APL Problem Solving Competition
Phase 2

Introduction

Phase 2 is similar to Phase 1 in that you submit solutions for each problem separately. In contrast to Phase 1, Phase 2 solutions are likely larger and more complex, and they should be adequately commented. You need to have submitted at least one correct Phase 1 solution before you can submit anything for Phase 2.

Each Phase 2 problem consists of one or more tasks. You must complete all of the tasks for the problem to be considered complete and be judged by the competition committee. You can write additional subfunctions in support of your solutions if necessary.

Each problem description contains one or more examples. However, the judging committee will also submit your solutions to additional testing.

Submission format

You can write your solutions using any combination of tradfns or dfns. The only requirement is that the function name and syntax must match the task description. For example, if the task description is:

> Write a function named `Plus` which:
>
> - takes a numeric array right argument.
> - takes a numeric singleton left argument.
> - returns a result that is the same shape as the right argument and whose values are the sums of the left argument added to each element of the right argument.

then either of the following would be valid solutions:

```
∇ r←a Plus b
  r←a+b
∇

Plus←{⍺+⍵}
```

Judging Guidelines

Phase 2 will mainly be judged based on:

- Did you solve the problem?
- Does your solution demonstrate appropriate use of array-oriented techniques? Solutions that use looping where an obvious array-based solution exists will be judged lower.
- Did you comment your solution? It's not necessary to write a novel, or add a comment to every line, but comments describing non-trivial lines of code are advised. These help the judging committee determine your level of understanding of the problem and its solution.
- Is your solution original? Your solution should be your own work and not a copy or near-copy of an already-published solution.

Tips

- Read the descriptions carefully.
- Don't make any assumptions about shape, rank, datatype, or values that are not explicitly stated in the description. For example, if an argument is stated to be a numeric array then it can be any numeric type (Boolean, integer, floating point, complex) and of any shape or depth.
- Make sure that your functions return a result rather than just displaying output to the session.
- Pay attention to any additional judging criteria that may stated in an individual problem description.

1: Bowling Them Over... 🎳 (2 tasks)

Ten-pin bowling is an activity in which a player rolls a bowling ball down a bowling lane toward ten pins arranged in an equilateral triangle. The objective is to knock over the pins with the ball. A game consists of ten frames. A **frame** consists of one or two rolls of the ball. If the first roll results in all ten pins being knocked down, this is known as a **strike** and the frame is over. Otherwise, the player rolls the ball a second time to attempt to knock down the remaining pins. If all of the remaining pins are knocked down, this is known as a **spare**. If any pins remain standing after the second roll, this is known as an **open frame**. Note that if the first roll knocks down zero pins yet the second roll knocks down all ten pins, this is still considered a spare and not a strike.

Scoring:

Each frame on a bowling scoresheet keeps a record of each roll as well as a running total. Frames are scored as follows:

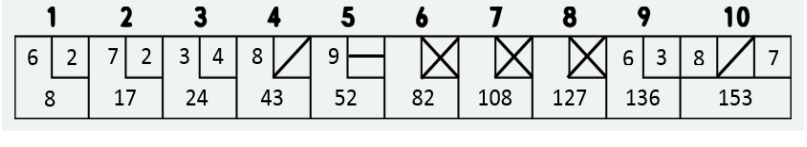
- **Open frame:** The total of the two rolls of the frame are added to the running point total.
- **Spare:** The frame receives 10 points plus a bonus of the number of pins knocked down on the next roll. A spare in the tenth frame receives an additional roll for bonus points.
- **Strike:** The frame receives 10 points plus a bonus of the number of pins knocked down on the next two rolls. A strike on the first roll of the tenth frame receives two additional rolls for bonus points.

Frame Notation:

- A roll which does not knock down any pins is notated by **–**.
- A single roll resulting in a strike is notated by **X**.
- A second roll resulting in a spare is notated by **/**.
- Otherwise the number of pins knocked down is notated.

In the image below of a scored game of bowling there are 5 open frames (frames 1, 2, 3, 5 and 9), 2 spares (frames 4 and 10) and 3 strikes (frames 6, 7 and 8).

- The scoring for the open frames just adds the total number of pins knocked down in the frame to the running total.
- The scoring for frame 4 is 24 (the previous total in frame 3) plus 10 (for the spare) plus 9 (as the bonus from the next roll in frame 5) giving a score of 43.
- The scoring for frame 6 is 52 (the previous total in frame 5) plus 10 (for the strike) plus 20 (as the bonus from the next two rolls which also happen to be strikes) giving a score of 82.
- The scoring for frame 10 is 136 (the previous total in frame 9) plus 10 (for the spare) plus 7 (as the bonus roll) giving a final score of 153.



A game of bowling can be represented as a character vector. The game in the image can be represented as:

```
game←'627348/9-XXX638/7'
```

A perfect game consisting of 12 strikes (and scoring 300) would be represented as:

```
game←'XXXXXXXXXXXX'
```

At the other end of the spectrum, a game scoring 0 points would be represented as:

```
game←'--------------------'
```

Task 1: Write a monadic function named `ValidGame` that:

- takes a character array right argument that represents a bowling game.
- returns a Boolean scalar, where 1 indicates that the right argument represents a complete and valid bowling game; 0 otherwise.

Examples

```
      ValidGame '6272348/9-XXX638/7' ⍝ game from image above
1
      ValidGame ''  ⍝ bad length
0
      ValidGame 'X'  ⍝ not a vector
0
      ValidGame 1 12⍴'X' ⍝ not a vector
0
      ValidGame¨20 20 20 21 12⍴¨'6' '9-' '64' '9/' 'X'
0 1 0 1 1
```

Task 2: Write a monadic function named `BowlingScore` that:

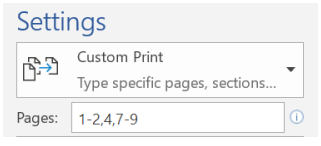
- takes a character vector right argument that represents a complete and valid bowling game.
- returns a 10-element integer vector representing the frame-by-frame running total score of the game.

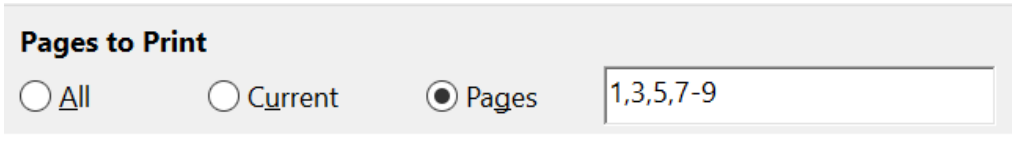
Examples

```
       BowlingScore '6272348/9-XXX638/7' ⍝ game from image above
8 17 24 43 52 82 108 127 136 153
      BowlingScore '--------------------'
0 0 0 0 0 0 0 0 0 0
      BowlingScore 'XXXXXXXXXXXX'
30 60 90 120 150 180 210 240 270 300
```

2: Make a List... (1 task)

This problem is inspired by the "pages to print" specification in many print dialog boxes, where you can enter a set of pages by entering any combination of comma-separated individual page numbers (as in 1,3,5), or hyphenated page number ranges (as in 4-7). Below are two examples of such dialog boxes.





This problem expands on that notation, allowing negative numbers to be used in the list specification. Negative numbers can be specified by prepending either the APL high minus (`¯`) or a hypen (`-`).

Task 1: Write a monadic function named `MakeList` that:

- takes a right argument that is a character vector or scalar representing a list specification.
- returns a vector of the integer(s) specified in the list, in the order in which they were specified.

Examples

```
      1×MakeList '7' ⍝ the 1× is to verify that the result is numeric
7
      MakeList '7,42,-4,¯5'
7 42 ¯4 ¯5
      0≡MakeList ''  ⍝ empty list yields an empty numeric vector
1
      1×MakeList '44-42,-4--7,¯1-¯3,42-44'
44 43 42 ¯4 ¯5 ¯6 ¯7 ¯1 ¯2 ¯3 42 43 44
```

3: As Evenly as Possible... ⚖️ (1 task)

Task 1: Write a dyadic function named `Divvy` that:

- has a right argument `array` that is a simple (non-nested) vector or scalar array.
- has a left argument `n` that is a positive integer scalar.
- returns a result `r` that is a nested vector such that the following are true:
  - there are `n` elements in the result: `n = ≢r`
  - the lengths of each element in the result must not differ by more than 1: `1≥(⌈/-⌊/)≢¨r`
  - the elements of the result occur in the same order as the elements the right argument: `r ≡ö∈ array`

Note that if the elements of the result are of unequal lengths, then there is no requirement that the longer (or shorter) elements occur in any particular order. This means that:

```
┌─┬─┬─┬─┬─┐
│1│2 3│4 5│
└─┴─┴─┴─┴─┘
```

```
┌─┬─┬─┬─┬─┐
│1 2│3│4 5│
└─┴─┴─┴─┴─┘
```

```
┌─┬─┬─┬─┬─┐
│1 2│3 4│5│
└─┴─┴─┴─┴─┘
```

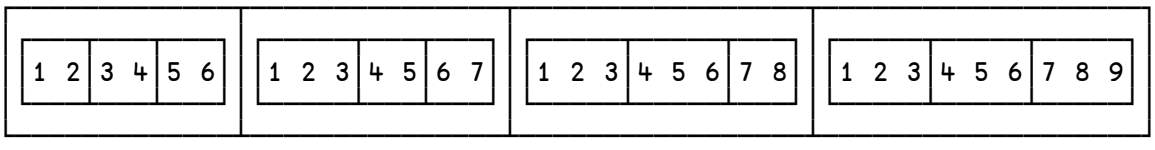
are all correct answers for

```
      3 Divvy 1 2 3 4 5
```

Examples (using `]Box on`)

As noted above, the grouping in your results when the elements of `r` are of different lengths might be different from these examples.

```
      3 Divvy¨ ⍳¨6 7 8 9
```



```
      1 Divvy 1 ⍝ returns a nested vector
```

```
┌─┐
│1│
└─┘
```

```
      5 Divvy 'Dyalog APL'  ⍝ works with character arrays
```

```
┌──┬──┬──┬──┬──┐
│Dy│al│og│ A│PL│
└──┴──┴──┴──┴──┘
```

```
      4 Divvy ''
```



```
      5 Divvy 'APL'
```



```
      1 2 3 Divvy¨ 1
```



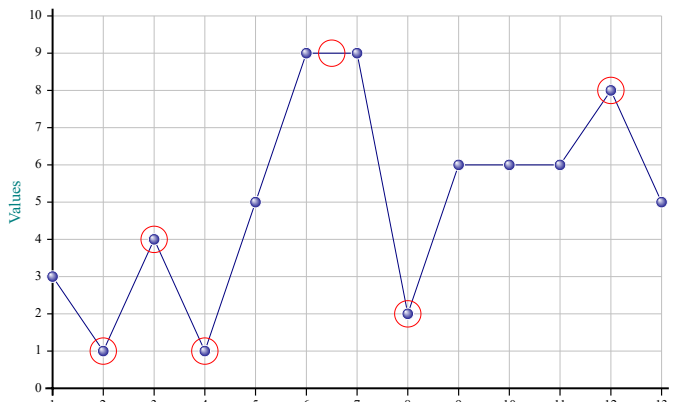
```
      ⎕←a←,(⍳5),[1.1]'ABCDE'  ⍝ mixed character and number
1 A 2 B 3 C 4 D 5 E
      3 Divvy a
```

```
┌─────┬─────┬─────┐
│1 A 2│B 3 C│4 D 5 E│
└─────┴─────┴─────┘
```

4: Going to (Local) Extremes... 📈 (1 task)

Given a numeric vector that represents Y values on a graph (the X value is implied by its index within the vector), the midpoint of local minima or maxima can be thought of as the average of the index values that are bounded by positive and then negative slopes or negative and then positive slopes. Usually the midpoints will coincide with the minima or maxima; if there are multiple consecutive minima or maxima then the midpoint will be the mean of the extreme values.

For example, in the graph below, the maxima and minima midpoints as indicated by the red circles are 2 3 4 6.5 8 12. The 6.5 value is due to the consecutive local maximum values at indices 6 and 7.



Endpoints are not considered local minima or maxima. There is no local minima or maxima if there is no inflection of slope, implying that the argument must have at least 3 values.

Task 1: Write a monadic function named `MinMax` that:

- takes a right argument that is an integer vector or scalar.
- returns a numeric vector of the midpoints, if any, of the local minima and maxima.

Examples

```
      MinMax 3 1 4 1 5 9 9 2 6 6 6 8 5 ⍝ from the example above
2 3 4 6.5 8 12

      MinMax 0 ⍝ returns an empty vector (not enough values)

      MinMax 12 ⍝ returns an empty vector (not enough values)

      MinMax 3 3 3 ⍝ returns an empty vector (no inflection point)

      MinMax 1 3 2 4
2 3
```

5: Lexicographically Ordered... (2 tasks)

These tasks are inspired by two problems from the excellent Bioinformatics website rosalind.info.

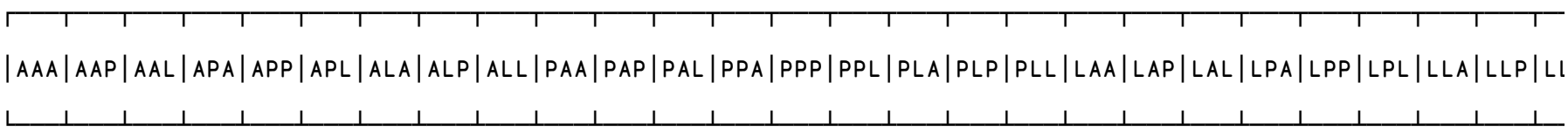
Task 1: Write a dyadic function named `lexf` that:

- has a right argument `A` that is a character scalar or vector of at most 10 unique symbols that define an ordered alphabet.
- has a left argument `n` that is an integer scalar in the range [0,6].
- returns a vector of character vectors that represent all the strings of length `n` that can be made from `A` ordered lexicographically (in the same order as found in `A`).

The inspiration for this task originates from Enumerating k-mers Lexicographically. Note that this task varies slightly from the the original problem description in that we expect your function to address edge cases such as an empty alphabet or a left argument of 0.

Examples using `]Box on`

```
      3 lexf 'APL'
```




```
      3 lexf '' ⍝ returns an empty vector
      3 lexf 'Q'
```


```
      0 lexf 'DNA' ⍝ returns a 1-element vector containing an empty vector
```

```
      1 lexf 'DYALOG' ⍝ returns a vector of 1-element vectors
```

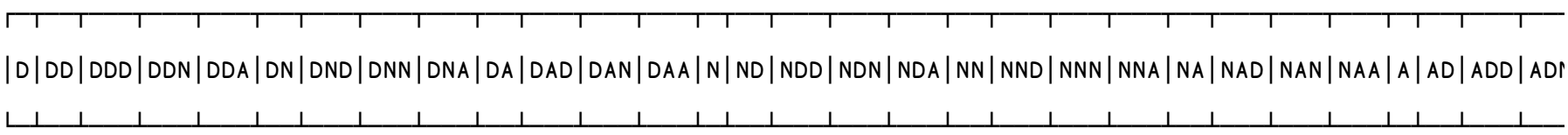
Task 2: Write a dyadic function named `lexv` that:

- has a right argument `A` that is a character scalar or vector of at most 12 unique symbols that define an ordered alphabet.
- has a left argument `n` that is a positive integer scalar in the range [1,4].
- returns a vector of character vectors that represent all strings of up to length `n` that can be made from `A` ordered lexicographically (in the same order as found in `A`).

The inspiration for this task originates from Ordering Strings of Varying Length Lexicographically. Note that this task varies slightly from the the original problem description.

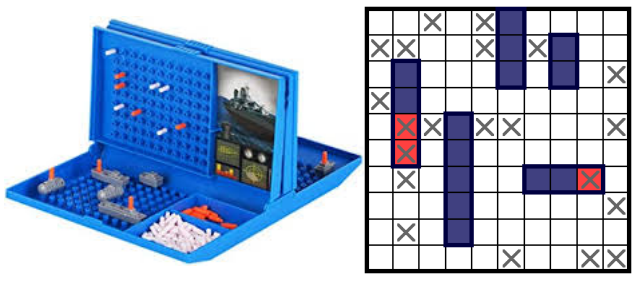
Examples using `]Box on`

```
      3 lexv 'DNA'
```




```
      3 lexv '' ⍝ returns an empty vector
      1 lexv 'DYALOG'
```

6: You Sunk My Battleship! ▦ (2 tasks)



Battleship is a strategy-type guessing game for two players. It is played on ruled grids (paper or board) on which each player's fleet of ships are marked. The locations of the fleets are concealed from the other player. Players alternate turns calling "shots" at the other player's ships, and the objective of the game is to locate and destroy the opposing player's fleet.

The board game is typically played on a 10×10 grid with each player having ships of lengths 5,4,3,3, and 2. Ships are placed horizontally and/or vertically and cannot overlap or intersect (but can touch other ships). However, for this problem, we'll allow a grid of any size and fleets of arbitrary size with ships of any length (that will fit within the grid).

Your first task will be to validate that a board is an acceptable configuration given the size and fleet parameters.

Task 1: Write a dyadic operator named `ValidBoard` that:

- has a left operand `size` that is a 2-element integer vector specifying the size of the board.
- has a right operand `fleet` that is an integer scalar or vector defining the size of ships in the fleet.
- has a right argument `board` that is an integer matrix where:
  - a `0` indicates an empty cell.
  - a non-zero value `n` indicates a cell occupied by the ship at `fleet[n]`
- returns an integer scalar or vector where:
  - 1 indicates the board is valid for the size and fleet specified.
  - 0 indicates the board is not valid for the size and fleet specified.

Examples

```
      ⊢board←3 3⍴2 2 2 1 3 3 1 0 4
2 2 2
1 3 3
1 0 4
      size←3 3
      fleet←2 3 2 1
      (size ValidBoard fleet) board ⍝ valid 3×3 board for fleet of 4 ships of lengths 2 3 2 1
1

⍝ ship 1 is not in contiguous cells
      (3 3 ValidBoard 2 3 2 1) □←3 3⍴2 2 2 1 3 3 0 1 4
2 2 2
1 3 3
0 1 4
0

⍝ there is not a 5th ship in the fleet
      (3 3 ValidBoard 2 3 2 1) □←3 3⍴2 2 2 1 3 3 1 5 4
2 2 2
1 3 3
1 5 4
0

⍝ ship 2 is not the correct length
      (3 3 ValidBoard 2 3 2 1) □←3 3⍴0 2 2 1 3 3 1 0 4
0 2 2
1 3 3
1 0 4
0

⍝ empty fleet can be valid if the board contains no ships
      (3 3 ValidBoard ⍬) □←3 3⍴0
0 0 0
0 0 0
0 0 0
1

⍝ board shape does not match the specified shape
      (4 3 ValidBoard 2 3 2 1) □←3 3⍴2 2 2 1 3 3 1 0 4
2 2 2
1 3 3
1 0 4
0
```

examples continue...

```
⍝ ship 3 is not contiguous
      (3 3 ValidBoard 2 3 2 1) ⎕←3 3⍴2 2 2 3 1 3 0 1 4
2 2 2
3 1 3
0 1 4
0

⍝ can't have ships on an empty ocean
      (0 0 ValidBoard 2 3 2 1) 0 0⍴0
0

⍝ Zen Battleship – empty fleet on an empty ocean
      (0 0 ValidBoard ⍬) 0 0⍴0
1
```

For the second task, we're going to change things a bit by allowing ships to be placed diagonally as well. We know this stretches the laws of physics a bit because ships don't dynamically change length depending on their direction, but play along with us on this one... However, there's a limit to the laws of physics we're willing to violate, for example, ships still must not intersect:

```
      (2 2 ValidBoard2 2 2) ⎕←2 2⍴1 2 2 1
1 2
2 1
0
```

Task 2: Write a dyadic operator named `ValidBoard2` that has syntax identical to `ValidBoard` but which allows ships to be placed on diagonals.

Examples

```
      ⎕←board←3 3⍴2 2 2 1 3 3 0 1 4 ⍝ ship 1 is diagonal

2 2 2
1 3 3
0 1 4
      (3 3 ValidBoard 2 3 2 1) board
0
      (3 3 ValidBoard2 2 3 2 1) board
1

      (3 3 ValidBoard2 1 2 3 2 1) ⎕←3 3⍴1 2 3 2 3 4 3 4 5
1 2 3
2 3 4
3 4 5
1

      (3 3 ValidBoard2 3 2 2) ⎕←3 3⍴1 1 1 0 2 0 2 3 3
1 1 1
0 2 0
2 3 3
1

      (2 2 ValidBoard2 2 2) ⎕←2 2 ⍴1 2 2 1 ⍝ ships may not intersect
1 2
2 1
0

⍝ Zen Battleship - empty fleet on an empty ocean
      (0 0 ValidBoard2 ⍬) 0 0⍴0
1
```

7: Subsequence-ially 🧬 (2 tasks)

These tasks are inspired by two problems from the excellent Bioinformatics website rosalind.info.

A subsequence of a string is a collection of symbols contained in order (though not necessarily contiguously) within the string. For instance, 'RTPN' is a subsequence of "THE **R**AIN ON **T**HE **P**LAI**N** IN SPAIN". The indices of a subsequence are the positions in the string where the symbols of the subsequence appear. For the previous example, the indices can be represented by `(5 13 17 21)`. Note that a subsequence can have multiple collections of indices. For the previous example `(5 13 17 24)`, `(5 13 17 30)` and `(5 13 27 30)` are also valid collections of indices.

The inspiration for the first task originates from Finding a Spliced Motif. Note that this task varies slightly from the original problem in that we expect your solution to handle empty arguments.

Task 1: Write a dyadic function named `sseq` that:

- has a right argument `string` that is a character scalar or vector of up to 1000 elements
- has a left argument `sub` that is also a character scalar or vector of up to 1000 elements
- returns a 2-element vector where:
  - the first element is 1 or 0 indicating whether `sub` is a subsequence of `string` (1) or not (0)
  - the second element is one of the collections of indices for `sub` if it is a subsequence of `string`, otherwise `0`

You can download sseqData.txt, which contains a sample dataset, from rosalind.info and use this file to test your solution on a larger dataset than those provided in the examples below. To use this file, substitute `{your_foldername}` with the location where you've saved the downloaded file:

```
(string sub)←→⎕NGET '{your_foldername}/sseqData.txt' 1
string sseq sub
```

Examples using `]Box on`

```
      'ACGTACGTGACG' sseq 'GTA' ⍝ note: 3 4 10, 3 8 10 or 7 8 10 would also be acceptable
```

1	3 4 5

```
      'ACGTACGTGACG' sseq 'TTT'
```

0	

```
      'ACGTACGTGACG' sseq ''
```

1	

```
      '' sseq 'TTT' ⍝ returns empty
```

0	

```
      'T' sseq 'T'
```

1	1

The inspiration for this second task originates from Finding a Shared Spliced Motif. Note that this task varies slightly from the original problem in that we expect your solution to handle empty arguments.

A string *u* is a common subsequence of strings *s* and *t* if the symbols of *u* appear in order as a subsequence of both *s* and *t*. For example, "ACTG" is a common subsequence of "AACCTTGG" and "ACACTGTGA".

Task 2: Write a dyadic function named `lcsq` that:

- has a right argument `s` that is a character scalar or vector
- has a left argument `t` that is also a character scalar or vector
- returns a character vector that is a longest common subsequence of `s` and `t`. There might be more than one longest common subsequence; in this situation you need return only one.

You can download sseqData.txt, which contains a sample dataset, from rosalind.info and use this file to test your solution on a larger dataset than those provided in the examples below. The verified length of the longest common subsequence for this sample dataset is 619. To use this file, substitute `{your_foldername}` with the location where you've saved the downloaded file:

```
(s t)←→⎕NGET '{your_foldername}/lcsqData.txt' 1
      t lcsq s
```

Examples

```
      'AACCTTGG' lcsq 'ACACTGTGA' ⍝ AACTTG, ACCTTG, ACCTGG are also valid results
AACTGG
      'ACGTACGTGACG' lcsq '' ⍝ returns empty

      'ACGTACGTGACG' lcsq 'XYZZYX' ⍝ returns empty
```