ECM2418 Computer Languages and Representations

D. Wakeling

Laboratory 5

Hymn Number Anagrams

Every week, *The Sunday Times* newspaper publishes a Teaser. Teaser 3160 was as follows.

> At Church on Sunday, the hymn board showed just four different hymn numbers, each having the same three, different, non zero digits, but in a different order, I noticed that the first hymn number was a perfect square, and that there was at least one other perfect square, formed from the same three digits, which did not appear on the board. The sum of the four hymn numbers was a prime number. If I told you the first digit of that prime number, you would be able to tell me the first hymn number of the board.
>
> **What was the first hymn number on the board?**

Question 1.1

The plan, or *design pattern*, for solving Teasers is always the same: generate a list of all of the values that *might* be solutions to the Teaser, and then filter this list for values that *are* solutions to the Teaser [1]. This leads immediately to an important *separation of concerns*, and to the program

```
import Data.List

main :: IO ()
main =
  print (filter tester generator)
```

As a first step towards the `generator`, define the auxiliary function `special` that is true for strings that do not contain the character '`0`', or any duplicate character, so

```
special "123"
   ===> True
```

The `generator` must produce a list of tuples of hymn numbers $(S1, S2, S3, S4)$, that might be solutions to the Teaser. See Figure 1. The hymn number $S1$ must be special.

S1	2	5	6
S2	2	6	5
S3	5	2	6
S4	5	6	2

Figure 1: A hymn board.

All four hymn numbers, $S1$, $S2$, $S3$ and $S4$ must be different, but must have the same three different digits, something that is most easily checked with a string representation

```
generator :: [(String,String,String,String)]
```

This function will be assessed by the number of tests that it passes, as counted by the `x_generator` function below. The expected answer is 10.

```
x_generator :: Int
x_generator =
  length [t | t <- ts, t `elem` g]
  where
  g  = generator
  ts =
     [ ("123","213","321","231")
     , ("168","618","861","186")
     , ("236","326","263","623")
     , ("284","824","482","842")
     , ("351","531","153","315")
     , ("397","937","379","739")
     , ("467","647","764","674")
     , ("524","254","425","542")
     , ("581","851","518","158")
     , ("639","369","936","396")
     ]
```

(10 marks)

Question 1.2

As a first step towards the `tester`, define the auxiliary function `perfectSquare` that is true for integers that are perfect squares, so

```
perfectSquare 256
    ===> True
```

As a second step towards the `tester`, define the auxiliary function `prime` that is true for integers that are prime numbers, so

```
prime 31
    ===> True
```

The `tester` is true for tuples of hymn numbers $(S1, S2, S3, S4)$ that might be solutions to the Teaser. That is, for which $S1$ is a perfect square, the sum of $S1$, $S2$, $S3$ and $S4$ is prime, and there is at least one other perfect square formed from the digits of $S1$ that are not $S1$, $S2$, $S3$ and $S4$ on the board.

This function will be assessed by the number of tests that it passes, as counted by the `x_tester` function below. The expected answer is 10.

```
x_tester :: Int
x_tester =
  length [t | t <- ts, tester t]
  where
  ts =
     [ ("196","691","961","619")
     , ("196","619","691","961")
     , ("256","526","652","265")
     , ("256","526","562","265")
     , ("256","652","265","526")
     , ("961","691","619","196")
     , ("961","691","196","619")
     , ("961","196","619","691")
     , ("961","196","691","619")
     , ("961","916","619","691")
     ]
```

(10 marks)

Question 1.3

On `blue18.ex.ac.uk`, my program computes the result

```
[ ("196","691","961","619") – 2467
, ("196","691","619","961") – 2467
, ("196","961","691","619") – 2467
, ("196","961","619","691") – 2467
, ("196","619","691","961") – 2467
, ("196","619","961","691") – 2467
, ("256","526","652","265") – 1699
, ("256","526","562","265") – 1609
, ("256","526","265","652") – 1699
, ("256","526","265","562") – 1609
, ("256","652","526","265") – 1699
, ("256","652","265","526") – 1699
, ("256","562","526","265") – 1609
, ("256","562","265","526") – 1609
, ("256","265","526","652") – 1699
, ("256","265","526","562") – 1609
, ("256","265","652","526") – 1699
, ("256","265","562","526") – 1609
, ("961","691","619","196") – 2467
, ("961","691","619","916") – 3187
, ("961","691","196","619") – 2467
, ("961","691","916","619") – 3187
, ("961","619","691","196") – 2467
, ("961","619","691","916") – 3187
, ("961","619","196","691") – 2467
, ("961","619","916","691") – 3187
, ("961","196","691","619") – 2467
, ("961","196","619","691") – 2467
, ("961","916","691","619") – 3187
, ("961","916","619","691") – 3187
]
```

in 0.004 seconds. On the same machine, can you tune your program so that it computes this answer within 1.0 seconds?

Note that I have been unable to deduce the stated answer `961` from the result above. Thoughts?

(5 marks)

References

[1] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, January 1989.