18.1 Sorting One-Dimensional Arrays in Memory

A. Purpose

Sort one-dimensional arrays of Integers, Single Precision floating point numbers, Double Precision floating point numbers or Character strings. Facilities are provided to sort the arrays in place, or to produce a permutation vector defining the sorted order.

B. Usage

Usage for sorting numeric arrays in place, sorting numeric arrays using a permutation vector, sorting character string arrays in place, and sorting character string arrays using a permutation vector, is described in Sections B.1 through B.4 below, respectively.

B.1 Sorting One-Dimensional Arrays of Numbers in Place

B.1.a Program Prototype, Integer

INTEGER M, N, I(≥N)

Assign values to M, N and I(M:N). Require $1 \le M \le N$.

CALL ISORT (I, M, N)

Following the call to ISORT the elements I(M:N) will have been put into ascending order according to their algebraic (signed) values.

B.1.b Argument Definitions

I() [inout] An array containing the integers to be sorted.

M [in] The lower bound, in I(), of the array of integers to be sorted.

N [in] The upper bound, in I(), of the array of integers to be sorted. The only elements of I referenced are I(M:N).

B.1.c Program Prototype, Real

To sort an array of real numbers, change the type of I() from INTEGER to REAL, and change the subprogram name from ISORT to SSORT.

B.1.d Program Prototype, Double Precision

To sort an array of double precision numbers, change the type of I() from INTEGER to DOUBLE PRECISION, and change the subprogram name from ISORT to DSORT.

B.2 Sorting One-Dimensional Arrays of Numbers using a Permutation Vector

B.2.a Program Prototype, Integer

INTEGER M, N, I(≥N)**, IP**(≥N)

Assign values to M, N and I(M:N). Require $1 \le M \le N$.

CALL ISORTP (I, M, N, IP)

Following the call to ISORTP the contents of IP(M:N) define the sorted order.

B.2.b Argument Definitions

I() [in] An array containing the integers to be sorted.

M [in] The lower bound, in I(), of the array of integers to be sorted.

N [in] The upper bound, in I(), of the array of integers to be sorted. The only elements of I and IP referenced are I(M:N) and IP(M:N).

IP() [out] An array to contain the definition of the sorted sequence. IP(M:N) is set so that the $(1 + J - M)^{th}$ element of the sorted sequence is I(IP(J)) for $M \le J \le N$.

B.2.c Using a Pre-specified Permutation Vector

INTEGER M, N, I(≥ *max*)**, IP**(≥N) [*max* is the maximum value appearing in IP(M:N).]

Assign values to M, N, IP(M:N) and elements of I() indexed by IP(M:N). Require $1 \le M \le N$. IP(M:N) must be distinct and positive, but it is not necessary that they be a permutation of a sequence of length $N - M + 1$.

CALL ISORTQ (I, M, N, IP)

Following the call to ISORTQ the contents of IP(M:N) define the sorted order.

B.2.d Argument Definitions

The arguments for ISORTQ are the same as those for ISORTP, except that IP has [inout] intent instead of [out].

B.2.e Program Prototype, Real

To sort an array of real numbers, change the type of I() from INTEGER to REAL, and change the subprogram name from ISORTP or ISORTQ to SSORTP or SSORTQ, respectively.

B.2.f Program Prototype, Double Precision

To sort an array of double precision numbers, change the type of I() from INTEGER to DOUBLE PRECISION, and change the subprogram name from ISORTP or ISORTQ to DSORTP or DSORTQ, respectively.

©1997 Calif. Inst. of Technology, 2010 Math à la Carte, Inc.

B.3 Sorting One-Dimensional Arrays of Character Strings in Place

B.3.a Program Prototype

INTEGER M, N, K, L

CHARACTER*(≥L) C(≥N), CTEMP

Assign values to M, N, K, L and C(M:N). Require $1 \le M \le N$ and $1 \le K \le L$.

CALL CSORT (C, M, N, K, L, CTEMP)

Following the call to CSORT the contents of C(M:N) are such that the $(1 + J - M)^{th}$ element of the sorted sequence is C(IP(J)) for $M \le J \le N$. The J^{th} element of the sorted sequence is ".LE." the $J+1^{st}$ element. The effect of ".LE." applied to character strings depends on the computer system.

B.3.b Argument Definitions

C() [inout] An array containing the array of character strings to be sorted.

M [in] The lower bound, in C(), of the array of character strings to be sorted.

N [in] The upper bound, in C(), of the array of character strings to be sorted. The only elements of C() referenced are C(M:N).

K [in] The lower bound in each element of C() of the part of the character string that is to determine the order.

L [in] The upper bound in each element of C() of the part of the character string that is to determine the order.

CTEMP [scratch] A scalar character string having a length at least as long as elements of C().

B.4 Sorting One-Dimensional Arrays of Character Strings using a Permutation Vector

B.4.a Program Prototype

INTEGER M, N, K, L, IP(≥N)

CHARACTER*(≥L) C(≥N)

Assign values to M, N, K, L and C(M:N). Require $1 \le M \le N$ and $1 \le K \le L$.

CALL CSORTP (C, M, N, K, L, IP)

Following the call to CSORTP the contents of IP(M:N) define the sorted order.

B.4.b Argument Definitions

C() [in] An array containing the array of character strings to be sorted.

M [in] The lower bound, in C(), of the array of character strings to be sorted.

N [in] The upper bound, in C(), of the array of character strings to be sorted. The only elements of C() and IP() referenced are C(M:N) and IP(M:N).

K [in] The lower bound in each element of C() of the part of the character string that is to determine the order.

L [in] The upper bound in each element of C() of the part of the character string that is to determine the order.

IP() [out] An array to contain the definition of the sorted sequence. The contents of IP(M:N) are permuted so that the $(1+J-M)^{th}$ element of the sorted sequence is C(IP(J)) for $M \le J \le N$. The J^{th} element of the sorted sequence is ".LE." the $J+1^{st}$ element. The effect of ".LE." applied to character strings depends on the computer system.

B.4.c Using a Pre-specified Permutation Vector

INTEGER M, N, K, L, IP(≥N)

CHARACTER*(≥L) C(≥ *max*) [*max* is the maximum value appearing in IP(M:N).]

Assign values to M, N, K, L, IP(M:N) and elements of C() indexed by IP(M:N). Require $1 \le M \le N$ and $1 \le K \le L$. IP(M:N) must be distinct and positive, but it is not necessary that they be a permutation of a sequence of length $N - M + 1$.

CALL CSORTQ (C, M, N, K, L, IP)

Following the call to CSORTQ the contents of IP(M:N) define the sorted order.

B.4.d Argument Definitions

The arguments for CSORTQ are the same as those for CSORTP, except that IP has [inout] intent instead of [out].

C. Examples and Remarks

C.1 Example

The program DRSSORT illustrates the use of SSORT and SSORTP to sort 1000 randomly generated real numbers. The output should consist of the two lines

SSORTP succeeded

SSORT succeeded

C.2 Sorting According to Other Orders

To sort an array of numbers into descending order, replace each element of the array I by the negative of its original value. To sort an array of numbers into ascending order according to the absolute values of its elements, replace each element by its absolute value. To sort an array of numbers into descending order according to the absolute values of its elements, replace each element by the negative of its original absolute value. If the original signs are important, save the original data, and sort using a permutation vector. To sort the rows or columns of a rectangular array, copy the desired row or column to an auxiliary array, and use ISORTP, SSORTP, DSORTP or CSORTP, as appropriate. The resulting permutation vector defines the order of the rows or columns, as appropriate.

C.3 Stability

A sorting method is said to be stable if the original relative order of equal elements is preserved. The quicksort algorithm is not inherently stable. The GSORTP subprogram, described in Chapter 18.2, may be used to impose stability.

D. Functional Description

All of the subprograms use the quicksort algorithm, due originally to C. A. R. Hoare, as modified by T. N. Hibbard and R. Sedgewick. In the basic quicksort algorithm the sorting problem, say P, is divided into two subproblems, say P_1 and P_2, so that every element in P_1 should be sorted before any element of P_2. Each subproblem is then sorted, using quicksort recursively.

The lower bound for the running time of an algorithm that sorts by comparing elements is $O(n \log n)$, where n is the number of elements to be sorted. The expected running time of quicksort is $O(n \log n)$, but the worst case running time is $O(n^2)$. Nevertheless, because of its low internal overhead, quicksort is usually the fastest sorting method.

The Hibbard modification of the basic quicksort algorithm replaces recursion by using a stack to keep track of the unsorted subproblems, and always puts the larger of the two subproblems onto the stack before the smaller. By putting subproblems onto the stack in this order, the size of the stack is bounded by $\log_2 n$.

Let L and R be the left and right boundaries of the present subproblem. The Sedgewick modifications of the basic quicksort algorithm consist of the following:

(a) The partitioning element, the element used to decide whether another element is in subproblem P_1 or P_2, is chosen to be the median of the L^{th}, R^{th} and middle elements. This choice makes it less likely that the running time will be $O(n^2)$.

(b) The L^{th}, R^{th} and middle elements are sorted so the L^{th} element is the smallest and the R^{th} is the largest. This removes the need for a check in the partitioning loop, the innermost loop of the sort algorithm.

(c) The median element is exchanged with the $R - 1^{th}$ element. This modification, with the previous one, allows the partitioning loop to operate on R − L −2 elements instead of R − L + 1 elements.

(d) Small subproblems are not put onto the stack. Thus the data are sorted into small blocks, but not necessarily sorted within the blocks. The final sorting step uses an insertion sort, which has running time that is $O(n)$ if the data are partly ordered in blocks of size bounded by a constant.

These modifications together have the effect of reducing the average running time by 20% to 30%, as compared to a naïve implementation of the basic quicksort algorithm.

References

1. Robert Sedgewick, **Algorithms**, Addison Wesley, Reading, Mass. (1983).

E. Error Procedures and Restrictions

None of the subprograms detects or reports an erroneous condition.

The only limitation on the size of array that can be sorted is a limitation on the amount of memory available to contain the array, and the depth of an internal stack in each routine. Due to the Hibbard modification of Quicksort, the stack depth cannot exceed $\log_2 n$. The internal stack has a maximum depth of 32. This permits N − M to be as large as 4,294,467,295. The limit on N − M is not checked.

F. Supporting Information

The source language for these subroutines is ANSI Fortran 77.

Designed and coded by W. V. Snyder, JPL 1988. Modified 1992.

Entry	Required Files	Entry	Required Files
CSORT	CSORT	**ISORT**	ISORT
CSORTP	CSORTP	**ISORTP**	ISORTP
CSORTQ	CSORTP	**ISORTQ**	ISORTP
DSORT	DSORT	**SSORT**	SSORT
DSORTP	DSORTP	**SSORTP**	SSORTP
DSORTQ	DSORTP	**SSORTQ**	SSORTP

DRSSORT

```
c     program DRSSORT
c>>   1994-10-19 DRSSORT  Krogh  Changes to use M77CON
c>>   1989-06-13 DRSSORT  CLL Changed "sranua (r" to "sranua (d"
c>>   1988-11-20 DRSSORT  Snyder  Initial code.
c--S replaces "?": DR?SORT, ?SORT, ?SORTP, ?RANUA
c
c     Test driver for SSORT and SSORTP.
c
c     Construct an array of 1000 random numbers using SRANUA.
c     Sort it using SSORTP.
c     Check whether it is in order.
c     Sort it using SSORT.
c     Check whether it is in order.
c
      logical OK
      integer I, P(1:1000)
      real            X(1:1000)
c
c     Generate 1000 random numbers
      call sranua (x, 1000)
c     Sort them using SSORTP.  Assume the sort will work.
      ok=.TRUE.
      call ssortp (x,1,1000,p)
c     Check the order.
      do 10 i = 2, 1000
         if (x(p(i)).lt.x(p(i-1))) ok=.FALSE.
   10 continue
c     Print the results.
      if(ok)then
         print *,'SSORTP succeeded'
      else
         print *,'SSORTP failed'
      end if
c     Sort them using SSORT.  Assume the sort will work.
      ok=.TRUE.
      call ssort (x,1,1000)
c     Check the order.
      do 20 i = 2, 1000
         if (x(i).lt.x(i-1)) ok=.FALSE.
   20 continue
c     Print the results.
      if(ok)then
         print *,'SSORT succeeded'
      else
         print *,'SSORT failed'
      end if
c
      end
```