2019 APL Problem Solving Competition – Phase II Problem Descriptions

Errata

This section will contain descriptions of corrections, if any, that are made to this document after its initial release. When such corrections are made, newer versions of this document and the **Contest2019.zip** file will be uploaded to the contest website. If you have registered for the competition, then we will notify you by email when updated Phase II materials are made available.

5 June 2019

The following amendments were made:

- The input in the example for Medium Problem Set 1, Problem 2 – Double Degree Array, has been corrected from:

```
DoubleDegrees 5 2 ⍴ 1 2 2 3 4 3 2 4
```

  to:

```
DoubleDegrees 5 2 ⍴ 5 4 1 2 2 3 4 3 2 4
```

- The input in the example for Medium Problem Set 1, Problem 3 – Connected Components, has been corrected from:

```
Components 14 2 ⍴ 1 2 1 5 5 9 5 10 9 10 3 4 3 7 3 8 4 8 7 11 8 11 11 12 8 12
```

  to:

```
Components 14 2 ⍴ 12 13 1 2 1 5 5 9 5 10 9 10 3 4 3 7 3 8 4 8 7 11 8 11 11 12 8 12
```

- The input in the example for Easy Problem Set 3, Problem 2 – Num Num Num has been corrected from:

```
NumNumNum '_10713223141516271819'
```

  to:

```
NumNumNum '10713223141516271819'
```

- A missing "be" has been inserted in the introductory remarks for Easy Problem Set 2:
  …the argument to the functions you will write will be a character array which could be a scalar…
- A superscript in the descripton for Medium Problem Set 2, Problem 3 – Romberg's Method, has been corrected from R_n^n to $R_n^{\boldsymbol{m}}$

11 June 2019

The following amendments were made:

- The output in the examples for Difficult Problem Set 2, Problem 1 – Imagine the Possibilities has been corrected from:

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R w KQ -' ⍝ White to move
```

Ra2	Ra3	Rb1	Rc1	Rd1+	Kd1	Kd2	Kxf2	O-O-O	Rf1	Rg1	a5	b8=Q+	b8=R+	b8=B	b8=N	h3	h4

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R b KQ -' ⍝ Black to move
```

Kc7	Kd7	Ke7	Ke8	i4	Rxc2+	Rg2	Rxh2	Ri8+	Ri6	Ri5

  To:

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R w KQ -' ⍝ White to move
```

Ra2	Ra3	Rb1	Rc1	Rd1+	Kd1	Kd2	Kxf2	O-O-O	Rf1	Rg1	a5	b8=Q+	b8=R+	b8=B	b8=N	e3	e4	h3	h4

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R b KQ -' ⍝ Black to move
```

Kc7	Kd7	Ke7	Ke8	f4	Rxc2+	Rg2	Rxh2	Rf8+	Rf6	Rf5

- A superscript in the descripton of the result for Medium Problem Set 2, Problem 3 – Romberg's Method, has been corrected from R_n^m to $R_n^{\boldsymbol{n}}$.

15 June 2019

The following amendments were made:

- The output in the examples for Difficult Problem Set 2, Problem 1 – Imagine the Possibilities has been corrected from:

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R w KQ -' ⍝ White to move
```

Ra2	Ra3	Rb1	Rc1	Rd1+	Kd1	Kd2	Kxf2	O-O-O	Rf1	Rg1	a5	b8=Q+	b8=R+	b8=B	b8=N	e3	e4	h3	h4

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R b KQ -' ⍝ Black to move
```

Kc7	Kd7	Ke7	Ke8	f4	Rxc2+	Rg2	Rxh2	Rf8+	Rf6	Rf5

To:

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R w KQ -' ⍝ White to move
```

Ra2	Ra3	Rb1	Rc1	Rd1+	Kd1	Kd2	Kxf2	O-O-O+	Rf1	Rg1	a5	b8=Q+	b8=R+	b8=B	b8=N	e3	e4	h3	h4

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R b KQ -' ⍝ Black to move
```

Kc7	Kd7	Ke7	Ke8	f4	Rxe2+	Rg2	Rxh2	Rf1+	Rf3	Rf4

- The syntax description for Medium Problem Set 2, Task 3 – Romberg's Method has corrected from

  Write an APL operator, **`Romberg`**, that:
  - takes a left operand which is a scalar function.
  - takes a positive integer left argument which is an integer number of subintervals.
  - takes a 2-element numeric vector right argument which represents an interval [a,b] where a < b.
  - returns $\boldsymbol{R_n^n}$ for the given function and interval.

  To:

  Write an APL operator, **Romberg**, that:
  - takes a left operand which is a scalar function.
  - takes an integer left argument greater than or equal to 0 representing *n*.
  - takes a 2-element numeric vector right argument which represents an interval [a,b] where a < b.
  - returns $\boldsymbol{R_n^n}$ for the given function and interval.

- The output in the examples for Difficult Problem Set 1, Problem 3 – Boggle Minding or Mind Boggling? has been corrected from:

OUR	THOU	ROUGH	ROUTH	THOUGH	THROUGH

To:

OU	OUR	THOU	ROUGH	ROUTH	THOUGH	THROUGH

And:

```
      ≠found←SOWPODS FindWords board
167
      found[10↑⍒≠⍤found]
```

ANTIRUST	QUIETUS	RUINATE	URINATE	GENIUS	GLUTEN	GUINEA	GUNITE	LUNATE	SULTAN

To:

```
      ≠found←SOWPODS FindWords board
188
      found[10↑⍒≠⍤found]
```

ANTIRUST	QUIETUS	GLUTAEI	RUINATE	URINATE	GENIUS	GLUTEI	GLUTEN	GUINEA	GUNITE

- The output in the examples for Difficult Problem Set 1, Problem 4 – Making a Point, has been corrected from:

```
      BoggleScore found      ⍝ the variable found is from the previous problem
195 143

      ⍝ putting it all together
      BoggleScore (GetWordList 'sowpods.txt') FindWords ⎕←4 4 MakeBoard BoggleDice
NGET
RINR
KLCC
PTHM
129 91
```

To:

```
      BoggleScore found      ⍝ the variable found is from the previous problem
240 164

      ⍝ putting it all together
      BoggleScore (GetWordList 'sowpods.txt') FindWords ⎕←4 4 MakeBoard BoggleDice
NGET
RINR
KLCC
PTHM
152 98
```

7 July 2019

The following amendment was made:

The image showing the initial chess board position in the introduction for Difficult Problem Set 2, How About A Nice Game of Chess?, was incorrect and has been replaced by a correct image. This change affects only the introduction and none of the problems in the problem set.

Welcome!

This year's Phase II consists of nine problem sets across a variety of domains, including Bioinformatics, Mathematics, Data Science, Gaming, Cryptography, Language Processing and Recreational Computing. There are three levels of problem set difficulty – low, medium and hard. A problem set consists of three or more problems, and each problem will require you to write one or more functions or operators. You need to solve all of the problems within a problem set for the problem set to be considered for judging. You do not need to solve all the Phase II problem sets to be considered eligible for the top three prizes, but you do need to complete, at a minimum, one problem set of each level of difficulty.

You can solve more than the minimum number of problem sets; you can solve all the problem sets if you choose to do so. Solving more than the minimum problem sets can work in your favor in two ways:

- We judge based on what we deem to be your best submission from each level of difficulty, so solving more problem sets will never lower your score.
- If entries from two people are of similar quality but one person has solved more problem sets, then that entry will receive higher consideration by the judging committee.

Good Luck and Happy Problem Solving!

Note

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

```
      ('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 │ │
└──────────────┴──────────────┴─┘
```

How to Participate in Phase II

You'll need to download the file named Contest2019.zip from the competition website https://dyalogaplcompetition.com. This file contains several files including:

- This document (which is also separately downloadable).
- Contest2019.dws – the contest workspace to use if you want to use Dyalog APL's editor.
- Contest2019.dyalog – a template namespace script that contains syntactically-correct stub functions for all Phase II problems. This allows you to use your preferred editor to build your solutions.
- Any data files necessary to solve the problems.

Develop your solutions using either the workspace or the template.

Your solutions should consist of only functions or operators and not depend on any global variables. You may write other functions as you deem necessary to organize your code.

Using the Contest2019.dws workspace

To use the Contest2019.dws workspace, you must use Dyalog APL v16.0 or later. If you use an earlier version of Dyalog APL, you should the Contest2019.dyalog template file described below. The workspace contains:

- `#.Problems` – a namespace in which you will develop your solutions. `#.Problems` contains:
  - syntactically-correct stubs for all of the functions described in the problem descriptions.
    The function stubs are implemented as traditional APL functions (tradfns) but you can change the implementation to dynamic functions (dfns) if you want to do so.
    Either form is acceptable.
  - any sample data elements mentioned in the problem descriptions.
  - any sub-functions you develop as a part of your solution should be located in `#.Problems`
- `#.SubmitMe` – a function used to package your solution for submission. On Microsoft Windows this function will display a GUI form to allow you to enter a description of yourself and any feedback on the competition. On non-Windows platforms, you'll be presented with a character-based prompt and response interaction.

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 **Contest2019.dyalog** which will contain any code or data you placed in the `#.Problems` namespace. You will then upload the **Contest2019.dyalog** file using the contest website.

Using the Contest2019.dyalog template file

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 `AboutMe` and `Reaction` variable definitions as indicated at the top of the file and upload the file using the contest website.

If you use a non-Dyalog APL system to develop your application, it will still need to execute under Dyalog APL; it is recommended that your solution uses APL features that are common between your APL system and Dyalog.

If use Dyalog APL, including versions prior to v16.0, you can load the Contest2019.dyalog file into your session using the `]load` user command. For instance, assuming you've unzipped the zip file into a folder named /contest/, you would use

```
]load /contest/Contest2019
```

Use of tacit or derived functions in Phase II

If you use the `Contest2019.dws` workspace and you want to use tacit or derived functions in your Phase II solutions, they need to either:

- Be defined inline within a tradfn or dfn as in

```
    ∇ r←foo w;avg
[1]   avg←+/÷≢
[2]   ...
      ∇
```

or

```
goo←{
      avg←+⌿÷≢
      ...
}
```

- Be "wrapped" in a tradfn as in

```
      ∇ avg←avg
[1]   avg←+⌿÷≢
      ∇
```

If you use the `Contest2019.dyalog` template file, you may enter tacit or derived functions directly in the file.

Easy Problem Sets

Easy Problem Set 1 – Shhh! It's a Secret!

Secret codes and ciphers have been used throughout history in attempts to keep communication private. In this problem we will explore a few of the methods in use before the advent of the electronic computer.

Easy Problem Set 1, Problem 1 – *Cooking on the Grille*

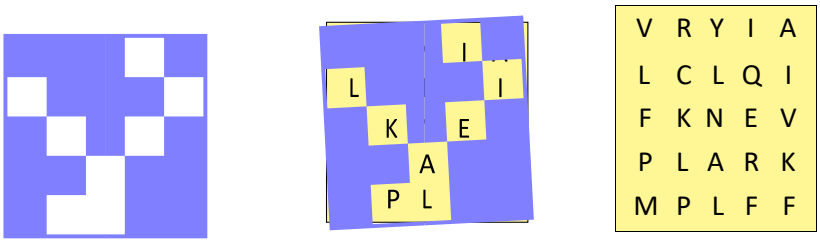


Figure 1 – The application of a grille to a character matrix.

A Grille is a cryptographic device consisting of a square sheet with holes cut out which, when laid on top of a similarly-sized character matrix, reveals a hidden message (see figure 1).

Write an APL function, **`Grille`**, that implements an electronic version of a grille which:

- takes a character matrix left argument, where a hash (`'#'`) represents opaque material and a space (`' '`) represents a hole.
- takes a similarly-shaped character matrix right argument containing the message.
- returns the hidden message as a character vector.

Example

```
      (2 2⍴'# # ') Grille 2 2⍴'LHOI'
HI

      grid←6 6⍴'ESVWGTHOWTHZHIVSAICASSACFAAUCMNYMPCE'
      grille← 6 6⍴'##### #### ## # # ## ### ## ## #####'
      grille grid  ⍝ display both
┌──────┬──────┐
│##### │ESVWGT│
│#### #│HOWTHZ│
│# # # │HIVSAI│
│## ###│CASSAC│
│ ## ##│FAAUCM│
│ #####│NYMPCE│
└──────┴──────┘
      grille Grille grid
THISISFUN
```

Easy Problem Set 1, Problem 2* – *The Secret is the Key

Figure 2 – Columnar Transposition

In a columnar transposition cipher the plaintext is entered into a character matrix row-wise and the ciphertext is read out column-wise, in a way controlled by a secret keyword. The length of the keyword determines the width of the matrix, and the order of the letters in the keyword determines the order of the columns in the ciphertext as shown in figure 2. In case the same letter appears more than once in the secret key, we assume a left-to-right ordering. If the length of the message is not a multiple of the length of the keyword, pad the message with spaces to make it so.

Write a pair of APL functions, **`TransEnc`** and **`TransDec`**, where:

- **`TransEnc`**
  - takes a character vector left argument secret key containing at least one element.
  - takes character vector right argument of the plaintext to encode.
  - returns a character vector result from applying the columnar transposition as described above.
- **`TransDec`**
  - takes a character vector left argument secret key containing at least one element.
  - takes character vector right argument of a columnar transposition message.
  - returns the character vector plaintext message.

Examples

```
      'APL' TransEnc 'OUT OF ORDER'
O  DTFRRUOOE

      'APL' TransDec 'O  DTFRRUOOE'
OUT OF ORDER

      'BANANA' TransDec ⎕←'BANANA' TransEnc 'COLUMNAR TRANSPOSITION'
ORSIUTONNAI CANTL POMRS
COLUMNAR TRANSPOSITION
```

Easy Problem Set 1, Problem 3 – *When in Prison, Do as the Prisoners Do*

A tap code isn't used so much to conceal messages but rather to transmit them when the medium of transmission is limited. Tap codes have been commonly used by prisoners to communicate with each other by tapping either the metal bars, pipes or the walls inside a cell. To use a tap code, each character is first converted to a pair of numbers, by looking up its row and column (respectively) in a Polybius square (see table 1). Since the Polybius square only has 25 slots, I and J share the same encoding and must be distinguished from context by the receiver. Then the numbers are transmitted as sequences of taps or knocks, with short pauses in between.

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I/J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

Table 1 – Polybius Square

Write a pair of APL functions, **`TapEnc`** and **`TapDec`**, where:

- **`TapEnc`**
  - takes character scalar or vector right argument of the message to convert to tap code.
  - returns a character vector representing the tap code where asterisk (`'*'`) represents a tap and space (`' '`) represents a pause.
- **`TapDec`**
  - takes character vector right argument representing a tap code where an asterisk (`'*'`) represents a tap and space (`' '`) represents a pause.
  - returns the character vector message represented by the tap code.
  - For any message that uses the I/J cell (`** ****`) you may assume it is an I.

Examples

```
      TapEnc 'APL'
* * *** ***** *** *

      TapDec '* * *** ***** *** *'
APL
```

Easy Problem Set 2 – Say What??

Processing natural language is a basic skill for many humans. However, computers have a much more difficult time. The field of processing and analyzing human language using computers is known as Natural Language Processing.

In each of the tasks below:

- you may assume that the inputs will contain only the letters "a-z", "A-Z" and spaces; text is assumed to be English (no accented characters!).
- "aeiouAEIOU" are considered to be the only vowels ("yY" are not considered a vowel for these tasks).
- the argument to the functions you will write will be a character array which could be a scalar (single character) or a vector of length 0 or more. If it's a vector, it might be a single word (no spaces), or a set of words separated by spaces.

Easy Problem Set 2, Problem 1 – Ubbi Dubbi

Ubbi dubbi is a language game spoken with the English language. It was popularized in the 1970s in the United States by the Public Broadcasting Service television show Zoom. There's even a Google Chrome extension to translate a web page's content into Ubbi Dubbi. The basic rule for Ubbi Dubbi is to insert "ub" before each vowel sound.

For instance, "hello" becomes "hubellubo".

While this may be fairly easy to understand, it's difficult to implement precisely without having an understanding of the specific pronunciation of a given word, so we've simplified it for this competition – rather than inserting "ub" before each vowel sound, insert "ub" before each vowel. Our goal is not to implement the perfect Ubbi Dubbi translator, but to see how you manipulate character vectors. 😊

Write an APL function, **`UbbiDubbi,`** that:

- takes a character array right argument as described in the problem set introduction.
- returns a character vector with "ub" inserted before every vowel.

Example

```
      UbbiDubbi 'I am a good example'
ubI ubam uba guboubod ubexubamplube

      UbbiDubbi 'single'
subinglube
```

Easy Problem Set 2, Problem 2 – *Rövarspråket*

Rövarspråket is a Swedish language game. The English translation of Rövarspråket is "the robber language". It became popular after the books about Bill Bergson by Astrid Lindgren, where the children use it as a code, both at play and in solving actual crimes. Today, the books (and subsequent films) are so well known in Sweden, and also in Norway, that the language is part of the culture of schoolchildren. Most Scandinavians are familiar with it.

The formula for encoding is simple. Every consonant (spelling matters, not pronunciation) is doubled, and an "o" is inserted in between the newly-doubled letters. Vowels ("aeiou") are left intact.

Write an APL function, **`Rovarspraket`**, that:

- takes a character arrary right argument as described in the problem set introduction.
- returns a character vector of the Rövarspråket translation of the argument.

Examples

```
      Rovarspraket 'hello'
hohelollolo

      Rovarspraket 'so long'
soso lolonongog

      Rovarspraket 'a'
a

      Rovarspraket 'My dog ate the cat'
MoMyoy dodogog atote tothohe cocatot
```

Easy Problem Set 2, Problem 3* – *Pig Latin

Pig Latin is a language game in which words in English are altered to conceal the words from others not familiar with the rules. To do this, the following transformations are applied:

- For words that begin with consonants, the letters before the first vowel move to the end and then "ay" is added as a suffix; "good day stranger" becomes "oodgay ayday angerstray".
- For words that begin with vowels, the 'ay' suffix is added; "every august day" becomes "everyay augustay ayday".
- For single vowel words ('A', 'a', 'I'), the suffix 'way' is added; "I ran faster" become "Iway anray asterfay".

Strictly, Pig Latin only applies the rule about words starting with consonants to words that sound like they start with a consonant. This can make the rule difficult to implement without having an understanding of the specific pronunciation of a given word, so we've simplified it for this competition – the rule for words that start with consonants is applied irrespective of the true pronunciation of that word so that "honest" (with a silent "h") becomes "onesthay " rather than the strictly-correct "honestay". Our goal is not to implement the perfect Pig Latin translator, but to see how you manipulate character vectors. 😊

Write an APL function, **`PigLatin`**, that:

- takes a character array right argument as described in the problem set introduction.
- returns a character vector of the Pig Latin translation of the argument.

Example

```
      PigLatin 'Hello honest one'
elloHay onesthay oneay

      PigLatin 'I won one game'
Iway onway oneay amegay

      PigLatin 'a'
away
```

Easy Problem Set 3 – Potpourri

This problem set presents a collection of unrelated problems that nonetheless seemed interesting enough to include in the competition.

Easy Problem Set 3, Problem 1 – What's Your Address?

Devices on the Internet are assigned a unique IP address for identification and location definition. Internet Protocol version 6 (IPv6) uses a 128-bit address and theoretically has up to 2^{128} addresses. IPv6 addresses are represented as eight groups of four hexadecimal digits, with the groups being separated by colons, for example, 2001:0db8:0000:0042:0000:8a2e:0370:7334, but methods to abbreviate this full notation exist.

For convenience, an IPv6 address can be abbreviated to shorter notations by application of the following rules:

- One or more leading zeros from any groups of hexadecimal digits are removed; this is usually done to either all or none of the leading zeros. For example, the group 0042 is converted to 42.
- Consecutive sections of zeros are replaced with a double colon (::). The double colon can only be used once in an address, as multiple use would render the address indeterminate.

Write a pair of APL functions, **`AbbreviateIPv6`** and **`ExpandIPv6`** where:

- **`AbbreviateIPv6`**
  - takes a 39-element character vector right argument representing a full IPv6 address.
  - returns a character vector result representing the abbreviated form of that address.

  Note: You may want to use https://www.ultratools.com/tools/ipv6Compress to check your results.

- **`ExpandIPv6`**
  - takes character vector right argument representing an abbreviated IPv6 address.
  - returns a 39-element character vector representing the full IPv6 address.

  Note: Do not use https://www.ultratools.com/tools/ipv6Expand to check your results as it does not fully expand the address with leading zeros.

Both **`AbbreviateIPv6`** and **`ExpandIPv6`** should use uppercase letters in their hexadecimal representations.

Example

```
      AbbreviateIPv6 '2001:0DB8:0000:0042:0000:8A2E:0370:7334'
2001:DB8:0:42:0:8A2E:370:7334

      ExpandIPv6 '2001:DB8:0:42:0:8A2E:370:7334'
2001:0DB8:0000:0042:0000:8A2E:0370:7334

      ExpandIPv6 ⎕←AbbreviateIPv6 '2041:0000:140F:0000:0000:0000:875B:131B'
2041:0:140F::875B:131B
2041:0000:140F:0000:0000:0000:875B:131B
```

Easy Problem Set 3, Problem 2 – Num Num Num

This problem was Inspired by the "The Riddler" column – https://fivethirtyeight.com/features/how-many-numbers-contain-the-numbers-of-their-numbers/.

The number 21322314 acts as its own inventory. That is, it contains two 1s, three 2s, two 3s and one 4. Another example is 22 — it contains two 2s. These numbers consist of alternating tallies and numerals. First comes the tally, then the numeral being tallied, then another tally, and so on. These are also known as autobiographical numbers and are described in The On-Line Encyclopedia of Integer Sequences at https://oeis.org/A047841.

Write an APL function, **NumNumNum**, that determines whether the number represented by its argument is an autobiographical number. **NumNumNum**:

- takes a character vector right argument representing the number to be analyzed. We use a character vector argument because the larger autobiographical numbers exceed a 32-bit integer representation. There are ways around this in Dyalog APL, but they are ignored to keep this problem simple.
- returns a Boolean scalar indicating whether the number is an autobiographical number (1) or not (0).

For the purposes of this problem, there are two key differences from the references:

- the numerals can be tallied in any order, so '21321423' and '21322314' would both qualify as autobiographical.
- The tally will always be a single digit, for example, the number '101112213141516171819', which has eleven 1s, is outside the domain of this function (this is to keep the problem simple). This means that all qualifying numbers will have an even number of digits.

Example

```
      NumNumNum '21322314'
1

      NumNumNum  '32142123'
1

      NumNumNum ¨ '22' '42'
1 0

      NumNumNum '107132231415162718 19'
1
```

Easy Problem Set 3, Problem 3* – *Watch Out for That Mine!

Minesweeper is a single-player puzzle video game. The objective of the game is to clear a rectangular board containing hidden "mines" or bombs without detonating any of them, with help from clues about the number of neighboring mines in each field. The game originates from the 1960s and has been written for many computing platforms in use today.

The player is initially presented with a grid of undifferentiated squares. Some randomly selected squares, unknown to the player, are designated to contain mines. The game is played by revealing squares of the grid by clicking or otherwise indicating each square. If a square containing a mine is revealed, the player loses the game. If no mine is revealed, a digit is instead displayed in the square, indicating how many adjacent squares contain mines.

In our version, the board will be represented by an integer matrix where `¯1` indicates a mine, and all other cells are populated with the count of adjacent mines (`¯1`s).

Write a pair of APL functions, **`MakeMines`** and **`CountMines`** where:

- **`MakeMines`**
  - takes a 2-element integer vector right argument representing the rows (height) and columns (width) of the board to create.
  - takes a single integer left argument indicating the number of mines to randomly place on the board.
  - returns an integer matrix of the specified dimensions with the correct number of mines placed randomly.
- **`CountMines`**
  - takes an integer matrix right argument of a board as returned by **`MakeMines`**.
  - returns an integer matrix of the same dimension where each cell containing a 0 has been updated to reflect the number of mines in immediately-adjacent cells.

Example
(note that your results will likely be different due to the randomness of the problem)

```
      ⎕←board←11 MakeMines 5 8   ⍝ make a 5×8 board with 11 mines
 0 ¯1  0  0  0 0 0  0
¯1 ¯1 ¯1  0  0 0 0 ¯1
 0 ¯1 ¯1 ¯1  0 0 0  0
 0  0  0  0 ¯1 0 0  0
 0  0 ¯1  0  0 0 0 ¯1

      CountMines board
 3 ¯1  3  1  0 0 1  1
¯1 ¯1 ¯1  3  1 0 1 ¯1
 3 ¯1 ¯1 ¯1  2 1 1  1
 1  3  4  4 ¯1 1 1  1
 0  1 ¯1  2  1 1 1 ¯1
```

Medium Problem Sets

Medium Problem Set 1 – Bioinformatics

Rosalind.info is a platform for learning bioinformatics through problem solving. We've found the expressive power and elegance of APL to be well-suited for these types of problems, often resulting in solutions of only a few lines where other languages need dozens. If you have an interest in bioinformatics, or problem solving in general, we encourage you to explore rosalind.info (and of course, APL 😊).

In the **/rosalind/** folder in the **Contest2019.zip** file we have included three sample dataset files downloaded from rosalind.info for you to test your solutions. All three are suitable for testing the first two problems, however, rosalind_graph3.txt is recommended for testing the third problem. You can also register for rosalind.info yourself to obtain other datasets and verify your solutions.

If you are using Dyalog APL to solve these problems, then you can easily read in the files using:

```
graph←↑⍎¨⎕NGET <filename> 1
```

For example:

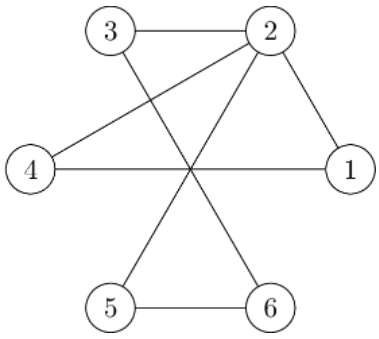
```
graph←↑⍎¨⎕NGET 'c:/Contest2019/rosalind/rosalind_graph1.txt' 1
```

If you're using some other APL system, refer to their documentation for reading files.

Medium Problem Set 1, Problem 1 – Degree Array

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

Edge list format is a technique for representing a graph with a numeric matrix. The first row contains metadata about the graph and the remaining rows contain information about the graph's edges (connections between vertices). A sample graph and its representation in edge list format are found below.



6 7 – 6 vertices, 7 edges; the remaining rows describe the 7 edges
1 2
2 3
6 3
5 6
2 5
2 4
4 1

A degree array indicates the degree, or number of edges, that meet at a given vertex.

Write an APL function, **`Degrees`**, where:

- the right argument is a 2-column integer matrix representing a graph in edge list format.
- returns an integer vector result where the **i**th element indicates the degree of vertex **i**.

Examples (using the above data)

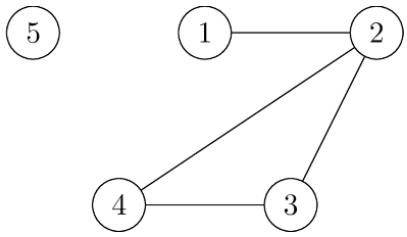
```
      Degrees 8 2 ⍴ 6 7 1 2 2 3 6 3 5 6 2 5 2 4 1 4
2 4 2 2 2 2

      Degrees 1 2 ⍴ 4 0 ⍝ 4 vertices, 0 edges
0 0 0 0
```

Medium Problem Set 1, Problem 2* – *Double Degree Array

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

A double degree array is similar to the degree array in problem 1, except that the i^{th} element of the result is the sum of the degrees of i's neighbors.



5 4 – 5 vertices, 4 edges; the remaining rows describe the 4 edges
1 2
2 3
4 3
2 4

Write an APL function, **`DoubleDegrees`**, where:

- the right argument is a 2-column integer matrix representing a graph in edge list format.
- returns an integer vector result where the **i^{th}** element indicates the sum of the degrees of the neighbors of vertex **i**.

Example (using the above data):

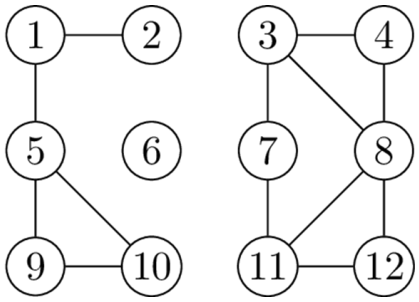
```
      DoubleDegrees 5 2 ⍴ 5 4 1 2 2 3 4 3 2 4
3 5 5 5 0
```

Medium Problem Set 1, Problem 3* – *Connected Components

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

An undirected graph is *connected* if there is a path between any pair of vertices. The graph of the following figure is not connected because, for instance, there is no path from 1 to 12. However, it does have three disjoint connected regions, corresponding to the following sets of vertices:

{1,2,5,9,10}{3,4,7,8,11,12}{6}



These regions are called *connected components*: each of them is a subgraph that is internally connected but has no edges to the remaining vertices.

The above graph's representation in edge list format is:

```
12 13 – 12 vertices, 13 edges
 1  2
 1  5
 5  9
 5 10
 9 10
 3  4
 3  7
 3  8
 4  8
 7 11
 8 11
11 12
 8 12
```

While the problem description on rosalind.info specifies to use a depth-first search, you can use any technique, including array-oriented approaches, to solve this problem.

Write an APL operator, **`Components`**, where:

- the right argument is a 2-column integer matrix representing a graph in edge list format.
- returns a singleton integer indicating the number of connected components in the graph.

Example (using the above data):

```
      Components 14 2 ⍴ 12 13 1 2 1 5 5 9 5 10 9 10 3 4 3 7 3 8 4 8 7 11 8 11 11 12 8 12
3
```

Medium Problem Set 2 – Mathematics – Numeric Integration

The definite integral of a real valued function can be interpreted as the area under its graph over some interval (unless the function is negative or the endpoints are flipped but let's not get into that).

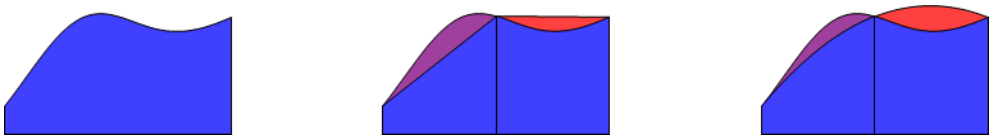


Figure 3

The same integral (left) approximated using the trapezoid (middle) and Simpson's rule (right).
Red is overestimate and purple is underestimate.

Contrary to what introductory courses in calculus might lead you to believe, symbolic integration is not in general feasible. The function you want to integrate might not have an antiderivative in closed form (expressed in terms of "standard" mathematical functions; and even if it does it might be too hard to find), or the function itself might not be given in closed form, but rather as the result of some measurement, simulation, or something similar.

In such cases, numerical methods must be employed. There are several such methods, three of which we will implement in this problem set as APL user-defined operators.

Medium Problem Set 2, Problem 1 – Trapezoid Rule

In the trapezoid rule, the integral of a function f over an interval [a,b] is estimated by dividing [a,b] into n sub-intervals of size $\Delta x = (b - a)/n$, and approximating f by a straight line within each (see figure 3). This means that f only needs to be evaluated in the n + 1 points $x_i = a + i\Delta x$. Putting it all together we get:

$$T_n = \frac{\Delta x}{2}(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n))$$

Write an APL operator, **`Trapezoid`**, that:

- takes a left operand which is a scalar function.
- takes a positive integer left argument which is the number of subintervals.
- takes a 2-element numeric vector right argument which represents an interval [a,b] where a < b.
- returns ***T_n*** for the given function and interval.

Example

```
      1 ⍟Trapezoid 1,*1
0.8591409142

      (⍳4) ⍟Trapezoid ¨ ⊂1,*1
0.8591409142 0.9623362015 0.9829803154 0.9903650088
```

Medium Problem Set 2, Problem 2* – *Simpson's Rule

Using Simpson's rule the interval is similarly divided but, instead of approximating f by a straight line, the sub-intervals are paired up and f is approximated by a parabola (see figure 3). In general, this reduces the error but leads to the slightly more involved formula:

$$S_n = \frac{\Delta x}{3}\big(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n)\big).$$

Write an APL operator, **`Simpson`**, that:

- takes a left operand which is a scalar function.
- takes an even, positive, integer left argument which is the number of subintervals.
- takes a 2-element numeric vector right argument which represents an interval [a,b] where a < b.
- returns ***Sₙ*** for the given function and interval.

Example

```
      2 ⍟Simpson 1,*1
0.9967346307

      (2×⍳4) ⍟Simpson¨ ⊂1,*1
0.9967346307 0.9997079446 0.999936071 0.9999788955
```

Medium Problem Set 2, Problem 3* – *Romberg's Method

Romberg's method generalizes the Trapezoid and Simpson's rule. As it turns out, given that f has enough continuous derivatives, by using Taylor's formula, the error of the Trapezoid rule can be expressed in terms of these. Then, using a technique known as Richardson extrapolation one can combine approximations using different numbers of subintervals to cancel out term after term of the error. Glossing over a ton of (really cool!) detail we can define the Romberg method using the following recurrence:

$$R_n^0 = T_{2^n}$$

$$R_n^m = \frac{1}{4^m - 1}(4^m R_n^{m-1} - R_{n-1}^{m-1})$$

Write an APL operator, **`Romberg`**, that:

- takes a left operand which is a scalar function.
- takes an integer left argument greater than or equal to 0 representing n.
- takes a 2-element numeric vector right argument which represents an interval [a,b] where a < b.
- returns $\boldsymbol{R_n^n}$ for the given function and interval.

Solutions that perform no unnecessary computations of f or R will be judged more favorably. This means that f should be computed at most once in each point, and R_n^m should be computed at most once for each m and n.

Example

```
      (¯1+⍳4)⍟Romberg¨⊂1,*1 ⍝ Recognize the first two values?
0.8591409142 0.9967346307 0.9999061655 0.9999984001
```

Medium Problem Set 3 – Data Science

According to Wikipedia, data science is a multi-disciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from structured and unstructured data. Data science is the same concept as data mining and big data: use the most powerful hardware, the most powerful programming systems, and the most efficient algorithms to solve problems.

APL's expressiveness and power makes it well-suited for data science; it helps the data scientist to focus more on the data than the mechanics of implementing solutions on a computer.

In this problem set we'll be looking at a variety of metrics for all countries of the world and doing some analysis. The data has been downloaded from the CIA World Factbook. The files are located in the **/datascience/** folder in the **Contest2019.zip** file and are named as follows:

deathrate.txt – annual death rate per 1000 people	**internet.txt** – internet users
education.txt – education expenditures (as a % of GDP)	**life.txt** – life expectancy at birth
gdp.txt – gross domestic product	**obesity.txt** – adult prevalence rate
health.txt – health expenditures (as a % of GDP)	**population.txt** – country population

In data science, not all source data is clean and consistent; this problem set presents some of those aspects. For instance, there is some inconsistency in population.txt file in the last column of data:

```
231     Christmas Island                          2,205       July 2016 est.
232     Norfolk Island                            1,748       2016 est.
233     Niue                                      1,618
234     Paracel Islands                           1,440       2014 est.
```

Not all metrics have the same number of countries; for example, **population.txt** has 238 countries while **education.txt** has only 175. In problem 2, you'll be given a way to represent missing data.

If you are using Dyalog APL to solve these problems then you can easily read in the files using

```
      data←↑⊃⎕NGET <filename> 1
```

For example:

```
      data←↑⊃⎕NGET 'c:/Contest2019/datascience/population.txt' 1
```

You can extract the filename (without the extension) using

```
      2⊃⎕NPARTS 'c:/contest2019/datascience/population.txt'
population
```

If you're using some other APL system, refer to their documentation for reading and manipulating files.

Medium Problem Set 3, Problem 1 – Do you get it?

In this problem you'll write a function to read and parse CIA World Factbook ranking files. Each file has a format similar to the following extraction from the **population.txt** file. First there's a title row indicating the metric for the file: 'Country Comparison :: ' followed by the metric name. Then a blank line followed by one line for each country, with space-separated data of: [1] rank, [2] country name, [3] metric value and, optionally, [4] the date of the metric for that country. For these problems we'll ignore column [4] and assume that all the data is current. The first several lines of **population.txt** are:

```
Country Comparison :: Population

1       China                                1,384,688,986       July 2018 est.
2       India                                1,296,834,042       July 2018 est.
3       United States                          329,256,465       July 2018 est.
4       Indonesia                              262,787,403       July 2018 est.
```

Write an APL function, **`ReadCIA`**, that:

- takes a character vector right argument which is the name of the file to be read.
- returns a two-column matrix where the first column is the country name and the second column is the numeric metric. The first row should contain the word 'country' and the name of the file (without the extension).

Your solution should not assume that the data elements always begin in specific columns (even though in this case, they do), but may assume that at least two spaces (' ') will separate data elements within a row.

Example

```
      5↑ReadCIA 'internet.txt'     ⍝ displayed with ]Boxing on
```

country	internet
China	730723960
India	374328160
United States	246809221
Brazil	122841218
Japan	116565962

```
      7↑ReadCIA 'life.txt'     ⍝ displayed with ]Boxing off
country     life
Monaco      89.4
Japan       85.5
Singapore   85.5
Macau       84.6
San Marino  83.4
Hong Kong   83.1
Iceland     83.1
```

Medium Problem Set 3, Problem 2 – Merge It

Now that we have a way, from the first problem, to read in individual metrics, it's time to combine them into a single matrix. Not all countries are found in all of the metrics. As such, we need a way to indicate missing data. We can't use a number to represent missing data because that number could conceivably be a valid value for some metric, so we'll use ⊂⍬ in cells that don't have data for the metric in that column.

Write an APL function, **Merge**, that:

- takes a left argument which is a matrix as returned by **ReadCIA**.
- takes a right argument matrix where column 1 contains country names and the remaining columns contain metric data or ⊂⍬ if the country has no data for that metric.
- returns a matrix that appends the metric column of the right argument to the left argument, adding new rows as needed for countries that are in the right argument but not the left argument.

Example

(your path may be different depending on where you unzip the **Contest2019.zip** file)

```
      education ← ReadCIA '/contest2019/datascience/education.txt'
      health ← ReadCIA '/contest2019/datascience/health.txt'
      ≢¨ education health
176 193
      ≢ health Merge education
205
      5↑ health Merge education    ⍝ displayed with ]Boxing on
```

country	education	health
Cuba	12.8	11.1
Micronesia, Federated States of	12.5	13.7
Grenada	10.3	6.1
Solomon Islands	9.9	5.1

If we used **ReadCIA** to read each of the files into individual variables, we could use reduction to combine all the metrics in a single statement:

```
      5↑ All ←⊃Merge/ deathrate education gdp health internet life obesity population
```

country	population	obesity	life	internet	health	gdp	education	deathrate
China	1384688986	6.2	75.8	730723960	5.5	16700		8
India	1296834042	3.9	69.1	374328160	4.7	7200	3.8	7.3
United States	329256465	36.2	80.1	246809221	17.1	59800	5	8.2
Indonesia	262787403	6.9	73.2	65525226	2.8	12400	3.6	6.5

We'll use this table of all metrics (All) in the next problem.

Medium Problem Set 3, Problem 3 – What's the Correlation?

Now that we have a dataset we can inspect, is there any correlation between any of the metrics? To accomplish this, we'll use the population Pearson correlation coefficient, commonly represented by the Greek letter ρ (rho). It has a value between +1 and −1, where 1 is total positive linear correlation, 0 is no linear correlation, and −1 is total negative linear correlation. It is widely used in the sciences. One definition is

$$\rho_{X,Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

where:

- E is the expectation
- μ_X is the mean of X
- μ_Y is the mean of Y
- σX is the standard deviation of X
- σY is the standard deviation of Y

Write an APL function, **Pearson**, and an APL operator, **Analyze**, where:

- **Pearson**
  - takes right and left arguments of numeric vectors of equal length representing two variables.
  - returns a single number representing the population Pearson correlation coefficient for the variables.
- **Analyze**
  - takes a left operand which is the matrix of all metrics from Problem 2.
  - takes right and left arguments of the metric names to compare.
  - returns a single number representing the population Pearson correlation coefficient for the specified metrics.
  - should filter from both arguments any elements where either or both arguments have a ⊂⍬ value.

Example

```
      Pearson⍨⍳10                 ⍝ very high correlation of a variable to itself
1
      (⍱⍳10) Pearson ⍳10          ⍝ very high negative correlation
¯1
      (100?100) Pearson (100?100) ⍝ expected low correlation between 2 random vars
¯0.006180618062

      'population' (All Analyze) 'internet' ⍝ All is from Problem 2
0.9414070586

      ∘.(All Analyze)⍨1↓,1↑All ⍝ outer product (formatted to fit)
 1          ¯0.051737 ¯0.066879   0.18784    0.034547  ¯0.23227  ¯0.20465   0.0073577
¯0.051737    1        ¯0.056299   0.40642   ¯0.027905   0.16243   0.33977  ¯0.075523
¯0.066879   ¯0.056299  1           0.11477    0.0051597  0.65697   0.24982  ¯0.064319
 0.18784     0.40642   0.11477     1          0.043422   0.24202   0.29493  ¯0.053967
 0.034547   ¯0.027905  0.0051597   0.043422   1          0.060386 ¯0.10777   0.94141
¯0.23227     0.16243   0.65697     0.24202    0.060386   1         0.45497  ¯0.029846
¯0.20465     0.33977   0.24982     0.29493   ¯0.10777    0.45497   1        ¯0.1735
 0.0073577  ¯0.075523 ¯0.064319  ¯0.053967   0.94141   ¯0.029846 ¯0.1735    1
```

Difficult Problem Sets

Difficult Problem Set 1 – Boggle Your Mind

Boggle is a word game played with dice, first released by Parker Brothers (now Hasbro) in 1972. 16 dice with letters printed on the sides are rolled and placed in a 4×4 grid. Players are tasked with finding as many and as long words as possible, snaking around the grid in a given time frame. More specifically, a word can start anywhere, and proceeds to any of its neighbors, horizontally, vertically or diagonally. A word cannot revisit the same die twice. However, each word is independent and there is no restriction with regards to reuse of die faces between different words.

In this problem we generalize on the standard game in a number of ways:

- The board can be any size (but must still be rectangular).
- A die can have any number of sides greater than or equal to one. A game can include dice with different numbers of sides.
- Dice can have anything printed on their sides, not necessarily a single letter. Boggle itself does this by including the two-letter combination “QU” on one of its dice.

Difficult Problem Set 1, Problem 1 ***–Well, Shake It Up Baby Now*** *(with due apologies to the Beatles)*

Write an APL function, **`MakeBoard`**, that will create an initial, random, board. **`MakeBoard`**:

- takes a 2-element integer left argument that represents the dimensions of the board.
- takes a right argument of die definitions, one for each cell on the board. Any die can end up anywhere on the board.
- returns a matrix with shape as specified in the left argument where each cell has one of the randomly-generated die faces.

The Phase II template namespace contains a niladic function, **`BoggleDice`**, that returns the definitions of dice for the original Boggle game.

Example

```
      2 2 MakeBoard 'AB' 'CDE' 'FGHI' 'JKLMN' ⍝ 2×2 board with 2,3,4 and 5-sided dice
JD
AG
      BoggleDice
```

AACIOT	ABILTY	A B J M O QU	ACDEMP	ACELRS	ADENVZ	AHMORS	BIFORX	DENOSW	DKNOTU	EEFHIY	EGKLUY	EGINTV	EHINPS	ELPSTU	GILRUW

```
      4 4 MakeBoard BoggleDice
PIMO
SOEA
IYAF
AEUI
      4 4 MakeBoard BoggleDice    ⍝ if QU comes up, the result could look like this
```

T	P	QU	A
S	L	G	I
R	U	T	E
I	I	N	A

Difficult Problem Set 1, Problem 2* – *Words, Words, and More Words

The **Contest2019.zip** file includes **sowpods.txt** in the **/boggle/** folder – SOWPODS is the word list used in English-language tournament Scrabble in most countries. The file mostly contains one word on each line, but you'll need to filter out lines that are empty or contain more than one word.

Write an APL function, **`GetWordList`**, that:

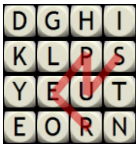
- takes a character vector right argument containing the name of the word list file to read.
- converts all the words to uppercase (you can use **`1(819⌶)`** to convert to uppercase).
- filters out empty lines and lines that contain more than one word.
- returns a vector of words (character vectors) read from the file.

Example

```
      ≠SOWPODS←GetWordList '/Contest2019/boggle/sowpods.txt'
267752
      SOWPODS[7?≠SOWPODS] ⍝ 7 random words
```

RETIE	PREPOSING	SMASHUP	TELEFILM	BOODLES	RECALLING	WILDER

Difficult Problem Set 1, Problem 3 – Boggle Minding or Mind Boggling?



Now we get into the mechanics of playing Boggle. In the image to the left, the word "SUPER" is created. In this problem, we'll find all the words of any length that can be created from a random board.

Write an APL function, **`FindWords`**, that:

- takes a right argument of a board as generated by **`MakeBoard`** from Problem 1.
- takes a left argument of a list of words as generated by **`GetWordList`** from Problem 2.
- returns a vector of all possible unique words (character vectors) from the left argument that can be created from the board in the right argument. The found words can be returned in any order.

Example

```
      SOWPODS FindWords ⍞←2 2⍴'TH' 'R' 'OU' 'GH'
```

TH	R
OU	GH

OU	OUR	THOU	ROUGH	ROUTH	THOUGH	THROUGH

```
      ⊢board←4 4 MakeBoard dice
```

T	P	QU	A
S	L	G	I
R	U	T	E
I	I	N	A

```
      ≠found←SOWPODS FindWords board
188
      found[10↑⍒≠¨found]
```

ANTIRUST	QUIETUS	GLUTAEI	RUINATE	URINATE	GENIUS	GLUTEI	GLUTEN	GUINEA	GUNITE

Difficult Problem Set 1, Problem 4* – *Making a Point

Characters	Score
Less than 3	0
3 or 4	1
5	2
6	3
7	5
8 or more	11

The game of Boggle uses a scoring system based on the length of each word found as shown in the table above.

Write an APL function, **`BoggleScore`**, that:

- takes a right argument of a vector of words (character vectors).
- returns a 2-element integer vector where the first element represents the total score for the words based on the table above and the second element is a count of the words that scored more than 0 (zero) points.

Example

```
      BoggleScore found      ⍝ the variable found is from the previous problem
240 164

      ⍝ putting it all together
      BoggleScore (GetWordList 'sowpods.txt') FindWords ⎕←4 4 MakeBoard BoggleDice
NGET
RINR
KLCC
PTHM
152 98
```

Difficult Problem Set 2 – How About A Nice Game of Chess?

Chess is a two-player strategy board game played on a checkered board with 64 squares arranged in an 8×8 grid. The game is played by millions of people worldwide. This problem set does not require a knowledge of chess strategy, just a basic understanding of how the pieces move.

From the movie "WarGames"

Initial Position

The co-ordinates on a chess board consist of ranks (rows) 8-1 from the top down and files (columns) a-h from left to right. Each square is identified by its file and rank, for example, `'e4'`.

Forsyth-Edwards Notation (FEN) is a standard notation for describing a particular board position of a chess game. It describes a complete chess board state in a single line using only ASCII characters. Portable Game Notation (PGN) is a plain text format for recording chess games and is the most common format used by chess programs. PGN moves are formatted using Standard Algebraic Notation (SAN).

Resources/References

If you're unfamiliar with chess, or need a refresher	https://en.wikipedia.org/wiki/Rules_of_chess#Movement
Forsyth-Edwards Notation	https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation
Portable Game Notation	https://en.wikipedia.org/wiki/Portable_Game_Notation#Movetext
Standard Algebraic Notation	https://en.wikipedia.org/wiki/Algebraic_notation_(chess)
Online tool to verify your work; copy and paste PGN and FEN to test and visualize your results	https://www.chess.com/analysis

A FEN "record" consists of 6 fields separated by a space. Our problems will use only the first 4:

1. Piece placement (from White's perspective). Each rank is described, starting with rank 8 and ending with rank 1. Ranks are separated by `'/'`. Within each rank, files "a-h" are described using the letters `'KQRBNP'` respectively for King, Queen, Rook, Bishop, Knight, Pawn. White pieces use uppercase `'KQRBNP'` and black pieces used lowercase `'kqrbnp'`. Digits `'12345678'` are used to indicate the number of consecutive empty squares within a rank.
2. Active color. `'w'` means White moves next, `'b'` means Black moves next.
3. Castling availability. If neither side can castle, this is `'-'`. Otherwise, this has one or more letters: `'K'` (White can castle kingside), `'Q'` (White can castle queenside), `'k'` (Black can castle kingside), and/or `'q'` (Black can castle queenside).
4. En passant target square. If there's no en passant target square, this is `'-'`. If a pawn has just made a two-square move, this is the position "behind" the pawn.

The FEN for the initial board (above) is:

```
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -'
```

Difficult Problem Set 2, Problem 1* – *Imagine the Possibilities

Write an APL function, **`AllMoves`**, which

- takes a character vector right argument representing a chess board state in FEN.
  As only the first four FEN fields are pertinent for this problem, fields 5 and 6 are optional.
- returns a vector of character vectors representing the set of all legal moves, in SAN, from that board state.
  The order of the moves is not significant.

Example

```
      AllMoves 'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -' ⍝ initial board
```

a3	a4	b3	b4	c3	c4	d3	d4	e3	e4	f3	f4	g3	g4	h3	h4	Na3	Nc3	Nf3	Nh3

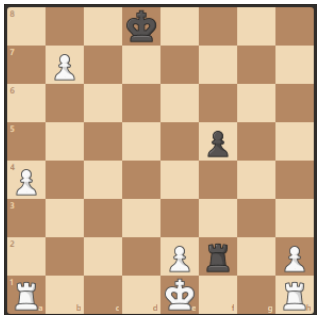
The next two examples use the board found below

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R w KQ -' ⍝ White to move
```

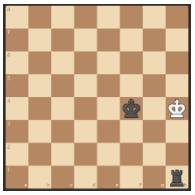
Ra2	Ra3	Rb1	Rc1	Rd1+	Kd1	Kd2	Kxf2	O-O-O+	Rf1	Rg1	a5	b8=Q+	b8=R+	b8=B	b8=N	e3	e4	h3	h4

```
      AllMoves '3k4/1P6/8/5p2/P7/8/4Pr1P/R3K2R b KQ -' ⍝ Black to move
```

Kc7	Kd7	Ke7	Ke8	f4	Rxe2+	Rg2	Rxh2	Rf1+	Rf3	Rf4



Difficult Problem Set 2, Problem 2 – ***Checkmate!***



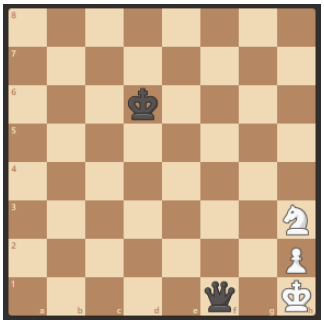
One of the ways a chess game can end is by checkmate, when a player's king is in check (under attack) and there is no legal move to get the king out of check (by moving the king to a cell that is not under attack, capturing the attacking piece, or moving a piece between the king and the attacking piece).

Write an APL function, `Checkmate`, that:

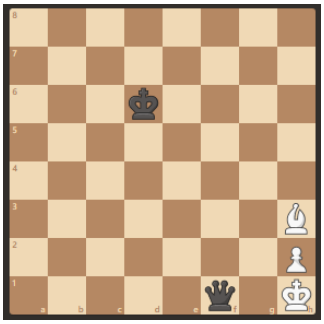
- takes a character vector representing a chess board state in FEN.
  As only the first four FEN fields are pertinent for this problem, fields 5 and 6 are optional.
- returns a Boolean indicating whether the active player's king is checkmated (1) or not (0).

Example

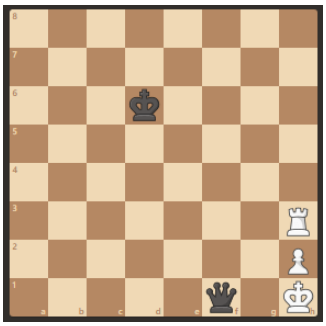
```
      Checkmate '8/8/3k4/8/8/7N/7P/5q1K w - -'
0
```



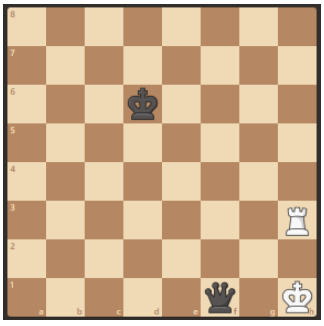
```
      Checkmate '8/8/3k4/8/8/7B/7P/5q1K w - -'
0
```



```
      Checkmate '8/8/3k4/8/8/7R/7P/5q1K w - -'
1
```



```
      Checkmate '8/8/3k4/8/8/7R/8/5q1K w - -'
0
```



Difficult Problem Set 2, Problem 3 – Moving Along

PGN is a common format for storing chess games; PGN files typically have a ".pgn" extension. We have included two PGN files, **game1.pgn** and **game2.pgn**, in the **/chess/** folder of the **Contest2019.zip** file for your use during development. PGN has two sections, tags followed by movetext.

Tags contain information about the game, its players, their ratings, the result of the game, and so on. Tags are name/value pairs enclosed in brackets []. There is one tag for each line. Movetext is essentially any non-blank line following the tags.

Moves are in the format:	`nn. www bbb`
where:	`nn.` is the move number followed by a period '.' and optionally a space
	`www` is the white move in SAN followed by a space
	`bbb` is the black move in SAN followed by a space

Moves are delimited by spaces. Movetext terminates with an annotation indicating the result of the game – '1-0' (white won), '0-1' (black won), '1/2-1/2' (draw), or '*' (ongoing).

SAN has features to further annotate moves with commentary and other annotations, but for the purposes of this problem you can assume none of that exists in the files you'll be processing. A sample PGN file looks like:

```
[Event "World Blitz 2018"]
[Site "St Petersburg RUS"]
[Date "2018.12.29"]
[Round "8.1"]
[White "Carlsen,M"]
[Black "Firouzja,Alireza"]
[Result "1-0"]
[WhiteElo "2835"]
[BlackElo "2607"]
[ECO "E62"]

1.c4 g6 2.g3 Bg7 3.Bg2 Nf6 4.Nc3 O-O 5.Nf3 d6 6.d4 Nc6 7.O-O e5 8.dxe5 dxe5
9.Bg5 Be6 10.Qa4 Qb8 11.Bxf6 Bxf6 12.Nd2 Bg7 13.Rfd1 Rd8 14.Nb3 Nd4 15.Nc5 c6
16.e3 Bg4 17.exd4 exd4 18.N3e4 Bxd1 19.Rxd1 Qc7 20.Qb3 b6 21.Nd3 Rab8 22.c5 bxc5
23.Qc4 Qa5 24.Nexc5 Bf8 25.b4 Qa3 26.Ne5 Qxb4 27.Qxf7+ Kh8 28.Qf6+ Bg7 29.Qg5 Re8
30.Nxc6 Qc4 31.Nxb8 Qe2 32.Qd2  1-0
```

Write an APL function, **`PositionAfter`**, that:

- takes a character vector right argument in the form nnC where nn is a move number and C indicates the color ('W' for white, 'B' for black).
- takes a character vector of a game in PGN format as the left argument. Tags are optional and can be ignored; only the movetext is significant for this problem.
- returns a character vector that represents the game board position in FEN after the move indicated in the right argument.

Example

```
      game1←⊃⎕NGET '/Contest2019/Data/Chess/game1.pgn' ⍝ your path may be different
      game1 PositionAfter '6B'
r1bq1rk1/ppp1ppbp/2np1np1/8/2PP4/2N2NP1/PP2PPBP/R1BQK2R w KQ -

      game2←⊃⎕NGET '/Contest2019/Data/Chess/game2.pgn' ⍝ your path may be different
      game2 PositionAfter '21B'
r4rk1/1b5p/p3p1p1/1p1q1p2/3n1P2/PP4N1/2P3PP/B1qQRBK1 w - -
```

Difficult Problem Set 3 – Do The Twist

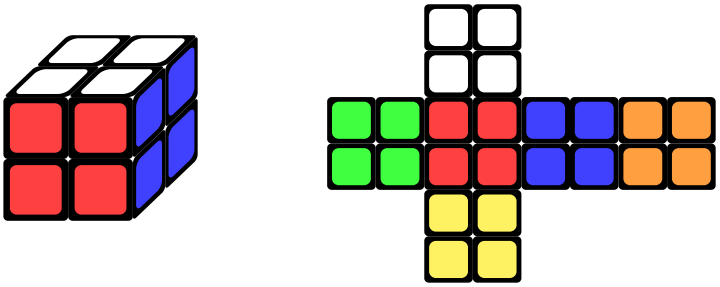


Figure 4 – A folded and a sorted Pocket Cube

The Pocket Cube is the 2 × 2 × 2 version of Rubik's Cube. While it is widely reported that Ernő Rubik designed the cube to teach his students to think in three dimensions, that is not actually the case. Nevertheless, this problem is designed with exactly that purpose in mind.[1]

Difficult Problem Set 3, Problem 1 – Construction and Unfolding

Define a variable **`Cube`** representing a sorted Pocket Cube in an internal representation of your choosing. For a cube to be sorted all stickers on each face must be of the same color. It does not matter how the cube is rotated, but the pairs of colors red-orange, white-yellow and blue-green must be opposite each other, and furthermore the orientation of the cube (in the linear algebraic sense) must be correct – this can be checked by (for example) finding the red-blue-yellow corner and seeing if it has those colors going around clockwise.

W - white	G - Green
R - Red	B - Blue
O - Orange	Y - Yellow

Table 2 - Face Colors Key

Write an APL function, **`Unfold`**, that:

- takes a right argument of your internal representation of a Pocket Cube.
- returns a character matrix representing your cube unfolded, using table 2 above to denote the face colors.

Note that while this task might in principle be solved by defining **`Cube`** to be anything and making **`Unfold`** a constant function, this will not be accepted as it won't work for the next task.

Example

```
      Unfold Cube
  WW
  WW
GGRRBBOO
GGRRBBOO
  YY
  YY
```

[1] If you have a Rubik's Cube, you can simulate a Pocket Cube by disregarding the middle layers and only looking at the corners.

Difficult Problem Set 3, Problem 2 – Twist and Shout

F Front face	**B** Back face	**z** Whole cube from front
U Top face	**D** Bottom face	**y** Whole cube from top
R Right face	**L** Left face	**x** Whole cube from right

Table 3 – Singmaster Notation

In the speedcubing community, specific sequences of twists (known as algorithms) are often described succinctly using Singmaster notation (after David Singmaster). Single letters denote clockwise twists of the different faces or the whole cube, as shown in table 3, for example, **yF** would mean "rotate the whole cube to the left and twist the front face clockwise". Letters followed by prime symbols denote counterclockwise twists, so that a successive **F'** would restore the cube to its previous configuration (although still rotated). To get an interactive visualization of Singmaster Notation, see https://ruwix.com/the-rubiks-cube/notation/advanced/.

Since prime symbols might be hard to type we use single quotes instead; don't forget that APL translates two single quotes within a character vector to be a single quote, for example, **'F'''** results in **F'**.

Write an APL function, **Twist**, that:

- takes a right argument representing a Pocket Cube in your internal representation.
- takes a character vector left argument representing an algorithm expressed in Singmaster notation.
- returns the transformed cube in your internal representation.

Once you're written **Twist**, you can experiment…

```
      moves←,'FBzUDyRLx'∘.,'' ''''              ⍝ all possible moves
      c←(∈(⊂?10⍴≢moves)⌷moves) Twist Cube       ⍝ randomize the starting Cube

⍝ The next statement allows you to play interactively with the cube created above
⍝   type an algorithm and press enter (double '' not needed)
⍝   press enter without typing an algorithm to exit

      {}{⍞←Unfold ⍵ ⋄ ⍞ Twist ⍵}⍣≡c
```

Example

```
      Cube≡'yFy''R''' Twist Cube
1
      {Unfold ⍵ Twist Cube}¨'F' 'yFR''UBR'
```

```
┌────────┬────────┐
│  WW    │   GO   │
│  GG    │   GR   │
│GYRRWBOO│WWOBWYGR│
│GYRRWBOO│RYBRYBWY│
│  BB    │   OG   │
│  YY    │   BO   │
└────────┴────────┘
```

Difficult Problem Set 3, Problem 3 – Permutation Order

Since the Pocket Cube has a finite number of configurations (about 3.7 million), any algorithm, if performed repeatedly, will get you back to where you started.

Write an APL function, **`Order`**, that:

- takes a character vector right argument representing an algorithm in Singmaster notation.
- takes a left argument of an initial Cube configuration in your internal representation.
- returns a single integer that represents the fewest (>0) number of times the algorithm needs to be repeated to return the Cube to its initial configuration.

Example

```
      Cube Order 'R'
4
      Cube∘Order ¨ 'RL''' 'RU' 'RU''R''U'
1 15 6
```