# 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, and is presented in stable and unstable variants. The usual double test of insert sort can be avoided by using a sentinel, but sentinels create minor problems — such as choosing an appropriate value and requiring extra memory. The insert sort here avoids both problems.**

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 the 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], which has the advantage of neither requiring extra memory nor a special value (such as `MAX_VALUE`) that cannot be sorted, however repeatedly re-computes a sentinel value, an operation which itself 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.

*Note.* This document is a supplement to the previous paper (*Software—Practice & Experience*, Vol.XX, No.XX:pp.XX–XX, 2003). It illustrates how an automatic code inclusion technique, warping, can be used to produce a free standing paper with reliable code.
*Correspondence to: H. Thimbleby, UCLIC, University College London, 26 Bedford Way, London, WC1H 0AP. Email: `h.thimbleby@ucl.ac.uk`
†Professor Timo Raita died in April 2002.

The new algorithm takes three steps to sort an array `a`: (*1*) determine the sentinel; (*2*) insert it into the array; and (*3*) 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 it more versatile: as written here, any array of a class that supports the `Comparable` interface can be sorted. Otherwise for simple values, whether in Java or other languages, expressions written here as `a[i].compareTo(v) < 0` may be rewritten as `a[i] < v`.

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 in the array*

```
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 into correct position in the array*

The sentinel could in principle be inserted at either end of the array; it is however more efficient to insert it at `a[0]`. 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 the rest of the array*

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 (namely, `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;
}
```

The two statement body of the `while` loop can be written more concisely and possibly faster as, for instance, `a[j] = a[--j]`, but this short-cut may be ill-advised in some languages as it depends on the order of evaluation of the left and right hand sides (e.g., it does not work in *Mathematica*). Clarity is more important, and it is generally better to leave such trivial optimisations to compilers.

The source code used here is available at `http://www.uclic.ucl.ac.uk/harold/warp`.

## 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.