1 The problem

This is a problem 68 from Project Euler.

2 Definitions

```haskell
import Data.List (delete, (\\))
import Data.Maybe (maybeToList)
import Control.Monad (guard)
```

Let's start with solving a problem for 3-gon ring.

```haskell
data Ring = Ring { outer :: [Int]
  , inner :: [Int]}
  deriving (Show, Eq)


ring_size = 5
```

The ring is described in a strongly defined way, so we need to "normalize" a ring - describe it from the lowest outer element. If the order is wrong, we'll just rotate a ring 60° clockwise and call *normalize* again, recursively.

```haskell
normalize :: Ring -> Ring
normalize r = if head (outer r) == minimum (outer r)
  then r
  else normalize $ Ring { outer = tail $ outer r ++ [head $ outer r]
    , inner = tail $ inner r ++ [head $ inner r]}
```

Another concept in the problem description is a "description string".

```haskell
describe :: Ring -> [[Int]]
describe r = describer 0
  where describer i | i == ring_size = []
     | otherwise = [(outer r !! i)
                   ,(inner r !! i)
                   ,(inner r !! ((i + 1) `mod` ring_size))] :
                   describer (i + 1)
```

There are 6! = 720 variants of rings for 3-gon rings without rotations. That can be just brute-forced. To help us in bruteforcing, we define a $describe^{-}1$ function.

```haskell
undescribe :: [[Int]] -> Maybe Ring
undescribe [] = Just $ Ring {inner = [], outer = []}
undescribe ([o, i1, i2] : ns) = do partial <- undescribe ns
  new_inner <- case inner partial of
```

```
    [] → Just [i1, i2]
    xs | length xs ≡ ring_size →
        if last xs ≡ i1
        then Just $ i1 : (init xs)
        else Nothing
    (ip1 : xs) | ip1 ≡ i2 → Just (i1 : ip1 : xs)
    _ → Nothing
  return $ Ring { outer = o : outer partial
    , inner = new_inner }
```

The ring must be "magical": that is, sum of all its chunks must be constant:

```
is_magical :: Ring → Bool
is_magical r = and $ map ((≡ sum (head chunks)) ∘ sum) (tail chunks)
  where chunks = describe r
```

Let's begin enumerating. Let's select "outer" numbers first:

```
enumerate_rings′ = enumerator [1 . . ring_size ∗ 2]
  where enumerator [d] = [Ring [] [d]]
    enumerator digits
      | length digits ≡ ring_size ∗ 2 =
        do d ← filter (≢ 10) $ filter (⩽ ring_size + 1) digits
          r ← filter (and ∘ map (>d) ∘ outer) $ enumerator (d ‘delete‘ digits)
          return $ r { outer = d : outer r }
      | length digits > ring_size =
        do d ← digits
          r ← enumerator (d ‘delete‘ digits)
          return $ r { outer = d : outer r }
      | otherwise =
        do d ← digits
          r ← enumerator (d ‘delete‘ digits)
          return $ r { inner = d : inner r }
```

Another approach is to generate by chunks:

```
digits = [1 . . ring_size ∗ 2]


get_first_chunk = do digit1 ← digits
  digit2 ← digit1 ‘delete‘ digits
  digit3 ← digits \\ [digit1, digit2]
  return [digit1, digit2, digit3]


get_chunk :: [[Int]] → [[Int]]
get_chunk alr
  | length alr ≡ ring_size − 1 =
```

$$
\begin{array}{l}
\quad\quad \mathbf{let}\ \mathit{prev} = \mathit{last}\ \mathit{alr} \\
\quad\quad\quad \mathit{next} = \mathit{head}\ \mathit{alr} \\
\quad\quad\quad s = \mathit{sum}\ \mathit{next} \\
\quad\quad\quad \mathit{digit2} = \mathit{last}\ \mathit{prev} \\
\quad\quad\quad \mathit{digit3} = \mathit{head}\ (\mathit{tail}\ \mathit{next}) \\
\quad\quad\quad \mathit{digit1} = s - \mathit{digit2} - \mathit{digit3} \\
\quad\quad\quad \mathit{dav} = \mathit{digits} \setminus\setminus (\mathit{digit2} : \mathit{digit3} : (\mathit{concat}\ \mathit{alr}))\ \mathbf{in} \\
\quad\quad \mathbf{if}\ ([\mathit{digit1}] \equiv \mathit{dav}) \wedge (\mathit{digit1} > \mathit{head}\ \mathit{next}) \\
\quad\quad \mathbf{then}\ [[\mathit{digit1}, \mathit{digit2}, \mathit{digit3}]] \\
\quad\quad \mathbf{else}\ [\,] \\
\quad \mid \mathit{otherwise} = \mathbf{do}\ \mathbf{let}\ \mathit{prev} = \mathit{last}\ \mathit{alr} \\
\quad\quad\quad\quad\quad\quad \mathbf{let}\ \mathit{next} = \mathit{head}\ \mathit{alr} \\
\quad\quad\quad\quad\quad\quad \mathbf{let}\ s = \mathit{sum}\ \mathit{prev} \\
\quad\quad\quad\quad\quad\quad \mathbf{let}\ \mathit{digit2} = \mathit{last}\ \mathit{prev} \\
\quad\quad\quad\quad\quad\quad \mathbf{let}\ \mathit{dav} = \mathit{digit2}\ \text{`}\mathit{delete}\text{`}\ \mathit{digits} \setminus\setminus (\mathit{concat}\ \mathit{alr}) \\
\quad\quad\quad\quad\quad\quad \mathit{digit1} \leftarrow \mathit{filter}\ (>\mathit{head}\ \mathit{next})\ \mathit{dav} \\
\quad\quad\quad\quad\quad\quad \mathbf{let}\ \mathit{digit3} = s - \mathit{digit1} - \mathit{digit2} \\
\quad\quad\quad\quad\quad\quad \mathit{guard}\ (\mathit{digit3} \in (\mathit{digit1}\ \text{`}\mathit{delete}\text{`}\ \mathit{dav})) \\
\quad\quad\quad\quad\quad\quad \mathit{return}\ [\mathit{digit1}, \mathit{digit2}, \mathit{digit3}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{enumerate\_rings} = \mathbf{do}\ \mathit{chunk} \leftarrow \mathit{get\_first\_chunk} \\
\quad \mathit{chunk2} \leftarrow \mathit{get\_chunk}\ [\mathit{chunk}] \\
\quad \mathit{chunk3} \leftarrow \mathit{get\_chunk}\ [\mathit{chunk}, \mathit{chunk2}] \\
\quad \mathit{chunk4} \leftarrow \mathit{get\_chunk}\ [\mathit{chunk}, \mathit{chunk2}, \mathit{chunk3}] \\
\quad \mathit{chunk5} \leftarrow \mathit{get\_chunk}\ [\mathit{chunk}, \mathit{chunk2}, \mathit{chunk3}, \mathit{chunk4}] \\
\quad d \leftarrow \mathit{maybeToList}\ \$\ \mathit{undescribe}\ [\mathit{chunk}, \mathit{chunk2}, \mathit{chunk3}, \mathit{chunk4}, \mathit{chunk5}] \\
\quad \mathit{return}\ d
\end{array}
$$

That's not very scalable, but works, and works fast. Long live Haskell!

$$
\begin{array}{l}
\mathit{main} :: \mathit{IO}\ () \\
\mathit{main} = \mathit{print}\ \$\ \mathit{concat}\ \$\ \mathit{map}\ \mathit{show}\ \$\ \mathit{concat}\ \$\ \mathit{maximum}\ \$\ \mathit{map}\ \mathit{describe}\ \mathit{enumerate\_rings}
\end{array}
$$

Problem solved, the answer is `"6531031914842725"`.