*JFP* **29**, e8, 9 pages, 2019.  
doi:10.1017/S0956796819000030

# FUNCTIONAL PEARL

# *How to find a fake coin*

RICHARD S. BIRD
*Department of Computer Science, Oxford University*
*Wolfson Building, Parks Road, Oxford, OX1 3QD, UK*
(*e-mail:* bird@cs.ox.ac.uk)

## 1 Introduction

The aim of this pearl is to solve the following well-known problem that appears in many puzzle books, for example Levitin & Levitin (2011) and Bellos (2016), usually for the particular case $n = 12$.

> There are $n$ coins, all identical in appearance, one of which may be fake. The fake coin, if it exists, is either lighter or heavier than the fair coins, but it is not known which, nor by how much. Given is a balance with two pans but no weights. Equal number of coins can be placed on each pan and weighed. There are three possible outcomes: the left-hand pan may be lighter than the right-hand pan, or of equal weight, or heavier. Design a recipe for determining the minimum number of weighings guaranteed to determine the fake coin, if it exists, and whether it is lighter or heavier than the others.

The problem is not solvable if there is only one coin for there is no other coin to weigh it against. If there are two coins, then weighing one against the other will determine whether or not one is fake but not which. So the problem is not solvable in the case $n = 2$ either and we will assume $n \geq 3$.

What is required as the recipe is a decision tree of minimum height. Internal nodes of the tree are labelled with tests, where a test consists of a single weighing with equal numbers of coins on both pans. Each internal node has three children, corresponding to whether the test resulted in the left-hand pan being lighter, of equal weight, or heavier than the right-hand pan. The tree also has leaf nodes that identify the fake coin, if any, and say whether it is lighter or heavier.

## 2 States and tests

The first task is to formulate a suitable representation for any current state of knowledge about the coins. How would you do it? Pause for a minute to think about this.

---

One obvious first step is to name the coins, say from 1 to $n$. Initially nothing is known about the coins, so the initial state (for $n = 6$) would be something like

$$\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \pm & \pm & \pm & \pm & \pm & \pm \end{array}$$

where the classifier $\pm$ signifies that the coin may be lighter or heavier than the rest. After weighing, say coins $\{1, 2\}$ in the left pan against $\{3, 4\}$ in the right pan, the three possible outcomes would be

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| left pan lighter | $-$ | $-$ | $+$ | $+$ | $=$ | $=$ |
| balanced | $=$ | $=$ | $=$ | $=$ | $\pm$ | $\pm$ |
| left pan heavier | $+$ | $+$ | $-$ | $-$ | $=$ | $=$ |

where $-$ signifies possibly lighter, $+$ possibly heavier and $=$ known to be fair. This representation is basically the one used in Levitin & Levitin (2011). An equivalent way of describing states is by three sets $L$, $H$ and $F$ of possibly light, possibly heavy and known to be fair coins. In a final state, either both of $L$ and $H$ will be empty, signifying that all coins are fair, or one will be empty and the other will contain a single coin, the fake one.

However, there is a simpler representation and choosing it is key to finding a solution fairly quickly. The point to realise is that the coins are *anonymous* and do not have to be named. Initially there is just a pile of $n$ coins. After the first test, say matching $k$ coins with $k$ other coins (so $2k \leq n$), there will either be two or three piles of coins on the table. The first case arises when the pans balance. There are now $n - 2k$ unknown coins and $2k$ coins known to be fair. If the weighing shows the left pan to be lighter, then there are three piles: a pile $L$ consisting of $k$ coins, a similar pile $H$ of $k$ coins and a pile $F$ of $n - 2k$ coins. It is the number of coins in each pile that is important, not their names. At the end of the process there will still be three piles, but at most one coin in the first two. That coin, if there is one, is known to be lighter if it is in the $L$ pile and heavier if it is in the $H$ pile.

Given that the initial state can be represented by two piles, one containing $n$ coins and the other containing 0 coins, we can formulate a state to be

```
data State = Pair Int Int | Trip Int Int Int
```

with `Pair u r` signifying a state in which there are $u$ unknown coins and $r$ fair coins, and with `Trip l h r` signifying a state in which there are $l$ coins in the $L$ pile, $h$ coins in the $H$ pile and $r$ coins in the $F$ pile.

Here, for example, is a list of possible final states when there are six coins:

```
Trip 1 0 5   Trip 0 1 5   Pair 0 6
```

The final state `Trip 0 0 6` is impossible: as we will see, a `Trip` state is introduced only when the scales fail to balance, so a fake coin is known to exist. In fact, any `Trip` state whose first two components are zero is impossible for exactly the same reason.

Next we turn to the representation of tests. Here we can only weigh some coins against an equal number of other coins. There are two kinds of test; one is when there are two piles of coins on the table and we have to select two pairs of numbers, one pair for each of the pans. The other kind is when there are three piles on the table where we have to select a triple of numbers, again one triple for each pan. Hence we define

https://doi.org/10.1017/S0956796819000030 Published online by Cambridge University Press

```
data Test = T1 (Int,Int) (Int,Int)
          | T2 (Int,Int,Int) (Int,Int,Int)
```

A test `T1 (a,b) (c,d)` in state `Pair u r` denotes a weighing of $a$ coins from the $u$ unknowns and $b$ coins from the $r$ knowns against $c$ coins from the $u$ unknowns and $d$ coins from the $r$ knowns. So we have $a+b=c+d$, $a+c\leq u$ and $b+d\leq r$. Similarly, a test `T2 (a,b,c) (d,e,f)` in a state `Trip l h r` denotes a weighing of $a$ coins from $l$ possibly light coins, $b$ coins from $h$ possibly heavy coins and $c$ coins from the $r$ fair coins against $d$, $e$ and $f$ coins from the same three piles. Hence $a+b+c=d+e+f$, $a+d\leq l$, $b+e\leq h$ and $c+f\leq r$. The main constraint on tests is that the sum of the numbers of coins in each pan is the same. Other constraints will follow shortly—and even cleverer constraints in the final section.

As a next step we will construct a function

```
outcomes :: State -> Test -> [State]
```

for determining the three possible outcomes of a test in a given state, represented as a list rather than a triple: The first clause is

```
outcomes (Pair u r) (T1 (a,b) (c,d)) =
 [Trip a c (r+u-a-c), Pair (u-a-c) (r+a+c), Trip c a (r+u-a-c)]
```

The first outcome is when the left pan is lighter, the second is when they balance and the third is when the right pan is lighter. In the first case, one of $a$ is light or one of $c$ is heavy; dually for the third case. If the two pans balance, then $a+c$ coins are known to be fair and can be removed from the unknown pile. The second clause is

```
outcomes (Trip l h r) (T2 (a,b,c) (d,e,f)) =
   [Trip a e (r+l-a+h-e),
    Trip (l-a-d) (h-b-e) (r+a+d+b+e),
    Trip d b (r+l-d+h-b)]
```

If the left pan is lighter, then one of $a$ is light or one of $e$ is heavy; dually for the case when the left pan is heavier. If both pans balance, then $a+d$ coins can be removed from the potentially light group and $b+e$ coins from the potentially heavy group.

The next task is to construct a function `tests :: State -> [Test]` for determining the possible tests in a given state. Here is your second question: apart from the obvious fact that there have to be enough coins in each pile to make the necessary selections, can you think of any other straightforward constraints on the kinds of test to be carried out? Later on we will devise clever tests, but for now we only want to omit tests that are uninformative and pointless. Again, pause for a while to think about this.

---

One obvious constraint is that it is never necessary to have fair coins on both pans: if one pan has $a$ fair coins and the other pan $b$ fair coins, the same outcome would result in having $a-b$ fair coins on the first pan (if $a>b$) and 0 fair coins on the other.

Another obvious constraint is that the two pans, like the coins, are also anonymous, so there is no point in having both the tests

https://doi.org/10.1017/S0956796819000030 Published online by Cambridge University Press

```
T2 (a,b,c) (d,e,f)    and    T2 (d,e,f) (a,b,c)
```

We can maintain anonymity by constructing tests only for $(a, b, c) \leq (d, e, f)$, that is to say, the contents of the left pan is lexicographically no greater than those of the right pan.

A third constraint is that there is no point in having a test that is not guaranteed to reduce uncertainty. For example, consider the state `Trip 3 0 6`. Here the test `T2 (0,0,3) (3,0,0)` is not a sensible one. With this test, the three outcomes are

```
[Trip 0 0 9,Trip 0 0 9,Trip 3 0 6]
```

The first two represent impossible outcomes. How can three fair coins be lighter or of equal weight than three coins, one of which is known to be light? The third outcome is possible, but it is exactly the same state as we started with, so the test is pointless.

The above considerations suggest defining

```
tests :: State -> [Test]
tests st = [t | t <- weighings st, better (outcomes st t) st]
```

where

```
better :: [State] -> State -> Bool
better sts st = and [less st' st | st' <- sts]

less :: State -> State -> Bool
less (Pair a b) (Pair c d)     = a<c
less (Trip a b c) (Pair d e)   = True
less (Trip a b c) (Trip d e f) = a+b < d+e
```

It remains to construct `weighings`. Suppose the argument is `Pair u r`. We want to construct a left pan $(a, b)$ and a right pan $(c, d)$ so that five conditions are satisfied:

$$\begin{aligned} &a + b = c + d \qquad (\neq 0)\\ &b \times d = 0\\ &a + c \leq u\\ &b + d \leq r\\ &(a, b) \leq (c, d) \end{aligned}$$

In words, an equal number, not zero, of coins in each pan, not both pans containing fair coins, a sufficient number of fair and unknown coins for the test, and a tie-breaking condition to avoid duplication. If we take $b = 0$, then the last condition is effectively $(a, 0) \leq (a - d, d)$ which is satisfiable only if $d = 0$ as well. So we take $d = 0$ and look for pairs $(a, b)$ and $(a + b, 0)$, where $a + b \neq 0$, $2a + b \leq u$ and $b \leq r$. That all leads to

```
weighings :: State -> [Test]
weighings (Pair u r) =
 [T1 (a,b) (a+b,0) | a <- [0 .. u `div` 2],
                     b <- [0 .. min r (u-2*a)], a+b /= 0]
```

For example, with three unknowns and two fair coins, the sensible tests are

```
[T1 (0,1) (1,0), T1 (0,2) (2,0), T1 (1,0) (1,0), T1 (1,1) (2,0)]
```

https://doi.org/10.1017/S0956796819000030 Published online by Cambridge University Press

Suppose now we have three piles containing $l$, $h$ and $r$ coins, respectively. We want a test involving two triples $(a, b, c)$ and $(d, e, f)$ of nonnegative numbers such that the following six conditions hold:

$$\begin{aligned} a+b+c &= d+e+f \qquad (\neq 0)\\ c \times f &= 0\\ a+d &\leq l\\ b+e &\leq h\\ c+f &\leq r\\ (a,b,c) &\leq (d,e,f) \end{aligned}$$

One way to thread through these constraints and generate the valid triples via a list comprehension is to define a subsidiary function `choices` so that for a given $k$ and state $(l, h, r)$, all possible selections $(i, j, k-i-j)$ involving $k$ coins are returned. That means $0 \leq i \leq l$, $0 \leq j \leq h$ and $0 \leq k-i-j \leq r$. These conditions imply $k-i-r \leq j \leq k-i$, so we can define

```
choices :: Int -> (Int,Int,Int) -> [(Int,Int,Int)]
choices k (l,h,r) =
 [(i,j,k-i-j) | i <- [0..l], j <- [max 0 (k-i-r).. min (k-i) h]]
```

Now we can define

```
weighings (Trip l h r) =
    [T2 (a,b,c) (d,e,f)
    | k <- [1..(l+h+r) `div` 2],
      (a,b,c) <- choices k (l,h,r),
      (d,e,f) <- choices k (l-a,h-b,r-c),
      0<d+e, c*f==0, (a,b,c) <= (d,e,f)]
```

There are at most $\lfloor (l+h+r)/2 \rfloor$ choices for $k$. The last three conditions assert that there is at least one unknown on the right pan, that there are no fair coins on both pans, and symmetry is broken. For example, for `Trip 2 2 2` there are 16 possible weighings, only one of which, `T2 (0,2,0) (2,0,0)`, is rejected as a test because the outcome does not reduce uncertainty.

A methodological point is worth emphasising: the success of this representation is due solely to the refusal to name things that do not have to be named, a point frequently made by Edsger Dijkstra in his many writings.

## 3 Trees

It is high time to introduce decision trees. We will want a tree whose height is minimum and, to save recomputing heights, we will add in height information at each node. Here is the type of decision trees:

```
data Tree = Stop State | Node Int Test [Tree]
```

https://doi.org/10.1017/S0956796819000030 Published online by Cambridge University Press

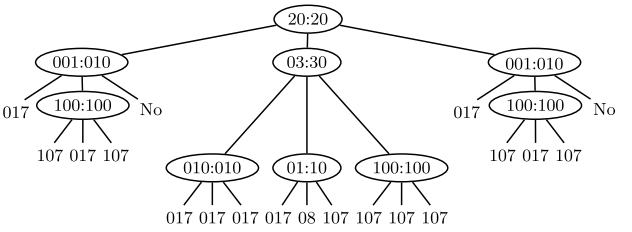


Fig. 1. A decision tree for eight coins.

Each `Stop` node is a leaf that represents a possible final outcome. Each `Node` has exactly three offspring, represented as a list of three trees instead of a triple. We define

```
hgt :: Tree -> Int
hgt (Stop _)     = 0
hgt (Node h _ _) = h
```

and construct trees via the smart constructor

```
node :: Test -> [Tree] -> Tree
node t ts = Node h t ts   where h = 1 + maximum (map hgt ts)
```

Finally, we are in a position to solve our problem by brute force. Since a tree of minimum height can be constructed from subtrees with minimum height, we have

```
mktree :: State -> Tree
mktree st =
 if final st then Stop st
 else best [node t (map mktree (outcomes st t)) | t <- tests st]
```

where

```
final :: State -> Bool
final (Trip l h r) = l+h <= 1
final (Pair u r)   = u==0
```

and

```
best :: [Tree] -> Tree
best = foldr1 better
better t1 t2 = if hgt t1 <= hgt t2 then t1 else t2
```

It is a tribute to the expressive power of Haskell, especially its provision of list comprehensions, that enables a solution to the problem to be constructed so easily.

Computing `mktree (Pair n 0)` works well enough for small values of $n$. For example, it took about seven seconds with GHCi (the Glasgow Haskell Compiler Interpreter) to compute the tree of Figure 1 for $n = 8$, in which the tests have been shown in a simplified form. There are two aspects of this tree worthy of note. Firstly, the left subtree is exactly the same as the right subtree. This is because both trees deal with the state `Trip 2 2 4`. Nevertheless, both trees are computed from scratch, wasting computation. The second

https://doi.org/10.1017/S0956796819000030 Published online by Cambridge University Press

point is that in two places there is a test with apparently the same outcomes. For example, the bottom left tree has the three outcomes `Stop (Trip 0 1 7)`. So why is the test necessary? The answer is that the three outcomes are actually quite different. The current state before the test is `Trip 0 3 5` and if we name the potentially heavy coins as $a$, $b$ and $c$, and $a$ is weighed against $b$, then the first outcome asserts that $b$ is the heavy coin, the second that $c$ is and the third that $a$ is.

For larger values of $n$, the computation slows down to an unacceptable crawl and we need something better. One standard method for dealing with a recursive function like `mktree` that computes the solution to subproblems more than once is to employ dynamic programming, either with a top-down memoization scheme or a bottom-up tabulation scheme. However, there is another path for improving the performance of `mktree`.

## 4 An improved algorithm

The problem with `mktree` is that a large number of tests are considered at each step. The alternative, if it can be made to work, is to construct only the best tests, tests that will guarantee a tree of minimum height. The improved algorithm will take the form

```
mktree :: State -> Tree
mktree st = if final st then Stop st
            else node t (map mktree (outcomes st t))
            where t = choose (bestTests st)
```

where `choose` selects an arbitrary test; for example, `choose = head`. The revised definition of `mktree` still computes solutions to subproblems more than once and can benefit from memoization, but we will leave that particular optimisation aside and concentrate on the more immediate question: can we construct `bestTests`?

To answer the question we need to delve deeper into the arithmetic of the problem. With a state `Trip l h r` we know there are $l + h$ possible final outcomes: either one of $l$ is light or one of $h$ is heavy. They can't all be fair because a `Trip` state is only introduced after a weighing that has seen the two pans fail to balance. Now with $t$ tests there are at most $3^t$ possible outcomes since each node has only three offspring. In particular, any tree for `Trip 2 2 r` has height at least two because there are four possible final outcomes.

On the other hand, with a state `Pair u r` there are $2u + 1$ possible final outcomes: either one of the $u$ coins is light, or one is heavy or all are fair. In particular, the state `Pair 5 r` cannot be solved in two tests because there are 11 possible outcomes. More subtly, `Pair 4 r` can be done in two tests if $r \neq 0$, but not if $r = 0$. In the second case there are only two possible tests, `T1 (1,0) (1,0)` and `T1 (2,0) (2,0)`; with the first test and a balanced outcome there are five possible final outcomes, while with the second and an unbalanced outcome there are four possible final outcomes. Neither of these can be resolved with a single additional test.

We can generalise the above information with the following claim:

**Claim:** The states that can be solved with at most $t$ tests are `Trip l h r` with $l + h \leq 3^t$, and `Pair u r` with $r \neq 0 \wedge 2u + 1 \leq 3^t$, or $r = 0 \wedge 2u + 2 \leq 3^t$. Equivalently, a problem

https://doi.org/10.1017/S0956796819000030 Published online by Cambridge University Press

`Trip l h r` can be solved with $\lceil \log_3(l+h) \rceil$ tests, and `Pair u r` can be solved with $\lceil \log_3(2u+2) \rceil$ tests if $r = 0$ or $\lceil \log_3(2u+1) \rceil$ tests if $r \neq 0$.

We have already seen from above that this number of tests is a lower bound, and the proof is completed by showing by direct construction that every state satisfying the constraints can be solved with the given number of tests, thus establishing that it is also an upper bound.

Let us concentrate on defining `bestTests (Pair u r)` first. Since every test returned by `weighings (Pair u r)` is of the form `T1 (a,b) (a+b,0)` and since

```
outcomes (Pair u r) (T1 (a,b) (a+b,0))
  = [Trip a (a+b) (r+u-2*a-b),
     Pair (u-2*a-b) (r+2*a+b),
     Trip (a+b) a (r+u-2*a-b)]
```

it follows that we have to seek two pans $(a, b)$ and $(a+b, 0)$ such that

$$2a + b \leq 3^{t-1} \wedge u - 2a - b \leq (3^{t-1} - 1)/2$$

where $t = \lceil log_3(2u+1) \rceil$ if $r \neq 0$ or $t = \lceil log_3(2u+2) \rceil$ if $r = 0$. Hence we can define

```
bestTests (Pair u r) =
 [T1 (a,b) (a+b,0) | a <- [0 .. u `div` 2],
                     b <- [0 .. min r (u-2*a)],
                     a+b /= 0, 2*a+b <= p, u-2*a-b <= q]
 where p = 3^(t-1)
       q = (p-1) `div` 2
       t = ceiling (logBase 3 (fromIntegral (2*u+k)))
       k = if r==0 then 2 else 1
```

For `bestTests (Trip l h r)` we require two pans $(a, b, c)$ and $(d, e, f)$ for which the following constraint is satisfied in addition to those provided by `weighings`:

$$a + e \leq 3^{t-1} \ \wedge \ b + d \leq 3^{t-1} \ \wedge \ l - a - d + h - b - e \leq 3^{t-1}$$

where $t = \lceil \log_3(l+h) \rceil$. That leads to

```
bestTests (Trip l h r)
 = [T2 (a,b,c) (d,e,f)
   | T2 (a,b,c) (d,e,f) <- weighings (Trip l h r),
     (a+e) `max` (b+d) `max` (l-a-d+h-b-e) <= p]
 where p = 3^(t-1)
       t = ceiling (logBase 3 (fromIntegral (l+h)))
```

To test the claim we can construct a function that finds all the best solutions:

```
findAllBest :: State -> [Tree]
findAllBest st
  | final st  = [Stop st]
  | otherwise = [node t ts
                | t <- bestTests st,
                  ts <- cp (map findAllBest (outcomes st t))]
```

https://doi.org/10.1017/S0956796819000030 Published online by Cambridge University Press

where the Cartesian product function `cp` is defined by

```
cp :: [[a]] -> [[a]]
cp []       = [[]]
cp (xs:xss) = [y:ys | y <- xs, ys <- yss]
              where yss = cp xss
```

Now we have, under GHCi:

```
> all (==3) $ map hgt $ findAllBest (Pair 12 0)
True
```

This shows that any choice of a best test at each step leads to a tree of height 3 for the 12 coins problem. As to the number of possible best trees, we have

```
> length $ findAllBest (Pair 12 0)
40500
```

The puzzle books do not mention that there are 40500 solutions to choose from.

**Acknowledgments**

Thanks to Jeremy Gibbons and the referees for making numerous valuable suggestions for polishing the pearl.

**References**

Bellos, A. (2016) *Can You Solve My Problems?* Guardian Books: London, UK.
Levitin, A. & Levitin, M. (2011) *Algorithmic Puzzles*. Oxford University Press.