COMP304 Programming Languages Assignment 1: Basic Haskell

David Thomsen, 300052209

1. Variations on a search

(a) A function that counts the number of times a particular item occurs in a list of items and returns an integer.

```
count :: (Eq a) => a -> [a] -> Int
count x [] = 0
count x (y : ys) = count' 0 x (y : ys)
count' c _ [] = c
count' c x (y : ys) = if x == y then count' (c+1) x ys
                      else count' (c) x ys
```

Tests for the 'count' function.

```
testCount1 = count 1 [] == 0
testCount2 = count 1 [1,2,1,2,1] == 3
testCount = and [testCount1, testCount2]
```

(b) A function that takes a given item and a list of items and returns an array of the indexes of all occurrences of the item.

```
allPos :: (Num b, Eq a) => a -> [a] -> [b]
allPos x [] = []
allPos x (y : ys) = allPos' [] 1 x (y : ys)
allPos' :: (Num b, Eq a) => [b] -> b -> a -> [a] -> [b]
allPos' l _ _ [] = l
allPos' l i x (y : ys) = if x == y then allPos' (l++[i]) (i+1) x ys
                          else allPos' l (i+1) x ys
```

Tests for the 'all positions' function.

```
testAllPos1 = allPos 1 [1,2,1,2,1] == [1,3,5]
testAllPos2 = allPos 3 [1,2,1] == []
testAllPos3 = allPos 1 [] == []
testAllPos = and [testAllPos1, testAllPos2, testAllPos3]
```

(c) A function that takes an given item and a list of items and returns a pair of the first and last locations of the item, or (0,0) otherwise. I split this into two sub-functions, the first returns the first index, the second returns the last index.

```
firstLastPos x [] = (0,0)
firstLastPos x (y : ys) = (firstLastPos' x 1 (y : ys),
                           firstLastPos'' 0 x 1 (y : ys))
firstLastPos' x i [] = 0
firstLastPos' x i (y : ys) = if x == y then i
                              else firstLastPos' x (i+1) ys
firstLastPos'' r x i [] = r
```

```haskell
firstLastPos'' r x i (y : ys) = if x == y then firstLastPos'' i x (i+1) ys
                                else firstLastPos'' r x (i+1) ys
```

Tests for the 'first and last positions' function.

```haskell
testFirstLastPos1 = firstLastPos 1 [1,2,1,2,1] == (1,5)
testFirstLastPos2 = firstLastPos 2 [1,2,1,2,1] == (2,4)
testFirstLastPos3 = firstLastPos 3 [1,2,1] == (0,0)
testFirstLastPos4 = firstLastPos 1 [] == (0,0)
testFirstLastPos = and [testFirstLastPos1, testFirstLastPos2,
                        testFirstLastPos3, testFirstLastPos4]
```

2. Sorting

(a) A selection sort algorithm. I chose this one because it sorts the list using forward propagation. My algorithm works but ended up being a lot less elegant than it possibly could have been, however this is a learning process. It iterates through the array to find the lowest value to put at the beginning, and again to find the index of the highest value. If I had had more time to work on this I would have tried to achieve both with the same iteration, although I believe this implementation is still n^2?

Another implementation I had strongly considered was bubble sort, however I was put off by the fact that I would have had to find some way to shorten the distance of the run through the array with each iteration, whereas with selection sort I could use head and tail constructor from (x:xs) to move forwards.

```haskell
sort1 :: (Ord a) => [a] -> [a]
sort1 [] = []
sort1 (y:ys) = (mini y (y:ys) : sort1 (swap (minDex 1 1 y (y:ys)) (y:ys)))

mini :: (Ord a) => a -> [a] -> a
mini x [] = x
mini x (y:ys) = if x < y then mini x ys
               else mini y ys

first :: (Ord a) => [a] -> a
first (y:ys) = y

minDex :: (Ord a) => Int -> Int -> a -> [a] -> Int
minDex i j x [] = j
minDex i j x (y:ys) = if x > y then minDex (i+1) i y ys
                     else minDex (i+1) j x ys

swap :: (Ord a) => Int -> [a] -> [a]
swap i (y:ys) = if i == 1 then ys
                else swap' i 2 y ys

swap' :: (Ord a) => Int -> Int -> a -> [a] -> [a]
```

```haskell
swap' i j x (y:ys) = if i == j then (x:ys)
                     else (y:swap' i (j+1) x ys)
```

Tests for selection sort algorithm

```haskell
testSort1a = sort1 [1,9,2,8,3,7,4,6,5] == [1,2,3,4,5,6,7,8,9]
testSort1b = sort1 [1,2,3,2,1] == [1,1,2,2,3]
testSort1 = and [testSort1a, testSort1b]
```

(b) A merge sort algorithm. I started with the merge method because it seemed a quick way to make progress. I decided that instead of splitting my arrays down the middle, I would split by alternating and putting the odd index values in one list and even indexes in the other list, which is the same thing for all intensive purpoises. I wasn't sure how to create a method that would return two lists with one iteration so I had to iterate over the list twice, once for odds and once for evens, which I believe is still n log n.

I decided not to do a quicksort algorithm because I felt it would be a lot more complicated, although I am sure there is a very efficient way to do it in Haskell that I haven't thought of. I guess I would start with (x:xs) and go through xs until I found a value less than x, and would swap the two. I would continue doing that process, starting again with xs, until I had reached the end of the array. I would begin the same process again with a list of values before the final index of x, and also a list of values after the x index, and glue them all back together with x in between.

```haskell
sort2 :: (Ord a) => [a] -> [a]
sort2 [] = []
sort2 (y:ys) = if ys == [] then [y]
               else merge (odds (y:ys)) (evens (y:ys))

odds :: (Ord a) => [a] -> [a]
odds l = sort1 (getOdds' True l)

evens :: (Ord a) => [a] -> [a]
evens l = sort1 (getOdds' False l)

getOdds' :: (Ord a) => Bool -> [a] -> [a]
getOdds' _ [] = []
getOdds' True (x:xs) = (x:getOdds' False xs)
getOdds' False (x:xs) = getOdds' True xs

merge :: (Ord a) => [a] -> [a] -> [a]
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys) = if x < y then (x:merge xs (y:ys))
                      else (y:merge ys (x:xs))
```

Tests for mergesort.

```
testSort2a = sort2 [1,9,2,8,3,7,4,6,5] == [1,2,3,4,5,6,7,8,9]
testSort2b = sort2 [1,2,3,2,1] == [1,1,2,2,3]
testSort2 = and [testSort2a, testSort2b]
```

3. Implementing a map

An implementation of a map. I found this section quite straightforward compared with previous sections.

```
type Map a b = [(a, b)]

emptyMap :: Map a b
emptyMap = []

hasKey :: (Eq a) => a -> Map a b -> Bool
hasKey a [] = False
hasKey a ((b,c):l) = if a == b then True
                     else hasKey a l

setVal :: (Eq a) => a -> b -> Map a b -> Map a b
setVal a b [] = [(a,b)]
setVal a b ((c,d):l) = if a == c then ((a,b):l)
                       else ((c,d):setVal a b l)

getVal :: (Eq a) => a -> Map a b -> b
getVal a [] = error "Key not present in Map"
getVal a ((b,c):l) = if a == b then c
                     else getVal a l

delKey :: (Eq a) => a -> Map a b -> Map a b
delKey a [] = error "Key not present in Map"
delKey a ((b,c):l) = if a == b then l
                     else ((b,c):delKey a l)
```

Tests for my map implementation. I have not tested for anything that should return an error as we have not covered that in the course. The first test, testMap1, I had difficult with because I could not write < testMap1 = emptyMap == [] without the compiler complaining, so I compromised and made sure I could add a value to a map returned by emptyMap.

```
testMap1 = setVal 1 'a' emptyMap == [(1,'a')]
testMap2 = hasKey 1 [] == False
testMap3 = hasKey 2 [(1,'a'), (2,'b')] == True
testMap4 = hasKey 1 [(2,'b')] == False
testMap5 = setVal 1 'b' [(1,'a'),(2,'b')] == [(1,'b'),(2,'b')]
testMap6 = setVal 3 'c' [(1,'a'),(2,'b')] == [(1,'a'),(2,'b'),(3,'c')]
testMap7 = getVal 2 [(1,'a'),(2,'b')] == 'b'
```

```
testMap8 = delKey 1 [(1,'a'),(2,'b')] == [(2,'b')]
testMap = and [testMap1, testMap2, testMap3, testMap4, testMap5, testMap6,
               testMap7, testMap8]
```

4. Building a map

This was the most straightforward section of all, having already learned how Haskell works and implemented a map.

```
buildMap :: (Eq a) => [a] -> Map a Int
buildMap l = buildMap' l emptyMap

buildMap' :: (Eq a) => [a] -> Map a Int -> Map a Int
buildMap' [] m = m
buildMap' (x:xs) m = if hasKey x m then
                        buildMap' xs (setVal x ((getVal x m)+1) m)
                     else buildMap' xs (setVal x 1 m)
```

Tests for map building function.

```
testBuildMap1 = buildMap [1,2,3,2,1] == [(1,2),(2,2),(3,1)]
testBuildMap2 = buildMap [1,2,3] == [(1,1),(2,1),(3,1)]
testBuildMap3 = buildMap [1,2,3,4,5,4,3,2,1,2,3,4,5] ==
                         [(1,2),(2,3),(3,3),(4,3),(5,2)]
testBuildMap = and [testBuildMap1, testBuildMap2, testBuildMap3]
```

Test for all tests.

```
testAll = and [testCount, testAllPos, testFirstLastPos, testSort1, testSort2,
               testMap, testBuildMap]
```