

Overview

The Phase II problems are divided into three sets, one for each of the three categories (Cryptography, Physics, and Neural Networks). Each set comprises 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 also includes a submission for a second category, the judging committee may take that into consideration in a tie-break situation. In short, your ranking in the competition can never be hurt by completing more than one problem set.

Judging Guidelines:

- Solutions which appropriately use an array-oriented approach will be judged higher. Translation: Not all problems benefit from array-oriented thinking, but those that do should be solved using an array-oriented approach.
- Comments should be used to document the approach you take and any complex statements. We look for clarity of your understanding of the problem and its solution. But it is not necessary to comment every statement nor write a novella-sized description.

Thank you for participating!

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 Contest2018 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 `Contest2018.DWS`

The Contest2018 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 (tradfn, dfn, or tacit function) is acceptable.

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

The namespaces are:

- `#.crypto` – for the cryptography problem set
 - `#.neural` – for the neural networks problem set
 - `#.physics` – for the physics 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 `Contest2018.dyalog` which will contain any code or data you placed in the `#.crypto`, `#.neural`, and `#.physics` namespaces.

If you use some other APL system, you can use the template script file `Contest2018.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.

Submitting your entry

To submit your entry, upload the completed `Contest2018.dyalog` file using the competition website. Be sure to examine the file to verify your work is included in it. You should submit one file only even if you complete more than one problem category – each category's solutions should be in their respective namespaces.

Cryptography Problem Set

The Cryptography Problem Set comprises 3 problems consisting of 4, 3, and 3 tasks respectively. To be considered for the grand prize in this category, you must complete all 10 tasks.

Problem 1 – Crypto Basics

Cryptography is deeply rooted in number theory. Factors, primes and understanding of combinatorics are often used in making sure messages can be stored and sent securely.

Task 1 - Greatest Common Divisor

Write a function **GCD** that calculates the greatest common divisor of its arguments without using the **GCD** primitive function **v**.

GCD has the following syntax:

```
r ← a GCD b
```

a and **b** are arrays of positive integers of compatible shapes

r is the resultant greatest common divisor

Examples:

```
725 GCD 150
25
```

```
725 GCD 58 131 290
29 1 145
```

```
314 15 926 GCD 471 100 463
157 5 463
```

The following function train can be used to test that your function performs the same as the APL primitive version. It does **v** and **GCD** between all combinations of the left and right arguments and compares the results.

```
test←∘.v≡∘.GCD
test~ι300 ⌞ test all combinations of 1..300
1
test~500?~1+2*31 ⌞ test 500 random numbers from 1..2147483647
1
```

Task 2 – Exclusive Or - XOR

Write an XOR function without using ANY of the following APL comparison functions:

< ≤ = ≥ > ≠ ≡ ≠

Exclusive OR is a Boolean operation with the following truth table:

a	b	r
0	0	0
0	1	1
1	0	1
1	1	0

The APL primitive function ≠ achieves this result for Boolean arguments.

XOR has the following syntax:

r ← a XOR b

a and **b** are Boolean arrays of compatible shapes

r is the resultant Boolean array

Examples:

```
1 XOR 1
```

```
0
```

```
1 1 0 0 XOR 1 0 1 0
```

```
0 1 1 0
```

```
∘.XOR ∼ 0 1 A truth table selfie
```

```
0 1
```

```
1 0
```

The following function train can be used to test that your function performs the same as the APL primitive version. It performs ≠ and XOR between all combinations of the left and right arguments and compares the results.

```
test←∘.≠≡∘.XOR
```

```
test∼0 1 A test all combinations
```

```
1
```

Task 3 – Next Prime Number

Write a function `NextPrime` that generates the next prime number after the given input

`NextPrime` has the following syntax:

`r ← NextPrime n`

- `n` is the lower limit in the search range for the next prime
For this problem `n` has a maximum value of 2147483646 (2^{31})
- `r` is the first prime number greater than `n`

Examples:

```
NextPrime 4
5
NextPrime 16
17
NextPrime 17
19
NextPrime 435
439
```

Task 4 – Catalan Numbers

Write a function `Catalan` that will output all the Catalan Numbers below the given input.

There has been some interest in using Catalan numbers for cryptographic applications. The Catalan numbers C_n are a sequence of integers 1, 1, 2, 5, 14, 42, 132. . . that have a variety of applications including in combinatorics, quantum mechanics, and the theory of disordered systems. For instance, C_n gives the number of expressions containing n pairs of parentheses that are correctly matched; there are 5 expressions where 3 pairs of parentheses are correctly matched.

((())) ()(()) ()() ()() ()()

The n^{th} Catalan number is given by

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

The sequence can be described as

$$C_0 = 1 \text{ and } C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \text{ for } n \geq 0$$

`Catalan` has the following syntax:

`r ← Catalan n`

`r` is a vector of Catalan Numbers in increasing order

`n` is the upper limit of the list

Extra credit will be awarded for recursive solutions.

Extra extra credit will be awarded for tail-recursive solutions.

Examples:

```
Catalan 20
1 1 2 5 14
Catalan 150
1 1 2 5 14 42 132
Catalan 10000
1 1 2 5 14 42 132 429 1430 4862
```

Problem 2 - XOR encryption

Background:

XOR encryption is a modern encryption method that encrypts given message by applying a bitwise exclusive OR operation on each character using a given key.

UTF-8 is a variable-width character encoding capable of encoding all valid code points in Unicode using one to four 8-bit bytes. The name is derived from Unicode Transformation Format – 8-bit. UTF-8 was designed for backward compatibility with ASCII. Code points with lower numerical values, which tend to occur more frequently, are encoded using fewer bytes. The first 128 characters of Unicode correspond one-to-one with ASCII, so that valid ASCII text is also valid UTF-8-encoded Unicode.

Every character is encoded in memory as a unique number and that number is stored in bits. This means you can perform an XOR operation on a character by doing the operation on its bit representation.

The process of encryption is to create the bit representation of the message to be encrypted and perform a XOR operation against the bit representation of the key. For the most secure encryption, the bit representation of the key is the same length as the bit representation of the plain text message, and the key is made up of random bytes. The user would keep the encrypted message and the encryption key in different locations; without both keys it is extremely difficult to decrypt the message. Unfortunately, this method is impractical for most users, so the modified method is to use a password as a key. If the bit representation of the password is shorter than the bit representation of the message, which is likely, then the bit representation of the key is repeated cyclically until it matches the length of the bit representation of the message. The balance for this method is using a password key that is sufficiently long for security but short enough to be memorable.

Task 1 - Change the Base

Write a function `UTF8Bits` that will convert a given string into its bit representation

`UTF8Bits` has the following syntax:

```
r ← UTF8Bits str
```

`str` is a character or string of characters

`r` is a Boolean vector containing the bit representation of the UTF-8 encoding of `str`.

You can convert any character to its UTF-8 integer representation using the Dyalog system function

`⊞UCS`. The first 128 Unicode code points correspond to the ASCII character set.

```
'UTF-8' ⊞UCS 'Testing 1 2 3'  
84 101 115 116 105 110 103 32 49 32 50 32 51
```

Examples:

```
UTF8Bits 'A'  
0 1 0 0 0 0 0 1
```

```
UTF8Bits 'Dyalog APL'  
0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 1 0 1 1 0 0 0 0 1 0 1 1 0 1 1 0 0 0 1 1 0 1 1  
1 1 0 1 1 0 0 1 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 0  
1 1 0 0
```

Task 2 - Encode text

Write a function `XOREncrypt` that will take a character input and apply XOR encryption

`XOREncrypt` has the following syntax:

```
r ← key XOREncrypt text
```

text is the text to be encrypted

key is the passphrase used for the encrypting

r is the resultant encrypted text

Using XOR and UTF8Bits, write a function that will take a text input as a right argument and a passphrase of any length as the left argument and return encrypted text using an XOR encryption.

To simplify things, you may assume that key and text will always be comprised of ASCII characters (the first 128 Unicode code points).

Examples:

```
⊞UCS 'APL' XOREncrypt 'Dyalog'  
5 41 45 45 63 43
```

```
'APL' XOREncrypt 'APL' XOREncrypt 'Dyalog'  
Dyalog
```

Task 3 - Crack the code

Write a function `CrackIt` that will decrypt a secret message.



Your job is to decrypt the message stored in `cipher_text.txt` which is included in the zip file that you should have downloaded for Phase 2. The catch is you don't have the pass phrase.

What you know:

- The ciphertext was encrypted using XOR encryption
- The pass key has three letters all lower case
- The plain text was ASCII only
- The original text made use of common English words

`CrackIt` has the following syntax:

```
r ← CrackIt cipher_text
```

`cipher_text` is the encrypted version of the message

`r` is the decrypted version of the message

The encrypted message is in the file, `cipher.txt`, in the Problem Solving Competition zip file.

If you're using Dyalog APL you can read the encrypted text into the workspace using:

```
cipher_text←{(⊖NUNTIE←{⊖NREAD ω 80 ⁂1})ω ⊖NTIE 0}'[your_file_location]/cipher.txt'
```

If you're using a different APL system, please refer to its documentation for how to read the contents of a text file into the workspace

Problem 3 – Big Integers

42

**A big integer,
but not the kind we care about**

```
2987948729873987948793987498789749828761
8718740398729187627628763876826876287643
8638764862876387628374628376283682687642
5625321757645752765376535748769379569810
1001984398497476187638765190309820499509
8509808082021203
```

We care about integers with lots of digits

RSA is one of the first public-key cryptosystems used by modern computers to encrypt and decrypt messages. It is an asymmetric cryptographic algorithm. Symmetric cryptography uses a single key to both encode and decode the message whereas asymmetric cryptography uses two different keys where either key can be used to encode the message and the other key is used to decode. This is also called public key cryptography, because one of them can be given to everyone (the other key must be kept private). The security of RSA encryption relies on the fact that finding the factors of a large integer is difficult.

A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept secret. Anyone can use the public key to encrypt a message and if the public key is large enough, only someone with knowledge of the prime factors can feasibly decode the message.

Rather than ask you to implement RSA cryptography, we present 3 tasks to perform operations on big integers.

For these tasks, you will use character arrays of digits to represent the integers. For example:

```
'8'
'42'
'9820982095098371002099820984763762767476576659398204'
```

Negative numbers have a leading '-'

```
'-42'
'-39989488959820809845087279759875987398739874579657985'
```

Leading 0s are ignored

```
'00000042' is treated the same as '42'
'-000000000000042' is treated the same as '-42'
```


Physics Problem Set

The Physics Problem Set comprises 3 problems consisting of 3, 2, and 2 tasks respectively. To be considered for the grand prize in this category you must complete all 7 tasks.

Problem 1 – Celestial Bodies



The orbit in space of one body around another, such as a planet around the Sun, is rarely circular. In general, it takes the form of an ellipse, with the body sometimes closer in and sometimes further out. It is also important to note that the body being orbited does not necessarily have to be at the centre of the orbit.

The closest and furthest distances to the centre of an ellipse have special significance in geometry, the same can be said about the closest and furthest distances from the centre of orbit in astronomy. The closest distance and furthest distances to the centre of an orbit are referred to as the semi-major axis and semi-minor axis respectively. The other important set of distances with orbiting bodies are the points at which the orbital body is closest and furthest from the body being orbited. The closest distance an orbital body makes to the body it is orbiting (for example, the Sun) is referred to as the **Perihelion** distance and the furthest distance is the **Aphelion** distance.

A vast amount of information about the movement of celestial bodies can be derived from simply knowing the Perihelion distance, the velocity at the point of the Perihelion and the mass of the body being orbited.

Note: The inputs for the examples for the tasks use information from 4 NASA planetary fact sheets:

- <https://nssdc.gsfc.nasa.gov/planetary/factsheet/mercuryfact.html>
- <https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html>
- <https://nssdc.gsfc.nasa.gov/planetary/factsheet/marsfact.html>
- <https://nssdc.gsfc.nasa.gov/planetary/factsheet/saturnfact.html>

The results for the examples may vary slightly from some of the figures in the fact sheets. This is due to differences in precision of the calculations and the gravitational and mass of the Sun constants.

Task 1 – Aphelion Velocity

Write a function `AphelionVelocity` that calculates the Aphelion velocity.

Let the Perihelion distance and velocity be l_1 and v_1 respectively.

When solved, the following quadratic equation returns the Perihelion and Aphelion velocities. The Aphelion velocity is the smaller of the two roots of the equation.

$$v_2^2 - \frac{2GM}{l_1 v_1} v_2 - \left[v_1^2 - \frac{2GM}{l_1} \right] = 0$$

G is Newton's gravitational constant $G = 6.67408 \times 10^{-11} \text{ m}^3 \text{ kg}^{-1} \text{ s}^{-2}$

M is the Mass of the body being orbited, in our case the Sun thus $M = 1.98855 \times 10^{30} \text{ kg}$

So, the problem becomes one of solving using the quadratic formula.

For equations in the form $ax^2 + bx + c = 0$, the roots of the equation are

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

`AphelionVelocity` has the following syntax:

`v2 ← v1 AphelionVelocity l1`

`v1` is the velocity at the Perihelion in meters/second

`l1` is the Perihelion distance in meters

`v2` is the velocity at the Aphelion rounded to the nearest meter/second

Examples:

```
58.98E3 AphelionVelocity 46.00E9      A Mercury
38855
```

```
30.29E3 AphelionVelocity 147.09E9     A Earth
29287
```

```
26.50E3 AphelionVelocity 206.62E9     A Mars
21977
```

```
10.18E3 AphelionVelocity 1352.55E9    A Saturn
9098
```

Task 2 – Major-Minor

Write a function `Axis` that given the Perihelion distance and velocity will return the semi-major and semi-minor axes of a celestial body orbiting the sun.

Let the Perihelion and Aphelion distances be l_1 and l_2 respectively and let the velocities at the Perihelion and Aphelion be v_1 and v_2 respectively.

Kepler's second law gives us a relationship between the Perihelion and Aphelion distances and velocities:

$$l_1 v_1 = l_2 v_2$$

Thus, given l_1 , v_1 and v_2 it's a simple matter to find the remaining value l_2 (the Aphelion distance). The equations for the semi-major axis and semi-minor axis are described below, where a and b are the semi-major axis and semi-minor axis respectively.

$$a = \frac{1}{2}(l_1 + l_2)$$
$$b = \sqrt{l_1 l_2}$$

`(a b) ← v1 Axis l1`

- `l1` is the Perihelion distance in meters
- `v1` is the velocity at the Perihelion in meters/second
- `a` is the semi-major axis in meters
- `b` is the semi-minor axis in meters

Examples: (note your results may not match exactly)

```
58.98E3 Axis 46.00E9      A Mercury
5.791266467E10  5.66742018E10
```

```
30.29E3 Axis 147.09E9    A Earth
1.496098321E11  1.495886103E11
```

```
26.50E3 Axis 206.62E9    A Mars
2.278793245E11  2.268854946E11
```

```
10.18E3 Axis 1352.55E9   A Saturn
1.432996581E12  1.430736716E12
```

Task 3 – What's in a Year?

Write a function `OrbitalPeriod` that given the Perihelion distance and velocity will return the orbital period of a celestial body orbiting the sun.

Using the functions developed in the previous tasks we can now tackle calculating the orbital periods for celestial bodies in the solar system.

The equation for the orbital period is:

$$T = \frac{2\pi ab}{l_1 v_1}$$

`OrbitalPeriod` has the following syntax:

```
T ← v1 OrbitalPeriod l1
```

- `l1` is the Perihelion distance in meters
- `v1` is the velocity at the Perihelion point in meters/second
- `T` is the orbital period in days

Examples:

```
58.98E3 OrbitalPeriod 46.00E9      A Mercury  
87.97565041
```

```
30.29E3 OrbitalPeriod 147.09E9     A Earth  
365.2938559
```

```
26.50E3 OrbitalPeriod 206.62E9     A Mars  
686.6881442
```

```
10.18E3 OrbitalPeriod 1352.55E9    A Saturn  
10828.53976
```

Problem 2 - Symmetry

This problem was inspired by a game in the app Logic Games for Android and iOS. The game in question is called Galaxies and involves partitioning a grid into symmetrical sections centred around specific points on the grid. While solving the game itself would make an interesting application, it's probably more difficult than we want to tackle here. Instead, the two tasks for this problem will deal with detecting symmetry and building symmetric shapes.

For this problem, we will be using character matrices to represent the grid. Each "cell" will have one of

- indicating an empty (available) cell
- indicating a cell in the symmetrical shape
- * indicating an unavailable cell

A valid shape must consist of contiguous horizontal and/or vertical cells; diagonally adjacent cells are not considered to be contiguous. A shape is symmetric if it matches itself when rotated 180 degrees.

For example:

Symmetric		Not Symmetric	Symmetric but invalid
□□□--	--□□--	□□□-*	□*□--
--□□--	*□□--	--□□--	--□□--
--□*-	--□□*	--□--	--*□*-
--□□-	--*□□-	--*□□□	--□□-
*-□□□	-----	--□□□	--□-□

Task 1 – Is It Contiguous?

Write a function `Contiguous` that given a character matrix representing a grid of available, occupied, and unavailable cells will return a Boolean indicating whether all the occupied cells are contiguous (adjacent either horizontally and/or vertically).

`Contiguous` has the following syntax:

`r ← Contiguous mat`

`mat` is a character matrix as described in the problem introduction

`r` is a Boolean indicating whether all the occupied cells are contiguous

Examples:

`Contiguous 1 1p'█'` A a single occupied cell is contiguous

1

`Contiguous 2 2p'-'` A Zen question, is a non-shape contiguous?

1

```

█←mat← 5 5p'███*--██----*█----*██----███'
███*-
-██--
-*█--
-*██-
--███

```

`Contiguous mat`

1

`█←mat←3 3p'█--'`

```

█--
█--
█--

```

`Contiguous mat`

1

`█←mat←3 3p'█---'`

```

█--
-█-
--█

```

`Contiguous mat`

0

`Contiguous 2 2p'██-█'` A contiguous, but not symmetric

1

Task 2 – Is It Symmetric?

Write a function `Symmetric` that given a character matrix representing a grid of available, occupied, and unavailable cells will return a Boolean indicating if the all the occupied cells represent a symmetric shape.

`Symmetric` has the following syntax:

```
r ← Symmetric mat
```

`mat` is a character matrix as described in the problem introduction

`r` is a Boolean indicating if the all the occupied cells represent a valid symmetric shape

Examples:

```
Symmetric 1 1p'█' A a single occupied cell is symmetric
1

Symmetric 2 2p'-' A Zen question, is a non-shape symmetric?
1

█←mat←5 5p'███*--██----*█----*██----████'
███*-
-██--
-*█--
-*██-
--████
Symmetric mat
1

█←mat←3 3p'█--'
█--
█--
█--
Symmetric mat
1

█←mat←3 3p'█---'
█--
-█-
--█
Symmetric mat A symmetric, not contiguous
1
```

Task 3 – Is It Valid?

Using `Contiguous` and `Symmetric`, write a function `Valid` that given a character matrix representing a grid of available, occupied, and unavailable cells will return a Boolean indicating if the all the occupied cells represent a contiguous, symmetric shape.

`Valid` has the following syntax:

`r ← Valid mat`

`mat` is a character matrix as described in the problem introduction

`r` is a Boolean indicating if the all the occupied cells represent a valid symmetric shape

Examples:

`Valid 1 1p'█'` A a single occupied cell is symmetric
1

`Valid 2 2p'-'` A Zen question, is a non-shape symmetric?
1

`█←mat← 5 5p'███*--██---*█---*██---███'`
███*--
-██--
-*█--
-*██-
--███
`Valid mat`
1

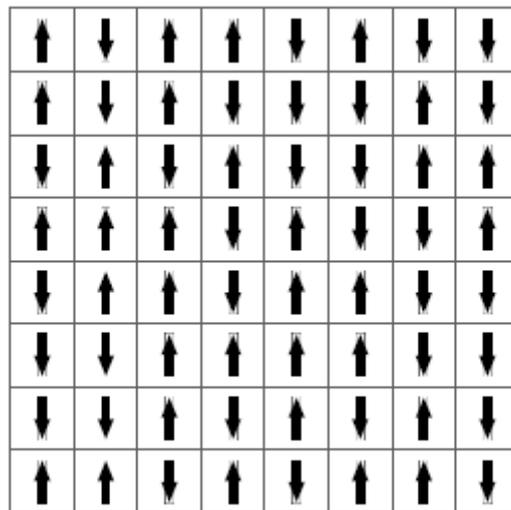
`█←mat←3 3p'█--'`
█--
█--
█--
`Valid mat`
1

`█←mat←3 3p'█---'`
█--
-█-
--█
`Valid mat A symmetric, not contiguous`
0

`Valid 2 2p'██-█'` A contiguous, but not symmetric
0

Problem 3 – Magnets

The Ising model is a theoretical model of a magnet. The magnetisation of a magnetic material is made up of the combination of many small magnetic dipoles spread throughout the material. If these dipoles point in random directions then the overall magnetisation of the system will be close to zero, but if they line up so that all or most of them point in the same direction then the system can acquire a macroscopic magnetic moment - it becomes magnetised. The Ising model is a model of this process in which the individual moments are represented by dipoles or “spins” arranged on a grid or lattice:



In this case we are using a square lattice in two dimensions, although in principle the model can be defined for any lattice in any number of dimensions. The spins themselves, in this simple model, are restricted to point in only two directions, up and down. Mathematically the spins are represented by variables $s_i = \pm 1$ on the points of the lattice, 1 for up-pointing spins and -1 for down-pointing spins. Dipoles in real magnets can typically point in any spatial direction, not just up or down, but the Ising model, with its restriction to just the two directions, captures a lot of the important physics while being significantly simpler to understand.

The magnetic potential energy due to the interaction of two dipoles is proportional to their dot product, but in the Ising model this simplifies to just the product $s_i s_j$ for spins at position (i, j) on the lattice, since the spins are one-dimensional scalars, not vectors. Then the actual energy of interaction is $-J \times s_i s_j$, where J is a positive interaction constant. The minus sign ensures that the interactions are ferromagnetic, meaning the energy is lower when dipoles are lined up. A ferromagnetic interaction implies that the material will magnetise if given the chance. (In some materials the interaction has the opposite sign so that the dipoles prefer to be anti-aligned. Such a material is said to be antiferromagnetic, but we will not look at the antiferromagnetic case here.)

Task 1 – Ising Model

Write a function `TotalEnergy` that calculates the total energy of an $n \times m$ lattice of dipoles.



Normally it is assumed that spins interact only with those that are immediately adjacent to them on the lattice, which gives a total energy for the entire system equal to

$$E = -J \sum_{\langle ij \rangle} s_i s_j,$$

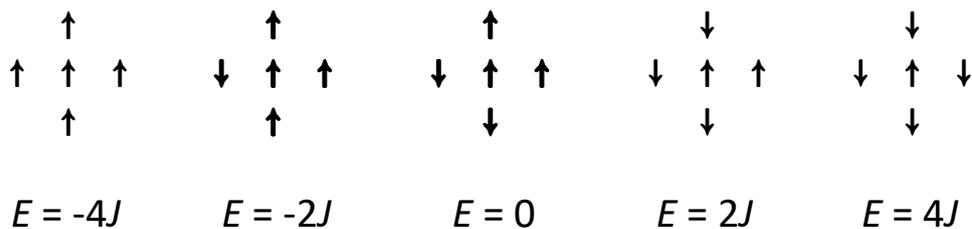
where the notation $\langle ij \rangle$ indicates a sum over pairs i, j that are adjacent on the lattice. On this square lattice each spin has four adjacent neighbors with which it interacts – left, right, up and down.

To clarify, consider a spin s with adjacent neighbors N, E, S, and W



The energy at s is $E = -J \times s \times (N + E + S + W)$

To illustrate:



The total energy for the system is the sum over the entire lattice, but each unique pair of adjacent spins contributes only once in the sum. Thus, there is a term s_1s_2 if spins 1 and 2 are adjacent to one another, but you do not also need a term s_2s_1 .

For simplicity assume J is 1.

Hint: one technique could be to use Dyalog's Stencil operator: 

TotalEnergy has the following syntax:

E ← TotalEnergy lattice

E is the total magnetic energy

lattice is a matrix of 1s and -1s representing the different spins

Examples:

TotalEnergy 5 5 ρ ⁻¹ 1
40

TotalEnergy 5 5 ρ 1
-40

TotalEnergy 5 5 ρ ⁻¹
-40

TotalEnergy 5 5 ρ ⁻¹ 1 ⁻¹ 1 ⁻¹
0

TotalEnergy 20 30 ρ ⁻¹ 1 1 ⁻¹ 1
-230

TotalEnergy ⁻¹ 1 [?150 150ρ2] # your answer may be different
162

Task 2 – Metro Styling

Write a function `Simulate` that employs Metropolis-style Monte Carlo simulation of the Ising model.

Another important feature of many magnetic materials is that the individual dipoles in the material can interact magnetically in such a way that it is energetically favourable for them to line up in the same direction. This means that if a dipole was attempting to flip in a direction that increased the overall energy of the system, then the system would work to keep the change where as if the flip was to decrease the overall energy the system would work to reverse the change.

Your function will simulate changes to the system by applying the procedure:

1. Take an initial lattice of 300×300 in as an argument with spin variables set to ± 1 .
2. Then choose an interior (not on an edge) spin at random, flip it, and calculate the new energy after it is flipped.
3. Then decide whether to accept the flip using the following acceptance formula:
 - if the change in energy $\Delta E < 0$, accept the change
 - otherwise accept the change with the probability $p = e^{\left(\frac{-\Delta E}{T k_B}\right)}$
T is the temperature and defaults to 1
kB is the Boltzmann constant and is also 1
4. Run the procedure n times, where n is an input.
5. Return the updated map

`Simulate` has the following syntax:

```
r ← n Simulate lat
```

n is the number of iterations

lat is the starting 300×300 matrix of **1**s and **-1**s

r is the resultant matrix (300×300 matrix of **1**s and **-1**s) after **n** iterations.

Neural Network Problem Set

The Neural Network Problem Set comprises 3 problems of increasing difficulty. Problem 1 has 2 tasks, Problem 2 has 2 tasks and Problem 3 has 2 tasks. To be considered for the grand prize in this category, you must complete all 7 tasks.

Problem 1 - Transfer Functions

Transfer functions are used in a range of domains. They define a relationship between the input and the output of a system and are often used to map an infinite range of inputs to a finite range of outputs. They are a fundamental part of neural networks.

Task 1 - Step Function

Write a function `StepFn` that returns 1 if the input is greater than or equal to 0 and -1 otherwise.

`StepFn` has the following syntax:

```
r ← StepFn in
```

Examples:

```
StepFn [-5+10]
[-1 -1 -1 -1 1 1 1 1 1 1]
```

Task 2 - Sigmoid

Write a **Sigmoid** function.

The sigmoid function maps any value to a value between 0 and 1.

$$S(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}}$$

Note the mathematical constant e (approximately 2.71828) can be found in Dyalog using the monadic exponential function `*`.

`*1` returns 2.718281828

Sigmoid has the following syntax:

```
r ← Sigmoid in
```

Examples:

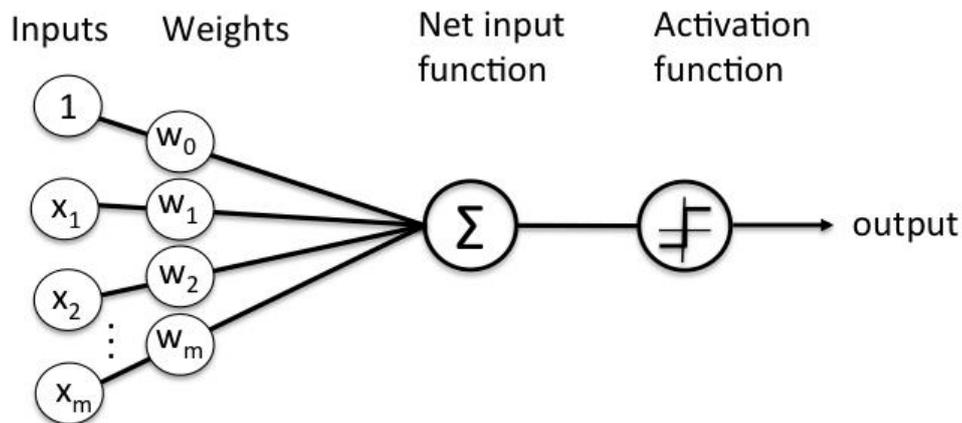
```
Sigmoid ^-2+ι5  
0.2689414214 0.5 0.7310585786 0.880797078 0.9525741268
```

Problem 2 – Perceptron

Task 1 - Perceptron

Write an operator `Perceptron` that, given a set of weights, inputs and an activation function, returns an output as defined by the rule of a perceptron.

A perceptron is the simplest neural network possible: a computational model of a single neuron. A perceptron consists of one or more inputs, a processor, and a single output.



A perceptron follows the “feed-forward” model, meaning inputs are sent into the neuron, processed, and result in an output. In the diagram above, this means the network (one neuron) reads from left to right: inputs come in, output goes out.

Mathematically, we can represent the perceptron as:

$$o = f\left(\sum_{k=1}^n i_k \cdot W_k\right)$$

where:

- o** is the output of the perceptron
- i** are the inputs
- W** are the weights
- f** is the activation function

Perceptron has the following syntax:

```
r ← input (fn Perceptron) weights
```

input is a vector of input values
weights is a vector of weights
fn is an activation function
r is the perceptron's output

Examples:

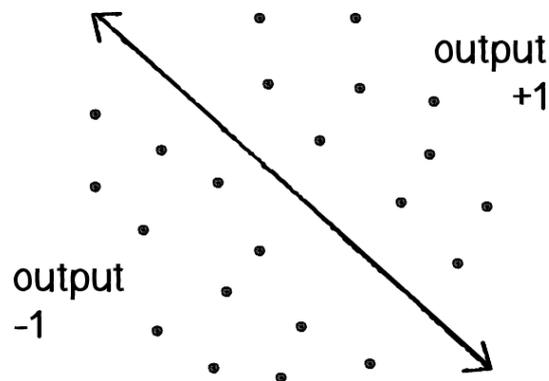
```
2 3 (StepFn Perceptron) 0.2 0.4  
1
```

```
13 -4 (Sigmoid Perceptron) -3 1.2  
9.503896381E-20
```

Task 2 - Are you in or are you out?

Write an operator S i d e that will take a function representing a straight line and will output 1 or -1 based upon which side of the line it is on.

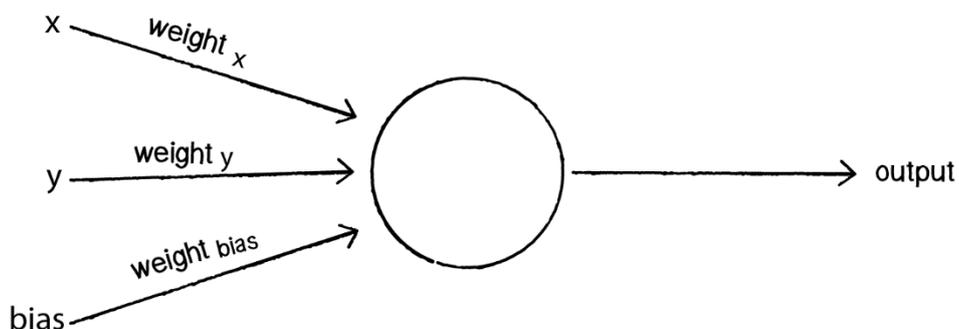
Neural networks are often used for pattern recognition applications, such as facial recognition and language translations. Even simple perceptron's can demonstrate the basics of classification.



Consider a line in two-dimensional space. Points in that space can be classified as being on either one side of the line or the other. Given a perceptron that take two inputs and uses a **step function** we can classify the input as either above or below the line. In the above diagram, we can see how each point is either below the line (-1) or above (+1). Points lying directly on the line shall be treated as being above (+1).

However, the perceptron that simply takes in weights and inputs has its limitations. Consider the point (0,0). The sum of its weighted input will necessarily be 0. Given the point (0,0) can be above or below various lines in Cartesian space this problem needs to be dealt with. Introducing a bias input will circumvent this problem.

A *bias* input is an additional input that always has the value of 1 and is also weighted. Here is our perceptron with the addition of the bias:



Consider again the point (0,0). Here are our inputs:

$0 \times \text{weight for } x = 0$

$0 \times \text{weight for } y = 0$

$1 \times \text{weight for bias} = \text{weight for bias}$

The output is the sum of the above three values, 0 plus 0 plus the bias' weight. Therefore, the bias, on its own, answers the question as to where (0,0) is in relation to the line. If the bias' weight is positive, (0,0) is above the line; negative, it is below. It "biases" the perceptron's understanding of the line's position relative to (0,0).

A perceptron has no knowledge about its inputs or what the desired output is; so for your perceptron to return the correct result for a given problem it is necessary for it to have the correct weights. One of the methods of adjusting weights is *supervised learning*. This process requires that the desired outcome is known (or easily calculatable) and the correct result is used to incrementally adjust the weights until the perceptron shows a sufficient level of accuracy.

With this method, the network is provided with inputs for which there is a known answer. This way the network can find out if it has made a correct guess. If it's incorrect, the network can learn from its mistake and adjust its weights. The process is as follows:

1. Provide the perceptron with inputs for which there is a known answer.
2. Let the perceptron attempt to answer.
3. Compute the error.
4. Adjust all the weights according to the error.
5. Return to Step 1 and repeat.

ERROR = DESIRED OUTPUT - GUESS OUTPUT

In the case of using a perceptron to determine which side of a line a point lies on, the output has only two possible values: **+1** or **-1**. This means there are only three possible errors. If the perceptron guesses the correct answer, then the guess equals the desired output and the error is 0. If the correct answer is -1 and we've guessed +1, then the error is -2. If the correct answer is +1 and we've guessed -1, then the error is +2.

Desired	Guess	Error
-1	-1	0
-1	+1	-2
+1	-1	+2
+1	+1	0

The error is the determining factor in how the perceptron's weights should be adjusted. For any given weight, we want to calculate the change in weight (Δ weight).

$$\text{new weight} = \text{weight} + \Delta\text{weight}$$

where Δ weight is calculated as the error multiplied by the input.

$$\Delta \text{weight} = \text{error} \times \text{input}$$

Therefore:

$$\text{new weight} = \text{weight} + \text{error} \times \text{input}$$

Side has the following syntax:

```
weights ← fn Side start_weights
```

weights are weights that will return the correct result when used in **Perceptron** with **StepFn**

fn is the line-defining function

start_weights is a three element vector each representing a starting weight in the range [-1, 1]. Note the third is present to cater for the bias.

Examples:

```
fn ← {ω}           A y = x
fn Side ^1+2×?3p0
^-765.652021 767.4662006 2.766102422
```

```
fn ← {3+2×ω}      A y = 2x + 3
fn Side ^1+2×?3p0
^-6572.054317 3196.163596 ^-3836.827018
```

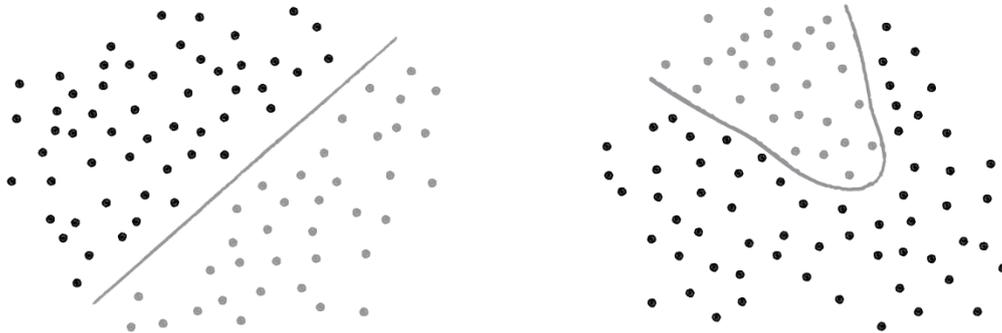
```
fn ← {-4+0.5×ω}   A y = 0.5x - 4
fn Side ^1+2×3p0
^-1489 3103 8947
```

```
fn ← {0}          A y = 0
fn Side ^1+2×?3p0
26.39678882 161.6949529 2.119097298
```

Note: due to the nature of the problem it is likely that your results will be different to those above. If the resultant weights can be used to perform the operation they were trained for with sufficient accuracy, your submission stands in good stead.

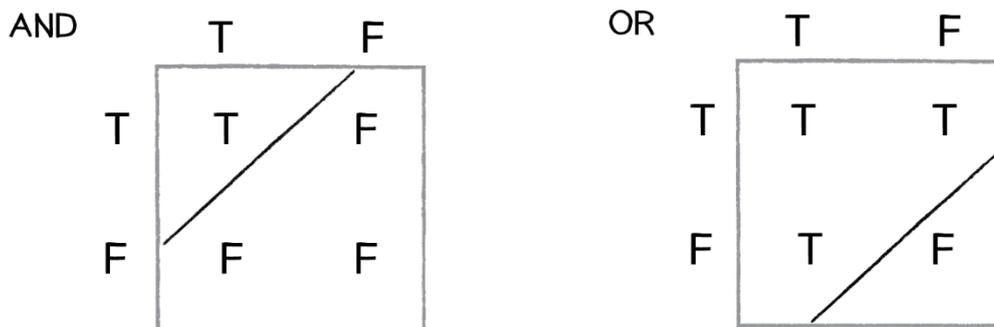
Problem 3 - XOR Net

Perceptrons are limited in that they can only solve **linearly separable** problems, that is if a data set can be separated with a straight line it is linearly separable.

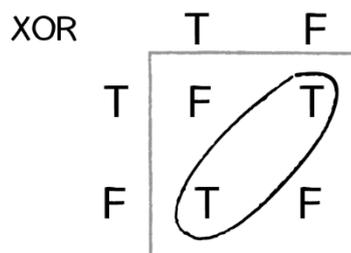


The figure on the left is linearly separable while the one on the right is not.

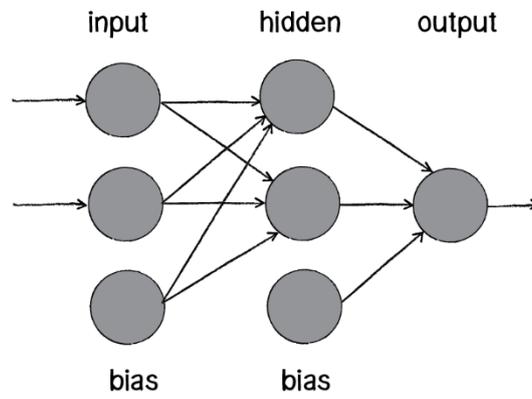
One of the simplest examples of a non-linearly separable problem is *XOR*, or “exclusive or.” Looking at the truth tables for AND and OR we can see that they are both linearly separable.



Whereas looking at the truth table for XOR it is clear that it is not linearly separable.



One perceptron is not enough to solve an XOR problem but by creating a network of perceptrons it becomes possible.



The above diagram is known as a *neural network*, a network of many neurons. Some are input neurons and receive the inputs, some are part of what's called a "hidden" layer (as they are connected to neither the inputs nor the outputs of the network directly), and then there are the output neurons, from which we read the results.

Task 1 - Feed Forward

Write a function `FeedForward` that returns the output of a multi layered network using a Sigmoid activation function.

Since we are not using a mechanism to predefine the number of layers or the number of nodes in each layer, the shape of the weights and inputs will determine the size of the network. This means that there can any number of nodes on the input, hidden layers or output layers.

Note: Every layer except the output layer should have a bias.

`FeedForward` has the following syntax:

```
r ← input FeedForward weights
```

weights is a matrix of weights one for each connection across each layer

input is a vector of inputs

r the resultant value of the inputs passing through the network

Examples:

```
R a a network comprised of a lone perceptron with two inputs
```

```
1 2 FeedForward (3 1p0.5 0.2 ^0.1)
```

```
0.6899744811
```

```
1 2 1 (Sigmoid Perceptron) 0.5 0.2 ^0.1
```

```
0.6899744811
```

```
R a a 3-layer network with 2 input nodes, 3 nodes in the hidden layer  
R and 1 output node
```

```
^4 2 FeedForward (3 3p0.5 0.2 ^0.1) (4 1p0.1 ^0.1)
```

```
0.4863122671
```

```
R A 4-layer network: 3 input nodes, 2 hidden layers with 7 and  
R 3 nodes respectively and 2 output nodes
```

```
3 5 ^7 FeedForward (4 7p1.2) (8 3p^0.1) (4 2p3.2)
```

```
0.9981617589 0.9981617589
```

Task 2 - Back Propagation

Write a function `BackProp` that trains a neural network to solve an XOR computation. Your neural network should have two input nodes, three nodes in the hidden layer and one output node excluding the biases.



Training a neural network with multiple layers is where the real work comes in. It is much more complicated than training a lone perceptron. With the single perceptron, it was easy to calculate how to change the weights according to the error. With a neural network there are many different connections, each in a different layer of the network. How does one know how much each neuron or connection contributed to the overall error of the network?

One of the common approaches for training deep neural networks is a weight optimising algorithm known as **backpropagation**. Below is a guide to implementing a backpropagation algorithm to train a network to perform an XOR operation.

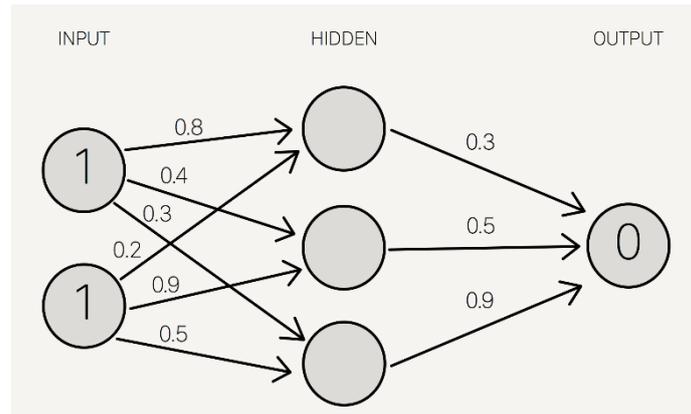
First up we need a measure of how well the network is performing. To do this we introduce an error term:

$$error = target - output$$

The aim is to now use this error and propagate it backwards through the network adjusting the weights to get closer to the target output. This can be achieved using a process called **gradient descent**. The basic idea is to look at each weight and find the impact a small change has on the total error and then to use that to adjust the weights so that each weight is moved proportionally to its overall impact. This means that if one or two weights are close to their optimal value they will not be heavily affected, but the weights that are far from their optimal value will experience much larger changes.

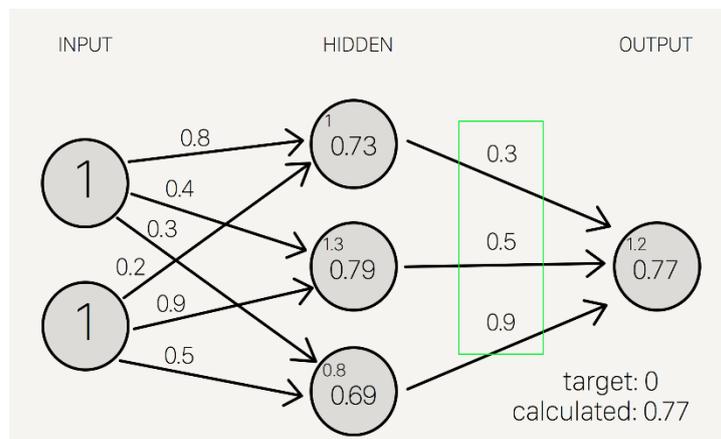
In the equations that follow, we use matrix notation to help illustrate the process. However, we have not applied this rigorously – for instance, scalar values could possibly be better represented in your solution as 1×1 matrices or vectors could be 1-row or 1-column matrices. There may be instances where the matrix shown is the transposition of the actual data, but as there are no duplicate shapes in this network, it should be obvious when this is the case and you should adjust accordingly in your solution.

Consider the following three-layer network:



Running a feed forward algorithm, making use of a Sigmoid activation function (which you wrote in Problem 1, Task 2) on the sum of the inputs to the hidden layer, we can calculate the following values for the hidden layer.

$$\begin{aligned}
 layer_{hidden} &= S([inputs] \cdot [weights_{input \leftrightarrow hidden}]) \\
 layer_{hidden} &= S\left([1 \ 1] \cdot \begin{bmatrix} 0.8 & 0.4 & 0.3 \\ 0.2 & 0.9 & 0.5 \end{bmatrix}\right) \\
 layer_{hidden} &= S([1.0 \ 1.3 \ 0.8]) \\
 layer_{hidden} &= [0.7310585786 \ 0.785834983 \ 0.6899744811]
 \end{aligned}$$



The output sum is the sum of the product of the hidden layer values and the hidden output weights.

$$\begin{aligned}
 output &= [layer_{hidden}] \cdot [weights_{hidden \leftrightarrow output}] \\
 output &= [0.7310585786 \ 0.785834983 \ 0.6899744811] \cdot \begin{bmatrix} 0.3 \\ 0.5 \\ 0.9 \end{bmatrix} \\
 output &= [1.233212098]
 \end{aligned}$$

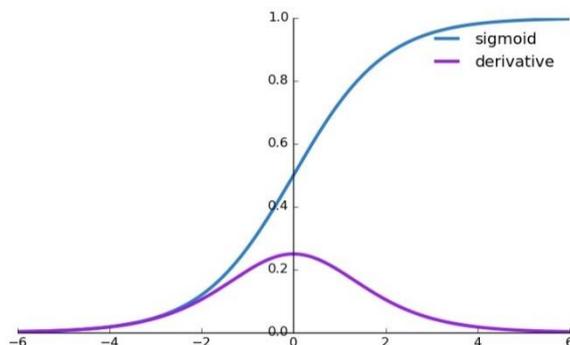
Then applying the activation function gives the final output result:

$$\begin{aligned}
 layer_{output} &= S(output) \\
 layer_{output} &= S([1.233212098]) \\
 layer_{output} &= [0.774380272]
 \end{aligned}$$

Given our definition of error, we have $error = -0.774380272$

This error must now be propagated back through our network in an attempt to find the optimal amount by which to adjust the weights.

First, we want to know if a weight is too big or too small. Once that is established we want to know how big an adjustment is safe to make. If big adjustments are made, there is the risk of overshooting the targeted value, but if too small a step is taken then it could take a long time to get to the desired result. To answer these two questions, let us look at the derivative of our activation function.



Remember the Sigmoid function:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Its derivative is as follows.

$$S'(x) = S(x)(1 - S(x))$$

The derivative describes the slope of the curve. The slope is biggest in the middle and is a maximum at the point $S(x) = 0.5$. Note that this is also the point that is furthest away from either of our desired results, namely 1 or 0. If we move right we get to 1 if we move left we get to 0. The derivative tails off to 0 to both the left and the right. It looks like the derivative will be useful with for determining whether it is safe to take big steps – when at the furthest point from either possible answer it is at a maximum meaning big steps and it gets smaller as it moves towards one of the answers meaning smaller steps. But we still need to know which direction to move in.

For this we use the error term. Since it is signed it can indicate which way is the right direction to move in. Putting these ideas together we get a metric called the *output delta* which is the derivative of the output sum multiplied by the error.

$$\Delta_{output} = S'(output) \times error$$

$$\Delta_{output} = S'(1.233212098) \times -0.774380272$$

$$\Delta_{output} = 0.1747154663 \times -0.774380272 = -0.1352962103$$

Now we can work our way back through the network, adjusting the weights, until we reach the input layer. The contribution to the error from the hidden layer can be established by multiplying the weights in the output layer by Δ_{output} :

$$error_{hidden} = \Delta output \times [weights_{hidden \leftrightarrow output}]$$

$$error_{hidden} = -0.1352962103 \times \begin{bmatrix} 0.3 \\ 0.5 \\ 0.9 \end{bmatrix} = \begin{bmatrix} -0.04058886310 \\ -0.06764810517 \\ -0.12176658930 \end{bmatrix}$$

We can now find the *output delta* for the hidden layer. This gives an indication of how best to shift the first set of weights.

$$\Delta output_{hidden} = error_{hidden} \times S'(layer_{hidden})$$

$$\Delta output_{hidden} = \begin{bmatrix} -0.04058886310 \\ -0.06764810517 \\ -0.12176658930 \end{bmatrix} \times S' \left(\begin{bmatrix} 0.7310585786 \\ 0.7858349830 \\ 0.6899744811 \end{bmatrix} \right)$$

$$\Delta output_{hidden} = \begin{bmatrix} -0.007980254842 \\ -0.011385065320 \\ -0.026047054160 \end{bmatrix}$$

The next step is find the weight adjustment amounts. the process is as follows:

For the weights between the input and hidden layers:

$$\Delta weights_{input \leftrightarrow hidden} = [input] \cdot [\Delta output_{hidden}]$$

$$\Delta weights_{input \leftrightarrow hidden} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot [-0.007980254842 \quad -0.01138506532 \quad -0.02604705416]$$

$$\Delta weights_{input \leftrightarrow hidden} = \begin{bmatrix} -0.007980254842 & -0.01138506532 & -0.02604705416 \\ -0.007980254842 & -0.01138506532 & -0.02604705416 \end{bmatrix}$$

And for the weights between the hidden and output layers:

$$\Delta weights_{hidden \leftrightarrow output} = [layer_{hidden}] \cdot [\Delta output]$$

$$\Delta weights_{hidden \leftrightarrow output} = [0.7310585786 \quad 0.7858349830 \quad 0.6899744811] \cdot [-0.1352962103]$$

$$\Delta weights_{hidden \leftrightarrow output} = \begin{bmatrix} -0.09890945522 \\ -0.10632049520 \\ -0.09335093252 \end{bmatrix}$$

The weights can now be updated using:

$$weight_i^+ = weight_i + \Delta weight_i$$

The above process must then be repeated a sufficient number of times until the network converges on a solution.

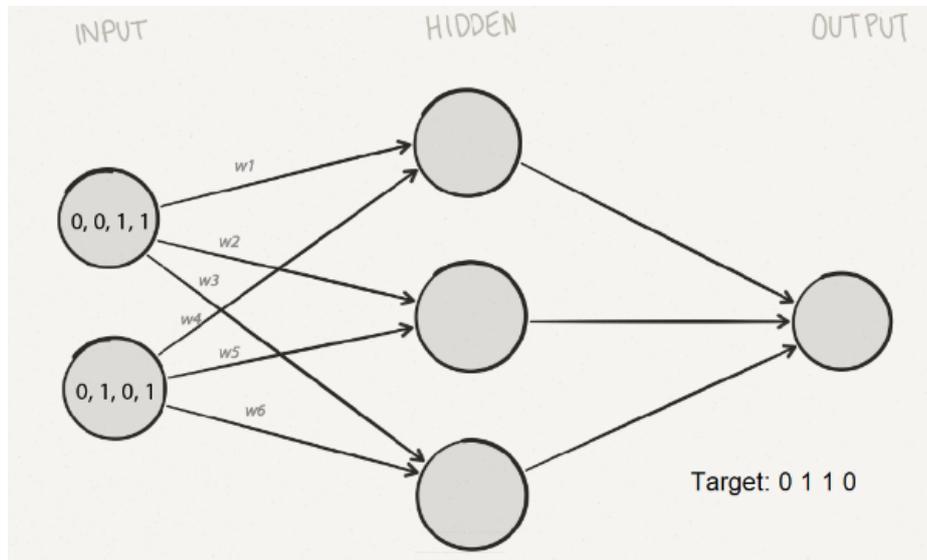
Using the above as a guide write a function that will take the shape of a network, a set of expected results and their matching inputs and return the weights required to perform the operation defined by the input/target set.

- Your starting set of weights should be set random numbers between -1 and 1.
- The number of iterations required to train the weights is discretionary.

Your submission shall be considered complete if your code satisfies the primary goal of training a 2-3-1 layer XOR network based of the above example, but the following additional features will be considered favourable:

- Your network can be used to train other Boolean operations such as AND or NAND
- Your function can handle a variable number of nodes in the hidden layer (but always 2 input nodes and one output node).
- Your function can be trained based on the complete set of inputs.

For example your network would look like:



Where your hidden layer sum would be:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \omega_1 & \omega_2 & \omega_3 \\ \omega_4 & \omega_5 & \omega_6 \end{bmatrix}$$

BackProp has the following syntax:

```
weights ← network_shape BackProp (input target)
```

network_shape a numeric vector describing the number of nodes in each of the layers

weights are the resultant weights needed for the specified network to compute XOR

Examples:

]boxing on

```
A a network with 3 layers,
```

```
A 2 input nodes,
```

```
A 3 hidden nodes
```

```
A 1 A output node
```

```
network_shape ← 2 3 1
```

```
input ← (1 1)(1 0)(0 1)(0 0)
```

```
target ← 0 1 1 0
```

```
□ ← weights ← network_shape BackProp input target
```

120.4374354	1.493461405	-119.3929741	-42.66514616
-118.8198287	1.486570786	121.0097061	64.25818411
			-42.59829271

Note: Due to the nature of the problem it is likely that your results will be different to those above. As long as the resultant weights can be used to perform the operation they were trained for with sufficient accuracy, your submission stands in good stead.