2013 APL Problem Solving Competition – Phase II Problem Descriptions

This year, the problems in Phase II are divided into three sets representing three general categories: Math/Computer Science, Engineering and Biology. Unlike previous years, you do NOT need to solve all the Phase II problems to be considered eligible for the top three prizes. Instead, each set has three problems - one each of low, medium and high difficulty levels. To be eligible for consideration for one of the top prizes you 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 instance, 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 favor; if entries from two people are of similar quality but one has solved more or higher difficulty problems, that entry will receive higher consideration by the judging committee.

Good Luck and Happy Problem Solving!

Description of the Contest2013 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 may use the template workspace Contest2013.DWS

The Contest2013 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 (trad-fns), but you can change the implementation to dynamic functions (d-fns) if you care 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.
- The `#.Chart` and `#.Graph` functions described in the Biology problem set and the subset of RainPro, Dyalog's graphics package, necessary for `#.Graph`.

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 Contest2013.dyalog which will contain any code or data you placed in the #.Problems namespace. You will then upload Contest2013.dyalog file via the contest website.

If you use some other APL system, you may use the template script file `Contest2013.dyalog`

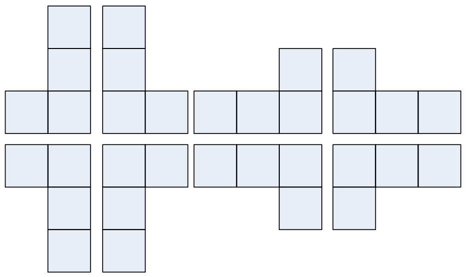
This file contains the correct structure for submission. You may populate it with your code, but do not change the namespace structure. Once you have developed your solution, edit the variable definitions as indicated at the top of the file and upload the file via the contest website.

Math/Computer Science Problem Set
Polyominoes - The Shapes of Things to Come…

From Wikipedia: "A polyomino is a plane geometric figure formed by joining one or more equal squares edge to edge. It may be regarded as a finite subset of the regular square tiling with a connected interior. Polyominoes are classified by how many cells they have."

Number of cells	Name
1	Monomino
2	Domino
3	tromino or triomino
4	Tetromino
5	Pentomino
6	hexomino
7	heptomino
8	octomino
9	nonomino or enneomino
10	decomino
11	undecomino or hendecomino
12	dodecomino

A polyomino can be rotated in 90° increments as well as reflected along its horizontal or vertical axis. While the monomino □ is fairly boring in this respect, higher order polyominoes get more interesting. For instance, the "L" tetromino can be presented in 8 different ways. The video game "Tetris" familiarized millions of people with this concept.



Problem 1 (low difficulty) – One of These Things Is Not Like the Other…

Given two Boolean matrices that each contain a polyomino, write an APL function named **SimilarPoly** that returns 1 if the second polyomino is some combination of rotation and/or reflection of the first polyomino and 0 otherwise. The left and right arguments for **SimilarPoly** are the Boolean matrices.

Example:

```
      ⎕←p1←2 3⍴1 0 0 1 1 1
1 0 0
1 1 1

      ⎕←p2←2 3⍴0 1 0 1 1 1
0 1 0
1 1 1

      ⎕←p3←3 2⍴1 1 0 1 0 1
1 1
0 1
0 1

      ⎕←p4←3 4⍴0 0 0 0 1 0 0 0 1 1 1
0 0 0 0
0 1 0 0
0 1 1 1

      p1 SimilarPoly p2
0

      p1 SimilarPoly p3
1

      p1 SimilarPoly p4
1
```

Problem 2 (medium difficulty) – In Good Shape?

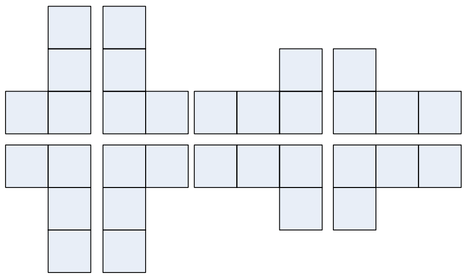
As seen in Problem 1, a polyomino can be represented by a Boolean matrix. However, not all Boolean matrices represent valid polyominoes. Write an APL function named **`ValidPoly`** which, when passed a Boolean matrix, will return a 1 if the matrix represents a valid polyomino and a 0 otherwise:

Examples:

```
      ⎕←p1←1 1⍴1
1

      ValidPoly p1
1

      ⎕←p2←2 3⍴1 1 1 0 1 0
1 1 1
0 1 0

      ValidPoly p2
1

      ⎕←p3←2 3⍴1 0
1 0 1
0 1 0

      ValidPoly p3
0

      ⎕←p4←3 3⍴1 1 1 1 0
1 1 1
1 0 1
1 1 1

      ValidPoly p4
1
```

Problem 3 (high difficulty) –Polyominoes - The Next Generation…

"Free" polyominoes are the shapes distinct from any other – that is, they're not a translation, rotation or reflection of another polyomino. Think of "Free" polyominoes as being the unique shapes from which other polyominoes can be created through some combination of rotation and/or flipping.

For instance,  are 8 representations of the same free polyomino.

The table below shows the number of free polyominoes based on the number of cells polyomino.

Number of cells	Name	Free Polyominoes
1	monomino	1
2	domino	1
3	tromino or triomino	2
4	tetromino	5
5	pentomino	12
6	hexomino	35
7	heptomino	108
8	octomino	369
9	nonomino or enneomino	1285
10	decomino	4655
11	undecomino or hendecomino	17073
12	dodecomino	63600

Write an APL function named **PolyGen** which, when passed an integer representing the number of cells, will compute and return a vector of Boolean matrices representing the set of free polyominoes for that number of cells. While your solution should work for any number of cells, it will only be validated up to octominoes. The elements of the result set can be in any order.

Example:

```
      PolyGen 4
 1   1 1   1 0   1 0   1 1
 1   1 0   1 1   1 1   1 1
 1   1 0   1 0   0 1
 1
```

```
      ]disp PolyGen 4   ⍝ you can use the ]disp user command to show structure
┌→┬────┬────┬────┬────┐
│1│1 1│1 0│1 0│    │
│1│1 0│1 1│1 1│1 1│
│1│1 0│1 0│0 1│1 1│
│1↓    ↓    ↓    ↓    ↓
└→┴~──→┴~──→┴~──→┴~──→┘
```

Engineering Problem Set

The Duck Test (for background see http://en.wikipedia.org/wiki/Duck_test)

Dyalog has just released APL for the Raspberry Pi and is using this to build a robot – if you are interested, you can read about this project on our blog at http://cto.dyalog.com. A “PiCam” has become available, and we are eagerly awaiting the delivery of a camera for our robot. At the moment the cameras are in such high demand that they will not be available to us until after the deadline for the programming contest, so we thought we would get some free programming help to prepare to do image processing while we wait for the camera to arrive!

Problem 1 (low difficulty) - Get the Picture?

We’ll limber up with an imaginary black-and-white camera that creates images containing only 0 for white and 1 for black – and we’ll carefully make sure that there is only one black object in the camera’s field of view while we practice our skills.

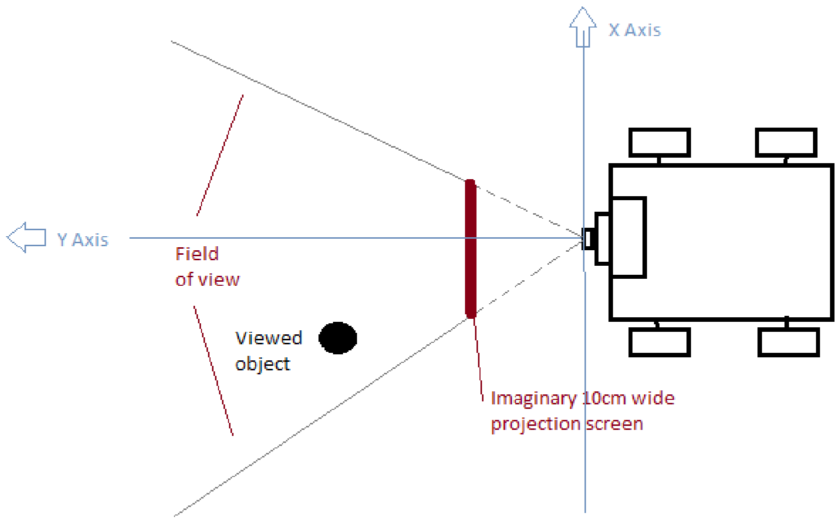
Given a matrix representing an image containing a single black object taken by a black-and-white digital camera, we want to determine the position of the object. Write a function **FindCG** which will return the co-ordinates of the “center of gravity” of the object, imagining that each pixel has the same weight. For example:

```
      img001
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0
0 1 1 1 1 0 0 0 0 0
0 0 1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

      2⍕FindCG img001 ⍝ Format result to 2 decimals
 ¯1.83 ¯0.61
```

Note that, since we have an even number of pixels on both dimensions, integer co-ordinate positions are “between” the columns of pixels; the center of the top left pixel has co-ordinates (`¯4.5 3.5`).

Problem 2 (medium difficulty) - But where *is* that exactly?

We have now mounted our 8 by 10 black and white camera on a robot so that it can take pictures in the direction ahead of it:



To avoid (or attack!) obstacles, our robot needs to translate the co-ordinates of objects in the digital image into geographical co-ordinates on the ground. We will make a couple of assumptions: first, that the floor the robot is driving on is perfectly horizontal. Second, since we can only guess at the shape of the objects, we will assume that a point in the image which is vertically below the center of gravity, half-way down toward the lowest observed point, is a good estimate for a point that is resting on the floor.

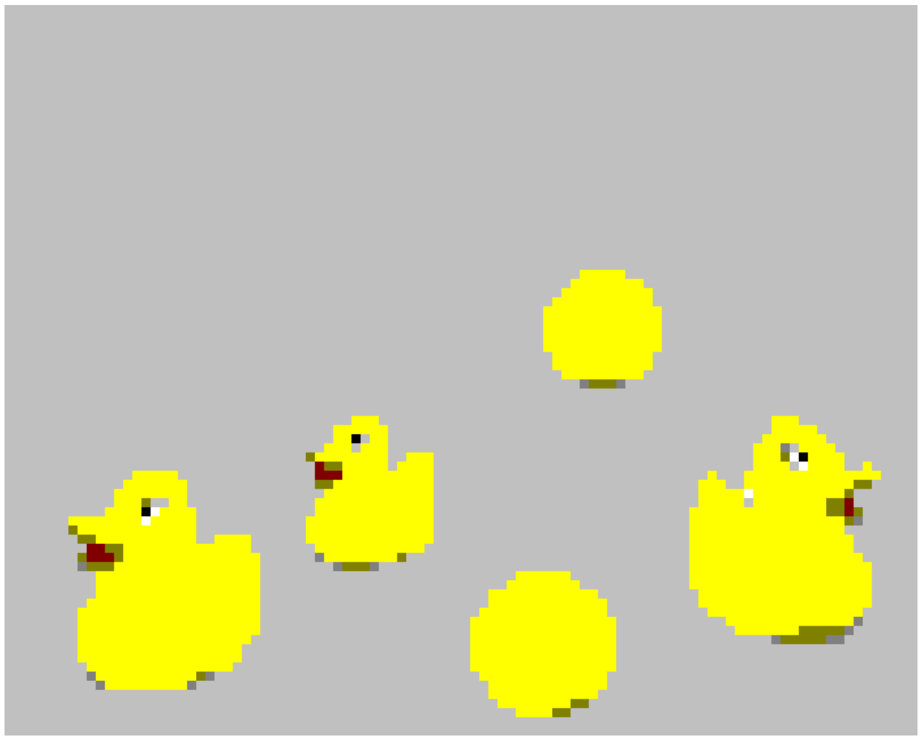
Our lens and image sensor have characteristics that allow us to think of each pixel as representing a 1x1cm square on a virtual image projected onto an 8cm by 10cm screen, mounted 5cm in front of the observer (camera). The camera is mounted 10cm above the ground.

Write a function **Locate** that takes the same argument as **FindCG**, but returns a 2-element vector that must contain the estimated X (negative left, positive right) and Y co-ordinates (forward) of a point where the viewed object is resting on the ground (distances to be reported in centimeters). Using the same input as above:

```
      1⍕ Locate img001
¯17.4 47.4
```

Problem 3 (high difficulty) – Quacker Tracker

Our camera has been upgraded, it now produces images in 16 colors – and the resolution has increased to an incredible 80 by 100 pixels! Just in time, because a bunch of yellow ducks have escaped, and we'd like to use our robot to locate them for us (we'll discuss what to do with them later). The geometry is the same as in problem 2, but now we're getting an image in which each pixel represents a 1mm by 1mm square projected onto our virtual screen – and in color:



Robots-eye view: 80 by 100 pixels, 16 colour bitmap

The ducks are trying to hide in a room that also contains a number of yellow balls, hoping that the image recognition algorithm will be unable to tell the difference between ducks and balls. The good news is that the ducks don't like each other (or the balls) very much, so they are staying apart - this means that they don't overlap in our pictures.

Write a function `FindDucks` that estimates the location and size of all ducks visible in the image. The right argument will be an 80 by 100 integer matrix with colour numbers, where 11 represents yellow. The result must be a matrix with one row for each duck and three columns containing values in centimeters that give:

[;1] The distance left (negative) or right (positive) from the robot's centerline
[;2] The "forward" distance from the camera to the duck
[;3] The estimated height of the duck

For example:

```
      1 ⍕FindDucks img002
 ¯5.6 28.8  9.2
 14.8 21.2 10.2
¯11.0 17.0  8.1

      ⍴img002
80 100
```

The data for the above example can be found as the variable named **`img002`** in the contest workspace. It can also be found in the file `img002.bmp`, which is a Windows bitmap file accessible using the 'BitMap' object – and finally it is provided as a text file containing one row of numbers per line called `img002.txt` (if you are running on a platform without bitmap support in APL).

Biology Problem Set

Cute as a bunny, cunning as a fox…

One of APL's many uses is building models of systems. In this problem set, we'll explore population models.

Problem 1 (low difficulty) - Life on the Wild Side

The goal of this problem is to build a simple population density model. Our model assumes that the population density of an organism has an equilibrium point; if the density exceeds this point, then the population will decrease and if the density is below the equilibrium, then it will increase. The rate at which the population increases or decreases has a linear dependence on the distance from the equilibrium. The following equation can be used to model the population for the next generation based on the current generation's observed population.

$$N_{t+1} = \left(1.0 - B\left(N_t - N_{eq}\right)\right) N_t$$

Where:

N_{eq}	= equilibrium population density
N	= observed population density
B	= slope of the reproductive curve
t	= generation number

Write two APL functions. The first, **NextGen**, takes as its right argument a 3-element vector comprising the slope of the reproductive curve, the equilibrium population size and the observed population and returns the population of the next generation. If the population decreases to 0 (or less), then it must return 0 (representing extinction). If the population grows so large that a **DOMAIN ERROR** occurs, then it must return the value **⌊/⍬**, which is the largest representable number and for our purposes here can be considered infinity.

Example:

```
      B←.011     ⍝ slope of the reproductive curve
      Neq←100    ⍝ equilibrium population density
      N←10       ⍝ observed population density
      ⎕PP←5      ⍝ set print precision to 5

      ⎕←g1←NextGen B Neq N    ⍝ compute generation 1
19.9

      ⎕←g2←NextGen B Neq g1   ⍝ compute generation 2
37.434

      ⎕←g3←NextGen B Neq g2   ⍝ compute generation 3
63.197

      ⎕←g4←NextGen B Neq g3   ⍝ compute generation 4
88.781

      ⎕←g5←NextGen B Neq g4   ⍝ compute generation 5
99.737

      ⎕←g6←NextGen B Neq g5   ⍝ compute generation 6
100.03
```

```
      ⎕←g7←NextGen B Neq g6  ⍝ compute generation 7
99.997

      ⎕←g8←NextGen B Neq g7  ⍝ compute generation 8
100

      ⎕←g9←NextGen B Neq g8  ⍝ compute generation 9
100
```

Beginning with the 8th generation, this population hits its equilibrium population density and will stay there.

The second APL function is named **GenX** and must have the same right argument as **NextGen**, but will also take an integer left argument representing the generation number whose population density is returned. Note: Non-looping solutions will be given greater consideration by the judges.

Example:

```
      ⎕←6 GenX B Neq N   ⍝ compute generation 6
100.03
```

Problem 2 (medium difficulty) – Some days you're the predator, others you're the prey…

Species can interact directly with one another by predation (meaning that one species eats another). The population model described in the previous problem can be modified to take into account the effect of a predatory species as follows:

$$N_{t+1} = \left(1.0 - B\left(N_t - N_{eq}\right)\right) N_t - C N_t P_t$$

Where:

P_t	= population density of predators in generation t
C	= a constant measuring the efficiency of the predator

all other terms are as described in the previous problem

The slope of the prey reproductive curve, B, can be computed using:

$$B = \frac{R - 1}{N_{eq}}$$

Where:

R	= maximum reproductive rate of the prey
N_{eq}	= equilibrium population density of the prey

If the predator reproductive rate is dependent on the number of prey available, then the predator population can be modeled as follows:

$$P_{t+1} = Q N_t P_t$$

Where:

P	= population density of the predator species
N	= population density of the prey species
t	= generation number
Q	= a constant measuring the efficiency of the utilization of prey for reproduction by predators

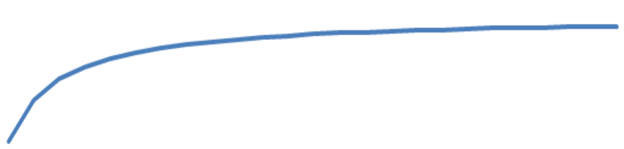
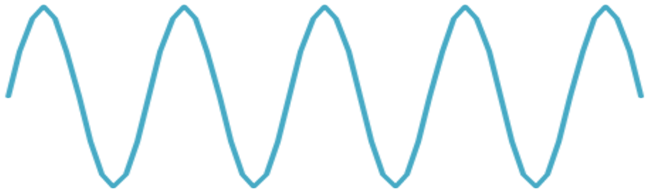
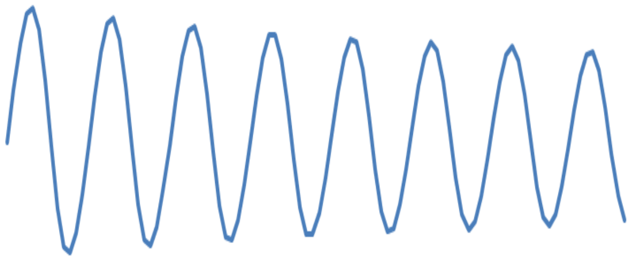
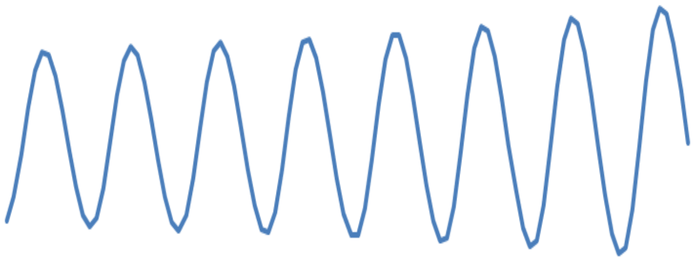
The constant Q can be computed as:

$$Q = \frac{S}{N_{eq}}$$

Where:

S	= maximum reproductive rate of the predator
N_{eq}	= equilibrium population density of the prey

There are four possible outcomes from these equations:

Stable equilibrium without oscillation - the population remains stable and does not oscillate 	Stable oscillation – similar to a sine wave, the population oscillates 
Convergent (damping) oscillation – the population oscillates and converges 	Divergent (growing) oscillation leading to the extinction of one or both species - the population oscillates but diverges 

Write 2 APL functions. The first function, **PredPrey**, will simulate a predator-prey model based on your inputs. The left argument is the number of generations to run. The right argument is a 6-element vector comprising the prey equilibrium population, the prey observed population, the prey reproductive rate, the predator observed population, the predator reproductive rate and the predator efficiency, C, from above. The result of **PredPrey** is a 2 row numeric matrix where the prey population for each generation is in the first row and the predator population for each generation is in the second row.

Example:

```
      gen←6   ⍝ number of generations to run
      Neq←100 ⍝ prey population equilibrium density
      N←50    ⍝ observed prey population density
      P←.2    ⍝ observed predator population density
      R←1.5   ⍝ prey reproductive rate
      S←2     ⍝ predator reproductive rate
      C←.5    ⍝ predator "efficiency" rate

      ⎕←model←0 2∘⍕¨gen PredPrey Neq N R P S C ⍝ format to 2 decimal places
  50.00   57.50   63.97   68.14   68.97   65.84
  0.20    0.20    0.23    0.29    0.40    0.55

     'Generation' 'Prey' 'Predator'⍪(⍳gen),⍉model ⍝ make an ad hoc report
 Generation  Prey    Predator
          1   50.00   0.20
          2   57.50   0.20
          3   63.97   0.23
          4   68.14   0.29
          5   68.97   0.40
          6   65.84   0.55
```

The second function, **OutcomeType** , takes the output from **PredPrey** and returns a character vector indicating which of the four possible outcomes the model represents.

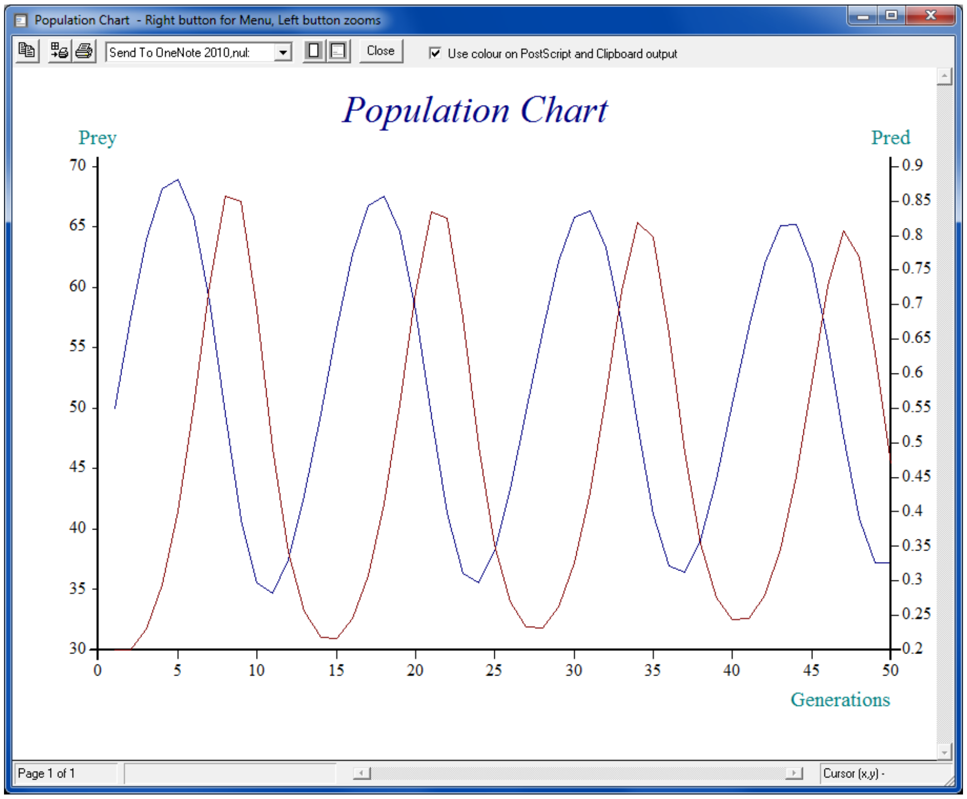
OutcomeType will return one of **'equilibrium' 'oscillation' 'convergent'** or **'divergent'** as appropriate.

Example:

```
      OutcomeType 100 PredPrey Neq N R P S C
convergent
```

The template workspace and script files contain a function, **`#.Chart`**, which you can use to produce simple character-based charts of time series data. The right argument to **`#.Chart`** is a numeric vector representing a single time series or a matrix where each row represents a time series. The optional left argument to **`#.Chart`** is an integer scalar indicating the height of the chart. The template workspace also contains a function, **`#.Graph`**, which uses the RainPro graphics package supplied with Dyalog APL to produce higher resolution graphs. **`#.Graph`** takes a right argument like the one used by **`#.Chart`**. If you are using the Linux version of Dyalog APL, you must supply a left argument which is a character vector of the name of the image file to write. If you are using the Windows version of Dyalog APL, omitting the left argument will cause **`#.Graph`** to display a window containing the graph produced.

```
      model←50 PredPrey Neq N R P S C ⍝ run 50 generations
      #.Graph model
```



```
    11 #.Chart model ⍝ produce an 11 row character-based chart
68.97-   **  oo        *  o                             -0.86
     |     *          * *  o       **  oo       **  oo  |
     |  *   o        *   o        *  *o        *  *o    |
     |      *  o         *  o            o              |
     | *            *            *    *       *    *  o |
     |     o            o            o            o     |
52.00-*      *  o  *      *  o  *         o  *          -1.00
     |                 o            o  *         o  *  o|
     |    o       *           o*           o*           |
     |   o    *  o    o    *       o    *       o    *  |
     |  o        *o  o      * *oooo      ***oooo      **|
34.67-oo       **  oo        *                          -0.20
```

Problem 3 (high difficulty) – Bunnies and Foxes

In Problem 2, you developed **PredPrey** to model the interaction of predator and prey species. In Problem 3 you will use **PredPrey** to model an island that is populated with bunnies (prey) and foxes (predators) as follows.

The island is represented by an M×M array of cells and is populated with starting populations of bunnies and foxes. More than one of a species can occupy a cell, as can more than one species. For instance, a cell could have 5 bunnies and 2 foxes.

For each "generation" of the island:

- Calculate the next generation populations
  - For each cell containing predator or prey, use the 7×7 cell block surrounding the current cell to compute the observed population density.

    For example, consider the population density calculations using 3×3 cell blocks in the following 16-cell island; the numbers represent the count of individuals in each cell.

4	3	4	0
2	2	3	3
0	5	4	2
3	0	2	1

    The population density is calculated by totaling the population in a 3×3 block and then dividing by the number of cells in the block. Cells that don't have a complete surrounding block should use the number of surrounding cells available. This is demonstrated below showing the population densities for the yellow-highlighted cells.

4	3	4	0
2	2	3	3
0	5	4	2
3	0	2	1

```
  ⎕←N←(+/4 3 4 2 2 3 0 5 4)÷9
3
```

4	3	4	0
2	2	3	3
0	5	4	2
3	0	2	1

```
  ⎕←N←(+/2 2 0 5 3 0)÷6
2
```

4	3	4	0
2	2	3	3
0	5	4	2
3	0	2	1

```
  ⎕←N←(+/4 2 2 1)÷4
2.25
```

  - use **PredPrey** to project the population density of the next generation

- Apply the next generation projection and, based on the projected density, compute the probability that each individual in the cell procreates or dies.
  - When the projected population is greater than the observed population, use the increase to determine the probability that an individual procreates.
    - For example, if the observed and projected populations are 40 and 50 respectively, then there is a 25% ((50-40)÷40) chance of that individual procreating.
  - If the probability exceeds 100%, limit the increase to a litter size of 4 for bunnies and 2 for foxes.
    - For example, if the observed and projected bunny populations are 2 and 20 respectively, then the chance of that individual procreating is 900% ((20-2)÷2). Limit the increase to the litter size of 4.
    - If the percent is fractional and smaller than the litter limit, then use the fractional part to determine the probability for the last individual. For example, if the observed and

        projected populations are 2 and 7 respectively (250% = (7-2)÷2), then increase the population by 2 and use a 50% probability to determine if there is an additional individual born.

    - When the projected population is less than the observed population, use the decrease to determine the probability that an individual dies.
        - For example, if the projected and observed populations are 2 and 4 respectively, then there is a 50% chance that the individual will die.
- Randomly migrate the populations
    - Each individual of each species may (or may not) move up to one cell in any direction. Individuals cannot fall off the edge of the island, so individuals in cells located at the edges of the island have fewer choices of where to move.

Repeat the above for the specified number of generations or until one or both species perish, whichever comes first.

The data structure to represent the island will be a 3-dimensional array of shape 2×M×M. The first plane will contain the bunny population and the second plane the fox population.

Write a function called **BunnyFox** that takes as its left argument an array representing the initial state of the island and as its right argument a 5-element vector comprising the number of generations to run, the prey equilibrium population density, the prey reproductive rate, the predator reproductive rate and the predator efficiency. **BunnyFox** must return as its result the state of the island as a 2×M×M integer matrix. The contest template workspace and script files have 3 variables, **Island10**, **Island50**, and **Island100**, containing representations of 10×10, 50×50 and 100×100 islands.

```
      gen←100  ⍝ number of generations to run
      Neq←100  ⍝ prey population equilibrium
      R←1.5    ⍝ prey reproductive rate
      S←2      ⍝ predator reproductive rate
      C←.5     ⍝ predator "efficiency" rate

      ⎕←island←Island10 BunnyFox gen Neq R S C
3 3 3 2 1 4 1 3 1 0
0 2 3 1 0 0 3 1 0
2 2 2 2 0 1 4 3 2 3
1 0 4 2 0 4 0 1 2 2
3 3 2 0 0 4 0 1 3 3
0 2 4 1 4 4 1 1 2 4
4 1 3 3 2 1 3 0 1 0
2 0 0 2 2 0 3 1 4 0
4 0 0 2 3 4 1 1 3 2
3 3 3 3 0 1 1 2 3 3

1 0 0 1 0 2 0 0 1 0
2 1 1 2 0 1 2 1 0 1
1 2 0 1 2 1 0 1 2 0
0 1 2 2 0 0 0 1 0 1
0 2 1 1 0 2 1 2 2 2
2 1 2 1 1 1 1 1 0 0
1 1 0 0 0 2 1 2 2 0
2 0 1 0 1 0 2 1 0 0
1 1 0 2 0 2 2 0 1 0
1 0 1 2 2 2 0 2 1 0
```

N.B. the above result is for purposes of example only, you should test your solution with larger island sizes.