Sort Template B/sortt

Purpose

To sort arrays.

§**1. Storage.** We are required to use a stable sorting algorithm with very low, ideally zero, auxiliary storage requirement. Exchanges are generally slower than comparisons for the typical application (sorting tables, where entire rows must be exchanged whereas only entries in a single column need be compared).

In fact, we store some details in global variables for convenience and to avoid filling the stack with copies, but otherwise we will hardly need any auxiliary storage.

```
Global I7S_Tab; ! The array to be sorted, which can have almost any format
Global I7S_Col; ! The "column number" in the array, if any
Global I7S_Dir; ! The direction of sorting: ascending (1) or descending (-1)
Global I7S_Swap; ! The current routine for swapping two fields
Global I7S_Comp; ! The current routine for comparing two fields
#ifdef MEASURE_SORT_PERFORMANCE;
Global I7S_CCOUNT; Global I7S_CCOUNT2; Global I7S_XCOUNT; ! For testing only
#endif;
```

§**2. Front End.** To perform a sort, we first call `SetSortDomain` to declare the swap and compare functions to be used, and then call `SortArray` actually to sort. (It would be nice to combine these in a single call, but I6 allows a maximum of 7 call arguments for a routine, and that would make 8.) These are the only two routines which should ever be called from outside of this template segment.

The swap and compare functions are expected to take two arguments, which are the field numbers of the fields being swapped or compared, where fields are numbered from 1. Comparison is like `strcmp`: it returns 0 on equality, and then is positive or negative according to which of the fields is greater in value.

```
[ SetSortDomain swapf compf;
    I7S_Swap = swapf;
    I7S_Comp = compf;
];
[ SortArray tab col dir size test_flag algorithm;
    I7S_Tab = tab;
    I7S_Col = col;
    I7S_Dir = dir;
    #ifdef MEASURE_SORT_PERFORMANCE;
    I7S_CCOUNT = 0;
    I7S_CCOUNT2 = 0;
    I7S_XCOUNT = 0;
    #endif;
    SortRange(0, size, algorithm);
    #ifdef MEASURE_SORT_PERFORMANCE;
    if (test_flag)
        print "Sorted array of size ", size, " with ", I7S_CCOUNT2, "*10000 + ", I7S_CCOUNT,
            " comparisons and ", I7S_XCOUNT, " exchanges^";
    #endif;
];
```

§**3. Sort Range.** This routine sorts a range of fields $x \leq i < y$ within the array. Fields are numbered from 0. The supplied `algorithm` is an I6 routine to implement a particular sorting algorithm; if it is not supplied, then in-place merge sort is used by default.

```
[ SortRange x y algorithm;
    if (y - x < 2) return;
    if (algorithm) {
        (algorithm)(x, y);
    } else {
        InPlaceMergeSortAlgorithm(x, y);
    }
];
```

§**4. Comparison and Exchange.** These are instrumented versions of how to swap and compare fields; note that the swap and compare functions are expected to number the fields from 1, not from 0. (This is convenient both for tables and lists, where rows and entries respectively are both numbered from 1.) The only access which the sorting algorithms have to the actual data being sorted is through these routines.

```
[ CompareFields x y;
    #ifdef MEASURE_SORT_PERFORMANCE;
    I7S_CCOUNT++;
    if (I7S_CCOUNT == 10000) { I7S_CCOUNT = 0; I7S_CCOUNT2++; }
    #endif;
    return I7S_Dir*I7S_Comp(I7S_Tab, I7S_Col, x+1, y+1, I7S_Dir);
];
[ ExchangeFields x y;
    #ifdef MEASURE_SORT_PERFORMANCE;
    I7S_XCOUNT++;
    if (I7S_XCOUNT < 0) { print "XO^"; I7S_XCOUNT = 0; }
    #endif;
    return I7S_Swap(I7S_Tab, x+1, y+1);
];
```

§**5. 4W37 Sort.** We now present three alternative sorting algorithms.

The first is the one used in builds up to and including 4W37: note that this is not quite bubble sort, and that it is unstable. It is now no longer used, but is so short that we might as well keep it in the code base in case anyone needs to resurrect a very early I7 project.

```
[ OldSortAlgorithm x y
    f i j;
    if (y - x < 2) return;
    f = true;
    while (f) {
        f = false;
        for (i=x:i<y:i++)
            for (j=i+1:j<y:j++)
                if (CompareFields(i, j) > 0) {
                    ExchangeFields(i, j); f = true; break;
                }
    }
];
```

§6. Insertion Sort. A stable algorithm which has $O(n^2)$ running time and therefore cannot be used with large arrays, but which has good performance on nearly sorted tables, and which has very low overhead.

```
[ InsertionSortAlgorithm from to
    i j;
    if (to > from+1) {
        for (i = from+1: i < to: i++) {
            for (j = i: j > from: j--) {
                if (CompareFields(j, j-1) < 0)
                    ExchangeFields(j, j-1);
                else break;
            }
        }
    }
];
```

§7. In-Place Mergesort. A stable algorithm with $O(n \log n)$ running time, at some stack cost, and which is generally good for nearly sorted tables, but which is also complex and has some overhead. The code here mostly follows Thomas Baudel's implementation, which in turn follows the `C++` STL library.

```
[ InPlaceMergeSortAlgorithm from to
    middle;
    if (to - from < 12) {
        if (to - from < 2) return;
        InsertionSortAlgorithm(from, to);
        return;
    }
    middle = (from + to)/2;
    InPlaceMergeSortAlgorithm(from, middle);
    InPlaceMergeSortAlgorithm(middle, to);
    IPMS_Merge(from, middle, to, middle-from, to - middle);
];

[ IPMS_Lower from to val
    len half mid;
    len = to - from;
    while (len > 0) {
        half = len/2;
        mid = from + half;
        if (CompareFields(mid, val) < 0) {
            from = mid + 1;
            len = len - half -1;
        } else len = half;
    }
    return from;
];

[ IPMS_Upper from to val
    len half mid;
    len = to - from;
    while (len > 0) {
        half = len/2;
        mid = from + half;
        if (CompareFields(val, mid) < 0)
```

```
            len = half;
        else {
            from = mid + 1;
            len = len - half -1;
        }
    }
    return from;
];

[ IPMS_Reverse from to;
    while (from < to) {
        ExchangeFields(from++, to--);
    }
];

[ IPMS_Rotate from mid to
    n val shift p1 p2;
    if ((from==mid) || (mid==to)) return;
    IPMS_Reverse(from, mid-1);
    IPMS_Reverse(mid, to-1);
    IPMS_Reverse(from, to-1);
];

[ IPMS_Merge from pivot to len1 len2
    first_cut second_cut len11 len22 new_mid;
    if ((len1 == 0) || (len2 == 0)) return;
    if (len1+len2 == 2) {
        if (CompareFields(pivot, from) < 0)
        ExchangeFields(pivot, from);
        return;
    }
    if (len1 > len2) {
        len11 = len1/2;
        first_cut = from + len11;
        second_cut = IPMS_Lower(pivot, to, first_cut);
        len22 = second_cut - pivot;
    } else {
        len22 = len2/2;
        second_cut = pivot + len22;
        first_cut = IPMS_Upper(from, pivot, second_cut);
        len11 = first_cut - from;
    }
    IPMS_Rotate(first_cut, pivot, second_cut);
    new_mid = first_cut + len22;
    IPMS_Merge(from, first_cut, new_mid, len11, len22);
    IPMS_Merge(new_mid, second_cut, to, len1 - len11, len2 - len22);
];
```