2016 APL Problem Solving Competition – Phase 2 Problem Descriptions

Overview

This year's Phase II problems are divided into three sets, representing three categories – Bioinformatics, Finance and General Computing. Each set has three problems; a problem can be broken down into multiple tasks.

Three grand prizes will be awarded, one in each of the categories.
To be considered for a grand prize, you must solve all of the problems in a category.

You may submit an entry that completes more than one category. However, you can only win once – for example, if your entries in all three categories are judged to be the best in each category, the judging committee will select the entry they deem the best and declare you the winner in that category while other competitors will be selected as the winners of the other categories.

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 Contest2016 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 `Contest2016.DWS`

The Contest2016 workspace contains:

- Three namespaces, one for each of the problem categories. Each namespace will contain:
    - stubs for all of the functions described in the problem descriptions. The function stubs are implemented as traditional APL functions (tradfns) but you can change the implementation to dynamic functions (dfns) if you want to do so. Either form is acceptable.
    - any sample data elements mentioned in the problem descriptions.

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

  The namespaces are:

    - `#.bio` – for the bioinformatics problem set
    - `#.fin` – for the finance problem set
    - `#.gen` – for the general computing 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 **Contest2016.dyalog** which will contain any code or data you placed in the **`#.bio`**, **`#.fin`**, and **`#.gen`** namespaces. You will then need to upload the **Contest2016.dyalog** file using the contest website.

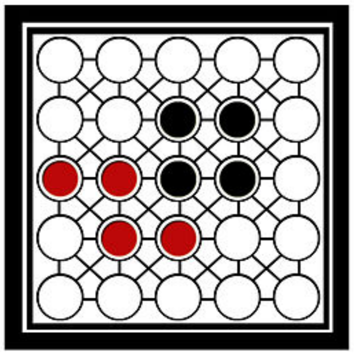
If you use some other APL system, you can use the template script file `Contest2016.dyalog`

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

Set 3: General Computing Problems

General Computing Problem 1 – (1 task)

Teeko



According to Wikipedia, Teeko is an abstract strategy game invented by John Scarne in 1937 and rereleased in refined form in 1952 and again in the 1960s. The Teeko board consists of twenty-five spaces arranged in a five-by-five grid. There are eight markers in a Teeko game, four black and four red. One player, "Black" plays the black markers, and the other, "Red", plays the red. Black moves first and places one marker on any space on the board. Red then places a marker on any unoccupied space; black does the same; and so on until all eight markers are on the board. The object of the game is for either player to win by having all four of his markers in a straight line (vertical, horizontal, or diagonal) or on a square of four adjacent spaces. Adjacency is horizontal, vertical, or diagonal, but does not wrap around the edges of the board. If neither player has won after the "drop" (when all eight pieces are on the board), then they move their pieces one at a time, with Black playing first. A piece may be moved only to an adjacent space.

In APL, a Teeko board can easily be represented as a 5 × 5 matrix of different values for empty, black, and red spaces. For instance the board in the image above could be represented by:

```
      ⊢board←5 5⍴'∘∘∘∘∘∘∘BB∘RRBB∘∘RR∘∘∘∘∘∘∘'
∘∘∘∘∘
∘∘BB∘
RRBB∘
∘RR∘∘
∘∘∘∘∘
```

***Task 1** – Do We Have a Winner?*

Write an APL function named `winner` which:

- takes a right argument of a 5×5 character matrix representing a Teeko board
- if there is a winner, returns the appropriate character for the winner, otherwise returns the character representing an unoccupied space.

Examples:
Using `board` as defined above:

```
      winner board
B
      ⊢board2←5 5⍴'∘∘∘∘∘∘∘B∘BRRBB∘∘RR∘∘∘∘∘∘∘'
∘∘∘∘∘
∘∘B∘B
RRBB∘
∘RR∘∘
∘∘∘∘∘

      winner board2
∘
```

General Computing Problem 2 – (2 tasks)

Chi-Square Test of Independence

A Chi-Square test for independence is applied when you have two categorical variables from a single population. It is used to determine whether there is a significant association between the two variables.

The Chi-Square test statistic is calculated using the following formula:

$$X = \sum_i \sum_j \frac{\left(OBS_{ij} - EXP_{ij}\right)^2}{EXP_{ij}}$$

The observed values are the number of observations in each category.
The expected values (EXP) are calculated as follows:

$$EXP_{ij} = \frac{\sum_i n_{ij} \sum_j n_{ij}}{\sum_i \sum_j n_{ij}}$$

Where n_{ij} = The number of observations in category i and in category j

For example, in a class of 38 students there are the following counts:

Observed Values	Left	Middle	Right	Total
Female	3	2	4	9
Male	8	9	12	29
Total	11	11	16	38

If the variable GENDER is independent of the variable POLITICS, then the expected counts would be:

Expected Values	Left	Middle	Right	Total
Female	2.6053	2.6053	3.7895	9
Male	8.3947	8.3947	12.211	29
Total	11	11	16	38

Observe that the row totals and column totals remain the same (the displayed numbers are rounded).

The expected values are calculated from the observed values.
For example, the expected number of Left Females is 2.6053:

$$\frac{\boldsymbol{NUMBER\ OF\ LEFTs \times NUMBER\ OF\ FEMALEs}}{\boldsymbol{NUMBER\ IN\ SURVEY}} = \frac{\mathbf{11 \times 9}}{\mathbf{38}} = \mathbf{2.6053}$$

Task 1 ***– Meeting Expectations***

Write an APL function named **`expected`** which:

- takes a right argument of a numeric matrix of counts
- returns a numeric matrix of expected values

Example:

```
      ⊢counts←2 3⍴3 2 4 8 9 12
3 2  4
8 9 12
      expected counts
2.605263158 2.605263158  3.789473684
8.394736842 8.394736842 12.21052632
```

Task 2 ***– Chi-Square Test***

Write an APL function named **`chiSquareTest`** which:

- takes a right argument of a numeric matrix of counts
- returns the Chi-square test statistic

Example:

```
      ⊢counts←2 3⍴3 2 4 8 9 12
3 2  4
8 9 12
      chiSquareTest counts
0.2779519331
```

General Computing Problem 3 – What's in a Name? (1 task)

For this problem, imagine that you work in the pharmaceutical industry.
Your company has a database which consists of a list of normalized drug names.
Your company also collects information from physicians regarding the drugs they are prescribing. Ideally, all of the names in the list from physicians would be found in the database of normalized names. However, data entry errors can occur or the physician's office may enter additional, unnecessary, information – a dosage amount in addition to the drug name for instance. Therefore there are a number of entries that do not match exactly. The mismatched names need to be analyzed and matched, if possible, with the database names. Identifying and correcting the mismatches by hand is a costly process. As such, you've been tasked with automating as much of the process as possible.

The two lists are found in text files included in the zip file downloaded here. The database data is found in **/data/drugs.txt** and the input data is found in **/data/inputs.txt** in the folder to which you unzipped the competition files.

```
database←{(⎕NUNTIE ⍵)⊢('.'⎕R'\u0'⎕OPT'Mode' 'L')⍵}'unzipped_path/data/drugs.txt' ⎕NTIE 0
inputs←{(⎕NUNTIE ⍵)⊢('.'⎕R'\u0'⎕OPT'Mode' 'L')⍵}'unzipped_path/data/inputs.txt' ⎕NTIE 0
```

You can copy and paste the above statements into your Dyalog APL session and execute them after replacing "unzipped_path" with the actual path name where you unzipped the competition files.
The statements will (for their respective text files):

- tie (open) the text file (using ⎕NTIE)
- read the contents of the file, splitting each line into a separate element (⎕OPT 'Mode' 'L')
- normalize the elements by making them uppercase ('.'⎕R'\u0')
- untie (close) the text file (using ⎕NUNTIE)
- return the normalized elements (⊢)

Pretty cool, huh?

If you use another APL to solve this problem, you can use whatever tools are available in that environment to accomplish the same.

The sample data files contain 2729 drug names and 4400 inputs.

Task 1 – *Match Maker*

Write an APL function named `matchDrugs` which:

- takes a right argument representing the list of physician inputs
- takes a left argument representing the list of database drug names
- returns an integer vector where each element is index in the list of drug names representing the best match for the corresponding element in the list of inputs.

Example:

```
      d←'ASPIRIN' 'METAMUCIL' 'PENICILLIN'
      i←'PENICILIN' 'ASPIRIN 250MG' 'METAMUSEL' 'ASPRIN'
      d matchDrugs i
3 1 2 1
```