Paper 232

ALTERNATIVE APPROACH TO SORTING ARRAYS AND STRINGS: TUNED DATA STEP IMPLEMENTATIONS OF QUICKSORT AND DISTRIBUTION COUNTING

Paul M. Dorfman
Information Management Alternatives Plus
Jacksonville, FL

ABSTRACT

Sorting arrays and strings is common in SAS® applications development. Most often, the following approach is adopted:

- unload the object into a SAS dataset
- save the rest of the data
- sort the dataset using PROC SORT
- reload the sorted data and restore the status quo.

Would not it be better to sort our object in place by simply coding something like

```
SORT (OBJECT,DESCENDING)
```

without leaving DATA step boundaries? As no such function is supplied as part of base SAS, the only alternative to the unload-sort-load cycle is to write a custom sorting routine.

Two efficient sorting algorithms stand out. Quicksort combines speed with practically negligible auxiliary memory, and is excellent for general array sorting. Distribution counting has narrower applicability, and may use more auxiliary memory. However, within its domain of ordering limited range data, its performance is unmatched.

This paper discusses a number of tuned DATA step implementations of these algorithms. It shows that they run either in the ballpark or faster than the method based on PROC SORT. As stand-alone macro-based DATA step functions, these routines provide SAS developers with less-to-program, efficient specialized sorting tools.

INTRODUCTION

Whenever an array or string needs to be sorted, there are two choices:

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. So, we end up with at least two extra steps and a lot of I/O traffic, all for the sake of making PROC SORT work for us. And then, is it logical to program this way? The array (string) already resides in memory, and it is exactly where any sorting process would invariably place it, as a whole or piece by piece. So, why should we write it to a file even once?

When these reasons become compelling enough to take the second path, bubble sort is chosen most often because of its simplicity. Unfortunately, it works satisfactorily only for small number of items N; as N grows, its performance becomes intolerable. In spite of that, programming of efficient sorting algorithms in the DATA step is rarely attempted.

It is true that well-performing techniques like quicksort are more complex and tricky than simple schemes. However, the time taken to code and tune a decent sorting routine is richly rewarded in the long run by savings in both programming and machine time.

This paper contains a brief description of two high-performance sorting algorithms: Quicksort and distribution counting. It presents their DATA step implementations, usage examples, and performance evaluation.

QUICKSORT

Suppose that we need to sort an array A containing N elements. Quicksort is one of all-around fastest and resource-efficient comparison-based methods. It is good for several reasons:

1. Because it is comparison-based, it is insensitive to the data type of the array.
2. It sorts in place.
3. As its name implies, it is pretty quick. Among methods sorting in O(N*logN) time, quicksort is fastest.
4. Its auxiliary memory requirements are practically negligible.

MAIN IDEA

Step 1. Take one element called "pivot" and determine its final position in the sorted array, say, S. Assuming ascending order, this will occur when A(S) 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(1)...A(S-1) and partition A(S+1)...A(N).

Step 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 all partitions consist of one element, the array is 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 index I to scan A from left to right. When an item greater than the pivot is found, stop.
2. Use index 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, 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, we should make two important "think ahead" points.

TWO PITFALLS

1. Since we are trying to implement a divide-and-conquer paradigm, we might intuitively expect that partitioning will run in O(logN) time. But certain input arrangements can cause it to perform quite poorly! That will 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 subdivided will have been reduced by only one element! In this case, quicksort will run in O(N**2) time, no better than bubble sort. In practice, this situation will arise, for instance, if A is sorted into ascending order, and the leftmost element in each partition is chosen as a pivot.

2. How do we prevent the indices I and J from running out of bounds and should we worry about it? Yes, we should, for 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 pivot. If the pivot is the greatest item, the same will happen to I, only at the upper bound. Of course, an explicit condition could be coded to check the range. However, it is known, both theoretically and experimentally, that adding one mere instruction to the internal loop of quicksort may cause its performance to plummet, no matter how innocent the offending instruction might seem to be.

DIVIDE...

Fortunately, there exists a trick capable of resolving both problems at the same time. It is called "median-of-the-three partitioning". Namely, we first choose the pivot as the median of the left, right, and middle items. Then, we swap the pivot with the second element, thus tucking it out of the way, and sort the first, second, and last items of the partition in place "by hand". When done with partitioning, the pivot is inserted into its final place, and the second item is replaced at its location. This simple procedure achieves two important effects at the same time. First, now the leftmost and rightmost elements serve as natural sentinels for I and J. Second, choosing the median element as a pivot makes it exponentially improbable for any particular input to elicit the worst-case behavior. As an added benefit, it results in more balanced partitioning.

Apparently, we will have to exchange items quite often, so to shorten the code and make it more clear, it is convenient to create a little macro %SW(X,Y) :

```
%MACRO SW (X,Y);
   DO; T = &X; &X = &Y; &Y = T; END;
%MEND SW;
```

This allows swapping of any two equal length items A and B of the same type simply by coding %SW(A,B). Now we are ready to implement a rather refined partitioning scheme. Let L and R point to the left and right elements, and let K be the pivot value:

```
J = (L+R) * 0.5;
%SW(A(J),A(L+1));
IF A(L)   > A(R)
   THEN %SW(A(L),A(R));
IF A(L)   > A(L+1)
   THEN %SW(A(L),ARR(L+1));
IF A(L+1) > A(R)
   THEN %SW(A(R),A(L+1));
I = L+1;
J = R+1;
K = A(L+1);
DO WHILE (1);
   DO UNTIL(A(I) => K);
      I ++ 1;
   END;
   DO UNTIL (K => A(J));
      J +- 1;
   END;
   IF I => J THEN LEAVE;
   %SW(A(I),A(J));
END;
A(L+1) = A(J);
A(J)   = K;
```

At this point, partitioning is complete, and J points to the pivot item, already in its proper place in the sorted array. L and R point to the left item of the left partition and right item of the right partition, respectively.

...AND CONQUER

Now we have to subdivide two unsorted partitions further in the same fashion. But only one of them can be processed at a time! In lieu of recursion, we can store the endpoint indices of the longer partition in a stack and subdivide the shorter one. Repeating the process, we will eventually find that a shorter partition cannot be subdivided any further. At this point, it is time to pop the stack and start partitioning the most recently stored partition in exactly the same manner as above. If the stack is empty, quicksorting is finished.

The stack can be implemented as a temporary array combined with a variable S that is increased by 1 every time before an item is pushed, and decreased by 1 after it is popped. But how many entries should be reserved for the stack? Both theory and practice show that if, as suggested above, shorter partitions are divided at once, while the processing of longer partitions is postponed, quicksort needs at most 2log(N) stack entries to store endpoints of the deferred partitions. It means that a stack array STACK(0:1,0:50) will suffice for sorting 2**50, or more than 1E15, items, so stack size can be safely coded as the constant.

Last but not least: Is it efficient to do partitioning until all partitions are reduced to 1 element? The answer is no. In fact, the most efficient approach is to perform quicksorting only until a given partition contains 9 or fewer items, and finish the process by a single pass of straight insertion sort. Insertion sort is a poor choice for

sorting a large disordered object, but it excels if the object is well preordered. After any number of partitioning steps, all items located between partitions are already in their final place since they 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. So, when all partitions hold only small number of elements, very few inversions remain, thus placing the process squarely in the domain where insertion sort is fastest.

PUTTING IT ALL TOGETHER

Finally, all building blocks can be put together:

```
ARRAY STACK(0:1,0:50) _TEMPORARY_;
L = LBOUND(A);
R = HBOUND(A);
S = 0;
IF R-L > 9 THEN DO WHILE(1);
   /* Insert partitioning code */
   IF J-L > 9 AND R-J > 9 THEN DO;
      S ++ 1;
      IF (R-J) < (J-L) THEN DO;
         STACK(0,S) = L  ;
         STACK(1,S) = J-1;
         L          = J+1;
      END;
      ELSE DO;
         STACK(0,S) = J+1;
         STACK(1,S) = R  ;
         R          = J-1;
      END;
   END;
   ELSE IF J-L <= 9 AND R-J <= 9 THEN DO;
      IF S > 0 THEN DO;
         L = STACK(0,S);
         R = STACK(1,S);
         S +- 1;
      END;
      ELSE LEAVE;
   END;
   ELSE IF J-L  > 9 AND R-J <= 9
        THEN R = J-1;
        ELSE L = J+1;
END;
L = LBOUND(A);
R = HBOUND(A);
DO J=L+1 TO R;
   IF A(J-1) > A(J) THEN DO;
      K = A(J);
      DO I=J-1 TO L BY -1;
         IF K => A(I) THEN LEAVE;
         A(I+1) = A(I);
      END;
      A(I+1) = K;
   END;
END;
```

The last nested loop is straight insertion sort. The most efficient sequence of IF-THEN-ELSE logic was determined by sorting a randomly populated array and monitoring the number of different instructions executed in each logical case. Then the cases were arranged into descending order by the frequency of operations weighed by their cost. The program above presents the final tuned version.

FORGET ABOUT THE ENGINE, JUST DRIVE

Although the code can be embedded into a DATA step program, it is far more convenient to formalize the routine as an encapsulated macro module. Such a macro, %QSORT, is presented in Appendix 1. The name of an existing array must be supplied to the parameter ARR=. Desired sort order can be requested by coding SEQ=A (default) or SEQ=D. Parameters FIRST= and LAST= default to the lower and upper bounds and can be used to specify a range of indices to be included in sorting. The latter may be convenient if the array must be ordered in a special way. For instance, ordering an array A(100) half ascending, half descending can be accomplished with two consecutive calls to %QSORT:

```
%QSORT(ARR=A,SEQ=A,FIRST=1, LAST=50);
%QSORT(ARR=A,SEQ=D,FIRST=51,LAST=100);
```

Using defaults, the same can be coded simpler and more dynamically:

```
%QSORT(ARR=A,LAST=DIM(A)*.5);
%QSORT (ARR=A,SEQ=D,FIRST=DIM(A)*.5+1);
```

DO WE GET THERE ON TIME?

Can DATA step quicksort make our programming life easier? Let us consider the following task: A SAS dataset A consists of numeric variables A1-A200 and 5,000 observations, and we need to sort A1-A200 into ascending order in every row. To be able to use PROC SORT, we could code something like this:

```
DATA VERT (KEEP=ID VA);
   ARRAY A(200);
   SET A;
   ID + 1;
   DO I=1 TO DIM(A);
      VA = A(I);
      OUTPUT;
   END;
RUN;
PROC SORT; BY ID VA; RUN;
DATA A (KEEP=A1-A200);
   ARRAY A(200);
   RETAIN A1-A200;
   SET VERT;
   BY ID;
   IF FIRST.ID THEN N = 0;
   N + 1;
   A(N) = VA;
   IF LAST.ID;
 RUN;
```

Running against random, ordered, and reverse-ordered inputs in the mainframe batch, these steps execute, on the average, in 4.5, 35.5, and 7.5 CPU seconds. Using %QSORT, the same output can be produced with a single macro call:

```
DATA A;
   ARRAY A(100);
   SET A;
   %QSORT (ARR=A);
RUN;
```

This single step accomplishes the whole task in 11.5 CPU seconds, on the average. Thus, using %QSORT not only saves programming time; it also runs three times faster than PROC SORT alone. As a benchmark, on the same machine, it takes %QSORT 2.5 CPU seconds, on the average, to sort an array with 100,000 elements.

DISTRIBUTION COUNTING

Dealing with quicksort, we made no assumptions about the nature of the data to be sorted. It could be anything – and it is where comparison-based algorithms, and quicksort in particular, excel. However, in many cases the very nature of input data lends itself to most

efficient sorting solutions. Let us, for example, ask the following question: What is the best way to sort a string that might, in principle, contain any character? Of course, we could once again resort to the PROC SORT approach, which, as we have seen, is cumbersome at best. Or, now that we have quicksort up and running, we could break the string into 1-byte pieces, store them in a temporary array, order it using %QSORT, and assemble the string again in the desired order.

SOME SIMPLE WISDOM

But there is a better way to do the job! We know full well that no byte in the string can take on more than 2**8=256 distinct values numbered from 0 to 255 in the collating sequence. So, instead of comparing elements and moving them around in the memory, we can take an entirely different path:

1. Create an auxiliary array C(0:255) with 256 buckets.
2. Scroll through the string and for each character, determine its number in the collating sequence. If it is 0, then add 1 to C(0), if it is 1, add 1 to C(2), and so on. Hence, at the end of the pass C(I) will represent the number of times the character with the collating number I occurs in the input string.
3. Scroll through C. If a bucket C(I) is empty, skip it. If not, determine which character in the collating sequence corresponds to the number I. Move this character as the next byte in string C(I) times. If duplicates have to be deleted, move it just once, empty the bucket and proceed to the next bucket.

Note that no comparisons have been made, yet at this point the string has been completely sorted in place!

DATA STEP IMPLEMENTATION

The algorithm is so simple and straightforward that it can be implemented in the SAS DATA step immediately:

```
ARRAY C(0:255) _TEMPORARY_;
DO I=1 TO LENGTH(STR);
   C(RANK(SUBSTR(STR,I,1))) ++ 1;
END;
P = 0;
DO I=0 TO 255 BY +1;
   CH = BYTE(I);
   DO WHILE (C(I) > 0);
      P ++ 1;
      SUBSTR(STR,P,1) = CH;
      C(I) +- 1;
   END;
END;
DO I=P+1 TO LENGTH(STR);
   SUBSTR(STR,I,1) = ' ';
END;
```

Counter P points to the position in STR into which the next character CH must be placed. To sort descending, we need to scroll C in the reverse direction, that is, from 255 to 0. To eliminate duplicates, it suffices to set C(I) to 0 instead of decreasing it by 1. For each duplicate character deleted from STR, the last 3 lines of code pad STR on the right with a blank, although any character could be used.

PERFORMANCE

It follows from the nature of the algorithm that no sorting method is likely to do better: The entire string is sorted in a single pass. It means that distribution counting sort runs in O(N) time.

The routine is also self-cleaning. At the end, the count array C is populated with zeroes and missing values, so it is ready for the next call without initializing. As with quicksort, it is convenient to have this program on call as a self-sustained macro routine. Such a macro, %STRSORT, whose arguments allow one to specify sort sequence, duplicate elimination, and a padding character, is presented in Appendix 2. STR= is the only mandatory parameter: the name of an existing character variable must be specified. PAD=, defaulting to a quoted blank, accepts any other character for padding. Duplicates will remain intact unless NODUP=Y is coded because NODUP=N is a default. Finally, to specify a desired sort order, SEQ=D or SEQ=A should be coded; by default, ascending order is assumed.

USAGE EXAMPLE

Assume we have a SAS dataset S with 5,000 rows containing, among others, a 200-byte character variable STR. We need to create a dataset where STR would be sorted into descending order in every observation, with duplicates squeezed to the right and replaced by low values. %STRSORT provides a straightforward, one-step way of solving the problem:

```
DATA S;
   SET S;
   %SORTSTR (STR=STR,SEQ=D,NODUP=N,
             PAD='00'X);
RUN;
```

On the same machine, this step finishes the whole task in 7.0 CPU seconds on the average. For the sake of comparison, PROC SORT approach was coded in the manner similar to that in the quicksort section, with the added complexity of padding and preserving the information not contained in STR. The three steps of the program, run, on the average, in 4.5, 38.5, and 8.0 CPU seconds, respectively.

OTHER APPLICATIONS

Distribution counting sort is not limited to sorting strings. It will sort, faster than anything else will, any object whose items can be inexpensively converted to non-negative integers falling into a limited range. In the case of string sorting, the range is naturally restricted to 256 distinct values. Practically, "limited range" is only limited by the maximum size of the array C. Nowadays, creating a temporary array having 1,000,000 items poses no memory problems. It means, for example, that to sort an array A consisting of 6-digit non-negative integers, all we have to code is

```
ARRAY C(0:999999) _TEMPORARY_;
 DO I=1 TO DIM(A);
    C(A(I)) ++ 1;
 END;
 P = 0;
 DO I=0 TO HBOUND(C);
    DO WHILE (C(I) > 0);
       P ++ 1;
       A(P) = I;
       C(I) +- 1;
    END;
 END;
```

Once again, reversing the direction of the outer loop reverses the sort sequence, and duplicates are

eliminated by replacing the instruction C(I)+-1 with C(I) = 0. After sorting is done, counter P will point to the rightmost non-duplicate array element.

Not only is this program remarkably short and simple; it also runs with a truly blazing speed, spending no more than 4 CPU seconds to sort an array containing 1 million elements.

CONCLUSION

1. Efficient sorting of arrays and strings can be accomplished without resorting to PROC SORT and crossing DATA step boundaries. SAS DATA step provides ample means for programming efficient, high-performance sorting algorithms iteratively, without invoking any recursion mechanisms.

2. 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. Working on a single array, it runs in the ballpark of PROC SORT, but in certain situations, it can be much faster.

3. If the values of items to be sorted fall into a limited range, as in the case of sorting character strings or numeric arrays whose elements contain small number of digits, distribution counting sort is simplest and fastest. It usually consumes more memory than quicksort, but uses it wisely: In the domain of its applicability, it can run up to ten times faster than any other sorting method.

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 contributions of the following individuals in the preparation of this paper.

Doris H. Bogar
Colin Earle
Victor P. Dorfman
Michael V. Dorfman
Yuri Katsnelson
Vladimir A. Kirillov
Eugenia P. Kravchenko
Michael A. Raithel
Vera Voloshin
Matthew Wilson

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
10023 Belle Rive Blvd. 817
Jacksonville, FL 32256
(904) 564-1931
sashole@earthlink.net

APPENDIX 1: %QSORT

```
%MACRO QSORT
(ARR=,SEQ=A,FIRST=LBOUND(&ARR),LAST=HBOUND(&ARR));
   %* Randomize internal variable names;
   %LOCAL LIST I J K R L T S STACK;
   %LOCAL STRONE STRALL PNUM VNAME FLN FL N;
   %LET LIST   = I J K R L T S STACK;
   %LET STRONE = _ABCDEFGLASTJKLMNOPQRSTUVWXYZ;
   %LET STRALL = 0123456789&STRONE;
   %LET PNUM   = 1;
   %LET VNAME  = %SCAN(&LIST,&PNUM);
   %DO %WHILE(&VNAME NE);
      %LET FLN = %SCAN(%SYSEVALF(%SYSFUNC
         (RANUNI(0))*26+1),1,%STR(.));
      %LET FL  =  %SUBSTR(&STRONE,&FLN,1);
      %LET ALL = ;
      %DO N=1 %TO 7;
         %LOCAL ALL&N;
         %LET ALL&N =
         %SCAN(%SYSEVALF(%SYSFUNC
            (RANUNI(0))*36+1),1,%STR(.));
         %LET ALL = &ALL%SUBSTR(&STRALL,&&ALL&N,1);
      %END;
      %IF &VNAME NE STACK %THEN %STR(DROP &FL&ALL;);
      %LET &VNAME  = &FL&ALL;
      %LET PNUM    = %EVAL(&PNUM+1);
      %LET VNAME   = %SCAN(&LIST,&PNUM);
   %END;
   %* Assign sequence suffices;
   %IF %UPCASE(&SEQ) = A %THEN %LET SEQ = L;
   %ELSE %LET SEQ = G;
   %* Define swap function;
   %MACRO SW(X,Y);
       DO; &T = &X; &X = &Y; &Y = &T; END;
   %MEND SW;
   %* First stack index: 0=left, 1=right;
   ARRAY &STACK (0:1,0:50) _TEMPORARY_;
   %* Initialize partition and stack pointers;
   &L = &FIRST;
   &R = &LAST;
   &S = 0;
   %* If fewer than 10 items skip to insertion;
   IF &R-&L > 9 THEN DO WHILE (1);
   %* Perform medium-of-the-three partitioning;
      &J = (&L+&R) * 0.5;
      %SW (&ARR(&J),&ARR(&L+1));
      IF &ARR(&L)   &SEQ.T &ARR(&R)   THEN
         %SW (&ARR(&L),&ARR(&R));
      IF &ARR(&L)   &SEQ.T &ARR(&L+1) THEN
         %SW (&ARR(&L),&ARR(&L+1));
      IF &ARR(&L+1) &SEQ.T &ARR(&R)   THEN
         %SW (&ARR(&R),&ARR(&L+1));
      &I = &L+1;
      &J = &R+1;
      &K = &ARR(&L+1);
      DO WHILE (1);
         DO UNTIL(&ARR(&I) &SEQ.E &K);
            &I ++ 1;
         END;
         DO UNTIL(&K &SEQ.E &ARR(&J));
            &J +- 1;
         END;
         IF &I => &J THEN LEAVE;
         %SW (&ARR(&I),&ARR(&J));
      END;
      &ARR(&L+1) = &ARR(&J);
      &ARR(&J)   = &K;

      %* Handle the stack;
      IF &J-&L > 9 AND &R-&J > 9 THEN DO;
         &S ++ 1;
         IF &R-&J < &J-&L THEN DO;
            &STACK (0,&S) = &L  ;
            &STACK (1,&S) = &J-1;
            &L            = &J+1;
         END;
         ELSE DO;
            &STACK (0,&S) = &J+1;
            &STACK (1,&S) = &R  ;
            &R            = &J-1;
         END;
      END;
      ELSE IF &J-&L <= 9 AND &R-&J <= 9 THEN DO;
         IF &S > 0 THEN DO;
            &L = &STACK (0,&S);
            &R = &STACK (1,&S);
            &S +- 1;
         END;
         ELSE LEAVE;
      END;
      ELSE IF &J-&L > 9 AND &R-&J <= 9
           THEN &R = &J-1;
           ELSE &L = &J+1;
   END;
   %* Finish up using straight insertion;
   &L = &FIRST;
   &R = &LAST;
   DO &J=&L+1 TO &R;
      IF &ARR(&J-1) &SEQ.T &ARR(&J) THEN DO;
         &K = &ARR(&J);
         DO &I=&J-1 TO &L BY -1;
            IF &K &SEQ.E &ARR(&I) THEN LEAVE;
            &ARR(&I+1) = &ARR(&I);
         END;
         &ARR(&I+1) = &K;
      END;
   END;
%MEND QSORT;
```

APPENDIX 2: %SORTSTR

```
%MACRO SORTSTR (STR=,SEQ=A,NODUP=N,PAD=' ');
   %LOCAL B C I P ;
   %* Randomize internal variables;
   %LET B = B%SUBSTR(%SYSFUNC(RANUNI(0)),4,7);
   %LET C = C%SUBSTR(%SYSFUNC(RANUNI(0)),4,7);
   %LET I = I%SUBSTR(%SYSFUNC(RANUNI(0)),4,7);
   %LET P = P%SUBSTR(%SYSFUNC(RANUNI(0)),4,7);
   DROP &B &I &P;
   ARRAY &C(0:255) _TEMPORARY_;
   %* Populate count array buckets;
   DO &I=1 TO LENGTH(&STR);
      &C(RANK(SUBSTR(&STR,&I,1))) ++ 1;
   END;
   %Initialize position pointer;
   &P = 0;
   %* Repopulate STR in specified order;
   %IF %UPCASE(&SEQ) = A %THEN %DO;
   DO &I=0 TO 255;
   %END;
   %ELSE %DO;
   DO &I=255 TO 0 BY -1;
   %END;
      &B = BYTE(&I);
      DO WHILE (&C(&I) > 0);
         &P ++ 1;
         SUBSTR(&STR,&P,1) = &B;
         %IF %UPCASE(&NODUP) = N %THEN %DO;
            &C(&I) +- 1;
         %END;
         %ELSE %DO;
            &C(&I) = 0;
         %END;
      END;
   END;
   DO &I=&P+1 TO LENGTH(&STR);
      SUBSTR(&STR,&I,1) = &PAD;
   END;
%MEND SORTSTR;
```