SESUG Paper 288-2018

Sorting Arrays Using the Hash Object

Paul M. Dorfman, Independent Consultant

ABSTRACT

Before the advent of the SAS® Hash Object, two options had been available for sorting SAS arrays: (1) using the SORTN or SORTC call routines or (2) implementing a sorting algorithm, such as the quick sort, heap sort, etc. The first option is limited by its dependence on the data type and inability to handle duplicate keys and parallel arrays. The second option requires rather sophisticated custom programming. The SAS hash object, with its built-in ability to sort its hash items internally, is devoid of the above deficiencies. In this paper, we show how it can be used to sort SAS arrays simply and efficiently.

INTRODUCTION

It often happens that sorting an array is the simplest and fastest way to get from a DATA step program exactly what you want. Suppose, for example, that you want to get the K smallest or largest values from N variables incorporated in an array; or you want to find the common values between two sets of variables in every observation; and so on. Many a SAS programmer can come up with a number of scenarios where such "horizontal sorting" may be conducive to one's goal. This is a plausible reason why questions like "How to sort an array?" had been standard fare on SAS online forums before the advent of SAS 9.2. Offered solutions ranged from "Why do you need it?" to full-fledged DATA step implementations of such algorithms as quicksort or combsort (e.g. Dorfman, 2001). The introduction of the rapid SORTN/SORTC call routines in SAS 9.2 seemed to render such custom-coding excesses nugatory. Well, not quite: For all their agility, they can sort only ascending and offer no provisions for handling duplicates or sorting parallel arrays. Though by the time SAS had introduced the hash object, its ability to sort data has remained largely obscure. This paper aims to fill in this gap explicitly.

ARRAY SORTING PRELIMINARIES

Let us sort first things first and agree on some nomenclature related to array sorting (sometimes also termed "horizontal sorting"). Suppose that we have an array K defined and initialized as:

```
data _null_ ;
  array K [11] (9 2 8 1 7 6 2 5 7 4 3) ;
run ;
```

There can be hardly any disagreement that to sort K in *ascending* order means to *permute* the values of the variables K1-K9 in such a way that after the permutation, their values from left to right *never decrease*. And, conversely, to sort it in *descending* order means that after the permutation, they *never increase*. In other words, if we first (1) leave the array intact, then (2) sort it ascending, then (3) sort it descending, the content of K after each stage listed above will look like in the following picture:

Order	K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11
Original	9	2	8	1	7	6	2	5	7	4	3
Ascending	1	2	2	3	4	5	6	7	7	8	9
Descending	9	8	7	7	6	5	4	3	2	2	1

Array K can be easily sorted in *ascending* order using the SORT*N* call routine:

```
data _null_ ;
  array K [9] (9 2 8 1 7 6 2 5 7 4 3) ;
  call sortN (of K[*]) ;
```

```
run ;
```

As the reader may have guessed, the notation K for the array name is a hint that we view its values as *key-values* for sorting. Note that if the array were of the character type, the SORT*C* routine would have to be used instead. Since neither call routine has a provision for descending order, if we wanted to sort descending, we would have to do the extra work of swapping the opposite elements of K after having sorted it ascending.

Note that the array K contains duplicate values. Often times, it may be desirable to eliminate them from the sorted array in the fashion similar to using the NODUPKEY option with the SORT procedure. In other words, the result could look as follows (N=unsorted, A=ascending, D=descending):

Order	K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11
N	9	2	8	1	7	6	2	5	7	4	3
A	.	.	1	2	3	4	5	6	7	8	9
D	9	8	7	6	5	4	3	2	1	.	.

We may want, depending on the situation, to have such options as (a) placing the array items freed by the eliminated duplicates at the end or the beginning of K and/or (b) filling them with specific values (for example, "00"x or "FF"x in the case of a character array). However, the SORT*N* and SORT*C* call routines offer no such provisions, either.

PARALLEL ARRAYS

Suppose that instead of a single array, we have two *parallel* arrays, like so:

```
data _null_ ;
  array K [10] (4  1  3  4  3  2  4  2  3  4 ) ;
  array D [10] (41 11 31 42 32 21 43 22 33 44) ;
run ;
```

Now suppose that like before, we want to sort array K; but in addition, we also want to permute the values of array D (hint: "data") accordingly. In other words, array K serves as the sort key and the values of array D - as the associated data. The desired action is analogous to sorting a file with variables (K,D) by K.

There are two possible results that can be expected from such a sorting action. One of them, according to the accepted nomenclature, is called *stable sorting*. It occurs when, after the sorting is done, the relative positions of the data items associated with each key within every same-key group retain their relative positions they had in the original unsorted array. With the keys and data in our sample arrays as shown above, the result of stable sorting will look as follows:

Order	Array	Item1	Item2	Item3	Item4	Item5	Item6	Item7	Item8	Item9	Item10
N	K	4	1	3	4	3	2	4	2	3	4
	D	**41**	11	31	**42**	32	21	**43**	22	33	**44**
A	K	1	2	2	3	3	3	4	4	4	4
	D	11	21	22	31	32	33	**41**	**42**	**43**	**44**

Note, for example, that the order of the D-items for K=4 in the unsorted arrays D is (41,42,43,44). It remains exactly the same in the sorted arrays as well, even though their items have been rearranged according to the key order. The same is true for all other key-value groups. This is a hallmark of *stable* sorting. Within the realm of the SORT procedure, it is achieved by using the EQUALS option.

On the other hand, the result might also look, for example, as follows:

Order	Array	Item1	Item2	Item3	Item4	Item5	Item6	Item7	Item8	Item9	Item10
N	K	4	1	3	4	3	2	4	2	3	4
	D	**41**	11	31	**42**	32	21	**43**	22	33	**44**
A	K	1	2	2	3	3	3	4	4	4	4
	D	11	22	21	32	31	33	**43**	**42**	**44**	**41**

In this case, for all intents and purposes, the arrays are still properly sorted since (a) the keys are in order and (b) all D-items reside within their corresponding K-groups. The difference is that within a K-group, the relative order of the D-items does not replicate their relative order in the original unsorted array. Therefore, such sorting is *unstable*. This situation may occur with the SORT procedure if NOEQUALS option is in effect. It is not necessarily bad - it just depends on what kind of result one needs to obtain.

Whether or not a specific array sorting algorithm results in stable sorting, depends on the algorithm itself. For example, the famed quicksort algorithm, while fast and efficient, does *not* provide for stable sorting; while others, such as the merge sort, do. Leaping a bit ahead, when the hash object is used for array sorting, it always results in stable sorting due to its intrinsic properties.

Needless to say, parallel array sorting scenarios are not exhausted by sorting two parallel arrays. For example, there can be two or more *key arrays*, in which case the values from different arrays with the same *array index* form an equivalent of a composite key. Likewise, there can be more than one *data array* whose values must be permuted in accordance with the simple or composite key of their parallel array counterparts. Furthermore, unlike the sample arrays shown above, arrays involved in sorting can be of different data types and item lengths.

The same type of scenario arises when there is a need to sort a multi-dimensional array - in essence, a matrix - using some dimensions as keys and the others - as satellite data. In this case, the only circumstance that makes the task somewhat simpler is the fact that all dimensions are, obviously, of the same data type, i.e. either numeric of character and of the same length.

Can such horizontal sorting situations be handled using the SORT*N*/SORT*C* routines? In principle, they can. But it is easy to perceive what kind of programmatic contortions it takes to attain the goal even in the trivial case of the two arrays shown above. Suppose, for example, that we want to sort K descending and permute D correspondingly:

```
data _null_ ;
  array K [10] (4  1  3  4  3  2  4  2  3  4 ) ;
  array D [10] (41 11 31 42 32 21 43 22 33 44) ;
  array S [10] $ 5 _temporary_ ;
  do i = 1 to dim (K) ;
    S[i] = catx ("_", K[i], D[i]) ;
  end ;
  call sortC (of S[*]) ;
  do i = 1 to dim (K) ;
    K[i] = input (scan (S[i], 1 ,"_"), D.) ;
    D[i] = input (scan (S[i], 2 ,"_"), D.) ;
  end ;
  /* Swap into descending order */
  i = 1 ;
  j = dim (K) ;
  do while (i < j) ;
    _tK  = K[i] ; _td  = D[i] ;
    K[i] = K[j] ; D[i] = D[j] ;
    K[j] = _tK  ; D[j] = _tD  ;
    i ++ 1 ;
    j +- 1 ;
  end ;
run ;
```

Aside from the fact that it is quite a bit of code for a rather simple task, the programmer has to be cognizant of the array types, their lengths, must concatenate the array items, parse them apart on the way out, and hard code the informats accordingly. Furthermore, it does not provide for stable sorting should one be needed because in the above approach, both the keys and the data must be forced into a single composite key, and so the D-values within each K-value come out sorted descending rather than in the original relative order.

HASH OBJECT SORTING ORDER

In order to understand how the hash object can be used to sort arrays, we need to know how its items are ordered within the hash table. To get an idea, let us load the values of our two sample arrays K and N into the key and data portions of a hash object, respectively, output its contents to a SAS data set, and observe how the resulting hash table data are structured.

```
data sorted_A (keep = _: rename = (_K=K _D=D)) ;
  array K [10] (4  1  3  4  3  2  4  2  3  4 ) ;
  array D [10] (41 11 31 42 32 21 43 22 33 44) ;
  /* Create hash and hash iterator object instances */
  dcl hash s (multidata:"Y", ordered:"A", hashexp:8) ;
  s.defineKey  ("_K") ;
  s.defineData ("_K", "_D") ;
  s.defineDone () ;
  dcl hiter is ("s") ;
  /* Populate hash S from the arrays */
  do i = 1 to dim (K) ;
    _K = K[i] ;
    _D = D[i] ;
    rc = s.add() ;
  end ;
  /* Write hash table content to output data set */
  do i = 1 to s.num_items ;
    rc = is.next() ;
    output ;
  end ;
run ;
```

The argument tag MULTIDATA:"Y" tells the hash object to accept duplicate-key items; and the argument tag ORDERED:"A" tells it to sort the table ascending by _K. If we run the step as is and then rerun it again twice with the argument tag ORDERED:"N" (i.e. with no sorting order imposed) and ORDERED:"D" (i.e. to sort descending), every time changing the output data set name accordingly, we will get three differently ordered data sets. Below, their contents are displayed side by side:

Sorted_N		Sorted_A		Sorted_D	
K	D	K	D	K	D
2	21	1	11	4	41
2	22	2	21	4	42
1	11	2	22	4	43
3	31	3	31	4	44
3	32	3	32	3	31
3	33	3	33	3	32
4	41	4	41	3	33
4	42	4	42	2	21

Sorted_N		Sorted_A		Sorted_D	
K	D	K	D	K	D
4	43	4	43	2	22
4	44	4	44	1	11

From glancing at this illustration, we can make a number of observations:

1. Regardless of the sorting order imposed on the hash table (including no particular order in the case of ORDERED:"N"), hash items with the same key-values are *always grouped* together.
2. A particular sorting order determines only the positions of the whole same-key groups relative to each other.
3. Irrespective of the relative position of any same-key item group, the sequence of its items exactly replicates the sequence with which they have been added to the table. In other words, hash table sorting is always stable. (To make this property more eminently observable, in the table above the item group with K=4 is slightly shaded.)

As a side note, the ability of the hash object to have its items in order stems from its internal design. The word "hash" implies total disorder, and this is indeed the case with classic hashing algorithms. With the SAS hash object, however, the items are inserted into a number of ordered binary trees. Thus, when they are retrieved, they are traversed in perfect ascending key order. The number of such trees is defined by the argument tag HASHEXP whose default value is 8 (and so in the program above it could be simply omitted - it is included there just for the sake of making the point). For HASHEXP:X, the number of trees is defined as 2**X. Hence, if we code HASHEXP:0, then 2**X=1, and the entire hash table is a single binary tree. Therefore, if HASHEXP:0, the table is always internally ordered ascending even if no order is imposed on it via the argument tag ORDERED.

For more details on the subject, see (Dorfman and Henderson, 2018).

SINGLE KEY ARRAY

Now that we know that the items can be retrieved from a hash table in a predetermined key order, we have everything we need to sort any array or parallel arrays. In fact, the program shown above already contains all the necessary nuts and bolts - we only need to write the hash items back into the array rather than to the output data set. The most trivial task, of course, is to *sort a single array* by implementing the following simple scheme:

4. Create and instantiate a ordered hash object instance.
5. Create a hash iterator instance associated with the table.
6. Define a hash variable of the same type and length as the array in the key portion and the data portion. The latter is critical since only the values of hash variables defined in the data portion can be retrieved into the PDV.
7. Add the key and data array elements, one at a time, to the hash table as hash items.
8. Enumerate the table using the hash iterator - i.e. scan the table from the first hash item to the last - and assign the value of each item in the sequence to the array elements left to right.

Assuming that we want to sort ascending and keep the duplicates, let us and express the scheme above in the SAS language:

```
data _null_ ;
  array K [10] (4  1  3  4  3  2  4  2  3  4) ;
  dcl hash s (multidata:"Y", ordered:"A") ;
  s.defineKey  ("_K") ;
  s.defineData ("_K") ; /* In this case, can be omitted */
```

```
  s.defineDone () ;
  dcl hiter is ("s") ;
  do x = lbound (K) to hbound (K) ;
    _K = K[x] ;
    rc = s.ADD() ;
  end ;
  put "Unsorted: " K[*] ;
  do i = lbound (K) to hbound (K) ;
    is.NEXT() ;
    K[x] = _K ;
  end ;
  put "Sorted:   " K[*] ;
run ;
```

The program prints the following in the SAS log:

```
Unsorted: 4 1 3 4 3 2 4 2 3 4
Sorted:   1 2 2 3 3 3 4 4 4 4
```

The only change to the program needed to sort the array descending is to recode the value of the argument tag ORDERED as "D", in which case the program would print in the log, correspondingly:

```
Unsorted: 4 1 3 4 3 2 4 2 3 4
Sorted:   4 4 4 4 3 3 3 2 2 1
```

The program above merits two special notes:

1. The reason the range of the index X is chosen as lbound(K):hbound(K), rather than 1:dim(K), is that it works properly even if the lower bound does not start at 1. This way, the program would need no change if the array were declared as, for example, [-3:6] instead of [1:10].
2. We do not have to do anything special to define the PDV host variable _K with the proper data type and length. This is because assigning K[i] to _K causes the compiler to create _K in the PDV with exactly the same attributes as any variable in array K. The *hash* variable _K then inherits these attributes when the DEFINEDONE method is called at run time.

HANDLING DUPLICATES

The ability to handle items with duplicate key-values is built in the hash object. In fact, it has a rich set of features that allow, in addition to merely ignoring duplicate-key items, to select one of them conditionally. Since in the example above we deal only with the keys, all items in the same-key group are equivalent. So, if we need to eliminate duplicate array elements, it does not matter which duplicate to select for the sorted array. However, SAS arrays are fixed-length data structures, their number of elements being defined at compile time. Thus, if we want to eliminate the duplicates, we need to decide on (a) where in the sorted array the distinct values should reside and (b) what kind of values the rest of the array items should be populated with.

Suppose that we have an array with *ND* elements, out of which *NU* elements are unique. Most usually, the distinct values fill the first *NU* slots of the sorted array, and the remaining (*ND*-*NU*) items in the back of the array are filled with standard missing values. For instance, with this modus operandi, the demo array shown above, if sorted ascending and unduplicated, would look this way:

```
Sorted:   1 2 3 4 . . . . . .
```

However, specifications may vary; for example, it could be required that the missing values be placed up front instead:

```
Sorted:   . . . . . . 1 2 3 4
```

Either way, it can be easily accommodated using the hash object. One, rather obvious, approach to achieve it is to value the MULTIDATA argument tag with "N" instead of "Y" or just drop it from the program altogether. This will cause any ADD method call attempting to insert a duplicate key-value into the hash table to fail with a non-zero return code RC. Thus, only the first instance of any duplicate input key-value will end up in the table, which in the end will contain *NU* items with unique key-values.

However, this approach is *not* recommended - not because it does not work (it does) but because there is a better, more flexible way, which is also better suited to situations more complex that a single array. Thus, let us instead keep MULTIDATA:"Y" but change the method of inserting the items into the table:

```
data _null_ ;
  array K [-3:6] (4  1  3  4  3  2  4  2  3  4) ;
  dcl hash s (multidata:"Y", ordered:"A") ;
  s.defineKey  ("_K") ;
  s.defineData ("_K") ; /* In this case, can be omitted */
  s.defineDone () ;
  dcl hiter is ("s") ;
  do i = lbound (K) to hbound (K) ;
    _K = K[i] ;
    s.REF() ;                          /* Insert first dup key-value */
  * if s.CHECK() ne 0 then s.ADD() ; /* Same as s.REF()            */
  * s.REPLACE() ;                    /* Insert last  dup key-value */
  end ;
  put "Unsorted: " K[*] ;
  do i = lbound (K) to hbound (K) ;
    if i < lbound (K) + s.NUM_ITEMS then do ;
      is.NEXT() ;
      K[i] = _K ;
    end ;
    else K[i] = . ;
  end ;
  put "Sorted:   " K[*] ;
run ;
```

This program deserves some notes, too:

- Calling the REF method tells the program: If the key-value is not already in table S, insert the item, otherwise ignore it. This way, only the *first* duplicate-key input value is inserted into the table.
- The REF method obviates the need to use the combination of the CHECK and ADD methods, as shown in the corresponding (commented out) line, to achieve the same goal.
- If we want to ignore all duplicate input keys but the *last*, we can call the REPLACE method instead.
- In the case of a single array, it does not matter which occurrence of a duplicate input value to keep. However, as we will see later, it does matter when we have another, parallel, array filled with associated data.
- The NUM_ITEMS hash attribute tells us how many items are currently in the table. Since here we are inserting items with unique key-values, the attribute also tells us how many distinct values we have in the array without the need to count them.
- Hence, NUM_ITEMS is useful when we need to determine, as it is done above, when to start filling the slack in the array with a filler. It is also used to tailor the number of the NEXT iterator method calls exactly to the number of items in the table and thus avoid the error would occurs if the number were exceeded.
- In this case, the slack is filled with standard numeric missing values; yet any other value, if need be, can be chosen instead. Needless to say, it has to be of the same data type and length as the array itself.

The program as shown above prints in the log:

```
Unsorted: 4 1 3 4 3 2 4 2 3 4
Sorted:   1 2 3 4 . . . . . .
```

In other words, the slack caused by the discarded duplicate values is filled in the back of the array. In the case we needed to move the fillers to the front, we only have to slightly change the logic in the second DO loop, leaving everything else intact; for example:

```
do i = lbound (K) to hbound (K) ;
  if i <= hbound (K) - s.NUM_ITEMS then K[i] = . ;
  else do ;
    is.NEXT() ;
    K[i] = _K ;
  end ;
end ;
```

Having been changed in this manner, the program will print instead:

```
Unsorted: 4 1 3 4 3 2 4 2 3 4
Sorted:   . . . . . . 1 2 3 4
```

Depending on what is needed to be done with the array afterwards, it may be preferable since it makes the *entire* array, including the filler values, truly sorted.

PARALLEL KEY ARRAYS

A bit more interesting situation arises when we have two or more parallel *key* arrays. By key arrays we mean arrays whose values in the same index positions, as a combination, are viewed from the standpoint of sorting as a *composite key*. For example, let us consider two parallel arrays defined as follows:

```
array KN [10]    ( 4   1   3   4   3   2   4   2   3   4 ) ;
array KC [10] $1 ('F' 'A' 'E' 'G' 'D' 'C' 'F' 'B' 'D' 'H') ;
```

Above, the same-index value pairs (4 F), (1 A), (3 E) and so one are considered composite key-values, and the goal is to sort the two arrays in sync based on these values. For instance, if we wanted to sort them ascending and keep only the unique composite key-values, we would mean to obtain the following result after the sorting is done (assuming that we want the missing-filled slack up front):

```
KN: . . 1 2 2 3 3 4 4 4
KC:     A B C D E F G H
```

The hash object, with its built-in ability to handle composite keys regardless of the data type of their components, makes the task a breeze. The only change we need to make to the program above is to define the hash table with *two* key and *two* data variables and add the corresponding assignment statements. For example:

```
data _null_ ;
  array KN [10]    ( 4   1   3   4   3   2   4   2   3   4 ) ;
  array KC [10] $1 ('F' 'A' 'E' 'G' 'D' 'C' 'F' 'B' 'D' 'H') ;
  dcl hash s (multidata:"Y", ordered:"A") ;
  s.defineKey  ("_KN") ;
  s.defineKey  ("_KC") ;
  /* No DEFINEDATA call, so _KN/_KC are added to data portion by default */
  s.defineDone () ;
  dcl hiter is ("s") ;
  do i = lbound (KN) to hbound (KN) ;
    _KN = KN[i] ;
```

```
    _KC = KC[i] ;
    s.REF() ;
  end ;
  put "Unsorted: KN: " KN[*] / +10 "KC: " KC[*] ;
  do i = lbound (KN) to hbound (KN) ;
    if i <= hbound (KN) - s.NUM_ITEMS then do ;
      KN[i] = . ;
      KC[i] = . ;
    else do ;
      is.NEXT() ;
      KN[i] = _KN ;
      KC[i] = _KC ;
    end ;
  end ;
  put "Sorted:   KN: " KN[*] / +10 "KC: " KC[*] ;
run ;
```

Thus augmented, the step prints in the log:

```
Unsorted: KN: 4 1 3 4 3 2 4 2 3 4
          KC: F A E G D C F B D H
Sorted:   KN: . . 1 2 2 3 3 4 4 4
          KC:     A B C D E F G H
```

Now, what would we have to do if we had more than two parallel key arrays to sort? Not much, really: We would only have to extrapolate the program by adding extra lines of code corresponding to the extra arrays to the lines shown above in bold.

Note that in doing so, we do not have to do any work *at all* in order to account for the data types of the arrays involved in the process. This is because the types and lengths of the PDV variables _K and _N (and whatever variables we would add to them if we had more key arrays) are automatically defined by the assignment statements at compile time, and this information is automatically conveyed to the hash object constructor when the DEFINEDONE method is called.

MULTI-DIMENSIONAL KEY ARRAYS

A multi-dimensional array K[N,M] is, in essence, no more than a collection of N 1-dimensional arrays K1[M], ... , KN[M]. From the standpoint of sorting it, for every M, each set of array elements from K[1,M] to K[N,M] constitutes a composite key-value. Therefore, ordering it using the hash object is, in principle, no different than ordering N parallel key arrays - and we already know how to do it. In fact, it is even simpler since: (a) all array dimensions are of the same data type and (b) they can be addressed using the first-dimension index. To illustrate the process, let us assume that we have a 4-dimensional array with 13 items in each dimension defined as follows:

```
array K [4,13] (2 2 1 1 1 2 2 2 1 1 1 2 2
                3 4 4 4 4 3 4 3 4 3 4 4 3
                6 6 5 6 5 6 5 5 5 5 6 5 6
                7 7 7 8 7 7 8 7 7 7 8 8 7) ;
```

The tuples (2 3 6 7), (2 4 6 7), (1 4 5 7), and so on make up the composite key-values we want to sort on. Hence, after the sorting is done, we expect the array to look like this (assuming ascending order):

```
1 1 1 1 1 1 2 2 2 2 2 2 2
3 4 4 4 4 4 3 3 3 3 4 4 4
5 5 5 5 6 6 5 6 6 6 5 5 6
7 7 7 7 8 8 7 7 7 7 8 8 7
```

Or, if the composite duplicate key-values were to be discarded and filled with missing values up front, the resulting sorted array would look like this:

```
. . . . . 1 1 1 2 2 2 2
. . . . . 3 4 4 3 3 4 4
. . . . . 5 5 6 5 6 5 6
. . . . . 7 7 8 7 7 8 7
```

Implementing this sorting is no more than extrapolating the program for sorting two parallel key arrays, augmented with some array index gymnastics:

```
data _null_ ;
  array K [4,13] (2 2 1 1 1 2 2 2 1 1 1 2 2
                  3 4 4 4 4 3 4 3 4 3 4 4 3
                  6 6 5 6 5 6 5 5 5 5 6 5 6
                  7 7 7 8 7 7 8 7 7 7 8 8 7) ;
  dcl hash s (multidata:"Y", ordered:"A") ;
  /* Use array _K to define hash variables _K dynamically */
  array _K [4] ;
  do i = 1 to dim (_K) ;
    s.defineKey (vname (_K[i])) ;
  end ;
  s.defineDone() ;
  dcl hiter is ("s") ;
  /* Fill up hash table S from array K */
  do j = lbound (K,2) to hbound (K,2) ;
    do i = lbound (K,1) to hbound (K,1) ;
      _K[i] = K[i,j] ;
    end ;
    s.REF(); /* s.REF() kills duplicates. Use s.ADD() to keep them */
  end ;
  /* Stream sorted hash data back into array K */
  do j = lbound (K,2) to hbound (K,2) ;
    if j > Hbound (K,2) - s.NUM_ITEMS then do ; /* #1: fillers up front */
  * if j < Lbound (K,2) + s.NUM_ITEMS then do ; /* #2: fillers out back */
      is.NEXT() ;
      do i = lbound (K,1) to hbound (K,1) ;
        K[i,j] = _K[i] ;
      end ;
    end ;
    else do i = lbound (K,1) to hbound (K,1) ;
      K[i,j] = . ;
    end ;
  end ;
  /* Show sorted array content in SAS log */
  do i = lbound (K,1) to hbound (K,1) ;
    do j = lbound (K,2) to hbound (K,2) ;
      put K[i,j] @ ;
    end ;
    put ;
  end ;
run ;
```

A few notes about this program:

- Instead of hard coding DEFINEKEY method calls for every value of the first dimension from 1 to 4, we have made use of the hash object's ability to accept argument tag values as resolved expressions - in this case, in the form of the VNAME function.
- If the REF method is called, the duplicate values are discarded. If the fillers need to be put up front, use condition #1 within the second nested DO loop; if they are needed in the back, use condition #2.

- If the ADD method is called instead of REF, the duplicates are kept regardless of whether condition #1 or #2 is used. Alternatively, in this case this conditional logic can be discarded, and the entire nested DO loop - simplified as:

```
/* Stream the sorted hash data back into array K */
do j = lbound (K,2) to hbound (K,2) ;
  is.NEXT() ;
  do i = lbound (K,1) to hbound (K,1) ;
    K[i,j] = _K[i] ;
  end ;
end ;
```

PARALLEL KEY + DATA ARRAYS

Thus far, we have presented code for the scenario when all arrays involved in the sorting process are part of the sort key. However, whether the data collection representing the object of sorting is in the form of files or arrays, the most common scenario is sorting by key fields (single or composite) and permuting data fields in the same records (or array index positions) accordingly. In the bailiwick of array sorting, this kind of situation presents itself when we have one or more parallel *key arrays*, also parallel to one or more *data arrays*. The ideological intricacies of array sorting under such circumstances, along with the distinction between the stable and unstable sorting, were already discussed in detail in the section "Parallel Arrays". Now, however, we have to encapsulate those concepts in workable program code.

Since we already know how to extrapolate from a single key array to multiple parallel key arrays, let us return to the simple setting of one key array parallel to one data array. More complex cases of multiple parallel key arrays coupled with multiple parallel data arrays can be then worked out by induction.

Hence, let us consider a simple example of two parallel arrays:

```
array K [10] $1 ('D' 'A' 'C' 'D' 'C' 'B' 'D' 'B' 'C' 'D') ;
array D [10]    ( 0   4   7   1   8   5   2   6   9   3 ) ;
```

where K is the key array and D is the data array. The goal is to rearrange the array items into ascending order as follows:

```
K: A B B C C C D D D D
D: 4 5 6 7 8 9 0 1 2 3
```

Or, in the case of descending order:

```
K: D D D D C C C B B A
D: 0 1 2 3 7 8 9 5 6 4
```

Note that in both cases the original relative order of the D-items is preserved within every same-key group, i.e. we expect the sorting to be stable. Alternatively, we may want to discard the duplicate-key items from both arrays with two options: (1) for each same-key group, keep the K- and D-items with the *lowest* array index or (2) keep the items with the *highest* array index. If we agree that, after discarding the duplicates, the arrays need to be filled with missing values up front and assume ascending order, the picture we expect for option #1 would be:

```
K:             A B C D
D: . . . . . . 4 5 7 0
```

And for option #2, the expected picture would be:

```
K:             A B C D
D: . . . . . . 4 6 9 3
```

We already have all we need to achieve these goals. The only difference compared to sorting parallel *key* arrays is that only the items from the key array (or arrays if there are more than one) should be loaded into the key portion of the hash table, while the data portions should be reserved for the data array items only. With that kept in mind, let us express it in the SAS language:

```
data _null_ ;
  array K [10] $1 ('D' 'A' 'C' 'D' 'C' 'B' 'D' 'B' 'C' 'D') ;
  array D [10]    ( 0   4   7   1   8   5   2   6   9   3 ) ;
  dcl hash s (multidata:"Y", ordered:"A") ;
  s.defineKey  ("_K") ;
  s.defineData ("_K", "_D") ; /* _D is in the data portion only */
  s.defineDone () ;
  dcl hiter is ("s") ;
  /* Fill up sorted hash table from arrays */
  do i = lbound (K) to hbound (K) ;
    _K = K[i] ;
    _D = D[i] ;
     s.ADD() ;     /* Keep all array items - just sort  */
   * s.REF() ;     /* Keep the lowest-index duplicates  */
   * s.REPLACE() ; /* Keep the highest-index duplicates */
  end ;
  put "Unsorted: K: " K[*] / +10 "D: " D[*] ;
  /* Stream sorted hash data back to arrays */
  do i = lbound (K) to hbound (K) ;
    if i > hbound (K) - s.num_items then do ; /* fillers up front */
  * if i < lbound (K) + s.num_items then do ; /* fillers out back */
      is.NEXT() ;
      K[i] = _K ;
      D[i] = _D ;
    end ;
    else do ;
      K[i] = "" ;
      D[i] = . ;
    end ;
  end ;
  put "Sorted:   K: " K[*] / +10 "D: " D[*] ;
run ;
```

The program can be run with any combination of the following options:

- Sorting order: ORDERED:"A" for ascending and ORDERED:"D" - for descending.
- Do not discard any duplicates, just sort: Call the ADD method.
- Discard the duplicates and keep the lowest-index items: Call the REF method.
- Discard the duplicates and keep the highest-index items: Call the REPLACE method.
- Fill the discarded items with missing values up front or out back: Choose between the two IF statements.

Testing the program with all combinations of the above options shows that it works as expected in all cases. Again, if the ADD method is called and so no duplicates are discarded, the second DO loop can be simplified as:

```
do i = lbound (K) to hbound (K) ;
  is.NEXT() ;
  K[i] = _K ;
  D[i] = _D ;
end ;
```

If we have more than one key array and/or more than one data array, the program can be used as a template to add more program elements needed to accommodate the extra arrays by simple induction. We have already seen how it can be done in the case of multiple parallel key arrays.

HANDLING DUPLICATES: SPECIAL CASES

Above, we have discussed only two distinct cases of handling duplicate array values if only one duplicate for each key-value is to be kept: Either keep the lowest-index or highest-index occurrence. However, in a number of situations we need to choose which duplicate instance to keep not from these two options but rather based on a specific, data-driven condition.

For example, we may want, from all items with the given K-value, to select the item which:

- is the first-in with a specific D-value
- has the N-highest or N-lowest D-value
- is closest to the average of all D-values
- whatever else may strike one's fancy

It is even conceivable that, instead of choosing a specific D-value from each same-key item group, we may want to return some summary statistic, such as the average. In all cases of this nature, it is indispensable that we keep the MULTIDATA:"Y" argument tag, so that when the time comes to put the sorted data back in the arrays we will have, for each K-value, all the D-values on which to base our decision.

The step below presents a simple example of returning the average of all D-values for each same-key group of K-values:

```
data _null_ ;
  array K [10] $1 ('D' 'A' 'C' 'D' 'C' 'B' 'D' 'B' 'C' 'D') ;
  array D [10]    ( 0   4   7   1   8   5   2   6   9   3 ) ;
  dcl hash s (multidata:"Y") ;
  s.defineKey  ("_K") ;
  s.defineData ("_K", "_D") ; /* Note: _D is in the data portion only */
  s.defineDone () ;
  /* Hash U to store and enumerate unique K-values */
  dcl hash u (ordered:"A") ;
  u.defineKey ("_K") ;
  u.defineDone () ;
  dcl hiter iu ("u") ;
  do i = lbound (K) to hbound (K) ;
    _K = K[i] ;
    _D = D[i] ;
     s.ADD() ;
     u.REF() ;
  end ;
  do i = lbound (K) to hbound (K) ;
    if i > hbound (K) - u.num_items then do ;
      iu.next() ;
      _sum = 0 ;
      do _n = 0 by 1 while (s.DO_OVER() = 0) ;
        _sum + _D ;
      end ;
      K[i] = _K ;
      D[i] = divide (_sum, _n) ;
    end ;
    else call missing (K[i], D[i]) ;
  end ;
  put "Sorted: K: " K[*] / +8 "D: " D[*] ;
run ;
```

This is how the output printed in the SAS log looks:

```
Sorted: K:             A  B  C  D
        D: . . . . . . 4 5.5 8 1.5
```

The program works similarly to those shown above, with a few wrinkles:

- Auxiliary hash table U is used to store unique K-values only. Note that it is table U that is ordered, not table S.
- Table S if filled as usual by calling the ADD method, so all duplicates at this point are kept.
- Each unique K-value from table U is scanned using hash iterator IU. The K-values are accessed in the sorted order because table U is ordered.
- For each _K retrieved from U, the item group from table S with this key-value is scanned and the sum of all D-values and the number of items in the group are computed.
- The value of _K and the computed D-average are assigned to the corresponding index position of arrays K and D.

The program can serve as a template for selecting a specific duplicate item based on any other condition. For instance, if we wanted to return the highest D-value for each key-value group of K, we would recode the block of statements around and including the DO_OVER loop as follows:

```
 _max = . ;
do while (s.DO_OVER() = 0) ;
  _max = _max <> _D ;
end ;
K[i] = _K ;
D[i] = _max ;
```

With this change, the program would print in the log:

```
Sorted: K:             A B C D
        D: . . . . . . 4 6 9 3
```

ALTERNATING SORT ORDER

An alert reader has undoubtedly noticed that when we sort parallel key arrays, it is always either all ascending or all descending. It raises the question: Can we use the hash object to sort some component arrays ascending and the others - descending, as we can do when sorting files with the SORT or SQL procedures? The answer to this question is: No, we cannot do it *directly* because at least as of today, the hash object has no provision to specify opposite sort orders for different components of its composite key. It sorts the *entire key* either ascending or descending depending on the specification. But is there a way we can *trick* hash object into sorting using an alternating sort order?

NUMERIC ARRAYS

With numeric arrays, it is easy: Just multiply every array item loaded into the hash table by -1 and then multiply the hash variable by -1 before streaming it back to the array. For example, below, the composite key (AC[i],AN[i]) is sorted in this manner by AC[i] ascending and AN[i] descending within each same-key AC value group:

```
data _null_ ;
  array AC [10] $1 ('B' 'B' 'B' 'B' 'B' 'A' 'A' 'A' 'A' 'A') ;
  array AN [10]    ( 1   1   3   3   3   5   5   5   7   7 ) ;
  dcl hash s (multidata:"Y", ordered:"A") ;
  s.defineKey ("_AC", "_AN") ;
  s.defineDone () ;
```

```
  dcl hiter is ("s") ;
  do i = lbound (AC) to hbound (AC) ;
    _AC =   AC[i] ;
    _AN = - AN[i] ;
    s.ADD() ;
  end ;
  do i = lbound (AC) to hbound (AC) ;
    is.next() ;
    AC[i] =   _AC ;
    AN[i] = - _AN ;
  end ;
  put "AC: " AC[*] / "AN: " AN[*] ;
run ;
```

As evidenced by the SAS log, the result of sorting is, as requested:

```
AC: A A A A A B B B B B
AN: 7 7 5 5 5 3 3 3 1 1
```

CHARACTER ARRAYS

Although a more or less similar trick can be done, *in principle*, with character arrays as well, it cannot be done nearly as inexpensively. In a general case, to pull the feat, we would have to swap every character of each array element to its symmetrically opposite counterpart in the collating sequence en route from the array to the hash and then do the opposite on the way back. From the standpoint of computational cost, it is *extremely* expensive.

However, if the system length of the array characters does not exceed $6 (on ASCII systems, and $7 on EBCDIC systems), at least one inexpensive trick can be employed by making use of the PIB*w*. informat/format pair. This is because it is easy to use PIB*w*. to transform a character string - essentially a 256-radix number - to the equivalent decimal number. Then it can be multiplied by -1, sorted, multiplied by -1 again, converted back to the character type, and streamed back into the array. The length limitation is due to the fact that SAS numeric real binary variables cannot accurately store integers greater than 2**53 under ASCII and 2**56 under EBCDIC, which makes the said double conversion impossible to do accurately.

That having been said, as long as the array character length is within the limits stipulated above, alternating sorted order can be achieved almost as easily as with a numeric array. In the step below, character array AC is tricked into being sorted *descending* within each key-value of array AN sorted *ascending*:

```
data _null_ ;
  array AN [10]    ( 2   2   2   2   2   1   1   1   1   1 ) ;
  array AC [10] $1 ('A' 'A' 'A' 'B' 'B' 'C' 'C' 'D' 'D' 'D') ;
  dcl hash s (multidata:"Y", ordered:"A") ;
  s.defineKey ("_AN", "_AC") ;
  s.defineDone () ;
  dcl hiter is ("s") ;
  do i = lbound (AN) to hbound (AN) ;
    _AN =   AN[i] ;
    _AC = - input (AC[i], pib6.) ;
    s.ADD() ;
  end ;
  do i = lbound (AC) to hbound (AC) ;
    is.next() ;
    AN[i] = _AN ;
    AC[i] = put (- _AC, pib6.) ;
  end ;
  put "AN: " AN[*] / "AC: " AC[*] ;
run ;
```

The result of the sorting shown in the SAS log is, as requested:

```
AN: 1 1 1 1 1 2 2 2 2 2
AC: D D D C C B B A A A
```

Perhaps people smarter than me could ideate something that could perform this kind of trick easily and inexpensively against a character array with elements of any length all the way up to $32767. If so, I will be very curious to learn of their ideas.

MULTI-RECORD ARRAY SORTING

All sample programs presented so far have had to deal with a single instance of an array or collection of parallel arrays to be sorted. In the real data processing world, this kind of operation is usually needed to be done either for every input record or for each input BY group. In turn, it means the following:

- The hash object instance performing the action needs to be created and instantiated just once before the first incoming record is processed.
- Every time before a new set of array variables is sorted, the hash table has to be prepared to accept a fresh set of items without interfering with the items stored there earlier.

To clarify these points, let us assume that we have a data set with variables K1-K7 and D1-D7:

```
data unsorted ;
  input K1-K7 (D1-D7) (:$1.) ;
  format k: 3. ;
  cards ;
2 1 1 2 1 1 2  B A C B B C A
2 2 1 1 1 2 2  C C C C C B B
1 1 1 1 2 2 1  C C A A A B B
2 1 1 1 2 1 1  A C A C B C B
1 1 1 1 1 1 1  A C B A A A C
run ;
```

Let us also suppose that and we need to sort them in each record ascending using the values of K1-K7 as key-values and permuting the values of D1-D7 accordingly without worrying about handling duplicates. In other words, the expected result of *stable* sorting should look as follows (I indicates the array index):

```
I: 1 2 3 4 5 6 7   I: 1 2 3 4 5 6 7
---------------   ---------------
K: 1 1 1 1 2 2 2   D: A C B C B B A
K: 1 1 1 2 2 2 2   D: C C C C C B B
K: 1 1 1 1 1 2 2   D: C C A A B A B
K: 1 1 1 1 1 2 2   D: C A C C B A B
K: 1 1 1 1 1 1 1   D: A C B A A A C
```

To attain that, we have to execute the same "array-to-hash, hash-to-array" algorithm we have expounded upon in detail before. However, we also have to add a couple of twists to conform with the points in the bulleted list above; below, they are shown in bold:

```
data sorted (keep = K: D:) ;
  if _n_ = 1 then do ;
    dcl hash s (multidata:"Y", ordered:"A") ;
    s.defineKey  ("_K") ;
    s.defineData ("_K", "_D") ;
    s.defineDone () ;
    dcl hiter is ("s") ;
  end ;
  set unsorted ;
  array K [*] K: ;
```

```
    array D [*] D: ;
    do i = lbound (K) to hbound (K) ;
      _K = K[i] ;
      _D = D[i] ;
      s.ADD() ;
    end ;
    do i = lbound (K) by 1 while (is.next() = 0) ;
      K[i] = _K ;
      D[i] = _D ;
    end ;
    s.CLEAR() ;
  run ;
```

Thus, the block of code creating and instantiating the hash object instance is executed only *once* before the first record is read in. It is paramount because if we let it execute unconditionally, a new empty hash object instance would be created for every new observation. Though *this* program would still work, with many more observations its performance would quickly become unbearable for two reasons: (1) creating a new instance is relatively costly and (2) each instance of a hash object defined as above takes about 135 KB of memory. Hence, with mere 1 million observations, memory usage would exceed 128 GB.

Therefore, it makes a lot more sense to have just a single hash object instance and empty it out by using the CLEAR method, as shown above, before each new act of sorting. It keeps the memory footprint at paltry 135 KB of RAM regardless of the number of input records.

TEMPORARY ARRAYS

Throughout this paper, the examples illustrating the abilities of the SAS hash object to sort arrays have been based on arrays incorporating PDV variables. However, every bit of functionality presented here also works for temporary arrays with no need to alter the sample programs, save for the PUT statement array references like K[*] used for demo purposes.

CONCLUSION

The SAS hash object has a rich set of facilities for sorting data and can be easily adapted to sorting arrays regardless of their data type in a variety of ways. Though certainly not as fast as the dedicated SORTN and SORTC routines, it vastly exceeds them in functionality, such as the ability to sort single, parallel, and multi-dimensional arrays, both in ascending and descending order (and, with a degree of ingenuity, in alternating order as well), and handle duplicates. SAS programmers whose program specifications call for horizontal sorting should definitely include this functionality in their programming arsenal.

REFERENCES

Dorfman, P. 2001. "QuickSorting An Array". *Proceedings of SUGI 26*, Long Beach, CA. Available at http://www2.sas.com/proceedings/sugi26/p096-26.pdf.

Dorfman, P. and Henderson, D. 2018. "Data Management Solutions Using SAS Hash Table Operations. A Business Intelligence Case Study." Cary, NC: SAS Insitute Inc.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Paul M. Dorfman
(904) 260-6509
sashole@gmail.com