# Animated Algorithms

We discuss a method for visualizing the workings of three standard sorting algorithms. The method uses "hooks" in an auxiliary function to obtain run-time data without modifying the implementation of the sorting algorithms themselves.

*Roman E. Maeder*

## Three Standard Sorting Algorithms

We shall look at three algorithms for sorting arrays: insertion sort, selection sort, and quicksort. A description and analysis of these algorithms can be found in any textbook on algorithms, for example [Sedgewick 1990]. All three methods sort an array or list in place, that is, they do not use any auxiliary storage.

Insertion sort and selection sort proceed in a loop over the elements of the array. Both assemble the sorted elements in an initial segment of the array. Insertion sort maintains a sorted initial segment and each subsequent element of the array is inserted into this segment by shifting the elements that are larger to the right. Selection sort repeatedly finds the smallest of the unsorted elements and puts it into its proper place with a single exchange. Quicksort works recursively. In the first phase, the array is partitioned so that all elements in the first part are smaller than all elements in the second part. These two parts can then be sorted independently by two recursive calls of quicksort.

In *Mathematica*, we use lists to hold the elements. The primitive operation is the exchange of two list elements. Parallel assignment allows us to express this exchange in a single statement, without an auxiliary variable. If the variable *l* holds the list, elements *i* and *j* are exchanged by

$$\{l[[i]]\ l[[j]]\} = \{l[[j]],\ l[[i]]\}.$$

To make our programs more readable, we use an auxiliary procedure swap[*l*, *i*, *j*] to perform such an exchange (see Listing 1). Since we want to modify the value of the first argument (the list), swap needs the attribute HoldFirst.

In[1]:=  << SortAux.m

We define a list with symbolic elements.

In[2]:=  l = {a, c, b};

---

*Roman Maeder is one of the designers of Mathematica and the author of three books on Mathematica-related topics. He is now a professor of computer science at the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland.*

---

```
BeginPackage["SortAux`"]

swap::usage =
  "swap[l, i, j] exchanges elements i and j of the value of l."

Begin["`Private`"]

SetAttributes[swap, {HoldFirst}]

swap[l_Symbol, i_, j_] :=
    ({l[[i]], l[[j]]} = {l[[j]], l[[i]]}; l)

End[]

EndPackage[]
```

**LISTING 1:** SortAux.m: Exchanging elements of a list.

This command exchanges the second and third elements:

In[3]:=  swap[l, 2, 3]
Out[3]=  {a, b, c}

Listing 2 shows the code for the three sorting procedures.

## Sorting in Action

A straightforward way to visualize "sortedness" in an array is to restrict ourselves to lists containing a permutation of the integers from 1 to the length of the list. A ListPlot of such a list shows the amount of disorder present. A sorted list (the identity permutation) gives a plot with all points along the minor diagonal. Such diagrams appear in [Sedgewick 1990].

The auxiliary procedure PermutationPlot plots such permutations, choosing good values for the plotting options:

```
In[4]:=  PermutationPlot[l_List, opts___] :=
            ListPlot[ l,
              PlotRange -> { {0.5, Length[l]+0.5},
                            {0.5, Length[l]+0.5} },
              PlotStyle -> PointSize[0.75/Length[l]],
              opts, Axes -> None, FrameTicks -> None,
              Frame -> True, AspectRatio -> 1 ]
```
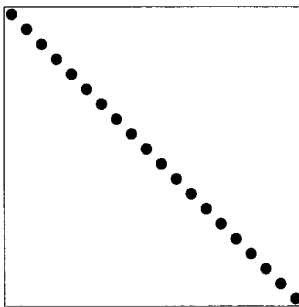
Here is the sorted list containing the numbers 1, 2, ..., 20.

In[5]:=  PermutationPlot[ Range[20] ];
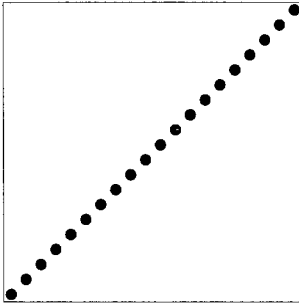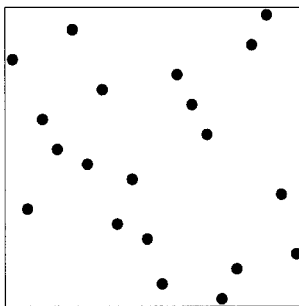


An important test case is a list sorted in reverse.

In[6]:=  PermutationPlot[ Reverse[Range[20]] ];



Here is a picture of a random permutation of 20 elements. The function RandomPermutation is defined in the standard package DiscreteMath`Combinatorica`.

In[7]:=  << DiscreteMath`Combinatorica`

In[8]:=  PermutationPlot[ RandomPermutation[20] ];



   To visualize the progress of a sorting algorithm, we generate a sequence of such permutation plots, one after each exchange. With most versions of *Mathematica*, the resulting sequence of plots can be animated. Here, we'll have to make do with a static version of animation, generated with the package FlipBookAnimation.m.

```
BeginPackage["Sorting`", "SortAux`"]

insertionSort::usage =
    "InsertionSort[l] sorts the list l using insertion sort."
selectionSort::usage =
    "SelectionSort[l] sorts the list l using selection sort."
quickSort::usage =
    "quickSort[l] sorts the list l using quicksort."

Begin["`Private`"]

insertionSort[list_List] :=
    Module[{l = list, i, n = Length[list], j},
        Do[ j = i-1;
            While[ j >= 1 && l[[j]] > l[[j+1]],
                    swap[l, j, j+1]; j-- ],
            {i, 2, n}];
        l
    ]

selectionSort[list_List] :=
    Module[{l = list, i, n = Length[list], min, minj, j},
        Do[ min = l[[i]]; minj = i;
            Do[ If[l[[j]] < min, min = l[[j]]; minj = j],
                {j, i+1, n} ];
            swap[l, i, minj],
            {i, 1, n-1}];
        l
    ]

quickSort[list_] :=
    Module[ {l = list}, qSort[l, 1, Length[l]]; l ]

(* auxiliary procedure *)

SetAttributes[qSort, HoldFirst]

qSort[l_, n0_, n1_] /; n0 >= n1 := l  (* nothing to do *)

qSort[l_, n0_, n1_] :=
    Module[{lm = l[[ Floor[(n0 + n1)/2] ]], i = n0, j = n1},
        While[ True,
            While[ l[[i]] < lm, i++ ];
            While[ l[[j]] > lm, j-- ];
            If[ i >= j, Break[] ];      (* l is partitioned *)
            swap[l, i, j];
            i++; j--
        ];
        (* recursion *)
        If[ i-n0 <= n1-j,
            qSort[ l, n0, i-1 ]; qSort[ l, j+1, n1 ];
            ,
            qSort[ l, j+1, n1 ]; qSort[ l, n0, i-1 ];
        ];
    ]

End[]

EndPackage[]
```

**LISTING 2:** Sorting.m: Three sorting methods.

The remaining problem is to change our implementations of the sorting procedures so that they generate the list of intermediate results. Instead of patching the code of all three procedures, we can modify the auxiliary procedure swap. It serves as our hook into the code. We can change the definition of swap to call another function whenever it is called inside one of the sorting procedures. The function to call is the value of the symbol $swapAction. With suitable values of $swapAction, we can achieve almost any desired effect. To collect data for later analysis or display, we have to use functions with side effects, a programming style normally frowned upon.

This technique of providing hooks inside an otherwise unaccessible piece of code is also used by *Mathematica*'s evaluator. You may be familiar with the hooks $Pre, $Post, and $PrePrint. These hooks inside *Mathematica* are described in the *Mathematica* book in Appendix A.7.

The "hooked" version of swap is given in the file SortAuxG.m (Listing 3). Note that the context name is still SortAux`. The reason is that the package Sorting.m expects this name in BeginPackage["Sorting`", "SortAux`"]. All we have to do is load our special version first, before loading Sorting.m. (Therefore, we restart the kernel at this point.) The default value of hook is the identity function.

We load the special version by giving an explicit file name.

In[1]:= Needs["SortAux`", "SortAuxG.m"]

The sorting functions are in Sorting.m.

In[2]:= Needs["Sorting`"]

The animation functions are needed, as well.

In[3]:= Needs["Graphics`Animation`"]

This package contains the functions for generating random permutations:

In[4]:= Needs["DiscreteMath`Combinatorica`"]

Our first application of $swapAction is to collect the intermediate partially sorted lists. This global variable will hold all the lists:

In[5]:= lists = {};

This value of $swapAction will append its argument to lists:

In[6]:= $swapAction = AppendTo[lists, #]& ;

Here is a permutation of the numbers 1–8:

In[7]:= list = RandomPermutation[8]
Out[7]= {7, 6, 5, 2, 1, 3, 8, 4}

We sort it using insertion sort.

In[8]:= insertionSort[ list ]
Out[8]= {1, 2, 3, 4, 5, 6, 7, 8}

```
(* instrumented version *)

BeginPackage["SortAux`"]

swap::usage =
  "swap[l, i, j] exchanges elements i and j of the value of l."

$swapAction::usage =
  "$swapAction is the function called after each swap."

Begin["`Private`"]

SetAttributes[swap, {HoldFirst}]

swap[l_Symbol, i_, j_] :=
    ({l[[i]], l[[j]]} = {l[[j]], l[[i]]}; $swapAction[l]; l)

End[]

$swapAction = Identity

EndPackage[]
```

**LISTING 3:** SortAuxG.m: Providing a hook.

The side effects of $swapAction result in this list of all intermediate permutations:

In[9]:= lists // Short

Out[9]//Short=
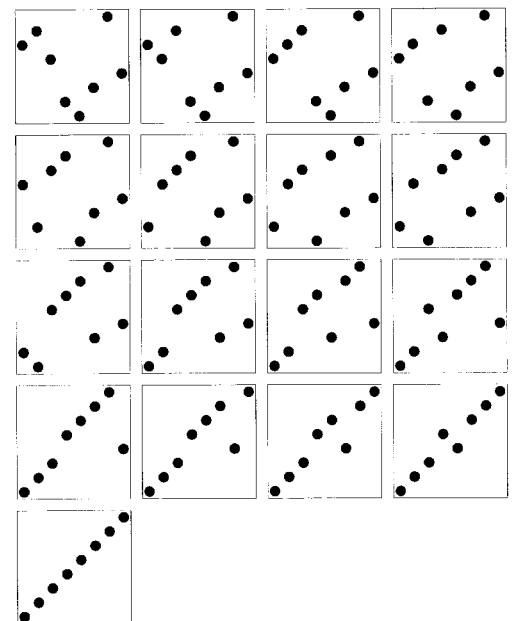    {{6, 7, 5, 2, 1, 3, 8, 4}, {6, 5, 7, 2, 1, 3, 8, 4},
    {5, 6, 7, 2, 1, 3, 8, 4}, <<13>, {1, 2, 3, 4, 5, 6, 7, 8}}

We convert each of these permutations into a permutation plot (without displaying it).

In[10]:= PermutationPlot[#, DisplayFunction -> Identity]& /@ lists;

Here is the animation (at least our static version of it).

In[11]:= ShowAnimation[%];



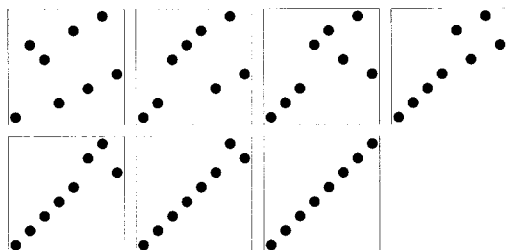Before we can perform another experiment, we have to reset the global variable.

In[12]:= lists = {};

This time, we use selection sort:

In[13]:= selectionSort[ list ];

We proceed as before to generate the frames of the animation.

In[14]:= PermutationPlot[#, DisplayFunction -> Identity]& /@ lists;

In[15]:= ShowAnimation[%];



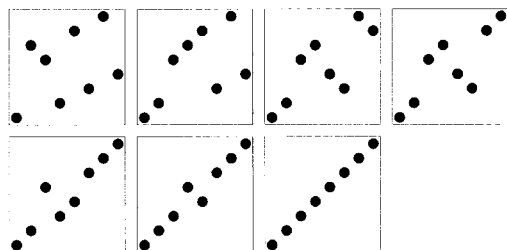And, finally, for quicksort:

In[16]:= lists = {};

In[17]:= quickSort[ list ];

In[18]:= ShowAnimation[
        PermutationPlot[#, DisplayFunction->Identity]& /@ lists];



At this time, we should think about writing a procedure sortAnimation[*sort*, *list*] that performs all these steps and produces the animation of *sort*[*list*] directly. To avoid disturbing the global variable $swapAction, we must use *dynamic binding*, that is, Block[{$swapAction = *value*}, ...]. This is one of the few remaining uses of Block that is not superseded by Module. Instead of the global variable lists, we use a local variable, declared in a Module.

The code declares its own static variable lists inside a Module, then changes the value of $swapAction and performs the sorting inside of the Block. The remaining statements are as before; they produce the list of graphics and animate it.

In[19]:= sortAnimation[sort_, list_] :=
        Module[{lists = {}},
            Block[{$swapAction = AppendTo[lists, #]&},
                sort[list]; ];
            ShowAnimation[
                PermutationPlot[#,
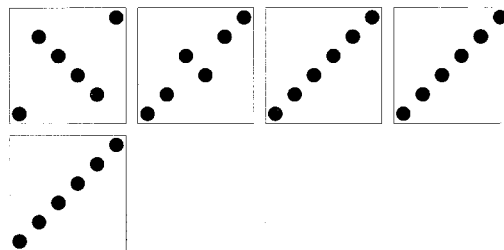                    DisplayFunction->Identity]& /@ lists ] ]

We use the new function to investigate the behavior of our three algorithms with a reversed list. Insertion sort performs badly.

In[20]:= sortAnimation[ insertionSort, Reverse[Range[6]] ];
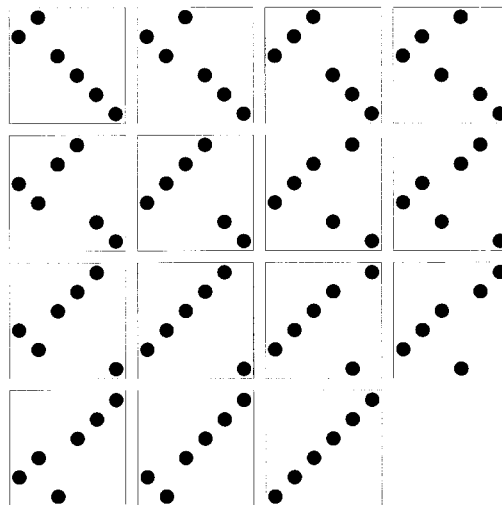


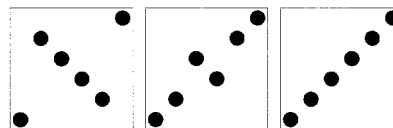Our version of selection sort may perform some trivial exchanges (that is, swap[*l*, *i*, *i*]).

In[21]:= sortAnimation[ selectionSort, Reverse[Range[6]] ];



Quicksort performs well: it sorts the entire array in the first phase.

In[22]:= sortAnimation[ quickSort, Reverse[Range[6]] ];



Note that the *Mathematica* front end allows a much simpler way to generate the animation. You can simply set $swapAction = PermutationPlot. The graphics will then be created one after another and neatly placed in a grouped sequence of cells. The notebook SortingExamples.ma in the electronic supplement contains examples using this method. The animations are best watched at a very slow speed. On some machines, the horizontal scrollbar can be used to traverse the frames by hand.

After playing around with toy examples, it is time to sort a few "real-world" lists. An animation or collection of pictures for each intermediate step is no longer feasible. Instead, we select a fixed number of equally spaced intermediate values.

The global value of $swapAction is still in effect, so we simply reset the variable lists.

In[23]:= lists = {};

We generate a random permutation of length 100.

In[24]:= list = RandomPermutation[100];

Insertion sort is not very efficient, so this computation takes a while:

In[25]:= insertionSort[list];

Here is the number of intermediate steps, that is, the number of exchanges that took place:

In[26]:= Length[lists]

Out[26]= 2538

We select 25 elements with an expression of the form lists[[{$e_1$, $e_2$, ..., $e_{25}$}]], where the 25 indices are generated with the help of Range. For shorter lists, it may not be possible to get exactly 25 elements, but that doesn't matter.

In[27]:= lists[[ Range[ 1, Length[lists],
                Ceiling[Length[lists]/25] ] ]];

Here are the 25 frames. The growing sorted initial segment is clearly visible.

In[28]:= ShowAnimation[
            PermutationPlot[#, DisplayFunction -> Identity]& /@ %
        ];
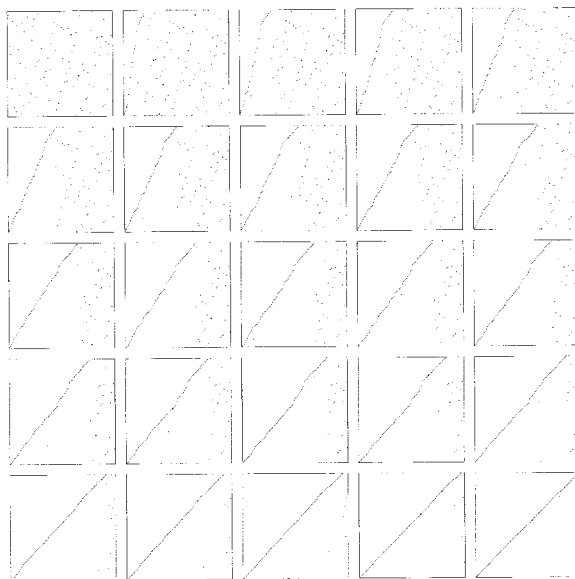


A second rule for sortAnimation with three arguments performs these steps.

In[29]:= sortAnimation[sort_, list_, frames_] :=
            Module[{lists = {}},
                Block[{$swapAction = AppendTo[lists, #]&},
                    sort[list];
                ];
                ShowAnimation[
                    PermutationPlot[#,
                        DisplayFunction->Identity]& /@
                    lists[[ Range[1, Length[lists],
                            Ceiling[Length[lists]/frames]] ]]
                ]
            ]

Here are the frames for quicksort with the same 100 element list. The effect of partitioning can be seen in the blocks forming along the minor diagonal.

In[30]:= sortAnimation[ quickSort, list, 25 ];
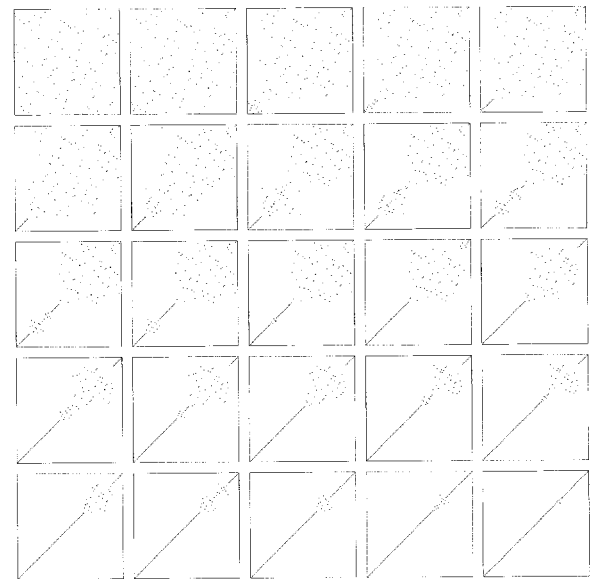


## Asymptotic Behavior

Sorting algorithms have been analyzed thoroughly and are theoretically well understood. Nevertheless, the applications in this section show how one might gather experimental data about less well understood programs.

Sorting a random list of $n$ elements with insertion sort requires on the order of $n^2/4$ comparisons and exchanges. The number of exchanges in selection sort is always equal to $n - 1$, but the number of comparisons is again proportional to $n^2$. Quicksort requires on the order of $n \log n$ comparisons and exchanges. This is the reason that quicksort is the method of choice in most applications.

It is easy to count the number of exchanges using $swapAction. We set it to be a function that simply counts how often it is called.

```
In[31]:=  swaps = 0;\
          $swapAction = swaps++& ;
```

We sort a random permutation of length 200 with quick-sort.

```
In[32]:=  quickSort[ RandomPermutation[200] ];
```

Here is the number of exchanges needed to sort it.

```
In[33]:=  swaps
Out[33]=  354
```

This command performs a number of experiments with random permutations of length $n$ and returns the average number of exchanges performed:

```
In[34]:=  avgSwaps[sort_, n_, trials_] :=
              Module[{swaps = 0},
                  Block[{$swapAction = swaps++&},
                      Do[ sort[RandomPermutation[n]], {trials} ];
                  ];
                  N[swaps/trials]
              ]
```

The call of SeedRandom guarantees reproducible experiments. You can leave it out if you plan to publish your results in the *Journal of Irreproducible Results*. We perform four measurements for sequences of values of $n$, up to $n = 100$.

```
In[35]:=  SeedRandom[1];\
          Table[ {n, avgSwaps[insertionSort, n, 4]},
              {n, 5, 100, 5} ];
```

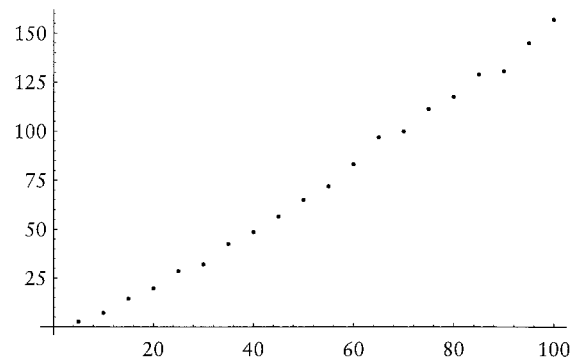Here is a graphic representation of the data for insertion sort.

```
In[36]:=  ListPlot[ % ];
```
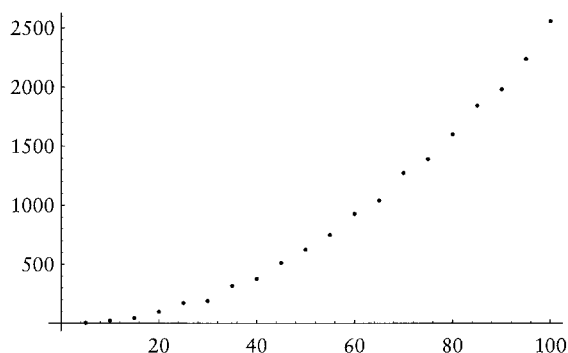


Here are the calculations and results for quicksort.

```
In[37]:=  SeedRandom[1];\
          Table[ {n, avgSwaps[quickSort, n, 4]},
              {n, 5, 100, 5} ];
```

```
In[38]:=  ListPlot[ % ];
```



## The Built-in Sorting Method

Can we analyze the built-in sorting method, Sort? It doesn't perform exchanges of the form used above, but simply re-assigns pointers to some internal data structure. There is one hook: we can give it our own ordering predicate.

We use this variable to count the number of comparisons performed.

```
In[39]:=  compar = 0;
```

Here is an ordering predicate that increments the counter and then behaves like the standard ordering function.

```
In[40]:=  myOrder[e1_, e2_] := (compar++; OrderedQ[{e1, e2}])
```

We sort a random permutation of 1000 elements.

```
In[41]:=  Sort[ RandomPermutation[1000], myOrder ];
```

Here is the number of comparisons performed.

```
In[42]:=  compar
Out[42]=  8711
```

## Conclusions

The static views of the animations in this article required an improved version of FlipBookAnimation.m. It is included in the electronic supplement and is also available on *MathSource*. The original version distributed with Version 2.2 of *Mathematica* cannot display lists of graphics with a prime number of frames in a reasonable way. When I developed it, I wasn't aware of the fact that GraphicsArray (used to assemble the frames in one picture) can cope with an incomplete last row of frames. As a consequence, lists of frames whose length had no proper divisors are displayed in a single row. The new version defines a global variable, Graphics`Animation`$Columns, that can be used to force the number of columns displayed to a certain value. We used it for the images in this article.

The three sorting methods from `Sorting.m` are taken from my textbook [Maeder 1993, Chapter 6]. It contains another method to visualize quicksort that shows the recursive subdivisions, which are more important for its performance than the number of exchanges.

## References

Maeder, Roman E. 1993. *Informatik für Mathematiker und Naturwissenschaftler – Eine Einführung mit Mathematica*. Addison-Wesley (Germany).

Sedgewick, Robert. 1990. *Algorithms in C*. Addison-Wesley.

Roman E. Maeder
ETH Zurich, Institute of Theoretical Computer Science,
ETH Zentrum IFW, 8092 Zurich, Switzerland
`maeder@inf.ethz.ch`

The electronic supplement contains the packages `Sorting.m`, `SortAux.m`, `SortAuxG.m`, the notebook `SortingExamples.ma`, and the new version of `FlipBookAnimation.m` that should replace the file in the `StartUp` directory of *Mathematica* Version 2.2.