DYALOG

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 comprises 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 task description contains one or more examples. If applicable, the judging committee could submit your solutions to additional testing beyond the specific example solutions.

Your solutions will be tested in the default Dyalog environment using `(⎕IO ⎕ML)←1`. Your code may employ a different, localized, setting for either of these if necessary.

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←{⍺+⍵}
```

The functions specified in the problem descriptions must be tradfns or dfns. You are free to use tacit definitions inside these, and as helper-functions. Any of the following would be valid solutions:

```
∇ r←a Plus b
  r←a(⊣+⊢)b
∇
```

```
Plus←{⍺(⊣+⊢)⍵}
```

```
Plus←{⍺ TacitPlus ⍵}
TacitPlus←⊣+⊢
```

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 display output to the session.
- Pay attention to any additional judging criteria that may be stated in an individual problem's description.

- Be aware that the examples serve to provide basic guidance and validation for your solutions and are not intended to be an complete exposition of all possible edge cases; the judging committee will submit your solutions to additional test cases.
- Be aware that the order that the problems are presented in does not necessarily reflect their level of difficulty – if you find yourself stuck then you might find the next problem more straightforward!

1: Sub-space Journey 📦 (3 tasks)

All the tasks in this problem are related to creating and detecting sub-spaces. APL's multidimensional capabilities are well-suited to address problems of this sort.

Task 1: Write a function named `runs` that:

- takes a non-negative integer scalar left argument *n* that which specifies the length of the result.
- takes a 2-column integer matrix right argument in which:
  - column 1 is a positive integer representing the index in the result where a run of 1s will start.
  - column 2 is a non-negative integer representing the length of the run (number of consecutive 1s) starting at the index indicated by column 1.
- returns a Boolean vector of length *n* comprising runs of consecutive 1s as indicated by the right argument.

Note: Any indices implied by the right argument that exceed the shape of the result should be ignored.

Examples

```
      10 runs 1 2⍴3 4
0 0 1 1 1 1 0 0 0 0

      10 runs 2 2⍴3 4 8 6 ⍝ a the result must be of length n, even if a run
is specified beyond n
0 0 1 1 1 1 0 1 1 1

      15 runs 0 2⍴ 5 3 ⍝ no runs here
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

      15 runs 2 2⍴3 6 5 7 ⍝ overlapping runs are permitted
0 0 1 1 1 1 1 1 1 1 1 0 0 0 0

      10 runs 1 2⍴6 0 ⍝ 0-length runs are permitted
0 0 0 0 0 0 0 0 0 0

      0 runs 2 2⍴2 3 5 6 ⍝ returns a 0-length (empty) vector
```

Task 2: Having written `runs` in Task 1, let's complicate things a bit...

Write a function named `fill` that:

- takes a non-negative integer scalar or non-empty vector left argument `size` that specifies the shape of the result. We'll also specify `rank←≢size`.
- takes a *`(2×rank)`*-column integer matrix `subspaces` where the first `rank` columns specify the index where a sub-space starts and the last `rank` columns specify the shape of the sub-space.
  For example, a row containing 2 1 3 6 4 5 describes a 6×4×5 sub-space starting at index (2,1,3) in a 3-dimensional array.
- returns an integer array of the shape specified in `size`, where each sub-space is filled with the row index in `subspaces` for that sub-space. Positions not in any described sub-space should be 0.

Note: Any indices implied by the right argument that exceed the shape of the result should be ignored.

Examples

```
      10 fill 1 2⍴3 4
0 0 1 1 1 1 0 0 0 0

      15 fill 2 2⍴3 6 5 7 ⍝ overlapping fills are permitted
0 0 1 1 2 2 2 2 2 2 2 0 0 0 0


⍝ a terrible way to implement Phase 1's "Pyramid Scheme" problem
      ⊢spaces←5 4⍴∊2/¨(⍳5),¨(⌽¯1+2×⍳5)
1 1 9 9
2 2 7 7
3 3 5 5
4 4 3 3
5 5 1 1

      9 9 fill spaces
1 1 1 1 1 1 1 1 1
1 2 2 2 2 2 2 2 1
1 2 3 3 3 3 3 2 1
1 2 3 4 4 4 3 2 1
1 2 3 4 5 4 3 2 1
1 2 3 4 4 4 3 2 1
1 2 3 3 3 3 3 2 1
1 2 2 2 2 2 2 2 1
1 1 1 1 1 1 1 1 1

      4 4 4 fill 3 6⍴6/1 2 3
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

0 0 0 0
0 2 2 0
0 2 2 0
0 0 0 0

0 0 0 0
0 2 2 0
0 2 3 3
0 0 3 3

0 0 0 0
0 0 0 0
0 0 3 3
0 0 3 3
```

Task 3: Now let's go the other way and write a function to describe the sub-spaces in an n-dimensional space...

Write a function **`subspaces`** with syntax:

```
      result ← subspaces space
```

This function:

- takes a non-negative, non-scalar, integer array `space` that has the following characteristics:
  - sub-spaces are defined as rectangular blocks of positive integers.
  - the positive integers that identify the sub-spaces start counting from 1 and are consecutive.
  - the total number of sub-spaces in `space` is arbitrary.
  - sub-spaces are completely within `space`.
  - sub-spaces do not overlap.
- returns an integer matrix with *`2×rank`* columns `rank` is the number of dimensions in `space`) where:
  - each row describes a sub-space.
  - rows are ordered by the positive integer that identifies the sub-space.
  - the first `rank` columns represent the first index of the sub-space.
  - the second `rank` columns represent the shape of the sub-space.

Examples

(annotated)

```
      subspaces 0 2 2 0 1 1 1 1 0 3
 5 4  ⍝ the 1s start at index 5 and have length 4
 2 2  ⍝ the 2s start at index 2 and have length 2
10 1  ⍝ the 3s start at index 10 and have length 1

      ⊢space←↑(⊢⍴⍨⊢,⊢)¨3 2 1
3 3 3
3 3 3
3 3 3

2 2 0
2 2 0
0 0 0

1 0 0
0 0 0
0 0 0
      subspaces space
3 1 1 1 1 1 ⍝ the 1s start at position 3 1 1 and span 1 plane, 1 row, and 1
column
2 1 1 1 2 2 ⍝ the 2s start at position 2 1 1 and span 1 plane, 2 rows, and 2
columns
1 1 1 1 3 3 ⍝ the 3s start at position 1 1 1 and span 1 plane, 3 rows, and 3
columns

      ⍴subspaces 5 4 3 2⍴0 ⍝ if no sub-spaces, the result should still have
the proper number of columns
0 8

      ⊢space←((3 3⍴5),(2 2⍴2)⍪1)⍪(2 2⍴4),2 3⍴3
5 5 5 2 2
5 5 5 2 2
5 5 5 1 1
4 4 3 3 3
4 4 3 3 3
      subspaces space
3 4 1 2
1 4 2 2
4 3 2 3
4 1 2 2
1 1 3 3
```

2: Reshaping Reshape ⍴ (2 tasks)

Task 1: Write a function named `reshape` that behaves like the primitive *reshape* function `X⍴Y` except that elements in the left argument can be negative integers, which indicates that the data is reversed along the corresponding axis. Your function `reshape` should:

- take an integer vector or scalar left argument named `dims` that represents the length and direction of each axis in the result.
- take an array right argument named `data`
- return an array of shape `(|dims)`, which is the same as `dims⍴data` except that the elements along the nth axis are reversed if `dim[n]<0`.

Note: Your function `reshape` is subject to the same limits as `⍴`, for example, it has a maximum of 15 axes.

Examples

```
      10 reshape ⍳4
1 2 3 4 1 2 3 4 1 2

      ¯10 reshape ⍳4
2 1 4 3 2 1 4 3 2 1

¯4 4 reshape ⎕A ⍝ rows are reversed
MNOP
IJKL
EFGH
ABCD


     ¯4 ¯4 reshape ⎕A ⍝ rows and columns are reversed
PONM
LKJI
HGFE
DCBA
```

```
      2 ¯2 ¯3 4  reshape ⍳48 ⍝ planes and rows are reversed, hyperplanes and
columns are not
  21 22 23 24
  17 18 19 20
  13 14 15 16

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


  45 46 47 48
  41 42 43 44
  37 38 39 40

  33 34 35 36
  29 30 31 32
  25 26 27 28


      2 ¯2 reshape'Adam' 'Brian' 'Michael' 'Morten'
 ┌──────┬───────┐
 │Brian │Adam   │
 ├──────┼───────┤
 │Morten│Michael│
 └──────┴───────┘

      ⍬ reshape 5 3 1 ⍝ returns scalar 5
5
```

Task 2: The primitive *reshape* function `⍴` repeats or truncate elements of the right argument as necessary to match the shape described by the left argument. Hence `4⍴1 2` returns `1 2 1 2`.

In this task we're going to extend this behavior by writing a function named **`reshape2`** that, in addition to doing what **`reshape`** above does, also:

- allows, at most, one element of the left argument (`dims`) to be one of 6 "special" values (`0.5 1.5 2.5 ¯0.5 ¯1.5 ¯2.5`) that affects the length of the corresponding axis based on the other dimensions and the length of the data. These values are interpreted as follows:
    - `0.5` means truncate the data if it doesn't fill a complete corresponding axis.
    - `1.5` means repeat the data, if necessary, to fill out a complete corresponding axis.
    - `2.5` means pad the data with an appropriate prototype, if necessary, to fill out a complete corresponding axis; similar to how the primitive function *take* `X↑Y` behaves.
    - The other three values (`¯0.5 ¯1.5 and ¯2.5`) have the same intepretation as their positive counterparts but also reverse the elements in the corresponding dimension.

    - If `dims` is a singleton and is one of the "special" values, then **`reshape2`** should return the elements of the right argument as a vector (reversed if `dims` is negative).

Note: Your function **`reshape2`** is subject to the same limits as `⍴`, for example, it has a maximum of 15 axes.

Your function should essentially:

1. truncate, repeat, or pad the data to make it conform to the shape derived from `dims`.
2. reverse, if necessary, along the approriate axes, as specified by `dims`.

Examples

```
      0.5 4 reshape2 ⍳10 ⍝ 0.5 truncates the data
1 2 3 4
5 6 7 8

      4 0.5 reshape2 ⍳10
1 2
3 4
5 6
7 8


      1.5 4 reshape2 ⍳10 ⍝ 1.5 repeats the data
1  2 3 4
5  6 7 8
9 10 1 2

      2.5 4 reshape2 ⍳10 ⍝ 2.5 pads the data
1  2 3 4
5  6 7 8
9 10 0 0

      ¯4 ¯2.5 reshape2 13↑⎕A ⍝ 4 rows with padding and the rows and columns
reversed
   M
LKJI
HGFE
DCBA
```

```
      ¯3 ¯2.5 reshape2 'brian' 'adam' 'morten' 'michael'
┌───────┬──────┐
│       │      │
├───────┼──────┤
│michael│morten│
├───────┼──────┤
│adam   │brian │
└───────┴──────┘

      ρ⎕←⍬ reshape2 'brian' 'adam' 'morten' 'micheal' ⍝ result is a scalar
┌─────┐
│brian│
└─────┘

      2.5 3 4 reshape2 ⍳21 ⍝ 3 axes
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21  0  0  0


      2 2.5 3 4 reshape2 ⍳26 ⍝ 4 axes
1  2  3  4
5  6  7  8
9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24


25 26  0  0
 0  0  0  0
 0  0  0  0

 0  0  0  0
 0  0  0  0
 0  0  0  0
```

```
      ¯2 2.5 3 ¯4 reshape2 ⍳26 ⍝ 4 axes with reversal
 0  0 26 25
 0  0  0  0
 0  0  0  0

 0  0  0  0
 0  0  0  0
 0  0  0  0


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

16 15 14 13
20 19 18 17
24 23 22 21
```

3: Meetings of the Minds (3 tasks)

Each year, Dyalog Ltd sponsors a user meeting where Dyalog staff and users have an opportunity to present topics of interest and interact with one another. Due to the impact of COVID-19, the meetings for 2020 and 2021 were conducted virtually using Zoom. People registered ahead of time and could then sign-on and attend, virtually, any or all sessions. There were two partial days of sessions in each year.

After the conclusion of the user meeting, Zoom sent Dyalog Ltd a CSV file containing information including when each attendee joined or left the meeting. The tasks in this problem involve analyzing this information. There are two files that you will need for this problem:

- Attendees.csv contains attendee information for all four days of the 2020 and 2021 Dyalog user meetings. This is a sub-set of the actual data sent to Dyalog Ltd by Zoom. All personally-identifiable information has been removed. The attendee names found in the files are ficticious and were randomly generated – no association with any real person is intended or should be inferred. This file has 4 columns:
  - Attendee – the ficticious attendee name
  - Join Time – a character vector representing the time the attendee joined the meeting
  - Leave Time – a character vector representing the time the attendee left the meeting
  - Date – a character vector representing the date for the entry

  Note: Some rows have join and leave times of '--' meaning the attendee registered for the user meeting but did not attend any sessions that day. When we combined the data for all four days into a single file, we added the Date column to indicate which date an attendee did not attend.

- Schedule.csv contains the user meeting schedules for all four days. This file has 4 columns:
  - Session – the session identifier
  - Title – the session title
  - Start Time – a character vector representing the start time of the session
  - End Time – a character vector representing the end time of the session

  You should use the `⎕CSV` system function to import the CSV data into the workspace as follows:

```
      attendees←⊃⎕CSV 'your-path-here/Attendees.csv' '' ⍬ 1
      schedule←⊃⎕CSV 'your-path-here/Schedule.csv' '' ⍬ 1
```

Notes:

- For the purpose of describing the tasks for this problem, we will be using matrices named *`attendees`* and *`schedule`* as defined above.
- You should replace *your-path-here* above with the path to the folder into which you downloaded the CSV files.
- When properly read, `attendees` and `schedule` should have 1446 and 48 rows respectively.
- The *Date-time* system function `⎕DT` could be helpful for this problem.

Task 1: Write a dyadic function **`Attended`** with syntax:

```
      result←attendees Attended schedule
```

where **`Attended`**:

- takes the matrix *`attendees`* (or a sub-set of its rows) as its left argument.
- takes the matrix *`schedule`* as its right argument.
- returns a 435×48 Boolean matrix in which:
  - the rows represent the list of unique attendees sorted alphabetically (`uattendees`).
    (Hint: Aaden Webster and Zoe Bright are the first and last attendees alphabetically).
  - the columns represent each session in `schedule`.
  - a *1* in position `[i;j]` indicates that `uattendees[i]` attended *`schedule[j;]`*. An attendee is considered to have attended a session if they were present for at least half of the time (in minutes) that the session was being held. We don't count the "leave time" minute or the "session end" minute. For example: for a session that runs from 14:00-14:30, if an attendee joins at 14:00 and leaves at 14:14, they would be considered to have **not** attended that session whereas if they left at 14:15, they would be considered to have attended.

Examples:

```
      who←'Zaria Matthews' 'Kathryn Stafford'  'Marlene Lin'  'Kayleigh 
Rodgers'

      who[⍋who],(attendees⌿⍨attendees[;1]∊who) Attended schedule ⍝ 
remember, the result of Attended is sorted by attendee name
 Kathryn Stafford  1 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 Kayleigh Rodgers  0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 0 0 0 0 0 
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
 Marlene Lin       0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 Zaria Matthews    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

      attendees⌿⍨attendees[;1]∊who[1]
Zaria Matthews  11/8/2021 16:12  11/8/2021 16:25  11/8/2021
Zaria Matthews  11/9/2021 14:04  11/9/2021 14:34  11/9/2021
```

To help you validate your solution, we've included two files attendeeTotals.json and sessionTotals.json containing the row and column totals of `attendees Attended schedule`. To use them to validate your work:

```
      map←attendees Attended schedule
      (⎕JSON ⊃⎕NGET 'your-path-here/sessionTotals.json')≡+⌿map
1
      (⎕JSON ⊃⎕NGET 'your-path-here/attendeeTotals.json')≡+/map
1
```

Task 2: Write a dyadic function **ShowedUp** with syntax:

```
      result←attendees ShowedUp schedule
```

where **ShowedUp**:

- takes the matrix `attendees` as its left argument.
- takes the matrix `schedule` as its right argument.
- returns a 2×5 integer matrix where the columns contain:
  - `[;1]` – the year of the user meeting
  - `[;2]` – the number of people who registered for that year
  - `[;3]` – the number of people who attended the first day of that year
  - `[;4]` – the number of people who attended the second day of that year
  - `[;5]` – the number of people who registered but did not attend either day that year
- uses the same criteria as **Attended** for determining attendance.
  Using the `'--'` entries in *`attendees`* is not sufficient to determine whether a user attended on a given day. This is demonstrated by Zaria Matthews' entry in the previous example – although she did join on 8 November 2021, the time was not sufficient to count as having attended a session.

Example:

As there's only one correct answer, we'll provide it here for you to validate your work.

```
      attendees ShowedUp schedule
      2020 298 205 184 78
      2021 238 166 149 54
```

Task 3: Write a dyadic function `Popular` with syntax:

```
      result←map Popular schedule
```

where `Popular`:

- takes the matrix `map` (the result of Task 1) as its left argument.
- takes the matrix `schedule` as its right argument.
- returns a 2-element vector (one element for each year) where each element is a 2-column matrix in which:
  - `[;1]` is the number of attendees for a session.
  - `[;2]` is the session title.
  - the matrix should be sorted by descending popularity each day.
  - the matrix should not include any sessions that are break periods.

Example:

As there's only one correct answer, we'll provide it here for you to validate your work.

```
      result←map Popular schedule
      ⍴result
2
      ⍴¨result
 16 2  19 2

      ⍪' '⍪¨(attendees Attended schedule) Popular schedule

178  The Road Ahead
169  Multi-line Input and Scripting
164  The .NET Core Bridge
160  Welcome to Dyalog '20
160  Dyalog's Docker Containers
159  Array Notation RC1
157  Time Travel Debugging and Statistical Distributions
151  Reworking Mastering Dyalog APL
146  How I Won the APL Problem Solving Competition
143  How I Won the APL Problem Solving Competition - Introduction
134  Rational Arithmetic
128  Building Applications using qWC (⎕WC) on the Web
127  Tracing Hanneke Vrome Numerically
114  APL Online!
113  Closing session
 76  Open Discussion

131  Scripting in Dyalog v18.2
129  The Road Ahead
122  APL in the Driver's Seat
120  Here's The Plan: Learn APL, and Write a Book About It
118  Welcome to Dyalog '21
115  Dado (Dyalog APL Development Operations)
112  Support for Statistical Distributions in Dyalog v18.2
112  Link v3.0
109  The 2021 APL Problem Solving Competition - Introduction
108  Extending the Domain of the Probability Operator in TamStat
107  The 2021 APL Problem Solving Competition - Runner-Up's Presentation
102  Python + APL = Py'n'APL
101  Packaging Dyalog Tools
 96  Highlights of Dyalog v18.2
 95  ⎕JSON Table Support
 84  APL Media Update 2021
 73  The Array Cast (live podcast recording)
 71  Closing session and open discussion
 26  Open Discussion
```

4: Instant-Runoff Voting (2 tasks)

Instant-runoff voting (IRV) is a type of ranked voting that can be used when there are more than two candidates running for a single seat. In this system, voters rank the candidates in order of preference. The votes for each candidate are tallied and if a candidate has a majority, they win. If no candidate receives more than half of the votes, then the candidate(s) who received the fewest votes are dropped from consideration. The voters who selected the defeated candidates then have their votes added to the totals of their next choice. This process continues until a candidate has more than half of the votes. Ballots on which all of a voter's ranked candidates are eliminated become inactive. IRV is used in a number of countries, provinces, states and municipalities.

Task 1: Write a dyadic function named **`Ballot`** that generates a sample ballot by randomly assigning candidate rankings and has syntax:

```
      r←candidates Ballot voters
```

where **`Ballot`**:

- takes a positive integer right argument representing the total number of voters.
- takes a non-zero integer left argument representing the number of candidates where:
  - the number of candidates is `|candidates`.
  - if `candidates>0`, each ballot entry must rank all candidates.
  - if `candidates<0`, at least one candidate must be ranked.
- returns a 2-column matrix in which:
  - `[;2]` contains unique vectors of length `|candidates`, where the ith element is the ranking for candidate i.
  - `[;1]` is the number of votes matching that ranking combination.

  NOTE: Every valid result should have a non-zero probability of appearing.

Examples:

Let's create sample ballots for 150 voters and 3 candidates – Bob, Mary, and Larry. Your results will likely be different because they are random.

```
      b←3 Ballot 150 ⍝ generate 150 voter rankings for 3 candidates
      b
┌──┬─────┐
│22│3 1 2│
├──┼─────┤
│25│2 1 3│
├──┼─────┤
│31│3 2 1│
├──┼─────┤
│19│1 3 2│
├──┼─────┤
│33│2 3 1│
├──┼─────┤
│20│1 2 3│
└──┴─────┘
            ('#',b[;1]),'Bob' 'Mary' 'Larry' ⍪↑b[;2]
 #  Bob  Mary  Larry
22    3     1      2  ⍝ 22 people ranked Mary first, Larry second, and Bob
third
25    2     1      3  ⍝ 25 people ranked Mary first, Bob second, and Larry
third
31    3     2      1  ⍝ 31 people ranked Larry first, Mary second, and Bob
third
19    1     3      2  ⍝ 19 people ranked Bob first, Larry second, and Mary
third
33    2     3      1  ⍝ 33 people ranked Larry first, Bob second, and Mary
third
20    1     2      3  ⍝ 20 people ranked Bob first, Mary second, Larry third

      b2←¯3 Ballot 150
      ('#',b2[;1]),'Bob' 'Mary' 'Larry' ⍪↑b2[;2]
 #  Bob  Mary  Larry
10    1     2      3  ⍝ 10 people ranked Bob first, Mary second, Lary third
11    2     1      0  ⍝ 11 people ranked Mary first, Bob second and no one
third
10    2     1      3  ⍝ you get the idea...
 9    1     3      2
13    0     1      2
14    0     0      1
12    3     1      2
16    1     0      0
21    0     1      0
 6    1     2      0
 6    2     0      1
 8    3     2      1
 6    0     2      1
 4    1     0      2
 4    2     3      1

       1 Ballot 150 ⍝ uncontested election!
150  1
```

Task 2: Write a monadic function named **`IRV`** with syntax:

```
      r←IRV ballot
```

where **`IRV`**:

- takes a right argument in the same format as the result returned by **`Ballot`**.
- returns a vector of matrices containing each round of tallying, followed by the candidate number of the winner, if there is one (meaning the election didn't end in a tie).

Examples

```
      IRV b ⍝ using b from the example above
┌────┬────┬─┐
│1 39│2 67│3│
│2 47│3 83│ │
│3 64│    │ │
└────┴────┴─┘
      IRV b2 ⍝ using b2 from the example above
┌────┬────┬─┐
│1 45│1 55│2│
│2 67│2 81│ │
│3 38│    │ │
└────┴────┴─┘
      ⎕←b3←300 200 100 50 50 100,⍪↓6 4⍴1 0 2 0 0 1 0 2 2 0 0 1 0 2 1 0 0 2 0
1 3 2 1 0
┌───┬───────┐
│300│1 0 2 0│
├───┼───────┤
│200│0 1 0 2│
├───┼───────┤
│100│2 0 0 1│
├───┼───────┤
│50 │0 2 1 0│
├───┼───────┤
│50 │0 2 0 1│
├───┼───────┤
│100│3 2 1 0│
└───┴───────┘
      IRV b3 ⍝ end in a tie, so there is no trailing winning candidate
element
┌─────┬─────┐
│1 300│1 400│
│2 200│2 400│
│3 150│     │
│4 150│     │
└─────┴─────┘

      IRV ¯10 Ballot 200000 ⍝ your results will likely be different
┌────────┬────────┬────────┬────────┬────────┬────────┬────────┬────────┬─────────

│ 1 19831│ 1 21812│ 2 24359│ 2 27375│ 3 31627│ 3 36880│ 3 43999│ 3 54868│ 5
73329│10│
│ 2 20023│ 2 21921│ 3 24793│ 3 27784│ 4 31364│ 4 36690│ 4 43968│ 5 55101│10
73790│  │
│ 3 20304│ 3 22308│ 4 24440│ 4 27491│ 5 31603│ 5 36814│ 5 44206│10 55328│
│  │
│ 4 19989│ 4 22005│ 5 24605│ 5 27620│ 6 31267│ 9 36428│10 44090│        │
│  │
│ 5 20142│ 5 22162│ 6 24416│ 6 27418│ 9 31351│10 36679│        │        │
│  │
│ 6 20015│ 6 22035│ 7 24325│ 9 27423│10 31512│        │        │        │
│  │
│ 7 19864│ 7 21873│ 9 24365│10 27515│        │        │        │        │
│  │
│ 8 19772│ 9 21967│10 24401│        │        │        │        │        │
│  │
```

9 20001	10 21992							
10 20059								

5: Base$_{85}$ (1 task)

Base85, also known as Ascii85, is a binary-to-text encoding that is more efficient than Base64. Base85 uses five ASCII characters to represent four bytes of binary data (a 25% size increase), whereas Base64 uses four characters to represent three bytes of data (a 33% size increase). Your task here are is to write a single function to encode a series of integers in the range [0,256] to a Base85 string and vice versa.

In theory, any set of 85 unique, single-byte, characters could be used as the encoding character set. Several "standard" variations exist. Two of them are "Original" and "Z85":

- The Original character set uses the ASCII characters 33-117 ('!'-'u'):
```
      ⎕←Original←⎕UCS 32+⍳85
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstu
```
- Z85 uses the following character set:
```
Z85←'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ.-:+=^!/
&<>()[]{}@%$#'
```

There are also several Base85-encoding methods, some of which have special treatment for compressing data, or to preserve the length of the encoding input, or to add prefix or suffix information. To (hopefully) avoid confusion, we'll describe the steps to encode and decode here.

To Encode:
1. If the length of the input data is not a multiple of 4, pad it with 0s to make its length divisible by 4.
2. Convert the Step 1 result from base 256 to base 85.
3. Use the Step 2 result to index into the encoding character set.
4. Drop as many elements from the end of the Step 3 result as you added in Step 1.

To Decode:
1. If the length of the input data is not a multiple of 5, pad it with as many of the last character in the encoding character set as needed to make its length divisible by 5.
2. Convert the Step 1 result to its ordinal positions in the encoding character set.
3. Convert the Step 2 result from base 85 to base 256.
4. Drop as many elements from the end of the Step 3 result as you added in Step 1.

Base85-encoded data can include whitespace and line-break characters that might be used for formatting or other purposes. These characters should be ignored when decoding. This convention can be extended such that any character that is not an element of the encoding character set should be ignored.

Task 1: Write a dyadic function named `Base85` with syntax:

```
      result←variant Base85 data
```

where `Base85`:

- has a left argument `variant` that is a length-85 character vector representing a valid encoding character set.
- has a right argument `data` that is one of the following:
  1. a character vector representing a valid Base85-encoded string to be decoded.
  2. a numeric vector or scalar with values in the range [0,255] representing the bytes to be Base85-encoded.
- returns respectively:
  1. a numeric vector with values in the range [0,255] representing the decoded binary data.
  2. a character vector representing the Base85-encoded argument.

Examples:

```
      Original Base85 ⎕UCS 'Hello World'
87cURD]i,"Ebo7
      Z85 Base85 ⎕UCS 'Hello World'
nm=QNzY&b1A+]m
      Original Base85 0 0 0 0 0 0
!!!!!!!!
      Original Base85 8⍴'!'
0 0 0 0 0 0

⍝ Your function should round-trip properly
      'Hello World'≡⎕UCS Z85 Base85 Z85 Base85 ⎕UCS 'Hello World'
1
⍝ Or more compactly
      Z85 (Base85⍣2 ≡ ⊢) ⎕UCS 'Hello'
1
      ⎕UCS Original Base85 '7!W 3WD ⍴ eC1 ⌈ Y:eU' ⍝ remember to ignore
characters not in the encoding character set
Dyalog APL
```

6: It's a Date! 📅 (1 task)

Dyalog version 18.0 introduced two date-related features:

- `⎕DT` which converts date and time stamps between almost every imaginable format.
- `1200⌶` which formats Dyalog Date Numbers according to a specified pattern and upon which this problem is based.

The integral part of a Dyalog Date Number is an offset from day 0 (31 December 1899). The fractional part of a Dyalog Date Number represents the timestamp fraction of a day. For example, 12:00:00 is 0.5.

Task 1: Write a dyadic function named **`DDN`** (for Dyalog Date Number) with syntax:

```
    ddn←pattern DDN string
```

where **`DDN`**:

- has a character vector left argument `pattern` that represents a **valid** left argument (formatting pattern) to `1200⌶`. Due to the complexity of this problem, we will limit what `pattern` can contain as follows:
  - `pattern` can contain any of the patterns in the **Variations** column found in the `1200⌶` documentation, excluding the fractional seconds patterns (fractional seconds patterns are excluded due to the variation in precision across platforms).
  - No variable length numeric fields will be placed immediately next to another numeric field. For example, you will not encounter a pattern such as `'hYYm'` as some of the possible solutions to `'12012'` could be 1:12 in a year ending in '20', or 12:02 in a year ending in '01'.
  - The only alphabetic characters in `pattern` will be part of a variation pattern. There will be no alphanumeric constants in `pattern`.
  - There will be no quoted substrings in `pattern`.
- has a character vector right argument `string` that is the result of
  ```
      string ← ⊃pattern (1200⌶) ddn
  ```
  where `ddn` is a Dyalog Date Number, and in which:
  - all day names and abbreviations, month names and abbreviations, AM/PM designations, and ordinals use their default English values.
  - the only alphanumeric characters will be formatted elements of the date/time.
- returns a Dyalog Date Number `ddn` that would satisfy
  ```
      string ≡ ⊃pattern (1200⌶) ddn
  ```
  There will be more than one value for `ddn` that satisfies the requirement; you only need to return one value.

Notes:

- It will be helpful, but not necessary, to solve this problem using Dyalog version 18.0 (or later).
- Your solution should satisfy:
```
      string ≡ pattern (1200⌶) pattern DDN string
```

Examples:

```
      'Ddd, DD-Mmm-YYYY hh:mm:ss' DDN 'Thu, 17-Feb-2022 15:10:07'
44608.63203

      'MM/DD/YY tP:mm' DDN '02/17/22 3P:39'
44608.65215

      'Dddd' DDN 'Thursday' ⍝ any Thursday will suffice
43208
```