Learning K programming, idiom by idiom

K2/K3 version

Eusebio C Rufian-Zilbermann

Learning K programming, idiom by idiom

K2/K3 version

Eusebio C Rufian-Zilbermann

Commentary and main text: Eusebio C Rufian-Zilbermann

List of Idioms, original K version by: Eugene McDonnell

Included Reference Documentation from: Arthur Whitney

Select ideas and comments by: Stevan Apter, John Earnest, Michal Wallace

Contents

Introduction to K

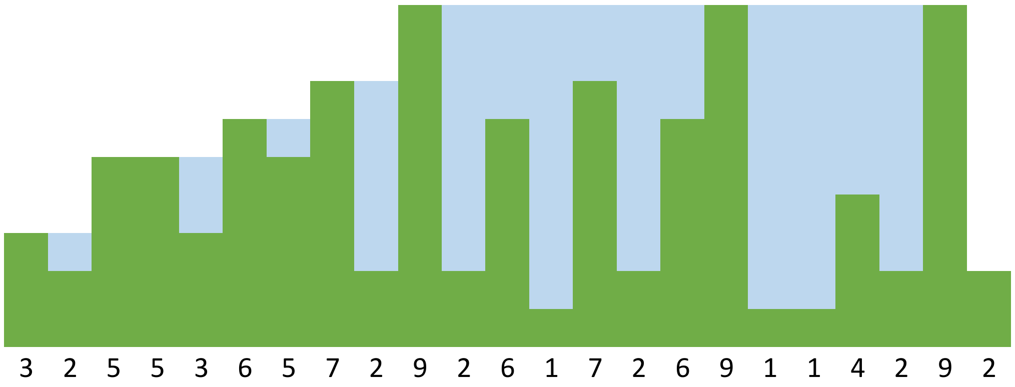
Why K?

A few years ago, I was asked at a job interview the “water reserves” problem. I would like to show you how K is a very expressive language that enables many high-level concepts to be translated directly into a program.

The problem statement is: You are working at an engineering company in charge of surveying a mountain range where you will be constructing a reservoir. You would like to know the depth that would be underwater in each of these areas when the water level is at its maximum capacity (the problem as asked in interviews may ask for data that can be easily derived from this, such as maximum water column height)

For the purposes of the exercise (so that it can fit in the time allowed for an interview) we are going to analyze the problem by representing the landscape as an East-West cross-section described by an array A of length N. Each element of A is the altitude of the area when there is no water at all. After the reservoir is full, the locations that don't have any higher location to their East and to their West will still be dry, but the lower locations for which it is possible to find higher locations both to their East and to their West will be holding water as high as their neighbors.

Here is a diagram of a random reservoir profile represented by the vector `3 2 5 5 3 6 5 7 2 9 2 6 1 7 2 6 9 1 1 4 2 9 2`. In this diagram we will also show how much water can be held



In this example the tallest columns of water have a height of 8, at locations where the altitude is 1 and have locations of altitude 9 to the East and to the West

The conceptual solution to this problem involves scanning the vector of heights and keeping track of the running maximum, this is the highest altitude to the West at each point, and then performing the same scan in reverse to find the highest altitude to the East at each point. The water level will be the smallest of the two at each point (if it were any higher, the water would fall off on either direction). We can then subtract the height from the water level to find the heights of the water columns and finally get the maximum from those.

K enables us to easily translate these high-level concepts into code:

1- Scanning an array x and finding the running maximum is expressed by `|\x` (read "max scan of `x`"), the highest altitude to the West.
2- Reversing an array is achieved by inserting a `|` ("reverse") before the array. `||\|x` ("reverse max scan of reverse `x`") will give us the highest altitude to the East.
3- Comparing two vectors element-wise is done with the `&` ("min") operation, this is the water level.
4- The element-wise subtraction is `-`. Adding parentheses to ensure correct sequence of evaluation, and brackets to make it a function that takes an argument, results in `{((|\x)&||\|x)-x}`. This is a function that returns the height of the column of water at every point.
5- Using this function, we can easily get derived results, for example for the maximum height we can use `|/` ("max over"), resulting in `|/{((|\x)&||\|x)-x}`.
6- For total water we can use `+/` ("sum over"), resulting in `+/{((|\x)&||\|x)-x}`.

We will stop at this point but if you feel more adventurous, you can think of how the problem would change if instead of a single 2D cross section the input is a 3D representation given by a collection of 2D cross sections.

This example shows some of the key characteristics of K:

- It is very terse. Solving problems in K often it feels much closer to writing mathematical equations than to writing programs in a language like C or Java. "Cramming" multiple operators in a single line is quite common.
- It is vector-oriented. You can subtract two values, two vectors and even two arrays or tensors, K will automatically extend, or "broadcast" it (to the corresponding element-wise) operation. It is also possible to mix vectors (arrays) and scalars (atoms) as operands.
- Operators and functions (collectively referred to as "verbs") can be accompanied by modifiers (called "adverbs") like "max scan" or "sum over". This is like the "reduce" operation in a map/reduce system.
- Primitives tend to be very short, just one or two characters in most cases.
- Functions are generic (not strongly typed). We didn't have to specify that we want an integer version of the running maximum. `|\x` will produce the running maximum whether it is applied to ints or floats.
- Functions can use default argument names. The names `x`, `y` and `z` are automatically assigned to the first, second and third arguments to a function, you don't need to name arguments unless you have more than 3. This is common practice for single-line functions.
- K does not follow PEMDAS operator precedence rules. The actual rules for parsing the language have some nuances but for most normal cases expressions are evaluated parenthesized elements first and then right-to-left.
- Most operators are overloaded with different meanings, primarily depending on whether they are applied to one operand (this is called the "monadic" form) or two operands (this is called the "dyadic" form). This is why `|` can mean "max" or "reverse" depending on the context.

This brings us back to the title question. Why learn K? Using it for interviews is probably not a very practical answer (although if you are interviewing at a company that allows for your choice of language it may be very tempting to use it. Writing a solution in K can be considerably shorter than writing the equivalent code in popular languages like Java/C#, C/C++ or Python). The main reason why I would recommend learning K is because it is a great language for manipulating data, and learning K will likely open your mind to a different way of thinking about programming, shifting from a loop-oriented approach to the higher level of abstraction provided by vector-oriented programming.

We could also compare K to other data-manipulation languages like SQL. The big difference in this case comes directly from the differences between the underlying models; a vector-oriented approach, where we can take advantage of ordering, and relational algebra, where the basic structure is a set, requiring ordering to be explicitly applied after the fact. For data manipulation, speed is key and if your data has some ordering to it (e.g., a time series), vector-oriented platforms and databases based on K such as 1010data, Shakti or KDB+ can provide a significant advantage over SQL.

You are probably familiar with the idea of stretching your comfort zone as a way to grow (whether it is physical exercise, learning to play a musical instrument or many other activities). If you are coming from a background in a different family of languages, learning K can often feel uncomfortable but that is because, as a different paradigm, it requires growing your "mental muscle", particularly at the beginning. When you feel this way, my best advice is to recognize the periods of growth for what they are and try to enjoy the journey instead. Once you become proficient in K you will probably get a certain degree of "addiction". When meeting former coworkers who used to program in K and are now at companies that use python or Go, more often than not I hear them say that they sometimes miss K and think how much easier and quicker it would be to solve certain problems extremely efficiently in K. The often "confess" to obtaining a free (demo) version of K to use in some of their personal projects.

Why study idioms?

"Each language, with its unique set of language constructs and library facilities, has a particular "best practice" method of use. These are the idioms that experienced users adopt, the modes of use that have become honed and preferred over time.

These idioms are important. They are what experienced programmers expect to read; they are familiar shapes that enable you to focus on the overall code design rather than get bogged down in macro-level code concerns. They usually formalize patterns that avoid common mistakes or bugs."

(Pete Goodliffe, Becoming a Better Programmer, O'Reilly Media, Inc., 2014)

Imagine that you are trying to memorize a series of letters. You may be able to remember a 9-letter arrangement like "bkxolsvaq" with a little bit of patience. What if instead you want to memorize the 9-letter arrangement "windshield"? Isn't that memorization much easier for our brain to handle? The reason is that our brain can use "chunks" for processing, and remembering syllables or words is not much different (in terms of effort) than remembering single letters. Just by having the recognizable pattern of a "word" we convert the problem of remembering 9 items into the much easier problem of remembering just 1 item.

Something similar happens in programming. You first learn the basic syntax and keywords, then, as you gain experience, your thinking shifts to larger patterns. A well-known example is the catalog of design patterns for object-oriented programming. Once you become familiar with a pattern you rarely need to think about the lower level details when you're using it. You will also be able to read and understand much faster other programs using these patterns. The equivalent learning process in APL languages has been the idiom lists, snippets of program for performing a task. By studying these idioms lists you learn how to use the language effectively, and programs written by other experienced programmers will become much easier to read, as you see the words and sentences instead of just seeing the letters.

This book will cover the K idioms list at http://web.archive.org/web/20010501133610/http://www.kx.com/technical/contribs/eugene/kidioms.html. This list of k idioms is derived from the FinnAPL "blue book" (http://www.finnapl.fi/texts/Idiot.htm) and was ported to K by Eugene McDonnell[1]. The classic APL Idiom List was written by Alan Perlis & Spencer Rubager and most likely the FinnAPL book has its roots in it.

The list in its original form was targeting people who were already familiar with K and would just search the list for an appropriate idiom, but in this book the goal is to cover them in a sequence oriented towards learning, grouped by techniques used and with an introductory text in each section describing the main principles. Note that I have chosen to preserve the original numbering, for ease of reference, even though it is not the sequence in which the idioms are presented. Another challenge that the original list presented for the learner is that there was little or no explanation for most of the idioms. While there is plenty of value in analyzing the idioms, it would be analogous to learning a language by analyzing the syntactical structure of sentences, good to practice every now and then and build up the skill, but tedious and not that useful for a whole book. Note also that I have chosen to skip idioms 1000 and above, because they primarily present solutions to complete problems, not really idioms.

Something to note about the style of these idioms is that they are written in a concise and generic form, e.g., most variables have single-letter names, and operations are condensed in one-liners more often than not. While this style is appropriate for an idioms list and for very short functions, in production code you often need to write functions that are much longer. If a high level of conciseness becomes too terse and interferes with the understanding of the code you are writing, by all means you should use longer variable names and write fewer operations per line.

Reference sections have been included at the end of this book. These sections contain very condensed information and it would be very hard for somebody new to the language to fully understand them on first (or second) read but you may want to initially skim over these sections before starting the idioms, to get an idea of what is available, and experiment with areas that look interesting, and later on refer back to them as needed, when it will hopefully be easier to absorb their meaning.

Getting K

Learning a language requires practicing it. Before even describing the syntax, it is important to indicate how to obtain it.

[1] https://kx.com/blog/q-idioms/

The canonical implementation for K is the one produced by its original author, Arthur Whitney. The version being actively developed (as of July 2018) is K7. K7 can be obtained from Shakti Software, their website is https://shakti.com. The previous commercial release is K4, which is integrated into Kx Systems KDB+/Q products (after starting Q, drop into “K mode” by typing a backslash at the console). The website for Kx systems is https://kx.com. An earlier version in active use is K3.33, used in the 1010data platform.

If you are looking at K for production code, and you aren't working for a company that already has a license, you will need to purchase one. The different versions represent different branches of development with different goals. You probably want to evaluate the different versions and see which one is better for your application.

For non-production use, you should be able to get demo versions of K7 (from Shakti), KDB+/Q/K4 (from Kx), or you may even be able to find a 32-bit version of K3.2 from some archives on the Internet. In addition to the commercial versions there are various re-implementations of K available. oK is a K interpreter written in JavaScript by John Earnest. This has the big advantage of running directly in a browser, without requiring an install, and it has some nice added functions for adding graphics. If you are looking for open-source, there is Kona (and, you may also find interesting the A+ language, the precursor to K, open sourced by Morgan Stanley). K has a well-earned reputation for being very fast, especially when compared to other analytical languages like Python or R, but bear in mind that the optimizations should be expected from the commercial versions, not the free re-implementations.

Running K

K is an interpreter, to run it you just run the corresponding executable and it should present a console where you can run commands directly. This console is sometimes called a REPL, because it is a program that reads, evaluates and then prints the result, in a loop.

If you make a mistake, the interpreter will print an error and, if possible, it will go into suspended mode (indicated by one or more `>` prompts) where you will be able to inspect values and run functions (e.g. to correct a problem). To exit one level of suspended mode you can type a `\` (backslash), to pop one level up the stack, back to the caller you can type a `'` (apostrophe) or, to continue execution you can type a `:` (colon)

At this point I'd suggest starting the interpreter and trying out some statements. Enter some numbers and mathematical operations like `+ - * %` (note that division is represented by `%` due to its similarity to ÷, and because `/` is used for “over” and to denote comments). The `\` command (when not in the suspend prompt) will display a quick help message about the available commands. Assigning values to variables is done with `:` (note that `=` is used for equality comparisons, you cannot use it for assignments). Note that variable names are case-sensitive, `Var` is different from `var` or `vAr`. You can add comments in the code by inserting a `/` (slash, preceded by a space to prevent ambiguity)

A typical “first test” for a computer language is writing a program that displays in the console the text “Hello world”. In the K REPL you can just enter “Hello world” between double quotes and it will print it out

Fundamental types

Before starting the idioms, we should describe the kinds of data that K can handle. The K type system can be broadly divided into atoms and vectors. The atomic types are:

1-Integers. These are 32-bit, signed integers. E.g. 1, 349, or -75389. In K3 There are 3 special visual representations: `0I`, `-0I` and `0N` that are convenient ways to express the largest and smallest integers, 2,147,483,647, -2,147,483,647, and -2,147,483,648 respectively. Something to note for arithmetic operations with integers is that integer overflow/underflow will occur when the result is larger than 32-bit, and this will impact the special representations as well, e.g. `0I+2` is `0I`

Integers are also used in K to represent false/true Boolean logic results as 0/1. Since these values are integers they will follow integer algebraic rules (e.g., in Boolean algebra True + True results in True but integer addition 1+1 results in 2)

2-Floating point. These are IEEE-754 floating point numbers. E.g. 8147483647.0, -12e9, or 3.1416e4. In K3 There are 3 special values, `0i`, `-0i`, and `0n` that correspond to positive and negative infinity and NaN respectively. These special values will follow the expected mathematical rules, e.g. `0i+7.` is `0i`

3-Characters. E.g., `"h"`, `"*"`, or `"7"`. in K3, Unicode is not supported for characters. In K3, characters can be converted from and to integers using _ci and _ic.

4-Symbols. These are blocks of text handled as a single unit. E.g., `` `foo ``, `` `bar ``, `` `"12345" ``, or `` `"my life in the woods" ``. They are preceded by a backtick at the beginning and, if they start by a number or contain spaces or characters that could be parsed as operators then they must also be enclosed in double quotes. A symbol can have zero characters, it is represented by just a backtick, and it is called the empty symbol. Symbol comparison is an atomic comparison and a more efficient operation than character vector comparison.

5-Dictionaries. A dictionary is a set of (key, value, attributes) triplets in K3 where the keys are symbols (with some restrictions, in K3 only symbols that wouldn't require double quotes are valid keys), value being any K type (atomic or vector), and the attributes being null or another dictionary.

6-Null. in K3 null is a different type from others

7-Functions (as data). K is a language with functional characteristics. Functions can be assigned to variables and be passed around as arguments, therefore functions also have a data type

Each atomic data type has a corresponding vector data type. There is an additional vector type, the mixed vector, that contains items of different atomic or vector types.

Character vectors are used very often for representing strings of text. For convenience, we can specify the vector inside a single set of enclosing double quotes. E. g.,

```
v:"Hello World"
```

the main difference between character vectors and symbols is that character vectors are intended for manipulating the data at a character level, while symbols are intended for manipulating as a unit. E.g.,

```
"a;b;c"=";"
0 1 0 1 0
```

```
`"a;b;c"=";"
0
`"a;b;c"=`"a;b;c"
1
"a;b;c"="a;b;c"
1 1 1 1 1
```

Mixed vectors are often used to represent a “vector of strings” (a mixed vector whose elements are character vectors), or matrices (a list of “rows”). Lists of matrices can be aggregated into a mixed vector to represent 3-dimensional tensors, and lists of tensors can be aggregated into a mixed vector to represent higher-dimensional tensors.

Mixed vectors can also contain elements of different types, and also vectors of different lengths (jagged arrays).

The terms “list” and “array” can be used interchangeably with “vector”. Often, “list” is used to differentiate mixed vectors from “true” vectors of uniform type, and “array” is used to refer to vectors, matrices and tensors as a group.

Idioms

Direct application of verbs

This introductory section explores the direct application of the built-in operators and functions, from details on simple usage to short combinations. It will be interesting observing the monadic and dyadic variations of verbs, how the interpretation of what they are doing can sometimes change when applied to atoms, vectors, lists, 0/1 values of vectors. It will also be interesting to see some simple combinations of operations.

575. Kronecker delta of x and y

```
  x:0 0 1 1
  y:0 1 0 1
  x=y
1 0 0 1
```

The Kronecker delta is a function of two states or sets of states that describes which items have changed from one state to the next (`1` indicating that the states are unchanged and `0` indicating that the states are different).

The `=` verb compares operands for equality, and when applied to two vectors it performs an element-wise comparison.

571. x but not y

```
  x:0 1 0 1
  y:0 0 1 1
  x>y
0 1 0 0
```

`>` performs a "greater than" operation. When applied element-wise to 0/1 vectors, it produces the same truth table as a Boolean "x but not y" (complement of implies) operation.

570. x implies y

```
  x:0 1 0 1
  y:0 0 1 1
  ~x>y
1 0 1 1
```

When applied to 0/1 vectors, the "not greater than" operation produces the same truth table as a Boolean logic "implies" operation. K does not provide a "less than or equal" verb but the same results can be obtained by applying a "greater than" and then negating the result.

Note how the `>` is applied between the two operands and then the result is negated with `~`. The negation `~` cannot be applied next to the `>`. This is a consequence of the monadic/dyadic overloading of operators. `x~>y` would be parsed much differently and take a completely different meaning, `>` would be interpreted as "grade down" and `~` as "match". We must ensure that two operands are provided when we want the dyadic interpretation of the verb to be used and only one operand when we want the monadic interpretation.

573. exclusive or

```
  x:0 0 1 1
  y:0 1 0 1
  ~x=y
0 1 1 0
```

Given 0/1 arguments, a negated equality comparison is equivalent to a Boolean exclusive-or operation.

41. indices of ones in Boolean vector x

```
  x:0 0 1 0 1 0 0 0 1 0
  &x
2 4 8
```

The result from `&` is a nonnegative integer vector containing indices of `x`, (see idiom 556), where each index `i` appears `x[i]` times.

The most common (but not only) usage is applying `&` to a 0/1 vector and in that case the indices of zeros will be present in the result zero times (i.e., not present) and the indices of oness will be present once, resulting in a "where" operation.

Another application for `&` is generating vectors of zeros of the specified size (or vectors of ones if we invert the result)

```
  &7
0 0 0 0 0 0 0
  ~&6
1 1 1 1 1 1
```

629. error to stop execution

```
&`
```

There are multiple ways to force an error when we want to intentionally stop the execution of a program (typically for debugging interactively). This idiom presents a short erroneous expression in K3 (because symbols are an illegal argument type for `&`)

516. multiply each column of x by y

```
  x:(1 2 3 4 5 6
>7 8 9 10 11 12)
  y:10 100
  x*y
(10 20 30 40 50 60
 700 800 900 1000 1100 1200)
  y*x
(10 20 30 40 50 60
700 800 900 1000 1100 1200)
```

The dyadic `*` verb multiplies two operands, and it does an element-wise multiplication when applied to two vectors.

When the first operand is a vector and the second is a matrix with outermost dimension of the same length as the vector, this will perform a multiplication of each row in the first operand by the corresponding scalar item in the second operand (and multiplication of vector by scalar will "broadcast" and multiply every item in the vector by the scalar), with a net result of multiplying each column of x by y. When reversing the operands, we are performing a scalar-by-vector operation that broadcasts to produce the same results as before.

Note that matrix multiplication is usually not commutative but element-wise multiplication is always commutative.

566. zero all items of Boolean x

```
  x:0 1 0 1 1 0 0 1 1 1 0
  0&x
0 0 0 0 0 0 0 0 0 0 0
```

The dyadic `&` operation returns the smaller ("min") of the two operands. We can zero all items of a 0/1 vector (or any integer vector with no negative numbers) by applying a zero-min to it and taking advantage of the operators being conformed (with the 0 being "broadcast" to the dimensions of the other operand). We can also use idiom 624.

624. zero numerical array

```
  x:2 3#99
  x
(99 99 99
 99 99 99)
  x*0
(0 0 0
 0 0 0)
```

We can zero the items of a numerical array by multiplying it by 0. This can be applied to vectors as well.

The data for this idiom is produced by the "reshape" operation, dyadic `#` that produces a result of the desired shape based on replication of an atom (or replication of the elements in a vector)

622. retain value of items marked by y, zero others

```
  x:3 7 15 1 292
  y:1 0 1 1 0
  x*y
3 0 15 1 0
```

This idiom is showing how we can use multiplication by a 0/1 vector to preserve the items marked by ones and zero out those marked by zeros.

331. identity for floating point and integer maximum, negative infinity -0i

```
/ identity for floating point and integer maximum, negative infinity
  -1e100|-0i
-1e+100
  -123456789|-0I
-123456789
```

The dyadic `|` operation returns the largest ("max") of its two operands. When negative infinity is an operand to `|` the operation will return the other operand in most cases.

Note that when the other operand is an integer “NaN”, this identity will not hold.

```
  -0i|0n
-0n
  -0I|0N
-0I
```

337. identity for floating point and integer minimum, positive infinity

```
/ identity for floating point minimum, positive infinity
  1e100&0i
1e+100
/ identity for integer minimum, positive infinity 0I
  123456789&0I
123456789
```

When positive infinity is an operand to the `&` (“max”) operation the result will be the other operand in most cases.

Note that this identity holds for floating-point, but it breaks down when the other operand is an integer “NaN”.

```
  0I&0N
0N
  0i&0n
0i
```

357, 544. does x match y

```
  x:("abc";`sy;1 3 -7)
  y:("abc";`sy;1 3 -7)
  x~y
1
  x:1 2 3
  y:1 4 3
  x~y
0
```

Dyadic `~` performs a match operation.

It is interesting to consider how is match different from equality comparison `=`. The key fundamental differences are that match will process each operand as a whole, and that it tolerates operands of potentially different types, returning 0 if they are different. Some important consequences of these differences are:

Match can be used for comparing to null. `_n` or variables with a null value are invalid arguments to `=`, however `x~_n` returns `1 or `0` depending on whether x is null or not.

Match will not conform (broadcast) atoms and vectors, `1~1 2 3` will return 0 because the types are different, while `1=1 2 3` will return `1 0 0`` because the operands will be conformed.

Match will not extend comparisons to be performed element-wise, `1 2 3~1 2 3` will return 1 while `1 2 3=1 2 3` will return `1 1 1`

Match will not perform automatic numeric conversions between integer and float, `1~1.0` returns 0 because the types are different, `1=1.0` returns 1 because the values are the same amount.

328. number of items

```
  #"abcd"
4
  #(1;2 3;4 5 6)
3
  #2 3 4#!24
2
```

Monadic `#` will count the number of items in a list. Note how when counting nested (or multi-dimensional) items it is only the outermost level that is counted.

411. number of rows in matrix x

```
  x:2 7#" "
  ^x
2 7
  #x
2
```

A count operation when applied to a matrix it will provide the count of rows (because it is the “outermost” dimension, as shown in the output of the shape operation)

445. number of columns in matrix x

```
  x:4 3#!12
  x
(0 1 2
 3 4 5
 6 7 8
 9 10 11)
  *|^x
3
```

For counting columns in K3 we can get the last item (first item of the reverse) of the shape of the matrix. Alternatively, we can index into the shape, or we can count the number of items in the first row `#*x`

388. drop y rows from top of matrix x

```
  x:6 3#!18
  x
(0 1 2
 3 4 5
 6 7 8
 9 10 11
 12 13 14
 15 16 17)
  y:2
  y _ x
```

```
(6 7 8
 9 10 11
 12 13 14
 15 16 17)
```

A cut operation, dyadic `_`, when applied to a matrix it will cut at the outermost dimension, which means it will cut rows. If we want to apply it at the next dimension we just need an "each" adverb, `y _' x` will drop the y leftmost columns instead. If we want to drop from the bottom (or from the right in the case of columns) we just need to negate `y`, `(-y)_ x` or `(-y)_' x`

154. range (nub; remove duplicate items)

```
  x:"wirlsisl"
  ?x
  "wirls"
  ?(1 2 3;4 5;1 2 3;4 5;1 2 3)
(1 2 3
 4 5)
```

The range (find uniques) operator is monadic `?`. It returns the unique elements in the input (a copy of the input where any subsequent duplicates of an item are removed). It is important to note that the order in the input is preserved; if the first occurrence of item `a` occurs before the first occurrence of item `b` in the input, then `a` will occur before `b` in the output.

70. remove duplicate rows

```
  x:("to"
  "be"
  "or"
  "not"
  "to"
  "be")
  ?x
("to"
 "be"
 "or"
 "not")
```

The find uniques operation `?` when applied to vectors it is applied to the entirety of the vector (not element-wise) therefore, it can be used for finding unique character vectors.

143. indices of distinct items

```
  x:"ajhajhja"
  =x
(0 3 7
1 4 6
2 5)
```

This idiom presents the "group" operator, monadic `=`. Given a list of items, for each unique item it returns the indices of all items matching the value. Group and range (see idiom 154) will calculate uniques in the same way, if you need to associate the groups with their values, use a range (find uniques) operation

228. is y a row of x

```
  x:("xxx";"yyy";"zzz";"yyy")
  x?"yyy"
1
```

We can use find, dyadic `?` to identify if a vector is present in a list of vectors. Note that find will give us the index and we need to compare to the count of rows if we want a 0/1 result

```
  x:("xxx";"yyy";"zzz";"yyy")
  x?"zzz"
2
  (#x)>x?"zzz"
1
```

An alternate way to determine if `y` is a row of `x` is provided in idiom 232

232. is y a row of x

```
  x:("aaa"
>"bbb"
>"ooo"
>"ppp"
>"kkk")
  y:"ooo"
  y _in x
1
```

We can use the built-in `_in` operator to check for the presence of a (character) vector in a list of (character) vectors

559. index of first marker in Boolean x

```
  x:0 0 1 0 1 0 0 1 1 0
  x?1
2
```

When `x` is a 0/1 vector we can get the index of the first marker (the first 1) with a direct application of find, dyadic `?`.

78. number from alphanumeric

```
  x:"1998 51"
  . x
1998 51
  3 + . x
2001 54
```

This is a very useful general technique, using eval `.` to let the interpreter evaluate a character vector as if it had been typed in. This dynamic evaluation however is one of the aspects of K (and APL languages in general) that make it harder to write a compiler for K instead of an interpreter.

In practice, we should strongly prefer the “cast” family of operations, e.g., `0$"42",` for converting strings to other values. Eval should never be used on user modifiable data without sandboxing it first (note: only a few versions of K support sandboxing). Taking user input and feeding it to `.:` opens the code to injection attacks

```
  42 + . "`0:,\"you're going to have a bad time.\";10"
you're going to have a bad time.
52
```

For trusted data (not directly modifiable by end users), eval can be very useful in scenarios that require deserializing more complex structures (dictionaries, list of lists, etc.) from locations that don't support saving/loading .l files directly (e.g., from tags in an object store or a structured document).

88. name variable according to x

```
  x:"test"
  y:2 3#!6
  . "var",($x),":y"
  vartest
(0 1 2
 3 4 5)
  x:123
  . "var",($x),":y"
  var123
(0 1 2
 3 4 5)
```

This is another application of `.` for invoking the interpreter from within a script and evaluating a character vector as if it had been typed in. This verb presents risks of injection attacks. See idiom 78 for warnings and best practices.

96. conditional execution

```
  @[+/;!6;:]
0 15
  @[+/;"abc";:]
(1;"type")```
```

This is the "error trapping" in K. Using a `:` as the third argument of a triadic `@` or `.` will return a tuple that, if the operation fails it will contain a 1 and a character vector with an error message, and if the operation succeeds it will contain a zero and the result of the operation. The difference between conditional execution using `.` and `@` is the interpretation of the 2nd argument as an atom or a vector, `.[f;,a;:]` is equivalent to `@[f;a;:]`. Also, if the function takes 2 or more arguments you need to use `.[f;(a;b);:]`, otherwise the vector of arguments will be interpreted as applying a monadic function to each of the arguments in the list.

115, 116, 117. case structure

```
  :[c0;t0;f]
  :[c0;t0;c1;t1;f]
  :[c0;t0;c1;t1;c2;t2;f]
  :[c0;t0;c1;t1;c2;t2;c3;t3;f]```
```

et cetera

The conditional operation `:[]` will evaluate conditions and return the argument following the condition if true. The pairs of condition/result will be sequentially evaluated until a condition is true and if all the conditions are false it will return the last argument if the count of arguments is odd, or null if it is even. It vaguely resembles the "case" construct of languages like C, although it uses full conditions (instead of requiring a "switch") and there is no "break"; "falling through to the next case" is not an option.

493. choose x or y depending on Boolean g

```
  x:"abcdef"
  y:"xyz"
  g:0
  :[g;x;y]
"xyz"
  g:1
  :[g;x;y]
"abcdef"
```

This is an application of the triadic conditional operator (with 3 arguments). When used this way it resembles the ternary operator `?:` in languages like C.

434. replace first item of x with y

```
  x:"abbccdefcdab"
  y:"t"
  @[x;0;:;y]
"tbbccdefcdab"
```

The amend operator, `@` with 4 arguments, will return the result of modifying the entry provided in the first argument, at the indices indicated in the second argument, applying the operation in the third argument to each item being modified, with additional arguments for the specified function being passed in as the fourth argument of amend (either one argument for each item to be modified, or an atom, which will be used for all the modified items).

This idiom shows how to use amend with a "return" operation as its 3rd argument to modify the item with index 0 (return the items in `x`, but if the index is 0 return `y`). Generally, the original variable will not be modified, if that is what we need we should assign the results of the amend back to the variable. In K3, if the first argument is a symbol representing the location of a global variable in the K tree (akin to a "handle" or "reference" in other languages) then the variable itself will be modified.

Specifically for replacing the first item, we could also use cut/append operations for a shorter solution `y,1_ x`.

433. replace last item of x with y

```
  x:"abbccdefcdab"
  y:"t"
  @[x;-1+#x;:;y]
"abbccdefcdat"
```

We can use an amend operation to modify the item corresponding to length-1, or we can use cut/append operations for a shorter solution `(-1_ x),y`.

406. add y to last item of x

```
  x:1 2 3 4 5
  y:100
  @[x;-1+#x;+;y]
1 2 3 4 105
```

This idiom is a variation of the previous one, using an addition operation instead of assignment. The alternative solution using takes and cuts would be `(-1_ x),y+-1#x`, we can observe how for less trivial cases amend becomes more expressive (and is not longer anymore).

449. limiting x between l and h, inclusive

```
  x: 5 6 _draw 100
  x
(58 9 37 84 39 99
 60 30 45 97 77 35
 49 87 82 79 8 30
 46 61 20 51 12 34
 31 51 29 35 17 89)
  l:30
  h:70
  l|h&x
(58 30 37 70 39 70
 60 30 45 70 70 35
 49 70 70 70 30 30
 46 61 30 51 30 34
 31 51 30 35 30 70)
```

The max and min operations when applied to a scalar and a matrix will make its arguments conform ("broadcast"), giving us a very concise and expressive way to apply both a max limit and a min limit.

495. indices of all occurrences of y in x

```
  x:"abcdefgab"
  y:"afc*"
  x _lin y
1 0 1 0 0 1 0 1 0
  &x _lin y
0 2 5 7
```

When y is an atom we would normally use an equality comparison to obtain a 0/1 vector of matches that can be converted to indices with `&` (see idiom 503). When `y` is a vector, we can use `=/:` instead of `=` and then or-over the results (`&|/x=/:y`), or we can just use `_lin` to obtain the 0/1 indicators of whether an item is in the intersection or not.

504. replace items of y satisfying x with g

```
  x:1 0 0 0 1 0 1 1 0 1
  y:"abcdefghij"
  g:" "
  @[y;&x;:;g]
" bcd f i "
```

This is an application of amend where the indices are calculated by applying `&` to convert a 0/1 vector to a vector of indices (see idiom 41).

569. change y to one if x

```
  y:10 5 7 12 20.0
  x:0 1 0 1 1
  y^~x
10 1 7 1 1.0
/ alternatively
  @[y;&x;:;1]
10 1 7 1 1
```

This idiom is presenting two ways to conditionally change vector items to 1. The first technique uses element-wise exponentiation, because `x^0` returns 1 and `x^1` returns `x`. This is a very concise expression, but it requires applying a transcendental function repeatedly, which may not be desirable. The second technique uses amend (a direct application of idiom 504).

556. all indices of vector x

```
  x:2 2 2 2
  !#x
0 1 2 3
```

We can obtain all the indices of a vector by enumerating the count of items in the vector.

535. avoiding parentheses using reverse

```
  x:1 2 3 4 5
  (#x),1
5 1
  |1,#x
5 1
```

This is an example of avoiding parentheses by using a reverse operation. This can be applied to concatenations of 2 atoms where the first operand is calculated using an expression that requires parentheses, but the second operand is a simple atom or does not require parentheses. The resulting expression is slightly shorter.

591. reshape vector x into 2-column matrix

```
  x:"abcdefgh"
  ((_ 0.5*#x),2)#x
("ab"
 "cd"
 "ef"
 "gh")
```

This is a simple application of the reshape operator, dyadic `#` with a vector as its first argument. Note that the result of multiplying by 0.5 is a floating-point number and needs to be truncated to make the first argument an integer vector.

595. one-row matrix from vector

```
  x:2 3 5 7 11
  x
2 3 5 7 11
  ^x
  ,5
  ,x
  ,2 3 5 7 11
  ^,x
1 5
```

This idiom shows that the shape of an enlisted vector is actually a one-row matrix.

616. scalar from one-item vector

```
  x:,8
  x
,8
  ""#x
8
  x[0]
8
  *x
8
```

This idiom presents different ways to convert a one-item vector into a scalar. The first is using a reshape operation with an empty vector (in general a character vector cannot be used as the first argument for dyadic `#`, however if it is empty it is ok in K3). The second is using indexing, and the third is using "first" (monadic `*`).

509. remove y from x

```
  x:"abcdeabc"
  y:"a"
  x _dv y
"bcdebc"
```

`_dv` removes matching values from a vector. Note that if `y` had been a list of characters instead of an atomic character we would have needed `_dvl` instead (see idiom 496).

510. remove blanks

```
  x:" bcde bc"
  x _dv " "
"bcdebc"
```

This is a specific application of the previous idiom.

496. remove punctuation characters

```
  x:"oh! no, stop it. you will?"
  y:",;:.!?"
  x _dvl y
"oh no stop it you will"
```

`_dvl` deletes items by value, similar to `_dv`, but the items to delete are a list. This idiom shows an application where the values to delete are a list of all punctuation characters to be removed.

177. indices of start of string x in string y

```
  x:"st"
  y:"indices of start of string x in string y"
  y _ss x
11 20 32
```

The `_ss` operation performs a “string search” and finds the indices where the first operand can be found in the second operand.

45. binary representation of positive integer

```
  x:16
  2 _vs x
  1 0 0 0 0
  x:20
  2 _vs x
1 0 1 0 0
```

The `_vs` operation performs the equivalent of a base conversion on a number (when the base is between 2 and 10). In this application, base 2 produces the binary representation.

84. scalar from Boolean vector

```
  x:1 0 0 1 1 1 0 1
  2 _sv x
157
```

The `_sv` operation performs a reverse base conversion. This idiom presents the reverse of idiom 45.

129. arctangent y%x

```
  x:_sqrt[3]
  y:1
  _atan[y%x]
0.5235988
```

K has multiple trigonometric functions. In K3 the `_atan` function calculates the arctangent.

561. numeric code from character

```
  x:" aA0"
  _ic[x]
32 97 65 48
```

In K3, `_ic` returns the value of a character within the internal character list (system dependent, generally ASCII or the OS-provided extended ASCII).

241. sum over subsets of x specified by y

```
  x:1+3 4#!12
  x
(1 2 3 4
 5 6 7 8
 9 10 11 12)
```

```
  y:4 3#1 0
  y
(1 0 1
 0 1 0
 1 0 1
 0 1 0)
  x _mul y
(4 6 4
 12 14 12
 20 22 20)
```

If we specify subsets of the rows in `x` as 0/1 columns of `y` we can just use matrix multiplication to obtain the subset sum. If `y` is a "mask" of the same shape as `x` we can add a transposition `x _mul +y`

245. randomize the random seed

```
  _t
-1154371779
  \r -1154371779
  \r
-1154371779
```

We can use the current time to initialize K3's pseudorandom number generator. Otherwise pseudorandom operations will produce identical results every time that a program is run.

61. cyclic counter, repeating 1 through n

```
  x:!10
  y:8
  1+x!y
1 2 3 4 5 6 7 8 1 2
```

Using the modulo operation we can set an upper limit to an enumeration. The result of applying the modulo will result in a 0 to `y-1` range, if we want 1 to `y` we need to add 1 to the result.

384. drop 1st, postpend 0

```
  x:3 4 5 6
  1 _ x,0
4 5 6 0
```

This is a composition of the operations in the title, shift a vector left inserting 0 on the right.

385. drop last, prepend 0

```
  x:3 4 5 6
  -1 _ 0,x
0 3 4 5
```

This is a composition of the operations in the title, shift a vector right inserting 0 on the left.

178. index of first occurrence of string x in string y

```
 x:"st"
  y:"index of first occurrence of string x in string y"
  *y _ss x
12
```

This is a variation of idiom 177. If we just need the first occurrence, after applying `_ss` we can just take the first result. This technique is very simple but, if y is long, we may want to use instead a search that ends processing on first match.

447. conditional drop of y items from array x

```
  x:4 3#!12
  x
(0 1 2
 3 4 5
 6 7 8
 9 10 11)
  y:2
  g:0
  (y*g) _ x
(0 1 2
 3 4 5
 6 7 8
 9 10 11)
  g:1
  (y*g) _ x
(6 7 8
 9 10 11)
```

If we want to remove the first `y` rows depending on a 0/1 condition `g`, instead of using an `if,` we can multiply the number of rows to drop by the condition. Note that this requires the condition to be 0/1, unlike `if`, that takes a zero/nonzero (this can be important if you're refactoring existing code and changing `if` to this idiom).

For a more general conditional execution of a verb it is possible to use the DO variant of `/`. For example, conditionally enlisting:

```
  (x~5),:/"foo"
,"foo"
  (x~3),:/"foo"
"foo"
```

448. conditional drop of last item of array x

```
  x:4 3#!12
  x
(0 1 2
 3 4 5
 6 7 8
 9 10 11)
  y:0
  (-y) _ x
(0 1 2
 3 4 5
 6 7 8
 9 10 11)
```

```
  y:1
 (-y) _ x
(0 1 2
 3 4 5
 6 7 8)
```

This is a variation on the previous idiom, using a negative number for dropping from the end (instead of the beginning) and eliminating the `g` if we just need to drop one item (there is no need to multiply by one)

549. alphabetic comparison (depends on storage values)

```
  "a"<"b"
1
  "a">"b"
0
  _ic";" / K3
59
  _ic"/" / K3
47
  ";"<"/"
0
```

The `<` and `>` operators will work on characters and perform an alphabetic comparison. If we need details like how punctuation characters will be sorted, this comparison is based on the same values that are returned by `_ic` (in K3 only). Since character vectors are processed like any other vectors, `<` and `>` are applied element-wise and generally not very useful if we want to compare text items. If we want to lexicographically compare words we can use symbol comparison, e.g.,

```
  `hello<`world
1
  `foo<`bar
0
```

Extending verbs with adverbs

Much of the power of K (and other languages directly related to APL) comes from the idea of extending verbs by using adverbs. Adverbs enable verbs (operators and functions) to be extended to collections of items. There are many languages that provide "for"\", "do" and "while" loop statements but in K, you should be able to express the same concepts using adverbs instead. Adverbs provide a higher level of abstraction that makes intentions clearer and prevents many of the problems inherent in loops. With adverbs there are no out-of-bounds problems, in the same manner as languages that have a "foreach" loop (note that "each" is only a subset of the available adverbs, compared to "foreach", adverbs provide a richer set of options, like "over" and "scan"). In deference to programmers used to other languages, K provides a "do" and a "while" keyword, but they are unnecessary, and their use is often frowned upon. The `/` adverb has "converge", "do" and "while" forms that can handle all the scenarios not already covered by other adverbs (and `\` also provides these forms, similar to `/` but producing all the intermediate results from the iteration, as a vector). One of the top K websites, nsl.com, takes its name from "no stinkin' loops". This chapter explores some of the power of adverbs.

335. maximum

```
  x:5 3 7 2
  |/x
7
```

The “max” operation, when modified with an “over” adverb will apply a max operation over all the elements of a sequence, comparing each element to the previously found maximum value. The “over” functionality is sometimes called “reduce” in other languages.

339. minimum

```
  x:5 3 7 2
  &/x
2
```

This is the “min over” operation, analogous to “max over” in the previous idiom but, calculating the minimum instead.

356. any

```
  x:&7
  |/x
0
  x:7#0 1
  |/x
1
```

A max operation over a 0/1 vector will yield a one if any of the items is a one. In the context of 0/1 vectors, `|` operates as a "Boolean Or".

360. all

```
  x:1 1 0 1
  &/x
0
  x:1 1 1 1
  &/x
1
  x:0 0 0 0
  &/x
0
```

A min over a 0/1 vector will produce a 1 only if all the items are 1. In the context of 0/1 vectors, `&` operates as a "Boolean And".

355. none

```
  x:&7
  x
0 0 0 0 0 0 0
  ~|/x
1
  x:7#0 1
  x
```

```
0 1 0 1 0 1 0
  ~|/x
0
```

A max operation over a 0/1 vector will yield a zero only if all the items are zero.

334. nonnegative maximum

```
  x:1 2 3 4 5
  |/x,0
5
  x:-1 -2 -3 -4 -5
  |/x,0
0
  x:!0
  x
!0
  |/x,0
0
```

By appending a zero before applying `|/` we can ensure that the result is numeric and nonnegative (even for empty vectors).

222. maximum of x with weights y

```
  x:1 2 3 4 5
  y:5 4 3 2 1
  |/x*y
9
```

This idiom is a straightforward translation of the requirements, showing the vector nature of K. It first applies the weights with `*` and then a max over `|/`. Compare it to the equivalent visibly longer code in a non-vector language like C.

```
  int max=INT_MIN;
  int candidate;
  for(int i=0;i<5;i++)
  {
    candidate=x[i]*y[i];
    if(candidate>max)
    {
      max=candidate;
    }
  }
```

223. minimum of x with weights y

```
  x:1 2 3 4 5
  y:5 4 3 2 1
  &/x*y
5
```

This idiom first applies the weights with `*` and then applies a min over `&/`. See comments in the previous idiom.

368. product

```
  x:1 2 3 4 5
  */x
120
```

Multiply over a simple vector gives us the product of all the elements in the vector.

374. sum

```
  x:1 2 3 4 5
  +/x
15
```

This is an application of sum over, `+/`, when the argument is a simple vector. Compare to Idiom 372 when the argument is a matrix instead.

370. count of 1s in Boolean list

```
  x:1 0 0 1 0 1 1
  +/x
4
```

The count of ones in a 0/1 vector is the same as the sum of the elements in the vector. We can also use `#&x`.

362. count of occurrences of x in y

```
  x:"q"
  y:"quaquaqua"
  +/x=y
3
```

The equality comparison in the case of an atom and a simple vector produces a 0/1 vector, and by summing it we can count the number of occurrences. We can also use `#&x=y`.

239. sum reciprocal series

```
  x:10 9 10 7 8
  y:80 63 70 63 64
  +/y%x
39.0
```

This is applying a sum operation over an element-wise division of vectors.

242. sum squares of x

```
  x:1 2 3 4 5
  +/x*x
55
```

This is a sum over the element-wise multiplication of the input by itself.

243. dot product of vectors

```
  x:1 2 3 4 5
  y:10 20 30 40 50
  +/x*y
```

```
550
```

This is a sum over the element-wise multiplication of the inputs.

372. sum columns of matrix

```
  x:1+3 4#!12
  x
(1 2 3 4
 5 6 7 8
 9 10 11 12)
  +/x
15 18 21 24
```

The `/` adverb applies an operation to the outermost level (most superficial elements) of its argument. In the case of a matrix, this outermost level is composed of the individual rows. The `+` operation, when applied to 2 rows it will perform the element-wise addition, which results in another vector that contains the sum of the items in the same column for both rows. `+/` applied to the entire matrix will result in a vector containing the column-wise sum.

361. parity

```
  x:2 _vs !8
  x
(0 0 0 0 1 1 1 1
 0 0 1 1 0 0 1 1
 0 1 0 1 0 1 0 1)
  (+/x)!2
0 1 1 0 1 0 0 1
```

The parity of a 0/1 vector is the modulo 2 of the sum of its items. If we apply this operation to a 0/1 matrix it will produce the parity of the columns (because the sum operation is applied element-wise to the rows)

310. running sum

```
  x:1 20 300 4000
  +\x
1 21 321 4321
```

The scan `\` adverb is similar to over, but it produces all the intermediate results instead of reducing to just the last result. Calculating the running sum can be done with a simple application of sum scan `+\`.

285. moving sum

```
  y:3
  x:1 2 3 4 5
  +\x
1 3 6 10 15
  (-y)
-3
  (-y)_ a:+\x
1 3
  0,(-y)_ a
```

```
0 1 3
  (y-1)_ a
6 10 15
  ((y-1)_ a)-0,(-y)_ a
6 9 12
  ms:{((y-1)_ a)-0,(-y)_ a:+\x}
  ms[x;y]
6 9 12
```

The moving sum is the sum of the prior `y` elements, where the set of elements being added is called the "window" for the moving sum (which means `y` is the "window width" or the "window length"). We can use the running sum `+\` as the starting point for a vector calculation of the moving sum of a given series `x` with width `y`. We want to subtract from the running sum a copy of the running sum itself displaced by the width of the window. Some minor adjustments are required before the subtraction. We need to prepend a zero to the displaced sum because the sum of the first `y` items doesn't need to have anything subtracted from it, and we also need to remove the initial `y-1` items from the running sum because they represent the part where we don't yet have enough information for a moving sum (e.g., a moving sum of 3 items does not have a meaningful value until we actually have 3 items). After these adjustments the length of the two vectors is the same and a vector subtraction will produce the desired result.

The moving sum can be interesting for scenarios that look for conditions relative to a moving average (e.g., finding the points where a series crosses its 50-item moving average). The shape of the moving sum and moving average are identical (one is just a scaled version of the other), but the moving sum is computationally simpler (it avoids a division operation over the entire vector).

309. running parity

```
  x:0 1 1 1 1 0 1 0 0
  (+\x)!2
0 1 0 1 0 0 1 1 1
```

The running parity can be calculated as the parity of the running sum.

306. invert all 1s after 1st 0

```
  x:1 1 0 1 0 1 0
  &\x
1 1 0 0 0 0 0
```

If we apply a running min to a 0/1 vector, all the values after the first zero will be set to 0

189. add x to each row of y

```
  x:1+!4
  y:3 4#2+!12
  y
(2 3 4 5
 6 7 8 9
 10 11 12 13)
  x+/:y
(3 5 7 9
 7 9 11 13
 11 13 15 17)
```

By applying the each-right adverb `/:` to the sum, we keep a fixed first argument and sum it to each entry in the second argument. Note that adding the first argument to each row of the second argument must be a valid operation, therefore the length of the first argument must be the same as (or must be conformable to) the number of columns in the second argument.

192. add x to each column of y

```
  x:1+!2
  x
1 2
  y:2 5#3+!10
  y
(3 4 5 6 7
 8 9 10 11 12)
  x+'y
(4 5 6 7 8
 10 11 12 13 14)
```

When applying `'` to the sum we are adding x to each column of y. Compare to idiom 189.

273. join scalar to each list item

```
  x:"a"
  y:"01234"
  x,/:y
("a0"
 "a1"
 "a2"
 "a3"
 "a4")
  x:("01234";"56789")
  y:("abcde";"fghij")
  x,y
("01234"
 "56789"
 "abcde"
 "fghij")
  x,'y
("01234abcde"
 "56789fghij")
```

This is an exploration of different ways to concatenate character vectors. `,/:` prepends a character (or it could be a character vector) to each character in a vector. Note that we could reverse it to `,\:` and reverse the arguments to append instead of prepend. If instead we use a simple join, `,` we get a longer list of unmodified character vectors (a "vertical" append). If we use `,'` we will join each pair of character vectors (a "horizontal" append, where all inputs contain the same number of character vectors, and the result will also have the same length).

282. index of first blank

```
  x:"ab c d"
  x?" "
2
  x:("ab c d";" a bc";"abcd ")
  x
("ab c d"
 " a bc"
 "abcd ")
  x?\:" "
2 0 4
```

In its simplest form (applied to a character vector) this is a direct application of `?` but, to extend the operation to a list of character vectors we need to add an each left `\:`.

344. pairwise match

```
  x:("123";"123";"45";"45")
  x
("123"
 "123"
 "45"
 "45")
  ~':x
1 0 1
  (~':x),0
1 0 1 0
  pm:{(~':x),0}
  pm 1 1 1 1 2 2 3 4 4 4
1 1 1 0 1 0 0 1 1 0
```

This is a simple application of `~` with the `':` adverb. If we want the result to be the same length as the input, we need to append a zero.

371. scalar from 1-item list

```
  x:,5
  x
,5
  +/x
5
  a:,"a"
  a
  ,"a"
  +/a
type error
+/a
^
> \
  a[0]
"a"
```

For the more general case, we can just index into the first element (or use a monadic `*`). For numbers we can also apply an operation (`+`, `-`, `*` or `%`) over the vector, which will be idempotent for the 1-element case.

373. sum rows of matrix

```
  x:1+3 4#!12
  x
(1 2 3 4
 5 6 7 8
 9 10 11 12)
  +/'x
10 26 42
```

`+/'` will apply `+/` to each individual row in the matrix, producing a vector with the row-wise sums. A way to conceptualize this result is that adding a `'` enables us to apply the operation one level deeper in a matrix (similarly, adding `''` would apply the operation two levels deeper in a rank 3 tensor, and so forth).

383. pairwise difference

```
  x:9 3 5 2 0
  --':x
6 -2 3 2
```

We can convert a numeric series into its corresponding reverse deltas (series of decreases) by applying an each-prior adverb to a subtract operation `-':` and then negating the result (because each-prior will calculate the forward delta).

398. diagonals from columns

```
  x:(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)
  (-!5)!'x
(1 2 3 4 5
 10 6 7 8 9
 14 15 11 12 13
 18 19 20 16 17
 22 23 24 25 21)
```

We can convert columns to diagonals by rotating right each row according to its index. For the more generic case, use `(-#x)!'x`.

399. columns from diagonals

```
  x
(1 2 3 4 5
 10 6 7 8 9
 14 15 11 12 13
 18 19 20 16 17
 22 23 24 25 21)
  (!5)!'x
(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)
```

This is the complementary operation to idiom 398 (diagonals from columns), rotating left each column (instead of right). For a generic version, use `(!#x)!'x`.

419. pairwise ratios

```
  x:2 10 50 100
  %':x
5 5 2.0
```

This is a straightforward application of a division operator modified by an each prior adverb.

431. does item differ from next one

```
  x:"ceefffmeksc"
  (~=':x),1
1 0 1 0 0 1 1 1 1 1 1
```

This idiom is applying a not match operation, modified with an each prior adverb, and appending a one so that the results have the same length (as if the last item is different from the non-existing next item).

432. does item differ from previous one

```
  x:"ceefffmeksc"
  1,~=':x
1 1 0 1 0 0 1 1 1 1 1
```

This idiom is applying a not match operation, modified with an each prior adverb, and prepending a one so that the results have the same length (as if the first item is different from the non-existing previous item).

442. 1st difference

```
  x:+\1 2 3 4 5
  x
1 3 6 10 15
  -':x
2 3 4 5
  (*x),(-':x)
1 2 3 4 5
```

When we apply an each prior adverb, e.g., for obtaining the successive differences, the resulting vector is 1 element shorter than the original. If it is desirable having the same length, we can prepend the first element unmodified. Actually, the comma and second set of parentheses are unnecessary, because `f':` accepts a list initializer as its first operand `(*x)-':x`.

484. right to left scan

```
  x:1 2 3 4 5
  |+\|x
15 14 12 9 5
```

If we need to apply a running operation right-to-left, and the operation is commutative, we can reverse the target vector, apply the running operation and then reverse the result. This technique can be useful for right-to-left running max and running min as well.

511. apply f over all of x

```
  x:2 3 4#1+!24
  x
((1 2 3 4
  5 6 7 8
  9 10 11 12)
 (13 14 15 16
  17 18 19 20
  21 22 23 24))
  +//x
300
  ao:{[f;x]f//x}
  ao[+;x]
300
  ao[*;1.0*x]
6.204484e+023
  ao[+;-x]
-300
```

Applying a dyadic function over a list is done by modifying the function with the "over" `/` adverb. When applied to an array or tensor it will operate on the outermost dimension only. The result of composing a dyadic function with `/` is a monadic operation itself, which we can further modify with an additional `/`. `m/` is a "monadic converge" operation, that applies the operation until the result doesn't change. We can use this convergence for applying the operation successively until the operation has been applied over all dimensions of the matrix (or tensor). Observe also how K has functional characteristics and it is possible to define a function that takes another operator or function as one of its arguments. Typical applications of this technique include mathematical operations like adding or multiplying all elements of a matrix (or tensor), or "flattening" a matrix (or tensor) into a vector (e.g., as a step prior to doing a reshape, or as the final step of text manipulation to make the result a character vector).

98. execute rows of character matrix

```
  x1:4
  x2:9
  x:2 5#"y1:x1y2:x2"
  x
("y1:x1"
 "y2:x2")
  .:'x
(;)
  y1
4
  y2
9
```

We can store portions of a K program in a vector of character vectors and use eval `.` and each `'` to invoke the interpreter on them.

518. transpose matrix x on condition y

```
  x:2 3#!6
  x
(0 1 2
 3 4 5)
  y:1
  y +:/x
(0 3
 1 4
 2 5)
  y:0
  y +:/x
(0 1 2
 3 4 5)
```

This is a straightforward application of the “DO” form of adverb `/` that takes as arguments a count of times to apply `y`, a monadic function `+:` and its argument `x`. The resulting expression is even shorter than using a conditional, `:[y;+x;x]`.

533. reverse x on condition y

```
  x:1 2 3 4 5
  y:0
  y |:/x
1 2 3 4 5
  y:1
  y |:/x
5 4 3 2 1
```

This is another direct application of the “DO” form of `/`, applied `y` times to monadic reverse `|:`.

562. index of y in x

```
  x:" abcdefgh"
  y:"faded head"
  x?/:y
6 1 4 5 4 0 8 5 1 4
  y:"deaf adder"
  x?/:y
4 5 1 6 0 1 4 4 5 9
```

We can find the index in `x` of the first instance of each item in `y` by modifying `?` with an each-right `/:` adverb.

589. 2-column matrix from two vectors

```
  x:"abcd"
  y:"efgh"
  x,'y
("ae"
 "bf"
 "cg"
 "dh")
```

This is a very similar expression to idiom 583, with two different vectors as arguments.

592. vector from array

~~~~
```   x:2 1 2 1 2 1#!8
  ^x
2 1 2 1 2 1
  ,//x
0 1 2 3 4 5 6 7
  ^,//x
,8```
~~~~

This is the "flatten" operation applied to a tensor, concatenate until we reach convergence.

611. multiply each row of x by vector y

```
  x:3 4#1+!12
  x
(1 2 3 4
 5 6 7 8
 9 10 11 12)
  y:1 10 100 10000
  x*\:y
(1 20 300 40000
 5 60 700 80000
 9 100 1100 120000)
```

If we try to multiply a vector by a matrix it will multiply each row in `x` by the corresponding item in `y`. If instead we want the element-wise multiplication of each row in `x` by `y` we need to apply an each-left adverb (or an each-right and reverse the operands, `y*/:x`) so that the operation is applied at the desired depth.

Alternatively, we could use a transposition so that the operation will be applied to the matrix columns, multiply and then transpose again to undo the effects of the first transposition `+y*+x`.

615. first atom in x

```
  x:2 2 2 2#!8
  x
(((0 1
   2 3)
  (4 5
   6 7))
 ((0 1
   2 3)
  (4 5
   6 7)))
  *:/x
0
```

This idiom is doing a monadic "first" applied repeatedly until it converges.

249. offset enumeration

```
  x:10
  y:3
  x+!y
10 11 12
  oi:{x+!y}
  oi[x;y]
10 11 12
  x:10 20 30
  y:3 4 2
  oi'[x;y]
(10 11 12
 20 21 22 23
 30 31)
  ,/oi'[x;y]
10 11 12 20 21 22 23 30 31
```

The simple enumeration starting at an offset greater than 0 (offset indices) is obtained by adding the offset to the enumeration of the required length. For a version that can be applied to vectors of offsets and lengths we need to use an "each" and then concatenate over (flatten) the results. Something interesting in this idiom is the syntax of applying an each adverb between the function and the argument list, which applies a function to tuples of items generated from successive indexing of the inputs. The following expressions are equivalent

```
  oi'[x;y]
  oi .' +(x;y) / generate tuples by transposing, then apply function
to each
  (oi').(x;y) / create the "adverbialized" function then apply it
```

236. number of occurrences of x in y

```
  y:3+7 _draw 6
  y
6 4 7 7 6 6 4
  x:7
  +/x=y
2
  x(+/=)/:y
0 0 1 1 0 0 0
```

Counting the number of occurrences becomes more complex if we have an atom and a matrix (see idiom 362). The expression `+/x=y` can be also stated as `(+/=)[x;y]`, sum over the equality. We can extend this `(+/=)` with adverbs, resulting in `x(+/=)/:y` that applies the operation to each item in `y`. This expression is more general than idiom 362 and can be applied whether the operands are an atom and a simple vector, or an atom and a matrix.

```
  y:6 6#3+36 _draw 6
  y
(7 7 5 3 6 4
 4 8 5 4 8 7
 6 5 6 7 7 8
 4 5 6 4 3 6
 5  4  5  5  7  7
 6  3  6  7  8  4)
  x:7
  x(+/=)/:y
2 1 2 0 2 1
```

Applying operations at depth

When we have arguments that are matrices sometimes we want to apply operations at a "deeper" level (e.g., on the columns of a matrix instead of its rows). The `'` adverb (and its relatives `/:` and `\:`) increase the "depth" at which operations are performed. For tensors, or other higher-dimensional data we can repeat the adverb as needed to increase this depth, e.g. `''` or `'''` for operating at deeper levels.

514. apply to dimension 1 function defined on dimension 0

```
  x:3 4#1+!12
  x
(1  2  3  4
 5  6  7  8
 9  10  11  12)
  +/x
15 18 21 24
  +/'x
10 26 42
```

This idiom compares the application of a sum over, `+/` on a matrix. Without additional adverbs the operation is performed at the outermost dimension (rows) but by adding a `'` the operation is performed one level deeper (columns)

536. rotate rows left

```
  x:3 4#1+!12
  x
(1  2  3  4
 5  6  7  8
 9  10  11  12)
  1!'x
(2  3  4  1
 6  7  8  5
 10  11  12  9)
```

Left rotation is `!` with a positive first operand, for rotating rows in a matrix we need to apply it one level deeper with `'`. This idiom is rotating the rows by 1 position.

537. rotate rows right

```
  x:3 4#1+!12
  x
(1  2  3  4
 5  6  7  8
 9  10  11  12)
  -1!'x
(4  1  2  3
 8  5  6  7
 12 9 10 11)
```

Right rotation is `!` with a negative first operand. For rotating rows in a matrix, we need to apply it one level deeper with `'`. This idiom is rotating the rows by 1 position.

444. drop first y columns from matrix x

```
  x:4 3#!12
  x
(0 1 2
 3 4 5
 6 7 8
 9 10 11)
  y:2
  y _' x
(,2
 ,5
 ,8
 ,11)
```

If we want to apply the `_` operator on a matrix at a “deeper” level, we need to add a `'`, for cutting columns we use `_'`.

204. numeric array and its negative

```
  x:3+3 4#!12
  x
(3 4 5 6
 7 8 9 10
 11 12 13 14)
  x,''-x
((3 -3
  4 -4
  5 -5
  6 -6)
 (7 -7
  8 -8
  9 -9
  10 -10)
 (11 -11
  12 -12
  13 -13
  14 -14))
```

Given a matrix `x` we can generate a tensor with `x` and its negative `-x` by applying the each adverb `'` twice to the append operation. When working with multi-dimensional data (rank 2 or higher), applying adverbs multiple times allows us to apply an operation at different dimensions.

10. Replicate at depth; add new dimension y-fold after dimension x of array z

10a. depth (n is depth at which to apply f)

```
  d:{[n;f]:[n;_f[n-1;f]';f]}
  h:{d[x;y#,:]z}
  z:2 5#"abcdefghij"
  z
("abcde"
 "fghij")
  q:h[0;3;z]
  ^q
  3 2 5
  Q
(("abcde"
 "fghij")
("abcde"
 "fghij")
("abcde"
 "fghij"))
  r:h[1;3;z]
  ^r
2 3 5
  r
(("abcde"
 "abcde"
 "abcde")
("fghij"
 "fghij"
 "fghij"))
  s:h[2;3;z]
  ^s
2 5 3
  s
(("aaa"
 "bbb"
 "ccc"
 "ddd"
 "eee")
("fff"
 "ggg"
 "hhh"
 "iii"
 "jjj"))
```

`d` is converting `z` to `z` with the specified amount of "foreach" appended to it, e.g.,

```
  d[0;z]
("abcde"
 "fghij")
  d[1;z]
("abcde"
 "fghij")'
  d[2;z]
("abcde"
 "fghij")''
  d[3;z]
```

```
("abcde"
 "fghij")'''
  d[4;z]
("abcde"
 "fghij")''''
```

inside `h` there is a `y#,:` that is replicating an argument y times, and that operation is converted to a "depth `x` foreach" by the use of `d[]`. The result is applied to `z`

Updated version (without recursion):

```
  d:{x'}/
  h:{d[x;y#,:]z}
 z:2 5#"abcdefghij"
 z
("abcde"
 "fghij")
 q:h[0;3;z]
 ^q
3 2 5
q
(("abcde"
 "fghij")
("abcde"
 "fghij")
("abcde"
 "fghij"))
 r:h[1;3;z]
 ^r
2 3 5
 r
(("abcde"
 "abcde"
 "abcde")
("fghij"
 "fghij"
 "fghij"))
 s:h[2;3;z]
 ^s
2 5 3
 s
(("aaa"
 "bbb"
 "ccc"
 "ddd"
 "eee")
("fff"
 "ggg"
 "hhh"
 "iii"
 "jjj"))
```

396. remove columns y from x

```
  y:2 3 4#1+!24
  y
((1 2 3 4
  5 6 7 8
  9 10 11 12)
 (13 14 15 16
  17 18 19 20
  21 22 23 24))
  y _di\:\: 0 2
((2 4
  6 8
  10 12)
 (14 16
  18 20
  22 24))
```

This is showing the application of an operation at a deeper level using repeated `\:` instead of `'`, so that the "added depth" is applied to the left argument instead. In this case we are using a `_di` operation, to remove the corresponding entries from each row, for each of the specified items.

Set operations

K is a great language for handling sets of data, and set operations are commonly needed. This section goes over some fundamental set operations.

497. set union

```
  x:"12345"
  y:"4567890"
  y,x[&~x _lin y]
"4567890123"
```

Concatenation is almost a union operation but in strict set theory we should not have duplicates, so we can use indexing to remove the items not in the intersection, in one of the sets and before concatenating. A simpler (although potentially less memory-efficient) idiom is to concatenate and then extract uniques `?x,y`.

498. set difference

```
  x:"12345"
  y:"4567890"
  x _dvl y
"123"
```

Set difference is a direct application of `_dvl`. The assumption is that the inputs are already sets (otherwise we can add `?` to enforce the result being a set without duplicates).

500. set intersection

```
  x:"abcdefghijxyz"
  y:"yacqwopzbx"
  x[&x _lin y]
"abcxyz"
```

By applying idiom 495 (indices of all occurrences of y in x) to the initial vector `x` we can extract the items that are common to `x` and `y`.

351. is x a subset of y

```
  x:"abgk"
  y:"abcdefghijkl"
  &/x _lin\:y
1
  x:"abgx"
  &/x _lin\:y
0
```

This is a generalization of idiom 350. Similar to 350, if the inputs are vectors the `\:` is not needed. An alternative is `(?x)~?x,y`, which takes advantage of `?` preserving the unique elements in order of appearance, and not requiring to be sorted.

348. do x and y have items in common

```
  x:"aba"
  y:"cdeac"
  x _lin\: y
1 0 1
  |/x _lin\: y
1
  y:"edge"
  |/x _lin\: y
0
```

This is a simple application of `_lin`, reduced with an `|/`. Note that we don't need the `\:` if both inputs are vectors, `|/x _lin y` is sufficient

552. which items of x are not in y

```
  x:2 3 4 5 6 7 8 9 10 11
  y:2 3 5 7 11
  ~x _lin y
0 0 1 0 1 0 1 1 1 0
```

We can obtain a 0/1 vector indicating the set difference by negating the 0/1 vector of the set intersection.

Generating data

In K we often need to generate data (e.g., vectors and matrices) with a specific pattern or shape. This section shows various techniques for generating this data. Some of these generation techniques can also be interpreted as conversion between different representations of the data (converting a description of the shape into individual items)

563. empty vector

```
  !0
!0
  ^!0
```

```
,0
  ""
""
  ^""
,0
```

This idiom is showing how to produce an empty integer vector and an empty character vector. A more general technique for producing empty vectors is using `0#` followed by an instance of the desired type. This works for integers, floats, characters and symbols.

```
  0#0
!0
  0#.0
0#0.0
  0#"a"
""
  0#`a
0#`
```

For other types, and for mixed types, producing an empty vector of the desired type is not really possible, the result of "emptying" is an empty list `()`

```
  0#.,(`a;1)
()
  0#_n
()
  0#{}
()
  0#()
()
  0#(1;`a)
()
```

513. empty matrix

```
  x:,!0
  x
,!0
  ^x
1 0
```

If we desire an empty matrix (as opposed to an empty list `()`) we can use enlisted empty vector(s) depending on the desired outermost dimension (1 in the example).

165. list of x zeros preceded by (y-x) ones

```
  x:5
  y:9
  zo:{((y-x)#1),&x}
  zo[x;y]
1 1 1 1 0 0 0 0 0
```

Generating vectors of identical numbers can be done with `#` or `&` operators. This idiom presents both. There are many alternative ways to do the same, ordered from longer to shorter we could write `{((y-x)#1),x#0}` (using `#` for both) `{(~&y-x),&x}` (using `&` in both cases), `{~&(y-`

x),x} (using a single & and negation) or {|&x,y-x} (using a single & and then reversing the results). A different technique without # or & would be using a comparison to an enumeration (y-x)>!y.

167. list of x ones preceded by (y-x) zeros

```
  x:3
  y:9
  xr:{(&y-x),x#1}
  xr[x;y]
0 0 0 0 0 0 1 1 1
  x:2 5 7 4 9 3 6
  xr\:[x;y]
(0 0 0 0 0 0 0 1 1
 0 0 0 0 1 1 1 1 1
 0 0 1 1 1 1 1 1 1
 0 0 0 0 0 1 1 1 1
 1 1 1 1 1 1 1 1 1
 0 0 0 0 0 0 1 1 1
 0 0 0 1 1 1 1 1 1)
```

Similar to 165. This can be simplified to xr:{&(y-x),x} or xr:{~(y-x)>!y}.

168. list of x zeros followed by (y-x) ones

```
  x:3
  y:9
  (&x),(y-x)#1
0 0 0 1 1 1 1 1 1
  zl:{(&x),(y-x)#1}
  zl[x;y]
0 0 0 1 1 1 1 1 1
  x:2 5 7 4 9 3 6
  zl\:[x;y]
  x:2 5 7 4 9 3 6
  zl\:[x;y]
(0 0 1 1 1 1 1 1 1
 0 0 0 0 0 1 1 1 1
 0 0 0 0 0 0 0 1 1
 0 0 0 0 1 1 1 1 1
 0 0 0 0 0 0 0 0 0
 0 0 0 1 1 1 1 1 1
 0 0 0 0 0 0 1 1 1)
```

This is very similar to idiom 167, with rearranged arguments. It can be simplified to zl:{&x,y-x} or zl: {~x>!y}.

172. list of x ones followed by (y-x) zeros

```
  x:5
  y:9
  (x#1),&(y-x)
1 1 1 1 1 0 0 0 0
  xl:{(x#1),&y-x}
  xl[x;y]
1 1 1 1 1 0 0 0 0
  x:2 5 7 4 9 3 6
  xl\:[x;y]
(1 1 0 0 0 0 0 0 0
 1 1 1 1 1 0 0 0 0
 1 1 1 1 1 1 1 0 0
 1 1 1 1 0 0 0 0 0
 1 1 1 1 1 1 1 1 1
 1 1 1 0 0 0 0 0 0
 1 1 1 1 1 1 0 0 0)
```

This is very similar to 167 but the resulting zero-vector and one-vector are concatenated in reverse. Can be simplified `xl:{~&x,y-x}` or `{x>!y}`.

407. vector length y of x 1s, the rest 0s

```
  x:5
  y:12
  @[&12;!x;:;1]
1 1 1 1 1 0 0 0 0 0 0 0
```

We can use an amend operation to convert a vector of zeros of the desired length and set items with indices `!x` to 1 `@[&x;!x;:;1]`. We could also write a shorter solution based on `&`, `~&x,y-x`. An even simpler and shorter solution (suggested by John Earnest) is `x>!y`.

250. replicate y x times

```
  x:3
  y:10
  x#y
10 10 10
  rp:{x#y}
  rp[x;y]
10 10 10
  x:3 4 2
  y:10 20 30
  rp'[x;y]
(10 10 10
 20 20 20 20
 30 30)
  ,/rp'[x;y]
10 10 10 20 20 20 20 30 30
```

Basic replication is done with `#`. For applying it to vectors of times and items to replicate we use an "each" and then flatten the results, similar to idiom 249 (see comments in that idiom).

608. zeroing an array

```
  x:1 2 3 4
  x
1 2 3 4
  x[]:0
  x
```

```
0 0 0 0
  x:2 3#!6
  x
(0 1 2
 3 4 5)
  x[;]:0
  x
(0 0 0
 0 0 0)
  x:9
  x
9
  x:0
  x
0
```

If we want to zero the items of a vector (instead of zeroing the variable that holds the vector) we can use an empty (null) index to represent "all" of the items in the corresponding dimension, and the 0 will be conformed (broadcast) to the corresponding dimension. If we want to zero the items of a matrix we need to specify a null for each of the dimensions (if we only specify one null, we would assign zero to each of the rows, which are the outermost dimension). We can also use multiplication by zero, see idiom 566.

121. y-shaped array of numbers from x[0] to x[1]-1

```
  x:4 9
  y:3 4
  (*x)+y _draw --/x
(5 8 7 4
 8 7 5 8
 8 7 7 5)
```

This idiom is showing the form of `_draw` that uses a vector as first argument. We first obtain the range by subtracting the endpoints in `x`. Since `-/` subtracts the subsequent element, we need to either reverse the elements in x, or negate the results `--x`, we then apply vectorial `_draw` that generates a random tensor of the dimensions specified by the first argument, and finally we adjust the result by adding the first endpoint.

122. y objects selected with replacement from x (roll)

```
  y:3 5
x:7
y _draw x
(6 2 1 2 2
 4 4 6 3 0
 6 3 4 5 1)
```

This is a direct application of the form of `_draw` that uses a vector as first argument, and a positive integer as the second, that generates a random tensor of the dimensions specified by the first argument and the random numbers are between 0 and `x-1` with possible repetitions.

123. y objects selected without replacement from x (deal)

```
  y:2 3
  x:7
  y _draw -x
(1 6 4
 3 5 2)
  6 _draw -6
3 0 1 5 4 2
```

This is a direct application of the form of `_draw` that uses a vector as first argument, and a positive integer as the second, that generates a random tensor of the dimensions specified by the first argument and the random numbers are between 0 and `x-1` without any repetitions for the whole tensor. If the tensor is too large and there aren't enough numbers to fill it up, the interpreter will return a length error.

123.1 normal deviates (from interval (0,1))

```
  \p 4
  x:4
  x _draw 0
0.8683 0.5104 0.968 0.9831
```

This is a direct application of the form of `_draw` that uses an integer as first argument, and zero as the second, that generates a random vector of floating point numbers in the range [0,1).

247. interlace x[i]#1 and y[i]#0

```
  x:1 3
  y:2 4
  a:,/+(x;y)
  a
1 2 3 4
  b:(#x,y)#1 0
  b
1 0 1 0
  c:a#'b
  c
(,1
 0 0
 1 1 1
 0 0 0 0)
  d:,/c
  d
1 0 0 1 1 1 0 0 0 0
  ,/(,/+(x;y))#'(#x,y)#1 0
1 0 0 1 1 1 0 0 0 0
```

In order to interlace ones and zeroes as specified by two vector sequences we can interleave the two vectors `a`, generate a vector of 1 of the same length `b`, use a "take each" to apply the interleaved vector to the vector `1 0` and flatten the result. Alternatively, we can use the `&` operator on each `x[i] y[i]` pair, then flatten and negate (because `&` produces zeroes followed by ones instead of the other way around)

```
  x:1 3
  y:2 4
  ~,/&:'+(x;y)
1 0 0 1 1 1 0 0 0 0
```

252. x alternate takes of 1s and 0s

```
  x:1 2 3 4 5
  b:(#x)#1 0
  b
1 0 1 0 1
  c:x#'b
  c
(,1
 0 0
 1 1 1
 0 0 0 0
 1 1 1 1 1)
  d:,/c
 d
1 0 0 1 1 1 0 0 0 0 1 1 1 1 1
  ,/x#'(#x)#1 0
1 0 0 1 1 1 0 0 0 0 1 1 1 1 1
```

This is a simpler variation of idiom 247, and for this case we can also use the `&` operator to make it simpler. A plain `&` would produce a vector of the required lengths but with increasing numbers instead of alternating ones and zeros. We can use a modulo and a negation to convert it to the desired output

```
   x:1 2 3 4 5
  &x
0 1 1 2 2 2 3 3 3 3 4 4 4 4 4
  (&x)!2
0 1 1 0 0 0 1 1 1 1 0 0 0 0 0
  ~(&x)!2
1 0 0 1 1 1 0 0 0 0 1 1 1 1 1
```

408. initial empty row to start matrix of x columns

```
  x:15
  ,&x
,0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  ,1.0*&x
,0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.0
  ,x#" "
,"               "
```

This idiom is presenting different ways to generate a list of `x` items of different types and then enlisting it to generate a 1-row matrix. For symbols we would do `,x#`, For the floating-point version the `0` before the `.` is not necessary `,.0*&x`.

480. replace items of x in y by 0

```
  x:1 2 3 4 5
  y:2 4
  x _lin y
0 1 0 1 0
  x*~x _lin y
1 0 3 0 5
```

We can use `_lin` to get a 0/1 intersection vector indicating whether an item in `x` exists in `y`. Multiplying `x` by the negated intersection vector will zero the items not in the intersection (we could also use an amend, although that results in a longer expression, see idiom 474 for an example).

481.replace items of x not in y by 0

```
  x:1 2 3 4 5
  y:2 4
  x _lin y
0 1 0 1 0
  x*x _lin y
0 2 0 4 0
```

This is a variation on the previous idiom, if we skip the negation step we will zero the items in the intersection.

521. matrix with x columns y

```
  x:4
  y:"abc"
  x#'y
("aaaa"
 "bbbb"
 "cccc")
```

This is applying a "take" `#` operation one level deep (to replicate columns, instead of repeating the input vector until reaching a vector of the specified length).

593. matrix of y rows, each x

```
  x:"abcd"
  y:3
  y#,x
("abcd"
 "abcd"
 "abcd")
```

We can use a dyadic `#` to take `y` copies of `x`. We need to enlist `x` so that it is repeated as a unit (instead of applying `#` to its component items).

610. y cyclic repetitions of vector x

```
  x:"abcd"
  y:3
  (y*#x)#x
"abcdabcdabcd"
```

When using a dyadic `#`, if the 1st operand is greater than the length of the 2nd operand then the items from the 2nd operand will be repeated until they reach the lenght specified in the 1st operand.

303. smearing 1s between pairs of 1s

```
  x:0 1 0 0 1 0 1 0 1 0 1 1 0
  a:(+\x)!2
  a
0 1 1 1 0 0 1 1 0 0 1 0 0
  x|a
0 1 1 1 1 0 1 1 1 0 1 1 0
  x|(+\x)!2
0 1 1 1 1 0 1 1 1 0 1 1 0
```

In this idiom we first get the parity of the running sum `a` which is a 0/1 vector where ones correspond to values between pairs of ones in `x`. If we apply a `|` (max/Boolean or) with the original vector, we obtain the desired result.

614. array with shape of y and x as its rows

```
  y:3 4#!12
  y
(0 1 2 3
 4 5 6 7
 8 9 10 11)
  x:"abcd"
  (^y)#x
("abcd"
 "abcd"
 "abcd")
```

The result from the shape operator `^` can be used as the first operand of a reshape `#` operation so that the shape of an array can be applied to a different array.

183. maximum table

```
  x:!10
  x&\:x
(0 0 0 0 0 0 0 0 0 0
 0 1 1 1 1 1 1 1 1 1
 0 1 2 2 2 2 2 2 2 2
 0 1 2 3 3 3 3 3 3 3
 0 1 2 3 4 4 4 4 4 4
 0 1 2 3 4 5 5 5 5 5
 0 1 2 3 4 5 6 6 6 6
 0 1 2 3 4 5 6 7 7 7
 0 1 2 3 4 5 6 7 8 8
 0 1 2 3 4 5 6 7 8 9)
```

This is a table where each row and each column is an enumeration limited by the row or column index. We generate it by applying a min operation `&` to each item in the enumeration.

Sorting, grading and ranking

K uses an indexed approach to sorting. When we want to sort something in K, the native operation is not sorting but grading, the indices that allow us to produce a sorted result. Grading may feel just like an intermediate result needed for sorting however, when it is exposed it opens up possibilities like applying a grading result to a different, related item (e.g. different columns in a table), or combining sort operations. The results from grading (or any other rearrangement of the indices for a vector) is called PV (Position Vector or Permutation Vector).

35. sort ascending

```
  x:10 _draw 100
  x
84 63 31 42 95 58 9 37 84 39
  <x
6 2 7 9 3 5 1 0 8 4
  x[<x]
9 31 37 39 42 58 63 84 84 95
```

After performing a grading operation, if we need the sorted data we need to index the original vector using the results from grading.

5a. sort ascending

```
  sa:{x[<x]}
  sa 30 10 40 20
10 20 30 40
  sa"popfly"
"floppy"
```

`sa` makes idiom 35 into a function.

44. sort descending

```
  x:8 _draw 10
  x
3 5 0 4 5 2 4 5
  x[>x]
5 5 5 4 4 3 2 0
```

Variation on idiom 35, descending.

37. invert permutation (the inverse puts y in ascending order)

```
  x: 7 _draw -7
  x
5 3 2 0 6 4 1
  <x
3 6 2 1 5 0 4
  / check:
  x[<x]
0 1 2 3 4 5 6
```

This is a special case of idiom 35. If `x` is already a permutation (see idiom 20), `x[<x]` will return `!#x`, and `x~<<x` is true.

268. is x in ascending order

```
  x:2 5 7 9 6 8 3
  x~x[<x]
0
  x:0 1 1 1 7 8 9
  x~x[<x]
1
```

This is comparing a list to the sorted version of itself. We can also use adverbs to compare each pair and reduce the result `~|/<':x`.

36. sort y on x

```
  x:6 _draw 10
  y:6 _draw 20
  x
9 2 3 1 9 3
  y
7 8 17 11 16 6
  y[<x]
11 8 17 6 7 16
```

We can use the results from grading (or other rearrangement) of a vector and apply them to a different vector of the same length.

34. choose grading direction

```
  x:10 _draw 100
  x
30 45 97 77 35 49 87 82 79 8
  y:1
  <x*1 -1[y]
2 6 7 8 3 5 1 4 0 9
  y:0
  <x*1 -1[y]
9 0 4 1 5 3 8 7 6 2
```

In this idiom we are choosing the grading direction by either grading the original vector or grading the vector multiplied by -1. We need to map the 1/0 choice into 1/-1, and the simplest way to do that in k is to index the vector `(1;-1)`.

33. sort matrix x on column y

```
  x:5 6 _draw 100
  x
(37 41 41 72 60 0
 91 59 5 19 17 26
 24 90 28 63 42 56
 75 67 45 14 38 49
 85 11 23 61 64 44)
  y:2
  x[;y]
41 5 28 45 23
  <x[;y]
```

```
1 4 2 0 3
  x[<x[;y];]
(91 59 5 19 17 26
 85 11 23 61 64 44
 24 90 28 63 42 56
 37 41 41 72 60 0
 75 67 45 14 38 49)
```

Sorting multiple columns is achieved by grading the column we want to sort and applying the result to every column.

8. sort rows ascending

```
  sa:{x[<x]}
  x: 3 5 _draw 10
  x
(6 3 3 9 7
 9 4 4 7 9
 9 4 7 8 9)
  sa'x
(3 3 6 7 9
 4 4 7 9 9
 4 7 8 9 9)
```

Note that this is sorting each row independently (and columns are NOT preserved).

32. sort ascending indices x according to data y

```
  x:2 3 4 5 0 1 8 7 6
  y:79 74 78 76 77 75 73 72 71
  x[<y[x]]
8 7 6 1 5 3 4 2 0
```

This idiom shows that `<y` and `x[<y[x]]` are equivalent operations (which could be useful if we already have calculated `<y[x]` for some other reason).

18. sort ascending by internal alphabet

```
  w:("once";"more";"into")
  w
("once"
 "more"
 "into"")
  w[<`$w]
("into"
 "more"
 "once")
```

Something interesting to note about this idiom is that symbols can also be ordered alphabetically.

In this case, "internal alphabet" refers to the numeric values resulting from the character from integer conversion function (`_ci` in K3).

19. sort character matrix ascending

```
  m:("scion";"icons";"coins")
  m
("scion"
 "icons"
 "coins")
  m[<m]
("coins"
 "icons"
 "scion")
```

Thanks to how the sort operation extends to symbols (see previous idiom) and vectors, the sorting of "strings" (whether they are symbols or character vectors) works in all cases.

38. sort matrix descending

```
  x:5 6 _draw 3
  x
(0 0 0 1 0 1
 1 0 2 2 0 1
 0 2 1 1 1 1
 0 0 0 1 1 2
 1 2 2 0 2 1)
  x[<x]
(0 0 0 1 0 1
 0 0 0 1 1 2
 0 2 1 1 1 1
 1 0 2 2 0 1
 1 2 2 0 2 1)
  y:"abcde"[5 6 _draw 5]
  y
("dcdbed"
 "aaaaab"
 "dcdbdc"
 "baaace"
 "eedbec")
  y[<y]
("aaaaab"
 "baaace"
 "dcdbdc"
 "dcdbed"
 "eedbec")
```

Grading can be applied to a vector of vectors, and it is possible to use it for sorting a matrix by row, or for sorting character vectors (as seen in idiom 19).

13. ascending ordinals (ranking, all different)

```
  x:15 16 13 18 15 12 13
  <<x
3 5 1 6 4 0 2
```

Performing a grading operation twice produces the indices for putting back together a vector after sorting up. These operations for generating the permutation vector rerquired to restore the original ordering after a grading operation are called ranking.

17. descending ordinals (ranking, all different)

```
  x:15 16 13 18 14 11 12
  <>x
2 1 4 0 3 6 5
```

This provides the indices for restoring the original ordering of vector(s) after sorting down. E.g.,we can perform the following operations on a columnar table of prices and timestamps:

```
  t[`p]:94 93 94 95 93
  t[`ts]:-510994118 -510994120 -510994122 -510994150 -510994119
  byTs:@[t;_n;@[;>t`ts]]
  t
.((`ts
   -510994118 -510994120 -510994122 -510994150 -510994119
   )
  (`p
   94 93 94 95 93
   ))
  byTs
.((`ts
   -510994118 -510994119 -510994120 -510994122 -510994150
   )
  (`p
   94 93 93 94 95
   ))
  @[byTs;_n;@[;<>t`ts]]
.((`ts
   -510994118 -510994120 -510994122 -510994150 -510994119
   )
  (`p
   94 93 94 95 93
   ))
```

1. ascending ordinals (ranking, shareable)

```
  x:11 17 12 13 13 13 13 18
  x[<x]?/:x
0 6 1 2 2 2 2 7
```

This idiom generates a ranking that differs from a regular ascending ordinal (idiom#13) in that when an element is repeated, this will repeat the rank for the first item found and ignore all subsequent copies. If we are sorting a table by one of its columns, and then undoing the sort operation using the results from this idiom, the resulting rows will generally NOT be a permutation of the original rows, as some items could get replaced with a duplicate of an earlier item. Example:

```
  x:11 17 12 13 13 13 13 18
  y:`a`b`c`d`e`f`g`h
  (x@<x)@x[<x]?/:x / sort, then undo the sort with this idiom
11 17 12 13 13 13 13 18
  (y@<x)@x[<x]?/:x / sort by x, then undo the sort with this idiom
`a `b `c `d `d `d `d `h
```

20. is x a permutation

```
  x:7 _draw -7
  x
0 5 1 6 4 3 2
  x~<<x
1
  x:7 _draw 7
  x
4 3 3 6 0 5 4
  x~<<x
0
```

The goal is to determine if x contains the indices for a valid permutation of a vector, that contains all numbers from !#x, and no duplicates. For a permutation, grading is the inverse operation of itself ("grading a graded permutation" recovers the original permutation). If there are missing or added elements this will no longer hold true.

4. are x and y permutations of each other

```
  x:15 16 13 18 14 11 12
  y:15 16 13 19 14 11 12
  x[<x]~y[<y]
0
  y:15 16 13 14 18 12 11
  x[<x]~y[<y]
1
```

Determining if `x` and `y` are permutations of each other can be done by sorting `x` and `y` and checking if they match.

Another option is using `_dvl` to remove items in one list from the other and counting the remaining items.

```
  x:1000000 _draw -1000000
  y:(-500000#x),500000#x
  x[<x]~y[<y]
1
  \t  x[<x]~y[<y]
58
  0=#x _dvl y
1
  \t 0=#x _dvl y
10
```

Using `_dvl` is faster but it may fail to produce correct results if the lists contain duplicates.

Merging and inserting

The indexing capabilities in K allow for some interesting techniques for merging and inserting data

27. insert 0 after indices y of x

```
  x:"abc,def,gh"
  y:(&x=","),#x
  y
3 7 10
  <(!#x),y
0 1 2 3 10 4 5 6 7 11 8 9 12
  #x
10
  (#x)><(!#x),y
1 1 1 1 0 1 1 1 1 0 1 1 0
```

This idiom returns the vector that would result from converting elements of x into ones, and then inserting a zero at all positions indicated by y. It follows on the techniques of concatenating indices and operating on the results from grading, in this case looking at the items with indices higher than the size of the original vector (that is, the items that were concatenated).

The result is a “merge control” vector, like the ones we would use as input for idiom 16 (or 11).

11. merge x, y, z, … under control of g (mesh)

```
  x:"abcd"
  y:"123456789"
  z:"zz"
  g:1 0 1 1 2 1 2 1 1 0 1 0 1 0 1
  (x,y,z)[<<g]
"1a23z4z56b7c8d9"
```

This idiom effectively “meshes” x, y and z, interleaving items from the various sources x y z. Which source is selected for the next item is based on the control vector g.

16. merge x and y by g

```
  x:5 9 8 7 4 3
  y:10 20 30 40
  g:1 0 1 1 0 0 1 0 1 1
  (x,y)[<>g]
5 10 9 8 20 30 7 40 4 3
```

This is an interesting variation on Idiom 11, by changing << (ascending ordinal ranking, indices for recovering the original order after sorting up) to <> (descending ordinal ranking, indices for recovering the original order after sorting down) we reverse the meaning of the control vector g, where 1 now will cause the merged element to be taken from the first argument and 0 will take from the second argument.

For completeness, the reverses of << and <> are >< (ascending reverse ordinal ranking, indices for reversing the original order after sorting up) and >> (descending reverse ordinal ranking, indices for reversing the original order after sorting down).

31. merge

```
  x:"egg"
  y:"mrin"
  g:1 0 1 0 1 1 0
  (x,y)[<<g]
"merging"
```

This is a variation on idiom 16, if we want to reverse the meaning of the 1/0 merge control vector we can just reverse the direction of the outermost grading operation.

482. merge x and y under control of g

```
  x:1 2 3 4 5
  y:100 200 300 400 500
  g:1 0 0 1 1 0 1 0 0 1
  (x,y)[<<g]
100 1 2 200 300 3 400 4 5 500
```

This is an application of idiom 31 to integer vectors instead of character vectors.

30. grade up x according to key y

```
  x:"fig lime"
  y:" abcdefghijklmn"
  y?/:x
6 9 7 0 12 9 13 5
  <y?/:x
3 7 0 2 1 5 4 6
  x[3 7 0 2 1 5 4 6]
" efgiilm"
```

This idiom shows how to combine find and grade for specifying a custom sort. In this example, the specified key follows a normal ordering, and the result is the same as if we just do `x@<x`. The usefulness for this idiom would come from cases where we need an unusual custom sorting order, like sorting numbers after letters, or lowercase before uppercase.

28. insert g elements h after indices y of x

```
  x:"abcd=,def=,gh="
  y:&x="="
  y
4 9 13
  g:4
  h:"x"
  a:g*#y
  a
12
  a#y
4 9 13 4 9 13 4 9 13 4 9 13
  (x,a#h)[<(!#x),a#y]
"abcd=xxxx,def=xxxx,gh=xxxx"
```

This is a generalized version of 26, see the notes in that idiom for more detail.

26. insert y "*" after "=" in x

```
  x:"abc=,d=,fgh="
  g:&x="="
  g
3 6 11
  y:5
  (x,"*")[(#x)&<(!#x),(y*#g)#g]
```

```
"abc=*****,d=*****,fgh=*****"
```

The general technique is to first create a vector with all the components in the destination, in this case `(x,"*")` and then calculating appropriate indices to insert from each.

For this idiom, since we want to insert `y` items at each location `g`, we do `(y*#g)#g)` and we insert the indices at the end (because the character to be repeated is at the end of the vector to be indexed), we then concatenate the indices of the original vector with the addition and grade the results. Since the vector being graded is larger than the original (because of the repeated characters) we need to limit the max index to `#x` (so that the added char to `x` gets repeated)

Some alternative approaches are:

Instead of `(y*#g)#g)` we can use `,/y#'g` since the order doesn't matter (because of how we will be grading the result).

Instead of limiting the max index and re-using it, we can concatenate `y*#g` copies of the item(s) to be inserted `(x,(y*#g)#"*")[<(!#x),,/y#'g]`, this produces identical results when the inserted item is 1 char long but it works differently when the pattern to insert is a character vector instead of a single character. Note that for multi-character patterns, `y` should be a multiple of the pattern length.

```
/ If given a multi-character pattern,
/ everything after the first character gets ignored
  (x,"*.")[(#x)&<(!#x),(y*#g)#g]
"abc=*****,d=*****,fgh=*****"
  y:6
/ Multi-character patterns work ok
  (x,(y*#g)#".-")[<(!#x),,/y#'g]
"abc=.-.-.-,d=.-.-.-,fgh=.-.-.-"
```

29. insert g elements h before indices y of x

```
  x:"1234,234,34"
  y:0 5 9
  g:5
  h:"*"
  a:g*#y
  ((a#h),x)[<(a#y),!#x]
"*****1234,*****234,*****34"
```

This is a variation on idiom 28. For inserting before instead of after, we just need to reverse the order of arguments in the concatenations.

Finding, grouping and selecting items

This section explores different ways for finding indices of items meeting various conditions, and for applying these indices to data and selecting/slicing elements.

22. index of first occurrence of minimum of x

```
  x:5+13 _draw 10
  x
14 12 10 9 6 12 6 11 8 12 6 13 6
```

```
  *<x
4
  x?&/x
4
```

This idiom presents two ways of achieving the result: get all the indices for sorting up and then extract the first item, or calculate the minimum then find the first occurrence of it.

23. index of first occurrence of maximum of x

```
  x:5+13 _draw 10
  x
11 10 8 8 8 7 5 9 12 12 10 12 8
  x?|/x
8
  *>x
8
```

This is the complementary operation to the previous idiom. Find the max then find the index, or get all the indices for sorting down then get the first one

80. scattered indexing

```
  x:2 3 4# _ci 97+!24
  x
(("abcd"
 "efgh"
 "ijkl")
("mnop"
 "qrst"
 "uvwx"))
  x ./:(0 0 0;1 1 3;1 2 2)
"atw"
```

We can use `.` to index at depth (e.g. `1 1 3` meaning the 4th element in the 2nd element of the 2nd item in a list of lists of lists). If we have multiple at depth indices, we can use an each-right operation to obtain the element for each one.

145. count of x between endpoints (l,h)

```
  x:5 5 _draw 100
  l:10
  h:80
  x
(66 8 6 4 86
 82 91 52 53 89
 43 0 62 3 17
 0 26 81 2 12
 37 41 41 72 60)
  xl:x>l
  xl
(1 0 0 0 1
 1 1 1 1 1
 1 0 1 0 1
 0 1 1 0 1
 1 1 1 1 1)
  xh:x<h
  xh
(1 1 1 1 0
 0 0 1 1 0
 1 1 1 1 1
 1 1 0 1 1
 1 1 1 1 1)
  xbetween:xl&xh
  xbetween
(1 0 0 0 0
 0 0 1 1 0
 1 0 1 0 1
 0 1 0 0 1
 1 1 1 1 1)
  +/'xbetween
1 2 3 2 5
```

In this idiom we calculate 0/1 vectors for the required conditions and then use min `&` to perform a "Boolean and", finally we apply a sum over each row to calculate each total.

150. sum items of x given by y

```
  x:_log[1+!5]
  x
0 0.6931472 1.098612 1.386294 1.609438
  y:1 4 1 4 1
  a:=y
  a
(0 2 4
 1 3)
  +/'x[a]
2.70805 2.079442
  +/x[0 2 4]
2.70805
  +/x[1 3]
2.079442
  +/'x[=y]
2.70805 2.079442
```

We can use group operations `=` on `y` and apply the resulting indices to `x`.

151. efficient execution of f x where x has repeated values

```
fx:{[f;x](f'x)(x?:)?/:x}

/ The phrase (x?:) is the same as (x:?x), which means
/ x is assigned the unique elements of x.
/ The phrase a?/:b means find each item of b in a.
/ So (x?:)?/:x says find each item of x in the list of
/ unique items of x (and while you're at it, reassign x
/ to that list of unique items of x). The result is
/ the list of indices of the original list x
/ into the unique items of x.

  x:1 2 3 2 3 2 1
  ?x
1 2 3
  (?x)?/:x
0 1 2 1 2 1 0
  (x?:)?/:x
0 1 2 1 2 1 0
/ This works because K's order of execution is right to left
/ The original value of x is "captured" before the (x?:) reassigns it.
  x
1 2 3
  f:{10*x}
  f'x
10 20 30
/ The last bit is simply indexing:
  (f'x)[0 1 2 1 2 1 0]
10 20 30 20 30 20 10
/ But K allows you to leave out the brackets:
  (f'x)0 1 2 1 2 1 0
10 20 30 20 30 20 10
/ So the whole thing is
  (f'x)(x?:)?/:x
```

This is a generic optimization that "memoizes" all required calculations (by taking the uniques for the input) and then assembles the results by finding each input. If we don't want to modify the input, we can add an intermediate variable `(f'u)(u:?x)?/:x`.

152. sum items of y according by ordered codes g in x

```
  y:1+!20
  y
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  z:"abcde"
  x:z[20 _draw #z]
  x
"dcbbbaceeaeccbecbaea"
  xz:z,x
  yz:(&#z),y
  xz
"abcdedcbbbaceeaeccbecbaea"
  yz
0 0 0 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  +/'yz[=xz]
54 43 50 1 62
  sc:{+/'((&#z),y)[=z,x]}
  sc[x;y;z]
54 43 50 1 62
```

This is using grouping, similar to idiom 150. It prepends the items we want to order by, to ensure the ordering of groups with a weight of zero, so that they don't impact the sums. We can eliminate the concatenations and instead use a find operation for each unique `x` and index by it `sc2: +/'y@(=x)z?/:?x`.

153. index of rows of y in corresponding rows of x

```
  x:1+3 4#!12
  x
(1 2 3 4
 5 6 7 8
 9 10 11 12)
  y:(1 0 3 0
> 0 6 0 8
> 9 0 0 12)
  x ?/:'y
(0 4 2 4
 4 1 4 3
 0 4 4 3)
```

If we do a find-each-right on each pair of vectors we will get the indices for where the items in the rows of the second matrix can be found in the first one (or, if the item cannot be found, instead of the index we will get the length of the row).

156. classify y into x classes between min and max y

```
  x:10
  y:260 416 18 27 265 336 4
/ normalize y so minimum value is 0
  sm:{x-&/x}
  sm[y]
256 412 14 23 261 332 0
/ normalize again so values are in range 0 le y and y lt x
  nr:{y*x%|/y}
  nr[x;sm[y]]
6.21 10 0.34 0.558 6.33 8.06 0
/ compare against interior range boundaries, 1+!x-1
  ~nr[x;sm[y]]</:1+!x-1
(1 1 0 0 1 1 0
 1 1 0 0 1 1 0
 1 1 0 0 1 1 0
 1 1 0 0 1 1 0
 1 1 0 0 1 1 0
 1 1 0 0 1 1 0
 0 1 0 0 0 1 0
 0 1 0 0 0 1 0
 0 1 0 0 0 0 0)
/ count the number of boundaries passed by each
  +/~nr[x;sm[y]]</:1+!x-1
6 9 0 0 6 8 0```
```

Instead of normalizing the data, we could scale the range boundaries, e.g., `|/'&:'y>\:(!x)*((|/y)-&/y)%x` or `+/'1_'y>\:(!x)*((|/y)-&/y)%x`. Which one is preferable depends on what is larger, the length of the input vector `#y` or the number of classes `x`.

182. indices of consecutive repeated elements

```
  x:"aaabccccdeee"
  =x
(0 1 2
 ,3
 4 5 6 7
 ,8
 9 10 11)
  #:'=x
 3 1 4 1 3
  &1<#:'=x
0 2 4
  {x[&1<#:'x]}=x
(0 1 2
 4 5 6 7
 9 10 11)
  re:{{x[&1<#:'x]}[=x]}
  re x
(0 1 2
 4 5 6 7
 9 10 11)
```

We can get the indices of consecutive repeated elements by applying a group `=` operation and filtering it to include only those with more than 1 item

503. indices of all occurrences of y in x

```
  x:"abcdeabc"
  y:"a"
  &x=y
0 5
```

When `y` is an atomic character, we can use an equality comparison to obtain the 0/1 vector that can be converted to indices by `&`, as discussed in the comments for idiom 495.

176. ordinal of word in x pointed at by y

```
  ow:{+/~y<1+&x=" "}
  x:"ordinal of word in x pointed at by y"
  ow[x;5]
0
  ow[x;6]
0
  ow[x;7]
0
  ow[x;8]
1
  ow[x;26]
5
```

In this idiom we compare y against the positions of each space in `x` and add up the result (which produces the count of spaces prior to `y`). We could also take the first `y` items of `x` and count the spaces `{+/#&" "=y#x}`. Note that this idiom is expecting exactly one space between words, two consecutive spaces will be processed as if a zero-sized word exists between them.

79. index (1-origin) of first non-blank, counting from rear

```
  x:"blanks at end "
  (" "=x)
0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 1
  (" "=x) _sv 1
4
```

What happens when we reverse the inputs to the `_sv` operation? The documentation indicates that `_sv` is doing polynomial evaluation by applying `_sv: {{z + y * x}/[0; x; y]}`. The result of using a vector as first argument and an integer as the second is an evaluation of the binomial `y+x[i]*previous_result` over the successive values of the first argument.

If the input is a 1/0 vector this successive evaluation will “reset” to 1 whenever `x[i]` is a zero, and when the evaluation gets to a part of the vector that no longer has zeroes it will add their value, producing a final result of one plus the count of non-zero trailing items

We can use this result as an alternative way to count trailing spaces, by applying it to the result of `(" "=x)`.

If we compare this technique with other ways of counting trailing blanks like `ctb` in idioms 73a and 76a, what’s interesting in this idiom is that it doesn’t need to perform a reverse on the input vector.

261. first group of 1s

```
  x:1 1 1 0 1 0 1
  x&&\x=|\x
1 1 1 0 0 0 0
  x:0 0 0 1 1 0 1
  x&&\x=|\x
0 0 0 1 1 0 0
  x:0 1 0 1 0 1 0
  x&&\x=|\x
0 1 0 0 0 0 0
```

This is applying a running max `|\` operation to get a 0/1 vector that identifies the start of the first group of ones (zeros before the start, ones afterwards), then doing an element-wise equality comparison with the original vector, which will convert to ones the initial group of zeros in the original vector and leave the rest unmodified. If we apply a running min `&\` at this point, we will get a 0/1 vector with ones for the initial group of zeros and ones and zeros afterwards, which we can use to filter the original using a min operation (`&`, equivalent to Boolean “and”).

284. sum items of x marked by y

```
  x:1 2 3 4 5 6 7
  y:1 1 1 2 2 3 3
  =y
(0 1 2
 3 4
 5 6)
  x[=y]
(1 2 3
 4 5
 6 7)
  +/'x[=y]
6 9 13
  y:1 2 1 3 3 2 1
  +/'x[=y]
11 8 9
```

If we group by `y` and use those groups to index `x` we get the items of `x` marked by `y` and then we can just sum over them to obtain the desired result.

305. invert fields marked by pairs of 1s

```
  x:1 0 1 0 0 1 0 0 1
  a:(+\x)!2
  a
1 1 0 0 0 1 1 1 0
  ~x
0 1 0 1 1 0 1 1 0
  (~x)&a
0 1 0 0 0 0 1 1 0
  (~x)&(+\x)!2
0 1 0 0 0 0 1 1 0
```

In this idiom we use the parity of the running sum to identify items between pairs on ones and then apply a min operation with the complement of the original vector. This will set to 0 any items that were ones in the original vector or that are not between pairs of ones.

307. invert all 1s after 1st 1

```
  x:0 0 1 1 0 1
  &#x
0 0 0 0 0 0
  x?1
2
  @[&#x;x?1;:;1]
0 0 1 0 0 0
```

This is using an amend operation to modify a vector of zeros of the desired length `&#x` and setting to 1 just the location of the first one. We could also have done it by shifting the vector by one position and comparing to the original `x>0,-1_|\x`.

308. invert all 0s after 1st 0

```
  x:1 0 0 1 1 0
  a:x?0
  a
1
  !#x
0 1 2 3 4 5
  b:(a+1)_!#x
  b
2 3 4 5
  @[x;b;:;1]
1 0 1 1 1 1
  @[x;(1+x?0)_!#x;:;1]
1 0 1 1 1 1
```

This is calculating the indices for all the items after the first one, and using an amend for setting those entries to 1.

330. index of 1st occurrence of maximum item of x

```
  x:5 3 7 0 5 7 2
  x?|/x
2
```

This is a straightforward application of max over `|/` and find first `?`.

336. index of first occurrence of minimum

```
  x:5 3 7 2 5 3
  x?&/x
3
```

This is just an application of min over, and a find operation.

338. locate first occurrence in x of an item of y

```
  x:"abcdef"
  y:"dbf"
  &/x?/:y
1
  x[1]
"b"
```

This is applying a “find each right” operation and then finding the minimum within the results.

333. quick membership for nonnegative integers

```
  x:5 3 7 2
  y:8 5 2 6 1 9
  &1+|/x,y
0 0 0 0 0 0 0 0 0 0
  a:&1+|/x,y
  @[a;y;:;1]
0 1 1 0 0 1 1 0 1 1
  (@[a;y;:;1])[x]
1 0 0 1
  @[&1+|/x,y;y;:;1][x]
1 0 0 1
```

This idiom is building an identity mask with a length that can accommodate both inputs (a 0/1 vector with as many elements as the max of `x` and `y`, and with ones set at the positions described by `y`) and then indexing into this mask to determine if the item in `x` is present also in `y`. This technique may be fast, but it is not well suited as a general approach and we need to evaluate if it is appropriate for a specific use case. This technique requires all inputs to be positive, and it can also have significant inefficiencies depending on the input data (e.g., space allocation if the range of `x` or `y` is very large).

381. invert all but 1st 1 in group of 1s

```
  x:0 0 1 1 1 0 1 1 0 1
  x>-1 _ 0,x
0 0 1 0 0 0 1 0 0 1
  m381:{x>-1 _ 0,x}
  m381[x]
0 0 1 0 0 0 1 0 0 1
```

This is using a `>` comparison between the original vector and its right shifted version (prepending a zero and removing the last item) for identifying the transitions from 0 to 1. Alternatively, we can do `x[0],>':x`, applying a comparison of successive elements and prepending the first element.

437. remove leading zeros

```
 x:"000023456003450000"
  ((x="0")?0)_ x
"2345600345000"
```

We can remove the leading zeros by finding the index of the first nonzero (where `=0` becomes zero again) and cutting that many elements. If the number is not larger than `0I-1` we can remove leading zeros by converting into an int and then back into a string `$0$x`

438. index of 1st 1 following index y in x

```
  x:1 0 0 1 1 0 1 1 0
  y:3
  y+(y _ x)?1
3
  y+((y+1)_ x)?1
3
  (y+1)+((y+1)_ x)?1
4
  / a more K-like alternative
  &x
0 3 4 6 7
  y <&x
0 0 1 1 1
  (&x)[*&y<&x]
4
```

This is showing two approaches for solving the problem. First it cuts `y+1` elements, finds a one in the cut vector and adjusts the result by adding back the count of cut items `y+1`. The second approach is finding the location of all ones, comparing them to the target and taking the first one that is greater. Yet another alternative is `(x&y<!#x)?1` which is generating a 0/1 vector of the same length as `x`, with zeros in the first y positions and ones elsewhere, using it as a mask to zero the beginning of `x` (with a min operation) and then finding the first one. Something important to note is how these alternatives will signal that no ones were found. `(y+1)+((y+1)_ x)?1` and `(x&y<!#x)?1` rely on `?` and will return an index one higher than

the length of x, while (&x)[*&y<&x] relies on indexing and will return 0, which is undesirable if y can potentially be -1.

439. last 1s in groups of 1s

```
   x:0 1 1 0 1 1 1 0 0 1
   x>1 _ x,0
0 0 1 0 0 0 1 0 0 1
```

The last ones in groups of ones are the points where the next item is lower. We can left shift the vector and compare, to get the desired result

440. 1st 1 in groups of 1s

```
   x:0 1 1 0 1 1 1 0 0 1
   x>-1_ 0,x
0 1 0 0 1 0 0 0 0 1
```

The first ones in groups of ones are the points where the previous item is lower. We can right shift the vector and compare, to get the desired result. We can also use a greater each prior, and prepend the first item (*x),>':x.

466. remove every y-th item of x

```
   x:4+!10
   x
4 5 6 7 8 9 10 11 12 13
   y:3
   !#x
0 1 2 3 4 5 6 7 8 9
   (!#x)!y
0 1 2 0 1 2 0 1 2 0
   ~0=(!#x)!y
0 1 1 0 1 1 0 1 1 0
   &~0=(!#x)!y
1 2 4 5 7 8
   x[&~0=(!#x)!y]
5 6 8 9 11 12
```

The 0/1 vector indicating every y-th items is produced by first comparing an enumeration of all the indices, calculating the remainder of dividing them by y and then checking where the result is zero 0=(!#x)!y. The indices of all the non-y-th items is given by &~0=(!#x)!y, and once we have the indices we can use a simple indexing operation on the original vector.

467. select every y-th item of y

```
   x:4+!10
   x
4 5 6 7 8 9 10 11 12 13
   y:3
   !#x
0 1 2 3 4 5 6 7 8 9
   (!#x)!y
0 1 2 0 1 2 0 1 2 0
```

```
  0=(!#x)!y
1 0 0 1 0 0 1 0 0 1
  &0=(!#x)!y
0 3 6 9
  x[&0=(!#x)!y]
4 7 10 13
```

This is a variation on the previous idiom, if we skip the negation of the 0/1 vector we will select every y-th item (instead of removing them).

469. remove every second item

```
  x:"abcdefghijklmn"
  (!#x)!2
0 1 0 1 0 1 0 1 0 1 0 1 0 1
  &(!#x)!2
1 3 5 7 9 11 13
  x[&(!#x)!2]
"bdfhjln"
```

This is an application of idiom 466, setting `y:2`.

471. index of first occurrence of g in x (circularly) after y

```
  x: 15 _draw 10
  x
6 6 0 0 8 9 8 1 0 2 9 4 6 3 5
  g:0 6 5
  y:9
  (y+(y!x)?g)!#x
9
  (y+(y!x)?/:g)!#x
2 12 14
```

In this idiom we want a circular search after `y`, a way to do that is to rotate the search space `y` positions `(y!x)` and performing the search `?/:` (or just `?` if `g` is a atomic instead of a vector). The resulting indices need to be adjusted, e.g., by first adding back the number of positions rotated and then calculating the remainder of dividing by the count of items in `x`, to reduce any indices that may be higher than the length of the original vector (because of the indices “wrapping around” after the addition). Note that this will produce incorrect results if `g` is not present in `x`.

512. select items of x according to markers in y

```
  x:2 3 4#1+!24
  x
((1 2 3 4
  5 6 7 8
  9 10 11 12)
 (13 14 15 16
  17 18 19 20
  21 22 23 24))
  y:1 0 0 1
  x[;;&y]
```

```
((1 4
  5 8
  9 12)
 (13 16
  17 20
  21 24))
```

We can convert a 0/1 selection vector into a vector of indices by using `&`, and once we have a vector of indices, we can use plain indexing at the desired level (the innermost dimension in this example) to slice (select from) the original vector/matrix/tensor.

530. index of last occurrence of y in x

```
  x:10 _draw 5
  x
3 0 4 3 1 4 4 3 3 1
  y:4
  *|&x=y
6
  y:3
  *|&x=y
8
```

We can find the index of the last occurrence of `y` in `x` by finding all the indices `&y=x` and then taking the first of the reversed list. Alternatively we could reverse the vector, find the index for the first occurrence, add one and then subtract from the total length `(#x)-1+(|x)?y`.

531. replace each item of y with index of its last occurrence in x

```
  x:"aabbbcccc"
  y:x,"ddd"
  x
"aabbbcccc"
  y
"aabbbccccddd"
  (|x)?/:y
7 7 4 4 4 0 0 0 0 9 9 9
  #x
9
  (#x)-(|x)?/:y
2 2 5 5 5 9 9 9 9 0 0 0
  0|-1+(#x)-(|x)?/:y
1 1 4 4 4 8 8 8 8 0 0 0
```

This idiom is a variant of the alternative idiom in the comments of the previous one. Instead of an atom, `y` is a vector and we are finding each of the items in it. This idiom is further limiting the indices to be always 0 or greater (otherwise items that are not found will return a -1). It is unclear what is the purpose of this limitation, as 0 is the index of a valid item in x which does not match the corresponding item in y.

532. index of last occurrence of y in x, counted from the rear

```
  x:8 4 9 1 5 7
  y:8 2 3 4 9 5 7 1 10 6 8 2
  (|x)?/:y
5 6 6 4 3 1 0 2 6 6 5 6
```

This is a simpler variation on the theme from the two previous idioms, if we just need an offset from the end of the vector we don't need to adjust the reverse find by subtracting it from the length.

551. index of first differing item of x and y

```
  x:3 1 4 1 6 0
  y:3 1 4 1 5 9
  (~x=y)?1
4
```

`(~x=y)` will return a 0/1 vector with ones in differing elements, and we can just find the index for the first 1. For an even simpler expression we can skip the negation and find the first 0 `(x=y)?0`.

554. select from g based on index of x in y

```
  g:("William Shakespeare"
> "John Milton"
> "Jonathan Swift"
> "Jane Austen"
> "John Keats"
> "Charles Dickens")
  y:1564 1608 1667 1775 1795 1812
  x:1775
  g[y?x]
"Jane Austen"
```

We can apply the results of `?` for indexing a different list. A columnar table is a data structure where each column is represented by a vector (typically organized as a dictionary of columns where the keys are the column names and the values are the column vectors). We can select rows in the table by selecting items from each column using indices obtained from operations on any of the columns in the table. The column used to calculate selection criteria does not have to be the same as the column where we are selecting values.

567. select x or y depending on g

```
  x:`hot `white `short `old
  y:`cold `black `tall `young
  g:1 0 0 1
  x,'y
(`hot `cold
 `white `black
 `short `tall
 `old `young)
  (!#g),'g
(0 1
 1 0
 2 0
 3 1)
  (x,'y) ./: (!#g),'g
```

```
`cold `white `short `young
```

We can concatenate the input vectors `x` and `y` as columns in a matrix by using `,'` (as an alternative, we could also have transposed them as a list of rows, `+(x;y)`). The desired index pairs can be calculated by enumerating the indices and concatenating the values `(!#g),'g` and then we can use indexing at depth for each pair to obtain the desired result. A simpler expression can be produced by applying (shallow or at depth) the values in `g` as indices to each row `(x,'y)@'g` or `(x,'y).'g`.

574. y where x is 0

```
  x:0 7 8 0 2
  y:10 4 6 7 3
  x+y*x=0
10 7 8 7 2
```

We can multiply `y` by `x=0` to zero-out all positions in `y` where `x` is not 0, then a simple element-wise addition with `x` will produce the desired result. Alternatively we can use an amend operation `@[x;ix;:;y@ix:&x=0]` although that is a longer expression.

181. which class of y x belongs to

```
  cl:{-1++/x>/:y}
  x:87 9 931 7 27 92 654 1416 7 911
  y:0 50 100 1000
  +/x>/:y
2 1 3 1 1 2 3 4 1 3
  -1++/x>/:y
1 0 2 0 0 1 2 3 0 2
  cl[x;y]
1 0 2 0 0 1 2 3 0 2
```

By adding up how many values in `y` are smaller we can classify items in `x` according to `y`. Note that even if `y` is unsorted the resulting class numbering will be the same as if `y` were sorted.

587. first column as a matrix

```
  x:(0 1 2 3
>4 5 6 7
>8 9 10 11)
  x[;,0]
(,0
 ,4
 ,8)
```

We can extract the n-th column from a matrix `x` by indexing at the second dimension `x[;n]`. If we need the result as a columnar matrix instead, we could have applied an enlist operation to each row, but we can get an even simpler expression by enlisting the index.

602. choosing according to sign

```
  sg:{(x>0)-(x<0)}
  sg -4.5 0 6.78
-1 0 1
```

```
  y:"-0+"
  x:-54
  y[1+sg[x]]
"-"
  x:0
  y[1+sg[x]]
"0"
  x:1234.5
  y[1+sg[x]]
"+"
```

The sign function `sg` returns -1, 0 or 1 depending on the sign of its input. If we want to use its output to choose between 3 items depending on sign, we can convert the output range into 0,1,2 indices by adding 1.

607. vector from column of matrix

```
  x:3 4#!12
  x
(0 1 2 3
 4 5 6 7
 8 9 10 11)
  x[;0]
0 4 8
```

We can extract a column by indexing into the 2nd dimension of a matrix. Leaving the 1st dimension empty (null) represents the "all" selection. If we want to express "none", instead of null we would use an empty list.

623. conditional change of sign

```
  x:-9
  y:0
  x*-1^y
-9.0
  y:1
  x*-1^y
9.0
```

This idiom shows how we can use the powers of -1 to convert 0/1 to 1/-1. After the conversion we can just use multiplication to perform a conditional change of sign. Other alternatives, ordered by increasing length, could be: indexing of a constant vector `x*1 -1@y`, an “over form of DO” `y(-1*)/x` or a conditional `:[y;-x;x]`.

String and vector manipulation

K supports two ways to represent strings, as symbols (complete “words” or blocks of text, treated as a unit) or as character vectors (allowing access to individual characters). Some of the techniques in this section are specific for text manipulation but there are also many of the techniques that can be useful for both character and other types of vectors (numeric, mixed) as well.

25. Doubling quotes

```
  f:{_ssr[x;"\"";"\"\""]}
  a:"Did he say, \"Hello\"?"
  f a
"Did he say, \"\"Hello\"\"?"
```

This is a straightforward application of string search and replace

42. move all blanks to end of text

```
  x:" sign if i cant "
  x[<" "=x]
"significant "
```

This idiom generates a 0/1 vector with items intended for moving to the end having a 1, it then takes advantage of the grade up operation producing ascending indices whenever the items are identical, so that items marked with 0 remain in the same order, and items marked with 1 move to the end.

160. move blanks in x to end of list

```
  x:"sign if i cant"
  x[<x=" "]
"significant "
  be:{x[<x=" "]}
  y:("yo ho ho"
  "and a bottle"
  "of rum")
  be'[y]
("yohoho "
 "andabottle "
 "ofrum ")
```

We can use `'` to apply the the technique presented in idiom 42 to each character vector in a list of character vectors.

43. move elements of x with characteristic y to beginning

```
  x:"mjinase"
  y:0 1 0 0 1 1 0
  x[>y]
"jasmine"
```

This is a generalized version of idiom 42, it uses a characteristic for generating a 0/1 vector and then indexes based on grading so that items marked with 1 move in the direction of the grading. The example just uses a fixed 0/1 vector, but in practice it would be generated from an operation like `x _in "jas"`.

73. remove trailing blanks

```
x:"trailing blanks    "
```

73a. negative count of trailing blanks

```
  nctb:{-+/&\|" "=x}
  nctb x
-4
  rtb:{(nctb x)_ x}
  rtb x
"trailing blanks"
```

The first part `nctb` reverses the string, gets a 0/1 vector of which are spaces, gets the running minimum value (see idiom 3 for other application of min scan) to zero the ones corresponding to blanks that are non-trailing, adds the count of trailing blanks and finally negates the result.

The second part `rtb` just cuts `nctb x` items from `x` (since `nctb x` is negative, it will cut from the end of `x`)

Note that the removal operation can also be done by indexing instead of counting & cutting

```
  {x@&~|&\|" "=x}"trailing blanks    "
"trailing blanks"
```

76. justify right

```
  x:"trailing blanks "
```

76a. negative count of trailing blanks

```
  nctb:{-+/&\|" "=x}
  nctb x
-4
  rj:{(nctb x)!x}
  rj x
" trailing blanks"
  x:("a ";"bc ";"d e ";"fg h ";"ij kl ";"mnopqr")
x
("a "
 "bc "
 "d e "
 "fg h "
 "ij kl "
 "mnopqr")
  rj'x
(" a"
 " bc"
 " d e
 " fg h"
 " ij kl"
 "mnopqr")
```

This is a variation of idiom 73 that uses rotate instead of cut in order to move the trailing spaces to the beginning of the character vector.

If we wanted to do this through indexing, we could use a grade up `<` instead of where `&` so that the indices for the trailing spaces get sorted to the front

```
  {x@<~|&\|" "=x}"trailing blanks "
" trailing blanks"
```

A more efficient implementation (courtesy of S Apter) based on find `?` is

```
  a:"abc    "
  a
```

```
"abc    "
  " "=a
0 0 0 1 1 1 1
  |" "=a
1 1 1 1 0 0 0
  (|" "=a)?0                  / finding 0 (on the reverse)
4
  &\|" "=a                    / is the same as max-scan and ...
1 1 1 1 0 0 0
  +/&\|" "=a                  / ... then summing (on the reverse)
4
  -(|" "=a)?0                 / negate
-4
  (-(|" "=a)?0)!a             / rotate
"    abc"
```

147. locations of string x in string y (including overlaps)

```
  x:"**"
  y:"*abcugj**jy***kmhix**12"
  ss:{z[&y[z+\:!#x]~\:x]}
  ss[x;y;&((1-#x)_ y)=*x]
7 11 12 19
  y[7 11 12 19+\:0 1]
("**"
 "**"
 "**"
 "**")
```

`&((1-#x)_ y)=*x` is a list of candidate start locations for a match. At these points the first character of x matches and there are enough characters afterwards as to make a match possible

`z+\:!#x` is a list of vectors of indices, starting at the candidate start locations, of the same length as x

`y[z+\:!#x]~\:x` are the indices for the candidate locations that are actual matches.

Finally, `z[&y[z+\:!#x]~\:x]` filters the list of candidates, leaving only those that match.

Something interesting to note is that unlike the built-in `_ss` this idiom does not treat any characters as special, we can use it to match asterisks or other characters that would be normally used as wildcards (`_ss[y;x]` produces a `nonce` error)

184. right justify fields x of length y to length g

```
  x:"abcdefghij"
  y:2 3 4 1
  g:5
  a:+\0,-1_ y
  a
0 2 5 9
  a _ x
("ab"
 "cde"
 "fghi"
 ,"j")
  (g#" "),/:a _ x
("      ab"
 "      cde"
 "      fghi"
 "      j")
  b:(-g)#/:(g#" "),/:a _ x
  b
("   ab"
 "  cde"
 " fghi"
 "    j")
  ,/b
"   ab  cde fghi    j"
  rj:{[x;y;g],/(-g)#/:(g#" "),/:(+\0,-1 _ y) _ x}
  rj[x;y;g]
"   ab  cde fghi    j"
```

`a:+\0,-1_ y` converts the vector of lengths to indices of starting points, which we can use for a cut `_` operation. `(g#" "),/:a _ x` prepends `g` spaces to each cut, and if we take the last `g` items from each resulting string we obtain the right-justified cuts, which we can concatenate into a single string.

185. left justify fields x of length y to length g

```
  x:"abcdefghij"
  y:2 3 4 1
  g:5
  a:+\0,-1_ y
  a
0 2 5 9
  a _ x
("ab"
 "cde"
 "fghi"
 ,"j")
 ((+\0,-1_ y) _ x),\:g#" "
("ab     "
 "cde     "
 "fghi     "
 "j     ")
  g#/:((+\0,-1_ y) _ x),\:g#" "
("ab   "
 "cde  "
 "fghi "
 "j    ")
  lj:{[x;y;g],/g#/:((+\0,-1_ y)_ x),\:g#" "}
  rj[x;y;g]
"   ab  cde fghi    j"
  lj[x;y;g]
"ab   cde  fghi j    "
```

This is a variation on idiom 184, the difference is that instead of prepending spaces we append them and instead of taking items from the end we take from the beginning.

217. index of last nonblank in string

```
  x:("love's not "
> "time's fool "
> "though rosy ")
  ln:{*|&~" "=x}
  ln'x
9 10 10
```

This idiom is finding the items that don't match a space, getting their indices, reversing the resulting list and then outputting the first. `*|` is a convenient way to get the last item in a list but if the list is long it is preferable using `*-1#`

248. center text x in line of width y

```
  x:"1234567890"
  y:16
  a:y#x,y#" "
  a
"1234567890      "
  b:_ -0.5*y-#x
  b
-3
  c:b!a
  c
"   1234567890   "
  ct:{(_ -0.5*y-#x)!y#x,y#" "}
  ct[x;y]
"   1234567890   "
  p:("1234567890";"1234";"123456";"1234567890123456")
  p
("1234567890"
 "1234"
 "123456"
 "1234567890123456")
  ct\:[p;y]
("   1234567890   "
 "      1234      "
 "     123456     "
 "1234567890123456")
```

One way to center text is to append enough spaces (append as many as the length, then take as many characters as the length), and then rotate by half the amount of appended spaces. Note that this assumes that all original lines are shorter or equal than the specified length, it will produce incorrect results if it is longer.

259. removing leading and trailing blanks

```
  x:"   abcd e  fg    "
  a:~x=" "
  a
```

```
0 0 0 1 1 1 1 0 1 0 0 1 1 0 0 0
  (|\a)&(||\|a)
0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0
  &(|\a)&(||\|a)
3 4 5 6 7 8 9 10 11 12
  x[&(|\a)&(||\|a)]
"abcd e fg"
  lt:{x[&(|\a)&||\|a:~x=" "]}
  lt x
"abcd e fg"
```

If we compare the characters the input to not spaces, then get the minimum of the forward and backward running max, we can obtain a 0/1 vector with zeros for the leading and trailing spaces and ones for the other characters. We can then apply `&` and index the original character vector to remove leading and trailing blanks.

264. insert x[i] blanks after y[g[i]]

```
  x:3 2 1
  y:"abcdefg"
  g:2 4 6
  b:(0,g)_ y
  b
("ab"
 "cd"
 "ef"
 ,"g")
  c:b,'(x,0)#\:" "
  c
("ab   "
 "cd  "
 "ef "
 ,"g")
  ,/c
"ab   cd  ef g"
  ib:{[x;y;g],/((0,g)_ y),'(x,0)#\:" "}
  ib[x;y;g]
"ab   cd  ef g"
```

This is cutting `y` at the locations for inserting blanks, appending the required number of zeros to each cut, then concatenating the result.

266. remove trailing blanks

```
  x:"  phrase 266   "
  a:~x=" "
  a
0 0 1 1 1 1 1 1 0 1 1 1 0 0 0
  b:||\|a
  b
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
  x[&b]
" phrase 266"
```

This is a simpler variation on idiom 259. See idiom 73 for another variation (using counts to remove items, instead of indexing).

267. remove leading blanks

```
  x:"  phrase 267  "
  a:~x=" "
  a
0 0 1 1 1 1 1 1 0 1 1 1 0 0
  b:|\a
  b
0 0 1 1 1 1 1 1 1 1 1 1 1 1
  x[&b]
"phrase 267 "
```

This is another variation on idiom 259.

283. locate field y of fields beginning with first of x

```
  x:"abcabbbaccccaddd"
  y:2
  y=+\x=*x
0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0
  x[&y=+\x=*x]
"abbb"
  y:4
  x[&y=+\x=*x]
"addd"
```

If we compare x to its first char and then apply a running sum we get the index of the field they're in. After that, we can just compare to the desired field index and index the input where there is a match.

293. locate quotes and text between them

```
  x:"abc\"de\"f"
  #x
8
  a:x="\""
  a
0 0 0 1 0 0 1 0
  b:&a
  b
3 6
  c:(*b)+!1+--/b
  c
3 4 5 6
  @[&#x;c;:;1]
0 0 0 1 1 1 1 0
```

This works only in the case that there is a pair of double quotes (no more no less). First we use an element-wise comparison to a quote and find the indices, then we enumerate the indices between the two values (first index plus the enumeration of 1 + last index – first index) and finally we convert indices to a 0/1 vector (we amend a vector of zeros of the size of `x` and set to 1 the items at the indices).

294. locate text between quotes

```
  x:"abc\"de\"f"
  #x
8
  a:x="\""
  a
0 0 0 1 0 0 1 0
  b:&a
  b
3 6
  c:b+1 -1
  c
4 5
  d:(*c)+!1+--/c
  d
4 5
  @[&#x;d;:;1]
0 0 0 0 1 1 0 0
```

This is a variation on the previous idiom, just adding a vector `1 -1` to the indices, to remove the quotes themselves from the result.

295. depth of parentheses

```
  x:"a(b((c)de)f)g(h)"
  dp:{+\("("=x)--1 _ 0,")"=x}
  dp x
0 1 1 2 3 3 3 2 2 2 1 1 0 1 1 1
  x:"a(b((cde)f)g(ki)"
  dp x
0 1 1 2 3 3 3 3 3 2 2 1 2 2 2 2
  x:"ab((c)de)f)g(ki)"
  dp x
0 0 1 2 2 2 1 1 1 0 0 -1 0 0 0 0
  x:"a(b((c)de)f)g(h)"
  dp x
0 1 1 2 3 3 3 2 2 2 1 1 0 1 1 1
```

This is taking the element-wise comparisons to opening and closing parentheses, shifting the closing parentheses by one position (prepend a zero and cut the last) and applying a sum-scan operation (successive partial sums) to obtain the desired result.

297. spread marked field heads right

```
  x:"abcdef"
  y:1 1 0 0 1 0
  a:&y
  a
0 1 4
  b:#:'a _ x
  b
1 3 2
  c:b#'a
  c
(,0
 1 1 1
 4 4)
  d:,/c
  d
0 1 1 1 4 4
  x[d]
"abbbee"
  x[,/(#:'a _ x)#'a:&y]
"abbbee"
  sh:{x[,/(#:'a _ x)#'a:&y]}
  sh[x;y]
"abbbee"
```

This is removing the items at positions marked with a zero and, if there are any elements to its left marked with a one, then inserting copies of it in place of the removed items. This is done by cutting the input at the marked locations, getting the length of each cut and using those lengths to replicate the indices of the cutpoints. The result from the previous operation is a vector of indices that we can apply to the input to obtain the desired result. In the example this is applied to a character vector but it could be applied to any vector, and this could be useful for "filling the gaps" in a series (e.g. if we have a time series of quotes for all days of the month and we want to replace quotes on holidays and weekends by "spreading" the quotes from the last previous business day).

377. fill x to length y with x's last item

```
  x:"quiz"
  y:9
  a:(!y)&-1+#x
  a
0 1 2 3 3 3 3 3 3
  x[a]
"quizzzzzz"
  x[(!y)&-1+#x]
"quizzzzzz"
  / or, an alternative way:
  y#x,y#*|x
"quizzzzzz"
```

This idiom presents a solution based on indexing, by applying a max `&` operation to `(!y)` to limit it to `-1+#x`, and a solution based on appending a vector that repeats the last element `y` times and then taking `y` items from the result. As a third alternative we can use max scan and indexing `x@|\y#!#x`.

379. remove leading, multiple and trailing y's from x

```
  x:0 0 1 2 0 0 3 4 0 5 0 0 0
  y:0
  a:x=y
  b:~a&1!a
  a
1 1 0 0 1 1 0 0 1 0 1 1 1
  b
0 1 1 1 0 1 1 1 1 1 0 0 0
  x[&b]
0 1 2 0 3 4 0 5
  a[0]_ x[&b]
1 2 0 3 4 0 5
```

`a` is a 0/1 vector with the location of remove candidates. From those candidates we eliminate those preceded by a non-candidate by rotating the list of candidates and applying a min operation with the original and then negating the result. If there are leading instances of `y`, this will still leave one undesired leading char, which we can then remove by cutting `a[0]` items from the beginning. This cutting can be done after indexing, or before `x@a[0]_ &b`. This can be useful for removing and compressing whitespace in a character vector.

380. change items of x with value y[0] to y[1]

```
  x:"abcde"[15 _draw 5]
  x
"ddaeecadbbcbedc"
  y:"d "
  @[x;&x=*y;:;*|y]
"  aeeca bbcbe c"
```

Amend operations provide us with a good way to change specific items in a vector. Something important to note is how the operation behaves differently depending on the type of the first argument. If it is a variable name like this example, the operation will return the modified result. If instead it is a "handle" (a symbol that describes the location of a variable in the k tree) then the contents of the variable will be modified first and then the return value will be the handle being modified, e.g.,

```
  x:"abcde"[15 _draw 5]
  x
"ddaeecadbbcbedc"
  y:"d "
  @[`x;&x=*y;:;*|y]
`x
  x
"  aeeca bbcbe c"
```

This second form will only work for variables defined in the global k tree, that is, if used inside a function it will not work for modifying local variables.

We can observe the relationship between amending and indexing by comparing to `x[&x=*y]:y[1]`.

As a side note, the original version of this idiom requires that `y` has exactly one or 2 elements (if it has only one then nothing will be changed). If `y` can be 2 elements or longer we should use `y[1]` instead of `*|y`.

382. insert x in y after index g

```
  x:1 2 3
  y:10*1+!7
  y
10 20 30 40 50 60 70
  g:3
  ((g+1)#y),x,(g+1)_ y
10 20 30 40 1 2 3 50 60 70
```

This is a variation on idiom 375. We can also use an indexing-based solution `(y,x)[<(!#y),(#x)#g]`.

386. shift x right y, fill 0

```
  x:1+!12
  x
1 2 3 4 5 6 7 8 9 10 11 12
  y:3
  @[(-y)!x;!y;:;0]
0 0 0 1 2 3 4 5 6 7 8 9
```

This idiom is first rotating `x` by `y` positions, then amending the first `y` positions to be zero. We can also use prepend and take like this `(#x)#(y#0),x`.

387. shift x left y, fill 0

```
  x:1+!12
  x
1 2 3 4 5 6 7 8 9 10 11 12
  y:3
  @[y!x;((#x)-y)+!y;:;0]
4 5 6 7 8 9 10 11 12 0 0 0
```

This is a variation on the previous idiom, in the opposite direction. We can use cuts and appends instead, `y _ x,y#0`.

401. first word in string x

```
  x:"twas brillig and the slith"
  x?" "
4
  (x?" ")#x
"twas"
  fw:{(x?" ")#x}
  fw x
"twas"
```

We can extract the first space-separated word in a character vector by finding the index of the first space in the character vector and taking as many characters as that index. Note that this will work even if the string does not have any spaces (it will return the entire string) but it will not work for separators other than space (e.g., punctuation will be treated as part of the word), which require a "find each right" `(&/x?/:" \t\n\r.,;:!?")#x`.

424. single blank from multiples

```
  x:"a  b   c     d"
  x[&a|1 _ 1!1,a:~" "=x]
"a b c d"
```

This is calculating and applying the indices of items that are either a nonblank or a space preceding a nonblank (using a rotation of the 0/1 vector resulting from determining the nonblanks). A 1 is appended (prepended prior to the rotation) so that if the character vector ends in blanks, one of them will be preserved too.

490. insert spaces in text

```
  x:"wider"
  a:+,x
  a
(,"w"
 ,"i"
 ,"d"
 ,"e"
 ,"r")
  b:a,'" "
  b
("w "
 "i "
 "d "
 "e "
 "r ")
  ,/b
"w i d e r "
  ,/(+,x),'" "
"w i d e r "
```

This is converting the input character vector to a 1-column matrix, appending a column of blank spaces, and finally flattening the resulting matrix. We could also use amend `@[(2*#x)#" ";2*!#x;:;x]` or indexing `(x,a#" ")@<(!a:#x),!#x`, but in this case these techniques result in more complex/longer expressions.

507. insert blank in y after mark in x

```
  x:1 0 0 0 0 1 0 0
  y:"abcdefgh"
  x#\:" "
(," "
 ""
 ""
 ""
 ""
 ," "
 ""
 "")
  y,' x#\:" "
("a "
 ,"b"
 ,"c"
 ,"d"
 ,"e"
 "f "
 ,"g"
 ,"h")
  ,/ y,' x#\:" "
"a bcdef gh"
```

We can create a column of blanks to insert by applying `#` and using each item in `x` to indicate how many spaces to take. This will produce a column of spaces and empty strings that we can append to `y` (using `,'` so that the append operation is applied by column). After appending by column, we need to apply a "flatten" operation `,/` to get the desired result. Alternatively, we could use an amend (see idiom 538) or indexing and grading (see idiom 28) `(y,a#" ")@<(!a:#y),&x`.

508. conditional text

```
  x:0
  ,/((~x)#'"in"),"correct"
"incorrect"
  x:1
,/((~x)#'"in"),"correct"
  "correct"
```

This is presenting one way of producing conditional text using `#` to take 0 or 1 instances of the text to be added (and in this case, prepending and flattening the result).

Other alternatives for achieving the same results are:

```
  ("incorrect";"correct")@x / using indexing, translation-friendly
  (("in";"")@x),"correct" / using indexing and append
  :[x;"";"in"],"correct" / using the conditional operator and append
```

545. zero items of y not in x

```
  y: 2 3 4 5 6 7 8 9 10 11
  x:2 3 5 7 11
  y _lin x
1 1 0 1 0 1 0 0 0 1
  y*y _lin x
2 3 0 5 0 7 0 0 0 11
```

`_lin` will return the 0/1 intersection vector that we can multiply by `y` in order to zero out the items not in the intersection.

578. merge items from x and y alternately

```
  x:1 3 5 7
  y:2 4 6 8
  ,/x,'y
1 2 3 4 5 6 7 8
```

We can merge 2 vectors alternately by appending at 1-level deep and then flattening the resulting matrix. We can also create the matrix by transposing the stacked rows and flatten it `,/+(x;y)` although it results in a longer expression.

581. insert y after each item of x

```
  x:"abc"
  y:"d"
  ,/x,'y
"adbdcd"
```

This is the same expression as idiom 578 but applied to a vector and an atom. The atom will be appended after every item in `x`.

Text and block manipulation

Blocks of text are typically represented by a matrix of characters (square or jagged) where each row represents a line of text. This section presents several idioms targeted at manipulating lists of character vectors.

205. remove trailing blank rows

```
  x:+5 9#"abc de    "
  x
("aaaaa"
 "bbbbb"
 "ccccc"
 "     "
 "ddddd"
 "eeeee"
 "     "
 "     "
 "     ")
  x~\:(#+x)#" "
0 0 0 1 0 0 1 1 1
  ~x~\:(#+x)#" "
1 1 1 0 1 1 0 0 0
  ||\|~x~\:(#+x)#" "
1 1 1 1 1 1 0 0 0
  &||\|~x~\:(#+x)#" "
0 1 2 3 4 5
  rtr:{x[&||\|~x~\:(#+x)#" "]}
  rtr[x]
("aaaaa"
 "bbbbb"
 "ccccc"
 "  "
 "ddddd"
 "eeeee")
```

Given a list of character vectors (rows) of the same length we can get a blank row by `(#+x)#" "` (or, `(#*x)#" "`) then we can apply a technique similar to idiom 73 (remove trailing blanks) combining a comparison (not match each left) and a twice-reversed max scan to obtain a 0/1 vector of the rows we want to keep, and finally indexing where ones are present. We could also have used an equality comparison and used min over each `&/'` to get the per row results, instead of using match `x@&||\|~&/'x=" "`.

206. remove duplicate rows

```
  x:("abc"
  "def"
  "abc"
  "ghi"
  "jkl"
  "abc"
  "ghi"
  "abc")
  ?x
("abc"
 "def"
 "ghi"
 "jkl")
```

When applied to a list of character vectors that represent rows, monadic `?` will return the unique rows (that is the same operation as removing the duplicate rows).

207. indices in matrix x of rows of matrix y

```
  x:+3 8#"abcdefgh"
  x
("aaa"
 "bbb"
 "ccc"
 "ddd"
 "eee"
 "fff"
 "ggg"
 "hhh")
  y:+3 4#"afmc"
  y
("aaa"
 "fff"
 "mmm"
 "ccc")
  x?/:y
0 5 8 2
```

A matrix can be interpreted as a vector of rows, and a find each right will find in `x` each row of `y`.

209. remove trailing blank columns

```
  x:3 9#"abc de    "
  x
("abc de    "
 "abc de    "
 "abc de    ")
  +rtr[+x]
("abc de"
 "abc de"
 "abc de")
```

This idiom is transposing the original matrix and then applying the idiom for removing trailing rows `rtr:{x[&||\|~x~\:(#+x)#" "]}`. We can also use indexing to avoid transposition operations `x[;&||\|~&/x=" "]`.

210. remove leading blank columns

```
  x:3 9#"   ed cba"
  x
("   ed cba"
 "   ed cba"
 "   ed cba")
  +|rtr[|+x]
("ed cba"
 "ed cba"
 "ed cba")
```

We can use transpositions and reversals in combination with the idiom for removing trailing rows `rtr:{x[&||\|~x~\:(#+x)#" "]}` for removing leading blank columns, or we could use `x[;&|\~&/x=" "]`.

211. remove leading blank rows

```
  x:|+3 9#"abc de "
  x
(" "
 " "
 " "
 "eee"
 "ddd"
 " "
 "ccc"
 "bbb"
 "aaa")
  |rtr[|x]
("eee"
 "ddd"
 " "
 "ccc"
 "bbb"
 "aaa")
```

We can use reversals in combination with the idiom for removing trailing rows `rtr:{x[&||\|~x~\:(#+x)#" "]}` for removing leading blank rows, or we could use `x@&|\~&/'x=" "`.

216. rows of matrix x starting with y

```
  x:("sit";"sat";"sin";"tin")
  x
("sit"
 "sat"
 "sin"
 "tin")
```

```
  y:"si"
  y _lin/:x
(1 1
 1 0
 1 1
 0 1)
  &/'y _lin/:x
1 0 1 0
  &&/'y _lin/:x
0 2
  rb:{x[&&/'y _lin/:x]}
  rb[x;y]
("sit"
 "sin")
```

This idiom will actually produce all rows of x that contain y. In the example if we add “osi” to x, it would be output by `rb` even though it doesn’t start with “si”. If we want the rows of matrix x starting with y we can use one of the pattern matching functions in k3.3

```
  x:("sit";"sat";"sin";"tin";"osi")
  y:"si"
  rb:{x[&&/'y _lin/:x]}
  rb[x;y]
("sit"
 "sin"
 "osi")
  rb:{x@&_sm[x;y,"*"]}
  rb[x;y]
("sit"
 "sin")
```

218. single blank row from multiple

```
  s:"  a bc  def    g"
  a:~s=" "
  a
0 0 1 0 1 1 0 0 1 1 1 0 0 0 0 1
  b:~a
  b
1 1 0 1 0 0 1 1 0 0 0 1 1 1 1 0
  c:>':0,b
  c
1 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0
  d:a|c
  d
1 0 1 1 1 1 1 0 1 1 1 1 0 0 0 1
  e:&d
  e
0 2 3 4 5 6 8 9 10 11 15
  s[e]
" a bc def g"
  x:+3 10#"a  b c   d"
  x
```

```
("aaa"
 " "
 " "
 "bbb"
 " "
 "ccc"
 " "
 " "
 " "
 "ddd")
  a:x~\:(#+x)#" "
  a
0 1 1 0 1 0 1 1 1 0
  b:~a
  c:>':0,a
  c
0 1 0 0 1 0 1 0 0 0
  d:b|c
  d
1 1 0 1 1 1 1 0 0 1
  e:&d
  e
0 1 3 4 5 6 9
  x[e]
("aaa"
 "   "
 "bbb"
 "   "
 "ccc"
 "   "
 "ddd")
  rs:{x[{&(~x)|xf[x]}x~\:(#+x)#" "]}
  rs x
("aaa"
 "   "
 "bbb"
 "   "
 "ccc"
 "   "
 "ddd")
```

This idiom first shows a way to remove consecutive spaces in a character vector. This is achieved by first generating a 0/1 vector where ones are marking non-space charactert. The idiom then prepends a 0 to the 0/1 spaces and ranks each pair, this will identify the 0->1 transitions that correspond to the first space in each sequence, and uses a max operation ("bolean or") to combine it with the non-spaces. The original vector at locations where there is a one will have the blanks compressed to only one. The idiom then shows how this technique is extended to removing consecutive rows of blanks in rows of fixed length. The comparison for determining blank rows is the same used in idiom 205 (remove trailing blank rows), and then the operations on indices are the same as used for consecutive spaces in a character vector.

220. remove duplicate blank columns

```
  x:3 9#"a  b c  d"
  x
("a  b c  d"
 "a  b c  d"
 "a  b c  d")
  +rs[+x]
("a b c d"
 "a b c d"
 "a b c d")
```

This idiom is building on idiom 218, remove duplicate blank rows `rs:{x[{&(~x)|xf[x]}x~\:(#+x)#"  "]}`, and adding a transpose operation for removing duplicate blank columns.

225. remove blank rows

```
  x:("aaa"
> "bbb"
> "   "
> "ccc"
> "   ")
  x _dv "   "
("aaa"
 "bbb"
 "ccc")
```

Applying a `_dv` "delete value" operation to a list of character vectors, with a row of spaces as the second argument it will remove all rows matching the row of spaces.

226. remove blank columns

```
  x:+4 4#"xxxx    hhhh  ii"
  x
("x h "
 "x h "
 "x hi"
 "x hi")
  +(+x)_dv " "
("xh "
 "xh "
 "xhi"
 "xhi")
```

We can use transpositions to transform the previous idiom, remove blank rows, to remove blank columns instead.

231. which rows of x contain elements different from y

```
  x:("aaa";"bbb";"ooo";"pop")
  y:"o"
  ~x=y
(1 1 1
 1 1 1
 0 0 0
 1 0 1)
  |/'~x=y
1 1 0 1
```

The inequality comparison `~x=y` when applied to a matrix and a scalar it will be conformed into an element-wise comparison. If we want to reduce it to rows, we can apply a max over operation to each row `|/'`. We can make a small improvement by using De Morgan's law to perform the invert operation at the end, which will reduce the number of invert operations needed `~&/'x=y`.

359. locate blank rows

```
  x:+5 6#"a bc d"
  x
("aaaaa"
 "     "
 "bbbbb"
 "ccccc"
 "     "
 "ddddd")
  x~\:(#+x)#" "
0 1 0 0 1 0
```

This is producing a blank row by counting the columns (as the count of rows in the transposed matrix) and using that to repeat a single blank, then comparing each row to a blank row. Other alternatives would be counting the columns by counting the items in the first row `x~\:(#*x)#" "` or comparing a single blank to the uniques of each row and flattening the result `,/" "=?:'x`.

441. comma separated list from table

```
  x:("Swift";"Austen";"Dickens")
  x
("Swift"
 "Austen"
 "Dickens")
  ",",'x
(",Swift"
 ",Austen"
 ",Dickens")
  ,/",",'x
",Swift,Austen,Dickens"
  1 _ ,/",",'x
"Swift,Austen,Dickens"
```

We can insert comma separators in a list of character vectors by prepending a comma, flattening, and removing the extra "leading" comma. Note that this can be useful for producing csv (comma separated value) text files in simple cases but it does not handle some details of the csv format (e.g., quoted strings with commas inside).

485. append empty row on matrix

```
  x:("ab";"cd";"ef")
  x
```

```
("ab"
 "cd"
 "ef")
  x,,(*|^x)#" "
("ab"
 "cd"
 "ef"
 " ")
```

This idiom is getting the last dimension of the matrix `(*|^x)`, producing a row of blank spaces of the same length, enlisting it and then appending it to the matrix (the "empty row" in the title is referring to a row of blank spaces, not an empty string). Note that for getting the last dimension this is reversing the vector of dimensions and then getting the first element `*|`. This is a technique that is fine for short vectors, but for longer vectors it can be more efficient using `*-1#`. If the matrix is known to be square, we can use the length of the first row for a slightly simpler expression `x,,(#*x)#" "`.

487. insert empty row in x after row y

```
  x:("ab";"cd";"ef")
  x
("ab"
 "cd"
 "ef")
  y:1
  a:x,,(*|^x)#" "
  a
("ab"
 "cd"
 "ef"
 " ")
  b: <(!#x),y
  b
0 1 3 2
  a[b]
("ab"
 "cd"
 " "
 "ef")
  (x,,(*|^x)#" ")[<(!#x),y]
("ab"
 "cd"
 " "
 "ef")
```

This is combining the techniques in idioms 485 append empty row on matrix and 375 insert row `x` in matrix `y` after row `g`, first appending a row of blanks and then using indexing to rearrange the result. Using the length of the first row for the appended line we get `(x,,(#*x)#" ")[<(!#x),y]`.

489. make string y into table guided by marker x

```
  y:"eachwordinarow"
  x:1 0 0 0 1 0 0 0 1 0 1 1 0 0
  a:&x
  a
0 4 8 10 11
  b:a _ y
  b
("each"
 "word"
 "in"
 ,"a"
 "row")
  (&x)_ y
("each"
 "word"
 "in"
 ,"a"
 "row")
```

This is using monadic `&` to convert the 0/1 marker to indices and then using those indices with the cut operator, dyadic `_`.

499. rows of x starting with item of y

```
  x:("abcd";"efgh";"ijkl";"mnop")
  x
("abcd"
 "efgh"
 "ijkl"
 "mnop")
  y:"ai"
  x[;0] _lin y
1 0 1 0
  &x[;0] _lin y
0 2
  x[&x[;0] _lin y]
("abcd"
 "ijkl")```
```

When x is a list of character vectors we can extract the first character from each directly by indexing `x[;0]`, and then by applying idiom 495 we can identify the rows of x starting with any of the characters in y.

576. append y items g before each item of x

```
  x:1 3 5
  y:2
  g:10
  y#g
10 10
  (y#g),/:x
(10 10 1
 10 10 3
```

```
 10 10 5)
  ,/(y#g),/:x
10 10 1 10 10 3 10 10 5
```

We can append `y` items `g` before each item of `x` by generating a vector of `y` items of value `g` `y#g`, appending each item of `x` using `(y#g),/:x` and flattening the result `,/(y#g),/:x`.

577. append y items g after each item of x

```
  x:1 3 5
  y:2
  g:10
  ,/x,\:y#g
1 10 10 3 10 10 5 10 10
```

We can append `y` items `g` before each item of `x` by generating a vector of of value `g` of lenth `y` items `y#g`, prepending each item of `x` using an "each left" `x,\:y#g` and flattening the result `,/x,\:y#g`.

579. variable length lines

```
  x:"by and by"
  y:"God caught his eye"
  ,(x;y)
,("by and by"
 "God caught his eye")
```

We can create a collection of lines by "stacking" them as rows. If we want the text to be a a single item (e.g., to produce a collection of text blocks) we can further enlist the result.

Subvectors

Given a vector, we can derive subvectors (sometimes also called infixes) using several associated representations.

- A length list, providing the length of each subvector.
- An index list or cut list, indicating the indices where each subvector starts (this is the format expected by the "cut" `_` operator). Cut lists are always ascending and start at 0 (if they start with any other number, the result will not contain that count of items from the beginning of the original vector)
- A partition vector is a 0/1 vector of the same length as the original vector, with ones at the indices where subvectors start and zeros elsewhere.
- A subvector index is a vector of the same length as the original vector where each item specifies the subvector that the corresponding item belongs to
- Markers are 0/1 vectors of the same length as the original vector, that alternate between 0 and 1 changing at the indices at which the subvector starts. This is typically used for applying operations only to the subvectors marked by ones

For example, given a vector `0 1 2 3 4 5 6 7 8 9` we can specify subvectors `(0 1 2;3 4;5;6 7 8 9)` in the following ways

- length list: `3 2 1 4`

- cut list: `0 3 5 6`
- partition vector: `1 0 0 1 0 1 1 0 0 0`
- subvector index: `0 0 0 1 1 2 3 3 3 3`
- markers: `0 0 0 1 1 0 1 1 1 1`

Note: if our intended application requires the ability to specify empty subvectors (with 0 items) we can only use a length list or a cut list.

This section explores how to convert between these representations, and applying operations by subvector (instead of applying them to the whole vector)

5b. indices from lengths

```
  il:{(#x)#+\0,x}
  il 4 3
0 4
```

Given a length list, `il` converts these lengths to a cut list. Something to note is that the cut list has "tolerances" regarding the length of the last item, e.g.,

```
  il 4 3
0 4
  il 4 12
0 4
```

meaning, it doesn't matter if the input can produce a last item with length 3 or 12, when expressing subvectors a cut list we'll just get from the 5th item, index 4, until the end.

2. max scan x partition y

```
  x:1 1 0 0 0 1 0 0 1 1
  y:3 4 8 2 5 6 9 4 5 4
  ,/|\'(&x)_ y
3 4 8 8 8 6 9 9 5 4
```

Max scan x partition y takes a partition and an array and generates the running maximum for each subarray. `(&x)_ y` converts a partition specification into a cutlist. `|\'` does a max scan (running max) on each partition, and `,/` flattens the result.

Partitioning can be useful for purposes like dividing by specific calendar dates. E.g., if we want to partition elements in a table `t`, that has a timestamped (`_t` format) column named `ts`, in subtables that go from the 15th of each month to the 14th of the following month we can easily generate the corresponding partition vector with the formula

```
15=(*:'_ltime't`ts)!\:100
```

Combining both, for example we could do

```
,/|\'(&t`price)_ 1=(*:'_ltime't`ts)!\:100
```

to get the monthly running max for the price column in the table.

255. running sum of infixes of y indicated by x

```
  x:1 0 0 0 1 0 0 0 1
  y:1 2 3 4 5 6 7 8 9
  a:&x
  a
0 4 8
  b:a _ y
  b
(1 2 3 4
 5 6 7 8
 ,9)
  c:+\'b
  c
(1 3 6 10
 5 11 18 26
 ,9)
  d:,/c
  d
1 3 6 10 5 11 18 26 9
  rs:{,/+\'(&x)_ y}
  rs[x;y]
1 3 6 10 5 11 18 26 9
```

This is a variation on idiom 2, using sum instead of max.

3. min scan x partitions y

```
  x:1 1 0 0 0 1 0 0 1 1
  y:3 4 8 2 5 6 9 4 5 4
  ,/&\'(&x)_ y
3 4 4 2 2 6 6 4 5 4
```

This is a variation on the previous idiom, changing from max scan to min scan.

5. sort subvectors ascending

```
  sa:{x[<x]}
  il:{(#x)#+\0,x}
  x:10 30 20 50 60 40 5
  y:4 3
  ,/sa'(il y)_ x
10 20 30 50 5 40 60
```

By cutting, sorting each cut, then re-joining the sorted cuts, we can obtain the desired result.

6. subvector minima

```
  x:1 1 0 0 0 1 1 0 0 1
  y:3 4 8 2 5 6 9 4 5 4
  &/'(&x)_ y
3 2 6 4 4
```

This is using a partition specification to generate a cut list and then doing a min-over each cut.

14. subvector maxima

```
  x:1 1 0 0 0 1 1 0 0 1
  y:3 4 8 2 5 6 9 4 5 4
  |/'(&x)_ y
3 8 6 9 4
```

This is a variation on Idiom 6, changing min to max.

7. subvector grade up (see 15 for down)

```
  x:1 0 0 1 0 0 1 0
  y:14 12 18 16 13 15 11 17
  {,/x+'<:'x _ y}[&x;y]
1 0 2 4 3 5 6 7
```

Instead of using an anonymous function we could keep the cut list in a temporary variable

```
  x:1 0 0 1 0 0 1 0
  y:14 12 18 16 13 15 11 17
  ,/c+'<:'(c:&x)_ y
1 0 2 4 3 5 6 7
```

What this idiom is doing is: converting the partition specification into a cut list, grading each cut, adjusting the grading by their respective offset (which is given by the cut list) and finally flattening the results.

15. subvector grade down (see 7 for up)

```
  x:1 0 0 1 0 0 1 0
  y:14 12 18 15 13 16 11 17
  {,/x+'>:'x _ y}[&x;y]
2 0 1 5 3 4 7 6
```

This is a variation on idiom 7, changing the direction of grading.

21. rotate infixes of y determined by Boolean x to the left one place

```
  y:"abcdefghij"
  x:1 0 1 0 0 1 1 0 0 0
  y[<x++\x]
"badecfhijg"
```

This idiom starts from a partition vector, converting it to a subvector index list using `+\x`, then adding the partition vector again `x++\x` to increase the index for the first item in each subvector, which will cause the item to be moved to the end of the sub-vector while preserving the rest of the sub-vector, effectively rotating the subvectors when applying the index. This idiom depends on how `<` handles the sorting of identical items, where the earliest duplicate gets assigned a lower index (in other words, `<` is a stable sort).

39. reverse infixes in x of lengths y

```
  il:{(#x)#+\0,x}
  x:11+!8
  x
11 12 13 14 15 16 17 18
  y:3 3 2
  x[|>+\(!#x)_lin il y]
13 12 11 16 15 14 18 17
```

`Il` is idiom 5b “indices from length”.

`(!#x)_lin il y]` generates a partition vector from the lengths; another way to generate this partition vector would be `@[&#x;il y;:;1]`.

`+\(!#x)_lin il y]` produces a subvector index vector, (note that in this case it is 1-based and not 0-based, e.g. `1 1 1 2 2 2 3 3`).

The reversed grade down will produce the indices for the reversed sub-vectors, as if we had applied `|` to each one.

40. reverse infixes in x starting at indices y

```
  x:1 0 1 0 0 1 0 0 0 1
  y:1 2 3 4 5 6 7 8 9 10
  +\x
1 1 2 2 2 3 3 3 3 4
  >+\x
9 5 6 7 8 2 3 4 0 1
  |>+\x
1 0 4 3 2 8 7 6 5 9
  y[|>+\x]
2 1 5 4 3 9 8 7 6 10
```

This is a simpler version of idiom 39. If we already have a partition vector it takes fewer steps to get to the result.

202. indices of infixes of length y

```
  x:4+!5
  x
4 5 6 7 8
  y:3
  x+\:!y
(4 5 6
 5 6 7
 6 7 8
 7 8 9
 8 9 10)
```

If `x` contains a list of indices and we want a list of vectors of length `y` starting at each `x`, we can add a `!y` to each item in `x`.

213. maxima of infixes of x specified by Boolean list y

```
  x:-17 7 30 12 5 2 -5 6 -3 -19
  y:10#1 1 0
  y
1 1 0 1 1 0 1 1 0 1
  |/x[&y]
12
```

`|/` produces the max value of a list, which we filter by indexing on `&` of a 0/1 vector.

254. running parity of infixes of y indicated by x

```
  x:1 0 0 0 0 1 0 0 0 0 1 0 0 0
  y:1 0 0 1 1 1 0 0 1 0 1 1 0 0
  a:&x
  a
0 5 10
  b:a _ y
  b
(1 0 0 1 1
 1 0 0 1 0
 1 1 0 0)
  c:(+\'b)!\:2
  c
(1 1 1 0 1
 1 1 1 0 0
 1 0 0 0)
  d:,/c
  d
1 1 1 0 1 1 1 1 0 0 1 0 0 0
```

This is partitioning `y` as indicated by `x`, similar to idiom 2. The running parity will be the parity of the running sum for each subvector, modulo 2 (idiom 309, modified with a `'` in to apply it to each subvector). Note that the `\:` is not necessary, `c:(+\'b)!2` produces the same result. Finally, the result needs to be flattened with `,/`.

256. groups of 1s in y pointed at by x

```
  y:0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1
  x:0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1
  a:+\>':0,y
  a
0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2
  y&a=|\x*a
0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1
```

This idiom is first doing a pairwise comparison `>':` of `0,y` to generate a partition vector pointing to the starting points of groups of ones in `y`. After that it applies a max scan to convert the partition vector to a subvector index `a`. Multiplying `a` by `x` zeroes the subvector index at locations not pointed at by `x`. Applying a max scan converts the indices to a running max. Finally, comparing this running max to the subvector index produces zeroes at the locations of groups of ones not pointed at, and we can use this result to zero the not pointed at positions in `y`. An alternative way to do the same would be using a slightly modified min scan x partitions y (idiom 3) where we prepend a 0 to the cutpoints so that the entire `y` is processed (even if `x` starts with zeroes)

```
  x:0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1
  y:0 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1
  y&a=|\x*a:+\>':0,y
0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1
  ,/&\'(0,&x)_ y
0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1
```

257. sums of infixes of x determined by lengths y

```
  x:1+!10
  x
1 2 3 4 5 6 7 8 9 10
  y:2 3 2 3
  a:+\0,-1 _ y
  a
0 2 5 7
  a _ x
(1 2
 3 4 5
 6 7
 8 9 10)
  +/'a _ x
3 12 13 27
```

This idiom is converting the length vector to a cutpoint vector by removing the last (as cutpoints will cut to the end), prepending a zero (so that we process the beginning of `x`) and applying a running sum. Finally, applying a sum over each cut will produce the desired output.

265. insert x[i] zeroes after i-th infix of y

```
  y:0 0 1 0 1 0 1 1
  x:1 2 2 1
  ((0,-1_ 1+&y)_ y)
(0 0 1
 0 1
 0 1
 ,1)
  x#\:0
(,0
 0 0
 0 0
 ,0)
  ((0,-1_ 1+&y)_ y),'x#\:0
(0 0 1 0
 0 1 0 0
 0 1 0 0
 1 0)
  ,/((0,-1_ 1+&y)_ y),'x#\:0
0 0 1 0 0 1 0 0 0 1 0 0 1 0
```

This us using a technique analogous to the previous idiom. `((0,-1_ 1+&y)_ y)` cuts at the infixes, `x#\:0` generates the zeros to insert, and finally the results need to be flattened and concatenated, `,/((0,-1_ 1+&y)_ y),'x#\:0` .

277. end indicators from lengths

```
  x:1 2 3 4 5
  +\x
1 3 6 10 15
  -1++\x
0 2 5 9 14
  +/x
```

```
15
  @[&+/x;-1++\x;:;1]
1 0 1 0 0 1 0 0 0 1 0 0 0 0 1
```

To convert a list of lengths to infixes we can get the 1-based indices of the ones by calculating the running sum. We then add `-1` to converted it to a 0-based index, `-1++\x`. We can then generate a vector of zeros by calculating the total length of the expected result `+/x` and applying `&`. Finally, we can use an amend operation to modify this vector of zeros and set to 1 the items at the indices previously calculated.

278. start indicators from lengths

```
  x:1 2 3 4 5
  (!+/x) _lin\: +\0,x
1 1 0 1 0 0 1 0 0 0 1 0 0 0 0
```

`(!+/x)` produces all the indices in the target result. `+\0x` produces the indices of starting points (with an additional index past the end of the target result). By checking each index in the target whether they are a starting point we can generate the desired result. Note that `_lin\:` will work but the `\:` is unnecessary, we can just use `_lin` (or `_in\:`).

289. or-scan of infixes of y indicated by x

```
  y:1 0 0 1 0 1 0 0
  x:1 0 1 0 0 0 1 0
  a:&x
  b:a _ y
  b
(1 0
 0 1 0 1
 0 0)
  c:|\'b
  c
(1 1
 0 1 1 1
 0 0)
  ,/c
1 1 0 1 1 1 0 0
```

This is using the infix list to produce a cut list, applying it to `y`, performing an or-scan (to set to 1 all items after the first 1) and then concatenating the results.

290. and-scan of infixes of y indicated by x

```
  x:1 0 1 0 0 0 1 0
  y:1 0 0 1 0 1 0 0
  a:&x
  b:a _ y
  b
(1 0
 0 1 0 1
 0 0)
  c:&\'b
  c
```

```
(1 0
 0 0 0 0
 0 0)
  ,/c
1 0 0 0 0 0 0 0
```
This is a variation on the previous idiom, using an and-scan (set to 0 all items after the first 0).

291. sums of infixes of y indicated by x

```
  y:1 2 3 4 5
  x:1 0 1 0 1
  a:&x
  b:a _ y
  b
(1 2
 3 4
 ,5)
  c:+/'b
  c
3 7 5
```
This is another variation on the previous idioms, using a sum-over.

292. groups of 1s in y pointed to by x

```
  y:1 1 1 0 0 1 1
  x:0 1 0 1 0 0 0
  -1 _ 0,y
0 1 1 1 0 0 1
  +\y>-1 _ 0,y
1 1 1 1 1 2 2
  x&y
0 1 0 0 0 0 0
  a:+\y>-1 _ 0,y
  a
1 1 1 1 1 2 2
  a[&x&y]
,1
  a _lin ,1
1 1 1 1 1 0 0
  y&a _lin ,1
1 1 1 0 0 0 0
  go:{y&a _lin(a:+\y>-1 _ 0,y)[&x&y]}
  go[x;y]
1 1 1 0 0 0 0
```
This is first generating the indices `a` of 1..0 groups by comparing `y` to a right-shifted version of itself `-1_ 0,y` and applying a running sum, then using `&x&y` (where `x` and `y`) to identify the indices of ones pointed at, so that `a[&x&y]` produces the list of group indices pointed at. `a _lin  a[&x&y]` is a 0/1 vector indicating if an item in y is in a group pointed at, which we can filter with a `y&` (min of y or the 0/1 vector) to zero the entries in the input `y` that are not pointed at by `x`.

296. starting positions of infixes from lengths

```
  x:2 3 1 5
  +\-1 _ 0,x
0 2 5 6
  sl:{+\-1 _ 0,x}
  sl[x]
0 2 5 6
```

We can convert lengths to infixes by shifting the lengths one position (prepend a zero and cutting the last) then applying a running sum. When cutting the last length we do “lose” information but this is implicit in the problem statement, the starting positions of infixes by itself it cannot encode information about the length of the last infix, for complete information about the infixes it is necessary to combine it with additional info such as the total length.

300. g-th infix of y marked by x

```
  x:1 0 0 1 0 1 0 0 0 1 0
  y:"abcdefghijk"
  g:2
  a:&x
  a
0 3 5 9
  b:a _ x
  b
("abc"
 "de"
 "fghi"
 "jk")
  b[g]
"fghi"
  ((&x)_ y)[g]
"fghi"
```

This idiom is cutting at the indices for the infixes and then indexing into the gth item.

304. invert 0s following 1st 1

```
  x:0 0 1 0 0 1 1
  |\x
0 0 1 1 1 1 1
```

If we apply a running max to a 0/1 vector, all the values after the first one will be set to 1.

404. end points for x fields of length y

```
  x:5
  y:3
  @[&x*y;(y-1)+y*!x;:;1]
0 0 1 0 0 1 0 0 1 0 0 1 0 0 1
```

In this idiom we first calculate the enumeration of fields `!x`, scale it with `y*` and shift it with `(y-1)+` to obtain the indices for the end points. The length of the required result is `x*y` and we can use a `&` to generate a vector of zeros of the desired length. Finally, we can use an amend operation to change the zeros to ones at the indices calculated earlier.

405. start points for x fields of length y

```
  x:5
  y:3
  @[&x*y;y*!x;:;1]
1 0 0 1 0 0 1 0 0 1 0 0 1 0 0
```

This is a variation on the previous idiom. If we want the starting points, we don't need to shift the indices.

414. ending indices from field lengths

```
  x:4 7 13 15 20
  (1+*x),-':x
5 3 6 2 5
```

What this idiom is actually doing is: given a list of ending indices, obtain the length of each field.

This is done by first applying a subtraction operation modified by an "each prior" adverb. The result needs to be adjusted by prepending the calculation for the first field. We could also have prepended a -1 to the vector of indices `-':-1,x`, to represent that the field "prior" to the first one "ended" at position -1.

If we want to get the ending indices for the field lengths, we can do it by adding -1 to the running sum `-1++\x`, see idiom 277 for an example.

415. lengths of infixes of 1 in x

```
  x:0 0 1 1 1 0 0 1 1 1 1 0 1
  a:<':0,x,0
  b:>':0,x,0
  a
0 0 0 0 0 1 0 0 0 0 0 1 0 1
  b
0 0 1 0 0 0 0 1 0 0 0 0 1 0
  &a
5 11 13
  &b
2 7 12
  (&a)-(&b)
3 4 1
  (&<':0,x,0)-&>':0,x,0
3 4 1
  m415:{(&<':0,x,0)-&>':0,x,0}
```

In this idiom we are using a "widened" vector (with prepended and appended zeros) and identifying the 1→0 transitions `a` and the 0→1 transitions `b`. An element-wise subtraction of the indices where the transitions occur will return the desired result. Instead of using both transitions we can also use a running sum operation indexed by location of transitions, and do an each-prior subtraction `-':0,(0+\x)@&<':0,x,0`.

417. end points of equal infixes

```
  x:"baackkkegtt"
  (~(1 _ x)=-1 _ x),1
1 0 1 1 0 0 1 1 1 0 1
```

This is doing an element-wise equality comparison of the vectors resulting from cutting the first and the last item (which results in a comparison with the prior), negating the result and adjusting it by appending a one (as the last item is always an endpoint). We can also use an each prior inequality comparison directly `(~=':x),1`.

418. starting indices of equal item infix

```
  x:"baackkkegtt"
  1,~(1 _ x)=-1 _ x
  1 1 0 1 1 0 0 1 1 1 0
```

This is doing an element-wise equality comparison of the vectors resulting from cutting the first and the last item (which results in a comparison with the prior), negating the result and adjusting it by prepending a one (the first item is always a starting point). We can also use an each prior inequality comparison directly `1,(~=':x)`. Note that the result is a 0/1 vector with ones at the starting points. If we want the indices, we will need to apply a `&` operator, `&1,(~=':x)`.

423. lengths from start indicator

```
  x:1 0 1 0 0 1 0 0 0 1 0
  &x,1
0 2 5 9 11
  -':&x,1
2 3 4 2
```

This is doing an each prior subtraction of the location of the start indicators. Note that a 1 needs to be appended to account for the boundary condition in the last element.

426. change all multiple infixes of y in x to single

```
  x:"bccbceekl"
  y:"c"
  x[&a|-1 _ 1,a:~x=y]
"bcbceekl"
```

This is a generalized form of idiom 424, e.g., if we have other separators that need to be converted from multiple into a single one.

491. or-reduce infixes of y marked by x

```
  x:1 0 0 1 0 0 0 1 0 0 0 0
  y:0 0 0 0 1 0 0 0 0 0 1 0
  a:(&x)_ y
  a
(0 0 0
 0 1 0 0
 0 0 0 1 0)
  |/'a
0 1 1
```

This is converting the vector of infixes to a cutlist and using it to cut the original input, then applying an or-reduce (`|/`, or-over) to each of the resulting vectors.

492. and-reduce infixes of y marked by x

```
  x:1 0 0 1 0 0 0 1 0 0 0 0
  y:0 1 1 1 1 1 1 1 1 1 1 0
  (&x)_ y
(0 1 1
 1 1 1 1
 1 1 1 1 0)
  &/'(&x)_ y
0 1 0
```

This is converting the vector of infixes to a cutlist and using it to cut the original input, then applying an and-reduce (`&/`, and-over) to each of the resulting vectors.

529. markers for x at y

```
  x:"abcdefghijklmn"
  y:3 7 9
  @[&#x;y;:;1]
  0 0 0 1 0 0 0 1 0 1 0 0 0 0
```

We can convert indices to a 0/1 vector of the same length as another vector by first generating a vector of zeros of the same length as the original, `&#x`` and then amending the items at the desired indices, assigning a 1 to them.

539. Boolean vector of length y with zeros in locations x

~~~
```   x:2 3 4 8
  y:10
  /method A
  ~(!y) _lin x
1 1 0 0 0 1 1 1 0 1
  /method B
  ~@[&y;x;:;1]
1 1 0 0 0 1 1 1 0 1
~~~

This idiom is presenting two methods for obtaining a 0/1 vector based on specified indices for zeros. The first method uses the 0/1 intersection of `x` and the indices for all items in a vector of length `y`, negated so that the intersection results in zeros. The second method uses a direct application of amend to a vector of zeros of the desired length, also negated. We could also move the negation into the item being amended and assign zeros `@[~&y;x;:;0]` which has the same complexity but may be a more direct translation of the idiom's title.

540. markers in Boolean vector of length x at indices y

```
  x:10
  y:1 3 7
  / method A
  (!x) _lin y
0 1 0 1 0 0 0 1 0 0
  / method B
```

```
  @[&x;y;:;1]
0 1 0 1 0 0 0 1 0 0
```

This is the complementary operation for the previous idiom, for getting a vector of zeros with ones at the markers we just need to remove the negation step.

Matrices and Tensors

K allows nesting lists and vectors into other lists and vectors. This enables manipulation of matrices (2D) and tensors (3D and higher dimensions). The majority of this manipulation is done with standard "vector" functions, applying them at the desired depth as already covered. A notable exception is `_mul` which explicitly provides matrix multiplication. This section covers various idioms related to manipulation of matrices and tensors.

547. is x vector

```
  iv:{1=#^x}
  iv[0]
0
  iv[1 2]
1
  iv[(1;2)]
1
  iv[(1 2;3 4)]
0
  iv[2 3#!6]
0
  iv[!0]
1
```

Determining if a certain input is a vector can be done by counting the dimensions of the input. K provides a verb for determining if something is an atom, monadic `@`, but it doesn't distinguish vectors from matrices and tensors, while this idiom does. We can apply this same technique of counting dimensions to determine if something is a matrix (or a tensor).

601. number of rows in matrix x

```
  x:17 2#0
  #x
17
```

This is an example application of idiom 411

600. number of columns in matrix x

```
  x:3 19#0
  ^x
3 19
  (^x)[1]
19
```

This is another example of idiom 445. For a matrix (2-dimensional) the number of columns is the size of the 2nd dimension. We can also compute it with a simple expression as the length of the first row, `#*x`.

410. number of columns in matrix x

```
  x:2 7#" "
  ^x
2 7
  *|^x
7
```

This is applying a shape operation and returning the size of the last dimension (columns, if the input is a matrix).

599. number of columns in array x

```
  x:1 1 1 1 1 678#0
  ^x
1 1 1 1 1 678
  *|^x
678
```

This idiom is interpreting the size of the last (first of the reverse) dimension of tensor `x` as the number of columns.

203. one-column matrix from numeric list

```
  x:7 _draw 100
  x
34 31 51 29 35 17 89
  +,x
(,34
 ,31
 ,51
 ,29
 ,35
 ,17
 ,89)
```

A one-column matrix can be generated from a list by enlisting the original and transposing it.

588. 2-row matrix from two vectors

```
  x:"abcd"
  y:"efgh"
  (,x),(,y)
("abcd"
 "efgh")
```

If we want to stack 2 vectors as rows using an append operation, we need to enlist the vectors to ensure each will be interpreted as an complete item (instead of a simple concatenation. A simpler expression is to create a matrix with each vector as a row `(x;y)`.

50. connectivity list from connectivity matrix

```
  m:(1 0 1;1 0 1)
  m
(1 0 1
 1 0 1)
```

```
  ,/m
1 0 1 1 0 1
  &,/m
0 2 3 5
  (^m)_vs &,/m
(0 0 1 1
 0 2 0 2)
  lm:{(^x)_vs &,/x}
  lm m
(0 0 1 1
 0 2 0 2)
```

We can convert a 0/1 matrix to the `x,y` coordinates of the ones by first getting the indices of the ones in the linearized matrix and then applying a base conversion using the shape of the original matrix as the bases. This idiom corresponds to steps d and e in idiom 48.

71. connectivity matrix from connectivity list

```
  y
(0 1
 0 2
 1 0
 1 2
 2 2)
  x
3
  x _sv/:y
1 2 3 5 8
  (!9)_lin x _sv/:y
0 1 1 1 0 1 0 0 1
  (x,x)#(!x*x)_lin x _sv/:y
(0 1 1
 1 0 1
 0 0 1)
```

These are the reverse operations from idiom 50.

148. node matrix from connection matrix (inverse to 157)

```
  x:( 1 1 0 0 0
>  0 -1 0 1 1
> -1 0 1 -1 0
>  0 0 -1 0 -1)
/ Each column in matrix x represents a path between 2 nodes:
/ From node 0 to node 2
/ From node 0 to node 1
/ From node 2 to node 3
/ From node 1 to node 2
/ From node 1 to node 3
  a:1 -1=\:x
  a
((1 1 0 0 0
  0 0 0 1 1
  0 0 1 0 0
  0 0 0 0 0)
 (0 0 0 0 0
  0 1 0 0 0
  1 0 0 1 0
  0 0 1 0 1))
  b:_mul[a;!#x]
  b
(0 0 2 1 1
 2 1 3 2 3)
  nc:{_mul[1 -1=\:x;!#x]}
  nc x
(0 0 2 1 1
 2 1 3 2 3)
```

This idiom first uses comparisons to 1 and -1 to split the connection matrix into "from" and "to" nodes, then using matrix multiplication by an enumeration vector to convert position to index.

This operation `_mul[a;!#x]` is equivalent to a "columnar where" when the columns contain one and only one `1`. We could also use transpose and where `&` operations instead.

```
  *:'+:'&:''1 -1=\:+x
(0 0 2 1 1
 2 1 3 2 3)
```

157. connection matrix from node matrix (inverse to 148)

```
/ node matrix top and bottom rows give from and to nodes
  x: (0 0 2 1 1
  2 1 3 2 3)
/ enumerate count of range
  !#?,/x
0 1 2 3
/ where is x equal to each of it
  x=/:!#?,/x
((1 1 0 0 0
  0 0 0 0 0)
 (0 0 0 1 1
  0 1 0 0 0)
 (0 0 1 0 0
  1 0 0 1 0)
 (0 0 0 0 0
  0 0 1 0 1))
/ subtract "to" matrix from "from" matrix
  -/'x=/:!#?,/x
(1 1 0 0 0
 0 -1 0 1 1
 -1 0 1 -1 0
 0 0 -1 0 -1)
```

(no commentary required)

51. indices

```
  ix:{(^x)_vs!*/^x}
  ix[!6]
,0 1 2 3 4 5
  ix[2 3#!6]
(0 0 0 1 1 1
 0 1 2 0 1 2)
  ix[2 3 2#!12]
(0 0 0 0 0 0 1 1 1 1 1 1
 0 0 1 1 2 2 0 0 1 1 2 2
 0 1 0 1 0 1 0 1 0 1 0 1)
```

This idiom shows how we can generate all the indices for a multi-dimensional tensor or matrix (and it also works for vectors, although there are simpler ways for that case). It first generates a linearized index by multiplying over the dimensions of the matrix, then applies a base conversion using the shape of the matrix. The result is given by dimension, if we want tuples of indices, we will need to transpose it.

81. raveled index from general index

```
  x:2 3 4# _ci 97+!24
  x[1;1;3]
"t"
  ^x
2 3 4
  2 3 4 _sv 1 1 3
19
  ,//x
"abcdefghijklmnopqrstuvwx"
  (,//x)[19]
"t"
```

Idioms 50 and 51 present how to use base conversion functions for converting a “linear” index for a vector item into the corresponding index for a matrix obtained from reshaping the vector. This idiom presents the reverse operation where we obtain the “linear” (raveled) index.

58. pair each element of x with each element of y

```
  x:3
  y:4
  (x,y)_vs !x*y
(0 0 0 0 1 1 1 1 2 2 2 2
 0 1 2 3 0 1 2 3 0 1 2 3)
```

This idiom is a different way to interpret the results in idiom 51 for the case of a matrix, generating all indices for a matrix is the same as generating all the pairs for the enumeration of the dimensions of the matrix.

55. indices in x containing elements in y

```
  x:(3 7 8;2 5 9)
  y:(3 1 6;8 9 2)
  (^x)_vs &(,/x) _lin\: ,/y
(0 0 1 1
 0 2 0 2)
```

We first generate the 0/1 matrix of items meeting a condition, then combine the calculation with idiom 50 to obtain the corresponding indices.

100. indexing arbitrary rank array

```
  x:2 3 4 5#!120
  x[1]
((60 61 62 63 64
 65 66 67 68 69
 70 71 72 73 74
 75 76 77 78 79)
(80 81 82 83 84
 85 86 87 88 89
 90 91 92 93 94
 95 96 97 98 99)
(100 101 102 103 104
 105 106 107 108 109
 110 111 112 113 114
 115 116 117 118 119))
  x[0;0]
(0 1 2 3 4
 5 6 7 8 9
 10 11 12 13 14
 15 16 17 18 19)
  x[0;0;0]
0 1 2 3 4
  x[0;0;0;0]
0
```

We can use any number of dimensions we want for indexing into a tensor, as long as the dimension and item exist (otherwise we will get a rank error or an index error, respectively)

161. is y upper triangular

```
  x:(1 0 0 1
> 0 2 1 0
> 0 0 1 2
> 0 0 0 0)
  slt:{(!x)>\:!x}
  slt[#x]
(0 0 0 0
 1 0 0 0
 1 1 0 0
 1 1 1 0)
  x*slt[#x]
(0 0 0 0
 0 0 0 0
 0 0 0 0
 0 0 0 0)
  zm:{(x,x)#0}
  zm[#x]
(0 0 0 0
 0 0 0 0
 0  0  0  0
 0  0  0  0)
  iut:{zm[#x]~x*slt[#x]}
  iut  x
1
  +x
(1  0  0  0
 0  2  0  0
 0  1  1  0
 1  0  2  0)
  iut[+x]
0
```

This idiom calculates a 0/1 matrix `slt` with zeros in the upper triangular positions and ones elsewhere. For a matrix to be upper triangular, element-wise multiplication by `slt` must result in an all-zero matrix of the same dimension as the original. A possible alternative could have been constructed by finding the first nonzero value in each row, `&:'((x)>0)` and comparing to `!#x` but that fails in the case of rows that are all-zero, and adding the handling of this special case would result in a more complex, less efficient solution.

162. is x lower triangular

```
  x:(1  0  0  0
>0  2  0  0
>0  1  1  0
>1  0  2  0)
  sut:{(!x)<\:!x}
  sut[#x]
(0  1  1  1
 0  0  1  1
 0  0  0  1
 0  0  0  0)
  ilt:{zm[#x]~x*sut[#x]}
  ilt[x]
1
  ilt[+x]
0
```

This idiom uses the same principles as idiom 161, but `sut`, the matrix to multiply by, has zeros in the lower triangular positions and ones elsewhere.

525. main diagonal

```
  x:(1  2  3  4
 5  6  7  8
 9  10  11  12)
  y:2#'!#x
  y
(0  0
 1  1
 2  2)
  x  ./:  y
1  6  11
```

We can generate index pairs by taking 2 each of the indices of the rows, then we can apply each pair of indices "at depth" to index the original matrix.

429. matrix with diagonal x

```
  x:5 9 6 7 2
  (2##x)#,/x,'(2##x)#0
(5 0 0 0 0
 0 9 0 0 0
 0 0 6 0 0
 0 0 0 7 0
 0 0 0 0 2)
```

In this idiom, the desired shape of the output is `(2##x)`, e.g., if x has length 5 the result should be 5x5. In this idiom we first produce a vector that is very close (with some extra zeros at the end) to the unrolled version of the desired result. This is done by generating a matrix of zeros of the desired size and prepending `x` as a column (using an append operation modified with `'` so that the operation happens at a deeper level), then unrolling the result with `,/`. Finally, by using a reshape operation again, we can rebuild the unrolled version into a n×n matrix and discard the extra zeros in one operation.

197. identity matrix of order x

```
  id: {(!x)=\:!x}
  id 5
(1 0 0 0 0
 0 1 0 0 0
 0 0 1 0 0
 0 0 0 1 0
 0 0 0 0 1)
```

Enumerating the order and comparing the enumeration vector to each of its items will produce a triangular matrix of the given order. For the identity matrix we need to apply an equality comparison.

163. polynomial product

```
  x:1 2 1
  y:1 3 3 1
  y*/:x
(1 3 3 1
 2 6 6 2
 1 3 3 1)
  1 _'zm[#x]
(0 0
 0 0
 0 0)
  (1 _'zm[#x]),'y*/:x
(0 0 1 3 3 1
 0 0 2 6 6 2
 0 0 1 3 3 1)
  (!#x)!'(1 _'zm[#x]),'y*/:x
(0 0 1 3 3 1
 0 2 6 6 2 0
 1 3 3 1 0 0)
  +/(!#x)!'(1 _'zm[#x]),'y*/:x
1 5 10 10 5 1
  pm:{+/(!#x)!'(1 _'zm[#x]),'y*/:x}
  pm[x;y]
1 5 10 10 5 1```
```

This is a "traditional" or "schoolbook" multiplication algorithm, we multiply each item in one multiplicand by all the items in the other multiplicand, then add up the results shifted by the corresponding number of positions. For the "shift" we are prepending a matrix of zeroes (calculated with the previously seen `zm:{(x,x)#0}`) shortened by one column (because the lowest order item doesn't require a shift) and then using rotation `!`. If we add a step to process the "carry" we could use this same technique for arbitrary length "big int" multiplication where each coefficient is one "big int" digit. Adding this "carry" step is left as an exercise for the reader.

195. upper triangular matrix of order x

```
  x:5
  {~x>\:x}!x
(1 1 1 1 1
 0 1 1 1 1
 0 0 1 1 1
 0 0 0 1 1
 0 0 0 0 1)
  ut:{{~x>\:x}[!x]}
  ut 5
(1 1 1 1 1
 0 1 1 1 1
 0 0 1 1 1
 0 0 0 1 1
 0 0 0 0 1)
```

Enumerating the order and comparing the enumeration vector to each of its items will produce a triangular matrix of the given order. For the diagonal to be ones, we need a "less than or equal" (not greater) operation.

196. lower triangular matrix of order x

```
  lt:{{~x<\:x}[!x]}
  lt 5
(1 0 0 0 0
 1 1 0 0 0
 1 1 1 0 0
 1 1 1 1 0
 1 1 1 1 1)
```

Enumerating the order and comparing the enumeration vector to each of its items will produce a triangular matrix of the given order. For the diagonal to be ones we need a "greater than or equal" (not smaller) operation.

187. direct matrix product

```
  x:1+3 2#!6
```

```
  y:1+2 4#!8
  +:'x*\:\:y
(((1 2 3 4
   2 4 6 8)
  (5 6 7 8
   10 12 14 16))
 ((3 6 9 12
   4 8 12 16)
  (15 18 21 24
   20 24 28 32))
 ((5 10 15 20
   6 12 18 24)
  (25 30 35 40
   30 36 42 48)))
  dp:{+:'x*\:\:y}
```

This is multiplying each element in the first matrix by the whole second matrix (note that it is specific for rank 2, applying it to higher rank tensors would require additional `\:`).

188. Shur product

```
  x:1+3 2#!6
  y:1+2 4#!8
  x
(1 2
 3 4
 5 6)
  y
(1 2 3 4
 5 6 7 8)
((*|^x)#x)*(*^y)#'y
(1 4
 15 24)
```

The Schur (or Hadamard) product is the element-wise product of two matrices of the same dimensions, `x*y`. This idiom is converting its inputs to square matrices (using the number of columns for the first operand and the number of rows for the second operand) and doing an element-wise multiplication.

191. Shur sum

```
  x:1+3 2#!6
  y:1+2 4#!8
  x
(1 2
 3 4
 5 6)
  y
(1 2 3 4
 5 6 7 8)
((*|^x)#x)+(*^y)#'y
(2 4
 8 10)
```

The Schur sum is the element-wise sum of two matrices of the same dimensions, `x+y`. This idiom is converting its inputs to square matrices (using the number of columns for the first operand and the number of rows for the second operand) and doing an element-wise sum.

198. Hilbert matrix of order x

```
  hm:{%1+(!x)+/:!x}
  hm 5
(1 0.5 0.333 0.25 0.2
 0.5 0.333 0.25 0.2 0.167
 0.333 0.25 0.2 0.167 0.143
 0.25 0.2 0.167 0.143 0.125
 0.2 0.167 0.143 0.125 0.111)
```

The Hilbert matrix of order `x` has as rows the successive fractions of 1 starting at the fraction with denominator equal to the row index (1-based). We generate a vector of column indices `!x`, then rows by adding each of the row indices (also `!x`, since it is a square matrix), get the matrix of denominators by adding `1` (to convert indices from 0-based to 1-based) and finally do a element-wise invert (monadic `%`).

200. replicating a dimension of rank-3 array x y-fold

```
  x:2 3 3#1+!18
  y:3
  x[;,/(y#1)*\:!(^x)[1];]
((1 2 3
  4 5 6
  7 8 9
  1 2 3
  4 5 6
  7 8 9
  1 2 3
  4 5 6
  7 8 9)
 (10 11 12
  13 14 15
  16 17 18
  10 11 12
  13 14 15
  16 17 18
  10 11 12
  13 14 15
  16 17 18))
```

`!(^x)[1]` produces the indices of the dimension to replicate (the 2nd dimension in this example), `(y#1)*\:!(^x)[1]` replicates the indices `y` times (we could also have used reshape instead of multiplying by vectors of ones, `(y,(^x)1)#!(^x)1`). Finally, we index the corresponding dimension by the flattened replicated indices.

230. extend a transitive binary relation

```
  x:(0 1 0 0
> 0 0 1 1
> 1 0 0 0
> 0 0 1 0)
  x(|/&)\:x
(0 0 1 1
 1 0 1 0
 0 1 0 0
 1 0 0 0)
```

Given a binary relation matrix (in the example, row 0 is "connected" to 1, row 1 is connected to 2 and 3, row 2 is connected to 0 and row 3 is connected to 2) this is executing a transition where each original row is replaced by applying a boolean "or" on the connected rows. The operations used to perform this transition are `&` of each column in the matrix with the matrix itself and then `|/` the results.

240. matrix product

```
  x:(1 2 3
> 4 5 6)
  y:(1 2
> 3 4
> 5 6)
  x _mul y
(22 28
 49 64)
  x(+/*)\:y
(22 28
 49 64)
```

K has a built-in operator `_mul` for matrix multiplication (inner product, not element-wise). If we wanted to compose it instead, we can use a sum over the element-wise vector multiplication and then use an each-left to apply it to each row of `x` with `y`. The parentheses around the `+/*` operation are required because otherwise the `*` would be applied to the right operand as monadic (first) instead of applied to both operands as dyadic (multiply) as intended

```
   +/* (1 2;3 4;5 6)
3
  (1 2 3) +/ 3
4 5 6
  (1 2 3) +/* (1 2;3 4;5 6)
4 5 6
  +/*[(1 2 3);(1 2;3 4;5 6)]
22 28
```

244. product over subsets of x specified by y

```
  x:1+3 4#!12
  x
(1 2 3 4
 5 6 7 8
 9 10 11 12)
  y:4 3#1 0
  y
(1 0 1
 0 1 0
 1 0 1
 0 1 0
  x (*/^)\:y
(3 8 3.0
 35 48 35.0
 99 120 99.0)
```

This is a variation on the “composed” version of matrix multiplication in idiom 240, where instead of summing over each, we multiply over the power. When `y` is a 0/1 matrix this will produce the multiplication of the items at the indices where there is a one (note that for all-zero subset columns we will get a `1`)

313. value of two-by-two determinant

```
  det: {-/y*|x}
  x:(13 21;34 55)
  x
(13 21
 34 55)
  (13 * 55) - (34 * 21)
1
  13 21 * 55 34
715 714
  -/13 21 * 55 34
1
  det x
,1
```

For the value of the 2x2 determinant we can use element-wise multiplication of the first row by the reversed second row and then subtract the results.

375. insert row x in matrix y after row g

```
  y:4 3#1+!12
  y
(1 2 3
 4 5 6
 7 8 9
 10 11 12)
  x:13 14 15
  g:2
  a:y,,x
  a
(1 2 3
 4 5 6
 7 8 9
 10 11 12
 13 14 15)
  b: <(!#y),g
  b
0 1 2 4 3
  a[b]
```

```
(1 2 3
 4 5 6
 7 8 9
 13 14 15
 10 11 12)
  (y,,x)[<(!#y),g]
(1 2 3
 4 5 6
 7 8 9
 13 14 15
 10 11 12)
```

`a` is the concatenation of the original matrix with the vector to be inserted (which needs to be enlisted to perform matrix concatenation)

`(!#y),g` produces the indices of the rows of the original matrix, concatenated with the location where we desire to perform the insertion, which correspond to the desired ordering of the rows of `a` with the exception of the value of g will be repeated, the first instance corresponding to the last location before the insertion and the second instance corresponding to the insertion itself. K uses a stable grading algorithm (in the same sense as a "stable sort") so we can grade `(!#y),g` to obtain the permutation vector for indexing `a` that will produce the desired result.

We can also use `#` and `_` operations although the result is less elegant, requiring many parentheses `((1+g)#y),(,x),(1+g) _ y`. If instead we want to insert the row before position g it would be `((,x),y)[<g,!#y]` or `(g#y),(,x),g _ y`.

376. append y at the bottom of matrix x

```
  x:4 3#1+!12
  x
(1 2 3
 4 5 6
 7 8 9
 10 11 12)
  y:13 14 15
  x,,y
(1 2 3
 4 5 6
 7 8 9
 10 11 12
 13 14 15)
```

If we want to append a vector to a matrix as a row (instead of as individual items), we need to enlist the row first.

390. Conform table x rows to list y

```
  f390:{@[((1 0*+/^y)|^x)#0;!#x;:;x]}
  x:3 3#1+!9
  y:1 2 3 4
  f390[x;y]
(1 2 3
 4 5 6
 7  8  9
 0  0  0)
```

Conforming is done by augmenting the number of rows if needed (making the table have at least as many rows as `y`). In this idiom this is done by generating a matrix of zeros of the required size and then amending it at the positions that have a matching entry in `x`. The idiom in the original list is adding the count in all dimensions of `y` and it is not clear what is the intrnded application (if `y` is a list as indicated by the title we should just use its outermost dimension, e.g., if it is a list of character vectors we probably don't want the length of the character vectors to be a factor, just the count of character vectors).

This operation could be done more simply by calculating how many rows to append (the count of items in y minus the count of rows in x, with a lower bound of zero) and appending them `x,(0|(#y)-#x)#,&#x[0]`.

391. Conform table y columns to list y

```
  x:4 2 # 9
  y:5#8
  a:((0 1*+/^y)|^x)#0
  a
(0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0)
  a[;!(^x)[1]]:x
  a
(9  9  0  0  0
 9  9  0  0  0
 9  9  0  0  0
 9  9  0  0  0)
```

Conforming is done by augmenting the number of columns if needed (making the table have at least as many rows as `y`). In this idiom this is done by generating a matrix of zeros of the required size and then amending it at the positions that have a matching entry in `x` (the same process as idiom 390 but augmenting columns instead of rows). The idiom is adding the count in all dimensions of `y` and it is unclear why (if `y` is a list as indicated by the title we should just use its outermost dimension, e.g., if it is a list of character vectors we probably don't want the length of the character vectors to be a factor, just the count of character vectors).

This operation could be done more simply by calculating how many columns to append (the count of items in `y` minus the count of rows in `x`, with a lower bound of zero) and appending them to each row `x,\:(0|(#y)-#x)#&#x[0]`.

392. matrix from scalar or vector

```
  x:4
  ^x
!0
   (1+~#^x),:/x
,,4
  ^(1+~#^x),:/x
```

```
1 1
  x:7 8
  ^(1+~#^x),:/x
1 2
```

This idiom is calculating how many times the input needs to be enlisted, based on its count of dimensions `(1+~#^x)` which results in `2` for a scalar and `1` for inputs of higher dimension, and then uses a “DO form of over” (`n m/` where `n` is the count of iterations and `m` is a monadic function) for applying an enlist operation the required number of times. In this case, enlist needs to be specified as `,:` to clarify we want monadic `,`.

527. transpose planes of three-dimensional x

```
  x:2 2 2#!8
  x
((0 1
  2 3)
 (4 5
  6 7))
  +:'x
((0 2
  1 3)
 (4 6
  5 7))
```

We can apply a transpose (monadic `+:`) to each "row" of tensor `x`, in order to transpose at the deeper level.

528. vector (cross) product

```
  x:2 8 5 6 3 1 7 7 10 4
  y:6 9 1 1 6 7 1 4 1 5
  ((1!x)*-1!y)-(-1!x)*1!y
4 28 46 -27 -41 39 45 3 -19 -58
```

This is calculating the cross product by applying the formula that computes the cross-product in cartesian coordinates from the element-wise multiplication of the rotations in opposite directions of the coordinates of the vectors being multiplied and then subtracting the two rotations.

555. all axes of rectangular array x

```
  x:2 2 2 2#!16
  !#^x
0 1 2 3
```

We can produce indices for all axes of an array or tensor by enumerating the count of dimensions.

583. array and its negative

```
  x:1 -3 5
  x,'-x
(1 -1
 -3 3
 5 -5)
```

We can generate a columnar matrix from a vector and its negative by appending the vector and its negative at one level deep. We can also stack them as rows and transpose `+(x;-x)` although it results in a longer expression.

590. increasing rank of y to rank of x

```
  x:("abcd"
> "efgh")
  x
("abcd"
 "efgh")
  y:"ijkl"
  #^x
2
  #^y
1
  #^,y
2
  ((#^x)-#^y),:/y
,"ijkl"
```

We can use an enlist operation to increase the rank (count of dimensions) of a variable. If we calculate the difference in the rank of `x` and the rank of `y` and then use the DO form of `/` we will apply the enlist operation enough times to get the ranks equalized. Note that the rank of `y` must be less or equal to the rank of `x`, we cannot use the DO form of `\` with a negative count.

612. rank of array y (number of dimensions)

```
  x:2 1 2 1 3 1 4#0
  #^x
7
  #^9
0
  #^7 8 9
1
  #^(1 2 3;4 5 6)
2
```

The rank of a tensor is the count of its dimensions.

Charting and drawing

K does not have native graphics capabilities (with the notable exception of the oK implementation) but it is possible to provide some basic charting using characters (e.g., `X` or `*`).

166. bar chart of integer list x, down the page

```
  x:2 5 7 4 9 3 6
  xl:{(x#1),&y-x}
  " X"[|+xl\:[x;|/x]]
("    X   "
 "    X   "
 "   X X   "
 "   X X X"
 "  XX X X"
 "  XXXX X"
 "  XXXXXX"
 "XXXXXXX"
 "XXXXXXX")
  bd:{" X"[|+xl\:[x;|/x]]}
  bd[x]
("     X   "
 "     X   "
 "   X X   "
 "   X X X"
 "  XX X X"
 "  XXXX X"
 "  XXXXXX"
 "XXXXXXX"
 "XXXXXXX")```
```

`xl` generates a list of `x` ones followed by `y-x` zeroes (see 172). This function is applied to argument pairs of each x input and the largest value in the input. The results are "rotated counterclockwise" by transposing and reversing them, and then used to index the characters used for the barchart (spaces and `X`). We could also do the operation with a single `&`, `bd:{ "X "@|+{&x,y-x}\:[x;|/x]}`. If we use comparisons for generating the 0/1 vector we get `bd:{|+" X"x>\:!|/x}`.

170. horizontal bar chart of x with maximum z, normalized to length y

```
  x:4
  y:5
  z:10
  ad:{_ x*y%z}
  ad[2;5;10]
1
  ad[23;50;80]
14
  x:2 8 5 6 3 1 7 7 10 4
  xl:{(x#1),&y-x}
  xl\:[ad[x;y;z];y]
(1 0 0 0 0
 1 1 1 1 0
 1 1 0 0 0
 1 1 1 0 0
 1 0 0 0 0
 0 0 0 0 0
 1 1 1 0 0
 1 1 1 0 0
 1 1 1 1 1
 1 1 0 0 0)
  " X"[xl\:[ad[x;y;z];y]]
("X    "
 "XXXX "
```

```
 "XX    "
 "XXX   "
 "X     "
 "      "
 "XXX   "
 "XXX   "
 "XXXXX"
 "XX    ")
```

This is similar to idiom 166, using horizontal bars (eliminating the need for transpositions and reversals to do the counterclockwise rotation), but adding a normalization step `ad:{_ x*y%z}`. We can use the same simplification we did in 166, using `{&x,y-x}` and reversing the character vector to `"X "`.

```
  ad:{_ x*y%z}
  zl:{&x,y-x}
  x:2 8 5 6 3 1 7 7 10 4
  y:5
  z:10
  "X "[zl\:[ad[x;y;z];y]]
("X     "
 "XXXX "
 "XX    "
 "XXX   "
 "X     "
 "      "
 "XXX   "
 "XXX   "
 "XXXXX"
 "XX    ")
```

Or

```
  ad:{_ x*y%z}
  x:2 8 5 6 3 1 7 7 10 4
  y:5
  z:10
  " X"ad[x;y;z]>\:!y
("X     "
 "XXXX "
 "XX    "
 "XXX   "
 "X     "
 "      "
 "XXX   "
 "XXX   "
 "XXXXX"
 "XX    ")
```

171. horizontal bar chart of integer values x

(compare bh here with xl in 172)

```
  x:2 8 5 6 3 1 7 7 10 4
  bh:{@[&y;!x;:;1]}
  " X"[bh\:[x;|/x]]
("XX        "
 "XXXXXXXX  "
 "XXXXX     "
 "XXXXXX    "
 "XXX       "
 "X         "
 "XXXXXXX   "
 "XXXXXXX   "
 "XXXXXXXXXX"
 "XXXX      ")
```

This idiom is using `@` to amend a vector of all-zeros and changing the first `x` positions to `1`. After generating the 0/1 vector it uses the results to index into `" X"` (similar to idioms 166 and 170). Using comparisons instead, we could simplify this to `{" X"x>\:!|/x}`.

144. histogram

```
  x:13 _draw 12
  x
8 3 11 9 9 4 6 6 3 3 9 7 9
  h:{@[&1+|/x;x;+;1]}
  b:h[x]
  b
0 0 0 3 1 0 2 1 1 4 0 1
  c:(1+|/b)-b
  c
5 5 5 2 4 5 3 4 4 1 5 4
  d:|+(b#\:1),'(c#\:0)
  d
(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 1 0 0 0 0 0 1 0 0
 0 0 0 1 0 0 1 0 0 1 0 0
 0 0 0 1 1 0 1 1 1 1 0 1)
  " *"[d]
("            "
 "         *  "
 "   *     *  "
 "   *  *  *  "
 "   ** **** *")
```

Given a vector of nonnegative integers this idiom shows how to draw a histogram of frequencies using asterisks. The range for the histogram is given by `1+|/x`, the vector maximum plus one (for the zero). Once we have the range, we can use monadic `&` to generate a vector of zeros of the required size, and amend the items using the original vector as indices for which columns in the histogram to increment by one. The resulting vector `h` is a count of frequency, a “flat” histogram. To generate the “drawing” we first calculate the complementary count of frequency, the “whitespace” that we will need, by subtracting the count of frequency from its max value plus one (we want the “bars” to have at least one space between the top of the bar and the top of the chart). From the frequency count and its complement, we can generate 0/1 vectors and transpose and reverse them to get them into the desired shape, and finally we use the 0/1 vectors to index into spaces and asterisks to produce the final result.

An alternate way to produce the 0/1 vectors would be using monadic `&` on the vertically stacked c,b and then transposing the result.

```
  c,'b
(5 0
 5 0
 5 0
 2 3
 4 1
 5 0
 3 2
 4 1
 4 1
 1 4
 5 0
 4 1)
  &:'c,'b
(0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
 0 0 1 1 1
 0 0 0 0 1
 0 0 0 0 0
 0 0 0 1 1
 0 0 0 0 1
 0 0 0 0 1
 0 1 1 1 1
 0 0 0 0 0
 0 0 0 0 1)
  d:+&:'c,'b
  d
(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 1 0 0 0 0 0 1 0 0
 0 0 0 1 0 0 1 0 0 1 0 0
 0 0 0 1 1 0 1 1 1 1 0 1)
```

464. framing character matrix x

```
  x:4 4#"abcdefghijklmnop"
  x
("abcd"
 "efgh"
 "ijkl"
 "mnop")
  +"-",'(+"|",'x,'"|"),'"-"
("------"
 "|abcd|"
 "|efgh|"
 "|ijkl|"
 "|mnop|"
 "------")
```

If we have a character matrix with rows of fixed length we can “frame” it with `-` and `|` characters by prepending and appending a `|` to each row, then transposing, prepending and appending a `-` to each transposed row and finally, transposing again.

572. division by 0

```
  dz:{(~0=x)*y%x+x=0}
  y:10 15 -20
  x:2 0 0
  y%x
5 0i -0i
  dz[x;y]
5 0 0.0
```

If we want to produce a zero on divisions by zero instead of `0i`,`0n`, or `-0i`, one way to do that is to add a non-zero value to the denominator when `x=0` (e.g. the result of `x=0` itself, which adds 1), which avoids the division by zero, and then multiplying by zero if `x` is zero. An alternative solution using amend would be `@[y%x;&0=x;:;0]`. This idiom may be useful for some charting/plotting applications where it could be preferable displaying an "incorrect zero" rather than trying to chart infinites and NaNs.

605. indexing plotting characters with Boolean index

```
  x:~3 6 5 7 2<\:1+!7
  x
(1 1 1 0 0 0 0
 1 1 1 1 1 1 0
 1 1 1 1 1 0 0
 1 1 1 1 1 1 1
 1 1 0 0 0 0 0)
" *"[x]
("***    "
 "****** "
 "*****  "
 "*******"
 "**     ")
```

This is converting a list of 0/1 vectors into a corresponding list of space/asterisk vectors by using the 0/1 vectors to index a constant vector of the desired range.

174. move x into first quadrant

```
  sm:{x-&/x}
  x:(1 6 4;3 4 7;7 8 6)
  sm'[x]
(0 5 3
 0 1 4
 1 2 0)
```

This is adjusting each input vector by subtracting the minimum (ensuring that at least one item will be zero and the remaining items will be greater than zero)

Conversions between numbers and character vectors

This section presents various techniques for converting character vectors to numbers, in both directions.

93. numbers from alphanumeric matrix

```
  x:4 3#" 1 12 0.5"
  x
(" 1"
 " 12"
 "  "
 "0.5")
  x=" "
(1 1 0
 1 0 0
 1 1 1
0 0 0)
  &/'x=" "
0 0 1 0
  &&/'x=" "
,2
  z:.:'x
  z
(1;12;;0.5)
  @[z;,2;:;0]
(1;12;0;0.5)
/ Putting it all together
  @[.:'x;&&/'x=" ";:;0]
(1;12;0;0.5)```
/ But there may be no all-blank rows
  y=:4 3#" 1 123450.5"
  y
(" 1"
 " 12"
 "345"
 "0.5")
  @[.:'y;&&/'y=" ";:;0]
(1;12;345;0.5)
  na:{@[.:'x;&&/'x=" ";:;0]}
```

This is an application of `.` for evaluating character vectors `z:.:'x` and shows a way to make a special case for converting blank items to zeros, using tetradic `@`. We can use `@` to “build up” a vector with special cases, providing the base case in the first argument, a second argument of an index selection function (or a list of indices) that corresponds to items to be replaced with the special case, the assignment operator in the third argument and the value for the special case in the fourth (e.g., modifying this idiom to use `0N` for blank rows instead of zero would be a trivial change). A potential concern would be cases where the index selection function results in a null. Tetradic `@` will operate on all of the items if the second argument is null. Note that this concern would be rare as practically all index selection functions will result in empty lists when there are no matches, and an empty list in the second argument will return the first argument unmodified, as expected.

As any other applications of `.` eval, we must be careful about injection attacks, see idiom 78.

94. number from alphanumeric x, default y

```
  x:""
  y:"-1"
  .((x~"")#"y"),x
"-1"
  x:"234.5"
  .((x~"")#"y"),x
234.5
```

This idiom compares the input to an empty string, and uses the 0/1 result to determine how many copies of `"y"` to prepend before converting using eval `.`. Something to note is that `"y"` is what gets evaluated, not its contents, the result is the character vector `"-1"` not the number `-1`. There are many possible variations that achieve the same result, appending instead of prepending requires fewer parentheses . `x,(x~"")#"y"`. We can use conditional evaluation or indexing instead of prepend/append, `.:[x~"";"y";x]` or `.(x;"y")@x~""`, and we should use numeric conversion if the input is coming from a source that could be subject to injection attacks `(r 1;y)@(x~"")|*r:.[0.0$;x;:]`. It should be noted that numeric conversion can be quite nuanced, e.g., if importing data produced by a non-K source and it contains the string `0I`, do we want to handle that as erroneous data to be replaced by the default value or do we want to let it get converted to an infinity?

For application-strength conversion we should probably abstract number conversion into its own routine or a library, possibly implementing a parser for greater flexibility. Here is an example for such a parser. (courtesy of S. Apter)

```
  t:{-1+2_sv~x _in\:/:0 1_"-0123456789"}      / types
  m:(1 1 2;2 1 2;2 2 2)                        / states x types
  f:0 m\t@                                     / rule
  g:{:[(-1+#m)_in f y;x;. y]}                  / value
  h:g[0N]                                      / bad value
  a:"-12"
  b:"133"
  c:"12-13"
  d:"1a33b"
f a
f b
f c
f d
h a
h b
h c
h d
\

0 1 1 1

0 1 1 1

0 1 1 2 2 2
```

```
0 1 2 2 2 2

-12

133

0N

0N
```

111. count of format of x

```
  cf:{#$x}
  cf 12.345
6
  cf -1
2
  cf 1e-12
6
  $1e-12
"1e-012"
```

We can evaluate how much space will be needed for printing any non-character data by converting to a character vector and getting the length of the result.

95. numeric from proper alphanumeric nonnegative integer

```
  x:"123 438"
  d:"0123456789 "
  1 _ . "0 ",((#x)*&/x _lin d)#x
123 438
  x:"123 45a 789"
  1 _ . "0 ",((#x)*&/x _lin d)#x
0
```

This idiom presents another use of eval `.` for converting numbers, this time including a check that all input is a digit or a space and returning a zero if not.

Because of the constraints, this idiom is protected from injection attacks, although the choice of returning a zero on a failure to convert makes this idiom applicable exclusively for scenarios where zero is not a potential input.

99. numeric vector from evaluating rows of character matrix

```
  x:2 5#"1+2 41+3 6"
  x
("1+2 4"
 "1+3 6")
  .:'x
(3 5
 4 7)
  ,/.:'x
3 5 4 7
```

This is an application of idiom 98 (execute rows of character matrix), concatenating the results.

101. sum numbers in character matrix

```
  x:$!5
x
(,"0"
 ,"1"
 ,"2"
 ,"3"
 ,"4")
  +/ .:' x
10
```

This is converting each number using eval `.` and adding them up. See comments in idioms 78 and 94 for considerations about eval conversions of numbers.

106. leading zeros for positive integers x in field width y

```
  x:10 _draw 40
  x
37 36 17 38 29 4 31 12 35 25
  y:3
  z:$:'x+(y-1){x*10}/10
  z
("1037"
 "1036"
 "1017"
 "1038"
 "1029"
 "1004"
 "1031"
 "1012"
 "1035"
 "1025")
  u:1 _/: z
  u
("037"
 "036"
 "017"
 "038"
 "029"
 "004"
 "031"
 "012"
 "035"
 "025")
```

This idiom is adding leading zeroes by adding a number that is large enough and has zeroes in the target positions so that it does not modify our target digits, and then converting to a character vector and removing the number we added. The calculation for the number to add is done by successive multiplication of the number 10 (using multiplication plus the triadic "DO" form of over `/`), that for `y:3` results in adding 1000 to the numbers to format.

452. number of positions in integer x

```
  log10:{log[x]%log[10]}
  dp: {1+(x<0)+_ log10[_abs[x+0=x]]}
  dp 1234
4
  dp -1234
5
  dp 0
1
  dp 1
1
  dp 7
1
  dp 12345678
8
```

This is calculating the number of characters required to represent an integer by adding 1 to the log10 of the absolute value (adjusted to 1 if the input is the number 0), and then adding one more position for the sign if the number is negative. Of course, we can also do `dp:{#$x}`.

456. number of digits in nonnegative integer x

```
  np:{1+_ log10 x+0=x}
  x:0 13 523 16008
  np x
1 2 3 5
```

This is a simplified version of 452, taking advantage of the knowledge that the numbers are nonnegative.

Numeric base conversions

K provides functions for conversions between vectors and scalars (`_vs` and `_sv` in K3). These functions provide generic base conversions and this section explores its applications.

46. transposed formatted integers 1 through x

```
  x:15
  q:10 _vs 1+!x
  q
  (0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
   1 2 3 4 5 6 7 8 9 0 1 2 3 4 5)
  "0123456789"[q]
("000000000111111"
 "123456789012345")
```

When `_vs` is applied to a vector, it will produce a transposed base conversion of each item. By doing a base 10 conversion we can separate into decimal digits. And then by indexing into "0123456789" we can map each decimal digit into their character vector “formatted” representation.

49. hexadecimal from decimal

```
  hex:"0123456789abcdef"
  hd:{+hex[16 _vs x]}
  x:10 12 19 1 28 100
  hd x
("0a"
 "0c"
 "13"
 "01"
 "1c"
 "64")
  y:10 12 19 1 28 300
  hd y
("00a"
 "00c"
 "013"
 "001"
 "01c"
 "12c")
```

Continuing on the topic of base conversion, when the base is greater than 10, we will need to provide a mapping for the "traditional" representation. This idiom is an example for hexadecimal mapping.

52. truth table of order x

```
  tt:{2 _vs !_2^x}
  tt 1
,0 1
  tt 2
(0 0 1 1
 0 1 0 1)
  tt 3
(0 0 0 0 1 1 1 1
 0 0 1 1 0 0 1 1
 0 1 0 1 0 1 0 1)
```

The truth table of order n is the same as the digits for counting in binary up to 2^x. Note that the results from the exponentiation are a floating-point value, and we need to truncate them before we can enumerate them.

53. decimal digits from integer

```
  dd:{10 _vs x}
  dd 2001
2 0 0 1
  dd 123456789
1 2 3 4 5 6 7 8 9
```

Converting a decimal number to digits is the same as using a base-10 conversion.

63. represent x in radix 10 100 1000

```
  x:123456
  10 100 1000 _vs x
```

```
1 23 456
  x:123456789
10 100 1000 _vs x
  4 56 789
```

Base conversion is not limited to a single base. It is possible to provide a vector of bases, like this example with increasing base sizes.

54. represent x in base y

```
  x:256
  y:16
  y _vs x
1 0 0
  x:36
  y:2
  y _vs x
1 0 0 1 0 0
  x:123
  y:10
  y _vs x
1 2 3
```

This is a straightforward application of `_vs` to base conversion. As mentioned, converting to bases greater than 10 may require an additional mapping (see idiom 49).

56. hexadecimal from decimal characters

```
  hex:"0123456789abcdef"
  x:" abcdef"
  _ic x
32 97 98 99 100 101 102
  16 _vs _ic x
(2 6 6 6 6 6 6
 0 1 2 3 4 5 6)
  hex[16 _vs _ic x]
("2666666"
 "0123456")
  +hex[16 _vs _ic x]
("20"
 "61"
 "62"
 "63"
 "64"
 "65"
 "66")
  " ",'+hex[16 _vs _ic x]
(" 20"
 " 61"
 " 62"
 " 63"
 " 64"
 " 65"
 " 66")
  ,/" ",'+hex[16 _vs _ic x]
" 20 61 62 63 64 65 66"
  x:"GOLDEN"
  ,/" ",'+hex[16 _vs _ic x]
"47 4f 4c 44 45 4e"
```

This combines the `_ic` (integer from character) operation with idiom 49 and uses string concatenation to produce the hex values for a character vector string.

75. decimal from hexadecimal

```
  x:("ff";"a9";"8ac";"ffff")
  x
("ff"
 "a9"
 "8ac"
 "ffff")
  "0123456789abcdef"?/:"ff"
15 15
  "0123456789abcdef"?/:/:x
(15 15
 10 9
 8 10 12
 15 15 15 15)
  16 _sv/: "0123456789abcdef"?/:/:x
255 169 2220 65535
```

Converting from hex to decimal requires 2 steps, first finding the value of the digit using a mapping to convert from letters to integers, then using a reverse base conversion and getting a decimal value.

66. selection by encoded list

```
  x:1 0 1
  "abcdefgh"[2 _sv x]
"f"
  x:0 0 0
  "abcdefgh"[2 _sv x]
"a"
  x:1 1 1
  "abcdefgh"[2 _sv x]
"h"
```

`_sv` is the reverse operation of `_vs`. Idiom 45 shows how to convert an integer into binary vector, and this idiom uses the reverse conversion from binary vector to integer, then uses the result to index into an array

342. Arabic from roman number

```
  x:"MCMIX"
  "MDCLXVI"?/:x
0 2 0 6 4
  a:0,1000 500 100 50 10 5 1["MDCLXVI"?/:x]
  a
0 1000 100 1000 1 10
  a<1!a
1 0 1 0 1 0
  _ a*-1^a<1!a
0 1000 -100 1000 -1 10
  +/_ a*-1^a<1!a
1909
  ar:{+/_ a*-1^a<1!a:0,1000 500 100 50 10 5 1["MDCLXVI"?/:x]}
  ar[x]
1909
```

This starts with a simple lookup conversion, we do a find each right of `x` in `"MDCLXVI"` and use the result to index into a "weights" vector `1000 500 100 50 10 5 1`. We also prepend a zero, to prepare for rotation. We then rotate the resulting vector by one position and compare it to the original, to determine which elements should be subtracted and which ones added (note that instead of rotation we could have used `<':` (prepending the zero after the operation instead of before). We need to convert the 0/1 vector to a 1/-1 vector. In the idiom it is done through exponentiation of -1 and flooring `_` but we could also have done it through indexing `(1 -1)@a<1!a`. We finally multiply by the weights vector and sum, to obtain the desired result. The modified version without exponentiation would be `ar:{+/a*(1 -1)@a<1!a:0,1000 500 100 50 10 5 1["MDCLXVI"?/:x]}`.

Date and time manipulation

K has its own integer representation of time, with the lowest non-infinite integer corresponding to December 13th, 1966 at 20:45:54, and counting the number of seconds since that date. This section presents idioms for converting and formatting time/date variables, as well as other manipulations of time.

57. vector from date

```
  _t
-1155413254
  _gtime _t
19980522 35236
   *_gtime _t
19980522
  100000 100 100 _vs *_gtime _t
1998 5 22
```

We can use a base conversion to split number dates (as expressed in the first item returned by `_gtime` or `_ltime`), much in the same way as we used a base conversion to extract digits (in idiom 53). Since we want to “shift” to the next value by pairs of digits we use base 100 instead of base 10. For the leftmost part (year), we can use any number higher than the dates we want to be able to process.

```
  100000 100 100 _vs 19980522
1998 5 22
  200000 100 100 _vs 19980522
1998 5 22
  500000 100 100 _vs 19980522
```

```
1998 5 22
```

64. represent integer time hhmmss as character time, items separated by colons 64a. local time as integer of form hhmmss

```
  x:_ltime _t
  x@:1
  x
45628
  q:2 _ $ 1000 _sv 100 _vs 1000000 + x
  q
"04056028"
  @[q;2 5;:;":"]
"04:56:28"
```

This is showing how we can use base conversions to selectively split into pairs of digits and insert additional zeroes, then converting into a string and replacing selected zeroes by `":"`. We also need something to prepend zeroes when the numeric time is small (less than 1000000), for example adding 1000000 and then cutting the extra 1 from the result (or we can just prepend a string of zeroes and cut it).

Solving this problem can also be approached from an indexing perspective, which seems simpler:

```
  x:_ltime _t
  p:-7#"000000",($x 1),":"
  p
"074551:"
  p@0 1 6 2 3 6 4 5
"07:45:51"
```

72. encode date as integer

```
  x:*_ltime _t
  x
19980522
```

This is a straightforward application of `_ltime`, with a "first" `*` operation to extract just the date portion from the result

65. represent integer date in form yyyymmdd as character date, parts separated by "/"

```
  x:*_ltime _t
  x
19980521
  1000 _sv 10000 100 100 _vs x
1998005021
  q:$1000 _sv 10000 100 100 _vs x
  q
"1998005021"
  @[q;4 7;:;"/"]
"1998/05/21"
```

Very similar to idiom 64, with different sizes and no left-padding with zeroes.

If we approach the problem from a string indexing perspective this would be the solution:

```
  x:*_ltime _t
  x
19980521
  p:($x),"/"
  p@0 1 2 3 8 4 5 8 6 7
"1998/05/21"
```

104. date ascending format

```
  1 _ ,/".",/:$|.:'0 4 6 _ $* _ltime _t
"25.5.1998"
```

We can obtain the current date as a number from the first element in `_ltime`. If we convert it to a string and manipulate the characters individually we can reconstruct the date in whichever format we prefer. By converting to a string, cutting it up and then running eval `.` on each component we can remove any leading zeroes in month or day, and then by prepending with a period `"."`, flattening, and removing the initial period (because we are prepending one too many in the previous step) we obtain the date in the desired format. If we prefer not using eval, we can use a base conversion for removing the leading zeroes `1_,/".",'$|10000 100 100 _vs *_ltime _t`. If the leading zeroes are not a concern, we can use simple indexing `(".",$*_ltime _t)@7 8 0 5 6 0 1 2 3 4`.

107. current date, American

```
  x:*_ltime _t
  x
19980526
  10000 100 100 _vs x
1998 5 26
  1!10000 100 100 _vs x
5 26 1998
  $1!10000 100 100 _vs x
(,"5"
 "26"
 "1998")
  " ",/:$1!10000 100 100 _vs x
(" 5"
 " 26"
 " 1998")
  ,/" ",/:$1!10000 100 100 _vs x
" 5 26 1998"
  1 _ ,/" ",/:$1!10000 100 100 _vs x
"5 26 1998"
  "0123456789 "?/:1 _ ,/" ",/:$1!10000 100 100 _vs x
5 10 2 6 10 1 9 9 8
  d:"0123456789"
  e:d,"/"
  b:10000 100 100
  k:{1 _ x}
  s:" "
  e[d?/:k ,/s,/:$1!b _vs x]
```

```
"5/26/1998"
  da:{e[d?/:k,/s,/:$1!b _vs x]}
  da[*_ltime _t]
"5/26/1998"
```

This idiom goes through successive manipulation of base conversion, rotation of the first items, prepending spaces and concatenating, cutting the leading space, finding the digits, and replacing the spaces with slashes `"/"`

Instead of using spaces and converting to slashes, or converting to characters by using find `?`, we could make a small variation on the functions in the comments for idiom 104 and get the simpler form `1_,/"/",'$1!10000 100 100 _vs *_ltime _t` or, if leading zeroes in month and day are not a concern, we can just modify the indexing and not even need a rotation `("/",$*_ltime _t)@5 6 0 7 8 0 1 2 3 4`.

105. current time of 12-hour clock (AM & PM)

```
  y:_t
  y
-1155069892
  x:*|_ltime y
  x
201508
  hms:{1 _,/":",/:1 _'$100+100 _vs x!120000}
  hms x
"08:15:08"
  ap:{"AP"[115959<x],"M"}
  ap x
"PM"
  hm:{{(hms x)," ",ap x}[*|_ltime x]}
/ Ordinarily one would use `_t` as the argument to `hm`
  hm x
"08:15:08 PM"
```

`hms` is using a modulo operation to convert the 24h value to a 12h value, then applying a transformation similar to idiom 104 to convert it to hh:mm:ss format and finally comparing to 11:59:59 to determine if “AM” or “PM” needs to be appended.

A solution based on indexing would be:

```
  x: *|_ltime _t
  hm:{(($1000000+x-120000*(125959<x)),("A";"P")[115959<x],"M :")@1 2
10 3 4 10 5 6 9 7 8}
  x
74044
  hm x
"07:40:44 AM"
  hm x-70000
"00:40:44 AM"
  hm x+50000
"12:40:44 PM"
  hm x+60000
"01:40:44 PM"
```

463. is x a leap year

```
  ly:{(+/0=x!/:4 100 400)!2}
  x:1900 1901 1904 2000
  ly x
0 0 1 1
```

This idiom first calculates the remainders of dividing the year number by 4, 100 and 400, compares the result to 0 and checks if the sum of the results is even (since 100 is a multiple of 4, and 400 is a multiple of 4 and 100, this will identify integers divisible by 4 except if they are also divisible by 100, with an exception to the exception when they are also divisible by 400).

74. number of days in month x of Gregorian year y (ly from 463)

```
  x:1
  y:1904
  :[2=x
> 28+ly y
> (0,12#7#31 30)[x]]
31
  x:2
  :[2=x
> 28+ly y
> (0,12#7#31 30)[x]]
29
```

This idiom is first checking if the month is February and in that case returning 28 plus the result from ly (idiom 463, that returns 1 if the input is a leap year or zero otherwise). The result for any month other than February is calculated by indexing into the vector `(anyinteger 31 anyinteger 31 30 31 30 31 31 30 31 30 31)`. We can generate this vector by reshaping and repeating `(31 30)` vectors. Alternatively we could have used amending and indexing `:[2=x;28+ly y;@[13#31;4 6 9 11;:;30]@x]`.

Mathematical computations

This section presents how to apply K to a variety of mathematical problems

603. conditional change of sign

```
  y:1+!6
  y
1 2 3 4 5 6
  x:0 1 0 1 1 0
  y*1 -1[x]
1 -2 3 -4 -5 6
```

If we want to change signs based on a 0/1 vector, we can map the 0/1 vector into a `(1;-1)` vector by using indexing, and then use element-wise multiplication.

457. is x integral

```
  ii:{x=_ x}
  x:67 -120 3.83 -5.5
  ii x
```

```
1 1 0 0
```

We can compare a number to its truncation to determine if it is integral. The same operation can be applied to a vector to get the element-wise result.

62. integer and fractional parts of positive x

```
  x:12.3 23.4 5.33 8.999
  if:{(_ x),'x-_ x}
  if x
((12;0.3)
 (23;0.4)
 (5;0.33)
 (8;0.999))
```

`_ x` will truncate each element in the input vector, if we subtract it from the input vector the operation will be performed element-wise and produce the fractional part (see idiom 478 for an alternative, shorter calculation). By using concatenate each `,'` we get the column-wise concatenation. Alternatively, we could enlist each row, then concatenate and transpose, which seems to be marginally faster.

```
  if2:{+(,_ x),,x!1}
  if2 x
((12;0.3)
 (23;0.4)
 (5;0.33)
 (8;0.999))
```

Something else to note in the original example is that `if` is a particularly poor choice of name for a user-defined function, this is one idiom that you probably don't want to copy-paste verbatim into your code!

478. fractional part

```
  x:0 1 -2 3.4 -5.6 -6.1
  x!1
0 0 0 0.4 0.4 0.9```
```

We can calculate the fractional part as the remainder of dividing by one. Extracting the fractional part of a positive floating-point number is well defined but there are multiple ways of extending it to negative numbers. One definition uses floor subtraction (Graham, Knuth, Paashnik & Oren (1992) Concrete mathematics: a foundation for computer science), and the remainder used in this idiom provides a shorter expression for this calculation. A second definition uses the numbers after the decimal period (Dainith (2004), A Dictionary of Computing) and a third definition converts the second definition to an odd function by preserving the sign. These 2nd and 3rd definitions are presented in idioms 465 and 476.

465. magnitude of fractional part

```
  x:6.13 -6.13
  _abs[x]!1
0.13 0.13
```

We can extract the digits after the decimal period by taking the remainder of dividing the absolute value by one.

476. fractional part with sign

```
  x:0.2 2.3 -0.2 -1.8 0 5 -7
  sg:{(x>0)-(x<0)}
  (sg x)*(_abs x)!1
0.2 0.3 -0.2 -0.8 0 0 0```
```

We can calculate the signed fractional part by taking the remainder of dividing the absolute value by one (see idiom 465) and then restoring the original sign by multiplying with the `sg` operation (see idioms 363 and 475).

453. round to nearest even integer

```
  re:{_ x+~1>x!2}
  x:0.9 1 2.5 3.1 -0.2 -1.9
  re x
0 2 2 4 0 -2
```

This is calculating the vector of remainders and adding 1 to each element in the original vector if they corresponding remainder is greater than or equal to one `~1>`. Finally, the results are truncated to integers. Notice how this operation works for odd and negative numbers.

454. rounding, but to nearest even integer if fractional part is 0.5

```
  rn:{_ x+0.5*~0.5=x!2}
  x:23.6 40.5 3.2 -14.02 3.5 4.5
  rn x
24 40 3 -14 4 4
```

This is similar to the previous idiom but scaled by 0.5 (in the comparison to the remainder and in how much is added to the original vector).

460. round y to x decimals

```
  rn:{(10^-x)*_ 0.5+y*10^x}
  y:3.3256789
  x:3
  rn[x;y]
  rn[x;y]
3.326
```

We can round to x decimals by multiplying for the corresponding power of 10, adding 0.5 (for rounding to the nearest value), truncating the result and then dividing by the same power of 10 (multiplying by the negative power of 10). This can be applied to an atom or a vector.

461. round to nearest hundredth

```
  rh:{0.01*_0.5+x*100}
  x:3.1414 2.71828 -12.66666
  rh x
3.14 2.72 -12.67
```

This is a specific application of idiom 460 when the number of decimals is 2 and using precalculated values, `10^2` is `100` and `10^-2` is `.01`. This can be applied to an atom or a vector.

462. round to nearest integer

```
  ri:{_0.5+x}
  x:4.5 3.21 80.9 -2.4 -9.6
  ri x
5 3 81 -2 -10
```

This can be considered as another application of idiom 460 when the number of decimals is zero and we're using precalculated values. `10^0` is `1` and `10^-0` is `1` as well, therefore the corresponding multiplications can be omitted. This can be applied to an atom or a vector.

474. round x to zero if magnitude less than y

```
  x:1e-4 -1e-8 -1e-12 1e-16
  x*~y>_abs x
0.0001 -1e-008 0 0
```

We can produce a 0/1 vector identifying items with magnitudes less than `y` by computing `y>_abs x`. A simple solution could use `&` and an amend operation to produce the desired result `@[x;&y>_abs x;:;0.]` but this idiom is zeroing the elements in the original vector by using element-wise multiplication by the negated 0/1 vector instead, which results in shorter code.

87. number of decimals (maximum 7)

```
  nd:{:[1=#x;0;-2+#x]}
  ff:{$x-_ x}
  ff 6.567
"0.567"
  nd ff 1.234
3
  nd ff 1234
0
  nd ff 78.1234567
7
  nd ff 78.12345678
7
"0.1234568"
```

This is extracting the decimal part (see idiom 62) and converting it to a character vector, then counting the length (with a special case for length 1 which means the fractional part of the number is exactly zero). For negative numbers, ff will calculate the fractional part using, which is different from the numbers after the decimal point, but it has the same number of digits. The maximum number of decimals is limited by the configured precision `\p` which is 7 by default (but should not really be relied on, because it can be changed).

149. number of decimals in x, maximum y

```
  nd:{+/~0=(-_-x*10^y)!/:-_-(10^y)*10^-!y+1}
  x:1.4321 1.21 10
  y:3
  nd[x;y]
3 2 0
  10^-!y+1
1 0.1 0.01 0.001
  -_-(10^y)*10^-!y+1
1000 100 10 1
```

```
  -_-x*10^y
1433 1210 10000
  (-_-x*10^y)!/:-_-(10^y)*10^-!y+1
(433 210 0
 33 10 0
 3 0 0
 0 0 0)
  ~0=(-_-x*10^y)!/:-_-(10^y)*10^-!y+1
(1 1 0
 1 1 0
 1 0 0
 0 0 0)
  +/~0=(-_-x*10^y)!/:-_-(10^y)*10^-!y+1
3 2 0
```

This calculates the number of decimals without using conversions to string (as done in idiom 87), which enables support for any type of numbers, including those that would require mantissa and exponent notation. `10^-!y+1` is the value for decimal positions up to the maximum `y`. `-_-(10^y)*10^-!y+1` is the corresponding multiplier (`-_-` does a "round up" operation). This expression is using math to reverse the resulting list, but it is simpler if we just use reverse `|`. `-_-x*10^y` is the input multiplied by the largest multiplier. `(-_-x*10^y)!/:-_-(10^y)*10^-!y+1` is the modulus for each of the multipliers, which gives us successive removals of digits. Finally, we evaluate which entries are non-zero and count.

A potential problem in this idiom is that integer overflow can happen in some of the internal calculations (interestingly, some overflows may still produce a correct output for the purposes of this idiom). We can keep the intermediate results as floating point numbers and truncate at the end. The resulting expression is simpler: `+/~0=_(x*10^y)!/:|10^!y+1`.

470. items of x divisible by y

```
  x:10 _draw 100
  x
95 33 64 10 78 1 47 20 92 95
  y:4
  x!y
3 1 0 2 2 1 3 0 0 3
  0=x!y
0 0 1 0 0 0 0 1 1 0
  &0=x!y
2 7 8
  x[&0=x!y]
64 20 92
```

we can produce a 0/1 vector that identifies the items divisible by `y` by comparing to zero the remainder of dividing by `y`. From the 0/1 vector we can use indexing and `&` to extract the values from the original vector.

473. is x even

```
  x:1 2 3 4 5
  ~x!2
```

```
0 1 0 1 0
```

Determining if a number is even can be done by negating the remainder after dividing by two.

175. primes to n

```
  n:10
  x:1+!n
  x
1 2 3 4 5 6 7 8 9 10
  x!/:x:1+!n
(0 0 0 0 0 0 0 0 0 0
 1 0 1 0 1 0 1 0 1 0
 1 2 0 1 2 0 1 2 0 1
 1 2 3 0 1 2 3 0 1 2
 1 2 3 4 0 1 2 3 4 0
 1 2 3 4 5 0 1 2 3 4
 1 2 3 4 5 6 0 1 2 3
 1 2 3 4 5 6 7 0 1 2
 1 2 3 4 5 6 7 8 0 1
 1 2 3 4 5 6 7 8 9 0)
  +/0=x!/:x:1+!n
1 2 2 3 2 4 2 4 3 4
  2=+/0=x!/:x:1+!n
0 1 1 0 1 0 1 0 0 0
  &0,2=+/0=x!/:x:1+!n
2 3 5 7
  pn:{[n]&0,2=+/0={x!/:x}1+!n}
  pn 30
2 3 5 7 11 13 17 19 23 29
```

This idiom is finding all primes by using the definition of prime numbers as numbers that are divisible only by one and by itself. Given the range 1 to n it calculates all the division remainders and then counts the number of zeroes (exactly divisible items). Primes will have exactly two. Since the range is 1…n, it needs to be adjusted by 1, in this case it is done by prepending an item before applying `&` (we could also have simply added one, `1+&2=+/0=x!/:x:1+!n`). This technique is simple but not particularly optimized. If `n` is potentially large we may want to look for a sieve requiring fewer calculations.

260. first 10 figurate numbers of order x

```
  fg:{x+\/10#1}
  fg 0
1 1 1 1 1 1 1 1 1 1
  fg 1
1 2 3 4 5 6 7 8 9 10
  fg 2
1 3 6 10 15 21 28 36 45 55
  fg 3
1 4 10 20 35 56 84 120 165 220
  fg 4
1 5 15 35 70 126 210 330 495 715
```

This is taking a vector of ones of the desired length (10 in this case, as we're looking for the first 10 figurate numbers) and using the "do form" of monadic `/` to apply a sum scan as many times as required. It may be easier to read by adding parentheses to show how the expression is parsed `x(+\)/10#1`. Monadic `n f/` applies a monadic function `n` times, for example `3 (+\)/ x` is the same as `+\ +\ +\ x`.

302. x first triangular numbers

```
  x:6
  +\!x
0 1 3 6 10 15
```

Triangular numbers count the numbers of "dots" arranged in equilateral triangles with n dots in each successive row. They are the figurate numbers of order 2 and we could use idiom 260 but we can simplify it further by knowing that each successive row has one more "dot" than the previous so instead of the general figurate number formula we can use an enumeration as the starting point and just apply a running sum.

450. arithmetic precision of system in decimals

```
  log10:{(_log x)%_log 10}
  log10 3
0.47712125471966244
  _ _abs log10 _abs 1-3*%3
0I
  /This indicates that the precision is infinite (which isn't true)
  /and is a consequence of
  3*%3
1.0
```

This is an observation that K can sometimes optimize mathematical operations directly, rather than just calculating the floating-point results and propagating truncation errors.

459. leading digit of numeric code abbb

```
  ld:{_ x%1000}
  x:6 _draw 10000
  x
1319 8629 6581 6988 790 9045
  ld x
1 8 6 6 0 9
```

If we have a 4-digit numeric code, we can obtain the leading digit by dividing by 1000 and truncating the result. If the numeric code is smaller than 1000 the leading digit returned will be zero.

479. last part of abbb

```
  x:1234 5678 9012 345 6789
  x!1000
234 678 12 345 789
```

We can extract the last 3 digits of integers with 3 digits or more by calculating the remainder after dividing by 1000. If the integers have fewer than 3 digits then the whole integer will be returned.

475. increase absolute value without sign change

```
  x:0 -1 2 -3 4 -5
  sg:{(x>0)-(x<0)}
  y:10
  (sg x)*y+_abs x /problem with x=0
0 -11 12 -13 14 -15
```

This idiom adds the increment to the absolute value and then uses the “sign” operation (see idiom 363 "solve quadratic") to restore the original signs. Since `sg` returns 1 for positive, -1 for negative and 0 for zero that means that zero will not be increased. If we wanted to treat zeroes as positive and increase them, we should change the sign operation to `sg2:{(~x<0)-(x<0)}`.

477. square x retaining sign

```
  x:0 -1 2 -3 4
  x*_abs x
0 -1 4 -9 16
```

If we apply an element-wise multiplication to a vector and the absolute values of the same vector, we will get the square of the original vector but retaining the sign of the original items (instead of becoming all positive).

142. number of combinations of n objects taken k at a time

```
  fac:{[n]:[n>1; n * _f[n-1];1]}
  bin:{[n;k]fac[n]%fac[n-k]*fac[k]}
  bin[12;7]
792.0
  bin[10;4]
210.0
```

This defines the factorial function as a simple recursion (we could also have defined it iteratively:`fac:{[n]:*/1+!n}`), then uses it to define a function that returns the binomial coefficient.

135. number of permutations of n objects taken k at a time

```
  fac:{[n]:[n>1; n * _f[n-1];1]}
  bin:{[n;k]fac[n]%fac[n-k]*fac[k]}
  pn:{[n;k]fac[k]*bin[n;k]}
  pn[5;3]
60.0
```

This is using the factorial and binomial coefficient functions from idiom 142 and using them to calculate the number of permutations.

136. Pascal's triangle of order x (binomial coefficients)

```
  pt:{+':0,x,0}
  4 pt\1
```

```
(1
 1 1
 1 2 1
 1 3 3 1
 1 4 6 4 1)
  pt pt pt pt 1
1 4 6 4 1
  4 pt/1
1 4 6 4 1
```

Pascal's triangle is also known as Khayyam triangle, 杨辉三角/楊輝三角（Yang Hui's triangle), or Tartaglia's triangle. Given one line of the triangle, `pt` calculates the next line of the triangle by adding pairs of items with `+':`. To keep the start and end at 1, `pt` is appending and prepending zeros prior to the addition (the resulting expression requires fewer parentheses than prepending and appending ones to the result). Calculating the triangle can be done by applying a "DO scan" `n pt\v` with `n` indicating the number of lines to generate and `v` the initial value (1st line of the triangle). Calculating the nth line can be done by applying the "DO over" form instead `n pt/v`. For calculating individual elements of the triangle, see idiom 135.

Geometry and trigonometry

133. degrees from radians

```
  x:0.5
  57.295779513082323*x
28.64789
```

Direct multiplication by the conversion constant `180÷π`. We can also use `x*90%_asin 1`.

134. radians from degrees

```
  x:0.5
  z:57.295779513082323*x
  z
28.64789
  0.017453292519943295*z
0.5
```

Direct multiplication by the conversion constant `π÷180`. We can also use `z*(_asin 1)%90`.

179. contour levels y at points with altitude x

```
 cl:{y[-1++/~y>x]}
  y:-100 0 10 100 1000 10000
  cl[-5;y]
-100
  cl[0;y]
0
  cl[99;y]
10
  cl[9;y]
0
  cl[10;y]
```

```
10
```

This idiom is counting how many `y` are lower or equal to `x` and then returning the next `y`. Note that it will fail if `x` is smaller than the smallest `y`. We could also have expressed this as `cl:{y@|/&~y>x}`.

224. extend distance table to next leg

```
  x:( 0 50 80 20 999
> 50 0 20 40 30
> 80 20 0 999 30
> 20 40 999 0 10
> 999 30 30 10 0)
  / notice x[0;2] is 80 while x[0;1]+x[1;2] is 70
  x(&/+)\:x
(0 50 70 20 30
 50 0 20 40 30
 70 20 0 40 30
 20 40 40 0 10
 30 30 30 10 0)
```

This is operating on a matrix that is symmetrical around its diagonal and it is adding each row to the whole matrix then finding the minimum of each resulting column. The following expressions are all equivalent `x(&/+)/:x` `&/'x+\:x` `&/'x+/:x`.

318. area of triangle with sides x (Heron's rule)

```
  x:3 4 5
  hr:{(*/(+/x%2)-0,x)^0.5}
  hr[x]
6.0
```

This is Heron’s rule expressed in vector form. The “traditional” representation is the square root of `s(s-a)(s-b)(s-c)` where `s` is the semiperimeter, `(a+b+c)÷2`. By doing `s-(0;a;b;c)` we can get a simpler vector expression. A minor optimization could be calculating the semiperimeter as `.5*+/x` (one multiplication vs 3 divisions).

131. complementary angle (arccos sin x)

```
  x:0.25
  _acos[_sin x]
1.320796
  x+_acos[_sin x] / should be 0.5*pi, approximately
1.570796
  2*x+_acos[_sin x]
3.141593
```

Direct application of `_acos` and `_sin` and the corresponding trigonometric relationship.

132. rotation matrix for angle x (in radians) counter-clockwise

```
  ((_cos[x];-_sin[x]);(_sin[x];_cos[x]))
(0.9689124 -0.247404
 0.247404 0.9689124)
```

This idiom calculates a matrix that when multiplied (using `_matmul`) by another matrix whose column vectors `v` represent the cartesian coordinates of a point, results in another matrix whose column vectors contain the coordinates that correspond to the points obtained by rotating the original set of points counterclockwise through an angle θ about the origin of the Cartesian coordinate system.

Calculus and series

This section presents how to apply K to various problems of calculus and series

199. multiplication table of order x

```
  mt:{{x*\:x}[1+!x]}
  mt 5
(1 2 3 4 5
 2 4 6 8 10
 3 6 9 12 15
 4 8 12 16 20
 5 10 15 20 25)
```

`1+!x` produces a one-based range of the required length and `x*\:x` multiplies each item by a copy of the range.

155. greatest common divisor of list x

```
  gcd:{*|1+&&/'0=x!/:1+!&/x}
  x:6 9 12
  gcd x
3
  &/x
6
  !&/x
0 1 2 3 4 5
  1+!&/x
1 2 3 4 5 6
  x!/:1+!&/x
(0 0 0
 0 1 0
 0 0 0
 2 1 0
 1 4 2
 0 3 0)
  0=x!/:1+!&/x
(1 1 1
 1 0 1
 1 1 1
 0 0 1
 0 0 0
 1 0 1)
  &/'0=x!/:1+!&/x
1 0 1 0 0 0
  &&/'0=x!/:1+!&/x
0 2
```

```
  *|1+&&/'0=x!/:1+!&/x
3
```

The steps in the gcd calculation are:

a) Get an initial list of gcd candidates by obtaining the min value in the input and enumerating the numbers between one and this value (the gcd cannot be greater than the smallest input)

b) Find all the possible divisors for each input by getting the remainder (mod) after dividing each input value by the previously calculated enumeration and finding which ones are zero

c) Find all common divisors by using a "Boolean and" (min) on the rows to find those with all ones

d) Find the gcd by getting the larger of the common divisors

451. arithmetic progression from x to y with step g

```
  ap:{[x;y;g]x+g*!1+_(y-x)%g}
  ap[3;20;5]
3 8 13 18
  ap[3;-20;-5]
3 -2 -7 -12 -17
```

This idiom is calculating the length of the progression `1+_(y-x)%g`, enumerating it and then scaling and translating the enumeration `x+g*`.

557. arithmetic progression of y numbers from x with step g

```
  x:5
  y:8
  g:100
  x+g*!y
5 105 205 305 405 505 605 705
  ap:{[x;g;y]x+g*!y}
  ap[x;g;y]
5 105 205 305 405 505 605 705
```

We can calculate the arithmetic progression of `y` numbers from `x` with step `g` by enumerating the count of items `!y` and then multiplying by the step size to scale it `g*!y` and finally adding the starting point as a geometric translation `x+g*!y`.

301. alternating sum series

```
  x:1+!10
  x
1 2 3 4 5 6 7 8 9 10
  a:((#x)#1 -1)
  a
1 -1 1 -1 1 -1 1 -1 1 -1
  b:x*a
  +\b
1 -1 2 -2 3 -3 4 -4 5 -5
  as:{+\x*(#x)#1 -1}
  as[x]
```

```
1 -1 2 -2 3 -3 4 -4 5 -5
```

This is calculating the alternate sum by multiplying by a series of `(1 -1)` of the same length and then calculating the running sum.

369. alternating sum

```
  x:1+!10
  x
1 2 3 4 5 6 7 8 9 10
  a:((#x)#1 -1)
  a
1 -1 1 -1 1 -1 1 -1 1 -1
  b:x*a
  +/b
-5
  +/x*(#x)#1 -1
-5
```

This idiom is alternating addition and subtraction by generating a `1 -1` vector of the same length, appying an element-wise multiplication and then adding the elements of the resulting vector. Compare to idiom 301 (alternating sum series) and idiom 367 (alternating product).

367. alternating product

```
  x:1 2 3 4 5
  a:(#x)#1 -1
  a
1 -1 1 -1 1
  */x^a
1.875
```

This idiom is alternating multiplication and division by generating a `1 -1` vector of the same length, appying an element-wise exponentiation and then multiplying the elements of the resulting vector.

558. consecutive integers from x to y

```
  ci:{x+!1+y-x}
  ci[5;10]
5 6 7 8 9 10
```

We can calculate the consecutive integers in a range (that includes both ends) by calculating the count of items `1+y-x`, enumerating it, and applying a geometric translation `x+!1+y-x`.

164. divisors

```
  x:363
  dv:{&0=x!/:!1+x}
  dv[x]
1 3 11 33 121 363
  dv 365
1 5 73 365
  dv 367
1 367
```

```
  dv'[1 2 3 4 5 6 7 8 9 10]
(,1
 1 2
 1 3
 1 2 4
 1 5
 1 2 3 6
 1 7
 1 2 4 8
 1 3 9
 1 2 5 10)
```

We can obtain all the integer divisors for a given integer input by generating a vector of all numbers between 1 and the input and then finding all the items where the remainder of dividing the input by the corresponding entry in the generated vector is zero. This technique is not optimized, but it may be acceptable if the inputs are relatively small integers, otherwise we should apply some optimizations One such possible optimization would be to add the results of the division to the result as we find divisors, and reduce the size of the "try to divide by" vector as we process each, e.g., for 363 after finding that 1 is a divisor we add 363 to the result and truncate the "try" vector to 181 (`_ 363%1+1`), after finding that 2 is not a divisor we truncate the "try" vector to 121 (`_ 363%1+2`), after finding that 3 is a divisor we add 121 to the result and truncate the "try" vector to 90 (`_ 363%1+3`), and so forth.

47. polynomial with roots x

```
  x:1 -6 8
  {(x,0)-y*0,x}/1,x
1 -3 -46 48
  x:2 4
  {(x,0)-y*0,x}/1,x
1 -6 8
  x:1 2
  {(x,0)-y*0,x}/1,x
1 -3 2
  x:1 2 3
  {(x,0)-y*0,x}/1,x
1 -6 11 -6
```

Successive multiplication of binomials corresponding to each root results in the desired polynomial. We need to multiply over the subsequent binomials. Let's call the previous polynomial `p` and the root `r` (note: we're temporarily changing the terminology during this explanation so that it doesn't get confused with the mathematical `y` and `x` in `y=f[x]`).

The indices for the corresponding binomial `x-r` are `(1,-r)`. Multiplying `p` by `x-r` results in `px-pr`, which we can express as `(p,0)` (that is, the indices of `p` with an appended zero which corresponds to shifting one position to the left to express multiplication by `x`), minus `r*(0,p)` (that is, the indices of `p` multiplied by `r` and padded with one zero on the left so that all polynomials have the same length).

If we adjust the terminology back to K's automatic `x`, `y` variables `{[p,r](p,0)-r*(0,p)}` gets simplified to `{(x,0)-y*0,x}`. Note that the padding on the left can be done after the multiplication `{(x,0)-0,y*x}/1,x` which may be clearer.

67. Extrapolated value of abscissa x and ordinate y at g

```
  x:-1 0 1
  y:1 0 1.0 / y ~ x^2
  g:10
  |!#x
2 1 0
  x^/:|!#x
(1 0 1.0
 -1 0 1.0
 1 1 1.0)
  y _lsq x^/:|!#x
1 0 4.440892e-016
  g _sv y _lsq x^/:|!#x
100.0
  g:5
  g _sv y _lsq x^/:|!#x
25.0
```

The more general interpretation of `_sv` (that includes not just integers but also float arguments) is that of evaluation of a polynomial with the given coefficients. If we take a function and apply the least square methods to adjust a polynomial to it, we can use `_sv` to evaluate the result at a given point. In this example, we are adjusting values of `y` that have been calculated from $y=x^2$ and we are adjusting a 2nd degree polynomial, and the resulting polynomial is a near perfect match, $y=x^2+0.000000000000000444$, evaluating it at 5.0 and 10.0 yields the expected results of 25.0 and 100.0

69. value of descending polynomial coefficients y at points x

```
  x:-1 0 2
  y: 4 0 5 1
  x _sv\: y
-8 1 43
```

We can apply `_sv` to polinomial coefficients and each value of a vector to obtain the value of the polynomial at the different points. The polynomial in the example is $4x^3+5x+1$.

126. g-degree polynomial fit of points (x,y)

```
  x:64 70 77 82 92 107 125 143 165 189
  y:(5*x^3)+(-1*x^2)+(4*x)+182
  y
1307062 1710562 2277226 2750626 3885526 6114376 9750682 1.460134e+007
2.243424e+007 3.372156e+007
  g:3
  | _lsq[y;x^/:!g+1]
5 -1 4 182.0
  |(y _lsq x^/:!g+1)
5 -1 4 182.0
```

`x^/:!g+1` is a matrix with the values of `x` to the n-th power (from n=0 to the g-degree). Both postfix and infix notations are presented, and the coefficients are reversed to produce a result with coefficients of higher powers first.

363. solve quadratic

```
  qu:{(q%x),(z%q:qq[x;y;z])}
  qq:{-0.5*y+sg[y]*ds[x;y;z]}
  ds:{_sqrt[(y*y)-(4*x*z)]}
  sg:{(x>0)-(x<0)}
  a:1
  b:-1e30
  c:1
  sg[b]
-1
  ds[a;b;c]
1e+030
  qq[a;b;c]
1e+030
  qu[a;b;c]
1e+030 1e-030
  qu[1;-8;15]
5 3.0
```

sg is calculating the sign of an item (1 if positive, -1 if negative). `ds` is calculating the discriminant. The idiom as presented combines the traditional formula with the solution based on Muller's method. The more traditional solution would be `qut:{0.5*(%x)*(-y)+(1 -1)*ds[x;y;z]}`.

430. polynomial derivative

```
  x:1 2 3 4 5
  -1 _ x*|!#x
4 6 6 4
```

Each of the coefficients for the derivative of a polynomial is the original coefficient times the exponent corresponding to the coefficient (the reversed index, if the polynomial is expressed with highest power first as in the example), and with the coefficient that corresponds to the 0th power removed.

137. Taylor series with coefficients y at point x

```
  x:3
  y:1 1 1
  a:!#y
  fac:{[n]:[n>1; n * _f[n-1];1]}
  +/y*(x^a)%fac'[a]
8.5
  y:30#1
  x:1
  a:!#y
  +/y*(x^a)%fac'[a]
2.7182818308537429
```

This is the Maclaurin series formula (Taylor series at point 0) expressed as vector calculations and using a factorial function. The coefficients `y` correspond to the value of the n-th derivative at 0. The first example matches f(x)=½ x²+x+1, f′(x)=x+1, f″(x)=1, which produces f(0)=1, f′(0)=1, f″(0)=1 and f(3)=8.5. The second example is a 30th-degree polynomial approximation to natural exponentiation, which results in Euler's number when evaluated at 1. See idiom 281 for an alternate formulation.

281. value of Taylor series with coefficients y at x

```
  x:12
  y:7 5 6 6
  1+!-1+#y
1 2 3
  1.0,x%1+!-1+#y
1 12 6 4.0
  *\1.0,x%1+!-1+#y
1 12 72 288.0
  y**\1.0,x%1+!-1+#y
7 60 432 1728.0
  +/y**\1.0,x%1+!-1+#y
2227.0
```

The Taylor (Maclaurin) formula is a sum of coefficients of index n divided by the factorial of n and multiplied by the point of application to the power of n. This is an alternate formulation of idiom 137 where factorial calculation is embedded rather than defined as a function. `1+!-1+#y` are the indices for the provided components (with the 0th index removed), `1.0,x%1+!-1+#y` is the division of the point of application by the indices (prepended by 1.0, for the 0th index). `*\1.0,x%1+!-1+#y` is applying the factorial by successive multiplication. `y**\1.0,x%1+!-1+#y` are the terms of the series. `+/y**\1.0,x%1+!-1+#y` calculates the final result by adding up all the terms.

48. saddle point indices

```
  x:(4 2 4 4 2 4
> 5 3 5 5 3 5
> 4 2 4 4 2 4
> 1 2 4 4 2 4
> 5 3 5 5 3 5
> 4 2 4 4 2 4)
```

48a. row minimum

```
  rn:{x='&/'x}
  rn x
(0 1 0 0 1 0
 0 1 0 0 1 0
 0 1 0 0 1 0
 1 0 0 0 0 0
 0 1 0 0 1 0
 0 1 0 0 1 0)
```

This part of the idiom is calculating the minimum over each row and then comparing to each item in the row, which results in a 0/1 matrix with ones at the row minimums.

48b. column maximum

```
  cx:{x=\:|/x}
  cx x
(0  0  0  0  0  0
 1  1  1  1  1  1
 0  0  0  0  0  0
 0  0  0  0  0  0
 1  1  1  1  1  1
 0  0  0  0  0  0)
```

This part of the idiom is calculating the maximum of each column and then comparing the vector of max to each item in the row, which results in a 0/1 matrix with ones at the column maximums. It is interesting to note how these verbs operate element-wise when the arguments are vectors (we would have to enlist the arguments if we want to operate on the full vector itself e.g. `(,x)=,y` to compare if x and y are equal vectors).

48c. minmax of rows and columns

```
  minmax:{(rn x)&(cx x)}
  minmax x
(0  0  0  0  0  0
 0  1  0  0  1  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  1  0  0  1  0
 0  0  0  0  0  0)
```

This is combining the two previous results, to return a 0/1 matrix with ones denoting the elements that are both min for the row and max for the column.

48d. locate 1s in ravel of Boolean matrix

```
  ones:{&,/minmax x}
  ones x
7 10 25 28
```

In order to convert the minmax 0/1 matrix to coordinates, we first determine the linear indices `x+(row_length*y)` by joining the rows and finding the ones

48e. saddle point indices

```
  sp:{(^x) _vs ones x}
  sp x
(1  1  4  4
 1  4  1  4)
```

Finally we apply a vector from scalar (base) conversion using the shape of the original matrix to get the vectors of `x` and `y` coordinates, or we can transpose the results if we prefer a vector of `(x;y)`'s.

262. value of saddle point (see 48)

```
  x:(5  4  6  4  12  5
>  16  2  4  5  16  18
> 8 18 7 12 16 11
> 20 17 16 14 16 20
> 16 8 12 9 17 13)
  rn:{x='&/'x}
  cx:{x=\:|/x}
  minmax:{(rn x)&(cx x)}
  minmax x
(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)
  ones:{&,/minmax x}
  ones x
,21
  (,/x)[ones[x]]
,14
```

This is an application of idiom 48 where at the end instead of outputting the resulting saddle point matrix we take the location of ones in the ravel of Boolean matrix and use them to index the raveled input.

Ranges

This section presents various operations related to ranges of values

159. is range of x 1 (are all items of x equal)

```
  x:1 1 1 1 1
  1=#?x
1
  y:1 1 0 1 1
  1=#?y
0
```

Is the count of unique items equal to 1?

180. is x in range [y)

```
  hc:{1 0~/:~x<\:y}
  x:19 20 21 39 40 41
  y:20 40
  ~x<\:y
(0 0
 1 0
 1 0
 1 0
 1 1
 1 1)
  hc[x;y]
0 1 1 1 0 0
```

This idiom is comparing `x` to both values in the range `y`. If the result is `1 0` then the number is in the range. If we reverse the vector we're comparing, we can eliminate one negation operation `hc:{0 1~/:x<\:y}`.

233. is x within range [y)

```
  x:9
  y:(1 9
> 9 16
> 5 7
> 8 20
> 6 10)
x<y
(0 0
 0 1
 0 0
 0 1
 0 1)
  </'x<y
0 1 0 1 1
  x(</<)/:y
0 1 0 1 1
```

This is applying a less than `<` operator to a scalar and a matrix (comparing each element in the matrix to the scalar) and then checking if each result is `0 1` by applying a `</` to each row. An alternative infix notation is also shown (the each needs to be converted to each-right).

234. is x within the range (y]

```
  x:9
  y:(1 9
> 9 16
> 5 7
> 8 20
> 6 10)
  ~x>y
(0 1
 1 1
 0 0
 0 1
 0 1)
  </'~x>y
1 0 0 1 1
```

This is applying a not greater than comparison to a scalar and a matrix (comparing each element in the matrix to the scalar) and then checking if each result is `0 1` by applying a `</` to each row, similar to the previous idiom 233.

221. is x an integer in interval [g,h)

```
  g:6
  h:12
  x:18
  (x=_ x)&(~x<g)&(x<h)
0
  x:7
  (x=_ x)&(~x<g)&(x<h)
1
  x:7.1
  (x=_ x)&(~x<g)&(x<h)
0
```

The min `&` operation when applied to 0/1 integers acts as a Boolean and. We can calculate interval membership by combining comparisons. We can check if the number is an integer by comparing its integral part to itself (we could also have used a `4:` type operation). The parentheses in the rightmost expression is unnecessary, `(0=4:x))&(~x<g)&x<h`.

312. maximum separation of items of x

```
  x:10+7 _draw 9
  x
17 14 14 17 14 17 18
  (|/x)-(&/x)
4
```

The maximum separation is the difference between the max and min. We can simplify the expression by removing unneeded parenthesis `(|/x)-&/x`.

329. mask from positive integers in x

```
  x:-5+7 _draw 10
  x
2 3 3 -2 4 4 -1
  (!1+|/x) _lin x
0 0 1 1 1
```

The goal is finding which positive integers are present in the input and which are missing. This is first establishing the range (generating an enumeration up to the max `x`) and using `_lin` to determine which ones are actually present in `x`. This idiom assumes that there is at least one positive integer in `x`. For a more generic approach we can use `@[&1+|/x;x@&x>0;:;1]`.

345. do ranges of x and y match

```
  x:1 2 3
  y:3 2 1 1
  a:?x
  x
1 2 3
  a
1 2 3
  b:?y
  b
3 2 1
  a[a[<a]~b[<b]
1
  om:{x[<x]~y[<y]}
  om[?x;?y]
```

```
1
  om[?"bca";?"cabba"]
1
```

A way of finding if the ranges are the same is to compare if the sorted uniques match. Another way to think about the problem is to use `_lin` to check if all the items are present in the other `&/(x _lin y),y _lin x`.

350. is x 1s and 0s only (Boolean)

```
  x:0 1 0 1
  &/x _lin\:0 1
1
  x:1 1 1 1
  &/x _lin\:0 1
1
  x:1 0 1 2
  &/x _lin\:0 1
0
```

This is similar to a range check (see comments in idiom 345) where we take advantage of knowing one of the vectors beforehand. The `\:` is not needed if x is a vector, `&/x _lin 0 1`. Note that one difference with the range check is that a vector of all zeros or all ones should still return a `1`. An alternative calculation using comparisons could be `(x~x>0)&x~x<1`. An even simpler alternative is to match a double negation `x~~~x` (taking advantage of the double negation converting zero/nonzero to 0/1).

353. are items unique

```
  x:"abcdefg"
  (#x)=(#?x)
1
  x:"abcdefa"
(#x)=(#?x)
  0
```

A simple way to check if there are no duplicates is to compare the cardinality of the unique items in the vector to the cardinality of the vector. The second set of parentheses is not needed, `(#x)=#?x`. An even shorter alternative could be comparing the vector to its uniques `x~?x`, but that is probably less efficient.

366. count of scalars

```
  cs:{#,//x}
  cs[1]
1
  cs[1 2]
2
  cs[(1 2;3 4 5)]
5
  cs[(1 2;(3 4;5))]
5
  cs[("ab";("cd";"efg"))]
```

```
7
  cs[!0]
0
```

This is counting the results of repeatedly flattening a list until until it converges (`,/` is a monadic operation and `monadic/` is converge).

548. test if empty

```
  ie:{0 _in ^x}
  ie 7
0
  ie 8 9
0
  ie 2 3#6
0
  ie (1 0#5)
1
```

Emptyness in a vector, matrix or tensor can be defined by having one or more of its dimensions empty (e.g., see idiom 513, empty matrix). This idiom applies the definition by checking if 0 is present in the shape `^x`.

564. is x within range (y[0],y[1])

```
  ci:{(y[0]<x)&(x<y[1])}
  y:3 8
  x:5
  ci[x;y]
1
  x:3
  ci[x;y]
0
  x:2
  ci[x;y]
0
  x:8
  ci[x;y]
0
  x:9
  ci[x;y]
0
```

We can determine if a value is contained in a range (with ends of the range excluded) by using two `<` operations against the ends and using an "and" operation (the min `&` operation behaves as a boolean "and" when the operands are 0/1).

565. is x within range [y[0],y[1]]

```
  oi:{(~y[0]>x)&(~x>y[1])}
  y:3 8
  x:5
  oi[x;y]
1
```

```
  x:3
  oi[x;y]
1
  x:2
  oi[x;y]
0
  x:8
  oi[x;y]
1
  x:9
  oi[x;y]
0
```

This is a variation on the previous idiom, changing the `a<b` comparisons to `~a>b`, which can be interpreted as "not greater than" or "less than or equal". With this change, we are determining if the value is contained in the range, with the range ends included.

Statistics

This section presents how to apply K to descriptive statistics and statistical estimation.

325. average (mean)

```
  av:{(+/x)%#x}
  av[1 10 100]
37.0
```

This is an application of the formula $\frac{1}{n}\sum_0^n x_i$

237. average (mean) of x weighted by y

```
  y:78 80 90 88 72
  x:20 15 20 22 19
  x*y
1560 1200 1800 1936 1368
  +/x*y
7864
  (+/x*y)%#x
1572.8
```

This is a variation on weighted values (see idiom 222) applying a sum over the weighted value and then dividing by the count of items.

24. median

```
  x:11 _draw 100
  x
61 20 51 12 34 31 51 29 35 17 89
  x[<x]
12 17 20 29 31 34 35 51 51 61 89
  x[(<x)[_.5*#x]]
34
```

Get all the indices for sorting, then take the item with an index at the center of the list.

319. standard deviation

```
   x:44 77 48 24 28 36 17 49 90 91
  std:{((+/(x-(+/x)%#x)^2)%#x)^0.5}
  std[x]
25.48411
```

This is a straightforward application of the formula $\sqrt{\frac{1}{n}\sum_0^n\left(x_i-\frac{1}{n}\sum_0^n x_i\right)^2}$

320. variance (dispersion)

```
  x:44 77 48 24 28 36 17 49 90 91.0
  var:{(+/(x-(+/x)%#x)^2)%#x}
  var[x]
649.44
  (var[x])^0.5
25.48411
```

This is an application of the formula $\frac{1}{n}\sum_0^n\left(x_i-\frac{1}{n}\sum_0^n x_i\right)^2$

321. y-th moment of x

```
  x:44 77 48 24 28 36 17 49
  ym:{(+/(x-(+/x)%#x)^y)%#x}
  ym[x;2]
309.23
  ym[x;0]
1.0
  ym[x;1]
4.4409e-016
  ym[x;3]
3889.9
```

This is an application of the formula $\frac{1}{n}\sum_0^n\left(x_i-\frac{1}{n}\sum_0^n x_i\right)^y$

128. coefficients of best linear fit of points (x,y) (least squares)

```
  x:64 70 77 82 92 107 125 143 165 189
  y:56 60 66 70 78 88 102 118 136 155
  z:_lsq[y;x^/:0 1]
  z
4.587803 0.7927486
  z[0]+z[1]*x / should be y, approximately
55.32371 60.08021 65.62945 69.59319 77.52068 89.41191 103.6814
117.9509 135.3913 154.4173
```

`x^/:0 1` produces the value of `x` to the 0th and 1st power (for linear fit), then we apply `_lsq`.

125. predicted values of best linear fit (least squares)

```
  x:64 70 77 82 92 107 125 143 165 189
  y:56 60 66 70 78 88 102 118 136 155
  a:x^/:0 1
  a
(1 1 1 1 1 1 1 1 1 1.0
 64 70 77 82 92 107 125 143 165 189.0)
  _mul[+a;_lsq[y;a]]
55.32371 60.08021 65.62945 69.59319 77.52068 89.41191 103.6814
117.9509 135.3913 154.4173
/ This can also be written in infix form
  (+a) _mul (y _lsq a)
55.32371 60.08021 65.62945 69.59319 77.52068 89.41191 103.6814
117.9509 135.3913 154.4173
```

`a` is the value of `x` to the 0th and 1st power (for linear fit), then we calculate a linear fit by direct application of `_lsq` and finally evaluate the result at a given point by applying `_mul`.

127. coefficients of exponential fit of points (x,y)

```
  x:64 70 77 82 92 107 125 143 165 189
  y:56 60 66 70 78 88 102 118 136 155
  a: _lsq[_log[y];x^/:0 1]
  a
3.563398 0.00817742
  a[0]: _exp[a[0]]
  a
35.2829 0.00817742
  a[0]*_exp[x*a[1]]
59.54624 62.54071 66.22511 68.98898 74.86758 84.63791 98.05964
113.6098 136.0024 165.4933
  y
56 60 66 70 78 88 102 118 136 155.0
```

This is a simpler version of idiom 124.

124. predicted values of exponential fit

```
  x:64 70 77 82 92
  y:56 60 66 70 78
  a:x^/:0 1
  a
(1 1 1 1 1.0
 64 70 77 82 92.0)
  _log[y]
4 4.1 4.2 4.2 4.4
  _lsq[_log[y];a]
3.3 0.012
  _exp[_mul[+a;_lsq[_log[y];a]]]
56 60 66 70 78.0
```

`a` is the value of `x` to the 0th and 1st power (for linear fit), then we combine `_log` and `_lsq` to obtain the coefficients for an exponential fit, and finally we use `_exp` and `_mul` for evaluating the resulting fit at given points.

173. assign x to one of y classes of width h, starting with g

```
  f:{[x;y;g;h] -1+ -1 _ #:'=(1+!y),-_-(x-g)%h}
  x:32 56 36 48 36 24 28 44 52 32
  y:4
  h:10
  g:10
  f[x;y;g;h]
0 2 4 2
```

`(x-g)%h` applies a translation `g` and scaling `h`. `-_-` is rounding up any fractional part. `(1+!y),` prepends one entry for each class to ensures that all the classes will exist, even if the input would produce empty classes. `#:'=` counts the number of items in each group. `-1 _` removes the last class (note that this only works if the input fits between `0` and `y+1` classes, if there are inputs greater than class `y+1` it will produce incorrect results) and `-1 +` removes the “extra” entry added by `1+!y`. We can optimize the operations slightly by removing the extra classes before counting, and fix the potential issue of extra classes by using `y#` instead of `-1 _`

```
  f:{[x;y;g;h] -1+#:'y#=(1+!y),-_-(x-g)%h}
  x:72 32 56 36 48 36 24 28 44 52 32
  y:4
  h:10
  g:10
  f[x;y;g;h]
0 2 4 2
```

201. moving index y-wide for x

```
  x:"abcdef"
  y:3
  (!(#x)-y-1)+\:!y
3 4 5 6
```

A moving index should be an index we could use for calculating moving averages if applied to a list of numbers.

```
  x:26 40 39 28 27 48
  y:3
  (!(#x)-y-1)+\:!y
(0 1 2
 1 2 3
 2 3 4
 3 4 5)
```

Which is generating a y-wide index and then adding to it a range of the length of x minus the width of the moving index, adjusted to be 0-based. Applying it to calculating moving averages would result in:

```
  mi:{(!(#x)-y-1)+\:!y}
  x:26 40 39 28 27 48
  y:3
  mi:{(!(#x)-y-1)+\:!y}
  x@mi[x;y]
(26 40 39
 40 39 28
 39 28 27
 28 27 48)
```

```
  +/'x@mi[x;y]
105 107 94 103
  (%y)*+/'x@mi[x;y]
35 35.66667 31.33333 34.33333
```

546. is count of atoms 1 (uses cs from 366)

```
  cs:{#,//x}
  co:{1=cs[x]}
  co[35]
1
  co[,35]
1
  co[1 1#35]
1
  co[1 1 1#35]
1
  co[1 2]
0
  co[!0]
0```
```

This is a simple combination of `1=` with idiom 366 count of scalars.

Application of financial formulas

This section presents how to apply K to selected financial problems

77. present value of cash flows c at times t and discount rate d

```
/ Example: a 3-year bond with an annual 10% coupon
/ and discount rate of 0.9
  pv:{[c;t;d]+/c*d^t}
  c:0.1 0.1 1.1
  t:1 2 3
  d:0.9
  pv[c;t;d]
0.9729
/ It is better to use current prices to derive a discount function
pv:{[c;t;d]+/c*d[t]}
  pv[c;t;{[t] 0.9^t}]
0.9729
/ It is even more efficient to use the discount values directly
  d: 0.9^0 1 2 3
  d
1 0.9 0.81 0.729
  pv[c;t;d]
0.9729
```

These are direct applications of the formula, showing how it gets applied to vector elements.

82. future value of cash flows x at interest rate y

```
  x:10 15 20 25
  y:5
```

```
  +/x*(1+y%100)^|!#x
74.11375
  fv:{+/x*(1+y%100)^|!#x}
  fv[x;y]
74.11375
```

Application of the future value formula, as vector operations. The exponents are a decreasing series of periods to maturity (since the first amount stays deposited for a longer period of time). The assumption is that the periods match the rate period, e.g. deposits are done annually, and `y` is the annual percentage rate.

146. compound interest for principals y at percentages g for periods x

```
  x:1 2 3 4
  y:1 2 3 4
  g:0.5 1 1.5 2
  z:y*\:(1+g%100)^\:x
  \p 5
  z
((1.005 1.01 1.0151 1.0202
  1.01 1.0201 1.0303 1.0406
  1.015 1.0302 1.0457 1.0614
  1.02 1.0404 1.0612 1.0824)
 (2.01 2.02 2.0302 2.0403
  2.02 2.0402 2.0606 2.0812
  2.03 2.0604 2.0914 2.1227
  2.04 2.0808 2.1224 2.1649)
 (3.015 3.0301 3.0452 3.0605
  3.03 3.0603 3.0909 3.1218
  3.045 3.0907 3.137 3.1841
  3.06 3.1212 3.1836 3.2473)
 (4.02 4.0401 4.0603 4.0806
  4.04 4.0804 4.1212 4.1624
  4.06 4.1209 4.1827 4.2455
  4.08 4.1616 4.2448 4.3297))
```

This is the vector expression of the compound interest formula.

186. annuity coefficient for x periods at interest rate y%

```
  x:10 15 20 25
  y:8 9 10 15
  +1-(1+y%100)^\:-x
(0.537 0.578 0.614 0.753
 0.685 0.725 0.761 0.877
 0.785 0.822 0.851 0.939
 0.854 0.884 0.908 0.97)
  ac:{(y%100)%/:+1-(1+y%100)^\:-x}
  ac[x;y]
(0.149 0.156 0.163 0.199
 0.117 0.124 0.131 0.171
 0.102 0.11 0.117 0.16
 0.0937 0.102 0.11 0.155)
```

This is a vectorized implementation of the formula for annuity coefficients 1-(1+y÷100)^x

286. FIFO stock y decremented with x units

```
  x:5
  y:1 2 3 4 5
  (+\y)-x
-4 -2 1 5 10
  g:0|(+\y)-x
  g
0 0 1 5 10
  -':0,g
0 0 1 4 5
  ff:{-':0,0|(+\y)-x}
  ff[x;y]
0 0 1 4 5
```

This is simulating the changes in inventory of time-associated items after removing items using a First-in-first-out strategy (a typical application is for accounting and tax purposes of partial sales of stocks grouped and sorted by purchase date, but it could also be applied to other scenarios like perishable items on a retail shelf grouped and sorted by expiry date). If we calculate the running sum `+\y` and subtract the decrement `x` we will get the adjusted cumulative inventory levels. One caveat is that some values could be negative, and need to be adjusted to zero with `0|`. To convert the cumulative inventory back to individual inventories we prepend a zero and subtract each pair `-':0,` which is the reverse of `+\`.

Full problems

This section presents the solution in K to various interesting problems

389. Playing order of x ranked players

```
  log2:{_log[x]%_log[2]}
  tt:{2 _vs !_2^x}
  i:{1+2 _sv'+|tt[-_-log2[x]]}
  j:{@[x;&x>y;:;0]}
  k:{j[i[x];x]}
  x:6
  i[x]
1 5 3 7 2 6 4 8
  j[i[x];x]
1 5 3 0 2 6 4 0
  k[x]
1 5 3 0 2 6 4 0
```

This is calculating pairings for an initial phase of a single-elimination tournament (also known as knockout tournament), seeded by rank. `log2` is the base 2 logarithm, and `tt` is the truth table of order `x` (see idiom 60). `i` is the algorithm for calculating the pairings of players according to their seeding rank. `tt[-_-log2[x]]` is a matrix whose columns represent a binary count up that includes all the seeding ranks. Using base 2 powers/logs will allow successive "knockout" rounds to "fill" all the matches, but that may produce for the first round matches for which not enough players are available. The resulting binary count is transposed and mirrored then converted back from binary to decimal and adjusted to be 1-based instead of 0-based to generate the pairings. `j` is setting to zero all items in `x` above a threshold `y` and `k` is using `j` to zero the results from `i[x]`, to remove those matches for which not enough players are available

K 3.2 Quick reference

Here is a copy of the reference documentation for K3.2, for ease of reference offline.

Data:

```
        Scalar  Vector  Empty   Inf     Null
integer 0       1 2     !0      0I      0N
float   0.0     1 2.0   0#0.0   0i      0n
char    " "     "12"    ""              "\0"
symbol  `       `a`b    0#`

lambda  {}
null    _n (_n@i and _n?i are i; _n`v is _n)
list    (x;y;z)  () is empty  ,... is list of one

dict    .((symbol;value;attributes);...)
 !d             symbols
 d[string]      execute
 d[`v]  d[]     values
 d[`v.] d[.]    attributes
 d.v  is  d`v  is  d[`v]  is  d@`v  is  d .,`v
 (also `d)

~`v     attribute handle (`v.)
4:x     type: atom(1 to 7)[ifcsdnx] list(0 to -4)[KIFCS]
5:x     ascii representation
```

Verbs:

```
        Dyad            Monad
+       plus            flip
-       minus           negate
*       times           first
%       divide          reciprocal
&       min/and         where
|       max/or          reverse
<       less            upgrade
>       more            downgrade
=       equal           group
^       power           shape
!       mod/rotate      enumerate
~       match           not
,       join            enlist
#       take/reshape    count
_       drop/cut        floor
$       form/format     format
?       find/invert     unique  invert-guess
```

```
@       at              atom    m-amend d-amend
.       dot             value   m-amend d-amend

f[x]    is f .,x        is f@x  is f x
f[x;y]  is f .(x;y)
```

Adverbs:

```
f'      EACH
d\:     EACHLEFT
d/:     EACHRIGHT
d':     EACHPRIOR
d/      OVER    f/[init;...]    RECUR
d\      SCAN    f\[init;...]    TRACE
m/ CONVERGE     n m/ DO         b m/ WHILE
m\ CONVERGE     n m\ DO         b m\ WHILE
f(function) d(dyadic) m(monadic) b(boolean)
scatter selection       matrix'
transitive closure      vector/ vector\
state transition        matrix/ matrix\ matrix':

compose         2+  8$  ~=  ~<  ~>  *|:(first reverse)
project         pv:{[c;t;d]+/c*d^t}  pv[c:.1 .1 1.1;t:1 2 3]d:%1.1
invert          pv[c;t]?.97  pv[c;;d]?.98  pv[;t;d]?.99
modify          @[0 0 0;1 1 0 2 2 0 1 2;+;!8]
iterate         9{+':0,x,0}\1  9(|+\)\1 1.0
converge        (1+%:)\1.0  {x\'!#x}parent:0 0 1 1 0
integrate       +\2 3 4  *\2 3 4
differentiate   -':0 2 5 9  %':1 2 6 24

verbs default to dyad. use(:) for monad, e.g. (<;<:)
```

System verbs and nouns:

```
Math:   _log _exp _abs _sqr _sqrt _floor _dot _mul _inv
        _sin _cos _tan _asin _acos _atan _sinh _cosh _tanh
        y _lsq A is least squares x for y~+/A*x (i.e. Ax=y)

Rand:   x _draw y (from !y); x _draw -y (deal from !y); x _draw 0
(from (0,1))

Time:   _t is gmt seconds. _lt is local from gmt, e.g. _gtime _lt _t
        _jd yyyymmdd (and _dj) for to and from julian day number (0 is
monday)

List:   x _in y is 1 if x is an item of y; 0 otherwise (list: _lin)
        x _bin y is binary search for y in ascending x (list: _binl)
        x _dv y and x _di y to delete by value and index (list: _dvl)
        x _sv v (scalar from vector) and x _vs s (vector from scalar)
        _ci i (character from integer) and _ic c (integer from
character)
        x _sm y is string match. y can have *?[^-], e.g. files _sm
"*.[kK]"
```

```
        x _ss y is string/symbol search for start indices. y can have
?[^-].
        _ssr[x;y;z] is string/symbol search and replace. z can be a
function.
        _bd d (bytes from data) and _db b (data from bytes). linear
form.
        _getenv v (v _setenv s) gets(sets) environment variable v.
        _host addr; _host name; _size file; _exit code.

Vars:   _d(dir) _v(var) _i(index) _t(second) _f(function) _n(null)
_s(space)
        _h(host) _p(port) _w(who) _u(user) _a(address) _k(version)
_T(time)
```

Assign, define, control and debug:

```
Dyad            D-Amend         Monad           M-amend
v::y (or v:y)   .[`v;();:;y]
v+:y            .[`v;();+;y]    v-:             .[`v;();-:]
v[i]+:y         .[`v;,i;+;y]    v[i]-:          .[`v;,i;-:]
v[i;j]+:y       .[`v;(i;j);+;y] v[i;j]-:        .[`v;(i;j);-:]

@[v;i;d;y] is .[v;,i;d;y]       @[v;i;m] is .[v;,i;m]

{[a;b;c] ...}   function definition
 x y z          default parameters
 d:...          local variable

control(debug: ctrl-c stop)
 :[c;t;f]       conditional
 if[c; ... ]
 do[n; ... ]
 while[c; ...]
 / ...          comment
 \ ...          trace(escape)
 : ...          return(resume)
 ' ...          signal

trap signals with .[f;(x;y;z);:] and @[f;x;:]
```

I/O, dynamic load and client/server:

```
 !directory      list files(!"" root !"." current)
 0:f    f 0:x   read/write text(` for console)
 1:f    f 1:x   read/write data(default .l)
 6:f    f 6:x   read/write bytes
        f 5:x   append data, e.g. `log 5:,transaction

 (type;[,]delim)0:f     [names+]delimited text( IFCSDTZ)
 (type;width)0:f        fixedwidth text( IFCSDTZ)
 (type;width)1:f        fixedwidth data(cbsijfd IFCSDZMm)
 Blank skips. S strips. f can be (f;index;length).
```

```
client/server, k f -i 2001     Callback(default)
 w:3:(m;port)  open            .m.u users(all)
 w 3:msg       set/asynch      .m.s function(.:)
 w 4:msg       get/synch       .m.g function(.:)
 3:w           close           .m.c expression("")

 .m.s .m.g and .m.c can access _w, _u(trunc to 8) and _a.
 m is `machine(localhost) or "ddd.ddd.ddd.ddd" or _a.
 default msg v, (v;i), (v;i;u) or (v;i;u;d). v can be string.
 can't write or message _fn, \kr fn or fn[;x]

[f]2:(entry;argcount)  dynamic load
```

OS commands:

```
`3:string                 os set command, e.g. `3:"k bck"
`4:string                 os get command, e.g. `4:"dir"
```