21.4 Giant Squid

Part One

Bingo is a game played on a set of boards each consisting of a 5x5 grid of numbers. Numbers are chosen at random, and the chosen number is `marked` on all boards on which it appears. (Numbers may not appear on all boards.) If all numbers in any row or any column of a board are marked, that board `wins`. (Diagonals don't count.)

Write a function `Bingo` that takes as left argument the random numbers drawn and as right argument a 3D array where each major cell represents a board. The function should return the index of the board that wins first.

Here is a worked example:

```
      nums
7 4 9 5 11 17 23 2 0 14 21 24 10 16 13 6 15 25 12 22 18 20 8 19 3 26 1

      boards
22 13 17 11  0
 8  2 23  4 24
21  9 14 16  7
 6 10  3 18  5
 1 12 20 15 19

 3 15  0  2 22
 9 18 13 17  5
19  8  7 25 23
20 11 10 24  4
14 21 16 12  6

14 21 17 24  4
10 16 15  9 19
18  8 23 26 20
22 11 13  6  5
 2  0 12  3  7

      nums Bingo boards
3
```

This link should give you access to the auxiliary expression that builds the variable `boards`.

Alternatively, you can build it as such:

```
      boards ←, 22 13 17 11 0 8 2 23 4 24
      boards ,← 21 9 14 16 7 6 10 3 18 5
      boards ,← 1 12 20 15 19 3 15 0 2 22
      boards ,← 9 18 13 17 5 19 8 7 25 23
      boards ,← 20 11 10 24 4 14 21 16 12 6
      boards ,← 14 21 17 24 4 10 16 15 9 19
      boards ,← 18 8 23 26 20 22 11 13 6 5
      boards ← 3 5 5⍴boards,2 0 12 3 7
```

Here is the explanation of the result: after the first eleven numbers are drawn (`7`, `4`, `9`, `5`, `11`,`17`, `23`, `2`, `0`, `14`, and `21`), there are no winners, but the boards are marked as follows (shown here adjacent to each other to save space):

```
22 13 17 11  0         3 15  0  2 22         14 21 17 24  4
 8  2 23  4 24         9 18 13 17  5         10 16 15  9 19
21  9 14 16  7        19  8  7 25 23         18  8 23 26 20
 6 10  3 18  5        20 11 10 24  4         22 11 13  6  5
 1 12 20 15 19        14 21 16 12  6          2  0 12  3  7
```

Finally, `24` is drawn:

```
22 13 17 11  0         3 15  0  2 22         14 21 17 24  4
 8  2 23  4 24         9 18 13 17  5         10 16 15  9 19
21  9 14 16  7        19  8  7 25 23         18  8 23 26 20
 6 10  3 18  5        20 11 10 24  4         22 11 13  6  5
 1 12 20 15 19        14 21 16 12  6          2  0 12  3  7
```

At this point, the third board wins because it has at least one complete row or column of marked numbers (in this case, the entire top row is marked: `14 21 17 24 4`).

Part Two

When playing with sore losers, the best thing to do is to let them win (or not play with them in the first place). So, write a function `NoBingo` that takes the same left and right arguments as `Bingo`. Instead of returning the index of the board that would win first, return the index of the board that would win last.

In the above example, the second board is the last to win, which happens after `13` is eventually called and its middle column is completely marked. Thus, using the arrays `nums` and `boards` from before, we have:

```
      nums NoBingo boards
2
```

A larger set of `boards` and list `nums` is available in the file `resources/21.4.txt`, for which the functions should give the following results:

```
      nums Bingo boards
37
      nums NoBingo boards
73
```

15.2 I Was Told There Would Be No Math

Part One

You need to help the elves wrap some presents, but they are running low on wrapping paper, and so they need to submit an order for more. They have a list of the dimensions (length `l`, width `w`, and height `h`) of each present, and only want to order exactly as much as they need.

Fortunately, every present is a box (a perfect right rectangular prism), which makes calculating the required wrapping paper for each gift a little easier: find the surface area of the box, which is `(2×l×w) + (2×w×h) + (2×h×l)`. The elves also need a little extra paper for each present: the area of the smallest side.

For example:

- A present with dimensions 2cm × 3cm × 4cm requires `(2×6) + (2×12) + (2×8) = 52` square centimetres of wrapping paper plus `6` square centimetres of slack, for a total of `58` square centimetres.
- A present with dimensions 1cm × 1cm × 10cm requires `(2×1) + (2×10) + (2×10) = 42` square centimetres of wrapping paper plus `1` square centimetre of slack, for a total of `43` square centimetres.

The file `resources/15.2.txt` lists the dimensions of a series of presents that need to be wrapped, with one present per row. If they were in that file, the two presents from the examples above would be listed as such:

```
2x3x4
1x1x10
```

Write a function `WrappingPaperArea` that takes a single right argument character vector representing the path to a file with the format described and returns the total area of wrapping paper needed. For the file `resources/15.2.txt`, the correct answer is `1606483`.

Part Two

The elves are also running low on ribbon. Ribbon is all the same width, so they only have to worry about the length they need to order, which they would again like to be exact.

The ribbon required to wrap a present is the shortest distance around its sides, or the smallest perimeter of any one face. Each present also requires a bow made out of ribbon as well; the centimetres of ribbon required for the perfect bow is equal to the cubic centimetres of volume of the present. Don't ask how they tie the bow, though; they'll never tell.

For example:

- A present with dimensions 2cm × 3cm × 4cm requires `2+2+3+3 = 10` centimetres of ribbon to wrap the present plus `2×3×4 = 24` centimetres of ribbon for the bow, for a total of `34` centimetres.
- A present with dimensions 1cm × 1cm × 10cm requires `1+1+1+1 = 4` centimetres of ribbon to wrap the present plus `1×1×10 = 10` centimetres of ribbon for the bow, for a total of `14` centimetres.

Write a function `RibbonLength` that accepts a right argument similar to the function `WrappingPaperArea` and that returns the total length of ribbon needed to tie all the bows of all the presents listed in the right argument file path. For the file `resources/15.2.txt`, the correct answer is `3842356`.

15.5 Nice Strings

Part One

We need help figuring out which strings in a text file are annoying or nice.

A *nice string* is one with all of the following properties:

- It contains at least three vowels (`'aeiou'` only), like `'aei'`, `'xazegov'`, or `'aeiouaeiouaeiou'`.
- It contains at least one letter that appears twice in a row, like `'xx'`, `'abcdde'` (`'dd'`), or `'aabbccdd'` (`'aa'`, `'bb'`, `'cc'`, or `'dd'`).
- It does *not* contain the strings `'ab'`, `'cd'`, `'pq'`, or `'xy'`, even if they are part of one of the other requirements.

For example:

- `'ugknbfddgicrmopn'` is nice because it has at least three vowels (`'u...i...o...'`), a double letter (`'...dd...'`), and none of the disallowed substrings.
- `'aaa'` is nice because it has at least three vowels and a double letter, even though the letters used by different rules overlap.
- `'jchzalrnumimnmhp'` is annoying because it has no double letter.
- `'haegwjzuvuyypxyu'` is annoying because it contains the string `'xy'`.
- `'dvszwmarrgswjxmb'` is annoying because it contains only one vowel.

Write a function `NiceStrings` that counts how many lines of the file pointed to by the character vector right argument are nice strings. For example,

```
      NiceStrings 'path\to\resources\15.5.txt'
255
```

Part Two

Assume, now, that none of the old rules apply, as they are all clearly ridiculous. In all actuality, a nice string is one with all of the following properties:

- It contains a pair of any two letters that appears at least twice in the string without overlapping, like `'xyxy'` (`'xy'`) or `'aabcdefgaa'` (`'aa'`), but not like `'aaa'` (`'aa'`, but it overlaps).
- It contains at least one letter which repeats with exactly one letter between them, like `'xyx'`, `'abcdefeghi'` (`'efe'`), or even `'aaa'`.

For example:

- `'qjhvhtzxzqqjkmpb'` is nice because is has a pair that appears twice (`'qj'`) and a letter that repeats with exactly one letter between them (`'zxz'`).
- `'xxyxx'` is nice because it has a pair that appears twice and a letter that repeats with one between, even though the letters used by each rule overlap.
- `'uurcxstgmygtbstg'` is annoying because it has a pair (`'tg'`) but no repeat with a single letter between them.
- `'ieodomkazucvgmuy'` is annoying because it has a repeating letter with one between (`'odo'`), but no pair that appears twice.

How many strings are nice under these new rules? Write a monadic function `NicerStrings` that acts like the previous one, but uses the new rules for determining whether or not a string is nice. For example,

```
      NicerStrings 'path\to\resources\15.5.txt'
55
```

15.17 Filling Containers

Part One

Suppose you have containers of size `20`, `15`, `10`, `5`, and `5` litres. If you need to store `25` litres and containers can only be fully empty or fully... full, there are four ways to do it:

- `15` and `10`
- `20` and `5` (the first `5`)
- `20` and `5` (the second `5`)
- `15`, `5`, and `5`

Write a dyadic function `CanFill` that takes a positive integer scalar left argument and a positive integer vector right argument and determines in how many ways you can store `⍺` litres if you have at your disposal containers with sizes `⍵`. For example:

```
      25 CanFill 5 5 10 15 20
4
      150 CanFill 33 14 18 20 45 35 16 35 1 13 18 13 50 44 48 6 24 41 30 42
1304
```

Part Two

Now, suppose you want to do the same thing, but using as little containers as possible. In the first example above, the minimum number of containers to store `25` litres was two. There were three ways to use that many containers, and so the answer there would be `3`. Implement this behaviour in the dyadic function `MinFill` that should work as follows:

```
      25 MinFill 5 5 10 15 20
3
      25 MinFill 5 5 10 15 20 25  ⍝ Minimum is using just the container 25
1
      150 MinFill 33 14 18 20 45 35 16 35 1 13 18 13 50 44 48 6 24 41 30 42 18
```

16.3 Triangles

Part One

If you have three positive numbers representing the lengths of the sides of a triangle, those lengths only represent a valid triangle if the sum of any two sides is larger than the remaining side. For example, the "triangle" with side lengths `5 10 25` is impossible because `5 + 10` is not larger than `25`.

Write a function `CountTriangles` that accepts a positive numerical matrix as right argument with an arbitrary number of rows and three columns and counts how many rows of that matrix represent valid triangle side lengths. For example:

```
      CountTriangles 4 3⍴1 2 3,5 10 25,4 3 5,16 5 20
2
```

If you use the data from the file `path\to\resources\16.3.txt`, the answer should be `1050`.

Part Two

Now suppose that triangles are specified in groups of three *vertically*. Each set of three numbers in a column specifies a triangle. Columns are unrelated. For example, given the following specification, numbers with the same hundreds digit would be part of the same triangle:

```
101 301 501
102 302 502
103 303 503
201 401 601
202 402 602
203 403 603
```

Write a monadic function `CountColTriangles` that accepts a positive numerical matrix with a number of rows that is a multiple of 3 and 3 columns and counts how many triangle side length specifications are valid, according to the layout of the side length specifications defined above. For example, if you use the data from the file `path\to\resources\16.3.txt`, the answer should be `1921`.

Bonus

Can you define `CountColTriangles` as a function of `CountTriangles`?

16.6 Column Statistics

Write two monadic functions `MostCommon` and `LeastCommon`. Both functions accept a non-empty matrix and both functions return a vector with as many elements as columns in the input matrix. The result vector contains, in each position, the most/least common element of the corresponding matrix. You can break ties in any way you want.

If you use the character matrix from the file `path\to\resources\16.6.txt`, the result of `MostCommon` should be `'usccerug'` and the result of `LeastCommon` should be `'cnvvtafc'`.

Bonus

Can you relate the two functions? Through an operator, for example?