All-Permutations and other interesting matrices in K

In the K idioms list, #197 describes how to obtain the identity matrix of order x

```
id:{(!x)=\:!x} // identity matrix
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)
```

This operation simply compares the enumeration of x to each of the elements of the enumeration (and, as a side note, it is symmetrical, we can replace `=\:` with `=/:` and obtain the same results). We can replace equality with other comparisons or even non comparison operations and generate other interesting matrices

Upper and lower triangular matrices (excluding the diagonal) can be generated using less-than and greater-than comparisons

```
  triup: {(!x)<\:!x}
  triup 4
(0 1 1 1
 0 0 1 1
 0 0 0 1
 0 0 0 0)
  trilow: {(!x)>\:!x}
  trilow 4
(0 0 0 0
 1 0 0 0
 1 1 0 0
 1 1 1 0)
```

These comparisons can be logically inverted if we want to generate matrices that include the diagonal elements.

```
  triupd: {~(!x)>\:!x}
  triupd 4
(1 1 1 1
 0 1 1 1
 0 0 1 1
 0 0 0 1)
  trilowd: {~(!x)<\:!x}
  trilowd 4
(1 0 0 0
 1 1 0 0
 1 1 1 0
 1 1 1 1)
```

A matrix with a forward-slash diagonal and its parallels can be generated using sum (0 to 2x-2) or modulo (0 to x-1, periodically)

```
  d1:{(!x)+\:!x}
  d1 4
(0 1 2 3
 1 2 3 4
 2 3 4 5
 3 4 5 6)
  d2:{(!x)!\:!x}
  d2 4
(0 1 2 3
 1 2 3 0
 2 3 0 1
 3 0 1 2)
```

Of course, we can reverse each row for a vertical mirror of the results, e.g. for a “backslash” diagonal

```
  d3:{|(!x)+\:!x}
  d3 4
(3 4 5 6
 2 3 4 5
 1 2 3 4
 0 1 2 3)
  d4:{|(!x)!\:!x}
  d4 4
(3 0 1 2
 2 3 0 1
 1 2 3 0
 0 1 2 3)
```

We can generate “nested arrowheads” using bitwise-and and bitwise-or

```
  a1:{(!x)|\:!x}
  a1 4
(0 1 2 3
 1 1 2 3
 2 2 2 3
 3 3 3 3)
  a2:{(!x)&\:!x}
  a2 4
(0 0 0 0
 0 1 1 1
 0 1 2 2
 0 1 2 3)
```

We can use the identity matrix to improve on another idiom in the list, #429 converting a vector to a diagonal matrix. The solution in the list is based on appending zeros on each side of each item but, if we use an identity matrix we can just multiply each row by the desired diagonal, which has better performance

```
  id:{(!x)=/:!x}
  dv:{x*'id[#x]} // diagonal from vector
  dv 5 9 6 7 2
(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)
```

Another interesting matrix derived from the identity matrix is the graded identity matrix, which contains the indices necessary for performing a stable sort of an identity matrix. All the elements in this matrix are permutations of `!x`, and they are useful as an intermediate step in generating all possible permutations of `!x`, as described in the next section.

```
  id:{(!x)=/:!x}
  gi:{>:'id x} // graded identity matrix
  gi 4
(0 1 2 3
 1 0 2 3
 2 0 1 3
 3 0 1 2)
```

The all-permutations matrix is a non-square matrix that describes the indices that will produce all possible permutations of a vector. Generating the actual permutations afterwards will require just a simple indexing operation. My implementation is based on a recursive algorithm by Roger Hui, as described by Eugene McDonnell in Vector, Volume 20, Issue 2 (October 2003). The article from Vector is available online at https://code.jsoftware.com/wiki/Doc/Articles/Play202

(note: I wrote my code based on the textual description of the algorithm, not the J code)

It is a recursive algorithm, where the table for size n is based on 2 components. The first one is the graded identity matrix and the second is a seed obtained from adding 1 to the table of size n-1 and prepending a 0 to each row. Indexing each row of the graded identity by the seed will produce a set of tables that will result in the desired all-permutations matrix when concatenated. The base case for the recursion is size 1 which is a row with a single 0

```
  id: {(!x)=/:!x}
  gi:{>:'id x}
  permix:{:[2>x;x#,0;,/gi[x]@\:0,'1+_f[x-1]]} // permutation indices
  permix 3
(0 1 2
 0 2 1
 1 0 2
 1 2 0
 2 0 1
 2 1 0)
```

Note that for the base case I am using `x#,0`. This approach works when x=1 and it also results in an empty list for x=0, which probably makes sense (important: if your application may call this function with a negative input, please implement appropriate handling, as the current code will produce a vector of zeros, which is probably different from what you will need)

The conditional `:[]` is using k2/k3 syntax, for k4 and later you want to use `$[]`

```
  permix: {$[2>x;x#,0;,/gi[x]@\:0,'1+_f[x-1]]} // k4+ version
```

Once we have the indices, we can use them for obtaining all the permutations of any list.

```
  l:("foo";"bar";"baz")
  l@permix[#l]
(("foo"
  "bar"
  "baz")
 ("foo"
  "baz"
  "bar")
 ("bar"
  "foo"
  "baz")
 ("bar"
  "baz"
  "foo")
 ("baz"
  "foo"
  "bar")
 ("baz"
  "bar"
  "foo"))
```

As a last step, if we want to handle lists that can have repeated elements, we need to eliminate duplicates from the result

```
  id: {(!x)=/:!x}
  gi:{>:'id x}
  permix:{:[2>x;x#,0;,/gi[x]@\:0,'1+_f[x-1]]}
  perm:{?x@permix[#x]}
  l2:("foo";"foo";"bar")
  perm[l2]
(("foo"
  "foo"
  "bar")
 ("foo"
  "bar"
  "foo")
 ("bar"
  "foo"
  "foo"))
```

This can be easily converted to q

```
q)id:{til[x]=/:til[x]}
q)gi:{idesc each id[x]}
q)permix:{$[2>x;x#enlist 0;raze gi[x]@\:0,'1+.z.s x-1]}
q)perm:{distinct x permix count x}
q)permix 3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
q)perm ("foo";"bar";"baz")
"foo" "bar" "baz"
"foo" "baz" "bar"
"bar" "foo" "baz"
"bar" "baz" "foo"
"baz" "foo" "bar"
"baz" "bar" "foo"
q)perm ("foo";"foo";"bar")
"foo" "foo" "bar"
"foo" "bar" "foo"
"bar" "foo" "foo"
q)perm "abc"
"abc"
"acb"
"bac"
"bca"
"cab"
"cba"
```

For reference, this algorithm is different from the one presented in the Kx website, which uses rotations. On my machine it performs 5x to 10x faster. E.g., less than a second

```
q)id:{til[x]=/:til[x]}
q)gi:{idesc each id[x]}
q)permix:{$[2>x;x#enlist 0;raze gi[x]@\:0,'1+.z.s x-1]}
q)\t permix 10
832
```

Compared to 7.5 seconds for the code from
https://code.kx.com/q/learn/python/examples/string/#permute-a-string

```
q)p:{$[x=2;(0 1;1 0);raze(til x)(rotate')\:(x-1),'.z.s x-1]}
q)\t p 10
7514
```