2015 APL Problem Solving Competition – Phase II Problem Descriptions

This year's Phase II problems are divided into three sets representing three general categories – Bioinformatics, Applications and Recreation and Games. Each set has three problems, one each of low, medium and high difficulty levels. A problem may be broken down into multiple tasks. You do not need to solve all the Phase II problems to be considered eligible for the top three prizes but you do need to complete, at a minimum, one problem of each level of difficulty. The problems can be from different sets. Higher difficulty problems may be substituted for lower difficulty problems – for example, you could solve two medium difficulty problems and one high difficulty problem.

You can also solve more than the minimum number of problems; you can solve all the problems if you choose to do so. Doing more work by solving more problems can work in your favour; if entries from two people are of similar quality but one has solved more or higher difficulty problems, then that entry will receive higher consideration by the judging committee.

Good Luck and Happy Problem Solving!

Note:
Some of the examples are displayed using the user command setting `]boxing on` to more clearly depict the structure of the displayed data.

```
      ('Dyalog' 'APL')(4 4⍴⍳16) 5
  Dyalog  APL    1  2  3  4  5
                 5  6  7  8
                 9 10 11 12
                13 14 15 16

      ]boxing on
Was OFF

      ('Dyalog' 'APL')(4 4⍴⍳16) 5
┌──────────┬──────────┬─┐
│┌──────┬──┐│ 1  2  3  4│5│
│Dyalog│APL│ 5  6  7  8│ │
│└──────┴──┘│ 9 10 11 12│ │
│          │13 14 15 16│ │
└──────────┴──────────┴─┘
```

Description of the Contest2015 Template Files

Two template files are available for download from the contest website. Which file you use depends on how you choose to implement your problem solutions.

If you use Dyalog APL, you should use the template workspace `Contest2015Sample.DWS`

The Contest2015Sample workspace contains:

- **`#.Problems`** – a namespace in which you will develop your solutions and which itself contains:
  - stubs for all of the functions described in the problem descriptions. The function stubs are implemented as traditional APL functions (tradfns) but you can change the implementation to dynamic functions (dfns) if you want to do so. Either form is acceptable.
  - any sample data elements mentioned in the problem descriptions.
  - any sub-functions you develop as a part of your solution should be located in **`#.Problems`**
- **`#.SubmitMe`** – a function used to package your solution for submission.

Make sure you save your work using the `)SAVE` system command!

Once you have developed and are ready to submit your solutions, run the **`#.SubmitMe`** function, enter the requested information and click the **Save** button. **`#.SubmitMe`** will create a file called **Contest2015.dyalog** which will contain any code or data you placed in the **`#.Problems`** namespace. You will then need to upload the **Contest2015Sample.dyalog** file using the contest website.

If you use some other APL system, you can use the template script file `Contest2015.dyalog`

This file contains the correct structure for submission. You can populate it with your code, but do not change the namespace structure. Once you have developed your solution, edit the variable definitions as indicated at the top of the file and upload the file using the contest website. If you use some other APL system to develop your application, it will still need to execute under Dyalog APL; it is recommended that your solution use APL features that are common between your APL system and Dyalog.

Set 1: Bioinformatics Problems

The Bioinformatics problems presented here are based on problems presented on the website http://rosalind.info. The descriptions have been adapted to be more suitable for APL syntax and this competition.

Please note that your solutions should perform on arguments as described in the problem descriptions on rosalind.info.

Bioinformatics Problem 1 (low difficulty) (2 tasks)

Task 1 – Rabbits and Recurrence Relations

The description of this task can be found at http://rosalind.info/problems/fib/.

Write an APL function named `fibRabbits` which

- takes a 2-element integer vector right argument representing
  - `[1]` the number of months
  - `[2]` the number of rabbit pairs produced in a litter
- returns the number of rabbit pairs that will be present after the number of months specified in the argument

Example:

```
     fibRabbits 5 3
19
```

Task 2 – Translating RNA into Protein

The description of this task can be found at http://rosalind.info/problems/prot/.

Write an APL function named `RNAtoProtein` which

- takes a character vector right argument representing an RNA string
- takes a 2-column matrix representing the RNA codon table (this is provided for you as the variable `codon` in the Bio namespace in the downloaded materials)
  - `[;1]` RNA codon
  - `[;2]` amino acid ('*' is used in this table instead of 'Stop')
- returns a character vector representing the protein string encoded by the RNA string

Example:

```
     codon RNAtoProtein 'AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA'
MAMAPRTEINSTRING
```

Bioinformatics Problem 2 (medium difficulty) (2 tasks)

Task 1 – Creating a Restriction Map

The description of this problem can be found at http://rosalind.info/problems/pdpl/.

Write an APL function named `findLocs` which:

- takes an integer vector left argument representing the multiset L
- returns an integer vector representing the set X

Example:

```
      findLocs 2 2 3 3 4 5 6 7 8 10
0 2 4 7 10
```

Task 2 – Inferring mRNA from Protein

The description of this problem can be found at http://rosalind.info/problems/mrna/.

Using the **codon** table described in Problem 1, Task 2, write an APL function named `infermRNA` which

- takes a character vector right argument representing a protein string
- takes the codon table as its left argument
- returns an integer representing the number of different RNA strings from which the protein could have been translated modulo 1,000,000.

Example:

```
      codon infermRNA 'MA'
12
```

Bioinformatics Problem 3 (high difficulty) (3 tasks)

Task 1 – Reversal Distance

The description of this problem can be found at http://rosalind.info/problems/rear/.

Write an APL operator named `reversalDistance` which

- takes an integer vector right argument representing σ as described on Rosalind.info
- takes an integer vector left argument representing π as described on Rosalind.info
- returns an integer representing the reversal distance between σ and π

Examples:

```
      1 2 3 4 5 6 7 8 9 10 reversalDistance 3 1 5 2 7 4 9 6 10 8
9

      3 10 8 2 5 4 7 1 6 9 reversalDistance 5 2 3 1 7 4 10 8 6 9
4

      1 2 3 4 5 6 7 8 9 10 reversalDistance 1 2 3 4 5 6 7 8 9 10
0
```

Task 2 – Longest Subsequences

The description of this problem can be found at http://rosalind.info/problems/lgis/.

Write an APL function, `longestSubsequences` which

- takes an integer permutation vector of length ***n***
- returns a 2-element vector where
  - [1] is the longest increasing subsequence of the permutation vector
  - [2] is the longest decreasing subsequence of the permutation vector

Example:

```
      longestSubsequences 5 1 4 2 3
┌─────┬─────┐
│1 2 3│5 4 2│
└─────┴─────┘
```

Task 3 – Identifying Maximal Repeats

The description of this problem can be found at http://rosalind.info/problems/mrep/.

Write an APL function `maxRepeats` which

- takes a character vector representing a DNA string
- returns a vector of character vectors containing all maximal repeats in the DNA string having a length greater than or equal to 20

Example:

```
      dna←'TAGAGATAGAATGGGTCCAGAGTTTTGTAATTTCCATGGGTCCAGAGTTTTGTAATTTATTATATAGAGATAGAATGGGTCC
AGAGTTTTGTAATTTCCATGGGTCCAGAGTTTTGTAATTTAT'

      maxRepeats dna
```

TAGAGATAGAATGGGTCCAGAGTTTTGTAATTTCCATGGGTCCAGAGTTTTGTAATTTAT	ATGGGTCCAGAGTTTTGTAATTT

Set 2: Applications Problems

Applications Problem 1 (low difficulty) – HTML Creation (2 tasks)

When building a web-enabled application it is often be necessary to render application data as HTML. In each of the tasks in this problem, you'll be asked to construct HTML from an APL array. Use of the Dyalog system function ⎕XML is permitted, but not required. It is optional to "prettify" the format of the HTML that your functions return – the important criteria is that the HTML is well-formed and renders properly in a browser.

Task 1 – He's making a list...

Write an APL function named `htmlList` which

- takes either a character matrix right argument where each row contains a list element item or a vector of character vectors where each element contains a list element item. In each case, leading asterisks ('*') will indicate the level of nesting of the list element.
- returns a character vector containing HTML to render the list.

Example:

```
      htmlList 'Item 1' 'Item 2' '*Item 2.1' '*Item 2.2' '**Item 2.2.1' 'Item 3'
<ul>
  <li>Item 1</li>
  <li>
    Item 2
    <ul>
      <li>Item 2.1</li>
      <li>
        Item 2.2
        <ul>
          <li>Item 2.2.1</li>
        </ul>
      </li>
    </ul>
  </li>
  <li>Item 3</li>
</ul>
```

When viewed in a browser, it should render similar to:

- Item 1
- Item 2
    - Item 2.1
    - Item 2.2
        - Item 2.2.1
- Item 3

Task 2* – *Let's just table it...

Write an APL function named **`htmlTable`** which

- takes a matrix of data as its right argument
- returns a character vector of HTML to render the matrix as an HTML table

Example:

```
      htmlTable 2 3⍴'Dyalog' 'User' 'Meeting' 'Sept' '6-10' 2015
<table>
  <tr>
    <td>Dyalog</td>
    <td>User</td>
    <td>Meeting</td>
  </tr>
  <tr>
    <td>Sept</td>
    <td>6-10</td>
    <td>2015</td>
  </tr>
</table>
```

When viewed in a browser, it should render similar to:

Dyalog	User	Meeting
Sept	6-10	2015

Applications Problem 2 (medium difficulty) – Phrase searching optimisation (1 task)

In this problem, you're tasked with developing an algorithm to perform phrase searches in a full text database most efficiently. The database is an inversion of all word occurrences in the domain. In other words, all the occurrences for a particular word are stored together. Each record in the inversion space has information for the document in which a word occurs, the word's position within that document and the words that immediately precede and follow it. We also store information for every occurrence of a word and, therefore, have frequency information for the total number of occurrences for each word in the domain. It is this frequency data that you'll use to develop your algorithm. Why? Because we want to minimise the amount of data we read from the database as disk operations are typically much slower than in-memory operations.

For example, suppose the phrase you're searching for is "income tax" and looking up the respective frequency information we find the word "income" occurs 5000 times, and the word "tax" occurs 10000 times. So, to find all occurrences of "income tax", the more efficient way is to search the 5000 entries for "income" whose next word is "tax", rather than the 10000 entries for "tax" whose previous word is "income".

In this search engine "noise" words like "of" and "the" are still considered to be significant. In the phrase "The White House", "The" is significant. The same is true for "of" and "the" in "The Speaker of the House of Representatives of the United States".

Suppose the words "house" "of" and "cards" have frequencies of 10,000, 100,000 and 1,000 respectively. Searching for the phase "house of cards" could be accomplished by searching all 100,000 occurrences of "of" and finding those which are preceded by "house" and followed by "cards". But it's more efficient to first search the 1,000 occurrences of "cards" to find those preceded by "of" and then the 10,000 occurrences of "house" to find those followed by "of", then compare the two sets based on document ID and a position difference of ¯2 between the occurrences.

Another example:

Phrase	the	red	badge	of	courage
Word Frequency	1000000	2500	1500	1200000	900

The least work is done by searching:

- 900 courage (preceded by "of")
- 1500 badge (preceded by "red" and followed by "of")
- 2500 red (preceded by "the" and followed by "badge")

Task 1* – *That's an order!

Your task is, given a right argument of an integer vector of word frequencies, write a function named `searchOrder` that returns the indices of the words to search in order to do the least work (search the fewest number of word occurrences) yet ensure that all words in the phrase are accounted for.

Examples:

```
      searchOrder 5000 10000
1
      searchOrder 10000 100000 1000
3 1
      searchOrder 1000000 2500 1500 1200000 900
5 3 2
```

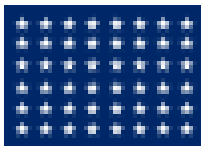
Applications Problem 3 (high difficulty) – The Star Spangled Banner (1 task)

The Betsy Ross Flag Company has decided to outsource its star-spangling operations. Since 1959, the United States flag has had the same pattern of 50 stars representing the 50 states on the blue portion of the U.S. flag which is called the union. But over the history of the United States, there have been 39 different flags with different patterns of stars as states were admitted.

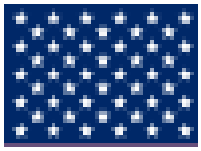
What happens to the flag if Washington DC, or Puerto Rico are admitted as states? Or if the rising seas completely submerge Florida? Or the US decides to auction off Hawaii to reduce its national debt?

While there is no law governing the arrangement of stars, the following precepts have generally been adhered to:

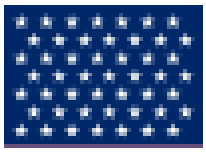
1) perfectly rectangular patterns such as used in the 6×8 48-star flag.



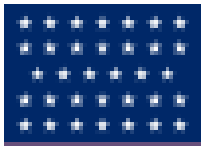
2) staggered patterns of alternating long and short rows, or the reverse, where a "long" row is only one star longer than a "short" row, provided that the total configuration is symmetric about its horizontal centerline. The current 50-star flag is an example, consisting of a 6-5-6-5-6-5-6-5-6 arrangement.



Its 49-star predecessor consisted of the converse but still symmetric pattern 5-6-5-6-5-6-5-6-5.



3) either the center row, when the number of rows is odd, or the center two rows, when the number of rows is even, may differ in length by one star from all the other rows. The 34-star flag is such a case, having a 7-7-6-7-7 arrangement.

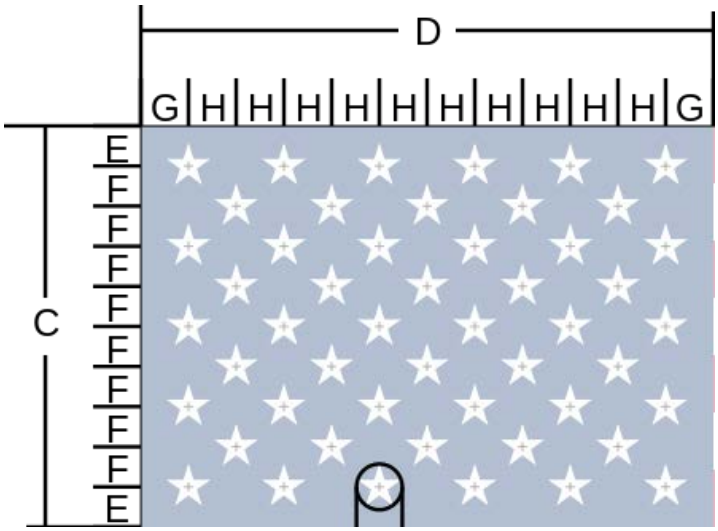


Task 1* – *Starting to Flag

Your task is to write a function called **`arrangeStars`** which:

- takes a right argument integer singleton representing the number of stars to place
- returns a 3-item vector describing a "reasonable" star arrangement where
    - the first item is a vector of the number of stars in each row
    - the second item is the horizontal spacing (in proportion to C) between columns of stars (G and H in the diagram)
    - the third item is the vertical spacing in (proportion to C) between rows of stars (E and F in the diagram)

To explain "reasonable" consider that a 2×25 array of 50 stars would not look correct when squeezed into a union whose length to height ratio (D:C in the diagram) is approximately 1.4 to 1. Stars will be printed on successive rows – in other words, no skipping rows.



For the purposes of this problem, assume that

- G = H
- E = F
- C : D = 1 : 1.4

Note: Given the vagueness of "reasonable", there may not be a single correct solution for a given number of stars. Entries will be judged primarily on code quality.

Example:

```
arrangeStars 32
```

8 8 8 8	0.1555555556	0.2

```
arrangeStars 50  A describes the current U.S. flag
```

6 5 6 5 6 5 6 5 6	0.1166666667	0.1

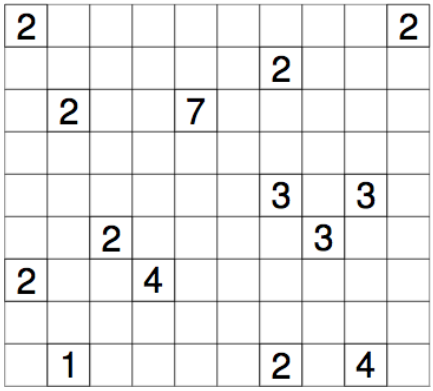
Set 3: Recreation and Games Problems

Recreation and Games Problem 1 (low difficulty) – Is It Nurikabe? (1 task)

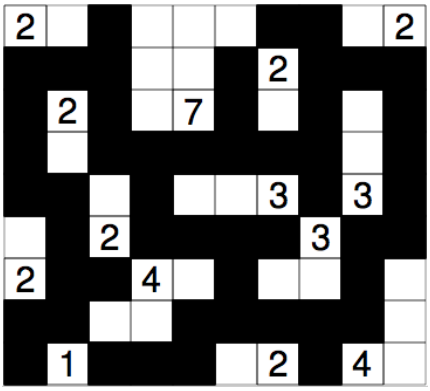
Nurikabe is one of a number of logic puzzles published by Japanese publisher Nikoli Co., Ltd. The puzzle is played on a typically rectangular grid of cells, some of which contain numbers. Cells are initially of unknown color, but can only be black or white. Two same-color cells are considered "connected" if they are adjacent vertically or horizontally, but not diagonally. Connected white cells form "walls", while connected black cells form "a stream". The challenge is to paint each cell black or white, subject to the following rules:

- Each numbered cell is a wall cell, the number in it is the number of cells in that wall.
- Each wall must contain exactly one numbered cell.
- There must be only one stream, which is not allowed to contain "pools", i.e. 2x2 areas of black cells.

Original Puzzle



Solved Puzzle



Task 1* – *Valid?

Your task is to write a function, **`checkNurikabe`**, which:

- takes an integer matrix left argument representing a "solved" Nukikabe puzzle.
  0s represent the stream, and numbers greater than 0 represent walls in theory containing that many cells.
  For example, the representation of the solved puzzle above would be:

```
2 2 0 7 7 7 0 0 2 2
0 0 0 7 7 0 2 0 0 0
0 2 0 7 7 0 2 0 3 0
0 2 0 0 0 0 0 0 3 0
0 0 2 0 3 3 3 0 3 0
2 0 2 0 0 0 0 3 0 0
2 0 0 4 4 0 3 3 0 4
0 0 4 4 0 0 0 0 0 4
0 1 0 0 0 2 2 0 4 4
```

- returns a Boolean scalar, where 1 indicates that the board represents a valid Nurikabe solution according to the rules above, 0 otherwise.

Examples:

```
      checkNurikabe 2 2⍴2 2 0 0
1
      checkNurikabe 2 2⍴2 2 1 0
0
      board ← 5 5 5 5 0 4 0 0 5 0 0 4 0 1 0 2 0 4 0 0 0 2 0 4
      board,← 5 5 0 0 0 0 5 0 2 0 1 0 5 0 2 0 0 3 5 0 0 0 3 3
      ⍞← board ← 8 6 ⍴ board
5 5 5 5 0 4
0 0 5 0 0 4
0 1 0 2 0 4
0 0 0 2 0 4
5 5 0 0 0 0
5 0 2 0 1 0
5 0 2 0 0 3
5 0 0 0 3 3
```

```
checkNurikabe board
```
1

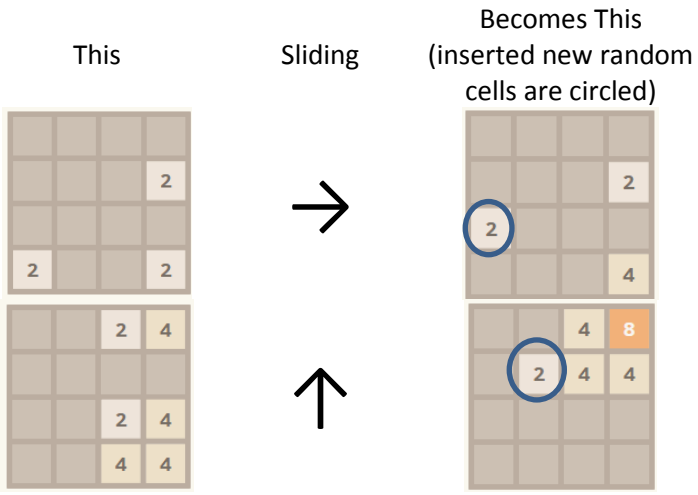
Recreation and Games Problem 2 (medium difficulty) – 2048 (2 tasks)

2048 is a game first implemented on the web in a weekend during March 2014 by Gabielle Cirulli. Since then, versions for the iPhone and Android platforms have also been developed. The code, largely written in JavaScript, can be found at https://github.com/gabrielecirulli/2048. The play of 2048 is simple:

- At the start, 2 random cells of a 4×4 grid are assigned a value of 2.
- The user indicates a direction (up, down, left or right) to slide the cells. This is done with arrow keys when played on a browser or by swiping in the direction on a touch-sensitive playing surface like a phone or tablet.
- Cells slide as far as possible in the chosen direction until they are stopped by either another cell or the edge of the grid. If two cells of the same number collide while moving, they will merge into a cell with the total value of the two cells that collided. The resulting tile cannot merge with another tile again in the same move. The merges are performed from the furthest to the nearest cell in the specified direction. For example: (0s indicate blank cells)

This				Sliding	Becomes This			
2	4	4	4	→	0	2	4	8
2	4	4	4	←	2	8	4	0

- After every move, a new tile with a value of either 2 (90% chance) or 4 (10% chance) will randomly appear in an empty cell of the grid.
- The user's score starts at zero, and is incremented whenever two tiles combine, by the value of the new tile.
- When the player has no legal moves (there are no empty spaces and no adjacent tiles with the same value), the game ends.



Task 1 – Shifty Thinking

Your task is to write a function, **`shift2048`**, which:

- takes a right argument which is a 4 element integer vector representing 4 cells (0 indicates a blank cell)
- takes a Boolean scalar left argument which indicates the direction to shift – 1 for shift to the right, 0 for shift to the left.
- returns a 4 element integer vector representing the result after the shift.

Examples:

```
      1 shift2048 2 4 4 8
0 2 8 8

      0 shift2048 2 4 4 8
2 8 8 0
```

Task 2 – All a board

Your task is to write a function, **`board2048`**, which:

- takes a right argument which is a 4×4 integer matrix representing a 2048 board (0s indicate blank cells)
- takes a left argument indicating the direction to shift using:
  - 0 for left
  - 1 for right
  - 2 for up
  - 3 for down
- returns a 2 element array where:
  - the first element is a 4×4 integer matrix resulting from the shift
    (do not insert a 2 or 4 into a random empty cell)
  - the second element is the score for the shift (the total of all cells that result from merges)

Example:

```
      ⍞← board ← 4 4 ⍴ 0 0 4 2 2 4 16 4 2 8 8 32 64 2 2 4
 0 0  4  2
 2 4 16  4
 2 8  8 32
64 2  2  4

      1 board2048 board
```

0	0	4	2	20
2	4	16	4	
0	2	16	32	
0	64	4	4	

Recreation and Games Problem 3 (high difficulty) – Can can you KenKen? (1 task)

KenKen is a type of arithmetic and logic puzzle invented in 2004 by Japanese math teacher Tetsuya Miyamoto. KenKen's goal is to fill a grid with digits — 1 through 4 for a 4×4 grid, 1 through 5 for a 5×5, etc. — so that no digit appears more than once in any row or any column. KenKen grids are divided into heavily outlined groups of cells (known as cages) and the numbers in the cells of each cage must produce a certain "target" number when combined using a specified mathematical operation (either addition, subtraction, multiplication or division).

- Digits may be repeated within a cage, as long as they are not in the same row or column.
- No operation is relevant for a single-cell cage: placing the "target" in the cell is the only possibility (thus being a "free space").
- The target number and operation appear in the upper left-hand corner of the cage.
- Because subtraction and division are non-associative, they are limited to 2-cell cages, but the numbers can appear in any order within the cage. For example: 2÷ could be solved with either 2,1 or 1,2.

6×6 KenKen Puzzle

11+	2÷		20×	6×	
	3-			3÷	
240×		6×			
		6×	7+	30×	
6×					9+
8+			2÷		

And its solution

11+ 5	2÷ 6	3	20× 4	6× 1	2
6	3- 1	4	5	3÷ 2	3
240× 4	5	6× 2	3	6	1
3	4	6× 1	7+ 2	30× 5	6
6× 2	3	6	1	4	9+ 5
8+ 1	2	5	2÷ 6	3	4

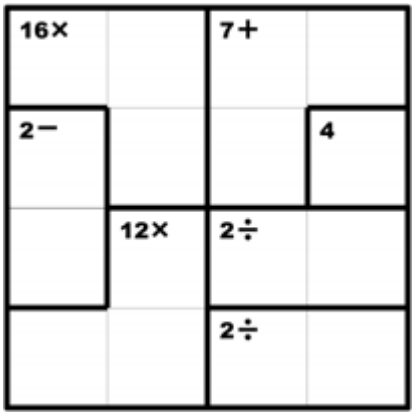
Task 1 – KenKen You Do It?

Your task is write a program, **KenKen**, which will solve a KenKen puzzle. The program:

- takes a 3-column matrix right argument
  - Column[;1] contains the integer target number
  - Column[;2] contains a character scalar representing the operation – one of + − × ÷
  - Column[;3] contains a vector of cell co-ordinates that make up the cage.
- returns an integer matrix representing the solution to the puzzle.

Example:

Using the puzzle to the right, the left argument would be:




```
      board←7 3⍴0
      board[;1]←16 7 2 4 12 2 2
      board[;2]←'×+- ×÷÷'
      board[;3]←((1 1)(1 2)(2 2))((1 3)(1 4)(2 3))((2
1)(3 1))(,⊂(2 4))((3 2)(4 1)(4 2))((3 3)(3 4))((4 3)(4
4))
      board
```

16	×	1 1 | 1 2 | 2 2
7	+	1 3 | 1 4 | 2 3
2	−	2 1 | 3 1
4		2 4
12	×	3 2 | 4 1 | 4 2
2	÷	3 3 | 3 4
2	÷	4 3 | 4 4

```
      KenKen board
2 4 1 3
1 2 3 4
3 1 4 2
4 3 2 1
```