# nag_stable_sort (m01ctc)

## 1. Purpose

**nag_stable_sort (m01ctc)** rearranges a vector of arbitrary type objects into ascending or descending order.

## 2. Specification

```
#include <nag.h>
#include <nag_stddef.h>
#include <nagm01.h>

void nag_stable_sort(Pointer vec[], size_t n, size_t size, ptrdiff_t stride,
     Integer (*compare)(const Pointer, const Pointer), Nag_SortOrder order,
     NagError *fail)
```

## 3. Description

nag_stable_sort sorts a set of $n$ data objects of arbitrary type, which are stored in the elements of an array at intervals of length **stride**. The function may be used to sort a column of a two dimensional array. Either ascending or descending sort order may be specified.

A stable sort is one which preserves the order of distinct data items that compare equal. This function uses nag_rank_sort (m01dsc), nag_make_indices (m01zac) and nag_reorder_vector (m01esc) in order to carry out a stable sort with the same specification as nag_quicksort (m01csc). nag_stable_sort will be faster than nag_quicksort (m01csc) if the comparison function **compare** is slow or the data items are large. Internally a large amount of workspace may be required compared with nag_quicksort (m01csc).

## 4. Parameters

**vec**[ ]
: Input: the array of objects to be sorted.
Output: the objects rearranged into sorted order.

**n**
: Input: the number $n$ of objects to be sorted.
Constraint: **n** $\geq 0$.

**size**
: Input: the size of each object to be sorted.
Constraint: **size** $\geq 1$.

**stride**
: Input: the increment between data items in **vec** to be sorted.
**Note**: if **stride** is positive, **vec** should point at the first data object; otherwise **vec** should point at the last data object.
Constraint: |**stride**| $\geq$ **size**.

**compare**
: User-supplied function: this function compares two data objects. If its arguments are pointers to a structure, this function must allow for the offset of the data field in the structure (if it is not the first).
The function must return:
$-1$ if the first data field is less than the second,
0 if the first data field is equal to the second,
1 if the first data field is greater than the second.

**order**
: Input: Specifies whether the array is to be sorted into ascending or descending order.
Constraint: **order** = **Nag_Ascending** or **Nag_Descending**.

**fail**
The NAG error parameter, see the Essential Introduction to the NAG C Library.

## 5. Error Indications and Warnings

**NE_INT_ARG_LT**
On entry, **n** must not be less than 0: **n** = ⟨*value*⟩.
On entry, **size** must not be less than 1: **size** = ⟨*value*⟩.
The absolute value of **stride** must not be less than **size**.

**NE_INT_ARG_GT**
On entry, **n** must not be greater than ⟨*value*⟩: **n** = ⟨*value*⟩.
On entry, **size** must not be greater than ⟨*value*⟩: **size** = ⟨*value*⟩.
On entry, |**stride**| must not be greater than ⟨*value*⟩: **stride** = ⟨*value*⟩.
These parameters are limited to an implementation-dependent size which is printed in the error message.

**NE_2_INT_ARG_LT**
On entry, |**stride**| = ⟨*value*⟩ while **size** = ⟨*value*⟩. These parameters must satisfy |**stride**| ≥ **size**.

**NE_BAD_PARAM**
On entry, parameter **order** had an illegal value.

**NE_ALLOC_FAIL**
Memory allocation failed.

## 6. Further Comments

The time taken by the function is approximately proportional to $n \log n$.

### 6.2. References

Knuth D E (1973) *The Art of Computer Programming (Vol 3, Sorting and Searching)* Addison-Wesley.

## 7. See Also

nag_quicksort (m01csc)
nag_rank_sort (m01dsc)
nag_reorder_vector (m01esc)
nag_make_indices (m01zac)

## 8. Example

The example program reads a three column matrix of real numbers and sorts the first column into ascending order.

### 8.1. Program Text

```
/* nag_stable_sort(m01ctc) Example Program
 *
 * Copyright 1990 Numerical Algorithms Group.
 *
 * Mark 2 revised, 1992.
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nag_stddef.h>
#include <nagm01.h>

#ifdef NAG_PROTO
static Integer compare(const Pointer a, const Pointer b)
#else
     static Integer compare(a,b)
```

```
    Pointer a, b;
#endif
{
  double x = *((double *)a);
  double y = *((double *)b);
  return (x<y ? -1 : (x==y ? 0 : 1));
}

#define MMAX 20
#define NMAX 20

main()
{
  double vec[MMAX][NMAX];
  Integer i, j, m, n, k;
  static NagError fail;

  fail.print = TRUE;
  /* Skip heading in data file */
  Vscanf("%*[^\n]");
  Vprintf("m01ctc Example Program Results\n");
  Vscanf("%ld%ld%ld", &m, &n, &k);
  if (m>=0 && m<=MMAX && n>=0 && n<=NMAX && k>=0 && k<=n)
    {
      for (i=0; i<m; ++i)
        for (j=0; j<n; ++j)
          Vscanf("%lf",&vec[i][j]);
      m01ctc((Pointer) &vec[0][k-1], (size_t) m, sizeof(double),
             (ptrdiff_t)(NMAX*sizeof(double)), compare,  Nag_Ascending, &fail);
      if (fail.code != NE_NOERROR)
        exit(EXIT_FAILURE);
      Vprintf("\nMatrix with column %ld sorted\n", k);
      for (i=0; i<m; ++i)
        {
          for (j=0; j<n; ++j)
            Vprintf(" %7.1f ", vec[i][j]);
          Vprintf("\n");
        }
      exit(EXIT_SUCCESS);
    }
  else
    {
      Vfprintf(stderr, "Data error: program terminated\n");
      exit(EXIT_FAILURE);
    }
}
```

### 8.2. Program Data

```
m01ctc Example Program Data
12 3 1
6.0 5.0 4.0
5.0 2.0 1.0
2.0 4.0 9.0
4.0 9.0 6.0
4.0 9.0 5.0
4.0 1.0 2.0
3.0 4.0 1.0
2.0 4.0 6.0
1.0 6.0 4.0
9.0 3.0 2.0
6.0 2.0 5.0
4.0 9.0 6.0
```

### 8.3. Program Results

```
m01ctc Example Program Results

Matrix with column 1 sorted
     1.0      5.0      4.0
     2.0      2.0      1.0
     2.0      4.0      9.0
     3.0      9.0      6.0
     4.0      9.0      5.0
     4.0      1.0      2.0
     4.0      4.0      1.0
     4.0      4.0      6.0
     5.0      6.0      4.0
     6.0      3.0      2.0
     6.0      2.0      5.0
     9.0      9.0      6.0
```