A Glitch In Grade

Don't see APL characters?
Download SImPL (Unicode)
Download APL385 Unicode
Download Firefox

Introduction

While implementing Howard J. Smith, Jr.'s clever **algorithm** for dyadic character grade, I encountered a puzzle the explanation of which illuminates just how this primitive achieves its remarkable result. Moreover, there are free implementations of the comparison routine of this primitive in other languages (C, Perl, PHP).

Note that throughout this presentation the index origin is 0.

Sensible Sorting

Back in the dark ages, sorting started out as a simple idea: `A < B < C ...`. Along the way, various algorithms provided better and better performance, but the algorithm to compare two rows changed very little. Then, there came multiple alphabets (upper and lower case, etc.) and the problems began. As Smith points out in his **APL79** article, when this issue was discussed, one camp wanted the two alphabets to be **sequential** (`ab...zAB...Z`) and another camp wanted them to be **interspersed** (`aAbB...zZ`, or should it be `AaBb...Zz`?). The two choices of alphabet orderings can be thought of as either **sort on case** (sequential) or **sort on spelling** (interspersed). Both camps could produce examples where the other's choice failed to do what was expected.

It wasn't until the problem was considered in an APL context that the logjam was broken. Smith changed the comparison algorithm so as to perform a major sort on spelling and a minor sort on case — all at the same time!

Smith's idea was to break the one-dimensional mold of the collating sequence and use multiple dimensions to distinguish major from minor sort orders. That is, the characters in two of the rows of the matrix to be sorted are compared with respect to their indices within a multi-dimensional collating sequence. Differences in the indices along the rightmost axis are more significant than differences along the rows which are more significant than differences along the planes, etc.

For example, to sort on spelling (major) and case (minor) with upper case before lowercase, use

```
          ⎕←cs2a← 2 27 ρ'ƀABCDEFGHIJKLMNOPQRSTUVWXYZ',
                         'ƀabcdefghijklmnopqrstuvwxyz'
ƀABCDEFGHIJKLMNOPQRSTUVWXYZ
ƀabcdefghijklmnopqrstuvwxyz
```

where the Blank Symbol (ƀ) is used in place of a space when specifying collating sequences as a visible reminder of its presence.

This two-dimensional collating sequence sorts the matrix in the lefthand column unchanged, which is exactly what we want. To sort the same matrix, but with lowercase before uppercase, switch the two rows of `cs2a`, to yield the matrix in the righthand column.

Upper < Lower	Lower < Upper
AM	am
Am	Am
am	AM
AMA	AMA
Amazon	Amazon
pH	pH
PhD	PhD

Here's why: The fundamental idea behind Smith's algorithm is that when two rows in a matrix compare equally w.r.t. one dimension's collating sequence, shift to the preceding dimension to break the tie, etc. This means that you should construct collating sequences such that the major sort order is in the last dimension, the next most important sort order is in the next to the last dimension, and so forth. **Thus dimensions in the collating sequence take the place of multiple sorting passes.** Once again, APL subsumes looping into its primitive functions so the programmer doesn't have to bother with all the messy house keeping.

For example: Using `cs2a` and comparing rows `AM` and `Am` using the last dimension, they are equal, both having column indices of `1` for `A` and `13` for `Mm`. Comparison then shifts to the rows, where the row indices for `AM` are `0 0`, and the row indices for `Am` are `0 1`. This means that `AM` sorts before `Am`, as desired.

Sorting The Unknown

An interesting side effect of Smith's algorithm is that if it encounters a character not in the collating sequence, it pushes that character to the very end. For example, using `cs2a`, the following matrix is sorted unchanged:

```
a%
a*
a?
a:.
a.:
a!@#
a#!@
a@#!
```

Here's why: Smith's algorithm groups equal length rows which are otherwise equal because the space in the collating sequence (`'b̸'`) precedes all characters. Although the various rows in the example above don't look identical, they are equal as far as the collating sequence is concerned. Remember that the special characters in the above matrix do not appear in the collating sequence, so they are treated as equal. We'll see why this feature is important shortly.

Sorting Numerically Without Numbers, Part I

How many times have you seen labels such as the ones in the lefthand column sorted into the mish-mash in the second column?

Before	**After** `cs2b`	**After** `cs2a`	**After** `cs2b` **then** `cs2a`

```
L23         L1             L3             L1
L3          L10            L2             L2
L11         L100           L1             L3
L10         L11            L23            L10
L12         L12            L11            L11
L13         L13            L10            L12
L2          L2             L12            L13
L22         L21            L13            L21
L1          L22            L22            L22
L21         L23            L21            L23
L100        L3             L100           L100
```

Collating sequence `cs2b` illustrates how to achieve the brain-dead sorting in column two, which is why you want to avoid using a one-dimensional lexicographic sort:

```
          ⎕←cs2b←'␢ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789',
          2 37 ⍴ '␢abcdefghijklmnopqrstuvwxyz␢␢␢␢␢␢␢␢␢␢'
␢ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
␢abcdefghijklmnopqrstuvwxyz␢␢␢␢␢␢␢␢␢␢
```

The above collating sequence considers numbers (really digits) to be just another set of characters in the alphabet, which is why it fails to produce a desirable result. More accurately, it fails because the problem requires two passes: one to group equal length rows, and one to sort the numbers.

Note how sorting with `cs2a` groups the equal length rows together as discussed above in **Sorting The Unknown**. This means that if we first sort by `cs2b` and then by `cs2a`, we'd get the desired order as in the last column.

Rather than performing two consecutive sorts, we can do it in one, if we're clever enough.

The idea is to sort numbers in a separate dimension from the alphabets. That is, distinguishing differences in spelling from case requires a two-dimensional collating sequence. To distinguish letters from numbers, we need another dimension.

This three-dimensional collating sequence is essentially `cs2a` with another dimension tacked onto it, such as this `10` by `2` by `28` array which was, for many years, the "standard" way to sort these arrays:

```
          cs3a← 10 2 28 ρ 'ƀ'
          cs3a[0;0;]←'ƀABCDEFGHIJKLMNOPQRSTUVWXYZƀ'
          cs3a[0;1;]←'ƀabcdefghijklmnopqrstuvwxyzƀ'
          cs3a[;0;27]←'0123456789'
```

Actually, the version of this collating sequence that persisted for many years included the additional (but unnecessary) statement

```
          cs3a[1;1;]←cs3a[0;1;]
```

This collating sequence goes through three separate sorts — minor, middle, and major. The minor sort is for **numbers** (and is the sole purpose of the first — length 10 — dimension of the array), the middle sort is for **case** (the dimension of length 2), and the major sort is for **spelling** (the dimension of length 28).

Sorting Numerically Without Numbers, Part II

Say you have some file names such as

```
rfc2086.txt
rfc3.txt
rfc822.txt
```

and you would like them sorted in numeric order:

```
rfc3.txt
rfc822.txt
rfc2086.txt
```

Again, Smith's algorithm let's you do this, but the above collating sequence `cs3a` needs a minor adjustment because of the period separating the filename from its extension:

```
          cs3b← 10 2 29 ρ 'ƀ'
          cs3b[0;0;]←'ƀABCDEFGHIJKLMNOPQRSTUVWXYZ.ƀ'
          cs3b[0;1;]←'ƀabcdefghijklmnopqrstuvwxyzƀƀ'
          cs3b[;0;28]←'0123456789'
```

The reason we need to include a period in the collating sequence is that without it, that character is sorted to the end of the alphabet, hence when the major sort is made, that character forces that row to occur later. In particular, a set of rows of similar form to the ones above are sorted such that all rows with a period in the same column are sorted together and rows with a period earlier in the row are sorted later in the result.

Of course, the same reasoning applies to any character not in the given collating sequence.

A Glitch

Alas, this wonderful algorithm can't do everything you might want with just one collating sequence, as the following example shows. Above, we maligned collating sequence `cs2b`, however it has its uses. It produces exactly what we want from the following matrix:

Before	**After** `cs2b`	**After** `cs3b`
`1.700`	`1.001`	`1.3`
`1.02`	`1.010`	`1.7`
`1.001`	`1.02`	`1.02`
`1.010`	`1.3`	`1.001`
`1.702`	`1.7`	`1.010`
`1.7`	`1.700`	`1.700`
`1.3`	`1.702`	`1.702`

Here's why: Collating sequence `cs2b` works in this instance because it **doesn't** have a separate pass to group rows by equal length. For that same reason, note how our much vaunted `cs3b` produces a less than optimal result. So, as always, know your data and choose a collating sequence appropriate to it.

Another way of looking at why `cs3b` failed to provide the desired result is that because the second column of the matrix is all the same value, that column is effectively ignored, and the matrix is sorted according to the remaining columns. As those columns contain only numbers, we get the actual (and, to me, unexpected) result because `13 < 17 < 102 < 1001 < 1010 < 1700 < 1702`.

Removing the decimal point from the example might change the way you view this.

With either of the collating sequences `cs3a` or `cs3b`, the following matrix sorts unchanged:

```
2b7
2b8
8b8
2b08
```

If the row consisting of `'8b8 '` is changed to, say, `'8a8 '` or `'8c8 '`, then that row is indeed sorted last, so the values in that column are significant; however when they are all the same that column is ignored.

Second Verse, Same As The First

No good idea is invented just once, as is the case with Smith's. In fact, his idea has been re-invented at least twice:

- In 1996, Stuart Cheshire came up with an idea he called **Natural Order Numerical Sorting** for the Apple Macintosh. His approach was to sort numbers in strings numerically rather than lexicographically. It applies to numbers only, not to alphabetic characters, so it doesn't handle case differences.

- In 2000, Martin Pool came up with an idea he called **Natural Order String Comparison**. His idea is essentially the same as Cheshire's in that it applies to numbers in strings, only.

- There are several implementations of this algorithm, all of which seem to trace back to either Cheshire or Pool.

However, Pool's algorithm (I don't have access to a Mac to try Cheshire's) handles the glitch mentioned above better than does Smith's. It was when I tried the examples mentioned on his web page (**x2-g8 < x2-y7 < x2-y08 < x8-y8**) that I found the glitch in Smith's algorithm.

Other Languages

My original goal was to implement Smith's algorithm in other languages so I could use it elsewhere, and I succeeded. You can **download** a ZIP archive with implementations in C, Perl, and PHP. Each file in the ZIP archive contains a subroutine suitable for calling via one of the language's sort routines along with a test bed which sorts three example arrays.

Fonts

If you have trouble displaying the APL characters on this page, likely it is due to either a browser setting (or an out-of-date browser) or a missing font. Both Mozilla Firefox 2.0 or later and Internet Explorer 7 or later display the APL characters perfectly, but IE6 has some trouble. If this page doesn't display well with either APL Unicode font using any version of Internet Explorer, please try it again with Mozilla Firefox. Links for the two APL Unicode fonts as well as for Mozilla Firefox appear at the top of this page.

Author

This page was created by Bob Smith -- please **direct** any questions or comments about it to me. See my other **APL projects**.
