2017 APL Problem Solving Competition – Phase 2 Problem Descriptions

Overview

This year's Phase 2 problems are divided into three sets, representing three categories – Healthcare, Maths and Statistics, and Computational Geometry. Each problem set has one or more problems consisting of one or more tasks.

To be considered for the grand prize, you must solve all the problems and tasks in any single category. You are encouraged to complete as many categories as possible. The judging committee will consider the level of effort put forth as one of the criteria for prize selection. For instance, if two submissions for a category are of comparable quality, but one of the submissions includes a second category, the judging committee may take that into consideration for tie-breaking.

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 Contest2017 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 `Contest2017.DWS`

The Contest2017 workspace contains:

- Three namespaces, one for each of the problem categories. Each namespace contains:
  - stubs for all of the functions described in the problem descriptions. The function stubs are implemented as traditional APL functions (tradfns) but any type of function is acceptable.
  - any sample data elements mentioned in the problem descriptions.

  Any sub-functions that you develop as a part of your solution should be co-located in the category namespace.

  The namespaces are:

  - `#.hlth` – for the healthcare problem set
  - `#.math` – for the maths and statistics problem set
  - `#.geom`– for the computational geometry problem set

- `#.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 **Contest2017.dyalog** which will contain any code or data you placed in the `#.hlth`, `#.math`, and `#.geom` namespaces. You will then need to upload the **Contest2017.dyalog** file using the contest website.

If you use some other APL system, you can use the template script file `Contest2017.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**, so your solution can only use APL features that are common between your APL system and Dyalog.

Healthcare Problem Set

The Healthcare problem set contains 1 problem with 4 tasks.
To be considered for a prize in this category, you must complete all 4 tasks.

Problem 1 – Drug Program Modelling

Task 1* – *Write a function named* `Projection` *to model this patient population

Background:
In a group of disease sufferers, 120,000 people are being treated for a particular strain of the disease.

- On average, 2.5% of the total treated patient population will go into remission (that is, they will no longer require treatment) **each month** after treatment.
- On average, a further 3.25% will die **each year** from the disease or due to complications resulting from it.

The treatment for this disease consists of three drugs A, B and C. Patients start on drug A and move through B and on to C depending on their response (or lack of it).

At time zero (T0) the group has 70,000 patients taking drug A, 30,000 taking drug B and 20,000 taking drug C; consider remission and death rates to be identical for all 3 drugs.

`Projection` ***has the following syntax:***

```
(sa sb sc rem dec) ← years Projection a b c
```

The right argument consists of:

`a`, `b`, and `c` – the number of patients currently taking drugs A, B, and C respectively

The left argument consists of:

`years` – the number of years to run the projection

The result represents the population at the end of the projection and consists of:

`sa`, `sb` and `sc` – the number of surviving disease sufferers taking drugs A, B and C respectively

`rem` – the number of patients in remission

`dec` – the number of deceased patients

Task 2* – *Write a function named* `Projection2` *to model the patient population and drug progression

Background:

Each month, 1,000 to 3,000 new occurrences are identified. For the purpose of this problem, you can assume a uniform (random) distribution. The side effects of each drug in the progression are increasingly severe, resulting in patients dropping out of the program.

For new patients:

- 80% are given drug A, meaning 20% of new patients are not given any drug therapy.
- For drug A, 70% of patients respond well and stay on it until they enter remission or die. 30% move on to drug B after 2 months.
- For drug B, 10% of the patients drop out of the treatment program immediately. Of the remaining 90%, 60% of patients respond well and stay on it until they enter remission or die and 40% move on to drug C after 3 months.
- For drug C, 20% of the patients drop out of the treatment program immediately. The remaining 80% stay until they enter remission or die.

`Projection2` has the following syntax:

```
(sa sb sc rem dec drop not) ← years Projection2 a b c
```

The right argument consists of:

`a`, `b`, and `c` – the number of patients currently taking drugs A, B, and C respectively

The left argument consists of:

`years` – the number of years to run the projection

The result represents the population at the end of the projection and consists of:

`sa`, `sb` and `sc` – the number of surviving disease sufferers taking drugs A, B and C respectively

`rem` – the number of patients in remission

`dec` – the number of deceased patients

`drop` – the number of patients who have dropped out

`not` – the number of people in the group who are not being treated

***Task 3** – Write a function named* `Projection3` *to model the patient populations in each country*

Background:

The study is extended to include other countries as shown in the table. Broadly speaking, the responsiveness of patients to their respective drugs is the same for each country. You can assume that there are 1,000 to 3,000 new occurrences each month, per country, also uniformly distributed.

	T0 group patients		
	Drug A	**Drug B**	**Drug C**
England	70,000	30,000	20,000
France	65,000	32,000	12,000
Germany	40,000	19,000	10,000
Spain	93,000	31,000	26,000
Italy	55,000	35,000	18,000

`Projection3` has the following syntax:

```
results ← years Projection3 drugs
```

The right argument is a 3-column matrix with one row per country and columns being:

`[;1]` – the number of patients currently taking drug A

`[;2]` – the number of patients currently taking drug B

`[;3]` – the number of patients currently taking drug C

The left argument is:

`years` – the number of years to run the projection

The result is a 7-column matrix with one row per country columns being:

`[;1]` – the number of surviving patients taking drug A

`[;2]` – the number of surviving patients taking drug B

`[;3]` – the number of surviving patients taking drug C

`[;4]` – the number of surviving patients in remission

`[;5]` – the number of deceased patients

`[;6]` – the number of patients who have dropped out

`[;7]` – the number of people who are not being treated

***Task 4** – Write a function named* `DrugD` *to model the effect of the drop-out rate for DrugD*

Background:

It has been found that the original disease progression data was inaccurate.

- Drug B is effective in 35% of patients, not 60%.
- Drug C is effective in 40% of patients.
- Patients who are not managed (patients who either never started the program or who dropped out) have a death rate of 11%.

A new drug, drug D, is due to be introduced. It will be used in patients who are not responsive to drug C after 3 months of treatment. The death rate for patients treated with drug D is 1% and it has a remission rate of 5%. However, the side effects of drug D are more serious and, as a result, it is expected that a percentage will drop out of this treatment per month.

Update the model to reflect the new data and introduce drug D.

`DrugD` has the following syntax:

```
(sa sb sc sd rem dec drop not) ← dropRate DrugD a b c
```

The right argument consists of:

`a`, `b`, and `c` – the number of patients currently taking drugs A, B, and C respectively

The left argument consists of:

`dropRate` – the monthly drop-out rate for patients taking drug D

The result represents the population at the end of the projection and consists of:

`sa`, `sb`, `sc` and `sd` – the number of surviving disease sufferers taking drugs A, B, C and D respectively

`rem` – the number of patients in remission

`dec` – the number of deceased patients

`drop` – the number of patients who have dropped out

`not` – the number of people in the group who are not treated

Maths and Statistics Problem Set

The Maths and Statistics problem set contains 3 problems with total of 7 tasks.
To be considered for a prize in this category, you must complete all 7 tasks.

Problem 1 – What's the Significance?

Significant figures of a number are digits that carry meaning contributing to its measurement resolution. For the purposes of this problem, the rules for identifying significant figures are:

- All non-zero digits are significant.
- All zeros between the first and last significant digits are significant.
- If a number has neither decimal point, nor non-zero digits, then **one** zero, if any is present, is significant.
- All other leading zeros (those to the left of the leftmost significant digit) are **not** significant.
- In a number **without** a decimal point, trailing zeros (those to the right of the rightmost significant digit) are **not** significant.
- In a number **with** a decimal point, trailing zeros (those to the right of the decimal point or rightmost significant digit) **are** significant.

***Task 1** – Write a function named* `SigFig` *that will determine the number of significant figures*

`SigFig` has the following syntax:

```
r ← SigFig num
```

The right argument consists of:

`num` – a number, but may be given as a character scalar or vector, because APL will remove leading zeros and trailing zeros to the right of a decimal point. For example, APL treats 100.00 as 100. You need not consider them numbers using scientific notation (e.g. `1.234E56`) or complex numbers (e.g. `1.2J3.4`).

The result is the number of significant figures in num

Examples:

```
      SigFig ''
0
      SigFig 0
1
      SigFig 4500
2
      SigFig '700.'
3
      SigFig 1023
4
      SigFig '0.00000'
5
      SigFig '6700.00'
6
```

Task 2 **– *Write a function named* SRound *which returns the next higher* s-round *number***

Now let's call a number *s-round* if it has s or fewer significant digits.
A number that is s-round is also (s+n)-round for n>0.
For example, 12000, 1.2, and 100 are 2-round; 100 is also 1-round.

SRound has the following syntax:

```
r ← s SRound num
```

The right argument consists of:
num - a single number
The left argument consists of:
s - a single positive integer which represents the maximum number of significant digits
The result is the first s-round number greater than or equal to num

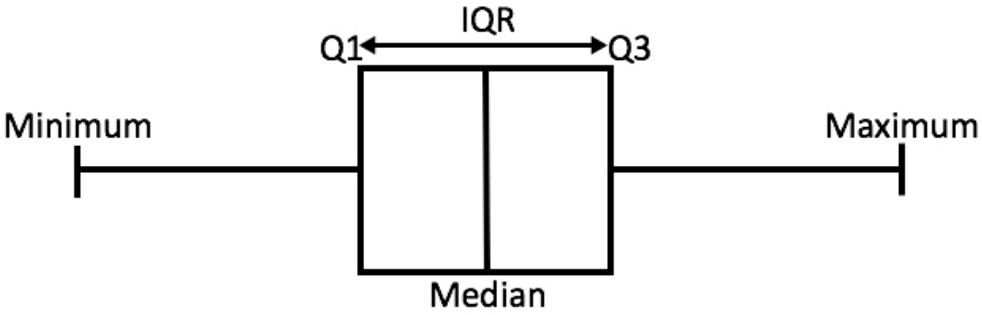
Examples:

```
      1 SRound 12345
20000

      2 SRound 100
100

      2 SRound 101
110

      3 SRound 0.314159
0.315
```

Problem 2 – Outlying in a Field

Background:

An *outlier* is an observation that lies an abnormal distance from other values in a random sample from a population.



There are many ways to define "abnormal distance" but for this problem we will be using the interquartile range (IQR) to measure whether or not a point in a dataset is an outlier. There are two common ways to compute quartiles when the data set has an odd number of elements, inclusive and exclusive, which dictate whether the middle value of the data set is included or excluded when determining the medians of the lower and upper portions of the data set. For the purposes of this problem, we will use the exclusive method:

1. Sort the data set.
2. Use the median to divide the ordered data set into two halves.
    - If there are an odd number of elements in the original ordered data set, **do not include** the median (the central value in the ordered list) in either half.
    - If there are an even number of elements in the original ordered data set, split this data set exactly in half.

3. The lower quartile value is the median of the lower half of the data. The upper quartile value is the median of the upper half of the data.

The interquartile range (IQR) is the difference between the upper and lower quartiles.

***Task 1** – Write a function* `IQR` *which finds the interquartile range of a vector of numbers*

`IQR` has the following syntax:

```
r ← IQR vec
```

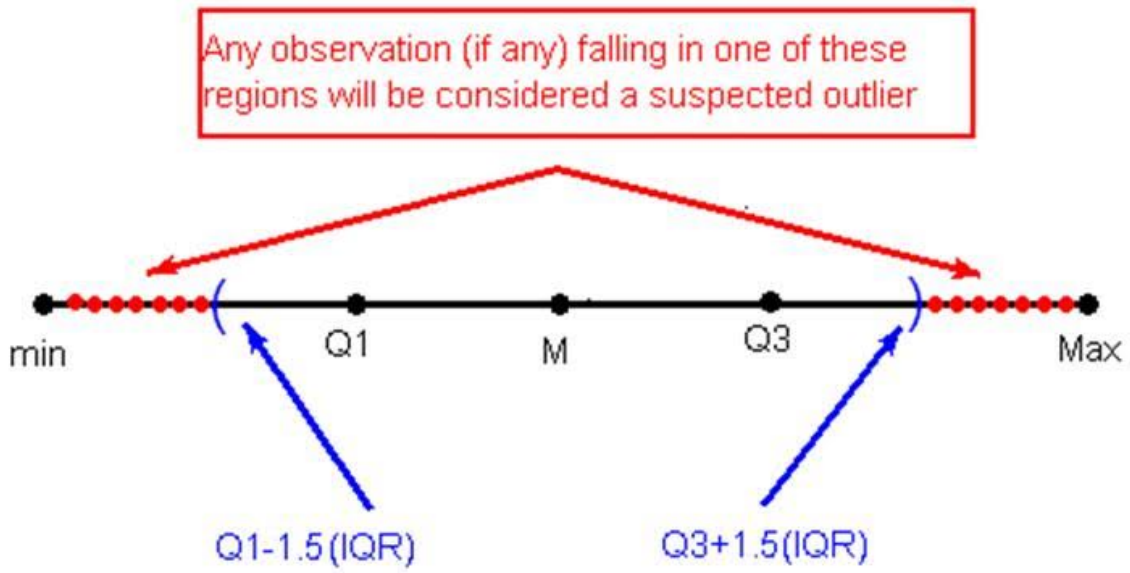
The right argument consists of vec - a numeric vector of at least 2 elements.
The result is a single number representing the vector's interquartile range

Examples:

```
      IQR 5 1 3 2 4
3
      IQR 20 15
5
```

Finding Outliers

For the purposes of this task we will define two severities of outliers: mild and extreme. An extreme outlier is a value that is either greater than `Q3+(3×IQR)` or less than `Q1-(3×IQR)`. A mild outlier is one that is either `Q3+(1.5×IQR)` or less than `Q1-(1.5×IQR)` and is not an extreme outlier.



Task 2 *– Write a function* `Outliers` *that identifies outliers in a vector of numbers*

`Outliers` has the following syntax:

```
r ← Outliers vec
```

The right argument consists of:

vec – a numeric vector of at least 2 elements

The result r is a numeric vector of the same length as vec where **`r[i]`** is:

`¯1` if **`vec[i]`** is an extreme outlier
`1` if **`vec[i]`** is a mild outlier
`0` otherwise

Example:

```
      Outliers 0 2 3 100 4 5 6
1 0 0 ¯1 0 0 0
```

Problem 3 – Clustering

Background:

Cluster analysis or *clustering* is the tasks of creating grouping within a dataset where each element in a group can be said to be *more similar* to the other elements in its group than those of other groups. Clustering has many useful applications in a variety of fields such as statistical data analysis, machine learning and pattern recognition, just to mention a few.

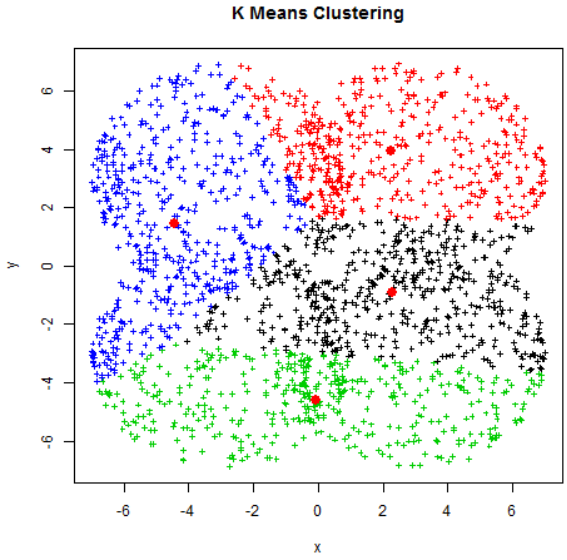
There is a range of clustering methods. One popular methodology, centroid-based clustering, uses a central vector to define each cluster. k-Mean clustering, another centroid-based clustering method, is a process whereby centers for each cluster are established and the data objects are assigned to the nearest of the k cluster centres such that the squared distances from the cluster centres are minimized.

First some definitions:

Euclidian Distance - The Euclidian Distance between two points A and B where $A = (a_1, a_2, \dots, a_n)$ and $B = (b_1, b_2, \dots, b_n)$ is:

$$= \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

Cluster Mean - The cluster mean is the average of the points in the cluster. For example, if you have the cluster: {(1,1) (2,2) (3,2)} then the cluster mean is the point (2,1.66)



When defining clusters using the k-Means method, the first step is to find an initial partition. This is done by selecting k points at random where k is the number of clusters you are trying to find. Next find the Euclidian distance from each from the k initial points to every other point in the dataset and as first pass at defining clusters on the data, group each point with the point from the initial partition that it is closest to (closest means has the smallest Euclidean distance).

The data is now clustered but there is no guarantee that all the points are in the right cluster. Calculate the cluster means and compare the Euclidean distance of each point with each of the cluster means. If it turns out that a point is closer to another cluster mean than the cluster it was assigned in the first pass, reassign the point to the cluster with the closer cluster mean. Let the iterative reallocation continue until no more reallocations occur (although note that there are cases where there is no exact solution hence it is advisable to put a reasonable limit on the number of iterations attempted).

***Task 1** – Write a function* `dist` *to compute the Euclidian distance between 2 points.*

`dist` has the following syntax:

```
r ← a dist b
```

The left and right arguments are each single coodinates in n-space.
The result a scalar number representing the Euclidian distance between the coordinates.

Example:

```
      1 1 dist 3 4          ⍝ distance in 2-space
5
      4 3 2 1 dist 5 6 7 8 ⍝ should work in any n-space
9.16515139
      0 dist 0              ⍝ even no space
0
```

***Task 2** – Write a function* **cMean** *to compute the cluster mean of a set of coordinates.*

cMean has the following syntax:

```
      r ← cMean coords
```

The right argument is a vector of coordinates in n-space
The result is a single n-space coordinate representing the cluster mean

Example:

```
      coords ← (1 1) (1.5 2) (3 4) (5 7) (3.5 5) (4.5 5) (3.5 4.5)

      cMean coords
3.142857143 4.071428571

      cMean 1 3 5 7 9  ⍝ 5 coordinates in 1-space
5

      cMean ,⊂1 3 5 7 9 ⍝ 1 coordinate in 5-space
1 3 5 7 9
```

***Task 3** – Write a function* **Cluster** *to compute cluster membership for a set of coordinates*

Cluster has the following syntax:

```
      r ← k Cluster coords
```

The right argument "coords" is a vector of coordinates in n-space.
The left argument "k" which is a single integer representing the number of clusters
The result is an integer vector where each element [n] represents the cluster number for coords[n]

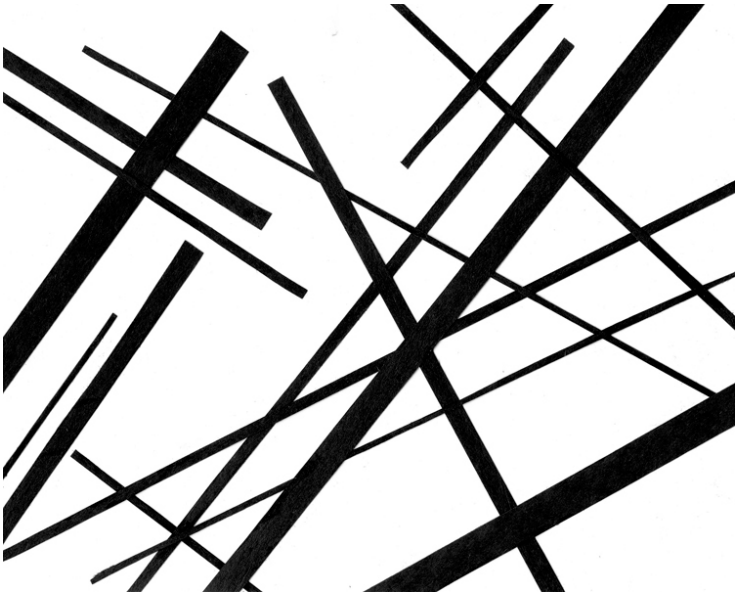
Example:

```
      coords ← (1 1) (1.5 2) (3 4) (5 7) (3.5 5) (4.5 5) (3.5 4.5)
      k←2

      k Cluster coords
1 1 2 2 2 2 2
```

Computational Geometry Problem Set

The Computational Geometry problem set contains 3 problems with a total of 6 tasks.
To be considered for a prize in this category, you must complete all 6 tasks.

Problem 1 – Please Wait at the Intersection



Backgound:

One technique to determine if two line segments intersect is to use orientation. For more information on the use of orientation, see http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/.

Task 1 – Write a function `Intersects` ***to determine if two line segments intersect***

`Intersects` has the following syntax:

```
r ← seg1 Interects seg2
```

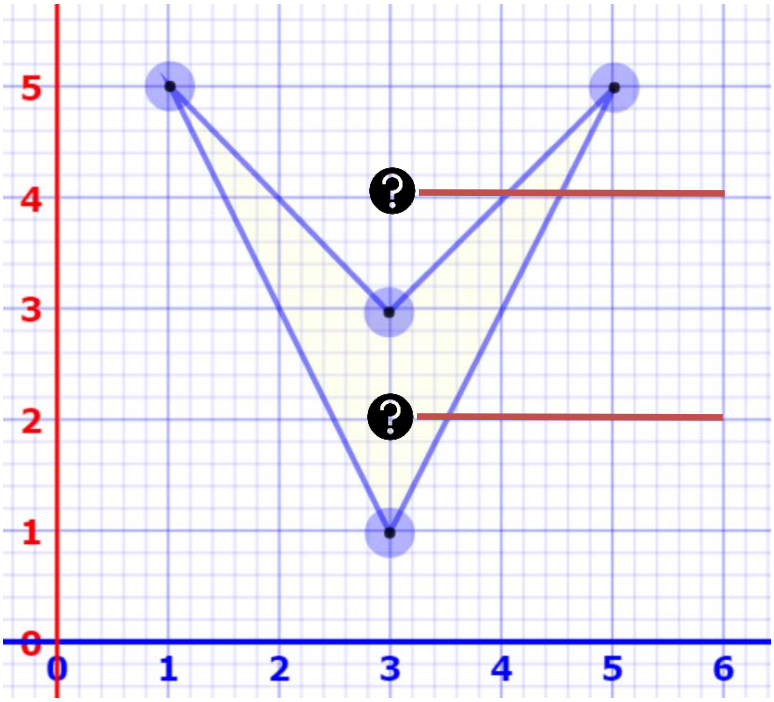
The right and left arguments are each a vector of 2 coordinates denoting the endpoints of a line segment
The result is a Boolean scalar where 1 indicates the line segments intersect and 0 indicates that they do not

Examples:

```
      (0 0)(2 0) Intersects (1 1)(2 2)
0

      (1 1)(5 5) Intersects (5 1)(1 5)
1

      (0 0)(3 3) Intersects (3 3)(5 5)
1
```

In or Out?

It's possible to determine whether a point is within the boundaries of a polygon by extending a horizontal line from the point to another point beyond the boundaries of the polygon and then counting the number of intersections with the polygon's edges. If the number of intersections is odd, the point lies within the polygon and if the number of intersections is even, the point does not lie within the polygon.



***Task 2** - Write a function* `isInside` *to determine if a point lies within a polygon.*

`isInside` has the following syntax:

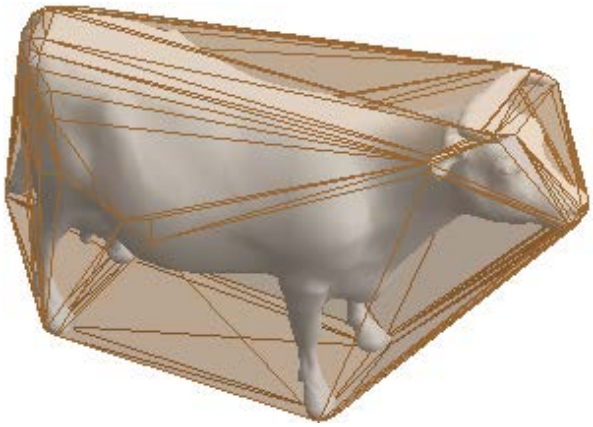
```
r ← point isInside nodes
```

The right argument `nodes` is a vector of node coordinates for the polygon drawn in either a clockwise or counterclockwise manner.
The left argument `point` is a 2-element vector of the (x,y) coordinates for the point to be tested.
The result `r` is a Boolean scalar where 1 indicates the point lies within the polygon, and 0 that it does not.

Examples:

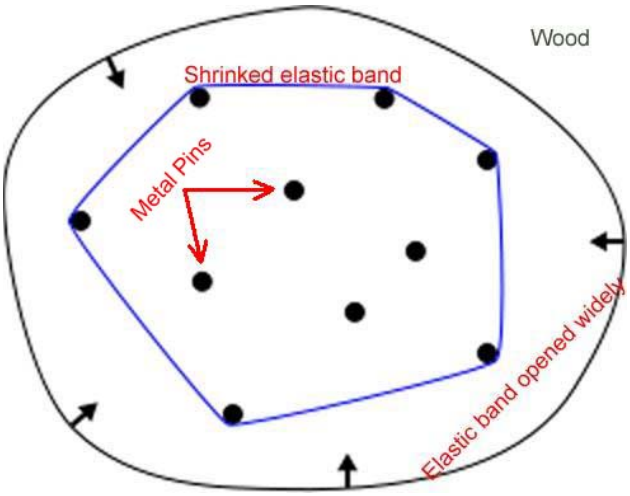
```
      coords ← (1 5)(3 3)(5 5)(3 1)

      3 2 IsInside coords
1

      3 4 IsInside coords
0
```

Problem 2 – The Convex Hull: How to Put a Cow in a Box



Background:

Imagine metal pins nailed into a wooden board, scattered about like points on the Cartesian Plane. Now imagine stretching an elastic band until it is large enough to encircle all the pins on the board, and letting it go. The elastic band will shrink and snap round the outer most nails. The outer segments of the elastic band form the convex hull of the set of pins.



In more mathematical terms, the convex hull of a given set of points in the plane is the unique convex polygon whose vertices are points from the set and contains all points in the set. In other words, the convex hull of a set of points is the smallest convex sub-set containing the set.

The convex hull is one of the first problems that was studied in computational geometry. In fact, convex hull is used in different applications such as collision detection in 3D games and Geographical Information Systems and Robotics.

A discussion of algorithms to compute the convex hull of a set of points can be found at https://en.wikipedia.org/wiki/Convex_hull_algorithms#Algorithms.

Task 1 **–** ***Write a function*** `ConvexHull` ***to compute the convex hull of a set of points.***

`ConvexHull` has the following syntax:

```
r ← ConvexHull points
```

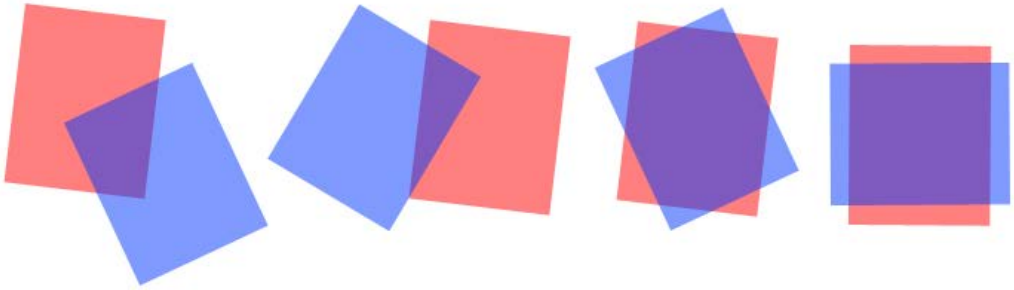
The right argument `points` is a vector of (x,y) pairs, in random order.
The result `r` is a vector of ordered (either clockwise or counterclockwise) points which define the convex hull for the point set.

Examples:

```
points ← (1 1)(2 2)(3 3)(1 3)
CovexHull points
```

1 1	3 3	1 3

Problem 3 – AND it is a Rectangle! OR is it…



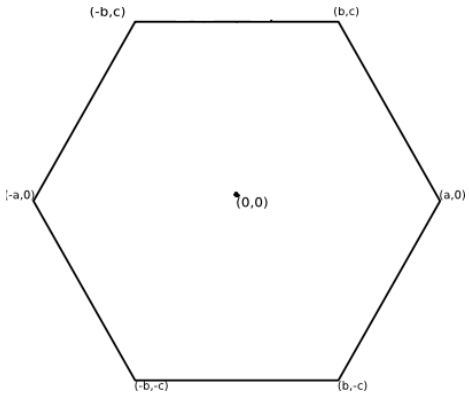
Background:

Boolean operations on polygons are often used in computer graphics and pop up in things like CAD and EDA software. The concepts of Boolean operation on polygons have ties to set operations in set theory: an AND of two polygons is equivalent to the intersection between two sets, similarly an OR of polygons is equivalent to the union of sets.

The difference between a set understanding and an understanding grounded in computational geometry are in the approach. Sets exist at a high level of abstraction and often contain infinite elements and an infinite list will certainly raise some hardware concerns!

A way around this (at least for simple polygons) is to define a polygon not by the points it encompasses but rather by its vertices (or corners). Simple polygons have a finite number of vertices, and when the corners are listed in a consistent manner a simple polygon can be unambiguously defined.

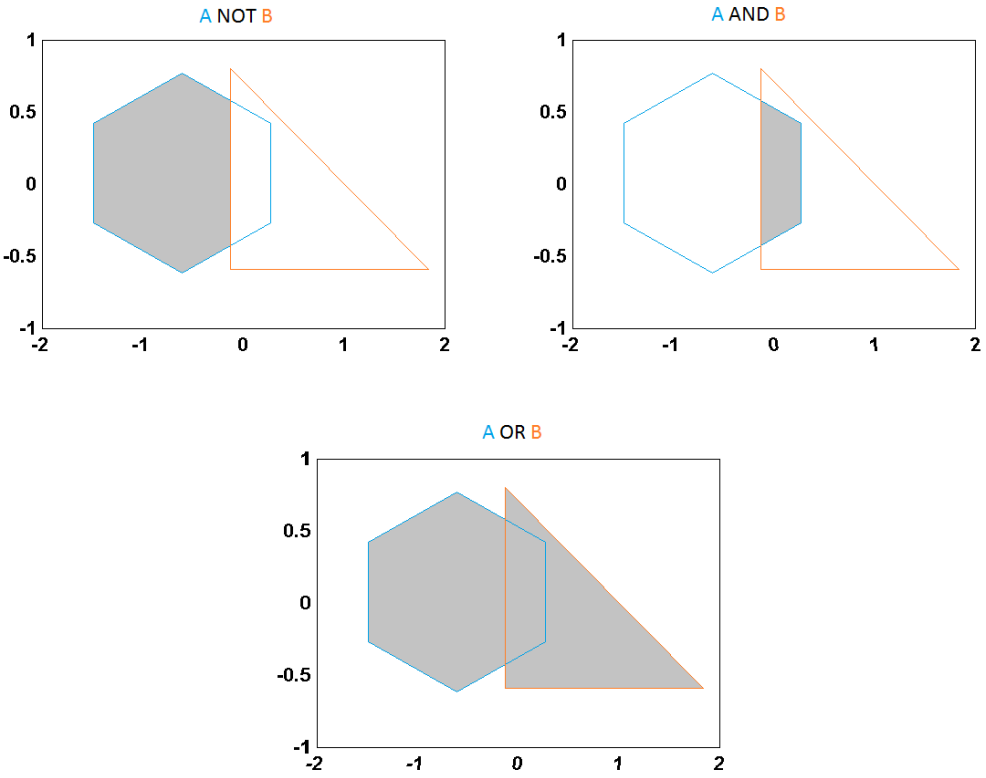
For example, the below hexagon can be described by picking a point and then listing the corners that follow if traveling in a clockwise fashion from corner to corner: (-b,c),(b,c),(a,0),(b,-c),(-b,-c),(-a,0)



This can be represented in Dyalog as an enclosed array such:

```
hexagon←((-b) c) (b c) (a 0) (b (-c)) (-b c) (-a 0)
```

Having a way to describe polygons allows us to perform computations on them.



***Task 1** – Write a function* **PolyOR** *that performs an OR operation on two polygons.*

PolyOR has the following syntax:

```
r ← poly1 PolyOR poly2
```

The left and right arguments `poly1` and `poly2` are each a vector of points, ordered clockwise, that define one polygon each.
The result `r` is a vector of clockwise ordered points of the new polygon created by performing an OR operation on the polygons represented by `poly1` and `poly2`.

Example:

```
poly1 ← (0 0)(0 2)(2 2)(2 0)

poly2 ← (1 1)(1 3)(3 3)(3 1)

poly1 PolyOR poly2
```

0	0	0	2	1	2	1	3	3	3	3	1	2	1	2	0

***Task 2** – Write a function* **PolyAND** *that performs an AND operation on two polygons.*

PolyAND has the following syntax:

```
r ← poly1 PolyAND poly2
```

The left and right arguments `poly1` and `poly2` are each a vector of points, ordered clockwise, that define one polygon each.
The result `r` is a vector of clockwise ordered points of the new polygon created by performing an AND operation on the polygons represented by `poly1` and `poly2`.

Example:

```
poly1 ← (0 0)(0 2)(2 2)(2 0)

poly2 ← (1 1)(1 3)(3 3)(3 1)

poly1 PolyAND poly2
```

1	1	1	2	2	2	2	1

***Task 3** – Write a function* **PolyNOT** *that performs a difference (NOT) operation on two polygons.*

PolyNOT has the following syntax:

```
r ← poly1 PolyNOT poly2
```

The left and right arguments `poly1` and `poly2` are each a vector of points, ordered clockwise, that define one polygon each.
The result is `r` is a vector of clockwise ordered points of the new polygon created by removing the part of the polygon defined by `poly1` which is covered by the polygon defined by `poly2`.
Note that, as opposed to PolyOR and PolyAND, this function is **not** commutative.

Example:

```
poly1 ← (0 0)(0 2)(2 2)(2 0)

poly2 ← (1 1)(1 3)(3 3)(3 1)

poly1 PolyNOT poly2
```

0	0	0	2	1	2	1	1	2	1	2	0