ERIC BAILEY

ADVENT OF CODE

Contents

2015

Day 25: Let It Snow

https://adventofcode.com/2015/day/25

Merry Christmas! Santa is booting up his weather machine; looks like you might get a white Christmas after all.

The weather machine beeps! On the console of the machine is a copy protection message asking you to enter a code from the instruction manual. Apparently, it refuses to run unless you give it that code. No problem; you'll just look up the code in the–

"Ho ho ho", Santa ponders aloud. "I can't seem to find the manual."

You look up the support number for the manufacturer and give them a call. Good thing, too - that 49th star wasn't going to earn itself.

"Oh, that machine is quite old!", they tell you. "That model went out of support six minutes ago, and we just finished shredding all of the manuals. I bet we can find you the code generation algorithm, though."

After putting you on hold for twenty minutes (your call is **very** important to them, it reminded you repeatedly), they finally find an engineer that remembers how the code system works.

The codes are printed on an infinite sheet of paper, starting in the top-left corner. The codes are filled in by diagonals: starting with the first row with an empty first box, the codes are filled in diagonally up and to the right. This process repeats until the infinite paper is covered. So, the first few codes are filled in in this order:

The paper is very thin so it can be folded up neatly into the manual.

	1	2	3	4	5	6
1	1	3	6	10	15	21
2	2	5	9	14	20	
3	4	8	13	19		
4	7	12	18			
5	11	17				
6	16					

For example, the 12th code would be written to row `4`, column `2`; the 15th code would be written to row `1`, column `5`.

The voice on the other end of the phone continues with how the codes are actually generated. The first code is `20151125`. After that, each code is generated by taking the previous one, multiplying it by `252533`, and then keeping the remainder from dividing that value by `33554393`.

So, to find the second code (which ends up in row `2`, column `1`), start with the previous value, `20151125`. Multiply it by `252533` to get `5088824049625`. Then, divide that by `33554393`, which leaves a remainder of `31916031`. That remainder is the second code.

"Oh!", says the voice. "It looks like we missed a scrap from one of the manuals. Let me read it to you." You write down his numbers:

	1	2	3	4	5	6
1	20151125	18749137	17289845	30943339	10071777	33511524
2	31916031	21629792	16929656	7726640	15514188	4041754
3	16080970	8057251	1601130	7981243	11661866	16474243
4	24592653	32451966	21345942	9380097	10600672	31527494
5	77061	17552253	28094349	6899651	9250759	31663883
6	33071741	6796745	25397450	24659492	1534922	27995004

"Now remember", the voice continues, "that's not even all of the first few numbers; for example, you're missing the one at 7,1 that would come before 6,2. But, it should be enough to let your– oh, it's time for lunch! Bye!" The call disconnects.

Santa looks nervous. Your puzzle input contains the message on the machine's console. **What code do you give the machine?**

Part Two

The machine springs to life, then falls silent again. It beeps. "Insufficient fuel", the console reads. "**Fifty stars** are required before proceeding. **One star** is available."

..."one star is available"? You check the fuel tank; sure enough, a lone star sits at the bottom, awaiting its friends. Looks like you need to provide 49 yourself.

Haskell solution

Rather than try to generate the codes sequentially, which can quickly result in a stack overflow, make some observations.

The positions in first column are the lazy caterer's sequence[1], i.e. A000124 in The Online Encyclopedia of Integer Sequences[2].

$$a(n) = n \times (n-1)/2 + 1 \tag{1}$$

Or equivalently in Haskell:

⟨*Define a000124* 7a⟩≡
```
a000124 :: (Integral a) => a -> a
a000124 n = n * (n - 1) `div` 2 + 1
```
This definition is continued in chunk 7b.
This code is used in chunk 10.

Since this problem might involve some rather large numbers, specialize the polymorphic `a000124` to Haskell's arbitrary precision **`Integer`**, and ensure the compiler inlines it.

⟨*Define a000124* 7a⟩+≡
```
{-# SPECIALIZE INLINE a000124 :: Integer -> Integer #-}
```
This code is used in chunk 10.

[1] https://en.wikipedia.org/wiki/Lazy_caterer's_sequence
[2] https://oeis.org/A000124

THEN TO CALCULATE THE DISTANCE from that position to that of another column c in the same row r, simply subtract the rth triangular number[3] from the $(c + r - 1)$th. The triangular numbers are known as A000217 in The Online Encyclopedia of Integer Sequences[4].

[3] https://en.wikipedia.org/wiki/Triangular_number

[4] https://oeis.org/A000217

$$T(n) = n \times (n + 1)/2 \tag{2}$$

Or equivalently in Haskell:

⟨*Define a000217* 8a⟩≡
```
a000217 :: (Integral a) => a -> a
a000217 n = n * (n + 1) `div` 2
```
This definition is continued in chunk 8b.
This code is used in chunk 10.

Just as before, specialize to **`Integer`** and inline.

⟨*Define a000217* 8a⟩+≡
```
{-# SPECIALIZE INLINE a000217 :: Integer -> Integer #-}
```
This code is used in chunk 10.

So, to find the position in the sequence of codes from a given column and row:

⟨*Find the position* 8c⟩≡
```
position = a000124 row + a000217 (column + row - 1) - a000217 row
```
This code is used in chunk 8e.

THE GENERATING FUNCTION for the sequence is as follows, where $n \in \mathbb{N}$ is the position.

$$f(n) = \begin{cases} 20151125, & \text{for } n = 0 \\ f(n-1) \times 252533 \pmod{33554393}, & \text{otherwise} \end{cases} \tag{3}$$

That recursive definition can be rewritten as a closed formula.

$$f(n) = 20151125 \times 252533^{n-1} \pmod{33554393} \tag{4}$$

Or equivalently in Haskell:

⟨*Find the code at a given position* 8d⟩≡
```
20151125 * (252533 ^ (position - 1)) `mod` 33554393
```
This code is used in chunk 8e.

Thus, Part One can be solved by implementing a function that takes a coordinate pair and returns the specified code.

⟨*Define partOne* 8e⟩≡
```
partOne :: Coordinates -> Integer
partOne (column, row) = ⟨Find the code at a given position 8d⟩
  where
    ⟨Find the position 8c⟩
```
This code is used in chunk 10.

There is nothing to solve for Part Two.

To parse the input, write a silly, overly explicit **`Parser`**.

⟨*Import tools for parsing the input* 9a⟩≡
```
import Control.Monad (void)
import Text.Trifecta (Parser, comma, natural, symbol)
```
This code is used in chunk 10.

⟨*Define coordinates* 9b⟩≡
```
coordinates :: Parser Coordinates
coordinates =
  do
    void $ symbol "To continue, please consult the code grid in the manual."
    row ← symbol "Enter the code at row" *> natural <* comma
    column ← symbol "column" *> natural
    pure (column, row)
```
This code is used in chunk 10.

`Coordinates` is just a two-tuple of **`Integer`**s.

⟨*Define Coordinates* 9c⟩≡
```
type Coordinates = (Integer, Integer)
```
This code is used in chunk 10.

Define the usual `getInput`.

⟨*Import some common utilities* 9d⟩≡
```
import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (inputFilePath)
```
This code is used in chunk 10.

⟨*Define getInput* 9e⟩≡
```
getInput :: IO Coordinates
getInput = parseInput coordinates $(inputFilePath)
```
This code is used in chunk 10.

Bring it all together.

⟨*Day25.hs* 10⟩≡

```
module AdventOfCode.Year2015.Day25 where

⟨Import some common utilities 9d⟩
⟨Import tools for parsing the input 9a⟩

⟨Define Coordinates 9c⟩

main :: IO ()
main =
  do
    putStr "Part One: "
    print . partOne =« getInput

⟨Define partOne 8e⟩

⟨Define getInput 9e⟩

⟨Define coordinates 9b⟩

⟨Define a000124 7a⟩

⟨Define a000217 8a⟩
```

Root chunk (not used in this document).

2018

Day 1: Chronal Calibration

Copy description
https://adventofcode.com/2018/day/1

Haskell Solution

A FREQUENCY CHANGE is represented by a summable integer.

⟨*Define data types to model the puzzle input.* 12a⟩≡
```
newtype FrequencyChange = FrequencyChange
  {unFrequencyChange :: Integer}
  deriving stock
    (Eq, Ord, Show)
  deriving
    (Semigroup, Monoid)
    via (Sum Integer)
```
This code is used in chunk 13.

PARSING THE PUZZLE INPUT for Day 1 is easy. The frequency changes are represented by signed integers, e.g.

```
parseString frequencyChanges mempty "+1\n-2\n+3" ==
Success [Sum {getSum = 1},Sum {getSum = -2},Sum {getSum = 3}]
```

⟨*Parse the input.* 12b⟩≡
```
getInput :: IO [FrequencyChange]
getInput = parseInput (some (FrequencyChange <$> integer)) $(inputFilePath)
```
This code is used in chunk 13.

COMPUTING THE ANSWER FOR PART ONE is also a cinch: just sum the changes in frequency.

⟨*Solve parts one and two.* 12c⟩≡
```
partOne :: [FrequencyChange] -> Maybe Integer
partOne = Just . unFrequencyChange . mconcat
```

This definition is continued in chunks 12d and 16.
This code is used in chunks 13 and 17.

TO SOLVE PART TWO, compute the list of frequencies reached and find the first duplicate.

⟨*Solve parts one and two.* 12c⟩+≡
```
partTwo :: [FrequencyChange] -> Maybe Integer
partTwo =
  fmap unFrequencyChange
    . findFirstDup
    . scan
    . cycle
```
This code is used in chunks 13 and 17.

BRING IT all together.

⟨*Day01.hs* 13⟩≡
```
{-# LANGUAGE DerivingVia #-}

module AdventOfCode.Year2018.Day01 where

import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (defaultMainMaybe, inputFilePath)
import AdventOfCode.Util (findFirstDup, scan)
import Data.Monoid (Sum (..))
import Text.Trifecta (integer, some)

⟨Define data types to model the puzzle input. 12a⟩

main :: IO ()
main = $(defaultMainMaybe)

⟨Parse the input. 12b⟩

⟨Solve parts one and two. 12c⟩
```
Root chunk (not used in this document).

Day 2: Inventory Management System

Copy description

https://adventofcode.com/2018/day/2

Haskell solution

DEFINE SOME CONVIENT type aliases.

A BoxID is just a **String**, and a Checksum is just an **Integer**.

⟨*Type aliases* 14a⟩≡

```
type BoxID = String

type Checksum = Integer
```

This code is used in chunk 15.

TO SOLVE PART ONE, **Just** compute the checksum.[5]

[5] See what I did there?

⟨*Compute the checksum.* 14b⟩≡

```
checksum :: [BoxID] → Checksum
checksum =
  fmap frequencies
    >>> filter (elem 2) &&& filter (elem 3)
    >>> length *** length
    >>> product
    >>> fromIntegral
```

This code is used in chunk 15.

⟨*Part One* 14c⟩≡

```
partOne :: [BoxID] → Maybe Checksum
partOne = Just . checksum
```

This code is used in chunk 15.

SOLVE Part Two.

⟨*Correct the box IDs.* 14d⟩≡

```
correctBoxIDs :: [BoxID] → Maybe (BoxID, BoxID)
correctBoxIDs = listToMaybe . mapMaybe go . tails
  where
    go (x : xs@(_ : _)) = (x,) <$> find (hammingSimilar 1 x) xs
    go _ = Nothing
```

This code is used in chunk 15.

⟨*Part Two* 14e⟩≡

```
partTwo :: [BoxID] → Maybe String
partTwo = fmap (uncurry intersect) . correctBoxIDs
```

This code is used in chunk 15.

Bring it all together.

⟨*Day02.hs* 15⟩≡

```
{-# LANGUAGE TupleSections #-}

module AdventOfCode.Year2018.Day02 where

import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (defaultMainMaybe, inputFilePath)
import AdventOfCode.Util (frequencies, hammingSimilar)
import Control.Arrow ((&&&), (***), (>>>))
import Data.List (find, intersect, tails)
import Data.Maybe (listToMaybe, mapMaybe)
import Text.Trifecta (letter, newline, sepEndBy, some)

⟨Type aliases 14a⟩

main :: IO ()
main = $(defaultMainMaybe)

getInput :: IO [BoxID]
getInput = parseInput (some letter `sepEndBy` newline) $(inputFilePath)

⟨Part One 14c⟩

⟨Part Two 14e⟩

⟨Compute the checksum. 14b⟩

⟨Correct the box IDs. 14d⟩
```

Root chunk (not used in this document).

Day 5: Alchemical Reduction

Copy description

https://adventofcode.com/2018/day/5

Haskell Solution

Thanks to Justin Le[6] for teaching me some neat group theory tricks[7]!

[6] https://github.com/mstksg

[7] https://blog.jle.im/entry/alchemical-groups.html

⟨*Define the Unit type.* 16a⟩≡

```
-- | Units' types are represented by letters, modelled by a finite number type
-- inhabited by exactly 26 values.
type Unit = Finite 26
```

This definition is continued in chunk 16b.
This code is used in chunk 17.

Describe this.

⟨*Define the Unit type.* 16a⟩+≡

```
fromChar :: Char -> Maybe (Either Unit Unit)
fromChar c
  | isLower c = Left <$> unit
  | isUpper c = Right <$> unit
  | otherwise = Nothing
  where
    unit = packFinite . fromIntegral $ ((-) `on` ord) (toLower c) 'a'
```

This code is used in chunk 17.

As per the documentation[8], `returnFree` is an injective[9] map that embeds generators into a free algebra[10] (**`FreeAlgebra`**[11]).

[8] https://hackage.haskell.org/package/free-algebras-0.1.2.0/docs/Data-Algebra-Free.html#v:returnFree

[9] https://en.wikipedia.org/wiki/Injective_function

[10] https://en.wikipedia.org/wiki/Free_algebra

[11] https://hackage.haskell.org/package/free-algebras-0.1.2.0/docs/Data-Algebra-Free.html#t:FreeAlgebra

⟨*Define helper functions.* 16c⟩≡

```
inject :: Char -> FreeGroupL Unit
inject = foldMap (either returnFree (invert . returnFree)) . fromChar

clean :: Unit -> FreeGroupL Unit -> FreeGroupL Unit
clean c = foldMapFree go
  where
    go :: Unit -> FreeGroupL Unit
    go d
      | d == c = mempty
      | otherwise = returnFree d

-- | Compute the order of a 'FreeGroupL'.
order :: FreeGroupL a -> Int
order = length . FG.toList
```

This code is used in chunk 17.

⟨*Solve parts one and two.* 12c⟩+≡

```
partOne :: String -> Int
partOne = order . foldMap inject
```

This code is used in chunks 13 and 17.

⟨*Solve parts one and two.* 12c⟩+≡

```
partTwo :: String -> Int
partTwo = minimum . cleanedPolymers . foldMap inject
  where
    cleanedPolymers :: FreeGroupL Unit -> [Int]
    cleanedPolymers polymer = order . flip clean polymer <$> finites
```

This code is used in chunks 13 and 17.

Bring it all together.

⟨*Day05.hs* 17⟩≡
```
{-# LANGUAGE DataKinds #-}

module AdventOfCode.Year2018.Day05
  ( main,
    partOne,
    partTwo,
  )
where

import AdventOfCode.TH (inputFilePath)
import Data.Algebra.Free (foldMapFree, returnFree)
import Data.Char (isLower, isUpper, ord, toLower)
import Data.Finite (Finite, finites, packFinite)
import Data.Function (on)
import Data.Group (invert)
import Data.Group.Free (FreeGroupL)
import qualified Data.Group.Free as FG

⟨Define the Unit type. 16a⟩

⟨Define helper functions. 16c⟩

⟨Solve parts one and two. 12c⟩

main :: IO ()
main =
  do
    input <- readFile $(inputFilePath)
    putStr "Part One: "
    print (partOne input)
    putStr "Part Two: "
    print (partTwo input)
```
Root chunk (not used in this document).

2019

Day 1: The Tyranny of the Rocket Equation

Copy description

https://adventofcode.com/2019/day/1

GAP Solution

fuel := mass\3 − 2

⟨*Day01.g* 20a⟩≡
```
FuelRequiredModule := function( mass )
    return Int( Float( mass / 3 ) ) - 2;
end;;
```

This definition is continued in chunk 20.
Root chunk (not used in this document).

⟨*Day01.g* 20a⟩+≡
```
PartOne := function( )
    local input, line, mass, sum;;
    sum := 0;
    input := InputTextFile ( "./input/day01.txt" );
    line := ReadLine( input );
    repeat
        mass := Int( Chomp( line ) );
        sum := sum + FuelRequiredModule( mass );
        line := ReadLine( input );
    until line = fail or IsEndOfStream( input );
    return sum;
end;;
```

⟨*Day01.g* 20a⟩+≡
```
TotalFuelRequiredModule := function( mass )
    local fuel;;
    fuel := FuelRequiredModule( mass );
    if IsPosInt( fuel ) then
        return fuel + TotalFuelRequiredModule( fuel );
    else
        return 0;
    fi;
end;;
```

⟨*Day01.g* 20a⟩+≡
```
PartTwo := function( )
    local input, line, mass, sum;;
    sum := 0;
    input := InputTextFile ( "./input/day01.txt" );
    line := ReadLine( input );
    repeat
        mass := Int( Chomp( line ) );
        sum := sum + TotalFuelRequiredModule( mass );
        line := ReadLine( input );
    until line = fail or IsEndOfStream( input );
    return sum;
end;;
```

Day 4: Secure Container

Copy description
https://adventofcode.com/2019/day/4

Haskell Solution

My puzzle input was the range 236491-713787, which I converted into a list of lists of `digits`.

⟨*Input* 21a⟩≡
```
getInput :: IO [[Int]]
getInput = pure $ reverse . digits 10 <$> [236491 .. 713787]
```
This code is used in chunk 22.

Spoiler: Parts One and Two vary only in the strictness of the definition of a double, so a generic solver can be parameterized by the binary operation to compare the number of adjacent digits that are the same with 2. In both parts of the puzzle, it must also be the case that the digits never decrease, i.e. the password `isSorted`.

⟨*Generic solver* 21b⟩≡
```
solve :: (Int → Int → Bool) → [[Int]] → Int
solve = count . (isSorted <&&>) . hasDouble
  where
    hasDouble cmp = any ((‘cmp‘ 2) . length) . group
```
This code is used in chunk 22.

For Part One, there must be a double, i.e. at least two adjacent digits that are the same.

⟨*Part One* 21c⟩≡
```
partOne :: [[Int]] → Int
partOne = solve (≥)
```
This definition is continued in chunk 24.
This code is used in chunks 22 and 25.

For Part Two, the password must have a strict double.

⟨*Part Two* 21d⟩≡
```
partTwo :: [[Int]] → Int
partTwo = solve (==)
```
This definition is continued in chunk 24e.
This code is used in chunks 22 and 25.

BRING IT all together.

⟨*Day04.hs* 22⟩≡

```
module AdventOfCode.Year2019.Day04 where

import AdventOfCode.TH (defaultMain)
import AdventOfCode.Util (count, (<&&>))
import Data.FastDigits (digits)
import Data.List (group)
import Data.List.Ordered (isSorted)

main :: IO ()
main = $(defaultMain)

⟨Input 21a⟩

⟨Part One 21c⟩

⟨Part Two 21d⟩

⟨Generic solver 21b⟩
```

Root chunk (not used in this document).

Day 8:

Add missing title

Copy description

https://adventofcode.com/2019/day/8

Haskell solution

A PIXEL can be black, white, or transparent.

⟨*Define a Pixel data type* 23a⟩≡

```
data Pixel
  = Black
  | White
  | Transparent
  deriving (Enum, Eq)
```

This code is used in chunk 25.

Show black pixels as spaces, white ones as hashes, and transparent as dots.

⟨*Implement* **Show** *for* `Pixel` 23b⟩≡

```
instance Show Pixel where
  show Black = " "
  show White = "#"
  show Transparent = "."
```

This code is used in chunk 25.

DEFINE A `Layer` as a list of `Row`s, and a `Row` as a list of `Pixel`s.

⟨*Define a few convenient type aliases* 23c⟩≡

```
type Image = [Layer]

type Layer = [Row]

type Row = [Pixel]
```

This code is used in chunk 25.

PARSE AN `Image`, i.e. one or more `Layer`s comprised of `height` `Row`s of `width` `Pixel`s.

⟨*Parse an image* 23d⟩≡

```
image :: Int → Int → Parser Image
image width height = some layer
  where
    layer :: Parser Layer
    layer = count height row
    row :: Parser Row
    row = count width pixel
```

This code is used in chunk 25.

Parse an encoded black, white, or transparent pixel.

⟨*Parse a pixel* 23e⟩≡

```
pixel :: Parser Pixel
pixel =
  (char '0' *> pure Black <?> "A black pixel")
    <|> (char '1' *> pure White <?> "A white pixel")
    <|> (char '2' *> pure Transparent <?> "A transparent pixel")
```

This code is used in chunk 25.

SOLVE Part One.

⟨*Part One* 21c⟩+≡
```
partOne :: Image → Int
```
This code is used in chunks 22 and 25.

Return the product of the number of ones (`White` pixels) and the number of twos (`Transparent` pixels) in the `layer` with the fewest `Black` pixels.

⟨*Part One* 21c⟩+≡
```
partOne layers = numberOf White layer * numberOf Transparent layer
  where
```
This code is used in chunks 22 and 25.

Find the `layer` with the fewest zeros, i.e. `Black` pixels.

sp?

⟨*Part One* 21c⟩+≡
```
    layer = minimumBy (compare `on` numberOf Black) layers
```
This code is used in chunks 22 and 25.

Return the number of elements equivalent to a given one, in a given list of lists of elements of the same type. More specifically, return the number of `Pixel`s of a given color in a given `Layer`.

There's gotta be a Data.List function for this..

⟨*Part One* 21c⟩+≡
```
    numberOf :: Eq a ⇒ a → [[a]] → Int
    numberOf x = sum . fmap (length . filter (== x))
```
This code is used in chunks 22 and 25.

SOLVE Part Two.

⟨*Part Two* 21d⟩+≡
```
partTwo :: Image → String
partTwo layers =
  unlines . map (concatMap show) $
    foldl decodeLayer (transparentLayer 25 6) layers
  where
    decodeLayer :: Layer → Layer → Layer
    decodeLayer = zipWith (zipWith decodePixel)
    decodePixel :: Pixel → Pixel → Pixel
    decodePixel Transparent below = below
    decodePixel above _ = above
```
This code is used in chunks 22 and 25.

DEFINE A HELPER FUNCTION to create a transparent layer.

⟨*A transparent layer* 24f⟩≡
```
transparentLayer :: Int → Int → Layer
transparentLayer width height = replicate height (replicate width Transparent)
```
This code is used in chunk 25.

Add some prose here.

⟨*Day08.hs* 25⟩≡
```
module AdventOfCode.Year2019.Day08 where

import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (defaultMain, inputFilePath)
import Control.Applicative ((<|>))
import Data.Function (on)
import Data.List (minimumBy)
import Text.Trifecta (Parser, char, count, some, (<?>))

⟨Define a Pixel data type 23a⟩

⟨Implement Show for Pixel 23b⟩

⟨Define a few convenient type aliases 23c⟩

main :: IO ()
main = $(defaultMain)

getInput :: IO Image
getInput = parseInput (image 25 6) $(inputFilePath)

⟨Part One 21c⟩

⟨Part Two 21d⟩

⟨Parse an image 23d⟩

⟨Parse a pixel 23e⟩

⟨A transparent layer 24f⟩
```
Root chunk (not used in this document).

2021

Day 1: Sonar Sweep

Copy description
https://adventofcode.com/2021/day/1

Haskell solution

THE INPUT is just a list of natural numbers.

⟨*Parse the input.* 28a⟩≡
```
getInput :: IO [Integer]
getInput = parseInput (some natural) $(inputFilePath)
```
This code is used in chunk 29.

THE GENERAL SOLUTION is to count pairwise increases.

⟨*Count pairwise increases.* 28b⟩≡
```
countPairwiseIncreases :: Ord a => Int -> [a] -> Int
countPairwiseIncreases n =
  count (== LT)
    . uncurry (zipWith compare)
    . (id &&& drop n)
```
This code is used in chunk 29.

FOR EXAMPLE, in the following list there are seven pairwise increases.

⟨*Example* 28c⟩≡
```
example :: [Integer]
example = [199, 200, 208, 210, 200, 207, 240, 269, 260, 263]
```
This code is used in chunk 29.

```
λ> countPairwiseIncreases 1 example
7
```

The seven pairwise increases are as follows:

```
[(199,200), (200, 208), (208, 210), (200, 207), (207, 240), (240, 269), (260, 263)]
```

FOR PART ONE, simply count pairwise increases.

⟨*Solve Part One.* 28d⟩≡
```
partOne :: [Integer] -> Int
partOne = countPairwiseIncreases 1
```
This code is used in chunk 29.

FOR PART TWO, count pairwise increases with an offset of 3.

⟨*Solve Part Two.* 28e⟩≡
```
partTwo :: [Integer] -> Int
partTwo = countPairwiseIncreases 3
```
This code is used in chunk 29.

Bring it all together.

⟨*Day01.hs* 29⟩≡

```
module AdventOfCode.Year2021.Day01 where

import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (defaultMain, inputFilePath)
import AdventOfCode.Util (count)
import Control.Arrow ((&&&))
import Text.Trifecta (natural, some)

main :: IO ()
main = $(defaultMain)

⟨Parse the input. 28a⟩

⟨Example 28c⟩

⟨Solve Part One. 28d⟩

⟨Solve Part Two. 28e⟩

⟨Count pairwise increases. 28b⟩
```

Root chunk (not used in this document).

Day 2: Dive!

https://adventofcode.com/2021/day/2

Now, you need to figure out how to pilot this thing.

It seems like the submarine can take a series of commands like `forward 1`, `down 2`, or `up 3`:

- `forward X` increases the horizontal position by `X` units.
- `down X` increases the depth by `X` units.
- `up X` decreases the depth by `X` units.

Note that since you're on a submarine, `down` and `up` affect your **depth**, and so they have the opposite result of what you might expect.

The submarine seems to already have a planned course (your puzzle input). You should probably figure out where it's going. For example:

```
forward 5
down 5
forward 8
up 3
down 8
forward 2
```

Your horizontal position and depth both start at `0`. The steps above would then modify them as follows:

- `forward 5` adds `5` to your horizontal position, a total of `5`.
- `down 5` adds `5` to your depth, resulting in a value of `5`.
- `forward 8` adds `8` to your horizontal position, a total of `13`.
- `up 3` decreases your depth by `3`, resulting in a value of `2`.
- `down 8` adds `8` to your depth, resulting in a value of `10`.
- `forward 2` adds `2` to your horizontal position, a total of `15`.

After following these instructions, you would have a horizontal position of 15 and a depth of 10. (Multiplying these together produces `150`.)

Calculate the horizontal position and depth you would have after following the planned course. **What do you get if you multiply your final horizontal position by your final depth?**

Part Two

Based on your calculations, the planned course doesn't seem to make any sense. You find the submarine manual and discover that the process is actually slightly more complicated.

In addition to horizontal position and depth, you'll also need to track a third value, **aim**, which also starts at `0`. The commands also mean something entirely different than you first thought:

- `down` `X` increases your aim by `X` units.
- `up` `X` decreases your aim by `X` units.
- `forward` `X` does two things:
  - It increases your horizontal position by `X` units.
  - It increases your depth by your aim **multiplied by** `X`.

Again note that since you're on a submarine, `down` and `up` do the opposite of what you might expect: "down" means aiming in the positive direction.

Now, the above example does something different:

- `forward 5` adds `5` to your horizontal position, a total of `5`. Because your aim is `0`, your depth does not change.
- `down 5` adds `5` to your aim, resulting in a value of `5`.
- `forward 8` adds `8` to your horizontal position, a total of `13`. Because your aim is `5`, your depth increases by $8 * 5 = 40$.
- `up 3` decreases your aim by `3`, resulting in a value of `2`.
- `down 8` adds `8` to your aim, resulting in a value of `10`.
- `forward 2` adds `2` to your horizontal position, a total of `15`. Because your aim is `10`, your depth increases by $2 * 10 = 20$ to a total of `60`.

After following these new instructions, you would have a horizontal position of `15` and a depth of `60`. (Multiplying these produces `900`.)

Using this new interpretation of the commands, calculate the horizontal position and depth you would have after following the planned course. **What do you get if you multiply your final horizontal position by your final depth?**

Haskell solution

A Direction is a change in horizontal position and a change in depth, represented by a 2-dimensional vector[12], monoidal under addition[13].

[12] https://hackage.haskell.org/package/linear/docs/Linear-V2.html#t:V2

[13] https://hackage.haskell.org/package/base/docs/Data-Monoid.html#t:Sum

⟨*Define some data types* 32a⟩≡

```
newtype Direction = Direction {unDirection :: V2 Int}
  deriving stock (Eq, Show)
  deriving
    (Semigroup, Monoid)
    via (Sum (V2 Int))
```

This definition is continued in chunk 33.
This code is used in chunk 35.

The ⟨*known directions* 32b⟩ are forward, down, and up.

⟨*known directions* 32b⟩≡

```
forward, down, up :: Int → Direction
```

This definition is continued in chunk 32.
This code is used in chunk 32f.

forward x increases the horizontal position by x units.

⟨*known directions* 32b⟩+≡

```
forward = Direction . flip V2 0
```

This code is used in chunk 32f.

down x increases the depth by x units.

⟨*known directions* 32b⟩+≡

```
down = Direction . V2 0
```

This code is used in chunk 32f.

up x decreases the depth by x units, i.e. down with a negated x.

⟨*known directions* 32b⟩+≡

```
up = down . negate
```

This code is used in chunk 32f.

Define a Direction parser using the ⟨*known directions* 32b⟩.

⟨*Define a Direction parser* 32f⟩≡

```
direction :: Parser Direction
direction = dir <*> (fromInteger <$> natural)
  where
    dir =
      symbol "forward" $> forward
        <|> symbol "down" $> down
        <|> symbol "up" $> up

  ⟨known directions 32b⟩
```

This code is used in chunk 35.

The puzzle input is a list of Directions.

⟨*Parse the input* 32g⟩≡

```
getInput :: IO [Direction]
getInput = parseInput (some direction) $(inputFilePath)
```

Root chunk (not used in this document).

The general solution of the puzzle is to sum a list of additive monoids, extract the final position, and compute the product of the horizontal position and depth.

⟨*Solve the puzzle* 33a⟩≡
```
solve :: Monoid m ⇒ (m → V2 Int) → [m] → Int
solve extract = product . extract . mconcat
```
This code is used in chunk 35.

For Part One, the additive monoid is Direction.

⟨*Solve Part One* 33b⟩≡
```
partOne :: [Direction] → Int
partOne = solve unDirection
```
This code is used in chunk 35.

For Part Two, the additive monoid is Aim, i.e. an integer.

⟨*Define some data types* 32a⟩+≡
```
newtype Aim = Aim Int
  deriving stock (Eq, Show)
  deriving
    (Semigroup, Monoid)
    via (Sum Int)
```
This code is used in chunk 35.

forward x increases the horizontal position by x units and increases the depth by the aim multiplied by x, forming a semi-direct product[14] of Direction (the sub-monoid) and Aim (the quotient monoid).

Define how Aim acts on Direction.

[14] https://hackage.haskell.org/package/monoid-extras/docs/Data-Monoid-SemiDirectProduct.html#t:Semi

⟨*Define some data types* 32a⟩+≡
```
instance Action Aim Direction where
  act (Aim a) (Direction (V2 x y)) = Direction (V2 x (y + a * x))
```
This code is used in chunk 35.

Use the **Action** to construct the semi-direct product Direction ⋊ Aim.

⟨*Define the semi-direct product* 33e⟩≡
```
lift :: Direction → Semi Direction Aim
```
This definition is continued in chunks 33 and 34a.
This code is used in chunk 34b.

forward, i.e. a Direction with a depth change of 0, doesn't affect the aim.

⟨*Define the semi-direct product* 33e⟩+≡
```
lift dir@(Direction (V2 _ 0)) = inject dir
```
This code is used in chunk 34b.

up or down, i.e. a Direction with a horizontal change of 0 and a non-zero depth change y, results in an aim change of y units.

⟨*Define the semi-direct product* 33e⟩+≡
```
lift (Direction (V2 0 y)) = embed (Aim y)
```
This code is used in chunk 34b.

Since `Direction` is not specific enough to prevent them, add a catch-all clause to handle invalid directions, e.g. `forward` and `up` simultaneously.

⟨*Define the semi-direct product* 33e⟩+≡
```
lift _ = error "Invalid direction"
```
This code is used in chunk 34b.

To solve Part Two, `lift` each `Direction` in the input to `Direction` ⋊ `Aim`, forgetting the `Aim` tag to extract the final position.

⟨*Solve Part Two* 34b⟩≡
```
partTwo :: [Direction] -> Int
partTwo = solve (unDirection . untag) . map lift
  where
    ⟨Define the semi-direct product 33e⟩
```
This code is used in chunk 35.

Bring it all together.

⟨*Day02.hs* 35⟩≡

```
{-# LANGUAGE DerivingVia #-}
{-# LANGUAGE MultiParamTypeClasses #-}

module AdventOfCode.Year2021.Day02 where

import AdventOfCode.Input (parseInput)
import AdventOfCode.TH (defaultMain, inputFilePath)
import Control.Applicative ((<|>))
import Data.Functor (($>))
import Data.Monoid.Action (Action (..))
import Data.Monoid.SemiDirectProduct.Strict (Semi, embed, inject, untag)
import Data.Semigroup (Sum (..))
import Linear (V2 (..))
import Text.Trifecta (Parser, natural, some, symbol)

⟨Define some data types 32a⟩

main :: IO ()
main = $(defaultMain)

getInput :: IO [Direction]
getInput = parseInput (some direction) $(inputFilePath)

example :: [Direction]
example =
  [ forward 5,
    down 5,
    forward 8,
    up 3,
    down 8,
    forward 2
  ]

⟨Solve Part One 33b⟩

⟨Solve Part Two 34b⟩

⟨Solve the puzzle 33a⟩

⟨Define a Direction parser 32f⟩
```

Root chunk (not used in this document).

Chunks

To-Do