Paper 96-26

QuickSorting An Array

Paul M. Dorfman
CitiCorp AT&T Universal Card
Jacksonville, FL

ABSTRACT

The article is a walk through a DATA step implementation of one of the most versatile, fast, and well-rounded sorting algorithms - Quicksort. Although a need in array sorting frequently arises in practical SAS programming, Base SAS does not provide a function or call routine to serve the purpose. However, the SAS Language is flexible and powerful enough to implement just about any algorithm. In this paper, Quicksort is presented as an encapsulated macro routine with simple invocation rules. Compared to a similar routine presented at SUGI 24, the one given below has been recoded from ground up. As a result, it has become more robust, richer on features, and about twice as efficient.

INTRODUCTION

When a SAS array needs to be sorted, two paths can be taken:

1. One way or another, make use of PROC SORT.
2. Code a custom DATA step sorting routine.

The first method is used most often, because it is known to get the job done. Also, if we only need to create a sorted array from a file in the beginning of a DATA step, it is almost perfect. It is much less perfect, though, if the array resides in the middle of the DATA step or has to be sorted in each observation. In such cases, the array and the rest of the data being processed have to be saved; after sorting is done, another step is needed to reorganize the data, restore status quo, and resume processing.

Let us imagine, for example, that a SAS data file WORK.A contains numeric variables A1-A200, and we need to sort the variables into ascending order in *each observation* (for instance, to rank the A-values or another purpose). To make use of PROC SORT, we have to reshape the data in the vertical direction, sort the result, and transpose the data back:

```
Data V (Keep=Id V);
   Array A(200);
   Set A;
   Id ++ 1;
   Do j=1 To Dim(A);
      V = A(j); Output;
   End;
Run;
Proc Sort Data=V; By Id V; Run;
Data A (Keep=A:);
   Array A(200);
   Do J=1 by 1 Until (Last.Id);
      Set V;
      By Id;
      A(J) = V;
   End;
Run;
```

Thus, we end up with three steps, each rewriting the same data, and hence with a lot of extra I/O traffic, all for the sake of making PROC SORT work for us. And then, is does not appear logical to program this way: The array elements already reside in memory, which is exactly where any sorting process invariably places them at some point in time. So, why should we write it to an intermediate file even once? The stream of the consciousness begs for the straightforward solution:

```
Data A (Keep=A1-A200);
   Array A(200);
   Set A;
  <Sort array A>;
Run;
```

Read an observation - sort the array A in place - write the observation: Pure and simple.

The problem is, no statement or function that would <Sort array A>, in place or otherwise, is provided as part of Base SAS. Situations like this have resulted in the question "How to I sort an array in place?" or something of its kind frequently asked on SAS-L. Solutions proposed by respondents are usually confined to (a) using PROC SORT more or less as described above, (b) ordering the array in place with a simple bubble or insertion sort, or (c) using the ORDINAL function to insert the first smallest, second smallest, and so on, array items into an auxiliary array one by one.

Unfortunately, these simple algorithms work satisfactorily only when the array dimension N is rather small: Since their sorting time grows as N*N, at moderately large N (several dozen items) their performance becomes intolerable. For instance, if bubble or insertion sort were used in the step above, and A had mere 10,000 records, it would run, in all practicality, "forever", to say nothing of the ORDINAL-based sort, which is the slowest of all and, besides, can be applied to numeric arrays only.

Programming more efficient array-sorting schemes in the DATA step is rarely attempted. This must be rooted more in the tradition of conventional SAS usage than SAS Language per se. Surely well-performing algorithms are more tricky than the simple schemes and therefore more difficult to program and debug. However, the good news is that it has to be done once, and the time spent on implementing and tuning a decent array-sorting routine may save plenty of development and machine time in the future.

Quicksort, invented in the sixties by C.A.R. Hoare [1, 2], is one of all-around fastest and resource-efficient comparison-based sorting techniques. It is good for several reasons:

1. It is comparison-based, so it is insensitive to the data type of the array.
2. Sorting is performed practically in place.
3. As its name implies, it is pretty quick. Among methods sorting in O(N*log2(N)) time, Quicksort is among the fastest.
4. Its internal bookkeeping requires practically negligible auxiliary memory (about 100 numeric entries, i.e. 1 Kb).

In this paper, Quicksort is briefly described as an algorithm. Then it is implemented in its basic DATA step form as a LINK subroutine accompanied by detailed comments. Furthermore, the code is encapsulated in a more feature-laden and user-friendly macro routine %Qsort(). Finally, a brief tutorial of %Qsort() usage is presented.

1. THE ALGORITHM

Quicksort is a so-called partition-exchange scheme. This is how it works. Suppose that variables L and H hold the lowest and highest array indices.

1. Take one element called "pivot" and determine its final position in the sorted array, say, J. Assuming ascending order, this will occur when A(J) is not less than any item to the left of it, and not greater than any item to the right. Thus, we have one element in its proper position and two unsorted partitions. Therefore, the original sorting problem is reduced to two simpler problems: to independently sort partition A(L)...A(J-1) and partition A(J+1)...A(H).
2. Apply this technique to each of the unsorted partitions. Every time step 1 is performed, another element is placed into its final position, and two unsorted partitions are formed. When no partition consists of more than one element, the array is apparently sorted.

It seems simple enough, but how do we find to what location in the sorted array should the pivot belong? Most of the existing partitioning methods exploits the following sequence:

1. Use pointer I to scan A from left to right. When an item greater than the pivot is found, stop.
2. Use pointer J to scan A from right to left, and stop at the element less than pivot.
3. If I is still to the left of J (I < J), the items A(I) and A(J) are out of order, so exchange them.
4. Repeat the process until I and J meet or overlap (the pointers "cross"), at which point the partitioning is complete.

This scheme forms the core of the algorithm and ultimately determines its speed, so it must be optimized as thoroughly as possible. So, before start coding, a couple of think-ahead points should be made.

Firstly, a divide-and-conquer paradigm might be intuitively expected to complete the partitioning process in O(logN) time. But certain input arrangements can cause it to run quite poorly! For example, it can happen if in each partition, the pivot turns out to be the maximum (or minimum) element. Thus, after partitioning, one partition will contain the pivot itself, and the other one – the rest of the items, so the size of the partition being divided will have been reduced by only a single element. In this case, Quicksort will run in O(N**2) time, i.e. no better than bubble or insertion sort. In practice, this situation is encountered when the array A is already sorted ascending, and the lowest element is chosen as a pivot.

Secondly, a way must be found to prevent the indices I and J from running out of the array bounds: If the pivot is the smallest item in the partition being divided, J will run past lower bound of A seeking an element greater than or equal to the pivot; or if the pivot is the greatest item, the same will happen to index I, only at the upper bound. Using comparisons to check for boundary conditions is not a good idea, for both theory and experiment have shown that adding a single extra instruction to the inner loop is very likely to cause Quicksort's performance to plummet, no matter how insignificant the additional instruction might seem to be.

Fortunately, there exists a trick capable of resolving both problems at once:

1. Split the partition into a small number N of equal chunks and take the A(L), A(H), and the rest of the items on the chunk boundaries. Rearrange them into sort order using explicit comparisons and swapping the items if necessary. Select one of the middle items (e.g., A(I)) as the pivot.
2. Swap A(I) and A(L). Now the lowest-index item is equal to the pivot, and the highest-index item is greater than or equal to it. Thus, now A(L) and A(H) can serve as natural sentinels stopping the partitioning pointers at the ends of the partition.
3. Partition the array section between (L+1)-th and R-th elements. At the end, index J will point to the node where the pivot shall finally reside.
4. Exchange the first item (now holding the pivot value) and the middle item. Now the entire partition between L-th and R-th elements is properly subdivided, with A(J) being the boundary between the two new partitions (and, of course, the J-th smallest array element).

This simple procedure achieves two important satellite effects: It makes it exponentially improbable for any particular input to elicit the worst-case behavior, and promotes well-balanced partitioning. To make things more practical, let us see how it works in concert against a dozen of sample keys. For the sake of simplicity, let us select N=2 chunks, thus having only one middle item, in the center of the partition. Below, the lowest-, middle-, and highest-index elements are shown in the boldface:

```
 01  02  03  04  05  06  07  08  09  10  11  12
------------------------------------------------
370 941 800 519 844 939 086 064 100 134 639 048
```

According to the plan, the three items are rearranged into ascending order "by hand":

```
048 941 800 519 844 370 086 064 100 134 639 939
```

Next, we choose the middle item A(6)=370 as a pivot, swap it with A(1)=048, and then assign the pivot value to a variable P:

```
370 941 800 519 844 048 086 064 100 134 639 939
```

Now we can partition the items A(2)--A(12). Scanning the array with index I from the left, we find that at I=2, A(I) > P: Stop. Scanning from the right with index J, we find that at J=10, A(J)<P: Stop. Since I<J, we switch A(I) and A(J):

```
370 134 800 519 844 048 086 064 100 941 639 939
```

A similar situation occurs again at (I=3, J=9), (I=4, J=8), and finally at (I=3, J=7):

```
370 134 100 519 844 048 086 064 800 941 639 939
370 134 100 064 844 048 086 519 800 941 639 939
370 134 100 064 086 048 844 519 800 941 639 939
```

The next time I is increased by 1 and J is decreased by 1, both converge at I=J=6, so the pointers have crossed, and the A(2)--A(12) partitioning is complete. Since J=6, it is where the pivot (it has been sitting at A(1) all along) must reside in the future sorted array. So, we complete the process by exchanging A(1) and A(6), thus moving the pivot where it belongs:

```
048 134 100 064 086 370 844 519 800 941 639 939
```

Since max(of A1-A5)<= A(6), and min(of A7-A12)=>A(6), the element A(6)=370 is the 6-th smallest, i.e. it rests in its final sorted location, no matter how disordered the two newly formed partitions A1-A5 and A7-A12 might be by themselves.

Now all we have to do is apply the same process to the newly formed partitions. But it is only possible to work on one partition at a time. We can solve this problem by using a small auxiliary array as a stack to hold the end pointers of the partitions whose processing has been postponed. Let us push the end pointers (J+1, R) defining the longer partition A7-A12 onto the stack and commence on dividing the smaller partition A1-A5. (Processing shorter partitions first guarantees that no more that log2(dim(A)) stack items will be needed. That is, a 50-element stack will suffice for sorting an array with 2**50>1E15 elements.) Each time two new partitions are

formed, the longer of the two goes onto the stack, and the shorter one is divided further. If a partition ends up consisting of a single element, it has fallen in its final place. Now we pop the most recently stored end pointers off the stack and subdivide the partition within their bounds. The entire process is repeated until the stack has been exhausted. Once it has happened, the last postponed partition has been processed, and therefore the entire array is in order. In the SAS log fragment below, the partition that has just been split is underlined, the pivot is shown in the italics, and the elements in their final "sorted" locations are shown in the boldface:

```
 01  02  03  04  05  06  07  08  09  10  11  12
-----------------------------------------------
370 941 800 519 844 939 086 064 100 134 639 048  Unsorted
-----------------------------------------------
048 134 100 064 086 370 844 519 800 941 639 939  L=01 J=06 H=12

048 064 086 134 100 370 844 519 800 941 639 939  L=01 J=03 H=05

048 064 086 134 100 370 844 519 800 941 639 939  L=01 J=01 H=02

048 064 086 100 134 370 844 519 800 941 639 939  L=04 J=04 H=05

048 064 086 100 134 370 639 519 800 844 941 939  L=07 J=10 H=12

048 064 086 100 134 370 639 519 800 844 939 941  L=11 J=11 H=12

048 064 086 100 134 370 519 639 800 844 939 941  L=07 J=08 H=09

048 064 086 100 134 370 519 639 800 844 939 941  Sorted
```

At last, it is actually more efficient to stop the process of quicksorting when none of the partitions contains M>1 elements, rather than M=1, and then finish the sorting by a single pass of straight insertion sort. Insertion sort performs poorly against a large disordered array, but it excels if the array is well preordered. After any number of partitioning steps, all items located between partitions are already in their final places since they have served as pivots. Besides, within any given partition, no element is greater than any element in a higher partition because of the nature of the algorithm. When all partitions are small, very few inversions remain, which makes insertion sort very fast. The optimum value of M is 9, but it can be retained as a tuning parameter.

2. THE "BASE" DATA STEP CODE

At this point, it should take much less than a Certified SAS Programmer to implement a correct SAS version of iterative Quicksort. But just before beginning to type away, let us observe that over the course of the algorithm, array items are frequently exchanged. By encapsulating the simple operation of swapping two array items A(I) and A(J) in a macro called as %Sw(I,J), we can avoid repetitive code and make the program more transparent. The variable T is used as an intermediate memory location, and it automatically assumes the same data type (numeric or character) and length as the data type and expression length of the array elements.

```
01    %Macro Sw (I,J);
02       Do; T = A(&I); A(&I) = A(&J); A(&J) = T; End;
03    %Mend Sw;
```

Now, it is time to code. In the DATA step below, Quicksort organized as a LINK subroutine orders a large array A into ascending sequence. Then a check is performed, just to make sure that the array is, indeed, sorted, and that it contains exactly the same data as the original (this is what array F is for). If these tests pass OK, the SAS log gets the messages.

```
04 %Let Seq = %Str(>); * (<) For Descending;
05 Data _Null_;
06    Array A (-123456:123456) _Temporary_; *Array To Sort;
07    Do J=Lbound(A) To Hbound(A);
08       A(J) = -2e5 + Ceil(Ranuni(1)*4e5);
09    End;
10    Lb = Lbound(A); Hb = Hbound(A);
11    Array F (-200000:200000) _Temporary_; *Frequency Array;
12    Do J=Lb To Hb; F(A(J)) ++ 1; End;
13
14    Link Qsort;
15
16    Do J=Lb To Hb-1 Until(A(J) &Seq A(J+1)); End;
17    If J = Hb Then Put 'Array Is Sorted.';
18    Do J=Lb To Hb; F(A(J)) +- 1; End;
19    Do J=Lbound(F) To Hbound(F) Until (F(J)) ; End;
20    If J = Hbound(F)+1 Then Put 'Data Are Valid.';
21 Stop;
22 Qsort:
23    Array Z (0:1,0:50) _Temporary_;
24    H = Hb; L = Lb; M = 9;
25    If H-L > M Then Do S=1 By 0 While (S);
26       J = (H-L)/3; I = L+J; J = I+J;
27       If A(L) &Seq A(I) Then %Sw(L,I);
28       If A(I) &Seq A(J) Then %Sw(I,J);
29       If A(J) &Seq A(H) Then %Sw(J,H);
30       If A(L) &Seq A(I) Then %Sw(L,I);
31       If A(I) &Seq A(J) Then %Sw(I,J);
32       If A(L) &Seq A(I) Then %Sw(L,I);
33       If H-L <= 3 Then Do;
34          L = Z(0,S); H = Z(1,S); S +- 1; Continue;
35       End;
36       %Sw(L,I); P = A(L); I = L;
37       Do J=H+1 By 0;
38          Do I=I+1 By +1 Until (A(I) =&Seq   P ); End;
39          Do J=J-1 By -1 Until (  P  =&Seq A(J)); End;
40          If I => J Then Leave;
41          %Sw(I,J);
42       End;
43       %Sw(L,J);
44       If      H-J => J-L > M Then Do S=S+1;
45          Z(0,S) = J+1; Z(1,S) = H; H = J-1;
46       End;
47       Else If J-L => H-J > M Then Do S=S+1;
48          Z(1,S) = J-1; Z(0,S) = L; L = J+1;
49       End;
50       Else If J-L > M => H-J Then H = J-1;
51       Else If H-J > M => J-L Then L = J+1;
52       Else Do;
53          L = Z(0,S); H = Z(1,S); S +- 1;
54       End;
55    End;
56    If M > 1 Then Do J=Lb+1 To Hb;
57       If A(J-1) &Seq A(J) Then Do;
58          P = A(J);
59          Do I=J-1 To Lb By -1;
60             If P =&Seq A(I) Then Leave;
61             A(I+1) = A(I);
62          End;
63          A(I+1) = P;
64       End;
65    End;
66 Run;
```

So, the entire honest-to-goodness iterative implementation of the Quicksort algorithm boils down to about 50 or so lines of SAS code. Not bad for the language held by some in low esteem as "unsuitable" for such holy a purpose! Comments would make the correspondence between the algorithm and code more transparent; however, in order to avoid polluting the program and be more descriptive, the comments are rather given below, line by line (LL).

LL 04-04: Some sane way is needed select a sort order without rummaging through the program and looking which inequalities to reverse (not all of them, that is for sure). Doing this by means of a global macro variable is perhaps not the worst way.

LL 06-06: To test the contraption, we need an array to sort, and the array has to be large, otherwise the difference between slow and fast sorting methods will be blurred. Negative and zero indices are included just to make sure the algorithm does not care about their sign or lack of it thereof.

LL 07-09: For the sake of testing, the array is populated with integers in [-200000:200000] range. Choosing such a range helps test differently signed array items and, at the same time, makes it possible to use another array to rapidly validate the sorted data (see LL 11-12 below).

LL 10-10: Through the two variables LB and HB, one can specify a particular array section to sort. Here, the entire array is being sorted.

LL 11-12: Even if after sorting, the array is sorted, it does not mean it contains the same data as the original object! We need to be able to validate that it does. In the above program it is done by key-indexing all keys from array A into an auxiliary array F and incrementing the contents of a bucket K each time a key equal to K is found. Thus, the K-th bucket holds the frequency of array items equal to K. Subsequently, the sorted array can be checked against F in a similar manner (see LL 18-20).

LL 14-14: Call on Qsort LINK module to sort array A.

LL 16-17: Check if array A is now sorted by examining the relative order of each consecutive pair of elements.

LL 18-20: Confirm that after the sorting, array A contains exactly the same data as before. To do so, key-index array F by the elements of array A again, now subtracting a unity from a bucket K each time a key whose value equals K is encountered. If at the end of this process, all F-buckets are empty, then for each possible K, N items equal to K in the original array have cancelled exactly N items in the sorted array, and hence the arrays may only differ by the order of their keys.

LL 23-23: Define an auxiliary array Z(0:1,0:50) to be used as a stack for the end pointers of yet-to-be-processed partitions. Z(0,s) will hold the left end pointer of the most recently postponed partition; and Z(1,s) will hold the right one.

LL 24-24: Initially, set the end pointers of the current partition to those of the entire section being sorted. Usually they default to the array bounds, but in some special cases, one may want to sort a selected array section leaving the rest of the array intact. If this is the case, it should be done before calling on Qsort LINK subroutine. Also, define the smallest partition for quicksort to process by setting M to a positive integer value from. Quicksort ignores all partitions shorter than M and passes them to insertion sort to handle. It has been found both theoretically and experimentally that the optimum value is M=9. (It might vary approximately from 5 to 11 under different operating systems, with almost negligible effect on the speed. When tuning the routine, it is useful to set M=1 temporarily, since otherwise the behavior of Quicksort is masked by the finishing insertion sort).

LL 25-25: If the size of the array section selected for sorting does not exceed the optimum partition size M, pass it to the straight insertion sort at once. Otherwise iterate through the body of the DO loop (executing Quicksort per se) while the stack Z has not yet been exhausted (at which time the stack pointer S=0).

LL 26-32: Perform the pre-partition procedure described in the algorithmic section above. In this "production" version, the current partition is split in 3 equal chunks with 4 boundary elements with indices L<I<J<H. The four items are sorted in place by explicitly comparing and swapping them if they are out of order.

LL 33-35: If M<=4, for instance, M=1, the current partition may have as few as 2 to 4 items. The process at LL 26-32 thus has them sorted, so there is no need to fuzz partitioning them, and the next partition to be processed can be simply taken off the stack.

LL 36-36. One of the median items (here the one in the lower location I) is selected as a pivot and switched with the leftmost element, while the pivot value is assigned to a "register" P. We are going to divide the partition between its second and last elements, (L+1, H), so initially the inner loop (see below) pointers I and J are set to the left of L+1 and to the right of H.

LL 37-42: This is the Inner Loop – the heart of Quicksort. It partitions the section (L+1, H), with J finally pointing to the array node where the pivot must reside in the sorted array.

LL 43-43: The pivot is returned to its proper location. At this point, partitioning of the entire subsection (L,H) is complete, and two new subpartitions (L,J-1) and (J+1,H) have been formed. All the elements A(L)—A(J-1) are less or equal to A(J); and all the elements A(J+1)—A(H) are greater or equal to A(J).

LL 44-55: Do the bookkeeping - decide which partition to process next. If both partitions are longer than M, push the end pointers of the longer partition onto the stack, and go on to process the shorter one. If only one of the partitions is longer than M, process it and ignore the other one. If neither partition is longer than M, pop the most recent end pointers off the stack and commence on working on the partition lying between these end pointers.

LL 56-65: The Quicksort algorithm, per se, is finished. Array A now consists of partitions no longer than M; and for any two given partitions, none of the elements in the lower partition exceeds any element in the higher partition. If M=1, the array is sorted, so the algorithm should be terminated. If M>1 (for instance, M=9), no more than M elements anywhere in the array are out of order. This situation is thus ideally suited for an application of insertion sort, and so it finishes the job.

Should one like to get a feel how much quicker the Quicksort routine really is compared to "simple" sorting methods (like stand-alone insertion or bubble sort), just reduce the dimension of the array A to mere 10,000 elements and run the program. Then comment out the lines 25-55, thus leaving insertion sort to do the job alone. Run the program again, and compare the run-times. Warning: In lieu of a lot of time and patience, running stand-alone insertion sort against substantially more than 10,000 elements is not recommended.

3. %QSORT() MACRO ROUTINE

Of course, Quicksort can be used "straight up" either in the form of a LINK subroutine shown above or code embedded in the DATA step where needed. However, we can do better than that by encapsulating the module in a macro, parameterized to accommodate additional functionality, namely:

1. It should allow for sorting an array into both ascending (default) and descending sequences. We already know from the above code that only the inequalities containing &SEQ reference must be reversed.
2. If should be able to permute the elements of several parallel arrays accordingly when one of these arrays (the KEY array) is being sorted.
3. Just like the LINK routine, it should allow for specifying the boundaries of an array section to be sorted. These boundaries should naturally default to the lower and upper bounds of the array.
4. The internal DATA step variables used in the routine should be given names making the possibility of their clashing with any other variable name that might exist in the calling DATA step infinitesimally small, no matter how many times %Qsort() is called. All such variables must be dropped automatically.

With these guidelines in mind, it is not difficult to concoct the needed macro routine. But first it is necessary to upgrade the auxiliary macro %Sw() in order to enable the parameter ARR= in the macro %Qsort() (see below) accept a list of parallel arrays rather than a single array. That is, %Sw(I,J) should accept the indices of a pair of array elements and interchange

the corresponding items in all parallel arrays. Below, the macro %Sw() is incorporated in the macro %Qsort() as a nested macro definition. Although with such an arrangement, the inner macro has to compile each time the shell macro is compiled, its negative performance effect is negligible. On the positive side, it has the advantage of encapsulating the entire routine better.

```
%Macro Qsort (
   Arr =                        /* Parallel array name list */
  ,By  = %QScan(&Arr,1,%Str( )) /* Key array name           */
  ,Seq = A                      /* Seq=D for descending     */
  ,LB  = Lbound(&By)            /* Lower bound to sort      */
  ,HB  = Hbound(&By)            /* Upper bound to sort      */
  ,M   = 9                      /* Tuning range: (1:15)     */
   );
   %Local _ H I J L N P Q S T W;

   %Macro Sw (I,J);
      %Local W;
      Do;
      %Do W = 1 %To &N;
         &T&W      = &&A&W(&I);
         &&A&W(&I) = &&A&W(&J);
         &&A&W(&J) =  &T&W    ;
      %End;
      End;
   %Mend Sw;

   %If %Upcase(&Seq) = %Upcase(A) %Then %Let Q = G;
   %Else                               %Let Q = L;

   %Do %Until (&&A&N EQ );
      %Let N  = %Eval(&N+1);
      %Local A&N;
      %Let A&N = %Scan(&Arr,&N,%Str( ));
   %End;
   %Let N = %Eval(&N-1);

   %Let _ = %Substr(%Sysfunc(Ranuni(0)),3,
      %Eval(7-%Length(&N)+5*(%Substr(&Sysver,1,1) GT 6)));

   %Let H = H&_; %Let I = I&_; %Let J = J&_; %Let L = L&_;
   %Let P = P&_; %Let S = S&_; %Let T = T&_; %Let Z = Z&_;

   Array &Z (0:1, 0:50) _Temporary_;
   &L = &LB; &H = &HB;

    If &H-&L GT &M Then Do &S=1 By 0 While (&S);
      &J = (&H-&L)/3; &I = &L+&J; &J = &I+&J;
      If &By(&L) &Q.T &By(&I) Then %Sw(&L,&I);
      If &By(&I) &Q.T &By(&J) Then %Sw(&I,&J);
      If &By(&J) &Q.T &By(&H) Then %Sw(&J,&H);
      If &By(&L) &Q.T &By(&I) Then %Sw(&L,&I);
      If &By(&I) &Q.T &By(&J) Then %Sw(&I,&J);
      If &By(&L) &Q.T &By(&I) Then %Sw(&L,&I);

     %If &M LE 3 %Then %Do;
      If &H-&L LE 3 Then Do;
         &L = &Z(0,&S); &H = &Z(1,&S); &S+-1;
         Continue;
      End;
     %End;

     %Sw(&L,&I); &P = &By(&L); &I = &L;
      Do &J=&H+1 By 0;
         Do &I=&I+1 By +1 Until(&By(&I) &Q.E &P); End;
         Do &J=&J-1 By -1 Until(&P &Q.E &By(&J)); End;
         If &I GE &J Then Leave;
         %Sw(&I,&J);
      End;
     %Sw(&L,&J);

      If      &H-&J GE &J-&L GT &M Then Do &S = &S+1;
         &Z(0,&S) = &J+1; &Z(1,&S) = &H; &H = &J-1;
      End;
      Else If &J-&L GE &H-&J GT &M Then Do &S = &S+1;
         &Z(0,&S) = &L; &Z(1,&S) = &J-1; &L = &J+1;
      End;
      Else If &J-&L GT &M GE &H-&J Then &H = &J-1;
      Else If &H-&J GT &M GE &J-&L Then &L = &J+1;
      Else Do;
         &L = &Z(0,&S); &H = &Z(1,&S); &S+-1;
      End;
   End;
```

The parameterization of %Qsort() is rather simple:

ARR=: Specify the name of array to be sorted, or a blank-separated name list of several parallel arrays.

BY=: If we have a list of parallel arrays, say Arr=A B C, and want to sort, for instance, array B, and rearrange the elements of A and C accordingly, then B is considered the key array and its name must be supplied as BY=B. If BY= is left null, the first array in the list is considered the key array by default. In this case, it would be equivalent to specifying BY=A. If the list consists of a single array, BY= assumes its name by default.

SEQ=: Sort order. If left null, SEQ=A (ascending) order is assumed. Any other value will result in descending order, but in this coding SEQ=D is recommended. The case of D is irrelevant.

LB=, HB=: Most of the time, it is desired to sort the entire array, but should it be needed to sort just a specific array section, LB= and HB= can be supplied the corresponding values as hard coded numbers, variables holding the necessary index values, or numeric expressions.

M=: This is a tuning parameter specifying the smallest partition size after which the Quicksort algorithm ignores a partition and leaves it to insertion sort to finish the job (see the algorithmic section above). By default, M=9, as the optimum theoretical and experimental value. If M=1, the macro assembles no insertion sort at all, and the pure Quicksort scheme is executed.

4. A QUICK %QSORT() TUTORIAL

To see what kind of things %Qsort() can do and how to call it to do them, let us consider three parallel arrays A, B, and C:

```
Array A (10)    ( 9   8   7   6   5   4   3   2   1   0 );
Array B (10) $  ('A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J');
Array C (10)    ( 0   1   2   3   4   5   6   7   8   9 );
```

Let us begin with the simplest operation.

1. Sorting a Single Array

The simplest - and by far the most common - task is to sort a single array into ascending order. Let us, for example, choose to bring array A into order:

```
%Qsort (Arr=A);
```

That is all it takes. SEQ=A, LB=1, HB=10, BY=A are all assumed as the default values. Sorting a single array descending is equally simple. For example, in the case of array B:

```
%Qsort (Arr=B, Seq=D);
```

If we want to do the same, but only for the elements B1-B8, the macro will have to be called as follows:

```
%Qsort (Arr=B, Seq=D, LB=1, HB=8);
```

Or, if the bounds 1 and 8 were stored in variables Low and High, we could tell the macro to accept them instead of the hard-coded values:

```
%Qsort (Arr=B, Seq=D, LB=Low, HB=High);
```

Actually, in place of Low and/or High, one can use any valid numeric expression yielding a valid index value.

2. Sorting Parallel Arrays

Now imagine that the elements of array C are considered as "keys", and the elements of arrays A and B are viewed as satellite "fields". Thus a set of array elements A(J), B(J), C(J)

having the same index J represents a "record" in memory with C(J) as the key. For example, visually speaking, the fields of the 5-th record are located on the vertical line drawn through all parallel arrays where index J=5. We want to sort the entire memory "file" by the values of key items C descending. That is, as the elements of C are permuted to place them in order, the satellite elements A and B should be rearranged accordingly. To do so, we should assign the list of array names to the parameter ARR=, and tell the macro through the parameter BY=, which array is to be viewed as the key array:

```
%Qsort (Arr=A B C, By=C, Seq=D);
```

To demonstrate how this works, the contents of the arrays above were contrived by populating them in the opposite-reverse-order manner. As a result, sorting of the entire (A B C) company by C must have the order of C, as well as the order of the parallel arrays A and B, reversed:

```
2   data _null_;
3     array a(10)    ( 9   8   7   6   5   4   3   2   1   0 );
4     array b(10) $  ('a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j');
5     array c(10)    ( 0   1   2   3   4   5   6   7   8   9 );
6    %Qsort (Arr=A B C, By=C, Seq=D);
7     put A(*) / B(*) / C(*);
8   run;

0 1 2 3 4 5 6 7 8 9
j i h g f e d c b a
9 8 7 6 5 4 3 2 1 0
```

Alternatively, the same effect can be achieved by coding the array list with C first, without even specifying BY=C, because the first array in the list becomes the key array if the BY= parameter is left null. As far as the satellite sequence goes, that is, whether it is B A or A B, it is immaterial, for it only determines, in which array, A or B, the corresponding pair of elements is swapped first at the time when the algorithm needs to perform the exchange.

```
%Qsort (Arr=C B A, Seq=D);
```

Note that the only limitation imposed on the parallel arrays whose list is submitted to the macro for sorting is the array equivalence. In other words, the indices of all arrays comprising the list must be valid and synchronized throughout the range defined by the values of LB= and HB=. Since the latter default to the lower and upper bounds of the key array (i.e. either the one specified to BY= parameter, or the first on the list if the parameter is left null), it means that if LB= and HB= are left null, all non-key arrays must have an index range equal to or wider than the index range of the key array. Of course, for any non-key array, the processing will be limited to the key array index range. In other words, if a key array were declared as X [-50:100], and a satellite array - as Y[-100:200], then only the elements Y[-50:100] of the non-key array would be permuted, the rest remaining intact.

Other than that, the arrays on the list can be of different data types (character, numeric), different storage types ("real", i.e. pointing to DATA step variables, or temporary), or even different reference types (explicit, implicit). For instance, one might consider the following wild array of arrays:

```
Data A;
   Array X (1:5)   _Temporary_ ( 1   2   3   4   5 );
   Array V (*  ) $  A B C D E  ('A' 'B' 'C' 'D' 'E');
   Array Y          Q1-Q5      ( 0  -2  -4  -6  -8 );
   Array Z (  5) $             ('F' 'G' 'H' 'I' 'J');
   Array U (*  ) 8  U0-U4      (9.9 8.8 7.7 6.6 5.5);
   Array W       $  W3-W7      ('Z' 'Y' 'X' 'W' 'V');

   %Qsort (Arr=U V W X Y Z, By=Z, Seq=D);

   Put V(*)/Y(*)/Z(*)/U(*)/W(*);
Run;
```

Those inclined to experimenting might try to replace the BY=Z specification with any of the array names in the list, leave it out, change the sort order, etc., and observe the SAS log.

5. BACK TO THE FUTURE

Now, we can return to the original task of ordering an array of variables in each observation mentioned in the beginning of the paper with. Now we have got something to put in place of the hypothetical <Sort array A> routine:

```
Data A (Keep=A1-A200);
   Array A (200);
   Set A;
   %Qsort (Arr=A);
Run;
```

With 10,000 observations in the data set A, this step executes in about half the time required by the "standard" transpose-PROC SORT-transpose approach. Maybe more importantly, from the standpoint of programming logic, coding this way appears more natural and straightforward.

CONCLUSION

Efficient sorting of arrays can be done without resorting to PROC SORT associated with the need of crossing DATA step boundaries. SAS DATA step provides ample means for programming efficient, high-performance iterative sorting algorithms without the need of any recursion mechanisms.

For sorting arrays of any type, no matter what range of values array their items may occupy, Quicksort can be recommended as a reasonably fast, memory-efficient sorting method. Both straight DATA step code in the form of a LINK subroutine and more sophisticated %Qsort() can be successfully used to order large single and parallel arrays in place with a speed that is likely to satisfy real-world needs in this kind of data processing.

REFERENCES

1. Donald E.Knuth, The Art of Computer Programming, v.3, Addison Wesley (1998).
2. C.A.R. Hoare, Comp. J. 5 (1962), 10-15.
3. R. Sedgewick, Acta Informatica 7 (1977), 327-356.
4. J.L. Bentley and M.D. McIlroy, Software Practice & Exper. 23 (1993), 1249-1265.
5. Paul M. Dorfman. Building Macro-based Data Step Functions Using Random Macro Aliases to Simulate Local Scope of Internal Function Variables. Proceedings of SESUG'98, 97.

ACKNOWLEDGEMENTS

The author would like to acknowledge the contribution of Doris H. Bogar in the preparation of this paper.

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
10023 Belle Rive Blvd. 817, Jacksonville, FL 32256
(904) 954-8533 (w) (904) 564-1931 (h)
paul.dorfman@citicorp.com sashole@bellsouth.net