

# Solving an Euler Project Problem

Henry Laxen

April 29, 2008

Warning! some of what you are about to see may be offensive to accomplished programmers. The end results you see in textbooks and on the Euler Project forum are *polished*, and do not necessarily reflect the blind alleys and false starts that people go through to arrive at the final result. The purpose of this paper is to travel down the path towards a solution, knowing full well that some approaches are doomed to failure.

For completeness I'll restate the problem here briefly. The problem is to count trinary strings of length  $k$  subject to the following rules:

1. Strings consist of the characters O, L and A
2. Strings may contain at most one L, anywhere.
3. Strings must not end with more than 2 A's

These strings are described in the problem as award strings, where O represents a student who attended class On time, L a student who came Late, and A a student who was Absent entirely. Here is a list of length 3 strings that satisfy the above conditions:

## Valid Strings of Length 3

---

AAL AAO ALA ALO AOA AOL AOO LAA LAO LOA  
LOO OAA OAL OAO OLA OLO OOA OOL OOO

The rules are simple enough, so lets attack the problem by starting there. We'll implement the rules as a filter over all possible strings of length  $k$ . Of course this can't work for long, for  $3^k$  grows, well exponentially. But at least it gives us a starting point to *play* with the problem. So I proceeded:

## Haskell Code

---

```
1
2 > import Data.List
3 > import Text.PrettyPrint
4 > import qualified Data.Set as S
```

```

5 > data Attendance = A | L | O deriving (Eq,Ord,Show)
6 >
7 > valid l =
8 >   let late = 1 >= (length $ filter (==L) l)
9 >       absent = not (any (\x -> A 'elem' x && length x >=3) (group l))
10 >   in (late && absent)
11 >
12 > combos = (inits . repeat) [A,L,O] >>= sequence
13 >
14 > cc n = filter valid $
15 >   takeWhile (\x -> length x == n) $ dropWhile (\x -> length x < n) combos

```

---

Now running cc 3 produces:

### Haskell Code

---

```

17     [[A,A,L],[A,A,O],[A,L,A],[A,L,O],[A,O,A],[A,O,L],[A,O,O],
18     [L,A,A],[L,A,O],[L,O,A],[L,O,O],[O,A,A],[O,A,L],[O,A,O],
19     [O,L,A],[O,L,O],[O,O,A],[O,O,L],[O,O,O]]

```

---

all wrapped together on one line, ugly, very ugly. Which brings me to one of the best and most overlooked development tools, namely the Text.PrettyPrint library. I added the following:

### Haskell Code

---

```

20
21
22 > lp n x d = if null a2 then render c else lp n a2 c
23 >   where
24 >     (a1,a2) = splitAt n x
25 >     b = map (text . concatMap show) a1
26 >     c = d $$ hsep b
27 > s l = putStrLn $ (lp 10 l empty) ++ "\n"

```

---

So now running s \$ cc 3 results in:

```

AAL AAO ALA ALO AOA AOL AOO LAA LAO LOA
LOO OAA OAL OAO OLA OLO OOA OOL OOO

```

Which is much easier to stare at. Well, maybe we'll get lucky, so let's try running cc over a bunch of values, counting the lengths and looking at the result.

### Haskell Code

---

```

29     map (\x -> length (cc x)) [3..10]
30     [19,43,94,200,418,861,1753,3536]

```

---

No obvious pattern, and least none that my brain can conjure up. Looks like we'll actually have to do some work. Let's define some functions that filter the output in various pieces, like those strings that contain no Ls or only one L, or not ending in one A or two As.

Wait a minute! How do you get from a string of length (n-1) to one of length n? You have to append either an O, an L or an A. Now you can append an O to anything, you can append an L to a string that doesn't contain an L, and you can append an A to a string that doesn't end in two A's. So we should have:

award n = award (n-1) plus an **O**  
+ noLs (n-1) plus an **L**  
+ notTwoAs (n-1) plus and **A**

## Haskell Code

---

```
31
32 > noL = filter (\x -> notElem L x)
33 > oneL = filter (\x -> elem L x)
34 > notOneA = filter (\x -> last x /= A)
35 > notTwoA = filter (\x -> take 2 (reverse x) /= [A,A])
36 > addX x 1 = map (\z -> z ++ [x]) 1
```

---

Let's play with these functions a while, and check out our analysis above:

```
*Main> s $ cc 4
AALA AALO AAOA AAOL AAOO ALAA ALAO ALOA ALOO AOAA
AOAL AOAOL AOLA AOLO AOOA AOOL AOOO LAAO LAOA LAOO
LOAA LOAO LOOA LOOO OAAO OALO OAOA OAOL
OAOO OLAA OLAO OLOA OLOO OAAA OOAL OOOA OOLA OOOO
OOOA OOOL OOOO

*Main> s $ addX O $ cc 3
AALO AAOO ALAO ALOO AOAOL AOLO AOOO LAAO LAOO LOAO
LOOO OAAO OALO OAOO OLAO OLOO OOOA OOOO

*Main> s $ addX L $ noL $ cc 3
AAOL AOAL AOOL OAAL OAOL OOAL OOOO

*Main> s $ addX A $ notTwoA $ cc 3
AALA AAOA ALAA ALOA AOOA AOLA AOOA LAAO LOAA LOOA
OALA OAOA OLAA OLOA OAAA OOLA OOOA
```

Well, the count is right, but to really make sure I'm not confused, I should write something to make sure that the union of the three lower sets equals the top set. We need a little more code.

## Haskell Code

---

```
38
39 > checkUnions n =
40 >   let c1 = cc n
41 >       c2 = S.fromList (cc (n+1))
42 >       c10 = addX O c1
43 >       c1A = addX L (noL c1)
44 >       c1L = addX A (notTwoA c1)
45 >       cU = S.unions $ map S.fromList [c10,c1A,c1L]
46 >   in S.difference c2 cU
```

---

A quick run of `checkUnions n` for small `n` (3,4,5,6) always returns the empty set, so this confirms our thinking above. Now we need to figure out how to count the size of these sets without generating them. So far we have figured out how to count `award (n)` in terms of `award (n-1)` and two as yet unknown functions, `noLs` and `notTwoAs`. We need to look at those now. Let's start with `notTwoAs`,

which is a string that doesn't end with two As. It can end with either an O, an A, or an L. Now any award (n-1) string to which we append an O will be a notTwoAs (n) string, since it doesn't end with two As. Also, any string that contains no Ls can have an L appended to it, and will become a string that doesn't end in two As. Finally, string that doesn't end with a single A can have an A appended to it and will become a string that doesn't end with two As. Expressing this in code we have:

### Haskell Code

---

```
49
50 > award1 3 = 19
51 > award1 n = award1 (n-1) -- + 0
52 >           + noLs (n-1) -- + L
53 >           + notTwoAs (n-1) -- + A
54 >
55 > notTwoAs 3 = 17
56 > notTwoAs n = notOneAs (n-1) -- + A
57 >           + award1 (n-1) -- + 0
58 >           + noLs (n-1) -- + L
```

---

Well, we solved one problem and created another, for we expressed our now known function notTwoAs in terms of a new unknown function notOneAs. Slog on.

### Haskell Code

---

```
61
62 > notOneAs 3 = 12
63 > notOneAs n = award1 (n-1) -- + 0
64 >           + noLs (n-1) -- + L
65 > --           + 0 since it can't end in an A
66 >
67 > noLs 3 = 7
68 > noLs n = noLs (n-1) -- + 0
69 >           + noLsANDnotTwoAs (n-1) -- + A
70 > --           + 0 since it can't contain an L
```

---

Is there no end to this? We nailed down notOneA, but when computing noLs we had to add another unknown function, namely noLsANDnotTwoAs, which, you guessed it, counts strings that contain no Ls and do not end with two As. Slog on.

### Haskell Code

---

```
73
74 > noLsANDnotTwoAs 3 = 6
75 > noLsANDnotTwoAs n = noLs (n-1) -- + 0
76 >           + noLsANDnotOneA (n-1) -- + A
77 >
78 > noLsANDnotOneA 3 = 4
79 > noLsANDnotOneA n = noLs (n-1) -- + 0
```

---

Finally we have covered all of the cases. Now let's make sure they agree with counting the sets explicitly.

### Haskell Code

---

```

82
83 > test n =
84 >   let a = cc n
85 >   in [ length a == award1 n,
86 >       length (noL a) == noLs n,
87 >       length (notTwoA a) == notTwoAs n ]

```

---

If we now run `map test [3..10]` we are inspired with confidence, watch all those Trues written on the screen. We do notice a slight pause as it prints out the last list, however. This is a harbinger of things to come, for if we type `award 20` the answer 2947811 comes back in a couple of seconds. We have the same problem here as with the fibonacci series, namely an exponential increase in the number of terms we must compute. However since the answer we seek is for the value 30, and 20 is tractable, I am going to cheat by defining each of the functions above for the value 20, rather than memoizing everything.

### Haskell Code

---

```

89 award 20 = 2947811
90 notTwoAs 20 = 2504754
91 notOneAs 20 = 1649022
92 noLs 20 = 223317
93 noLsANDnotTwoAs 20 = 187427
94 noLsANDnotOneA 20 = 121415

```

---

Unfortunately, you have to go back and paste these values into the code above, in order for `ghci` to be happy, but if you do so you can easily compute the value of `award 30`.

Are we done, well yes and no. At this point we can notice that all of functions `noLs`, `notTwoAs`, etc. are really special cases of a function we could call `hasM_LsAndEndsInN_As m n k`, which takes the number of Ls and As it contains as a parameter. In fact it might even be easier to look at it this way. Our `award` code then becomes:

### Haskell Code

---

```

95
96 > award 1 = 3
97 > award k = award (k-1) -- + 0
98 >   + sum [ hasM_LsAndEndsInN_As 0 i (k-1) | i<-[0..2] ] -- +L
99 >   + sum [ hasM_LsAndEndsInN_As i j (k-1) | i<-[0,1], j<-[0,1] ] -- +A
100 >
101 > hasM_LsAndEndsInN_As 0 0 1 = 1 -- 0
102 > hasM_LsAndEndsInN_As 1 0 1 = 1 -- L
103 > hasM_LsAndEndsInN_As 0 1 1 = 1 -- A
104 > hasM_LsAndEndsInN_As _ _ 1 = 0
105 >
106 > hasM_LsAndEndsInN_As m n k
107 > | m < 0 || n < 0 = 0
108 > | n == 0 = sum [ hasM_LsAndEndsInN_As (m-1) i (k-1) | i<-[0..2] ] -- +L
109 >   + sum [ hasM_LsAndEndsInN_As m i (k-1) | i<-[0..2] ] -- +0
110 > | n > 0 = hasM_LsAndEndsInN_As m (n-1) (k-1) -- + A
111 >
112 > problem191 n = do
113 >   let p a b c d = "hasM_LsAndEndsInN_As " ++
114 >                   foldl (\x y -> x ++ (show y) ++ " ") "" [a,b,c] ++
115 >                   "= " ++ (show d)
116 >   putStrLn $ "award " ++ (show n) ++ " = " ++ show (award n)
117 >   mapM_ [(i,j) -> putStrLn $ p i j n | (i,j) <- [(i,j) | i<-[0..1], j<-[0..2]]]
118 >

```

---

Now having read the Euler Project Forum for this project, I can tell you that there are many far more elegant solutions to this problem, but my point is that by *playing* with the problem, and looking at the results of your play, you can often get a feeling for what is going on, and then come up with a strategy that leads to a solution, writing little tests along the way to confirm and disprove your conjectures. I hope this helps all of you future Euler Project Puzzlers on the next problem, whatever it is.

This file is also available in pdf form as well as plain text (markdown) for your reading pleasure



Henry Laxen