SOFTWARE-PRACTICE AND EXPERIENCE, VOL. **19**(3), 303–307 (MARCH 1989)

Short Communication

USING SENTINELS IN INSERT SORT

HAROLD THIMBLEBY
Department of Computing Science, University of Stirling, Stirling, FK9 4LA, U.K

SUMMARY

The inner loop of insert sort can he simplified by using a sentinel value. We suggest a way of avoiding the extra storage normally required for the sentinel, and a way of avoiding the prior, and possibly restrictive, choice of its value. Algorithms are given in Pascal.

KEY WORDS: Sorting, Insert sort, Quicksort, Sentinel

NOTE: This paper was scanned; please refer to the original article in the journal for an accurately proof-read copy.

Although insert sort takes quadratic time in the worst case it has a very efficient inner loop. Its bestcase computing time is linear, which occurs when the data are already in sorted order. It is the method of choice for data stored in linked lists or small arrays.

Algorithm 1 is typical, and is written in extended Pascal.
Implementing the conditional '**cand**' operation (to guard against the error of subscripting the array a when j–1=0) is problematic in Pascal. Though other languages (Ada, C, Algol, etc.) can overcome this problem of expression, the algorithm nevertheless performs two tests within the loop when only one is sufficient, as we shall see below. There are other optimizations, covered by Knuth,' which we shall not discuss here, for instance:

(a) various minor transformations, such as initializing j to j–1, unwinding the inner loop and noticing that a[i]: =v is unnecessary when the inner loop is not executed

(b) more substantial transformations, such as inserting elements more than one at a time (e.g. taking advantage of monotonically increasing subsequences), moving elements more than one at a time (e.g. Shell sort) and non-linear methods such as binary insertion.

One awkward solution to the peculiarly Pascal problem would be to change the inner loop to **while** (j>2) **and** (a[j–1] > v) **do** ... and follow it with a special check for terminating at j=l, in which case an extra iteration is required.2 Apart from being a Pascal programming trick, this solution has little to commend it: it still requires two tests on each iteration.

Another solution would be to extend the array to include a dummy element a[0] so that there could be no subscript errors (and therefore the standard Pascal **and** would be adequate).

The value of the dummy element a[0] will be arbitrary. Thus it may be possible to choose a sentinel value for a[0], such that a[0]≤v for each possible v. This choice would obviate the necessity of the j>l test, since a[j–l]>v would be false for j=1. Assuming that SmallSentinel is a suitable sentinel value for a[0], Algorithm 2 results. Algorithm 2 contains a faster inner loop, having one comparison less per iteration, and has no Pascal problem implementing a conditional **and** operation.

Whatever the language, using a sentinel has the advantage that the number of tests in the inner loop is reduced, but has the twin disadvantages of requiring an extra element in the array and of knowing a sentinel value at least as small as any array element.* Even if a constant such as maxint is available (and we are sorting integers!), its correct use (e.g. to determine the minimum representable integer value) is problematic without knowing the machine architecture (e.g. that it is 2s complement).

However, by performing N–2 extra assignments, it is possible to avoid choosing a particular (and possibly restrictive) sentinel value. Algorithm 3 is due to Wirth,' and it is surprising that it is not more widely known. It relies on the fact that for each v inserted, a sentinel value less than or equal to v is v itself.

We now note that the extra array element needed for the sentinel may be a serious problem when writing a general-purpose routine, which may have no control over the memory allocation of the data it has to sort.

The disadvantage of extending the array can be avoided by using a[l] as the initial location of the sentinel value. The old value of a[1] has to be saved for later. The subarray a[2..N] is then sorted as before (but without a j>2 test). Then the displaced value of a[1] is inserted into the subarray a[2..N]. This insertion needs either a double test (with the usual problems) or another sentinel. In fact, this sentinel can be stored at a[N]. The somewhat obscure algorithm, Algorithm 4, demonstrates this solution.

Taking into account the reduced range of the for-loop, the time complexity of this solution is approximately a constant (five assignments) worse than the simple sentinel solution (which, indeed, would take another two assignments to save and restore a[0] if necessary). Note that Algorithm 4 requires knowing *both* a small and a large sentinel, since inserting the original a[l] value (saved in sl) requires an insert sort in the opposite direction. (Algorithm 4 has made the tacit assumption that N>l, whereas the other algorithms discussed here assume the weaker N–l.)

However, *both* disadvantages of the conventional sentinel are avoided by a simple algorithm based on the following observation: a sentinel value at a[0] is only required when a[l]>v. In this case, since a[l..N–1] are sorted, the value v will

---

* Or, a sentinel at least as big as any element, if the array is sorted in either the opposite order (so a[l] is instead largest at the end of the sort) or in the opposite direction (so the outer loop instead decrements i from N–1).

necessarily be inserted at a[l]. Alternatively, when a[1]≤v, then v will be inserted at some a[j] with j>l. Thus there are two cases which can be implemented by separate loops each with a single test. Algorithm 5 is suggested.

Algorithm 5 requires neither the extra array element a[0], nor prior knowledge of the range of values in a in order to choose a sentinel value. One optimization based on the observation that the t Inserting sl could be achieved without a sentinel, using a double test loop, based on the inner loop of Algorithm 1. This means that the overhead of the double test loop occurs for only one insertion, rather than all N≥1.

assignment a[l]:=v is not often executed (the comparison, a[l]>v is true approximately ln N times), is that it may be worth keeping a copy of a[l] in a variable (but the assignment to a[1] must be retained, since it is required for both inner loops to work correctly). This would provide a marginal improvement to the speed of the if a[1] > v then test (by avoiding the array subscription).

A full analysis and comparison is hardly worth while in the present context, since it would rely on assumptions about data distribution and representation, details of coding, and instruction repertoire and execution times. Instead, it is more interesting to compare Algorithm 5 with the earlier algorithms:

(a) Algorithm 5 is more efficient than the original (Algorithm 1), having taken a test outside the inner loop.

(b) However, in comparison with the conventional sentinel solution (Algorithm 2), Algorithm 5 requires at most an extra N comparisons depending on how the for–loop is optimized.

(c) Algorithm 5 avoids the memory overhead of a sentinel (and, of course, the programming overhead of organizing memory, e.g. to avoid subscript errors, so that a sentinel can be used).

(d) If the cost of comparison and assignment are comparable, Algorithm 5 will be faster than Wirth's Algorithm 3.

In all cases, the inner for-loop in Algorithm 5 is an efficient replacement for the *longest* iterations of the while-loop in Algorithms 1–4. The simple for-loop in Algorithm 5 can probably be performed directly by hardware (as a block move instruction). Indeed, a list insert-sort algorithm that maintained a pointer to the tail of the list could perform this step in constant time. This approach results in the next algorithm discussed.

The final algorithm shown here (Algorithm 6) is a version of Algorithm 5 for linked lists. Linked lists allow the for-loop to be replaced by a simple assignment. Note a few minor details of the coding: (1) the algorithm is *destructive* — no extra memory is used — by reusing list nodes; (2) a standard programming idiom has been used to achieve a fast inner while loop, avoiding the overhead of running two pointers down the list; (3) the list is not well-formed until the final assignment, and does not need to be (since t points invariantly to the tail of the sorted list, which is the sentinel beyond which the inner loop will not search).

Quicksort provides a useful application of Algorithm 5. Typically, Quicksort partitions an array into segments of length less than or equal to k, where k is

deemed to be a threshold below which the insert sort is faster than Quicksort. Algorithm 5 is used to sort the first k elements of the array. Then Algorithm 2 (without the explicit sentinel assignment) is used to sort the remaining Wk elements, for which a[k] will serve as a sentinel (the Quicksort algorithm guarantees that every element in the first k is less than or equal to the rest of the array elements).

SUMMARY

Algorithms 1–6 can be summarized as follows:

1. The conventional insert sort. It performs O(N') avoidable subscript comparisons, and, besides, leads to messy implementations in some languages.
2. The conventional insert sort with sentinel. The most efficient version considered here, but it requires an extra array element and a sentinel value. Choosing a sentinel value requires making assumptions or restrictions about the range of data to be sorted.
3. An algorithm avoiding choosing a fixed sentinel, but still using extra memory.
4. An obscure insert sort with sentinels. No extra array element, but requiring lower and upper sentinels. Only a constant worse than Algorithm 2. It can be modified along the lines of Algorithm 3 to avoid choosing fixed sentinel values.
5. A new algorithm that requires no extra memory and no choice of sentinel value, and therefore no assumptions about the range of data. It performs more comparisons than Algorithm 2, but against this must be set the fact that one of its inner loops may be coded more efficiently (the other being identical to the inner loops of Algorithms 1–4). Algorithm 5 is more efficient than Algorithm 3, particularly since the slowest inner loops are replaced.
6. The last algorithm considered is an efficient, linked list version of Algorithm 5. It performs approximately N - ln N data swaps. It uses a destructive in-place method, that is, it uses no extra list memory.

Algorithms 5 and 6 are clearly better in terms of lack of restrictions than the conventional Algorithms 1–3.

REFERENCES

1. M. Amnoodeh, *Abstract Data Types*, Macmillan, 1988.
2. D. E. Knuth, *The Art of Computer Programming*, Volume **3**, Sorting and Searching, Addison–Wesley, 1973.
3. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

```
{ sort a[l..N] in place }
for i := 2 to N do
          v := a[i];
j := i;
while (j > 1) 'cand' (a[j–1] > v) do
          begin     a[j] := a[j–1];
                    j := j–1
          end;
end;
a[1] := v
```

Algorithm 1

```
{ sort a[l..N] in place }
{ a[0] needed for sentinel }
a[0] := SmallSentinel;
for i := 2 to N do
begin     v := a[i];
          j := i;
          while a[j-11 > v do
          begin     a[j] := a[j–1];
                    j := j–1
          end;
          a[i] := v
end
```

Algorithm 2

```
{ sort a[l..N] in place }
{ a[0] needed for sentinel }
for i := 2 to N do
begin     v := a[i];
          a[0] := v;
          j := i;
          while a[j–1] > v do
          begin     a[j] := a[j–1];
          end;
          a[i] := v
end
```

Algorithm 3

```
{ sort a[l..N] in place }
:I= a[l];
a[l] := SmallSentinel;
for i :=3toNdo begin v := a[i];
          j := i;
          while a[j–1] > v do
          begin a[j] := a[j–1];
          end;
          a[i] := v
end;
j := 1;
s2 := a[N];
a[N] := BigSentinel;
while a[j+l] < sl do
begin a[j] := a[j+11;
end;
if s2 < s1 then
begin     a[j]:= s2;
          a[N] := al
end;
j := j+1
else
          begin a[i] := sl; a[N] := s2
end
```

Algorithm 4

```
{ sort a[l..N] in place }
for i := 2 to N do v := a[i];
if a[l] > v then
          begin
                    for j := i downto 2 do
                              a[j] := a[j–1];
                    a[l] := v
          end
else
          begin     j := i;
                    while a[j–1] > v do
                    begin     a[j] := a[j–1];
                              j := j–1
                    end;
                    a[I] := v
          end
end
```

Algorithm 5

```
{ Algorithm 5 modified for lists }
if a <> nil then
begin     h := a; t := a;
          a := a-.next;
          while a <> nil do
          begin     anext := a^.next;
                    if t-.value > a^.value then
                    begin     t-.next := a;
                    end
                    else
                    begin     j := h;
                    end;
                    a := anext
          end;
          a := h;
          t-.next := nil
end
while j-.value > a-.value do
          j := j^.next; a-.next := j-.next;
j-.next := a;
swap(a-.value, j-.value);
if j = t then
          t := j^.next
```

Algorithm 6