UNIVERSITY OF OXFORD
SOFTWARE ENGINEERING PROGRAMME

Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Tel +44(0)1865 283525 Fax +44(0)1865 283531
info@softeng.ox.ac.uk www.softeng.ox.ac.uk

Part-time postgraduate study in software engineering

Functional Programming, FPR

25th February – 1st March 2019

ASSIGNMENT

The purpose of this assignment is to test the extent to which you have achieved the learning objectives of the course. As such, your answer must be substantially your own original work. Where material has been quoted, reproduced, or co-authored, you should take care to identify the extent of that material, and the source or co-author.

Your answers to the questions on this assignment should be submitted using the Software Engineering Programme website — `www.softeng.ox.ac.uk` — following the submission guidelines. When submitting the assignment online, it is important that you formally complete all three assignment submission steps: step 1, upload your files; step 2, check your files; step 3, read through the declaration, and click the **I Agree** button followed by the **Submit now** button. The deadline for submission is 12 noon on Tuesday, 16th April 2019.

We hope to have preliminary results and comments available during the week commencing Monday, 3rd June 2019. The final results and comments will be available after the subsequent examiners' meeting.

ANY QUERIES OR REQUESTS FOR CLARIFICATION REGARDING THIS ASSIGNMENT OR PROBLEMS INSTALLING SOFTWARE SHOULD, IN THE FIRST INSTANCE, BE DIRECTED TO THE PROGRAMME OFFICE WITHIN THE NEXT TWO WEEKS.

---

1 Introduction

This assignment is about a famous puzzle. It is called "Twelve Coins" in the excellent book *Algorithmic Puzzles* by Anany and Maria Levitin (Oxford University Press, 2011, Puzzle 142), and described as follows (I've made minor changes):

> *There are $n > 2$ coins, identical in appearance; either all are genuine, or exactly one of them is fake. It is unknown whether the fake coin is lighter or heavier than a genuine one. You have a two-pan balance scale, without weights. The problem is to find whether all the coins are genuine and, if not, to find the fake coin and to establish whether it is lighter or heavier than the genuine ones. Design an algorithm to solve the problem, in the minimum number of weighings.*

Of course, with $n = 0$ there are no coins; with $n = 1$ you can't weigh anything; and with $n = 2$ you couldn't distinguish a fake coin from a genuine one. So we assume $n > 2$; the problem is usually presented with $n = 12$.

According to *Algorithmic Puzzles*, the puzzle first appeared in 1945, and it has been alleged that this puzzle "diverted so much war-time scientific effort that there was serious consideration of a proposal to inflict compensating damage by dropping it in enemy territory". I hope you find it fun!

My answers use a couple of functions from the standard libraries:

```
import Data.List (minimumBy, nub)
import Data.Ord (comparing)
```

2 States and tests

The essence of our approach is to simulate a physical solution to the problem that maintains up to four piles of coins: pile U, of coins whose status is *unknown*; pile G, of coins known to be *genuine*; pile L, of coins that are possibly *light* (but known not to be heavy); and pile H, of coins that are possibly *heavy* (but known not to be light). Initially, all the coins are in pile U; at the end, pile U is empty, piles L and H have either zero or one coin together, and all the other coins are in pile G. So if piles L and H are both empty, then all the coins are genuine; otherwise, one of those piles is empty, the other has precisely one coin, that coin is fake, and which pile it is in tells you whether it is light or heavy. The challenge now is to find the best way of getting from the initial state to a final state.

However, we can be a bit more precise than just saying that there are four piles, some empty, because the physical solution will go through two distinct phases. In the first phase, we don't know whether any coin is fake; it turns out that we need consider only piles U and G. Initially, all coins are in U. If the two groups balance

the scales, we move $2k$ coins from pile U to pile G and continue. If the two groups don't balance, then one of those $2k$ coins must be fake; we move the k coins on the lighter pan of the scales to pile L, the k coins on the heavier pan to pile H, and all the remaining coins in pile U to pile G. We are now in the second phase, knowing that there is a fake coin; from this point, pile U remains empty, and we need consider only piles L, H and G.

We therefore model the state of the simulation by the datatype *State*:

```
data State = Pair Int Int | Triple Int Int Int
  deriving (Eq, Show)
```

A state *Pair u g* represents there being u coins in pile U and g coins in pile G (and piles L and H being empty). A state *Triple l h g* represents there being l coins in pile L, h coins in pile H, and g coins in pile G (and pile U being empty). Moreover, we only get into a state *Triple l h g* in the second phase, having deduced that there is a fake coin; therefore $l + h > 0$ remains true.

The simulation proceeds by conducting 'tests', which consist of weighing one group of k coins against another group of k coins. Because there are two kinds of state, there are two kinds of test:

```
data Test = TPair (Int, Int) (Int, Int) | TTrip (Int, Int, Int) (Int, Int, Int)
  deriving (Eq, Show)
```

Each kind of test denotes how many coins to take, from which piles, for the left and the right pan of the scales: *TPair* (a, b) (c, d) denotes weighing a coins from pile U plus b coins from pile G against c coins from pile U plus d coins from pile G; *TTrip* (a, b, c) (d, e, f) denotes weighing $a + b + c$ coins from piles L,H,G respectively against d, e, f coins from piles L,H,G. For example, in the initial state *Pair* 12 0 for the $n = 12$ instance of the problem, we can weigh 3 coins from pile U against 3 more coins from the same pile, represented by the test *TPair* $(3, 0)$ $(3, 0)$.

As the names suggest, a *TPair* test is only supposed to be conducted in a *Pair* state, and a *TTrip* test in a *Triple* state. Moreover, in each test, the number of coins should be the same in each pan of the scales (you learn nothing from weighing 3 coins against 4). And finally, there must be sufficiently many coins in the various piles for the test (you can't use 7 coins from pile U when there are only 5 coins left in it).

1. Define a function

   ```
   valid :: State → Test → Bool
   ```

   to determine whether a given test is valid in a given state, according to the above criteria. For example, *valid* (*Pair* 12 0) (*TPair* $(3, 0)$ $(3, 0)$) = *True*.

3 Choosing and conducting a test

For each state s and test t such that $valid\ s\ t = True$, we now work out the possible outcomes of test t in state s. There are always three possible outcomes, depending on whether the coins in the left pan of the scales are lighter than, the same weight as, or heavier than those in the right pan.

For example, conducting test $TPair\ (3,0)\ (3,0)$ in state $Pair\ 12\ 0$ can go three ways. If the 3 coins in the left pan are lighter than the 3 coins in the right pan, then we know there is a fake coin among those 6 coins; we move the 3 coins in the left pan to pile L, the 3 coins in the right pan to pile H, and the remaining 6 coins to pile G, ending up in state $Triple\ 3\ 3\ 6$. Symmetrically, if the 3 coins in the left pan are heavier than those in the right, we move the 3 coins in the left pan to pile H, those in the right pan to pile L, and again the remaining 6 coins to pile G, again ending up in state $Triple\ 3\ 3\ 6$. But if the pans balance, we know those 6 coins are all genuine; so we move all 6 to pile G, and end up in state $Pair\ 6\ 6$.

2. We represent the three outcomes of a test as a list. Define a function

$$outcomes :: State \to Test \to [State]$$

to compute the three outcomes of a valid test. For example,

$$\begin{aligned} &outcomes\ (Pair\ 12\ 0)\ (TPair\ (3,0)\ (3,0)) \\ &\quad = [Triple\ 3\ 3\ 6, Pair\ 6\ 6, Triple\ 3\ 3\ 6] \end{aligned}$$

We now consider how to determine the tests that are both valid and sensible in a given state. What do we mean by 'sensible'? For one thing, there is no virtue in having coins known to be genuine in both pans at once, because those coins are known to have the same weight. For example, we can ignore tests of the form $TPair\ (a,b)\ (c,d)$ in which both $b,d > 0$: supposing $b \geqslant d$, it suffices to consider just the test $TPair\ (a,b-d)\ (c,0)$ instead. Similarly, we can ignore tests of the form $TTrip\ (a,b,c)\ (d,e,f)$ with $c,f > 0$.

Secondly, the scales are symmetric; so we don't need to consider both the tests $TPair\ (a,b)\ (c,d)$ and $TPair\ (c,d)\ (a,b)$. We can break the tie between these two by picking the lexically ordered one, that is, with $(a,b) \leqslant (c,d)$ (remember that pairs are ordered!). And similarly for $TTrip$ tests.

A third criterion is that the test must be guaranteed to increase our knowledge about the coins. For example, consider state $Triple\ 3\ 0\ 6$ and test $TTrip\ (0,0,3)\ (3,0,0)$, for which the three outcomes are $[Triple\ 0\ 0\ 9, Triple\ 0\ 0\ 9, Triple\ 3\ 0\ 6]$. The third outcome is the same as the state we started with! In that case, we learned nothing from the test; to avoid getting stuck in a loop, we want to avoid such tests. (In fact, the outcome of this test is predetermined. The first two of these

three outcomes are actually impossible: the left pan contains three genuine coins, the right pan contains three coins one of which is known to be light—because it contains *all* the coins not known to be genuine—and so the left pan must be heavier than the right pan. But this is hard to spot.)

We will define a function

$$weighings :: State \to [\, Test \,]$$

to generate the sensible tests according to the first two criteria in the first instance; we will handle the third criterion after that.

3. In a state $Pair\ u\ g$, we want to generate tests $TPair\ (a,b)\ (c,d)$ such that:

$$\begin{array}{ll} a+b=c+d & \text{-- same number of coins per pan} \\ a+b>0 & \text{-- no point in weighing only air} \\ b\times d=0 & \text{-- don't put genuine coins in both pans} \\ a+c\leqslant u & \text{-- enough unknown coins} \\ b+d\leqslant g & \text{-- enough genuine coins} \\ (a,b)\leqslant(c,d) & \text{-- symmetry breaker} \end{array}$$

   The third equation above requires one of b, d to be 0; if we take $b = 0$ then the last condition is satisfiable only if $d = 0$ too (why?), so instead we take $d = 0$. That means that we're looking for pairs (a, b) and $(a + b, 0)$ such that $a + b \neq 0$, $2a \leqslant u$, and $b \leqslant g$. Complete the definition of the *Pair* clause of *weighings*:

$$weighings\ (Pair\ u\ g) = [\, TPair\ (a,b)\ (a+b,0) \mid \ldots \,]$$

   For example,

$$\begin{array}{l} *\rangle \quad weighings\ (Pair\ 3\ 3) \\ [\, TPair\ (0,1)\ (1,0), TPair\ (0,2)\ (2,0), TPair\ (0,3)\ (3,0), \\ \ \ TPair\ (1,0)\ (1,0), TPair\ (1,1)\ (2,0) \,] \end{array}$$

4. The *Triple* case is a bit trickier. One approach is to define a subsidiary function *choices* such that $choices\ k\ (l,h,g)$ returns all valid selections $(i,j,k-i-j)$ of k coins. That means $0 \leqslant i \leqslant l$, $0 \leqslant j \leqslant h$, and $0 \leqslant k-i-j \leqslant g$.

$$choices :: Int \to (Int, Int, Int) \to [\, (Int, Int, Int) \,]$$

   For example:

$$\begin{array}{l} *\rangle \quad choices\ 3\ (2,2,2) \\ [\, (0,1,2), (0,2,1), (1,0,2), (1,1,1), (1,2,0), (2,0,1), (2,1,0) \,] \end{array}$$

   Define *choices*.

5. Now, in a state *Triple l h g*, we want to generate tests *TTrip* (a, b, c) (d, e, f) such that:

$$\begin{array}{lll} a+b+c = d+e+f & \text{-- same number of coins per pan} \\ a+b+c > 0 & \text{-- no point in weighing only air} \\ c \times f \quad = 0 & \text{-- don't put genuine coins in both pans} \\ a+d \quad \leqslant l & \text{-- enough light coins} \\ b+e \quad \leqslant h & \text{-- enough heavy coins} \\ c+f \quad \leqslant g & \text{-- enough genuine coins} \\ (a,b,c) \leqslant (d,e,f) & \text{-- symmetry breaker} \end{array}$$

We use *choices* to pick valid numbers of coin for each pan; suitable values of k range from 1 to $(l+h+g)$ `` `div` `` 2 (we have to have at least one coin in each pan, and can't have more than half the coins). Then for each suitable k, we choose k coins (a, b, c) from (l, h, g) for the left pan, another k coins (d, e, f) from what's left for the right pan, and make sure that $c \times f = 0$ and $(a, b, c) \leqslant (d, e, f)$. Hence complete the *Triple* clause of *weighings*:

weighings (*Triple l h g*) = [*TTrip* (a, b, c) (d, e, f) | ...]

6. Now for the third criterion, about making progress. We model this in terms of a special-purpose ordering on states: $s < s'$ when state s gives strictly more information about the coins than state s'. Because the number of coins is preserved, it generally suffices to do the comparison by the number of genuine coins: two *Pair* states and two *Triple* states are ordered by their G component (of course, more known-genuine coins means more information). We say that a *Triple* state always gives more information than a *Pair* state (since it represents progress, from the first to the second phase), and conversely a *Pair* state never gives more information than a *Triple* state. Complete the definition of the ordering on *State*.

instance *Ord State* **where** ...

7. Define a predicate *productive* on states and tests such that *productive s t* determines whether test t is guaranteed to make progress in state s, i.e. every possible outcome is a state providing more information according to the ordering above.

productive :: *State* → *Test* → *Bool*

In particular, *productive* (*Triple* 3 0 6) (*TTrip* $(0, 0, 3)$ $(3, 0, 0)$ = *False*, as we saw above.

8. Finally, we fulfill the third criterion by keeping only the *productive* tests among the possible *weighings*; define the function *tests* to do this.

   $tests :: State \to [\,Test\,]$

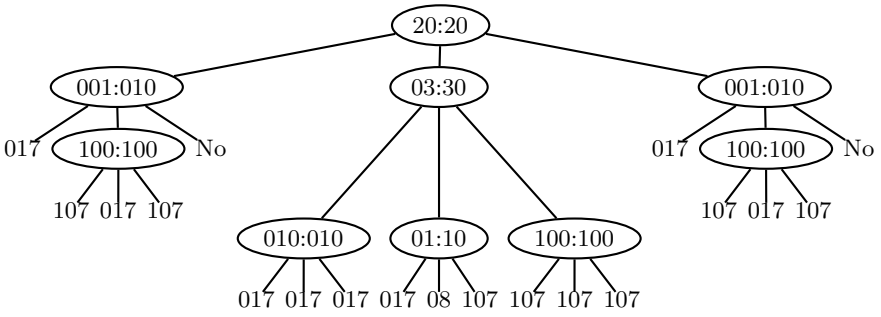
   For example, state *Triple* 3 0 6 admits 5 weighings, but one of these is *TTrip* $(0,0,3)$ $(3,0,0)$, which is unproductive, so *tests* returns only 4 productive weighings.

4 Decision trees

Now we can set about running the simulation of the weighing process. From a given start *State*, we build a decision tree of *Test*s, with final *State*s at the leaves. This is the type of tree we will start with:

```
data Tree = Stop State | Node Test [Tree]
  deriving Show
```

Each *Node* will have precisely three children, for the three possible outcomes of that test. We want to build a tree of minimum height—that is, which minimizes the length of the longest path from the root to any leaf—because that minimizes the number of tests we have to conduct. For example, the minimum height tree for $n = 8$ coins is:



(Each circled nodes shows a test, showing the number of coins drawn from the two or three active piles for each pan of the scales. The branching is for whether the outcome is that the left pan is lighter than, the same weight as, or heavier than the right pan. Each leaf shows a final state, with the number of coins in each active pile. For example, the middle leaf shows the final state with all 8 coins in pile G; immediately to its right is a final state with one coin in pile L.)

This three demonstrates that three tests are sufficient for $n = 8$; also, three tests are necessary, because there are 17 distinct final states (any one of the 8 coins could be light, or heavy, or they could all be genuine), and a ternary tree of height k has at most 3^k leaves.

9. Define a predicate

   $$final :: State \to Bool$$

   to determine whether a *State* is final—that is, whether it has determined either that all coins or genuine, or that one coin is fake, and in the latter case which coin and whether it is light or heavy.

10. Define a function

    $$height :: Tree \to Int$$

    to compute the height of a tree (such that the tree above has height 3).

11. Define a function

    $$minHeight :: [Tree] \to Tree$$

    that returns the tree of minimum height from a non-empty list of trees.

12. Hence define a function

    $$mktree :: State \to Tree$$

    to build a minimum-height decision tree of tests from a given start state *s*: *Stop* if *s* is *final*, and otherwise generate all possible productive tests with *tests s*, recursively build trees from the *outcomes* of each such test, and then pick the one that yields the best tree overall.

5 Caching heights

The program in the previous section works, but it is rather slow. It takes about 10 seconds on my (old) desktop to compute the minimum-height tree shown above for $n = 8$, and I gave up waiting with $n = 12$. One small source of inefficiency is that it is continually recomputing the heights of trees; it would be better to label every tree with its height, so that the height can then be looked up in constant time. To that end, we define the datatype

```
data TreeH = StopH State | NodeH Int Test [TreeH]
  deriving Show
```

with the intention that the label *h* in a tree *NodeH h t ts* is the height of the tree, so that height can be computed in one step:

$$
\begin{aligned}
&\mathit{heightH} :: \mathit{TreeH} \to \mathit{Int} \\
&\mathit{heightH}\ (\mathit{StopH}\ s) \quad\ = 0 \\
&\mathit{heightH}\ (\mathit{NodeH}\ h\ t\ \mathit{ts}) = h
\end{aligned}
$$

13. Define a function

    $$\mathit{treeH2tree} :: \mathit{TreeH} \to \mathit{Tree}$$

    to convert a labelled *TreeH* back to the corresponding *Tree*.

14. Define a 'smart constructor'

    $$\mathit{nodeH} :: \mathit{Test} \to [\,\mathit{TreeH}\,] \to \mathit{TreeH}$$

    that assembles a test and a 3-list of trees into a single tree, without recomputing heights from scratch.

15. Hence define a function

    $$\mathit{tree2treeH} :: \mathit{Tree} \to \mathit{TreeH}$$

    as an inverse to *treeH2tree*. Convince yourself that

    $$\mathit{heightH} \cdot \mathit{tree2treeH} = \mathit{height}$$

16. Now repeat Question 12, but for building a *TreeH* rather than a *Tree*.

    $$\mathit{mktreeH} :: \mathit{State} \to \mathit{TreeH}$$

Sadly, the results are disappointing—caching the heights in this way does not actually save much time:

```
*⟩  : set + s
*⟩  mktree (Pair 8 0)
(9.34 secs, 3,648,121,496 bytes)
*⟩  mktreeH (Pair 8 0)
(9.23 secs, 3,578,513,216 bytes)
```

6 A greedy solution

The inefficiency in our tree-building functions is not primarily down to recomputing heights; more fundamentally, we try too many possible tests. It turns out that a *greedy algorithm* is possible: that is, one that makes a simple local choice at each step, while still ending up with an optimal solution overall. The details are a bit messy, so I will simply present them here—I will provide a hint about where they come from in the assessment reports.

The bottom line is that it is possible to identify a predicate *optimal* that takes a state s and a valid test t for s and determines whether t is optimal:

$$
\begin{aligned}
&\mathit{optimal} :: \mathit{State} \to \mathit{Test} \to \mathit{Bool}\\
&\mathit{optimal}\ (\mathit{Pair}\ u\ g)\ (\mathit{TPair}\ (a,b)\ (ab,0))\\
&\quad = (2\times a + b \leqslant p) \wedge (u - 2\times a - b \leqslant q)\\
&\qquad \textbf{where}\ p = 3\ \hat{}\ (t-1)\\
&\qquad\qquad q = (p-1)\ \text{`}\mathit{div}\text{`}\ 2\\
&\qquad\qquad t = \mathit{ceiling}\ (\mathit{logBase}\ 3\ (\mathit{fromIntegral}\ (2\times u + k)))\\
&\qquad\qquad k = \textbf{if}\ g == 0\ \textbf{then}\ 2\ \textbf{else}\ 1\\
&\mathit{optimal}\ (\mathit{Triple}\ l\ h\ g)\ (\mathit{TTrip}\ (a,b,c)\ (d,e,f))\\
&\quad = (a+e)\ \text{`}\mathit{max}\text{`}\ (b+d)\ \text{`}\mathit{max}\text{`}\ (l-a-d+h-b-e) \leqslant p\\
&\qquad \textbf{where}\ p = 3\ \hat{}\ (t-1)\\
&\qquad\qquad t = \mathit{ceiling}\ (\mathit{logBase}\ 3\ (\mathit{fromIntegral}\ (l+h)))
\end{aligned}
$$

17. Define a function

    $$\mathit{bestTests} :: \mathit{State} \to [\,\mathit{Test}\,]$$

    that returns only the optimal tests among the valid weighings for a state.

18. Hence define a function

    $$\mathit{mktreeG} :: \mathit{State} \to \mathit{TreeH}$$

    to build a *TreeH* greedily, using only optimal tests. Since they are optimal, you can pick any of them—for example, just pick the first—and don't need to try them all and choose the best. This is much better:

    ```
    *> mktreeG (Pair 8 0)
    (0.01 secs, 782,792 bytes)
    *> mktreeG (Pair 12 0)
    (0.01 secs, 1,387,368 bytes)
    ```

19. Define a function

    $mktreesG :: State \to [\, TreeH \,]$

    that explores all possible optimal tests, returning a list of trees. You can check that they do indeed all have minimum height:

    ```
    *⟩  nub (map heightH (mktreesG (Pair 12 0)))
    [3]
    ```

    (Recall that *Data.List.nub* removes duplicates from a list. Hence, all those trees have height 3.) How many minimum-height decision trees are there for $n = 12$?

7 Guidance

The purpose of the assignment is for you to demonstrate your understanding of functional programming. In principle, you can do that without solving all of the questions above; and there is no need for you to avail yourself of any of the hints, if you can see a better way to do it. But if you're not sure, I advise following the steps above.

You may be able to find partial solutions to the problems on the web. You are advised not to use them, or at least, don't rely on them: they are often of dubious quality, they are likely to implement slightly different specifications, they typically won't help your critical review, and most importantly they won't help you learn about functional programming. Whatever you do, do make sure you *make clear the source and the extent* of any derivative material.

Your answer should take the form of a literate Haskell script (or several such scripts, if you want to use modules). You should submit the code and commentary *interleaved, in the same file*: the commentary should take the form of an essay, with the code and examples as illustrations (like the course practicals, and this assignment itself). Submit the literate script in two formats: as plain ASCII text, so that it can be run, and as PDF, so that it is easy to print. You may use fancy formatting in the PDF if you wish, but there is no need to do so—straightforward conversion from ASCII to PDF is fine. If you do want to use fancy formatting, I recommend using `pandoc`—see below. Please do make sure that the two formats contain equivalent content, so that there is no need for me to compare them before marking them.

Finally, please structure the PDF file so that there is a cover sheet which contains only your name, the subject and date, and a note of the total number of pages. Do not put any answer material on the cover sheet; begin your answer on a fresh page. Avoid putting your name on any page except the cover page. Please number the pages and sections.

8 Pandoc

I use LaTeX to format my teaching materials (like this assignment), and I recommend it for complex documents like your dissertation. But it's rather a sledgehammer for a simpler document such as this assignment, and for such purposes I recommend a tool called `pandoc`. This is an open-source set of translators between different document formats, including Markdown, HTML, Word, and LaTeX. Moreover, it is written in Haskell! In particular, you could write your assignment using Markdown format, which is simultaneously valid (and legible) literate Haskell; and generate the PDF for submission directly from this. I will make an example available on SSTL.

9 Assessment criteria

The assignment is intended to determine, in order of decreasing importance:

- have you understood the fundamental tools of functional programming: algebraic data types, pattern matching, higher-order functions, parametric polymorphism, and lazy evaluation?
- have you demonstrated an ability to apply these fundamental tools and accompanying techniques to a particular case study?
- do you have the ability to present clear arguments supporting design decisions and discussing trade-offs, concisely and precisely?
- how fluent is your expression in Haskell, and how elegant is the resulting code?