An improved insert sort algorithm

Olli Nevalainen[1], Timo Raita[1] and Harold Thimbleby[2*]

[1] *Department of Computer Science, University of Turku, Turku, Finland.*
[2] *UCLIC, University College London Interaction Centre, London.*

SUMMARY

A simple and efficient insert sort algorithm is presented in Java.

This paper uses warping (automatic code inclusion from actual program source) to ensure reliability of the published code; the code extracts in this paper are guaranteed compiled, run and tested. The preceding paper in this journal discusses both the tool warp and its rationale.

KEY WORDS: Insert sort, Java, Quicksort, Sentinels, Warp.

Introduction

Many algorithms involve a search, and they typically require two types of test: one test checks for the element being searched for, and another test is required to check the search remains within bounds of the data structure being searched. Both of these tests can be combined into one by using a so-called sentinel. In particular, sentinels can be used in insert sort algorithms to make them more efficient, effectively halving the number of tests in a loop. An insert sort suggested by Thimbleby[1] repeatedly re-computes a sentinel value, which incurs a time penalty. Since the sentinel value for insert sort is an extremal element, one can be found and placed in position in linear time, and this need only be done once.

Using this idea, this paper gives Java code to sort a given array `a` into increasing order. Algorithms are presented for both stable and (the more efficient) unstable sorts.

The new algorithm takes three steps to sort an array `a`: find the sentinel; insert it into the array (at `a[0]`); and finally perform an efficient insert sort on the rest of `a`. Note that in Java an array `a` has bounds from `0` to `a.length-1` inclusive. Although the algorithm can sort arrays of integers and other simple values directly, we use the Java `Comparable` interface to make

*Correspondence to: H. Thimbleby, UCLIC, University College London, 26 Bedford Way, London, WC1H 0AP. Email: `h.thimbleby@ucl.ac.uk`

it more versatile: as written here, any class that supports the `Comparable` interface can be sorted.

A full analysis of the algorithm can be found elsewhere:[3] a possibly useful advantage of the new algorithm is that its analysis is simpler than Thimbleby's algorithm.[1] There are of course various implementation-specific optimisations (e.g., depending on the relative costs of assignment and comparison), and which we will not consider here: see Knuth for the classic reference on sorting.[4]

The algorithm's steps are as follows:

1. Identify sentinel position and value

```
int sentinel_position = 0;
Comparable sentinel_value = a[sentinel_position];
for( int i = 1; i < a.length; i++ )
    if( a[i].compareTo(sentinel_value) < 0 )
        sentinel_value = a[sentinel_position = i];
```

A common application of insert sort is as the final phase of quicksort.[2] In this case, insert sort is applied to a partially sorted array, and we know that the sentinel element will be found in the lowest k elements (where k is some implementation-dependent constant, typically around 16, generally much less than `a.length`, as here). For use in quicksort, the code would need to be rewritten to take advantage of this fact, but the time taken by the first step could be improved in speed by a factor of $O(\texttt{a.length}/k)$.

2. Insert sentinel value

There are two ways to place the sentinel in its proper position, depending on whether an unstable or a stable sort is required.

For the more efficient unstable sort (one where elements that compare equal may have their relative order changed), positioning the sentinel value takes just two assignments:

```
// unstable sort
a[sentinel_position] = a[0];
a[0] = sentinel_value;
```

For a stable sort the order of elements that compare equal must be preserved; this is achieved by shuffling all the elements `a[0]` to `a[sentinel_position-1]` up one, then inserting the sentinel value at `a[0]`. Note that the sentinel was chosen to be the minimum element with the smallest index, and therefore its assignment to `a[0]` will preserve the order of elements that compare equal to it.

```
// stable sort
for( int i = sentinel_position; i > 0; i-- )
    a[i] = a[i-1];
a[0] = sentinel_value;
```

Stable sorts are required when wanting to preserve an existing order if objects are resorted without determining their full value, as when comparing only one of several object fields —

as when sorting people, having first sorted by name, one might want to preserve the ordering of names for people of the same age when subsequently sorting by age. When sorting simple values, such as integers, an unstable sort is always sufficient.

3. Insert sort

Given that `a[0]` is now a minimal element of the array (namely the sentinel), the next step of the algorithm, the insert sort itself, can therefore: (*i*) avoid any bounds check in its inner loop test, since the current item to be inserted, `v`, is guaranteed to be inserted within range; and (*ii*) start its outer loop at `i` = 2, since as `a[0]` is the sentinel the first two elements (`a[0]` and `a[1]`) of the array must be in correct sorted order already.

The final step of the algorithm is stable: thus the complete algorithm is stable provided the previous step (2) chosen for the algorithm is the stable alternative.

```
for( int i = 2; i < a.length; i++ )
{   int j = i;
    Comparable v = a[j];
    while( a[j-1].compareTo(v) > 0 )
    {   a[j] = a[j-1];
        j--;
    }
    a[j] = v;
}
```

REFERENCES

1. H. W. Thimbleby, "Using Sentinels in Insert Sort," *Software—Practice and Experience*, **19**(3):303–307, 1989.
2. R. Sedgewick, *Algorithms*, 3rd. edition, *Algorithms in Java*, **III**: Sorting, Addison-Wesley, 2002.
3. O. Nevalainen & T. Raita, "Insertion Sort with a Static or Dynamic Sentinel?" Technical Report, Department of Computer Science, University of Turku, Turku, Finland, 1990.
4. D. E. Knuth, *The Art of Computer Progamming*, **3** (Sorting and Searching), Addison Wesley, 2nd. ed., 1998.

Appendix: Complete Java source

A paper that includes code could be written in many ways. The code might be cut-and-pasted from working programs, it might even be completely untested. A reader cannot easily tell, for the end results, however it is done, look similar. In fact all the code exhibited in this paper was 'warped' or automatically extracted and marked up from a program source file that has been compiled, debugged and checked. This is far easier for the author and significantly increases the reliability of the published result.

The following is the complete Java source file, from which the code sections above were automatically extracted by the warp tool. This code would not normally be included in a paper, but is shown here to show how warpworks. warp converts a program source file to XML, passing through directly any comments that contain well-formed XML. It is then a

trivial matter using warp (or other XML tools) to find and extract the named sections of code and mark them up. In the present case warp extracted the sections of code used in the body of this paper and marked them up in LaTeX. As can be seen warp can also translate to HTML as well as LaTeX from a single document; the HTML version of the file can be found on the web site.

More details of warp can be found in the preceding paper ("Explaining code for publication") in this journal or at `http://www.uclic.ucl.ac.uk/harold/warp`.

```
/* <h1>Source code for Nevalainen, Raita &amp; Thimbleby paper, 2002</h1>
   See <a href="http://www.uclic.ucl.ac.uk/harold/warp"><!--
   -->www.uclic.ucl.ac.uk/harold/warp</a> for more details.
   <pre> */

/**
 * @author Harold Thimbleby, h.thimbleby@ucl.ac.uk
 * @version 1
 */

import java.lang.Comparable;

class InsertSort
{   final static int STABLE = 1, UNSTABLE = 2;

    public static void sort(Comparable a[], int sortType)
    {   // <warp file='identify.tex'>
        int sentinel_position = 0;
        Comparable sentinel_value = a[sentinel_position];
        for( int i = 1; i < a.length; i++ )
            if( a[i].compareTo(sentinel_value) < 0 )
                sentinel_value = a[sentinel_position = i];
        // </warp>

        if( sortType == UNSTABLE )
        {   // <warp file='unstableinsert.tex'>
                // unstable sort
                a[sentinel_position] = a[0];
                a[0] = sentinel_value;
            // </warp>
        }
        else
        {   // <warp file='stableinsert.tex'>
                // stable sort
                for( int i = sentinel_position; i > 0; i-- )
                    a[i] = a[i-1];
                a[0] = sentinel_value;
            // </warp>
        }

        // <warp file='sort.tex'>
        for( int i = 2; i < a.length; i++ )
        {   int j = i;
            Comparable v = a[j];
            while( a[j-1].compareTo(v) > 0 )
            {   a[j] = a[j-1];
                j--;
            }
            a[j] = v;
        }
        // </warp>
    }
} // </pre>
```

The code can be run, of course, to test it. Complex output often requires further explanation and faithful presentation, just as code does. Warp can be used to process output too: all that is necessary is to insert appropriate XML comments (such as `/* <save-this> */`) in the output stream and to warp *that*. The output will be marked up so it, or selected parts of it, can be included into the explanation.

The (not very surprising) warped output of a test run of the code from the example (included directly and automatically into this paper) is as follows:

```
Stable sort
    Before sorting [2, 9, 3, 4, 1, 5, 8, 1, 4, 8, 7, 6]
    After sorting  [1, 1, 2, 3, 4, 4, 5, 6, 7, 8, 8, 9]
Unstable sort
    Before sorting [2, 9, 3, 4, 1, 5, 8, 1, 4, 8, 7, 6]
    After sorting  [1, 1, 2, 3, 4, 4, 5, 6, 7, 8, 8, 9]
```