COMP 304 Assignment 1

Daniel Braithwaite

Part 1

Question A

Export our module so the tests can access it

```haskell
module COMP304A1 where
```

count returns the number of occorances of an element in an list

```haskell
count :: (Eq a) => a -> [a] -> Int
count e x = foldl (\acc a -> if a == e then acc + 1 else acc) 0 x
```

Alternative Methods

1. Could of done it recursively (like the folowing example) but using a fold was a shorter way to do it. Only down side would be that using a lambda makes the function a little more criptic, compared to the recursive solution which is alot easier to understand (SEE count')

```haskell
count' :: (Eq a) => a -> [a] -> Int
count' e [] = 0
count' e (x:xs)
  | e == x = 1 + (count' e xs)
  | otherwise = count' e xs
```

Question B

allPos returns an indicies of all occorances of an element in an list

```haskell
allPos :: (Eq a) => a -> [a] -> [Int]
allPos e x = collectIndexs x 0
    where collectIndexs [] _ = []
          collectIndexs (y:yx) i
              | e == y = i:(collectIndexs yx (i+1))
              | otherwise  = collectIndexs yx (i+1)
```

Question C

firstLastPos returns indicies of first and last occorance of element in list indicies start from 1 so if result is (0,0) then the element isnt in the list

```haskell
firstLastPos :: (Eq a) => a -> [a] -> (Int, Int)
firstLastPos e a = (firstPos a 0, lastPos a 0 0)
```

The fuction firstPos moves from left to right until we reach an occorance of the element we are looking for

```haskell
   where firstPos [] _ = 0
         firstPos (x:xs) i
           | e == x = i + 1
           | otherwise = firstPos xs (i+1)
```

The lastPos function reads left to right through the list keeping track of the last position we saw the element we are looking for

```haskell
         lastPos [] _ 0 = 0
         lastPos [] _ l = l + 1
         lastPos (x:xs) i l
           | e == x = lastPos xs (i+1) i
           | otherwise = lastPos xs (i+1) l
```

Part 2

Question A

I chose selection sort as it seemed the most natually recursive

Alternative Methods

1. I first considered using a left fold, using the accumulator as the sorted portion of the list, but then it seemed to get a bit messy, using a lambda wouldent be very easy to read aspecally sinse there was a few things to do. Could of defined some extra functions using a where clause to extract out the logic from the lambda but it still seemed more messy than it had to be

```haskell
sort1 :: (Ord a) => [a] -> [a]
sort1 = selectionSort

selectionSort :: (Ord a) => [a] -> [a]
```

Selection sort only has 1 base case

1. Empty list: In this case we return the empty list as the empty list is trivially sorted

```haskell
selectionSort [] = []
```

Otherwise while we take the minimum element in the unsorted portion of the list and place that at the end of the sorted list. Then recurse on the remaining unsorted list

```haskell
selectionSort x =
  let m = minimum x
      l = takeWhile (/=m) x
      r = drop ((length l) + 1) x
      remaining = l ++ r
  in m:(selectionSort remaining)
```

Alternative Solution

1. Insted of using the default minium function we could allow passing in the function which decides the smallest element, this would allow more flexibility

Question B

I chose merge sort as quick sort was in the tutorials I used to learn Haskell. It also seemed like a more interesting sorting algorythm to implement recursively

Alternative Methods

1. Rather than using the the default orderable '>' function to compare elements we could pass in our own function to do this. Making the sort more flexable

```haskell
sort2 :: (Ord a) => [a] -> [a]
sort2 = mergeSort

mergeSort :: (Ord a) => [a] -> [a]
```

The base cases of the merge sort algorythm are the folowing

1. **Empty Array:** In this case we want to return an empty list as the empty list is trivially sorted
2. **Single element:** Simlar to previous, just return element in list as singleton list is trivally sorted

```
mergeSort [] = []
mergeSort (x:[]) = [x]
```

Otherwise we want to split our list in half and recurse on the left and right then merge the result

```
mergeSort a =
  let mid = (length a) `div` 2
      l = take mid a
      r = drop mid a
  in merge (mergeSort l) (mergeSort r)
```

When merging we have two base cases

1. **Two empty lists:** in this case we just return an empty list
2. **One empty list:** Here we just return the list that is non empty

```
  where merge [] [] = []
        merge [] x = x
        merge x [] = x
```

Otherwise we recurse through the two lists build our merged list one element at a time, always placing the smaller of the two heads next in the list

```
        merge allX@(x:xs) allY@(y:ys)
          | x > y = y:(merge allX ys)
          | otherwise = x:(merge xs allY)
```

Question Extras

I also implemented tree sort as it nicely illustrates the power of the algeabraic data types. The binary tree isnt auto balancing so its not always as efficient as it could be.

Data type defining a binary tree as an element, a left tree and a right tree

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Eq)
```

treeSingleton takes an element and returns a tree containing only that element

```
treeSingleton :: a -> Tree a
treeSingleton x = Node x EmptyTree EmptyTree
```

treeInsert takes a tree and an element, returns a new tree containing the given element

```
treeInsert :: (Ord a) => Tree a -> a -> Tree a
treeInsert EmptyTree e = Node e EmptyTree EmptyTree
treeInsert (Node a l r) x
  | x < a = Node a (treeInsert l x) r
  | otherwise = Node a l (treeInsert r x)
```

treeFlatten takes a tree and traverse it using a depth first search, returns an ordered list containing all the elements that are in the

```
treeFlatten :: Tree a -> [a]
treeFlatten EmptyTree = []
treeFlatten (Node a l r) = (treeFlatten l) ++ [a] ++ (treeFlatten r)
```

treeSort takes a list of elements and sorts it by inserting all the elements into a tree and the flattening it

```
treeSort :: (Ord a) => [a] -> [a]
treeSort xs = treeFlatten (foldl (\acc e -> treeInsert acc e) EmptyTree xs)
```

Part 3

Define a map as an list of map elements

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

```
emptyMap :: Map a b
emptyMap = []
```

```
hasKey :: (Eq a) => a -> Map a b -> Bool
hasKey x m = any (\(k, v) -> x == k) m
```

setKey inserts a key value pair into a map. Overriting any exsisting pair with the given key

```
setKey :: (Eq a) => a -> b -> Map a b -> Map a b
setKey k v m
  | hasKey k m = setKey k v (delKey k m)
  | otherwise = (k, v):m
```

getVal takes a map and key, returns the value asociated with the give key in the map. If the key does not exsist an error is thrown

```
getVal :: (Eq a) => a -> Map a b -> b
getVal k m
  | not (hasKey k m) = error "Map does not contain that key"
  | otherwise = find m
  where find ((x, y):xs) = if k == x then y else find xs
```

delKey takes a key and map, returns a new map with the given key removed from the map. If the key is not contained in the map then an error is thrown

```
delKey :: (Eq a) => a -> Map a b -> Map a b
delKey k m
  | not (hasKey k m) = error "Map does not contain that key"
  | otherwise = foldl (\acc e@(x, y) -> if x == k then acc else e:acc ) [] m
```

Part 4

Build map takes an list of elements and returns a map where the keys are elements of the origonal list and the asociated values are the number of times a element occors in the list

```
buildMap :: (Eq a) => [a] -> Map a Int
buildMap xs = map (\g -> (g, (count g xs))) (uniquic xs)
```

the function uniquic finds all the uniquic elements in a list

```
  where uniquic xs = foldl (\acc e -> if e `elem` acc then acc else e:acc) [] xs
```