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 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 both Phase 2 problems to qualify for consideration for one of the top three student prizes or the non-student prize. You can write additional subfunctions in support of your solutions if necessary.

Each task description contains one or more examples. The judging committee can 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 can employ a different, localised, 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←{⍺+⍵}
```

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? For example, 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 are 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 a complete exposition of all possible edge cases; the judging committee will submit your solutions to additional test cases.

1: Bioinformatics🧬 (5 tasks)

All of the tasks in this problem set are derived from the excellent Rosalind.info bioinformatics website.

For most tasks, we provide at least one sample dataset consisting of an input file and a corresponding result file that has been verified on Rosalind.info. You can download these for your own testing purposes; the examples in this problem description assume the files have been downloaded into a folder named /tmp/ (if you download them to a different location, substitute the location you used). In addition to the sample datasets we provide in this problem description, you can also use the Rosalind website to validate your solutions.

The first four tasks in this problem will help you to build utilities to solve the last task.

Task 1: The origin for this task can be found at Transcribing DNA into RNA

DNA strings are formed from an alphabet containing 'A', 'C', 'G', and 'T'. RNA strings are formed from an alphabet containing 'A', 'C', 'G', and 'U'. The transcribed RNA string for a given DNA string is formed by replacing all occurrences of 'T' in the DNA string by 'U'.

Write a function named `rna` that:

- takes a character vector or scalar right argument representing a DNA string
- returns a character array of the same shape as the right argument representing the transcribed RNA string for the right argument

Sample dataset:

- Input: rosalind_rna_1_dataset.txt. Result: rosalind_rna_1_output.txt

Examples

```
      rna 'GATGGAACTTGACTACGTAAATT'
GAUGGAACUUGACUACGUAAAUU

      rna '' ⍝ returns an empty vector


      rna 'T' ⍝ returns a scalar
U

     ⍝ using the sample dataset - substitute your appropriate folder
name for /tmp/
     '/tmp/rosalind_rna_1_output.txt' ≡∘rna⍥{⊃⊃⎕NGET ⍵ 1}
'/tmp/rosalind_rna_1_dataset.txt'
1
```

Task 2: The origin for this task can be found at Complementing a Strand of DNA

In DNA strings, the symbols 'A' and 'T' are complements of each other, as are 'C' and 'G'. The reverse complement of a DNA string is formed by reversing the symbols and then taking the complement of each symbol.

Write a function named **`revc`** that:

- takes a character vector or scalar right argument representing a DNA string
- returns a character array of the same shape as the right argument representing the reverse complement of the right argument

Sample dataset:

- Input: rosalind_revc_1_dataset.txt. Result: rosalind_revc_1_output.txt

Examples

```
      revc 'AAAACCCGGT'
      ACCGGGTTTT

      revc '' ⍝ returns an empty vector


      revc 'T' ⍝ returns a scalar
A

      ⍝ using the sample dataset - substitute for /tmp/ as appropriate
      '/tmp/rosalind_revc_1_output.txt' ≡∘revc⍥{⊃⊃⎕NGET ⍵ 1}
'/tmp/rosalind_revc_1_dataset.txt'
1
```

Task 3: The origin for this task can be found at Translating RNA into Protein

The 20 commonly occurring amino acids are abbreviated by using 20 letters from the English alphabet. Protein strings are constructed from these 20 symbols. A *codon* is an RNA sequence of 3 nucleotides. There are 4 symbols used in RNA ('ACGU'), so there are 64 (`4*3`) possible codons. 61 codons specify amino acids and 3 are used as stop signals. The RNA codon table shows the mapping between the 64 codons and amino acids or stop signals:

UUU	F	CUU	L	AUU	I	GUU	V
UUC	F	CUC	L	AUC	I	GUC	V
UUA	L	CUA	L	AUA	I	GUA	V
UUG	L	CUG	L	AUG	M	GUG	V
UCU	S	CCU	P	ACU	T	GCU	A
UCC	S	CCC	P	ACC	T	GCC	A
UCA	S	CCA	P	ACA	T	GCA	A
UCG	S	CCG	P	ACG	T	GCG	A
UAU	Y	CAU	H	AAU	N	GAU	D
UAC	Y	CAC	H	AAC	N	GAC	D
UAA	Stop	CAA	Q	AAA	K	GAA	E
UAG	Stop	CAG	Q	AAG	K	GAG	E
UGU	C	CGU	R	AGU	S	GGU	G
UGC	C	CGC	R	AGC	S	GGC	G
UGA	Stop	CGA	R	AGA	R	GGA	G
UGG	W	CGG	R	AGG	R	GGG	G

Write a function named **`prot`** that:

- takes a non-empty character vector right argument representing an RNA string. If the length of the vector is not a multiple of 3, truncate the last 1 or 2 characters to make its length a multiple of 3.
- returns a character vector representing the protein string made by translating the codons into their corresponding amino acids.

Note: You can assume that the RNA string will contain at most one stop signal which, if present, will be at the end of the RNA string.

Sample dataset:

- Input: rosalind_prot_1_dataset.txt. Result: rosalind_prot_1_output.txt

Examples

```
      prot 'AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA'
MAMAPRTEINSTRING

      prot 'UUUAAAGG' ⍝ argument is length 8, use the first 6 elements
FK

      ⍝ using the sample dataset - substitute your appropriate folder name for /tmp/
      '/tmp/rosalind_prot_1_output.txt' (≡∘prot⍥{⊃⊃⎕NGET ⍵ 1}) '/tmp/rosalind_prot_1_dataset.txt'
1
```

Task 4: The origin for this task can be found at FASTA format

In bioinformatics, the FASTA format is a text-based format for representing genetic (RNA, DNA and protein) strings.

A FASTA file can contain one or more genetic string definitions, each of which is composed of some identifying information and the character representation of the genetic string itself. Specifically, each string comprises:

- an identifier line, which begins with the '>' character followed by an identifier for the string (and then, optionally, a space and additional descriptive text)
- additional lines containing the genetic string itself, up until either the end of the file or another line beginning with the '>' character (which marks the beginning of another genetic string definition).
  Note: The genetic string can be split across multiple contiguous lines in the file. This is typically done to maintain a line length of 80 or fewer characters.

Write a function named **`readFASTA`** that:

- takes a character vector right argument representing the name of a file containing data in FASTA format.
- returns a vector of 2-element vectors for each genetic string contained in the file in the order in which they appear in the file. Each sub-vector will contain:
  - the identifier for the genetic string
  - the genetic string itself

  Sample dataset:

  - Input: sampleFASTA.txt (contents appear below)

```
>Taxon1 Some description here
CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGCCTTCCC
TCCCACTAATAATTCTGAGG
>Taxon2
CCATCGGTAGCGCATCCTTAGTCCAATTAAGTCCCTATCCAGGCGCTCCGCCGAAGGTCT
ATATCCATTTGTCAGCAGACACGC
>Taxon3
CCACCCTCGTGGTATGGCTAGGCATTCAGGAACCGGAGAACGCTTCAGACCAGCCCGGAC
TGGGAACCTGCGGGCAGTAGGTGGAAT
```

Examples

```
      ⍝ using the sample dataset - substitute your appropriate folder
name for /tmp/
      ⍪ readFASTA '/tmp/sampleFASTA.txt'
┌──────────────────────────────────────────────────────────────────────
│┌──────┬───────────────────────────────────────────────────────────────
││Taxon1│CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGCCTTCCCTCCCACT
│└──────┴───────────────────────────────────────────────────────────────
├──────────────────────────────────────────────────────────────────────
│┌──────┬───────────────────────────────────────────────────────────────
││Taxon2│CCATCGGTAGCGCATCCTTAGTCCAATTAAGTCCCTATCCAGGCGCTCCGCCGAAGGTCTATATCCA
│└──────┴───────────────────────────────────────────────────────────────
├──────────────────────────────────────────────────────────────────────
│┌──────┬───────────────────────────────────────────────────────────────
││Taxon3│CCACCCTCGTGGTATGGCTAGGCATTCAGGAACCGGAGAACGCTTCAGACCAGCCCGGACTGGGAAC
│└──────┴───────────────────────────────────────────────────────────────
└──────────────────────────────────────────────────────────────────────
```

Task 5: The origin for this task can be found at Open Reading Frames

A reading frame is one of three possible ways to translate a given strand of DNA into amino acids depending on its starting position. For example, the DNA string ACGTACGT could be read as ACGTAC, CGTACG, and GTACGT ...

```
ACGTACGT
└────┘
 └────┘
  └────┘
```

...which can be translated to RNA strings ACGUAC, CGUACG, and GUACGU – these can then be mapped to amino acids TY, RT, and VR respectively.

A DNA string has 6 possible ways to be translated into amino acids - 3 reading frames for the DNA string itself, and 3 reading frames for the DNA string's reverse complement.

An open reading frame (ORF) is a sequence of DNA or RNA that is potentially able to encode a protein. An open reading frame starts with the start codon and ends with the stop codon without any other stop codons in between.

The start codon is the RNA string AUG, which is transcribed from the DNA string ATG. In the genetic code, the start codon corresponds to the amino acid methionine (M) and triggers the beginning of translation of a molecule of RNA into protein.

There are three possible stop codons; RNA strings UAG, UGA, and UAA, which are transcribed from DNA codons TAG, TGA, and TAA.

Write a function named **`orf`** that:

- takes a right argument that is the name of a file containing a single DNA string in FASTA format
- returns a vector of the distinct protein strings (character vectors) that can be translated from the ORFs of the DNA string. The strings can be in any order.

Sample datasets:

- Input: sampleORF.txt
- Input: rosalind_orf_1.txt Result: rosalind_orf_1_output.txt

Examples

```
      sort←(⊂⍋)⌷⊢

      ⍝ using the sample datasets - substitute your appropriate
folder name for /tmp/

      solution←'MLLGSFRLIPKETLIQVAGSSPCNLS' (,'M') 'MGMTPRLGLESLLE'
'MTPRLGLESLLE'
      solution ≡⍥sort orf '/tmp/sampleORF.txt'
1
      solution←' '~¨⍨⊃⎕NGET '/tmp/rosalind_orf_1_output.txt' 1 ⍝ read
solution as vector of vectors (removing blanks)
      solution ≡⍥sort orf '/tmp/rosalind_orf_1_dataset.txt'
1
```

2: Potpourri 🧺 (4 tasks)

The tasks in this problem set come from a variety of domains and sources.

Task 1: Identification Please

The origin for this task may be found at Check-digit calculation.

In the United States and Canada, a vehicle identification number (VIN) is a 17-character alphanumeric identifier used to uniquely identify a vehicle. VINs have a built-in check digit in the 9th position. The check digit is the result of a modulus 11 calculation computed from the other digits in the VIN.

The check-digit is calculated by:

1. Replacing any letters in the VIN with appropriate numerical counterparts as follows:

A: 1	B: 2	C: 3	D: 4	E: 5	F: 6	G: 7	H: 8	–
J: 1	K: 2	L: 3	M: 4	N: 5	–	P: 7	–	R: 9
–	S: 2	T: 3	U: 4	V: 5	W: 6	X: 7	Y: 8	Z: 9

   Note: the letters I, O, and Q are not allowed in a VIN.

2. Multiplying each number by a weight factor based on the position within the VIN:
   Note:

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Weight	8	7	6	5	4	3	2	10	0	9	8	7	6	5	4	3	2

3. Sum the products

4. Take the 11 modulo of the sum. The result, if not 10, is the check-digit. If the result is 10, use the letter X as the check-digit.

Write a function named **`vin`** that:

- takes a character vector right argument:
- returns a value that is dependent on the character vector right argument:
  - if the length of the argument is 16:
    takes the first 8 characters as the first 8 characters of a VIN and the last 8 characters as the last 8 characters of a VIN, computes the check-digit, and returns the complete VIN with the check-digit in the 9th position.
  - if the length of the argument is 17:
    computes the check-digit, compares it to the character in the 9th position, and

returns `1` if the computed check-digit matches the supplied one or `0` if it does not match.
  - if the length of the argument is not 16 or 17:
    returns `¯1`.
  - if the argument contains any characters that are not allowed in a VIN:
    returns `¯1`.

Examples

```
      vin '1M8GDM9AKP042788' ⍝ 16-character argument
1M8GDM9AXKP042788

      vin vin '1M8GDM9AKP042788'
1

      vin¨ '5GZCZ43D13S812715' 'SGZCZ43D13S812715'
1 0

      vin 'happy birthday'
¯1
```

Task 2: Get Your Versions in Order

Tatin is an APL package manager developed by the APL Team. Anyone can contribute packages to the public Tatin repository for use by those developing an APL application. A Tatin package is identified in the following format:

```
      groupName-packageName-major.minor.patch{-patchInfo}
```

For example "dyalog-jarvis-1.11.7" or "someOrg-somePackage-0.0.9-beta". Tatin also allows an optional build number field after patchInfo, but we'll ignore it for this problem.

Information about versions of packages that Dyalog has published can be retrieved by using the Tatin ListPackages user command:

```
     ]Tatin.ListPackages [tatin] -group=dyalog -noaggr
Registry: [tatin]
Group & Name
------------
dyalog-HttpCommand-5.1.10
dyalog-HttpCommand-5.1.11
dyalog-HttpCommand-5.1.12
dyalog-HttpCommand-5.1.13
dyalog-HttpCommand-5.1.14
dyalog-HttpCommand-5.1.5
dyalog-HttpCommand-5.1.6
dyalog-HttpCommand-5.1.7
dyalog-HttpCommand-5.1.8
dyalog-HttpCommand-5.1.9
dyalog-Jarvis-1.11.10
dyalog-Jarvis-1.11.5
dyalog-Jarvis-1.11.6
dyalog-Jarvis-1.11.7
dyalog-Jarvis-1.11.8
dyalog-Jarvis-1.11.9
```

The versions are sorted alphanumerically and not by major.minor.patch, meaning that HttpCommand 5.1.10 is listed before rather than after 5.1.9.

Write a function named `sortVersions` that:

- takes a character vector representing a single package identifier, or a vector of character vectors each representing a package identifier.
- returns a vector of character vectors of the package identifiers sorted by version number. If the right argument is a simple character vector representing a single package identifier, the result should be a 1-element vector of the enclosed right argument. If there is more than one unique groupName-packageName in the right argument, the entries should be grouped by groupName-packageName, but the order of groups is not significant.

Notes:

- While it might be useful, it is not necessary to have Tatin installed or available on your platform to solve this problem.
- The Tatin ListVersions function properly sorts by version number but cannot be used in this problem!
- The data for the example above can be downloaded from here and then brought into the workspace using:
  ```
        pkgs←⎕JSON ⊃⎕NGET '/downloads/pkgs.json'
  ```
  you will need to change "downloads" to the location where you downloaded pkgs.json

Examples

```
      ]Box on
Was OFF

      sortVersions 'a-b-1.2.1' 'a-b-1.2.10' 'a-b-1.2.2'
┌─────────┬─────────┬──────────┐
│a-b-1.2.1│a-b-1.2.2│a-b-1.2.10│
└─────────┴─────────┴──────────┘

      sortVersions 'a-b-1.2.10'
┌──────────┐
│a-b-1.2.10│
└──────────┘

      ]Box off
Was ON

      sortVersions 'a-b-1.2.3-test' 'a-b-1.2.3-prod' 'a-b-1.2.3-dev'
 a-b-1.2.3-dev  a-b-1.2.3-prod  a-b-1.2.3-test

      ⍪sortVersions ,pkgs  ⍝ pkgs is from the note above
 dyalog-HttpCommand-5.1.5
 dyalog-HttpCommand-5.1.6
 dyalog-HttpCommand-5.1.7
 dyalog-HttpCommand-5.1.8
 dyalog-HttpCommand-5.1.9
 dyalog-HttpCommand-5.1.10
 dyalog-HttpCommand-5.1.11
 dyalog-HttpCommand-5.1.12
 dyalog-HttpCommand-5.1.13
 dyalog-HttpCommand-5.1.14
 dyalog-Jarvis-1.11.5
 dyalog-Jarvis-1.11.6
 dyalog-Jarvis-1.11.7
 dyalog-Jarvis-1.11.8
 dyalog-Jarvis-1.11.9
 dyalog-Jarvis-1.11.10
```

Task 3: Time for a Change

I happened to make a purchase the other day and the change came to $1.32. The cashier gave me a one-dollar bill worth (1.00), three dimes (worth 0.30), and two pennies (worth 0.02) totaling $1.32. It struck me that this was a sub-optimal solution - because a one-dollar bill and 4 coins – a quarter (worth 0.25), a nickel (worth 0.05) and two pennies (worth 0.02) would total the same but with fewer coins.

It occurred to me that there are lots of combinations of denominations that can sum to $1.32 – after writing a solution to this task, I found there are 646 combinations.

Write a function named `makeChange` that:

- takes an ascendingly-sorted integer vector left argument (named `denom`) representing a set of denominations.
- takes a single integer right argument (named `amount`) representing an amount.
- returns an integer matrix (that we'll call `result`) with `(≢denom)` columns where each row represents a unique combination of elements of `denom` that total `amount`. There will be as many rows as unique combinations of `denom` that total `amount`. The order of the rows is not significant. The following will be true:
  - `amount ∧.= result +.× denom ⍝ all rows total amount`
  - `(≢result)=≢∪result ⍝ all rows are unique`

Some sample denominations (banknotes and coins):

- Euro: `euro←1 2 5 10 20 50 100 200 500 1000 5000 10000 20000`
  representing: 1c 2c 5c 10c 20c 50c €1 €2 €5 €10 €20 €50 €100 €200 (€1 = 100c)
- UK: `uk←1 2 5 10 20 50 100 200 500 1000 2000 5000`
  representing: 1p 2p 5p 10p 20p 50p £1 £2 £5 £10 £20 £50 (£1 = 100p)
- USA: `usa←1 5 10 25 50 100 500 1000 2000 5000 10000`
  representing: 1¢ 5¢ 10¢ 25¢ 50¢ $1 $5 $10 $20 $50 $100 ($1 = 100¢)

Notes:

- The number of combinations grows exponentially based on the number of elements in `denom` and the value of `amount`. You can, therefore, assume that all test cases that will be used to validate your solution will return fewer than 100,000 combinations.
- Although speed is not a primary criteria for judging this task, if two solutions are roughly equivalent in their correctness and use of array-oriented techniques, the faster solution will be judged more favourably.

Examples

```
      1 2 5 makeChange 5
0 0 1
1 2 0
3 1 0
5 0 0

      ⍴2 5 10 makeChange 3
0 3

      ≢usa makeChange 132
646

      ≢usa makeChange 200
2728

      ≢euro makeChange 200
73682
```

Task 4: Array Partitioning

In this task you will write a function to partition an array into a vector of sub-arrays. The *stencil* operator `⌺` can be used to perform operations on sub-arrays of an APL array. While you are under no obligation to use stencil in your solution, it may be a good place to start.

Write a function named `partition` that:

- takes a rank-1 or greater APL array right argument (named `array`)
- takes a left argument (named `spec`) that contains the specifications for extracting the sub-arrays:
  - the first element is a non-empty positive integer vector that has up to `≢⍴array` elements and specifies the shape of the sub-arrays
  - the second element is optional and, if it exists, it has the same shape as the first element and specifies the movement size for the partitioning. If not supplied or empty, the movement is assumed to be `(≢1⊃spec)⍴1`
  - the third element is also optional, has `≢⍴array` elements, and specifies the co-ordinates for the first partition. If not supplied or empty, the co-ordinates are assumed to be `(≢⍴array)⍴1`
    Note: If `spec` is a simple depth 0 or 1 array, you should enclose it and use it as the sub-array shape specification (the movement and co-ordinate elements should be defaulted).
- returns a vector of sub-arrays of `array`, each of the shape described in `spec[1;]`. If no sub-arrays matching `spec` exist, return an empty vector.

Examples

```
      3 partition ⍳10 ⍝ start with a simple case
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬──────┐
│1 2 3│2 3 4│3 4 5│4 5 6│5 6 7│6 7 8│7 8 9│8 9 10│
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴──────┘

      (,3)(,2) partition ⍳10 ⍝ movement is 2
┌─────┬─────┬─────┬─────┐
│1 2 3│3 4 5│5 6 7│7 8 9│
└─────┴─────┴─────┴─────┘

      (,3)(,2) 4 partition ⍳10 ⍝ start from the 4th position
┌─────┬─────┬──────┐
│4 5 6│6 7 8│8 9 10│
└─────┴─────┴──────┘

      5 5⍴⎕A
ABCDE
FGHIJ
KLMNO
PQRST
UVWXY

      3 partition 5 5⍴⎕A
┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
│ABC│BCD│CDE│FGH│GHI│HIJ│KLM│LMN│MNO│PQR│QRS│RST│UVW│VWX│WXY│
└───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘

      (2 2)(2 3) partition 5 5⍴⎕A
┌──┬──┬──┬──┐
│AB│DE│KL│NO│
│FG│IJ│PQ│ST│
└──┴──┴──┴──┘

      (2 2)(2 3)(2 1) partition 5 5⍴⎕A ⍝ 2×2 sub-arrays; movement 2
rows, 3 columns; starting at [2;1]
┌──┬──┬──┬──┐
│FG│IJ│PQ│ST│
│KL│NO│UV│XY│
└──┴──┴──┴──┘

      (3 2)(2 1)(2 2 2) partition 4 5 6⍴⍳120
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬───────┬───────┬───────┬
│38 39│39 40│40 41│41 42│68 69│69 70│70 71│71 72│ 98  99│ 99 100│100
101│101 102│
│44 45│45 46│46 47│47 48│74 75│75 76│76 77│77 78│104 105│105 106│106
107│107 108│
│50 51│51 52│52 53│53 54│80 81│81 82│82 83│83 84│110 111│111 112│112
113│113 114│
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴───────┴───────┴───────┴

      ⍴7 partition ⍳6 ⍝ can't get a 7-element sub-array from a 6-
element array
0

      ⍴⊃7 partition ⍳6 ⍝ but the shape of the prototype matches the
shape in spec
```

```
      1 partition ι5
┌─┬─┬─┬─┬─┐
│1│2│3│4│5│
└─┴─┴─┴─┴─┘

      ρ¨1 partition ι5
┌─┬─┬─┬─┬─┐
│1│1│1│1│1│
└─┴─┴─┴─┴─┘
```