2014 APL Problem Solving Competition – Phase II Problem Descriptions

The Phase II problems are divided into three sets representing three general categories – Bioinformatics, Cryptography and Recreation and Games. Each set comprises three problems, one each of low, medium and high difficulty levels, with each problem broken down into multiple tasks. You do not need to solve all the Phase II problems to be considered eligible for the top three prizes but you do need to complete, at a minimum, one problem of each level of difficulty. The problems can be from different sets. Higher difficulty problems may be substituted for lower difficulty problems – for instance, you could solve two medium difficulty problems and one high difficulty problem.

You can also solve more than the minimum number of problems; you can solve all the problems if you choose to do so. Doing more work by solving more problems can work in your favour; if entries from two people are of similar quality but one has solved more or higher difficulty problems, then that entry will receive higher consideration by the judging committee.

Good Luck and Happy Problem Solving!

Description of the Contest2014 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 `Contest2014.DWS`

The Contest2014 workspace contains:

- **`#.Problems`** – a namespace in which you will develop your solutions and which itself contains:
  - Stubs for all of the functions described in the problem descriptions. The function stubs are implemented as traditional APL functions (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.

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 **Contest2014.dyalog** which will contain any code or data you placed in the **`#.Problems`** namespace. You will then need to upload the **Contest2014.dyalog** file using the contest website.

If you use some other APL system, you can use the template script file `Contest2014.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.

Bioinformatics Problem Sets

The Bioinformatics problems presented here are based on problems presented on the website http://rosalind.info. The descriptions have been adapted to be more suitable for APL syntax and this competition.

Bioinformatics Problem 1 (low difficulty)

The description of this problem on Rosalind.info can be found at http://rosalind.info/problems/1a/ .

A *k*-mer is defined as a string of length *k*.

Task 1- k-mer Counting

Write an APL function named `Count` which

- takes a character vector left argument representing of a string of text
- takes a character vector right argument representing a *k*-mer pattern.
- returns the number of times that the *k*-mer pattern appears as a substring of the text.

Example:

```
     'ACAACTATGCATACTATCGGGAACTATCCT' Count 'ACTAT'
3
```

Note that:

```
    'CGATATATCCATAG' Count 'ATA'
3
```

returns 3 (not 2) since we should account for overlapping occurrences of the pattern in the text.

Task 2 – Most Frequent k-mers

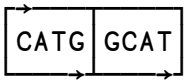
We say that *Pattern* is a **most frequent *k*-mer** in the text if it maximises Text Count Pattern among all *k*-mers. For example, "ACTAT" is the most frequent 5-mer in "ACAACTATGCATCACTATCGGGAACTATCCT" and "ATA" is the most frequent 3-mer of "CGATATATCCATAG".

Write an APL function called `MostFrequent` which

- takes character vector left argument representing a string of text
- takes an integer, k, representing the length of *k*-mer to find
- returns all most frequent *k*-mers in the text as a vector of character vectors

Example:

```
    ]disp 'ACGTTGCATGTCGCATGATGCATGAGAGCT' MostFrequent 4
```



NOTE: `]disp` is a user command supplied with Dyalog APL to aid the user by displaying structural information about an array.

Bioinformatics Problem 2 (medium difficulty)

Task 1 - Clump Finding

The description of this problem on Rosalind.info can be found at http://rosalind.info/problems/1d/ .

Given integers L and t, a string Pattern forms an (L, t)-clump inside a (larger) string Genome if there is an interval of Genome of length L in which Pattern appears at least t times. For example, TGCA forms a (25, 3)-clump in the following Genome: gatcagcataagggtcccTGCAaTGCAtgacaagccTGCAgttgttttac.

Write an APL function named `FindClumps` which

- takes a character vector left argument representing Genome
- takes a 3 element integer right argument representing k, L, and t
- returns a vector of character vectors of all distinct *k*-mers forming (L, t)-clumps in Genome

Example: Given

```
      Genome←'CGGACTCGACAGATGTGAAGAAATGTGAAGACTGAGTGAAGAGAAGAGGAAACACGACA
              CGACATTGCGACATAATGTACGAATGTAATGTGCCTATGGC'

      ]disp Genome FindClumps 5 75 4
```

```
┌→──────────────┐
│CGACA│GAAGA│AATGT│
└~~~~~┴~~~~~┴~~~~~┘
```

Task 2 – Approximate Pattern Matching

The description of this problem on Rosalind.info can be found at http://rosalind.info/problems/1f/ .

We say that position *i* in *k*-mers $p_1 \ldots p_k$ and $q_1 \ldots q_k$ is a **mismatch** if $p_i \neq q_i$. For example, CG**A**A**T** and CG**G**A**C** have two mismatches.

Find all approximate occurrences of a pattern in a string.

Write an APL operator named `ApproxMatch` which

- takes an integer left function modifier representing d, the maximum allowed mismatches
- takes a character vector left argument representing Text
- takes a character vector right argument representing the Pattern to match
- returns a vector of offsets representing all starting positions where Pattern appears as a substring of Text with at most d mismatches.

Example: Given

```
      Text←'CGCCCGAATCCAGAACGCATTCCCATATTTCGGGACCACTGGCCTCCACGGTACGGACGTCAAT
            CAAATGCCTAGCGGCTTGTGGTTTCTCCTACGCTCC'
      Pattern←'ATTCTGGA'

      Text (3 ApproxMatch) Pattern
6 7 26 27 78
```

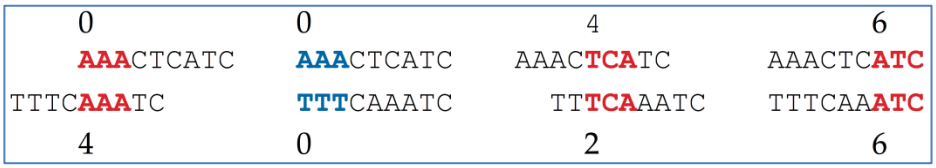
Bioinformatics Problem 3 (high difficulty)

Task 1 – Shared k-mers

The description of this problem on Rosalind.info can be found at http://rosalind.info/problems/6d/ .

We say that a *k*-mer is shared by two genomes if either the *k*-mer or its *reverse complement* appears in each genome. The reverse complement of a DNA string *s* is formed by reversing the symbols of s, then taking the complement of each symbol (for example, the reverse complement of "GTCA" is "TGAC"). A shared *k*-mer can be represented by an ordered pair (x, y), where x is the starting position of the *k*-mer in the first genome and y is the starting position of the *k*-mer in the second genome.

In the figure below, for the genomes "AAACTCATC" and "TTTCAAATC", these shared *k*-mers are (0, 4), (0, 0), (4, 2), and (6, 6). The second pair is shown in blue because they are reverse complementary[1].

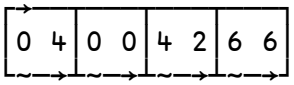


Write an APL operator named `SharedKmers` which

- takes an integer left function modifier representing k
- takes a character vector left argument representing Genome1
- takes a character vector right argument representing Genome2
- returns a vector of pairs of origin-0 indices representing the shared *k*-mer starting positions in Genome1 and Genome2

Example:

```
      ]disp 'AAACTCATC ' (3 SharedKmers) 'TTTCAAATC'
```



[1] See http://rosalind.info/problems/1b/ for the definition of reverse complementary

Task 2- Longest Shared Substring

NOTE: This task description has been updated as of 6 June 2014. The answer in the original example as copied from http://rosalind.info/problems/7e/ was incomplete. A corrected example is presented below.

The description of this problem on Rosalind.info can be found at http://rosalind.info/problems/7e/ .

Write an APL function named `LongestShared` which

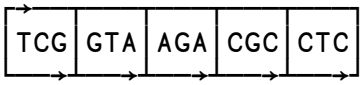
- takes a character vector left argument representing Text1
- takes a character vector right argument representing Text2
- returns vector of character vectors representing the longest substring(s) that occur in both Text1 and Text2

Original Example: (incorrect)

```
      'TCGGTAGATTGCGCCCACTC' LongestShared 'AGGGGCTCGCAGTGTAAGAA'
AGA
```

Corrected Example:

```
      ]disp 'TCGGTAGATTGCGCCCACTC' LongestShared 'AGGGGCTCGCAGTGTAAGAA'
```



Task 3 – Shortest Non-Shared Substring

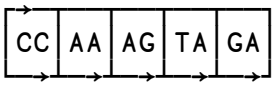
The description of this problem on Rosalind.info can be found at http://rosalind.info/problems/7f/ .

Write an APL function named `ShortestNonShared` which

- takes a character vector left argument representing Text1
- takes a character vector right argument representing Text2
- returns vector of character vectors representing the shortest substring(s) of Text1 that do not occur in Text2

Example:

```
      ]disp 'CCAAGCTGCTAGAGG' ShortestNonShared 'CATGCTGGGCTGGCT'
```



Task 4 – Edit Distance

The description of this problem on Rosalind.info can be found at http://rosalind.info/problems/5g/ and http://rosalind.info/problems/edit/ .

Given two strings *s* and *t* (of possibly different lengths), the **edit distance** *d*E(*s*,*t*) is the minimum number of **edit operations** needed to transform *s* into *t*, where an edit operation is defined as the substitution, insertion or deletion of a single symbol.

The latter two operations incorporate the case in which a contiguous interval is inserted into or deleted from a string; such an interval is called a **gap**. For the purposes of this problem, the insertion or deletion of a gap of length *k* still counts as *k* distinct edit operations.

Write an APL function named `EditDistance` which

- takes a character vector left argument representing s
- takes a character vector right argument representing t
- returns an integer representing the minimum number of edit operations needed to transform s into t

Example:

```
      'PLEASANTLY' EditDistance 'MEANLY'
5
```

Cryptography Problem Set

Cryptography Problem 1 (low difficulty) – Vigenère Cipher

The Vigenère cipher is a method of encrypting alphabetic text using a repeated keyphrase. It is a simple form of polyalphabetic substitution. Despite being easy to implement, for three centuries it resisted all attempts to break it.

To encrypt a piece of plaintext, a table of alphabets can be used, termed a *tabula recta*, *Vigenère square* or *Vigenère table*. It consists of the alphabet written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet. At different points in the encryption process, the cipher uses a different alphabet from one of the rows. The alphabet used at each point depends on a repeating keyphrase.

For example, suppose the message (M) to be encrypted is:

```
M: APLISFUNTOUSE
```

The sender of the message would choose a keyphrase (K), KEIVERSON in this case, and repeat it to the same length as the original message.

```
K: KEIVERSONKEIV
```

The message is encrypted by using each pair of letters in the keyphrase and original message as coordinates to look up the cipher letter in the Vigenère table. The lookup of the first three letters is shown in the Vigenère table below.

```
K: KEIVERSONKEIV
M: APLISFUNTOUSE
C: KTTDWWMBGYYAZ
```

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

Task 1 –Encryption

Write an APL function named VigEncrypt which

- takes a character vector left argument representing the keyphrase
- takes a character vector right argument representing the original message
- returns a character vector of the encrypted message

Example:

```
      'KEIVERSON' VigEncrypt 'APLISFUNTOUSE'
KTTDWWMBGYYAZ
```

Task 2 –Decryption

Write an APL function named VigDecrypt which

- takes a character vector left argument representing the keyphrase
- takes a character vector right argument representing the encrypted message
- returns a character vector of the original message

Example:

```
      'KEIVERSON' VigDecrypt 'KTTDWWMBGYYAZ'
APLISFUNTOUSE
```

Cryptography Problem 2 (medium difficulty) – Book Cipher Variation

A book cipher is a cipher in which the key is some aspect of a book or other piece of text. For instance, some book ciphers use a combination page number, paragraph and word position to indicate the words of the original text. Book ciphers need the sender and receiver to have identical copies of the book or text used to produce the cipher.

The cipher used for this problem uses positional information for individual letters rather than entire words.

Each letter in the original message is converted into a word position number and letter position from the beginning of the word. If the positions are limited to a maximum of the length of their corresponding word, someone could infer some information about the text used to generate the cipher. To counter this, this cipher allows for positions that extend beyond the end of the word.

For example, using the text
`THIS IS A TEST`
the letter "S" could be encoded as any of

Word	Positions(s)
1	4, 7, 13
2	2, 8
3	5
4	3

Task 1 – Let's Get Normal

The text of the Bill of Rights of the United States is included in the file **/Data/BillOfRights.txt** found in the competition zip file. The first task is to normalise the text into a space-delimited list of alphabetic words – changing any non-alphabetic characters into spaces, then removing multiple contiguous spaces and finally converting all letters to uppercase.

Write an APL function named `Normalise` which:

- takes a character vector right argument representing the name of the file
- reads the contents of the file (you may use `⎕SE.UnicodeFile.ReadText` for this)
- returns a character vector of the normalised text

Example: (your file location may be different)

```
      bor←Normalise 'c:/Contest2014/Data/BillOfRights.txt'
      70↑bor
THE PREAMBLE TO THE BILL OF RIGHTS CONGRESS OF THE UNITED STATES BEGUN
```

Task 2 – Encryption

Write an APL function named `BookEncrypt` which:

- takes a character vector left argument representing the normalised cipher text
- takes a character vector right argument representing the message text
- returns an integer vector result representing the encrypted message.

NOTE: The result will have twice as many elements as there are characters in the original message as there is a word number and offset for each character.

Your solution should attempt to minimise the number of duplicated pairs in the result. Do not use offsets greater than 20.

Example: (your results will most likely be different)

```
      cipher←bor BookEncrypt 'MYSECRETMESSAGE'
      cipher
404 13 344 19 713 17 345 6 203 10 67 10 695 4 45 5 404 9 276 19 176 8 662 5 392 19 250 5 691 10
```

Task 3 – Decryption

Write an APL function named `BookDecrypt` which:

- takes a character vector left argument representing the normalised cipher text
- takes an integer vector right argument representing the encrypted message text
- returns a character vector result representing the original message.

Example: (using `cipher` from Task 2)

```
      bor BookDecrypt cipher
MYSECRETMESSAGE
```

Cryptography Problem 3 (high difficulty) – Playfair Cipher

The Playfair cipher is a symmetric encryption technique that uses pairs of letters, also known as digraphs, instead of single letters. One advantage is that the use of digraphs makes frequency analysis much more difficult.

The Playfair cipher uses a 5×5 table containing a keyphrase. To generate the table, fill the spaces of the table with the letters of the keyphrase, removing duplicates. Then fill the rest of the table with the remaining letters of the alphabet in order. You can use any pattern to fill the table, left-to-right, top-to-bottom, in a spiral, etc. The keyphrase and the technique used to fill the table are all that's necessary to know. As this table only has 25 positions, treat the letters I and J interchangeably for the English alphabet.

This technique can apply to alphabets other than the English alphabet. The table formed need only be roughly square, e.g. the Danish alphabet, with 29 characters, could be represented as a 5×6 table using a space in the final position.

Task 1 – Squaring Off

Write an APL function named `PlayfairTable` which:

- takes a character vector right argument representing the keyphrase
- returns a 5×5 character matrix of the Playfair table
- optional: allow the function to take an integer left argument that indicates the fill technique and implement more than one fill technique.

Example:

```
      table←PlayfairTable 'KENNETHEIVERSON'
      table
KENTH
IVRSO
ABCDF
GLMPQ
UWXYZ
```

Task 2 – Encryption

To encrypt a message, break the message into digraphs (groups of 2 letters) such that, for example, "HelloWorld" becomes "HE LL OW OR LD" and map them out on the key table. If needed, append a "Z" to complete the final digraph. The two letters of the digraph are considered as the opposite corners of a rectangle in the key table. Note the relative position of the corners of this rectangle. Then apply the following 4 rules, in order, to each pair of letters in the plaintext:

- If both letters are the same (or only one letter is left), add an "X" after the first letter. Encrypt the new pair and continue. Thus "HE LL OW OR LD" becomes "HE LX LO WO RL DZ".
- If the letters appear on the same row of your table, replace them with the letters to their immediate right respectively (wrapping around to the left side of the same row if a letter in the original pair was on the right side of the row).
- If the letters appear on the same column of your table, replace them with the letters immediately below respectively (wrapping around to the top side of the same column if a letter in the original pair was on the bottom side of the column).
- If the letters are not on the same row or column, replace them with the letters on the same row respectively but at the other pair of corners of the rectangle defined by the original pair. The order is important – the first letter of the encrypted pair is the one that lies on the same row as the first letter of the plaintext pair.

Write an APL function named `PlayfairEncrypt` which:

- takes a character vector right argument representing the original message
- take a 5×5 character matrix left argument representing the Playfair table
- returns a character vector representing the encrypted message

Example: (using the table from the Task 1 example)

```
      table PlayfairEncrypt 'HELLOWORLD'
KNMWQVZVVMFY
```

Task 3 – Decryption

Write an APL function named `PlayfairDecrypt` which:

- takes a character vector right argument representing the encrypted message
- take a 5×5 character matrix left argument representing the Playfair table
- returns a character vector representing the original message

Example: (using the table from the Task 1 example and encrypted message from the Task 2 example)

```
      table PlayfairDecrypt 'KNMWQVZVVMFY'
HELXLOWORLDZ
```

Recreation and Games Problem Set

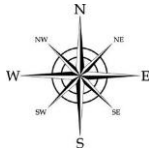
Recreation and Games Problem 1 (low difficulty) – Searching for the Right Words…

A word search puzzle consists of a rectangular table of letters and a list of words that are "hidden" in the table. The words can be hidden in forward or reverse letter order and in any direction – horizontal, vertical or diagonal.

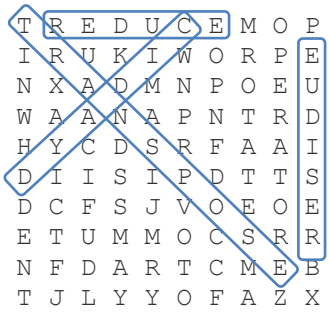
Task 1 – Word Search

Write an APL function named `WordSearch` which

- takes a character matrix representing the word search puzzle as its left argument
- takes a vector of character vectors representing the words to find as its right argument
- returns a 3-column matrix with one row per word in the right argument containing
    - the word
    - the row and column in the puzzle of the first letter of the word
    - the compass "direction" of the word (one of N,E,S,W,NE,SE,SW,NW)



Example: using the sample variables **`puzzle`** and **`words`** which are supplied in the competition template and represent the puzzle below




```
      ]disp puzzle WordSearch words
┌→──────────┬~──┬→┐
↓DYADIC     │6 1│NE│
├→──────────┼~──┼→┤
 REDUCE     │1 2│E │
├→──────────┼~──┼→┤
 RESIDUE    │8 10│N │
├→──────────┼~──┼→┤
 TRANSPOSE  │1 1│SE│
└→──────────┴~──┴→┘
```

Recreation and Games Problem 2 (medium difficulty) – Scrabble Scramble

APL is an excellent tool to use for data transformation and manipulation. This problem is representative of the common occurrence where you have an external data resource that may not be in the format you need.

Scrabble is a board game in which players randomly select 7 letter tiles and spell words on a grid. The letter tiles each have a point value based on their letter. There are 2 blank tiles that can be used as wild cards to spell words but have no point value. A bonus of 50 points is awarded if all 7 tiles are used to spell a word.

English-language editions of Scrabble contain 100 letter tiles, in the following distribution:

- 2 blank tiles (scoring 0 points)
- *1 point*: **E** ×12, **A** ×9, **I** ×9, **O** ×8, **N** ×6, **R** ×6, **T** ×6, **L** ×4, **S** ×4, **U** ×4
- *2 points*: **D** ×4, **G** ×3
- *3 points*: **B** ×2, **C** ×2, **M** ×2, **P** ×2
- *4 points*: **F** ×2, **H** ×2, **V** ×2, **W** ×2, **Y** ×2
- *5 points*: **K** ×1
- *8 points*: **J** ×1, **X** ×1
- *10 points*: **Q** ×1, **Z** ×1

For instance the set of tiles: (using ? to represent a blank tile)

Z O O L G Y ?

Could be used to spell "ZOOLOGY", scoring 69 points as follows

Z	O	O	L	?	G	Y	Bonus	Total
10	1	1	1	0	2	4	50	69

In this problem you are tasked with constructing a word list from a set of publicly available dictionary files (which are supplied in the contest download) and then using the word list to find the maximum scoring word(s) which can be constructed from a set of 7 letter tiles.

The dictionary files are found in the **/Data/Dictionary** folder in the contest zip file.

In the actual Scrabble board game, different positions on the grid affect letter and word scoring and each new word placed on the board must intersect with a word already existing on the board. To simplify this exercise, you need not consider either of these constraints.

Task 1 – Getting Wordy

Write an APL function named `BuildDictionary` which

- takes a character vector right argument representing the name of the folder containing the dictionary files
- returns a vector of words read in from the files

Notes:

- the system function `⎕SH` can be used to issue a command to the operating system and retrieve its output
- you can use the function `⎕SE.UnicodeFile.ReadNestedText` to read the file
- the dictionary files contain possessive words (words containing quotes) and abbreviations (words with uppercase letters). These should be removed from your word list.

Example (your folder name and output may be different):

```
      dictionary←BuildDictionary 'c:/Contest2014/Data/Dictionary/'

      ]disp 8↑dictionary
```

```
┌→────────┬─────┬──────┬─────────┬────────┬───────┬──────┬─────┐
│afterward│among│analog│apologize│behavior│catalog│center│color│
└─────────┴→────┴→─────┴→────────┴→───────┴→──────┴→─────┴→────┘
```

Task 2 – Max It Out

Write an APL function named `MaxWord` which

- takes a character vector right argument of the (up to) 7 tiles available in a player's hand (? is used to represent a blank tile)
- takes a left argument of the dictionary you built in Task 1
- returns a 2 element vector of
  - a character vector of the maximum scoring word
  - the score for the word

Example:

```
      ]disp dictionary MaxWord 'OZGYLO?'
```

```
┌→────────┬──┐
│ZOOL?GY│69│
└─────────┴→─┘
```

NOTE: Any of Z?OLOGY, ZO?LOGY, or ZOOL?GY would be correct

Recreation and Games Problem 3 (high difficulty) – Bridge Building

Contract bridge (or simply bridge) is a trick-taking game using a standard 52-card deck. It is played by four players in two competing partnerships, with partners sitting opposite each other around a table. The game consists of several deals each progressing through four phases: dealing the cards, the auction (also referred to as bidding), playing the cards, and scoring the results.

A key component of the auction phase is evaluating the strength of one's dealt cards (also referred to as one's hand). There are a number of evaluation methodologies in use by bridge players, but for the purposes of this exercise, we will use the following:

Hand Strength = High Card Points + Distributional Points + Corrections
where:

High Card Points	For each	Add this many points
	Ace	4
	King	3
	Queen	2
	Jack	1
Distributional Points	Add 1 point for each card beyond the 4th in a suit e.g. a suit with 5 cards would get 1 point, a suit with 6 would get 2, etc. Add points for each suit with 5 or more cards	
Corrections	Add 1 point if the hand has all 4 aces Subtract 1 point if a hand has no aces Subtract 1 point for each suit that has only a King, only a Queen and one other card only a Jack and two other cards	

Task 1 – Let's Make a Deal

Write an APL function named `DealHands` which simulates a random bridge deal (13 cards to each of 4 players) and

- is niladic. (Takes no arguments)
- returns a 4 element vector of 13 element vectors representing the 13 cards dealt to the 4 players. You can use a representation of your choosing for the card values.
  e.g. the 7 of Spades could be represented by
  - a 2 element character vector - e.g. '♠7' or '7♠'.
    The suit characters can be generated using `(⎕UCS 9823+⍳4)`.
  - a 2 element integer vector of indices into the card suits (♠,♡,♢,♣) and ranks (A,K,Q,J,T,9,8,7,6,5,4,3,2) - e.g. (1 8)
    Note 'T' is used for 10.
  - an integer encoding of the indices above - e.g. 108
  - Something clever that you come up with ☺.
    If you choose this option, be sure to document your result.

The following conditions should be true:

```
      hands←DealHands
      ⍴hands          ⍝ number of hands
4
      ⍴¨hands         ⍝ number of cards in each hand
13  13  13  13

      ⍴∪⊃,/hands      ⍝ there should be 52 unique cards
52
```

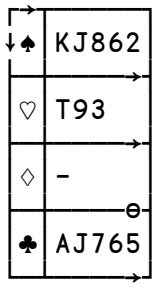
Task 2 – What's In Your Hand?

Write an APL function named `ShowHand` which

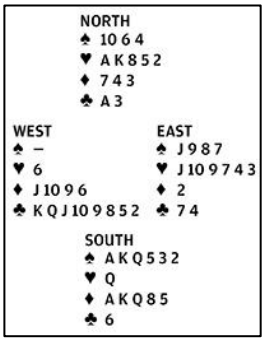
- takes a 13 element vector representing a bridge hand (one hand from the result of `DealHands`)
- returns a formatted, 2-column, properly sorted hand as follows
  - Suits (first column) are ranked high to low ♠ (Spades), ♡ (Hearts), ♢ (Diamonds), ♣ (Clubs).
  - Ranks within suits (values in the second column) are high to low A,K,Q,J,T,9,8,7,6,5,4,3,2.
- represents suits that have no cards by displaying a – (dash)

Example: (your values will likely be different)

```
      ]disp ShowHand 4⊃hands   ⍝ display the 4th hand
```



Just for fun… Once you've written `ShowHand` it is trivial to display an entire deal in a format that's similar to one commonly used in bridge publications.




```
      3 3⍴' '⍪'North' 'West' 'East' 'South'{⍪⍺ ⍵}⍤ShowHand⍤hands
            North
            ♠ AQ
            ♡ QJ7
            ♢ KJT973
            ♣ K9
West                    East
♠ T753                  ♠ 94
♡ A52                   ♡ K864
♢ AQ5                   ♢ 8642
♣ Q42                   ♣ T83
            South
            ♠ KJ862
            ♡ T93
            ♢ -
            ♣ AJ765
```

Task 3 – What's It Worth To Ya?

Write an APL function named `ValueHand` which

- takes as its right argument a 13 element vector representing a bridge hand (one hand from the result of `DealHands`)
- returns a 3 element integer vector of the total high card points, distributional points, and corrections as described earlier

Example: the four hands listed previously would be evaluated as follows:

Hand	Value
North ♠ AQ ♡ QJ7 ♢ KJT973 ♣ K9	16 High Card Points - 1 Ace, 2 Kings, 2 Queens, 2 Jacks 2 Distribution Points - 6-card ♢ Diamond suit ¯2 Corrections - ¯1 each for ♠Q and ♡J 16 Total Points
West ♠ T753 ♡ A52 ♢ AQ5 ♣ Q42	12 High Card Points - 2 Aces, 2 Queens 0 Distribution Points 0 Corrections 12 Total Points
East ♠ 94 ♡ K864 ♢ 8642 ♣ T83	3 High Card Points - 1 King 0 Distribution Points ¯1 Corrections - no Aces 2 Total Points
South ♠ KJ862 ♡ T93 ♢ - ♣ AJ765	9 High Card Points - 2 Aces, 2 Queens 2 Distribution Points - 2 5-card suits (♠ Spades and ♣ Clubs) 0 Corrections 11 Total Points

```
      ]disp ValueHand¨hands
┌→──────┬→──────┬→──────┬→──────┐
│16 2 ¯2│12 0 0 │3 0 ¯1 │9 2 0  │
└~──────→└~──────→└~──────→└~──────→
```

Task 4 – Simulation Stimulation

In contract bridge, the number of high card points a hand may have can vary from 0 (no aces, kings, queens, or jacks) to 37 (4 aces, 4 kings, 4 queens and 1 jack). When a bridge partnership (North-South or East West) takes all 13 tricks, it's known as a grand slam. One rule of thumb to bid for a grand slam is that the total high card point count between the two hands of the partnership should be at least 37.

Write a **non-looping** APL function named `Simulate` which

- takes an integer right argument indicating the number of deals to evaluate in the simulation.
- returns a 2 element vector
    - the first element is a 3 column matrix where column 1 contains the high card point values from 0 through 37, column 2 contains the number of hands that had the corresponding point value and column 3 contains the percent (expressed as a number between 0 and 1) of the total hands that had the corresponding point value.
    - The second element is a 2 element vector where the first element is the number of deals where either partnership (North-South or East-West) totaled 37 or more combined points, and the second element is the percentage of hands where this occurred.

Example: (your numbers will most likely be somewhat different)

```
      ]disp Simulate 100000 ⍝ simulate 100,000 deals
┌→──────────────────┬──────────┐
│ 0  1454 0.003635  │16 0.00016│
│ 1  3289 0.0082225 │          │
│ 2  5566 0.013915  │          │
│ 3  9831 0.0245775 │          │
│ 4 15177 0.0379425 │          │
│ 5 20778 0.051945  │          │
│ 6 26322 0.065805  │          │
│ 7 31954 0.079885  │          │
│ 8 35280 0.0882    │          │
│ 9 37316 0.09329   │          │
│10 38013 0.0950325 │          │
│11 35948 0.08987   │          │
│12 32059 0.0801475 │          │
│13 27450 0.068625  │          │
│14 22742 0.056855  │          │
│15 17616 0.04404   │          │
│16 13285 0.0332125 │          │
│17  9504 0.02376   │          │
│18  6500 0.01625   │          │
│19  4118 0.010295  │          │
│20  2551 0.0063775 │          │
│21  1492 0.00373   │          │
│22   858 0.002145  │          │
│23   474 0.001185  │          │
│24   207 0.0005175 │          │
│25   116 0.00029   │          │
│26    52 0.00013   │          │
│27    27 0.0000675 │          │
│28    12 0.00003   │          │
│29     5 0.0000125 │          │
│30     4 0.00001   │          │
│31     0 0         │          │
│32     0 0         │          │
│33     0 0         │          │
│34     0 0         │          │
│35     0 0         │          │
│36     0 0         ↓          │
│37     0 0         │          │
└~──────────────────→┴~────────→┘
```