QN 3.3) Solution:

We have,

```
cnt :: Eq a => a -> [a] -> Int
cnt x [] = 0
cnt x (y:ys)
      | x == y = 1 + (cnt x ys)
      | otherwise = cnt x ys
con :: [a] -> [a] -> [a]
con [] ys = ys
con (x:xs) ys = x : (con xs ys)
```

The cnt function is for counting the number of appearance of value of x in a list. And, the con function is for joining two lists into one single list containing the values of those two list.

To Prove:

cnt x (con s t) == (cnt x s) + (cnt x t)

Using the mathematical induction,

Basic Steps: Let's consider the empty list.

Taking s = []

So our expression would be,

=> cnt x (con [] t) == (cnt x []) + (cnt x t)

=> cnt x t == 0 + (cnt x t)

=> cnt x t == cnt x t   (True)

[As cnt returns Int, so numeric operations validate]

Inductive Steps:

Assumption: cnt x (con s t) == (cnt x s) + (cnt x t)

Let’s take a list k such that the length(k) = (length(s)+1), and the extra value that k contains be c.

```
To prove:

cnt x (con k t) == (cnt x k) + (cnt x t)
So, our expression would be,
=> cnt x (con k t) == (cnt x k) + (cnt x t)
=> cnt x (k ++ t) == (cnt x k) + (cnt x t)  [As, (con k t) = k ++ t]
```

From our program,

```
If x == c, the above expression is equal to
=> 1 + cnt x (s ++ t) == (1 + (cnt x s)) + (cnt x t)
=> 1 + cnt x (con s t) == 1 + (cnt x s) + (cnt x t)
                                [As, s ++ t = (con s t)]
=> 1 + (cnt x s) + (cnt x t) == 1 + (cnt x s) + (cnt x t)  (True)
                                [From our assumption,
                                cnt x (con s t) == (cnt x s) + (cnt x t)]

Otherwise,
=> 0 + cnt x (s ++ t) == (0 + (cnt x s)) + (cnt x t)
=> 0 + cnt x (con s t) == 0 + (cnt x s) + (cnt x t)
                                [As, s ++ t = (con s t)]
=> 0 + (cnt x s) + (cnt x t) == 0 + (cnt x s) + (cnt x t)  (True)
                                [From our assumption,
                                cnt x (con s t) == (cnt x s) + (cnt x t)]
```

Hence, we can conclude inductively that,

```
cnt x (con s t) == (cnt x s) + (cnt x t)
```

QN 3.4a) solution:

```
rotate :: Int -> [a] -> [a]

rotate _ [] = []

rotate x xs

       | (x `mod` length(xs)) == 0 = xs

rotate x ys = rotate ((x-1) `mod` length(ys)) (tail ys ++ [head ys])
```

QN 3.4b) solution:

```
--please note that you must load the rotate function from question 3.4a in order
to run this program successfully.

circle :: [a] -> [[a]]

circle [] = []

circle xs = [(rotate x xs) | x<-[0..(length(xs)-1)]]
```