GALAHAD SORT

USER DOCUMENTATION — GALAHAD Optimization Library version 5.0

1 SUMMARY

GALAHAD_SORT is a suite of Fortran procedures for sorting and permuting. It includes two algorithms (heapsort and quicksort) to sort integer and/or real vectors, another to reorder a sparse matrix from co-ordinate to row format, and a further three tools for in-place permutation and permutation inversion.

ATTRIBUTES — Versions: GALAHAD_SORT_single, GALAHAD_SORT_double. **Date:** March 2002. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory, and Ph. L. Toint, University of Namur, Belgium. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available using both single and double precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the USE statement

```
        USE GALAHAD_SORT_single
```

with the obvious substitution GALAHAD_SORT_double, GALAHAD_SORT_single_64 and GALAHAD_SORT_double_64 for the other variants.

If it is required to use more than one of the modules at the same time, the subroutines SORT_inplace_invert, SORT_inplace_permute, SORT_inverse_permute, SORT_reorder_by_rows, SORT_quicksort, SORT_heapsort_build and SORT_heapsort_smallest, (Section 2.2) must be renamed on one of the USE statements.

2.1 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords REAL(rp_) and INTEGER(ip_), where rp_ and ip_ are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default REAL for the single precision versions and DOUBLE PRECISION for the double precision cases, and correspond to rp_ = real32 and rp_ = real64, respectively, as supplied by the fortran iso_fortran_env module. The latter are default (32-bit) and long (64-bit) integers, and correspond to ip_ = int32 and ip_ = int64, respectively, again from the iso_fortran_env module.

2.2 Argument lists and calling sequences

There are seven procedures that may be called by the user.

1. The subroutine SORT_inplace_invert is used to invert a permutation vector without resorting to extra storage.
2. The subroutine SORT_inplace_permute is used to apply a given permutation to an integer vector and, optionally, to a real vector, without resorting to extra storage.
3. The subroutine SORT_inverse_permute is used to apply the inverse of a given permutation to an integer vector and, optionally, to a real vector, without resorting to extra storage.
4. The subroutine SORT_reorder_by_rows is used to reorder a sparse matrix from arbitary co-ordinate order to row order, that is so that the entries for row i appear directly before those for row $i+1$.

5. The subroutine `SORT_quicksort` is used to sort a given integer/real vector **in ascending order**, optionally applying the same permutation to integer and/or to a real vector(s). It uses the "quicksort" algorithm (see Section 4).
6. The subroutine `SORT_heapsort_build` is used to initialize a procedure to sort a vector of real numbers using the "heapsort" algorithm (see Section 4).
7. The subroutine `SORT_heapsort_smallest` is used to find, possibly repeatedly, the smallest component of a real vector to which `SORT_heapsort_build` has been previously applied (see Section 4). Successive application of this subroutine therefore results in sorting the initial vector **in ascending order**. Optionally, the order may be reversed so that the entries are sorted in descending order instead.

Note that the subroutines `SORT_heapsort_build` and `SORT_heapsort_smallest` are particularly appropriate if it is not known in advance how many successive smallest components of the vector will be required as the heapsort method is able to calculate the $k+1$-st smallest component efficiently once it has determined the first k smallest components. If a complete sort is required, the Quicksort algorithm, `SORT_quicksort` may be preferred. Both methods are guaranteed to sort all n numbers in $O(n \log n)$ operations.

We use square brackets `[ ]` to indicate `OPTIONAL` arguments.

2.2.1 In-place inversion of a permutation

A permutation p of size n is a vector of n integers ranging from 1 to n, each integer in this range occurring exactly once. Its inverse is another permutation q, also of size n, such that $q(p(i)) = i$ for all $i = 1, \ldots, n$. Inverting a given permutation without resorting to extra storage is done as follows:

```
CALL SORT_inplace_invert ( n, p )
```

`n` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set by the user to n, the size of the permutation to be inverted. **Restriction:** `n` > 0.

`p` is a rank-one `INTENT(INOUT)` array argument of dimension at least `n` and type either `INTEGER(ip_)` or `REAL(rp_)`, that must be set by the user on input so that its i-th component contains the integer $p(i)$. On exit, the elements of `p` will have overwritten by those of q, the inverse of p.

2.2.2 Applying a given permutation in place

Applying a given permutation p to a vector x consists in modifying the vector x such that its i-th component appears (after applying the permutation) in component $p(i)$. This is done without resorting to extra storage as follows:

```
CALL SORT_inplace_permute ( n, p [, x] [, ix] [, iy] )
```

`n` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set by the user to n, the size of the permutation to be applied. **Restriction:** `n` > 0.

`p` is a rank-one `INTENT(INOUT)` array argument of dimension at least `n` and type either `INTEGER(ip_)` or `REAL(rp_)`, that must be set by the user on input so that its i-th component contains the integer $p(i)$, that is the i-th component of the permutation one wishes to apply. It is unchanged on exit.

`x` is an optional rank-one `INTENT(INOUT)` array argument of dimension at least `n` and type `REAL(rp_)`, whose n first components must be set by the user. If `x` is present, the component `x(i)` will have been replaced by `x(p(i))` on exit.

`ix` is an optional rank-one `INTENT(INOUT)` array argument of dimension at least `n` and type `INTEGER(ip_)`, whose n fisrt components must be set by the user. If `ix` is present, the component `ix(i)` will have been replaced by `ix(p(i))` on exit.

`iy` is an optional rank-one `INTENT(INOUT)` array argument of dimension at least `n` and type `INTEGER(ip_)`, whose n first components must be set by the user. If `iy` is present, the component `iy(i)` will have been replaced by `iy(p(i))` on exit.

2.2.3 Applying the inverse of a given permutation in place

Applying a the inverse of a given permutation p to a vector x consists in modifying the vector x such that its i-th component appears (after applying the procedure) in component $q(i)$, where q is the inverse of p. Equivalently, this can be seen as modifying the vector x such that its $p(i)$-th component appears (after applying the procedure) in component i. This is done without resorting to extra storage as follows:

```
CALL SORT_inverse_permute ( n, p [, x] [, ix] )
```

`n` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set by the user to n, the size of the permutation whose inverse is to be applied. **Restriction:** `n` > 0.

`p` is a rank-one `INTENT(INOUT)` array argument of dimension at least `n` and type either `INTEGER(ip_)` or `REAL(rp_)`, that must be set by the user on input so that its i-th component contains the integer $p(i)$, that is the i-th component of the permutation whose inverse is to be applied. It is unchanged on exit.

`x` is an optional rank-one `INTENT(INOUT)` array argument of dimension at least `n` and type `REAL(rp_)`, whose n first components must be set by the user. If `x` is present, the component `x(i)` will have been replaced by `x(p(i))` on exit.

`ix` is an optional rank-one `INTENT(INOUT)` array argument of dimension at least `n` and type `INTEGER(ip_)`, whose n first components must be set by the user. If `ix` is present, the component `ix(i)` will have been replaced by `ix(p(i))` on exit.

2.2.4 Reordering a sparse matrix from co-ordinate to row order

The matrix **A** is reordered from co-ordinate to row order as follows:

```
CALL SORT_reorder_by_rows( nr, nc, nnz, A_row, A_col, la, A_val,  A_ptr, lptr,     &
                           IW, liw, error, warning, inform )
```

`nr` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set by the user to the number of rows in **A**. **Restriction:** `nr`> 0.

`nc` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set by the user to the number of columns in **A**. **Restriction:** `nc`> 0.

`nnz` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set by the user to the number of nonzeros in **A**. **Restriction:** `nnz`> 0.

`A_row` is a rank-one `INTENT(INOUT)` array argument of type `INTEGER(ip_)` and length `la`. On entry, `A_row`(k), $k = 1,\ldots,$ `nnz` give the row indices of **A**. On exit, `A_row` will have been reordered, but `A_row`(k) will still be the row index corresponding to the entry with column index `A_col`(k).

`A_col` is a rank-one `INTENT(INOUT)` array argument of type `INTEGER(ip_)` and length `la`. On entry, `A_col`(k), $k = 1,\ldots,$ `nnz` give the column indices of **A**. On exit, `A_col` will have been reordered so that entries in row i appear directly before those in row $i+1$ for $i = 1,\ldots,$ `nr`-1.

`la` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set by the user to the actual dimension of the arrays `A_row`, `A_col` and `A_val` **Restriction:** `la` $\geq$ `nnz`.

A_val is a rank-one INTENT(INOUT) array argument of type REAL(rp_) of length la. On entry, A_val(k), $k = 1,\ldots,$ nnz give the values of **A**. On exit, A_val will have been reordered so that entries in row i appear directly before those in row $i+1$ for $i = 1,\ldots,$ nr-1 and correspond to those in A_row and A_col.

A_ptr is a rank-one INTENT(OUT) array argument of type INTEGER(ip_) and length lptr. On exit, A_ptr(i), $i = 1,\ldots,$ nr give the starting addresses for the entries in A_row/A_col/A_val in row i, while A_ptr(nr+1) gives the index of the first non-occupied component of **A**.

lptr is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set by the user to the actual dimension of A_ptr. **Restriction:** lptr $\geq$ nr + 1.

IW is a rank-one INTENT(OUT) array argument of type INTEGER(ip_) and length liw that is used for workspace.

liw is a scalar INTENT(IN) argument of type INTEGER(ip_), that gives the actual dimension of IW. **Restriction:** liw $\geq$ MAX(nr,nc) + 1.

error is a scalar INTENT(IN) argument of type INTEGER(ip_), that holds the stream number for error messages. Error messages will only occur if error > 0.

warning is a scalar INTENT(IN) argument of type INTEGER(ip_), that holds the stream number for warning messages. Warning messages will only occur if warning > 0.

inform is a scalar INTENT(OUT) argument of type INTEGER(ip_). A successful call to SORT_reorder_by_rows is indicated when inform has the value 0 on exit. For other return values of inform, see Section 2.3.

2.2.5 Quicksort

The vector x is sorted in acending order as follows:

```
CALL SORT_quicksort ( n, x, inform [, ix ] [, rx] )
```

n is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set by the user to n, the number of entries of x that are to be sorted. **Restriction:** n > 0 and n $< 2^{32}$.

x is a rank-one INTENT(INOUT) array argument of dimension at least n and type either INTEGER(ip_) or REAL(rp_), whose first n components must be set by the user on input. On successful return, these components will have been sorted to ascending order.

inform is a scalar INTENT(OUT) argument of type INTEGER(ip_). A successful call to SORT_quicksort is indicated when inform has the value 0 on exit. For other return values of inform, see Section 2.3.

ix is an optional rank-one INTENT(INOUT) array argument of dimension at least n and type INTEGER(ip_). If ix is present, exactly the same permutation is applied to the components of ix as to the components of x. For example, the inverse permutation will be provided if ix(i) is set to i, for $i = 1,\ldots,n$ on entry.

rx is an optional rank-one INTENT(INOUT) array argument of dimension at least n and type REAL(rp_). If rx is present, exactly the same permutation is applied to the components of rx as to the components of x.

2.2.6 Heapsort

Building the initial heap. The initial heap is constructed as follows:

```
CALL SORT_heapsort_build ( n, x, inform [, ix, largest ] )
```

n is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set by the user to n, the number of entries of x that are to be (partially) sorted. **Restriction:** n > 0.

x is a rank-one INTENT(INOUT) array argument of dimension at least n and type either INTEGER(ip_) or REAL(rp_), whose first n components must be set by the user on input. On successful return, the elements of x will have been permuted so that they form a heap.

inform is a scalar INTENT(OUT) argument of type INTEGER(ip_). A successful call to SORT_heapsort_build is indicated when inform has the value 0 on exit. For other return values of inform, see Section 2.3.

ix is an optional rank-one INTENT(INOUT) array argument of dimension at least n and type INTEGER(ip_). If ix is present, exactly the same permutation is applied to the components of ix as to the components of x. For example, the inverse permutation will be provided if ix(i) is set to i, for $i = 1,\ldots,n$ on entry.

largest is an optional scalar INTENT(IN) argument of type INTEGER(ip_). If largest is present and set to .TRUE., the heap will be built so that the largest entry is at the root, and thus the subsequent sort will determine the largest entries in order. Otherwise, the heap will be built with the smallest entry is at the root so that the sort will find the smallest entries in order.

Finding the smallest entry in the current heap. To find the smallest entry in a given heap, to place this entry at the end of the list of entries in the heap and to form a new heap with the remaining entries:

```
    CALL SORT_heapsort_smallest ( m, x, inform [, ix, largest ] )
```

m is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set by the user to m, the number of entries of x that lie on the heap on entry. **Restriction:** m > 0.

x is a rank-one INTENT(INOUT) array argument of dimension at least m and type either INTEGER(ip_) or REAL(rp_) whose first m components must be set by the user on input so that they form a heap. In practice, this normally means that they have been placed on a heap by a previous call to SORT_heapsort_build or SORT_heapsort_smallest. On output, the smallest of the first m components of x will have been moved to position x(m) and the remaining components will now occupy locations $1, 2, \ldots$. m-1 of x and will again form a heap.

inform is a scalar INTENT(OUT) argument of type INTEGER(ip_). A successful call to SORT_heapsort_smallest is indicated when inform has the value 0 on exit. For other return values of inform, see Section 2.3.

ix is an optional rank-one INTENT(INOUT) array argument of dimension at least m and type INTEGER(ip_). If ix is present, exactly the same permutation is applied to the components of ix as to the components of x.

largest is an optional scalar INTENT(IN) argument of type INTEGER(ip_). If largest is present and set to .TRUE., the largest, rather than the smallest, entry will be found and placed in x(m) on exit, and the remaining heap rebuilt

Finding the k smallest components of a set of n elements. To find the k smallest components of a set, $\{x_1, x_2, \ldots, x_n\}$, of n elements, the user should firstly call SORT_heapsort_build with n $= n$ and x_1 to x_n stored in x(1) to x(n). This places the components of x on a heap. This should then be followed by k calls of SORT_heapsort_smallest, with m $= n - i + 1$ for $i = 1, \ldots, k$. The required k smallest values, in increasing order, will now occupy positions $n - i + 1$ of x for $i = 1, \ldots, k$.

2.3 Warning and error messages

A positive value of inform on exit from SORT_reorder_by_rows, SORT_quicksort, SORT_heapsort_build or SORT_heapsort_smallest indicates that an input error has occurred. The other arguments will not have been altered. The only possible values are:

1. One or more of the restrictions nr > 0, nc > 0, nnz > 0 (SORT_reorder_by_rows), n > 0 (SORT_quicksort, SORT_heapsort_build) or m > 0 (SORT_heapsort_smallest) has been violated.

2. One of the restrictions `la` $\geq$ `nnz` (`SORT_reorder_by_rows`) or `n` $< 2^{32}$ (`SORT_quicksort`) has been violated.

3. The restriction `liw` $\geq$ `MAX(nr,nc)+1` (`SORT_reorder_by_rows`) has been violated.

4. The restriction `lptr` $\geq$ `nr+1` (`SORT_reorder_by_rows`) has been violated.

5. All of the entries input in `A_row` and `A_col` are out of range.

A negative value of `inform` on exit from `SORT_reorder_by_rows` indicates that **A** has been successfully reordered, but that a a warning condition has occurred. The only possible values are:

-1. There were duplicate input entries, which have been summed.

-2. There were input row entries out of range, which have been ignored.

-3. There were input column entries out of range, which have been ignored.

3 GENERAL INFORMATION

Use of common: None.

Workspace: None.

Other routines called directly: None.

Other modules used directly: None.

Input/output: None.

Restrictions: `nr` > 0, `nc` > 0, `nnz` > 0, `la` $\geq$ `nnz`, `liw` $\geq$ `MAX(nr,nc)+1`, `lptr` $\geq$ `nr+1`, (`SORT_reorder_by_rows`), `n` > 0 (`SORT_quicksort` and `SORT_heapsort_build`), `m` > 0 (`SORT_heapsort_smallest`) and `n` $< 2^{32}$ (`SORT_quicksort`)

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

4.1 Quicksort

The quicksort method is due to C. A. R. Hoare (Computer Journal, **5** (1962), 10-15).

The idea is to take one component of the vector to sort, say x_1, and to move it to the final position it should occupy in the sorted vector, say position p. While determining this final position, the other components are also rearranged so that there will be none with smaller value to the left of position p and none with larger value to the right. Thus the original sorting problem is transformed into the two disjoint subproblems of sorting the first $p-1$ and the last $n-p$ components of the resulting vector. The same technique is then applied recursively to each of these subproblems. The method is likely to sort the vector x in $O(n \log n)$ operations, but may require as many as $O(n^2)$ operations in extreme cases.

4.2 Heapsort

The heapsort method is due to J. W. J. Williams (Algorithm 232, Communications of the ACM **7** (1964), 347-348). Subroutine SORT_heapsort_build is a partial amalgamation of Williams' Algol procedures *setheap* and *inheap* while SORT_heapsort_smallest is based upon his procedures *outheap* and *swopheap*.

The elements of the set $\{x_1, x_2, ..., x_n\}$ are first allocated to the nodes of a heap. A heap is a binary tree in which the element at each parent node has a numerical value as small as or smaller than the elements at its two children. The smallest value is thus placed at the root of the tree. This value is now removed from the heap and a subset of the remaining elements interchanged until a new, smaller, heap is constructed. The smallest value of the new heap is now at the root and may be removed as described above. The elements of the initial set may thus be arranged in order of increasing size, the i-th largest element of the array being found in the i-th sweep of the method. The method is guaranteed to sort all n numbers in $O(n \log n)$ operations.

5 EXAMPLE OF USE

The following example is a somewhat unnatural sequence of operations, but illustrates the use of the SORT tools. It uses the data vector

$$x = \{x_1, x_2, ..., x_{20}\} = \{-5, -7,\ 2,\ 9,\ 0, -3,\ 3,\ 5, -2, -6,\ 8,\ 7, -1, -8,\ 10, -4,\ 6, -9,\ 1,\ 4\}.$$

Suppose now that we wish to perform the following successful operations:

1. sort the components of x in ascending order and compute the associated inverse permutation,
2. apply this permutation to the resulting vector (in order to recover its original ordering),
3. restore the permutation to the identity by sorting its components in ascending order,
4. find the 12 smallest components of x and the associated inverse permutation,
5. inverse this permutation (which yields the permutation used to sort the 12 smallest components),
6. apply to the permuted x the inverse of the this latest permutation (thus recovering its original ordering again).

Then we may use the following code:

```
! THIS VERSION: GALAHAD 3.3 - 04/02/2020 AT 11:30 GMT.
   PROGRAM GALAHAD_SORT_EXAMPLE
   USE GALAHAD_SORT_double                   ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   INTEGER, PARAMETER :: n = 20
   INTEGER :: i, m, inform
   INTEGER :: p( n )
   REAL  ( KIND = wp ) :: x( n )
   x = (/ -5.0, -7.0, 2.0, 9.0, 0.0, -3.0, 3.0, 5.0, -2.0, -6.0,              &
           8.0, 7.0, -1.0, -8.0, 10.0, -4.0, 6.0, -9.0, 1.0, 4.0 /)  ! values
   p = (/  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,                     &
          15, 16, 17, 18, 19, 20 /)                                  ! indices
! write the initial data
   WRITE( 6, "( /' The vector x is' / 2( 10 ( F5.1, 2X ) / ) ) " ) x( 1:n )
   WRITE( 6, "(  ' The permutation is' /  20 ( 2X, I2 ) / ) " ) p( 1:n )
! sort x and obtain the inverse permutation
   WRITE( 6, "( ' Sort x in ascending order' / )" )
   CALL SORT_quicksort( n, x, inform, p )
   WRITE( 6, "( ' The vector x is now' / 2( 10 ( F5.1, 2X ) / ) ) " ) x( 1:n )
   WRITE( 6, "( ' The permutation is now' /  20 ( 2X, I2 ) / ) " ) p( 1:n )
! apply the inverse permutation to x
   WRITE( 6, "( ' Apply the permutation to x' / )" )
   CALL SORT_inplace_permute( n, p, x )
   WRITE( 6, "( ' The vector x is now' / 2( 10 ( F5.1, 2X ) / ) ) " )  x( 1:n )
! restore the identity permutation
   WRITE( 6, "( ' Restore the identity permutation by sorting' / )" )
   CALL SORT_quicksort( n, p, inform )
   WRITE( 6, "( ' The permutation is now' /  20 ( 2X, I2 ) / ) " ) p( 1:n )
! get the 12 smallest components and the associated inverse permutation
   WRITE( 6, "( ' Get the 12 smallest components of x' / )" )
   CALL SORT_heapsort_build( n, x, inform, ix = p ) !  Build the heap
   DO i = 1, 12
     m = n - i + 1
     CALL SORT_heapsort_smallest( m, x, inform, ix = p ) ! Reorder the variables
     WRITE( 6, "( ' The ', I2, '-th(-st) smallest value, x(', I2, ') is ',     &
    &       F5.1 ) " ) i, p( m ), x( m )
   END DO
   WRITE( 6, "( / ' The permutation is now' /  20 ( 2X, I2 ) / ) " ) p( 1:n )
! compute the direct permutation in p
   WRITE( 6, "( ' Compute the inverse of this permutation' / )" )
   CALL SORT_inplace_invert( n, p )
   WRITE( 6, "( ' The permutation is now' /  20 ( 2X, I2 ) / ) " ) p( 1:n )
! apply inverse permutation
   WRITE( 6, "( ' Apply the resulting permutation to x' / )" )
   CALL SORT_inverse_permute( n, p, x )
   WRITE( 6, "( ' The final vector is' / 2( 10 ( F5.1, 2X ) / ) ) " ) x( 1:n )
   STOP
   END PROGRAM GALAHAD_SORT_EXAMPLE
```

This produces the following output:

```

 The vector x is
 -5.0   -7.0    2.0    9.0    0.0   -3.0    3.0    5.0   -2.0   -6.0
  8.0    7.0   -1.0   -8.0   10.0   -4.0    6.0   -9.0    1.0    4.0

 The permutation is
   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20

 Sort x in ascending order

 The vector x is now
 -9.0   -8.0   -7.0   -6.0   -5.0   -4.0   -3.0   -2.0   -1.0    0.0
  1.0    2.0    3.0    4.0    5.0    6.0    7.0    8.0    9.0   10.0

 The permutation is now
  18  14   2  10   1  16   6   9  13   5  19   3   7  20   8  17  12  11   4  15

 Apply the permutation to x

 The vector x is now
 -5.0   -7.0    2.0    9.0    0.0   -3.0    3.0    5.0   -2.0   -6.0
  8.0    7.0   -1.0   -8.0   10.0   -4.0    6.0   -9.0    1.0    4.0
```

```
Restore the identity permutation by sorting

The permutation is now
   1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19   20

Get the 12 smallest components of x

The  1-th(-st) smallest value, x(18) is  -9.0
The  2-th(-st) smallest value, x(14) is  -8.0
The  3-th(-st) smallest value, x( 2) is  -7.0
The  4-th(-st) smallest value, x(10) is  -6.0
The  5-th(-st) smallest value, x( 1) is  -5.0
The  6-th(-st) smallest value, x(16) is  -4.0
The  7-th(-st) smallest value, x( 6) is  -3.0
The  8-th(-st) smallest value, x( 9) is  -2.0
The  9-th(-st) smallest value, x(13) is  -1.0
The 10-th(-st) smallest value, x( 5) is   0.0
The 11-th(-st) smallest value, x(19) is   1.0
The 12-th(-st) smallest value, x( 3) is   2.0

The permutation is now
   7   20   12   17    8    4   11   15    3   19    5   13    9    6   16    1   10    2   14   18

Compute the inverse of this permutation

The permutation is now
  16   18    9    6   11   14    1    5   13   17    7    3   12   19    8   15    4   20   10    2

Apply the resulting permutation to x

The final vector is
-5.0    -7.0     2.0     9.0     0.0    -3.0     3.0     5.0    -2.0    -6.0
 8.0     7.0    -1.0    -8.0    10.0    -4.0     6.0    -9.0     1.0     4.0
```