FUNCTIONAL SPECIFICATION OF ALGORITHMS, LAB EXERCISES WEEK 2, PART 2

DAAN MULDER, 10279245

Balance

Exercise 6. For exercise 6, we implement the Balance game in a similar way to the Mastermind implementation. The secret is a list of n Coins, which can be Light or Normal (Heavy was not implemented to make it easier to avoid situations where several options cannot be distinguished, such as [Light,Normal,Normal] and [Normal,Heavy,Heavy]). Guesses can be made with a list of n scale positions (either left, right or off the scale). `reaction` then adds the weights for each side (1 for a Light Coin and 2 for a Normal Coin) and gives the appropriate feedback.

Both Knuth's minimax strategy (`exercise6a`) and the 'maximize entropy' strategy (`exercise6b`) were implemented, though in tests they always behaved the same.

```
module Balance

where

import Data.List

data Coin = Light | Normal deriving (Eq,Show,Bounded,Enum)
data Feedback = Leftbound | Balanced | Rightbound deriving (Eq,Show)
data ScalePos = L | R | Off deriving (Eq,Show)

type Pattern = [Coin]
type Weighing = [ScalePos]

count :: [Coin] -> Int
count [] = 0
count (x:xs) = if x == Light then 1 + count xs else if x == Normal
               then 2 + count xs else error "Undefined coin"

gatherSide :: Pattern -> Weighing -> ScalePos -> [Coin]
gatherSide [] [] _ = []
gatherSide (x:xs) (y:ys) side = if y == side then (x:gatherSide xs ys side)
                                else gatherSide xs ys side

reaction :: Pattern -> Weighing -> Feedback
reaction secret guess = if left > right then Leftbound else if left < right then Rightbound
                        else Balanced where {left = count (gatherSide secret guess L);
                        right = count (gatherSide secret guess R)}
```

```
makeList :: [a] -> Int -> [[a]]
makeList xs 1 = [[x] | x <- xs]
makeList xs n = [[x] ++ y | x <- xs, y <- makeList xs $ n-1]

guessing :: Pattern -> Weighing -> [Pattern] -> [Pattern]
guessing secret guess xs = filter (\x -> reaction x guess == reaction secret guess) xs


exercise6amin :: [(Weighing, Int)] -> Weighing
exercise6amin xs = fst $ (filter (\ (_,b) -> b == minimum (map snd xs)) xs) !! 0

exercise6amax :: [(Weighing, [[Feedback]])] -> [(Weighing, Int)]
exercise6amax xs = map (\ (a,b) -> (a,maximum b)) $
                   map (\ (a,b) -> (a,map length b)) xs

exercise6alist :: Int -> [Pattern] -> [(Weighing, [[Feedback]])]
exercise6alist n xs = map (\ (a,b) -> (a, group b)) $
                      [(maybeGuess, [reaction maybeSecret maybeGuess |
                      maybeSecret <- xs]) | maybeGuess <- makeList [L,R,Off] n]

exercise6aplay :: Int -> Pattern -> [Pattern] -> Int -> Int
exercise6aplay _ secret (x:[]) i = if x == secret then i else -1
exercise6aplay n secret xs i = exercise6aplay n secret (guessing secret (exercise6amin $
                               exercise6amax $ exercise6alist n xs) xs) (i+1)

exercise6a :: Int -> Pattern -> Int
exercise6a n secret = exercise6aplay n secret firstList 0
                      where firstList = makeList [Light,Normal] n


exercise6bmin :: [(Weighing, Float)] -> Weighing
exercise6bmin xs = fst $ (filter (\ (_,b) -> b == minimum (map snd xs)) xs) !! 0

exercise6bentropy :: [(Weighing, [[Feedback]])] -> [(Weighing, Float)]
exercise6bentropy xs = map (\ (a,b) -> (a,sum $ map (\ x -> fromIntegral x *
                        (log $ fromIntegral x)) $ b)) $ map (\ (a,b) -> (a,map length b)) xs

exercise6bplay :: Int -> Pattern -> [Pattern] -> Int -> Int
exercise6bplay _ secret (x:[]) i = if x == secret then i else -1
exercise6bplay n secret xs i = exercise6bplay n secret (guessing secret (exercise6bmin $
                               exercise6bentropy $ exercise6alist n xs) xs) (i+1)

exercise6b :: Int -> Pattern -> Int
exercise6b n secret = exercise6bplay n secret firstList 0
                      where firstList = makeList [Light,Normal] n
```