Q for smarties [1]
Pune

Professor Dennis Shasha
2011

---

# q for smarties 1

**Introduction** Q is a successor to k which is a successor to A+, an APL successor. Arthur Whitney, the designer and implementor believes in concise, expressive, and efficient languages. In fact, q is implemented on top of k. The executable is under 400 kilobytes.

Some people like to develop directly in the interpreter. I tend to develop programs in files and then execute them.

The basic data types are atoms (single entities, e.g. a single number), lists (a sequence of atoms) which double as arrays, dictionaries, and tables. The language is very powerful: you can do bulk operators on entire arrays and tables and do interprocess communication. If treated well, it can also execute fast.

Operators (sometimes called "verbs") can operate on any data type but in different ways. A good way to get familiar with the language is to type

```
\l help.q
```

in a q window

```
q) help ‘ /this is a backtick not single quote. TBD
q) help ‘verbs /this is a backtick and not single quote. FIXME
```

Let's start off simple:

```

q) x: 4

q) y: 5

q) x+y

q) x * y

q) x: 4 40 400

q) y: 5 55 80

q) x + y

q) x * y
```

Note however that processing goes from right to left rather than through operator precedence. So, for example, what do you anticipate you would get from:

```
5 - 4 * 3

5 * 4 - 3

(5 * 4) - 3
```

```
sum x * y / ability to "compose" functions

x[1]

y[2]

z: ((1 2 3);(4 5 6);(7 8 9 10))

z[1;2]

z[1 2]
```

Some useful monadic (single argument) verbs: [2]

```
til 10
1 + til 10
x: 1 + til 10
y: 3 + til 10
reverse x


v: 3 2 6 4 9 8 6 5 2 3 4
distinct v
first v
min v
max v
sum v
sums v
prd v
prds v
count v
iasc v / permutation that will give sorted array
v[iasc v] / sorted array itself
avg v
```

Please take time tonight to review all the commands in basicmoves.q

```
/ https://code.kx.com/trac/wiki/QforMortals2/primitive_operations
/ https://code.kx.com/trac/wiki/QforMortals2/built_in_functions

/ This file contains basic operations and how to do them in q
/ I strongly recommend that
/ you try these one line at a time in the interpreter.
/ It will help you internalize frequently used operators and adverbs

/ Arithmetic:

5 * 6 / scalar mult

(5 6 7) * 3 / vector * scalar mult

(5 6 7) * (1 2 3) / vector mult -- arguments must be the same length

(5 6 7) % (1 2 3) / vector division -- note that division is %

/ Generating data

til 10 / whole numbers from 0 to 9

5 + til 10 / generate first and then add 5 to each element

10 ? 20 / 10 numbers drawn from 20 at random with replacement

(-10) ? 20 / 10 numbers drawn from 20 at random without replacement

10 ? 06:00:00 / 10 random times over 6 hours

10 ? ‘a‘bbb‘c / 10 random symbols from ‘a ‘bbb ‘c

/ starting at a random seed determined by current time
t: ltime .z.p / current datetime locally
millis: ‘int$t.time / milliseconds
x: (millis mod 767) ? 20
/ Now can use a random number as you like
possiblestocks: ‘ibm‘hp‘aapl‘goog
first 1 ? possiblestocks

/ arrays and lists

x: 12 ? 06:00:00
```

```
y: 12 ? 300
z: x,y / two lists can be concatenated together

x[2 4] / lists can be indexed like arrays

z[4 18 15] / index lists even of mixed types

m: max y

y ? m / find index of max element in y

y[y ? m] / just checking that we've found the max

x[y ? m] / find time corresponding to index of max element of y

asc x / sort a list

iasc x / find the permutation that would give the sort

x[iasc x] / net effect is to sort x

y[iasc x] / net effect is to sort y based on ordering of x

(asc x),'(y[iasc x]) / concatenate pairs in order


sum y

sums y / running sums

sums y[iasc x]

x[where y > 175] / indexes in x give selection in y

sums y[where x > 02:00:00] / notice how operators go from right to ↩
    left
     / unless there are parens

z: 50 ? 8 / 50 numbers with replacement from 0 through 7 inclusive

g: group z / group them

key g

value g
```

```
(key g),'(count each value g)

/ find greatest index of each key value



/ string manipulation

s: "i am a string."

s[11] / index into the string

s[0 4 6] / multiple indexes into the string

s,s / concatenate strings

s,\: "&" / append blank to each character in s and get many lists

s,\: "1 2 3" / append several numbers after each character in s

raze s,\: " " / append blank to each character in s and then put ↩
   lists together

raze (" "),/: s / figure out how this is different from previous

s ss " a" / looking for indexes of space followed by a

s like "*rin*" / does the string "rin" exist anywhere?

ssr[s;"trin";"un"] / replace trin by un in s


/ basic functions

/ indent the name and outdent the other lines
/ semi-colon after each line except the last one
/ return value preceded by colon (:)
foo:{[a;z]
 x: a+2*z; / semi-colon after each line
 if[a < z; :x]; / return value preceded by colon; use of if
 i: 0;
 while[i < 10;
  x: x+5; / add 5 to x
```

```
   i+: 1;
  ];
 x}  / last line doesn't need a semi-colon

foo[5;10]

foo[10;5]

foopair:{[mypair] foo[mypair[0]; mypair[1]]}

x: 5 ? 10
y: 10 +  5 ? 20

x,'y / pairwise each is just a '

foopair each x,'y / single argument each (monadic) is the word "each"

x,/:\:y / cross-product

raze x,/:\:y / cross-product all at top level

foopair each raze x,/:\:y / try to figure out what is going on


/ Matrix multiply

m1: (1 2 3; 4 5 6; 7 8 9)

m2: (10 20; 30 40; 50 60)

m1[0]
m2[;1] / column value

dotprod:{sum x * y}

mymatmult: {[m1;m2] m1 dotprod/:\: flip m2}
```

Now let's put some commands inside a file and execute them based on their command-line arguments.

Call the file findcount.q
The "I"$ means that the arguments should be integers.
The arguments are on the command line and are copied to .z.x
findcount.q:

```
count "I"$ .z.x

v: "I" $ .z.x
sum v

\\
```

```
q findcount.q 2 3 42 11 1
```

Exercise 1: Create a file that produces the arithmetic mean of the arguments on the command line. q findmean.q 2 1 4 0 9

A few dyadic (dual argument) verbs.

```
x + y
x * y
x % y  / notice that division is %, because the / is used for ↩
   comments and more
```

Now, we can start writing procedures to do things. For example, let us write a procedure to project the medicare budget into the future. We have to know the current level, the percent raise, and the number of years. Then we can write a procedure that will give us the budget after so many years.

```
project:{[current; interest; numyears]
 x: (1 + interest) xexp numyears;
 x * current}

/ 2010 medicare budget is about $453
/ So projected budget in  2030 will be:
project[453; 0.03; 20]
```

See budget.q:

```
project:{[current; interest; numyears]
 x: (1 + interest) xexp numyears;
 x * current }


project[453; 0.03; 20]


/ can define a function in terms of others

superproject:{[f; x; y;z] f[x;y;z]}

superproject[project; 453; 0.04; 20]
```

Note a few things:
Need a semi-colon after every line but the last one. (Forgetting this can lead to strange bugs.)
Every line except the function definition should be outdented.

Now compute into the future and see what the budget will be.

The present value of an income stream given an annual interest rate is the sum of the contribution from year 1, year 2, ... To compute the contribution from year k, you have to take the amount you receive in year k and find the amount you would have to deposit now in order to get that amount k years from now.

Exercise 2: Create a file that produces the present value of a sequence of floating point numbers given as payments for year 1, year 2, ... up to the length of the vector. Hint: prds and # may help. Hint: try 10 # 0.3

```
        q presval.q 100 200 100 400
```

Suppose you want to compute certain summary statistics of a collection of numbers. For example, if the original sample is

```
x: 10 ? 5

x
```

You might get a mean of:

```
avg x
```

but the numbers are jumping around much more than if you have

```
1.9 + 10 ? 0.2
```

Under the normal distribution assumption, we would compute a standard deviation.

```
sqrt var x
```

But what if we don't care to assume a normal distribution (because we think the data could have a bias). In resampling statistics, you don't assume a distribution. Instead you take the data as it is and take repeated samples of the same size "with uniform probability at random using replacement". What this means is that we might create samples like this:

```
x[(count x) ? count x]
```

and again

```
x[(count x) ? count x]
```

and again

```
x[(count x) ? count x]
```

That is, you might create such samples 10,000 times and then store all the means. If you then sort all the means and take the 250th and the 9,750th you have the limits of the 95% confidence interval.

To do this, you'll need to understand two constructs: to append to a list, you can do something like:

```
mylist: x[10 ? count x]
mylist,: x[10 ? count x]
```

To iterate, you can use a loop such as:

```
mylist: ();
do[10000; mylist,: avg x[(count x) ? count x]]
```

Exercise 3: Given a list of integers at the command line, compute the mean, 2.5% value and 97.5% value using resampling.

Ok, now continuing our discussion of resampling statistics, let us say that we have two lists and we want to compute the significance of the difference of two means. A classical application is drug testing. We test the drug vs. the placebo. The way we do this is again by resampling. We take the two lists, compute the mean difference and then see whether permuting the labels would give us the same difference or more. We do this many times and determine what fraction of the time random permutation gives a bigger difference. Whenever random permutation gives a bigger difference, we ascribe the difference to chance.

For example, if we have

```
drug: 50 + 20 ? 15
placebo: 30 + 30 ? 30
```

then we get an initial difference of means of

```
(avg drug) - (avg placebo)
```

To see how often this would happen if we permuted the labels, we take the whole vector

```
v: drug,placebo
```

and permute it 10,000 times. After each permutation, we take the average value of the first (in this case) 15 elements and the next 30. We count how often the computed average is greater than or equal to the measured average. That count divided by 10,000 is the $p-value$. (p-value = probability that the observed difference in the mean happened by chance.)

Exercise 4a: Given two lists drug and placebo, find the $p - value$ of the difference in their means.

```
  q evalsig.q 4 5 2 3 5 1 _ 1 2 3 1 2 1
```

Exercise 4b (easier than 4a): The present interest rate is 5%. Over each of the next five years (including the first year), the interest rate can go up by 0.5%, down by 0.5%, or stay the same, all with the same probability. Find the median net 5 year interest rate, as well as the 75th and 25th percentiles.

```
        q montecarlo.q 5 0.5
```

Exercise 4c (harder than 4b) Allow the number of years, the present interest rate and the increment of the interest rate to be variable. Find the median net $n$ year interest rate, as well as the 75th and 25th percentiles.

Switching gears a little and recognizing that finance is a lot about time, let's learn a little about date arithmetic. First, there is a date *datatype*:

```
mystart: "D"$(string 2030),(".01.01")
myend: "D"$(string 2030),(".12.31")

`week$ mystart / finds  the date of the monday just prior or equal to↩
    mystart
`week$ myend
```

To find the day of the week of mystart.

```
finddayofweek:{[x]
  numdays: x - `week$ x;
  vec:`mon`tues`wed`thurs`fri`sat`sun;
  vec[numdays]}

finddayofweek[mystart]
```

Exercise 5: Find all the weekend dates of a given year where the year is given as a command line argument.

```
        q findweekend.q 2020
```

Exercise 5b: Find all third Saturdays in every month for a given year.

```
        q findthirdsat.q 2025
```

Q can also handle strings. For example, suppose that we wanted to take a bunch of command line arguments and find out how many times each argument occurs. Overall, this means we want to collect the command line arguments, then group them, and count them.

Let's build our way up to this because there is a lot we can learn from this.

```
    x: "S"$ .z.x
```

Now can do something called "grouping"

```
g: group x
```

Grouping creates a "dictionary" consisting (in this case) of argument, position pairs. For example,

```
q countargs.q dave was not here was not was
```

yields the following g:

```
dave| ,0
was | 1 4 6
not | 2 5
here| ,3
```

Now type:

```
key g
```

Then

```
value g
```

Do you see what you are getting?

Now it is time to put the keys and the counts together.

```
(key g),'(count each value g)
```

What is going on here? *count* just counts the number in a list. The modifier (or "adverb") each means that each list should be counted separately. The ,′ means that we are taking each key and combining it with the corresponding count.

Exercise 6: Find each argument with its *count*.

Exercise 7: Find the words tied for having the highest counts.

Here are some hints:

```
y: (key g),'(count each value g)

y[;1]

x: 50 + 10 ? 6

max x

x[where x = max x]
```

**Set Operations:** Is every member of x also in y?

```
subset:{[x;y] min x in y}
```

Find the elements of the x vector that are in a y vector

```
intersect
```

Find those elements of x that are not in y

```
difference
```

```
intersect:{[x;y] x[where x in y]}

/ union is built in

subset:{[x;y] min x in y}

difference:{[x;y]
 x where not x in y}

propersubset:{[x;y] subset[x;y] and not subset[y;x]}
```

**String operations** Delete all blanks in a line

Exercise: Write a multiintersect function

```
/ deletes all blanks
delblanks:{[x]
 ii: where not x = ‘‘ ”;
 x[ii] }
```

Take a string and separate it by blanks

```
/ separate into words
sepwords:{[x]
  ii: where x = ‘‘ ”;
  ii: distinct 0, ii;
  y: delblanks each ii _ x;
  c: count each y;
  ii: where not c = 0;
  y[ii]}
```

*lower* takes a string and puts it in lower case *ltrim* removes leading blanks; *rtrim* trailing blanks

```
‘‘|” sv(‘‘hey”;‘‘there”)          /yields ‘‘hey|there”
‘ sv ‘foo‘bar                     /yields ‘foo.bar
‘ sv ‘:/q‘tutorial‘draft1         /yields ‘:/q/tutorial/draft1
2 sv 1000b                        /yields 8
‘‘ ” vs ‘‘hey there”              /yields (‘‘hey”; ‘‘there”)
```

Exercise 8: Write a function that takes its command line arguments, splits them based on underbars into k lists and finds the intersection among those lists. All words should be reduced to lower case.

```
q findinter A b c x _ D x a e b _ b b d X d
```

yields b x

### Other string ops

### Translation from k to q :

```
ss[x;y] == ,/ x ,/:\: y
til 10 == !10
distinct x == ? x
where x = 1 == & x = 1
first x == *x
min x == &/ x
max x == |/ x
reverse x == |x
x like y == x _sm y / note that we can do things
        like fl[ao]p / to mean either a or o
        like [09] / to mean 0 though 9
       like ‘‘*[^09]” / strings that do not end with a digit
x ss y == x _ss y
enlist x == ,x
raze x == ,/x
list1 cross list2 == ,/list1 ,/:\: list2

value group x == =x
/group x == dictionary consisting of ?x and =x
```

Some things to note 1. In functions, must end lines with a semi-colon yet it executes even without. But when it executes it doesn't do the right thing. 2. If I use find, I have to know the type of the entity I'm finding. For example if I find among bits then I must say x ? 1b not x ? 1

## References

[1] Shasha, Dennis “Q for smarties”, 2011

[2] Kx Systems, *https://code.kx.com/trac/wiki/Reference*, 2011