DYALOG

APL Problem Solving Competition Phase 1

Introduction

The Phase 1 problems are designed to be solved using short APL functions. If you find yourself writing more than a couple of statements in your solution, then there is probably a better way to do it.

Submission format

Each solution must be a single dfn or tacit function.

A dfn is one or more APL statements enclosed in braces `{}`. The left hand argument, if any, is represented in a dfn by `⍺`, while the right hand argument is represented by `⍵`. For example:

```
      'Hello' {⍺,'-',⍵,'!'} 'world'
Hello-world!
```

A dfn terminates on the first statement that is not an assignment. If that statement produces a value, then the dfn returns that value as its result. The diamond symbol `⋄` separates APL statements. For example:

```
      'left' { ⍵ ⋄ ⍺ } 'right'
right
```

More information on dfns can be found on the APL Wiki.

A tacit function is an APL expression that does not explicitly mention its arguments. In the example below, `(+⌿÷≢)` is a tacit function that computes the average of a vector (list) of numbers:

```
      (+⌿÷≢) 1 2 3 4 5 6
3.5
```

More information on tacit functions can be found on the APL Wiki.

Judging Guidelines

When you submit a Phase 1 solution, it will be automatically tested using a number of basic and edge cases. Solutions will mainly be judged based on:

- Generality: does your function handle the given basic and edge cases?
- Use of array-oriented thinking: did you write array-oriented APL or something that looks more like C# written in APL?

You should not include comments in your Phase 1 solutions.

Tips

- Several of the problem descriptions will describe arguments that can be a scalar (a single element) or a vector (a list). This is largely pedantic, but in such cases your functions should produce correct results for both types of input.
- The symbol `⍝` is the APL comment symbol. In some of the examples, we provide comments to give you more information about that particular example.
- Some of the problem test cases use "boxed display" to make the structure of the returned results clearer. Boxing is always active on TryAPL and can be enabled in your local APL Session with the `]Box on` user command:

```
      ⍳¨⍳4
 1  1 2  1 2 3  1 2 3 4
      ]Box on
Was OFF
      ⍳¨⍳4
┌─┬───┬─────┬───────┐
│1│1 2│1 2 3│1 2 3 4│
└─┴───┴─────┴───────┘
```

Sample problem

The content of the orange box shows what a typical Phase 1 problem description looks like. It also presents some possible solutions of varying quality, and explains how to provide your own solution.

Each problem starts with a task description; some also include a hint suggesting one or more APL primitives. These may be helpful in solving the problem, but you are under no obligation to use them. Clicking on a primitive in the hint opens the Dyalog documentation page for that primitive.

Each problem ends with some example cases. You can use these as a basis for implementing your solution.

Counting Vowels A

Write an APL function to count the number of vowels (A, E, I, O, U) in a character vector or scalar consisting of uppercase letters (A–Z).

Hint: The membership function `X∊Y` could be helpful for this problem.

Examples

```
      (your_function) 'COOLAPL'
3
      (your_function) ''    ⍝ empty argument
0
      (your_function) 'NVWLSHR'    ⍝ no vowels here
0
```

Below are three sample solutions. All three produce the correct answer, but the first two functions would be ranked higher by the competition judging committee as they demonstrate better use of array-oriented programming than the third one.

```
      ({+/⍵∊'AEIOU'}) 'COOLAPL'    ⍝ good dfn
3
      (+/∊∘'AEIOU') 'COOLAPL'    ⍝ good tacit function
3
      ⍝ suboptimal dfn:
      {(+/⍵='A')+(+/⍵='E')+(+/⍵='I')+(+/⍵='O')+(+/⍵='U')} 'COOLAPL'
3
```

1: Elimination Sort

An "Elimination Sort" is a somewhat farcical sorting algorithm which starts with the leftmost element and keeps subsequent elements that are at least as large as the previous kept element, discarding all other elements. For example:

```
      EliminationSort 1 3 7 3 5 8 5 8 1 6 1 8 1 10 8 4 3 4 1 4
1 3 7 8 8 8 10
```

Write a function that:

- takes a non-empty numeric vector right argument
- returns an "Elimination-sorted" vector of the right argument

Hint: The *progressive-maxima* idiomatic phrase `⌈\`, the *greater or equal* function `≥`, and the *replicate* function `/` could be helpful in solving this problem.

Examples

```
      (your_function) ⍳10
1 2 3 4 5 6 7 8 9 10

      (your_function) 2 1 4 3 6 5 8 7 10 9
2 4 6 8 10

      (your_function) 1000 2500 1333 1969 3141 2345 3141 4291.9 4291.8 4292
1000 2500 3141 3141 4291.9 4292

      EliminationSort 1 3 7 3 5 8 5 8 1 6 1 8 1 10 8 4 3 4 1 4
1 3 7 8 8 8 10
```

2: Put It In Reverse ⏪

The *find* function `X⍷Y` identifies the beginnings of occurrences of array `X` in array `Y`.

In this problem, you're asked to return a result that identifies the endings of occurrences of array `X` in array `Y`. To keep things simple, `X` and `Y` will be at most rank 1, meaning they'll either be vectors or scalars.

Write a function that:

- takes a scalar or vector left argument
- takes a scalar or vector right argument
- returns a Boolean result that is the same shape as the right argument where 1's mark the ends of occurrences of the left argument in the right argument

💡 **Hint:** The *find* function `⍷` and *reverse* function `⌽` could be helpful in solving this problem.

Examples

```
      'abra' (your_function) 'abracadabra'
0 0 0 1 0 0 0 0 0 0 1

      'issi' (your_function) 'Mississippi'
0 0 0 0 1 0 0 1 0 0 0

      'bb' (your_function) 'bbb bbb'
0 1 1 0 0 1 1

      (,42) (your_function) 42
0

      42 (your_function) 42
1

      (,42) (your_function) ,42
1

      'are' 'aquatic' (your_function) 'ducks' 'are' 'aquatic'
'avians'
0 0 1 0
```

3: Caesar Salad

A Caesar cipher, also known as a shift cipher, is one of the simplest encryption techniques. In a Caesar cipher, each letter in the plaintext is replaced by a letter some fixed number of positions away in the alphabet, effectively "shifting" the alphabet.

Write a function that:

- takes a single integer left argument representing the size of the shift
- takes a character vector right argument representing the plaintext message
- returns a result with the same shape as the right argument representing the encrypted message

Notes:

- Use `' ',⎕A` as the alphabet
- You can assume that the plaintext message will contain only characters found in the alphabet.

💡 **Hint:** The *rotate* function `⌽` could be helpful in solving this problem.

Examples

```
      4 (your_function) 'HELLO WORLDS'
LIPPSD SVPHW

      ¯4 (your_function) 'HELLO WORLDS' ⍝ negative shifts are okay
DAHHKWSKNH O

      0 (your_function) 'HELLO WORLDS' ⍝ no shift is okay
HELLO WORLDS

      27 (your_function) 'HELLO WORLDS'
HELLO WORLDS

      ¯10 (your_function) '' ⍝ returns an empty vector
```

4: Like a Version

One common software version numbering scheme is known as "semantic versioning". Typically, semantic versioning uses three numbers representing a major version number, a minor version number, and a build number.

- The major version is incremented when a new version of the software introduces changes that would make existing applications using the software fail or behave differently.
- The minor version is incremented when new features are added that didn't previously exist — no pre-existing use of the software will fail in this case.
- The build number is incremented for bug fixes and similar changes.

Write a function that:

- takes 3-element integer vector left and right arguments each representing a major version, minor version, and build number.
- returns
  - `¯1` if the left argument represents a version number older than the right argument
  - `0` if the left argument represents a version number equal to the right argument
  - `1` if the left argument represents a version number newer than the right argument

Hint: The *less* function `<` could be helpful in solving this problem.

Examples

```
      1 2 3 (your_function) 1 2 3
0
      1 2 3 (your_function) 1 0 9
1
      14 2 11 (your_function) 14 2 12
¯1
```

5: Risky Business

The board game Risk is a game of world domination where opposing players roll dice to determine the outcome of one player's armies attacking another's. The attacker can roll up to three 6-sided dice and the defender can roll up to two 6-sided dice. The resulting rolls are then individually compared from highest to lowest. If the attacker's die value is greater than the defender's, the defender loses one army. If the defender's die value is greater than or equal to the attacker's, the attacker loses one army. If one player rolls more dice than the other other player, the additional dice are not considered in the evaluation. For this problem, we'll generalize the task by allowing any number of dice for either the attacker or defender, and any integer values in the arguments.

Write a function that:

- takes a non-empty, descending integer vector left argument representing the attacker's dice rolls
- takes a non-empty, descending integer vector right argument representing the defender's dice rolls
- returns a 2-element vector where the first element represents the number of armies the attacker lost and the second element represents the number of armies the defender lost.

Note: The left and right arguments do not need to be the same length.

Hint: The *less* function `<` could be helpful in solving this problem.

Examples

```
      6 6 4 2 1 (your_function) 6 5 5 ⍝ attacker loses 2 armies,
defender loses 1 army
2 1

      6 (your_function)⍥, 5 ⍝ ⍥, ravels both arguments (making them
vectors) before passing them to your function
0 1

      4 0 ¯1  (your_function) 3 1 ¯2
1 2
```

6: Key/Value Pairs

Representing data as key/value pairs (also known as name/value pairs) is a very common technique. For example, it can be found in query strings in HTTP URIs, attribute settings in HTML elements, and in JSON objects. One common representation for a key/value pair is to have a character key (name) followed by an equals sign (=) followed by the value. Multiple key/value pairs can be separated by a delimiter character or characters. For example:

```
    key1=value1;key2=value2
```

Write a function that:

- takes a 2-element character vector left argument where the first element represents the separator character between multiple key/value pairs and the second element represents the separator between the key and the value for each pair.
- takes a character vector right argument representing a valid set of key/value pairs (delimited as specified by the left argument).
- returns a 2-column matrix where the first column contains the character vector keys of the key/value pairs and the second column contains the character vector values.

Note: You may assume that there will be no empty names or values in the right argument.

Hint: The *partition* function ⊆ could be helpful in solving this problem.

Examples

```
      ⍴ ⎕← ' ='(your_function)'language=APL dialect=Dyalog'
┌────────┬──────┐
│language│APL   │
├────────┼──────┤
│dialect │Dyalog│
└────────┴──────┘
2 2

      ⍴ ⎕← ';:'(your_function)'duck:donald'
┌────┬──────┐
│duck│donald│
└────┴──────┘
1 2

      ⍴ ⎕← '/:'(your_function)'name:Morten/name:Brian/name:Adám'
┌────┬──────┐
│name│Morten│
├────┼──────┤
│name│Brian │
├────┼──────┤
│name│Adám  │
└────┴──────┘
3 2
```

7: Let's Be Rational ÷

A rational number is a number that can be expressed as the quotient of 2 integers `p÷q` — a numerator `p` and a denominator `q`. For example, for `1.5`, `p` and `q` would be `3` and `2`, respectively.

Write a function that:

- takes a single non-zero positive number right argument.
- returns a 2-element "integer" result representing the smallest non-zero positive values for `p` and `q` respectively

Notes:

- We use "integer" in the result description because the result might contain a number larger than can be represented as an integer data type in Dyalog APL. For example, the result when applying the function to `⌊/⍬` would be `1.797693135E308 1` which is represented as a 64-bit floating point array.
- The test for the correctness of your solution will be that, given
  `r ← (your_function) a`
  your solution should satisfy both the following:
  - `r ≡ ⌊0.5+r`
  - `a = ÷/r`

💡 **Hint:** The *Lowest Common Multiple* function `∧` or *Greatest Common Divisor* function `∨` could be helpful in solving this problem.

Examples

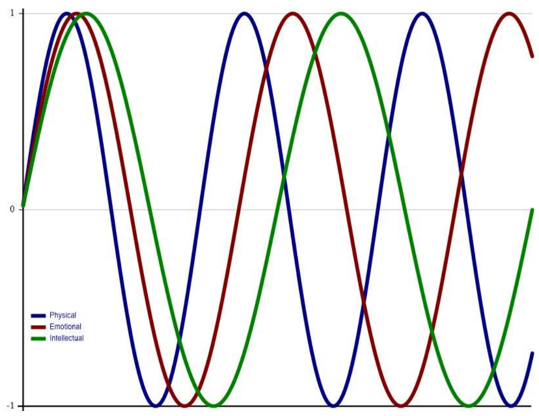
```
      (your_function) 1.2
6 5

      (your_function) 3.5
7 2

      (your_function) ÷3
1 3
```

8: Critical Thinking ∿

21 April 2023: Please see the note at the end of the problem description



The biorhythm theory is a pseudo-scientific idea that one's life is affected by three rhythmic cycles beginning from one's date of birth. The cycles are:

- The Physical cycle, with a periodicity of **23** days, affecting co-ordination, strength, and general well-being
- The Emotional cycle, with a periodicity of **28** days, affecting creativity, sensitivity, mood, perception, and awareness
- The Intellectual cycle, with a periodicity of **33** days, affecting alertness, analytical functioning, logical analysis, memory, and communication

"Critical days" are days when a cycle crosses the x-axis in either direction and are purported to be accompanied by unstable conditions in the corresponding area. A "double critical day" occurs when two of the three cycles cross the x-axis on the same day. Starting from one's birthdate, double critical days occur on multiples of the least common multiple of the half of the periodicities of the two involved cycles. Thus Physical-Emotional, Physical-Intellectual and Emotional-Intellectual double critical days can be calculated respectively using multiples of:

```
      23 23 28∧⍥(÷∘2)28 33 33
322 379.5 462
```

Fortunately, the dreaded "triple critical day", when all three cycles cross the x-axis on the same day, only occurs every `(∧/23 28 33÷2)` or 10,626 days (a bit more than 29 years).

Write a function that:

- takes a 3-element integer left argument representing a valid birthdate
- takes a 3-element integer right argument representing a valid date occurring on or after the left argument
- returns a 3-element integer array representing the date of the next double or triple critical day occurring on or after the date represented by the right argument.

Note: All the dates in this specification are to be in the form `year month day`

Hint: The *date time system function* `⎕DT` and *residue function* `|` could be helpful in solving this problem.

Examples

```
      1962 10 31 (your_function) 2023 1 1
2023 2 4

      1961 2 23 (your_function) 1961 2 23 ⍝ one's birthdate is a
triple critical day
1961 2 23
```

---

21 April 2023 Note:

We messed up... Even though we said a critical day occurs when a cycle crosses the x-axis in either direction, we didn't account for that in the subsequent description. (Unfortunately, neither did the reference we used when researching this problem.) We've amended the description above and copied the original text below.

To be fair to people who have already submitted "correct" solutions, we will accept solutions that conform to either the original or current descriptions.

We also express our thanks to the participants in The APL Orchard for bringing this to our attention.

Original text:

"Critical days" are days when a cycle crosses the x-axis in either direction and are purported to be accompanied by unstable conditions in the corresponding area. A "double critical day" occurs when two of the three cycles cross the x-axis on the same day. The periodicity of double critical days is the least common multiple of periodicities of the two involved cycles. So, the periodicities of the Physical-Emotional, Physical-Intellectual and Emotional-Intellectual double critical days can be calculated respectively as:

```
      23 23 28∧28 33 33
644 759 924
```

Fortunately, the dreaded "triple critical day", when all three cycles cross the x-axis on the same day, only occurs every `(∧/23 28 33)` or 21,252 days (a bit more than 58 years).

9: Flipping Pairs 🔁

This problem has no practical use in the real world (that the author can think of) other than to give your array manipulation muscles some exercise.

Write a function that:

- takes a non-empty non-scalar array right argument
- returns an array of the same shape as the argument, but with pairs of elements along the last axis "flipped". If the array has an odd number of elements in the last axis, leave the last element unchanged.

💡 **Hint:** Either the *reverse* function `⌽` used with the *partitioned enclose* function `⊂`, or the *grade up* function `⍋` used with the *index* function `⌷`, could be helpful in solving this problem.

Examples

```
      (your_function) ι10
2 1 4 3 6 5 8 7 10 9

      (your_function) ι9
2 1 4 3 6 5 8 7 9

      (your_function) 4 2ρι8
2 1
4 3
6 5
8 7

(your_function) 4 3ρι12
 2  1  3
 5  4  6
 8  7  9
11 10 12

      (your_function) 3 3 3ρι27
2  1  3
5  4  6
8  7  9

11 10 12
14 13 15
17 16 18

20 19 21
23 22 24
26 25 27

      (your_function) 2 3ρ'donald' 'duck' 'wrote' 'some' 'good'
'APL'
┌────┬──────┬─────┐
│duck│donald│wrote│
├────┼──────┼─────┤
│good│some  │APL  │
└────┴──────┴─────┘
```

10: Partition with a Twist 🧬

Splitting delimited data into sub-arrays using partitioning on a delimiter character (or characters) is a fairly common operation in APL. For instance, if you partition the character vector `'this is an example'` on each occurrence of the space character, there would be 4 sub-arrays: `'this' 'is' 'an' 'example'`. This problem adds a slight twist to the operation in that the left argument indicates how many sub-arrays to return.

Write a function that:

- takes a non-negative integer left argument, **N**
- takes a space-delimited character vector right argument, **string**
- returns an array of length **N** where:
  - if **N** is less than or equal to the number of sub-arrays in **string**, the first **N-1** elements of the result are the first **N-1** space-delimited partitions in **string**. The **N**th element of the result is the remaining portion of **string**.
  - if **N** is greater than the number of sub-arrays, pad the result with as many empty arrays as necessary to achieve length **N**.

  Note: Each space in **string** be considered as a delimiter. This means that leading, trailing, or contiguous spaces are potentially significant.

  💡 **Hint:** The *partitioned enclose* function `⊆` could be helpful in solving this problem.

Examples

```
      1 (your_function) 'this is a sample'
┌────────────────┐
│this is a sample│
└────────────────┘

      2 (your_function) 'this is a sample'
┌────┬───────────┐
│this│is a sample│
└────┴───────────┘

      4 (your_function) 'this is a sample'
┌────┬──┬─┬──────┐
│this│is│a│sample│
└────┴──┴─┴──────┘

      ⍴¨4 (your_function) 'this is a sample' ⍝ each sub-array is a
vector
┌─┬─┬─┬─┐
│4│2│1│6│
└─┴─┴─┴─┘

      13 (your_function) '  this  is  a sample  ' ⍝ note the
leading, trailing, and multiple interior spaces
┌┬┬────┬┬──┬┬─┬──────┬┬┬┬┬┐
│││this││is││a│sample││││││
└┴┴────┴┴──┴┴─┴──────┴┴┴┴┴┘

      0 (your_function) 'this is a sample' ⍝ returns an empty
vector


      4 (your_function) ''
┌┬┬┬┐
│││││
└┴┴┴┘
```