13 Countdown, the setup

Countdown is a daytime television 'quiz' programme, in which in one of rounds contestants are given six source numbers and a target, all positive integers. The aim is to find an arithmetic expression using some or all of the source numbers with a value as close as possible to the target. Each source number can be used at most once, expressions are built using only (+), (−), (∗), and (/), and all intermediate expressions have to be positive integers.

13.1 Abstract syntax trees

Expressions will be represented by trees

```
> data Expr = Val Int | App Op Expr Expr
> data Op = Add | Sub | Mul | Div
```

with all compound expressions represented by the same *App* node, labelled with its operator.

We will not use it here, but there would of course be a fold function

```
  fold :: (Int -> b) -> (Op -> b -> b -> b) -> Expr -> b
  fold val app = f where f (Val n)     = val n
                         f (App o l r) = app o (f l) (f r)
```

In the context of this problem it will make sense to *show* expressions with apparently unnecessary parentheses to reveal the structure.

The value of an expression, if it has one, is an *Int*.

```
> eval :: Expr -> [ Int ]
> eval (Val n)     = [ n | n > 0 ]
> eval (App o l r) =
>       [ apply o x y | x <- eval l, y <- eval r, valid o x y ]
```

The form of *eval* suggests *fold id apply* where

```
> apply :: Op -> Int -> Int -> Int
> apply Add = (+)
> apply Sub = (-)
> apply Mul = (*)
> apply Div = div
```

but the results have to pass a validity test, which is that every node in the evaluation satisfies

```
> valid :: Op -> Int -> Int -> Bool
> valid Add _ _ = True
> valid Sub x y = x > y
> valid Mul _ _ = True
> valid Div x y = x `mod` y == 0
```

ensuring that intermediate results are positive and divisions are exact.

We are using [*Int*] for the outcome instead of *Maybe Int* purely for the convenience of being able to use catenation later.

13.2 Checking solutions

A *Countdown* problem consists of some numbers *ns*, and a target number *n*; an expression *e* is a solution to that problem if the value of *e* is *n*, and the values that appear in *e* are chosen from *ns*.

```
> solution :: Expr -> [Int] -> Int -> Bool
> solution e ns n = (values e `elem` choices ns) && eval e == [n]
```

The values that appear in an expression can be found by flattening the expression

```
> values :: Expr -> [Int]
> values (Val n)     = [n]
> values (App _ l r) = values l ++ values r
```

which is again a fold on expressions.

The choices available from some list *xs* are subsequences of its permutations

```
> choices :: [a] -> [[a]]
> choices xs = [ ps | ys <- subs xs, ps <- permutations ys ]
```

The *permutations* function we have already seen; and subsequences can be defined by

```
> subs :: [a] -> [[a]]
> subs xs = [ y:zs | y:ys <- tails xs, zs <- [] : subs ys ]
```

in this problem, we only need non-empty subsequences, though including the empty one would not affect the value of *solution* because every result from *values* is non-empty.

There are 1956 permutations of non-empty subsequences of six numbers, so given a solution *e* it is straightforward to check that it is a *solution* to a given problem.

13.3 Generating solutions

The function *choices* already generates all permutations of subsequences of the available numbers. All expressions made up from those numbers can be constructed by building all possible trees on each one of these *choices*.

```
> exprs :: [Int] -> [Expr]
> exprs [n] = [Val n]
> exprs ns = [ App o l r | (ls,rs) <- split ns,
>                          l <- exprs ls,
>                          r <- exprs rs,
>                          o <- [ Add, Sub, Mul, Div ] ]
```

For non-leaves, the (non-empty) sequence of numbers must be split into two non-empty segments

```
> split :: [a] -> [([a],[a])]
> split   [_]  = []
> split (x:xs) = ([x],xs) : [(x:ls,rs) | (ls,rs) <- split xs ]
```

Each of these must be non-empty, so that the other is a strict subsequence of the whole.

13.4 Checking generated solutions

The solutions to a problem (ns, n) are those expressions e that can be generated from choices of ns and which have value n.

$$\mathit{solutions}\ \mathit{ns}\ n \quad = \quad [e \mid cs \leftarrow \mathit{choices}\ \mathit{ns}, e \leftarrow \mathit{exprs}\ \mathit{xs}, v \leftarrow \mathit{eval}\ e, v == n]$$

One way of organising this computation (guided by changes to be made later) is to collect all the expression-value pairs

```
> type Result = (Expr, Int)

> results :: [Int] -> [Result]
> results ns = [ (e, v) | e <- exprs ns, v <- eval e ]
```

and to filter these for whether they are solutions

```
> solutions :: ([Int] -> [Result]) -> [Int] -> Int -> [Expr]
> solutions results ns n
>     = [ e | cs <- choices ns, (e,v) <- results cs, v == n]
```

This completes a solver, which can (often) rapidly find a solution to a solvable problem,

```
*Lecture13> head (solutions results [1,3,7,10,25,50] 832)
7+(3*((1+10)*25))
(0.12 secs, 72,199,336 bytes)
```

but takes a very long time to search for a solution to a problem with no exact solution

```
*Lecture13> solutions results [1,3,7,10,25,50] 831
[]
(140.54 secs, 86,036,396,088 bytes)
```

because it explores more than thirty million possible expressions, more than four and a half million of which are valid.

13.5 Pruning the range of possible expressions

Since validity of an expression requires validity of its subsexpressions, there is no point constructing an expression out of invalid subexpressions and then checking for validity. A fusion of the test for validity with evaluation gives

```
> prunedresults :: [Int] -> [Result]
> prunedresults [n] = [(Val n, n) | n > 0 ]
> prunedresults ns = [ (App o l r, apply o x y)
>                    | (ls, rs) <- split ns,
>                      (l,x) <- prunedresults ls,
>                      (r,y) <- prunedresults rs,
>                      o <- [ Add, Sub, Mul, Div ],
>                      valid o x y ]
```

This program still constructs as many valid expressions, but dismisses the invalid ones more quickly

```
Lecture13> layn (solutions prunedresults numbers 832)
  1) 7+(3*((1+10)*25))
  2) 7+((3*(1+10))*25)
     ...
271) 50+((25*((10*3)+1))+7)
272) (50+(25*((10*3)+1)))+7
(8.17 secs, 4,335,352,752 bytes)
```

Even the pruned results include many expressions that add nothing to the existence of a solution: commutative operators mean that many subexpressions will appear both ways round with the same result. We could eliminate many expressions by requiring the values of the subexpressions in additions and multiplications to be ordered, and by eliminating multipication and division by one.

This could be done by strengthening the *valid* test to

```
> useful :: Op -> Int -> Int -> Bool
> useful Add x y = x <= y
> useful Sub x y = x > y
> useful Mul x y = x /= 1 && y /= 1 && x <= y
> useful Div x y = y /= 1 && x ‘mod‘ y == 0
```

Most of the solutions have been eliminated, leaving in this example fewer than a quarter of a million useful valid expressions, however there is guaranteed to be a remaining solution whenever there are any solutions.

```
Lecture13> layn (solutions usefulresults numbers 832)
  1) 7+(3*((1+10)*25))
  2) 7+((1+10)*(3*25))
  3) 7+(25*(3*(1+10)))
     ...
 13) 7+((50-25)*(3*(1+10)))
 14) 50+(7+(25*(1+(3*10))))
(1.39 secs, 744,579,248 bytes)
```

There remain many expressions that are equivalent up to the associativity of addition and multiplication. Eliminating wasted work here seems best achieved by generating only expressions in which each combination of associative operations appears in a canonical order.

This code

```
> gen :: [Op] -> [Int] -> [Result]
> gen ops [n] = [(Val n, n) | n > 0 ]
> gen ops ns = [ (App o l r, apply o x y)
>              | (ls, rs) <- split ns,
>                o <- ops,
>                (l,x) <- gen (left o) ls,
>                (r,y) <- gen (right o) rs,
>                useful o x y ]
```

restricts the operators that can appear to the left and right of an operation so that any sum is written as a list (represented as a right-spine tree) of factors, and any difference is a difference of sums. Similarly a quotient is a quotient of products each of which is a right-spine list of differences or sums.

```
> left, right :: Op -> [Op]
> left Add = [Mul,Div]
> left Sub = [Add,Mul,Div]
> left Mul = [Add,Sub]
> left Div = [Add,Sub,Mul]
> right Add = [Add,Mul,Div]
> right Sub = [Add,Mul,Div]
> right Mul = [Add,Sub,Mul]
> right Div = [Add,Sub,Mul]
```

Whilst this reduces the number of solutions

```
Lecture13> layn (solutions (gen [Add, Sub, Mul, Div]) numbers 832)
  1) 7+(3*((1+10)*25))
  2) 7+((1+10)*(3*25))
  3) 7+(25*(3*(1+10)))
     ...
 11) 7+((50-25)*(3*(1+10)))
 12) 50+(7+(25*(1+(3*10))))
(14.81 secs, 8,344,349,064 bytes)
```

it slows the process down considerably.

Why is this? Because of the order of the generators, the whole process of generating subtrees is repeated up to four time, for each *o* drawn from *ops*. This means that grandchildren are generated up to sixteen times, greatgrandchildren up to sixty-four times and so on. It turns out to be much cheaper to generate the subtrees once (independently of the operator at this node)

```
> gen' ops ns  = [ (App o l r, apply o x y)
>                | (ls, rs) <- split ns,
>                  ((l,x),(r,y)) <- gen' ops ls `cp` gen' ops rs,
>                  o <- ops,
>                  needed o l r,
>                  useful o x y ]
```

and to discard the ones not needed

```
> needed :: Op -> Expr -> Expr -> Bool
> needed Add (App Add _ _) _ = False
> needed Add (App Sub _ _) _ = False
> needed Add _ (App Sub _ _) = False
> needed Sub (App Sub _ _) _ = False
> needed Sub _ (App Sub _ _) = False
> needed Mul (App Mul _ _) _ = False
> needed Mul (App Div _ _) _ = False
> needed Mul _ (App Div _ _) = False
> needed Div (App Div _ _) _ = False
> needed Div _ (App Div _ _) = False
> needed _   _             _  = True
```