DM580 – EXERCISE SHEET #4

Solve the exercises below. Unless stated explicitly, you are not allowed to use list comprehensions or any of the built-in functions from the Haskell prelude or standard library.

(1) Write a recursive function `lastelem :: [a] -> a` that returns the last element of a list.

(2) Write a recursive function `haskey :: Eq a => [(a,b)] -> a -> Bool` that checks whether the given value is the first component in any of the entries in a given list of pair.

(3) Without using list comprehensions, write a recursive function `findall :: Eq a => [(a,b)] -> a -> [b]` that finds the second component of entries in a given list of pairs where the first component is the given value, for example

```
findall [(0,0),(0,1),(1,0),(1,2)] 0 == [0,1]
```

(a) *Bonus question:* Now write the same function using list comprehensions. Which one do you prefer?

(4) Write a recursive function `separate :: [(a,b)] -> ([a],[b])` that takes a list of pairs and returns a pair of lists where the first list consists of all of the first components, and the second list consists of all of the second components.

(5) Write a recursive function `euclid :: Int -> Int -> Int` that implement's *Euclid's algorithm* for computing the greatest common divisor of two integers: if the two integers are equal return that, otherwise subtract the smaller number from the larger one and repeat this process.

(6) Write a recursive function `merge :: (Ord a) => [a] -> [a] -> [a]` that merges two sorted lists into a single sorted list such that, e.g., `merge [1,3,6] [1,4,5] == [1,1,3,4,5,6]`.

(7) Use your recursive function `merge` to implement the *mergesort* algorithm: the empty list, and one element lists, are already sorted, and any other list is sorted by splitting it into two halves, sorting each half, and merging the results. To do this, you may need to write a helper function that accepts a list and splits it into two lists that differ in length by at most one.

(8) Recall that the *branches* of a function consist of all of the cases that can be reached by input data given to the function. When writing a test suite, we want to ensure that all branches are exercised by at least one test case; in this case, we say that the test suite *covers* (all branches of) the function.

Write a test suite that covers all branches of the function `insert` below (taken from Section 6.2 of *Programming in Haskell*).

```
insert :: Ord a => a -> [a] -> [a]
insert x []                 = [x]
insert x (y:ys) | x <= y    = x : y : ys
                | otherwise = y : insert x ys
```

(9) The *Hofstadter Male and Female Sequences* are given by the mutual recurrence relations M_n and F_n below.

$$F_0 = 1 \qquad\qquad M_0 = 0$$
$$F_n = n - M_{F_{n-1}} \qquad\qquad M_n = n - F_{M_{n-1}}$$

Define `male :: Int -> Int` and `female :: Int -> Int` that compute these two sequences. For example, it should be the case that

```
[female n | n <- [0..20]] == [1,1,2,2,3,3,4,5,5,6,6,7,8,8,9,9,10,11,11,12,13]
[male n | n <- [0..20]]   == [0,0,1,2,2,3,4,4,5,6,6,7,7,8,9,9,10,11,11,12,12]
```

(10) *Challenge:* You may use list comprehensions for this exercise. Use memoisation to improve the running time of your functions `male` and `female`. A properly memoised function will be able to compute `female 10000` in fractions of a second on a reasonably new computer.

(11) *Challenge:* Without using list comprehensions, write a recursive function

```
join :: Eq b => [(a,b)] -> [(b,c)]-> [(a,c)]
```

that takes two lists xs and ys and produces the list of every pair of the form (x, z) where there exists y such that (x, y) is in xs and (y, z) is in ys. For example, it should be the case that

```
join [(0,0),(0,1),(1,0)] [(0,1),(1,2),(2,3),(3,4)] == [(0,1),(0,2),(1,1)]
```

(a) *Bonus question:* Now write the same function using list comprehensions. Which one do you prefer?