CSCI 189 Assignment 7 Problem Set

This assignment provides an opportunity to learn about recurrence relations, recursive functions, and their implementation in Haskell.

Summary of Concepts in Section 2.4 and Haskell

Defining functions in Haskell is a discipline that requires mastery of recursion and recurrence relations. A "recurrence relation" is a style for defining functions that uses self-reference in order to achieve generality. Recurrence relations occur when we examine the following artifacts:

1. Sequences (lists) and sets
2. Functions that transform sequences and sets
3. Solving recurrence relations
4. Recursion and proofs by induction

We illustrate each of these in the discussion below, along with an interesting application in computing disk read/write time.

Recursively defined sequences and sets

A *sequence* is a list enumerated in some order, like $1, 2, 4, 8, 16, \ldots$. Each member of a sequence sequence can be defined using a *recurrence relation*, which is a function describing 1) how the first element can be computed and 2) how the nth element in the sequence can be computed from a previous one. For example, each member of the above sequence can be defined by the following recurrence relation.

$f(1) = 1$
$f(n) = 2f(n-1) \quad \text{for } n > 1$

Of course, the members of this sequence can be more simply defined by the function $f(n) = 2^{n-1}$. But many other sequences are more easily defined using a recurrence relation. Consider, for example, the Fibonacci sequence $1, 1, 2, 3, 5, 8, 13, \ldots$ is one such sequence, and its members are usually defined by the following recurrence relation:

$fibo(1) = 1$
$fibo(2) = 1$
$fibo(n) = fibo(n-2) + fibo(n-1) \quad \text{for } n > 2$

Sets can also be conveniently enumerated using recurrence relations. Consider the set B^* of all strings over the alphabet $B = \{0, 1\}$, which are the binary numbers. That is:

$B^* = \{\lambda, 0, 1, 00, 01, 01, 11, 000, 001, 010, 011, 100, 101, 110, \ldots\}$

Each member of this set can be defined by the following recurrence relation:

$\lambda \in B^*$
$Bb \in B^*$ if $b \in \{0, 1\}$ and $B \in B^*$

Notice that all three of these examples follow the same pattern: a "base step" and a "recursive step." The base step defines the starting point of the sequence, and the recursive step defines how to derive the next element in the sequence, given the previous one. Thus, in recurrence relations we have something akin to the proof by induction strategy that we learned earlier in the semester. We will see that induction and recurrence relations are strongly connected through the idea of a "recursive function."

Recursive functions

It is a short step from a recurrence relation to a so-called "recursive function." A recursive function is one that mirrors the definition of a recurrence relation, such as *f* or *fibo* in the above section. The nice thing about recursive functions is that we can write them directly in Haskell, and so they can be called to perform computational tasks for us. Here are the Haskell renditions of the relations *f*, *fibo*, and B^*:

```
> f n
>  | n < 1       = error "Improper input"
>  | n == 1      = 1
>  | otherwise   = 2*f(n-1)

> fibo n
>  | n < 1       = error "Improper input"
>  | n <= 2      = 1
>  | otherwise   = fibo(n-2) + fibo(n-1)

> bStar n
>   | n < 1            = error "Improper input"
>   | n == 1           = ""
>   | n == 2           = "0"
>   | n == 3           = "1"
>   | n `mod` 2 == 0   = bStar (n `div` 2) ++ "0"
>   | otherwise        = bStar (n `div` 2) ++ "1"
```

The first two of these are very straightforward. The third is a bit tricky, and it depends on an understanding of how each string of 0's and 1's can be generated from a predecessor in the sequence that has one fewer bits. It turns out that that predecessor is at position $\lfloor n/2 \rfloor$ in the sequence. For example, to generate the strings "110" and "111", which are at positions 14 and 15 in the sequence,

we need to append a "0" and a "1" respectively to the string "11" which is at position 7 in the sequence.

To visualize how these functions work, we can generate their respective sequences by mapping them over the positive integers {1, 2, ... }. This is easily done by making the following calls to the Haskell "map" function, which builds a list by applying the function of interest to each of the positive integers:

```
Asst7> map f [1,2..]
Asst7> map fibo [1,2..]
Asst7> map bStar [1,2..]
```

Since each of these function applications will (potentially) compute an infinite list of results, you will want to interrupt that computation after you see the first few elements of the list appear.

Recursive functions can also be designed to perform computations other than those which generate sequences. For example, the following function finds the index of the maximum element in a (finite) list of n elements.

```
> maxIndex aList
>   | n < 1      = error "Improper input"
>   | n == 1     = 1
>   | otherwise  = 1 + length (takeWhile (/= x) aList)
>   where x = maximum aList
>         n = length aList
```

This function can be useful in sorting a list, as discussed on page 129 of your text and illustrated in the laboratory tutorial for this assignment.

Another useful function that can be defined recursively is one that searches a list to see if a particular value is there or not. This is discussed on page 130 of your text, and is implemented in Haskell as follows:

```
> binarySearch aList i j x
>   | i > j               = 0
>   | x == aList!!(k-1)   = k
>   | x < aList!!(k-1)    = binarySearch aList i (k-1) x
>   | otherwise           = binarySearch aList (k+1) j x
>   where k = (i+j) `div` 2
```

To understand the dynamics of this function, review the discussion in your text and call it a few times using the laboratory tutorial.

Solving recurrence relations

To solve a recurrence relation, we want to find a "closed form" expression for it. For instance, the closed form expression for the recurrence relation:

$f(1) = 1$
$f(n) = 2f(n-1) \quad \text{for } n > 1$

is, in fact, $f(n) = 2^{n-1}$. How can we discover this and prove that it is correct?

One technique for discovering a closed form for a recurrence relation is to try some initial cases, guess the closed form, and then prove that your guess is correct. This technique is discussed on page 121 of your text. For instance, the above relation yields the following initial cases:

n	$f(n)$
1	1
2	2
3	4
4	$8 = 2^3$
5	$16 = 2^4$
$\vdots$	

So now we look at the second column and guess that the 6th entry will be $32 = 2^5$, suggesting that the kth entry will likely be 2^{k-1}. Finally, we need to prove (by induction) that our guess will work for all cases, not just these initial ones. Our induction proof must show:

The base case (i.e., $f(1) = 1$, by definition of f), and

The induction step: assume the hypothesis that, for some k, $f(k) = 2^{k-1}$. Then for $k+1$:

$$\begin{aligned} f(k+1) &= 2 \times f(k) && \text{by definition of } f \\ &= 2 \times 2^{k-1} && \text{by our hypothesis} \\ &= 2^{(k+1)-1} \end{aligned}$$

This completes the proof.

A second technique for discovering a closed form for a recurrence relation is by using the summation formula below. That is, all linear first order recurrence relations have the form $S(n) = cS(n-1) + g(n)$. This can be solved by finding c and $g(n)$ in the following expression:

$$\begin{aligned} S(n) &= c^{n-1}S(1) + c^{n-2}g(2) + \ldots + cg(n-1) + g(n) \\ &= c^{n-1}S(1) + \textstyle\sum_{i=2}^{n} c^{n-i}g(i) \end{aligned}$$

For example, consider again the sequence defined by:

$f(1) = 1$
$f(n) = 2f(n-1) \quad \text{for } n > 1$

and let $g(n) = 0$ and $c = 2$. Then $f(n) = 2^{n-1}(1) + \sum_{i=2}^{n} 0 = 2^{n-1}$. The trick here is to find c and g!

There's a nice example in your book on page 137 for computing the average read time for data on a disk. It uses this second method to solve a recurrence relation that expresses the average read time for a disk with n tracks.

Recursion and proofs by induction

Recursive Haskell functions can be used in inductive proofs about the properties of such functions. Consider the following two Haskell functions, one of which computes the length of a list, and the other concatenates (joins) two lists into one. Because both of these are predefined in Haskell (they're named `length` and `++`), we redefine them here with slightly different names so we can illustrate some inductive proofs of their properties. the numbers on the right will be used in our proofs about the properties of these functions.

```
> len []     = 0                      -- len.1  (base case)
> len (x:xs) = 1 + (length xs)        -- len.2  (recursive case)

> cat [] ys     = ys                  -- cat.1  (base case)
> cat (x:xs) ys = x : (cat xs ys)     -- cat.2  (recursive case)
```

Both of these are recursive functions, depending on the length of the list passed as an argument. The base case defines the function for an empty list (e.g., the length of an empty list is 0), and the recursive step shows how to compute the function based on a list slightly smaller than the current list. For example,

```
len [1,3,4,7]
   = 1 + len [3,4,7]
   = 1 + (1 + len [4,7])
   = 1 + (1 + (1 + len [7]))
   = 1 + (1 + (1 + (1 + len [])))
   = 1 + (1 + (1 + (1 + 0)))
   = 4
```

The first four calls use the second line of the textttlen function. Here are two inductive proofs: 1) that `cat (xs cat (ys zs)) = cat (cat xs ys) zs)`, and 2) that `len (cat xs ys) = len xs + len ys`. Notice in each of these proofs that the base case in the proof uses the base case in the recursive definition, and that the induction step in the proof uses the recursive step in the definition. (This method of proof about properties of lists and other data structures is called "structural induction.")

```
1) Proof that cat (xs cat (ys zs)) = cat (cat xs ys) zs):
Base case:  cat ([] cat ys zs) = cat ys zs              by cat.1
                               = cat (cat [] ys) zs     by cat.1
Inductive step hypothesis: that cat xs (cat ys zs) = cat (cat xs ys) zs for some list xs.
          Then for a list x:xs one element longer, we have:
          cat x:xs (cat ys zs) = x: cat (xs (cat ys zs))   by cat.2
                               = x: (cat (cat xs ys) zs)    by hypothesis
                               = cat (x:(cat xs ys) zs)     by cat.2
                               = cat (cat x:xs ys) zs)      by cat.2

2) Proof that len (cat xs ys) = len xs + len ys:
Base case: len (cat [] ys) = len ys                     by cat.1
                           = 0 + len ys                 by arithmetic
                           = len [] + len ys            by len.1
Inductive step hypothesis: len (cat xs ys) = len xs + len ys for some list xs.
           Then for a list x:xs one element longer, we have:
           len (cat x:xs ys) = len x: (cat xs ys)       by cat.2
                             = 1 + len (cat xs ys)      by len.2
                             = 1 + len xs + len ys      by hypothesis
                             = len x:xs + len ys        by len.2
```

Problems to be handed in

Section 2.4 (p 139) Exercises 3, 6, 8, 18, 25, 37, 41, 44, 47 (in Haskell), 51, 52a (in Haskell), 59 (in Haskell), 80, 83, 86 (give the recurrence relation, write a Haskell function, and solve it with a call to that function).

"In Haskell" means write a Haskell function. E.g., Exercise 52b in Haskell would be:

```
> minimum2 (x:xs)
>    |  xs == []      = x
>    |  x <= y        = x
>    |  otherwise     = y
>    where y = minimum2 xs
```

To test this function, we would write something like:

```
Asst7> minimum2 [3,2,4,1,7,6]
```

which would give the answer 1, since:

```
minimum2 [3,2,4,1,7,6]
       = minimum2 [2,4,1,7,6]     (the "otherwise" part of the definition)
       = minimum2 [4,1,7,6]       ("otherwise" again, with y = minimum2 [1,7,6])
       = minimum2 [1,7,6]         ("otherwise" again, with y = minimum2 [7,6])
       = 1                        (the "x <= y" part of the definition, since 1 <= 6)
```