop) replaces Empty by b and Node by f (which takes three arguments). Define versions of foldr in this style for the types in the previous question.

Note that GHC supports the Foldable typeclass (Data.Foldable), but this typeclass generalises foldr for lists in a different way, ignoring the structure and just using the elements within the data type. This form of foldr on a tree is equivalent to traversing the tree to get a list, then applying foldr to the list. Here we want more general functions which do not necessarily ignore the structure. For example, in the previous workshop, foldr_tree Node Empty is the identity function over Trees, which is not possible if we ignore the structure. Similarly, height_tree requires the tree structure.

QUESTION 17
Recall the following functions from lectures, for concatenating a list of lists and converting a cord into a list:

```
>concatr = foldr (++) []  -- Efficient
>concatl = foldl (++) []  -- Inefficient

>cord_to_list :: Cord a -> [a]  -- Inefficient
>cord_to_list Nil = []
>cord_to_list (Leaf x) = [x]
>cord_to_list (Branch a b) =
>    (cord_to_list a) ++ (cord_to_list b)

>cord_to_list2 :: Cord a -> [a]  -- Efficient
>cord_to_list2 c = cord_to_list' c []
>cord_to_list' :: Cord a -> [a] -> [a] -- Arg 2 is an accumulator
>cord_to_list' Nil rest = rest
>cord_to_list' (Leaf x) rest = x:rest
>cord_to_list' (Branch a b) rest =
>    cord_to_list' a (cord_to_list' b rest)
```

Recall the reason for the relative (in)efficiency is that ((a++b)++c) has to copy the elements and cons cells of the list `a' twice, whereas (a++(b++c)) only needs to do it once. The cost of ++ depends on the length of its first argument but not its second argument.

Define four versions of concat_rev, which concatenates the lists in reverse order (the inner lists are not reversed, for example concat_rev [[1,2],[3]] = [3,1,2]) using (1) foldr, (2) foldl, (3) recursion without an accumulator, and (4) recursion with an accumulator. Determine which versions are efficient.

How would you code cord_to_list_rev, which converts a cord to a list, but in the reverse order? Code it directly rather than creating an in-order list and then reversing it.

What is the relationship between foldr for cords and the code above and in what way is the code similar to both foldl and foldr?

QUESTION 18
The definition of the cord data type presented in lectures (see Q1 above) can represent the same nonempty sequence of items in more than one way. For example, the sequence [1, 2, 3] can be represented as

```
        Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
```

or as

```
        Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)
```

Without this flexibility, the operation to concatenate two cords couldn't be implemented in constant time. However, this type also has unnecessary flexibility. For example, it can represent the empty sequence in more than one way, including Nil, Branch Nil Nil and Branch Nil (Branch Nil Nil). Design a version of the cord data type that has only one way to represent the empty sequence.

QUESTION 19
Compare and contrast the following Haskell and Mercury code:

```
>append [] l = l
>append (j:k) l = j : append k l

>rev [] = []
>rev (a:bc) = append (rev bc) [a]
```

```
append([], L, L).
append([J | K], L, [J | KL]) :-
        append(K, L, KL).

rev([], []).
rev([A | BC], R) :-
        rev(BC, CB),
        append(CB, [A], R).
```

QUESTION 20
Write Haskell code which can check if one string (or, more generally, a list of elements in the Eq type class) is a substring (sublist) of another. The elements of the substring must occur next to each other in the bigger list, so that for example, "bcd" is a substring of "abcde", but not of "abcfde".

Note that there are some tricky cases. For example, when searching for "bcd" in "bcabcd", the first two characters of the substring match the initial part of the string but the next character (a) prevents a match at that point; nevertheless, there is a match later in the string.

Even more tricky are cases where you cannot skip characters that participated in an ultimately failed match. Consider looking for "bcbcd" in the string "bcbcbcd". When you look for a match starting at the first character, the first four characters match, while the fifth doesn't. However, there is a match starting at the third character of "bcbcbcd".

Your function could simply return a Boolean. What other information could it return?

How would your overall design differ in Mercury? Write a Mercury predicate that does the task in a manner appropriate for Mercury.

QUESTION 21
Write Haskell and Mercury implementations of queues, where a queue is represented as a list, the head of the list being the head of the queue. The code should support the following operations:

```
queue_new:       creates a new (empty) queue
queue_empty:     tests whether a queue is empty
```

queue_add: adds a new element to the back of the queue
queue_remove: returns a two-tuple containing (a) the element at the head of the queue and (b) the rest of the queue

What is the main disadvantage of this representation?
Would reversing the order of elements help?

QUESTION 22
Write Haskell and Mercury implementations of queues using a different representation: a pair of lists (lf, lr), where lf ++ (reverse lr) gives the single list representation above. Can this avoid the main disadvantage of the representation used in the previous question?