Natural Sorting

Doug Hennig

This is the second of several articles on components of Doug's in-house library. This issue focuses on why the normal sorting done by computers doesn't always work and how natural sorting solves the problem.

Computers typically sort in ASCII order. That is, "abc" comes before "abd" because although the first two characters of each string match, in the third character, "c" has a lower ASCII value than "d." Most of the time, that is the correct way to sort. However, there are some cases where a different type of sorting works better. For example, part numbers A-1, A-2, and A-10 ASCII sort as A-1, A-10, and A-2 but most people would expect to see A-1, A-2, and A-10 instead.

Wikipedia defines natural sort order as "an ordering of strings in alphabetical order, except that multi-digit numbers are ordered as a single character. Natural sort order has been promoted as being more human-friendly." Jeff Atwood wrote a blog article about this topic (http://tinyurl.com/hndt65e) about ten years ago, but it still isn't a widely known topic.

Having run into this issue several times myself, I decided to create a routine, NaturalSort.prg (**Listing 1**), that sorts an array using natural sort. NaturalSort accepts three parameters: the array to sort (passed by reference), the column to sort on (optional: if it isn't passed, the array is sorted on column 1), and .T. to sort in descending order or .F. or not passed for ascending order. This routine returns .F. if there is a problem with the parameters or the column in the array doesn't contain a homogeneous data type. If it returns .T., the array is sorted in natural order. The code is well-commented so it should be easy to follow.

Listing 1. NaturalSort.prg sorts an array in natural order.

```
lparameters taArray, ;
  tnColumn, ;
  tlDescending
local lnColumn, ;
  lnRows, ;
  lnCols, ;
  lnOrder, ;
  laArray[1], ;
  lnI, ;
  lcKey, ;
  lcString, ;
  laClone[1], ;
  lnIndex

* Ensure taArray is an array and tlDescending
* is logical if it's specified.

if type('taArray', 1) <> 'A' or ;
  (pcount() = 3 and ;
  vartype(tlDescending) <> 'L')
  return .F.
endif type('taArray', 1) <> 'A' ...

* If the column to sort on wasn't specified,
* assume 1.

lnColumn = iif(pcount() = 2, tnColumn, 1)

* Figure out the size of the source array.

lnRows = alen(taArray, 1)
lnCols = alen(taArray, 2)

* Ensure the column to sort on is valid.

if vartype(lnColumn) <> 'N' or ;
  not between(lnColumn, 1, max(lnCols, 1))
  return .F.
endif vartype(lnColumn) <> 'N' ...

* Figure out the order flag for ASORT().

lnOrder = iif(tlDescending, 1, 0)

* Get the data type of the first key value. If
* it isn't character, we don't have to do
* anything fancy; ASORT() will take care of it
* for us.

if vartype(taArray[1, lnColumn]) = 'C'

* Create an array we'll sort on.

  dimension laArray[lnRows, 2]

* Go through each element we're sorting on,
* get its natural sort key, and store the key
* and the original index in our sort array.

  for lnI = 1 to lnRows
    if lnCols = 0
      lcKey = taArray[lnI]
    else
      lcKey = taArray[lnI, lnColumn]
    endif lnCols = 0
    lcString = NaturalSortKey(lcKey)
    if not isnull(lcString)
      laArray[lnI, 1] = lcString
      laArray[lnI, 2] = lnI
    else
      return .F.
    endif not isnull(lcString)
  next lnI

* Now sort the array and reorder the values in
* the source array by cloning it and copying
* the values from the each row in the clone to
* the new row in the source array.
```

```
  asort(laArray, 1, -1, lnOrder, 1)
  acopy(taArray, laClone)
  for lnI = 1 to lnRows
    lnIndex = laArray[lnI, 2]
    if lnCols > 0
      for lnJ = 1 to lnCols
        taArray[lnI, lnJ] = laClone[lnIndex, ;
          lnJ]
      next lnJ
    else
      taArray[lnI] = laClone[lnIndex]
    endif lnCols > 0
  next lnI

* Just use ASORT to do the job.

else
  asort(taArray, lnColumn, -1, lnOrder, 0)
endif vartype(taArray[1, lnColumn]) = 'C'
return .T.
```

NaturalSort.prg calls NaturalSortKey.prg. Originally, the code in NaturalSortKey.prg was included in NaturalSort.prg but Mike Potjer pointed out that splitting the code into a separate routine would allow natural sorting in other places such as a SQL statement. NaturalSortKey (**Listing 2**) does the hard part: assigning a natural key to a string by left-padding numeric sections to a consistent length with zeros so they sort properly. It accepts two parameters: the string and the length to use for numeric sections (optional: if it isn't passed, a length of 20 is used). Again, the code should be easy to understand.

Listing 2. NaturalSortKey.prg assigns a natural key to a string.

```
lparameters tcKey, ;
  tnLength
local lnLength, ;
  llInNumeric, ;
  lcString, ;
  lnI, ;
  lcChar, ;
  llNumeric, ;
  lcNumeric

* Define the constants.

#define cnLENGTH 20
  && the length to pad numeric sections to
  && by default

* Bug out if the data type is wrong.

if vartype(tcKey) <> 'C'
  return .NULL.
endif vartype(tcKey) <> 'C'

* Use a default length if not specified.

do case
  case pcount() = 1
    lnLength = cnLENGTH
  case vartype(tnLength) = 'N' and ;
    between(tnLength, 1, 60)
    lnLength = tnLength
  otherwise
    return .NULL.
endcase

* Create a key that will sort properly by
* looking for numeric sections and left-
* padding them with zeros.

llInNumeric = .F.
lcString    = ''
for lnI = 1 to len(tcKey)
  lcChar    = substr(tcKey, lnI, 1)
  llNumeric = isdigit(lcChar) or ;
    (lcChar = '.' and ;
    isdigit(substr(tcKey, lnI + 1, 1)))
  do case
    case llNumeric and llInNumeric
      && if we have a digit and we're already
      && in a numeric
      && section, add to the numeric part
      lcNumeric = lcNumeric + lcChar
    case llNumeric
      && if we have a digit and we're not in a
      && numeric section, flag that we are in
      && such a section and add to the numeric
      && part
      llInNumeric = .T.
      lcNumeric   = lcChar
    case llInNumeric
      && we don't have a digit and we were in
      && a numeric section so pad the section
      && and add it to our string
      llInNumeric = .F.
      lcString    = lcString + ;
        padl(lcNumeric, lnLength, '0') + ;
        lcChar
    otherwise
      lcString = lcString + lcChar
  endcase
next lnI

* Finish the string if we were still
* processing a numeric section.

if llInNumeric
  lcString = lcString + ;
    padl(lcNumeric, lnLength, '0')
endif llInNumeric
return lcString
```

TestNaturalSortFiles.prg (**Listing 3**) tests NaturalSort.prg with filenames. It first lists the filenames in the order created by ASORT, then the results created by NaturalSort.prg.

Listing 3. TestNaturalSortFiles.prg tests sorting filenames.

```
* Create an array of filenames.

dimension laFiles[7]
laFiles[1] = 'a1.txt'
laFiles[2] = 'a2.txt'
laFiles[3] = 'a3.txt'
laFiles[4] = 'a10.txt'
laFiles[5] = 'a11.txt'
laFiles[6] = 'a100.txt'
laFiles[7] = 'a101.txt'

* Display the ASCII sort.

asort(laFiles)
clear
? 'ASCII sort'
for lnI = 1 to 7
  ? laFiles[lnI]
next lnI

* Display the natural sort.

NaturalSort(@laFiles)
?
? 'Natural sort'
for lnI = 1 to 7
  ? laFiles[lnI]
next lnI
```

Running the program shows that NaturalSort.prg sorts as expected:

```
ASCII sort
a1.txt
a10.txt
a100.txt
a101.txt
a11.txt
a2.txt
a3.txt

Natural sort
a1.txt
a2.txt
a3.txt
a10.txt
a11.txt
a100.txt
a101.txt
```

TestNaturalSortIDs.prg (Listing 4) is similar, but it sorts a set of product codes.

Listing 4. TestNaturalSortIDs.prg tests sorting product codes.

```
* Create an array of product IDs.

dimension laItems[10]
laItems[ 1] = 'A-1'
laItems[ 2] = 'A-2'
laItems[ 3] = 'A-3'
laItems[ 4] = 'A-10'
laItems[ 5] = 'A-11'
laItems[ 6] = 'A-100'
laItems[ 7] = 'A-101'
laItems[ 8] = 'A-1-1'
laItems[ 9] = 'A-1-10'
laItems[10] = 'A-1-2'

* Display the ASCII sort.

asort(laItems)
clear
? 'ASCII sort'
for lnI = 1 to alen(laItems)
  ? laItems[lnI]
next lnI

* Display the natural sort.

NaturalSort(@laItems)
?
? 'Natural sort'
for lnI = 1 to alen(laItems)
  ? laItems[lnI]
next lnI
```

Here are the results of running this program. Notice it properly handles multiple numeric sections.

```
ASCII sort
A-1
A-1-1
A-1-10
A-1-2
A-10
A-100
A-101
A-11
A-2
A-3

Natural sort
A-1
A-1-1
A-1-2
A-1-10
A-2
A-3
A-10
A-11
A-100
A-101
```

TestNaturalSortSQL.prg (Listing 5) tests sorting a cursor created by a SQL statement using NaturalSortKey. It uses the same product codes TestNaturalSortIDs.prg uses.

Listing 5. TestNaturalSortSQL.prg shows how to use NaturalSortKey in a SQL statement.

```
* Create a cursor of product IDs.

create cursor PRODUCTS (ID C(10))
insert into PRODUCTS values ('A-1')
insert into PRODUCTS values ('A-2')
insert into PRODUCTS values ('A-3')
insert into PRODUCTS values ('A-10')
insert into PRODUCTS values ('A-11')
insert into PRODUCTS values ('A-100')
insert into PRODUCTS values ('A-101')
insert into PRODUCTS values ('A-1-1')
insert into PRODUCTS values ('A-1-10')
insert into PRODUCTS values ('A-1-2')

* Display the ASCII sort.

select * ;
  from PRODUCTS ;
  into cursor ASCIISort ;
  order by 1
browse

* Display the natural sort. Unfortunately, the
* first statement gives a "SQL: ORDER BY
* clause is invalid" error, so we have to
* include the sort value in the cursor.

*select ID ;
*  from PRODUCTS ;
*  into cursor NaturalSort ;
*  order by NaturalSortKey(ID)
select ID, ;
    NaturalSortKey(ID) as SORT ;
  from PRODUCTS ;
  into cursor NaturalSort ;
  order by SORT
browse
```

Summary

NaturalSort.prg and NaturalSortKey.prg can sort things your application displays in the order users expect to see them rather than the ASCII order used by commands and functions such as ASORT, INDEX ON, and the ORDER BY clause of a SQL SELECT statement. It's just one more step to creating a polished application that works how users want.

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "VFPX: Open Source Treasure for the VFP Developer," "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 to 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).