Copyright Notice

The manuscript

EWD 817: An introduction to three algorithms for sorting in situ

was published as

Information Processing Letters 15, 3: 129–134, copyright © 1982.

It is reproduced here by permission of the publisher, Elsevier Science.

Single copies of the manuscript may be downloaded and printed for the reader's personal research and study.

An introduction to three algorithms for sorting in situ

Edsger W.Dijkstra and A.J.M. van Gasteren

Authors' addresses:

Edsger W.Dijkstra, Burroughs, Plataanstraat 5, 5671 AL NUENEN, the Netherlands.

A.J.M. van Gasteren, Dept. of Mathematics and Computing Science, University of Technology, 5600 MB EINDHOVEN, the Netherlands.

Abstract. The purpose of this paper is to give a crisp introduction to three algorithms for sorting in situ, viz. insertion sort, heapsort, and smoothsort. The more complicated the algorithm, the more elaborate the justification for the design decisions embodied by it. In passing we offer a style for the presentation of non-trivial algorithms that seems to have its merits.

* * *

Introduction and terminology

The purpose of this paper is to introduce three algorithms for sorting in situ. They are dealt with in a single paper so that the introductions can share what they have in common; they are presented in the order of increasing complexity. The rest of this section is devoted to the terminology that is used throughout this paper and to a central subalgorithm.

We assume the reader to be familiar with the notion of a "sequence of elements" and with the corresponding notion of a "successor" or "predecessor" of an element in the sequence.

We define a chain to be a finite, non-empty sequence of elements, each having (or owning) an integer value. Each element with a successor in the chain is called the father of that successor; each element with a predecessor in the chain is called the <u>son</u> of that predecessor. The element without father is called the <u>root</u> of the chain, the element without son is called the <u>leaf</u> of the chain. (Coincidence of root and leaf means the chain being a one-element chain.)

The <u>offspring</u> of an element is a recursively defined set of elements: the offspring of a leaf is empty, the offspring of a father is its son together with that son's offspring. Element e0 <u>dominates</u> element e1 means that e0 has a value that is at least the value owned by e1 ; element e dominates a set of elements means that e dominates each member of that set.

For an element of a chain, the predicate <u>strong</u> means that the element dominates its offspring; a chain is <u>descending</u> means that each father in the chain dominates its son, and we leave to the reader to convince himself that "chain c is descending" is equivalent to

P0: (<u>A</u> e: e <u>in</u> c: strong e) .

We are interested in making a chain c descending --i.e. in establishing P0-- without changing the bag of values owned by its elements. The latter requirement is met by the usual technique: values owned are changed only by swapping element values. We are in particular interested in this task when chain c initially satisfies

P1: (<u>A</u> e: e <u>in</u> c: strong e <u>or</u> e = root of c) .

(A chain c satisfying P1 is, for instance, formed by changing the value owned by the root of an initially descending chain or by extending an initially descending chain at root side by a further element.)

In view of P1 we introduce --with the intention that it be maintained--

P2: (<u>A</u> e: e <u>in</u> c: strong e <u>or</u> e = w) ,

which --thanks to P1-- can be established by the initialization w:= root of c . (Note that variable w is not of type integer but of type element.) Relation P2 is of interest since we are allowed to conclude P0 from

P2 <u>and</u> "w has no son it does not dominate" .

Note. "w has no son it does not dominate" can occur in more than one way: w may have no son at all or w may be a father, but of a son whose value is small enough. (End of Note.)

The program establishing P0 is

```
{P1} w:= root of c {P2: invariant}
; do "w has a son s it does not dominate"
     → "swap the values of w and s"; w:= s
  od {P0}    .
```

Proof. The above program terminates since w:= s reduces the cardinality of w's offspring. Furthermore relation P2 is invariant provided the swap establishes

(A e: e in c: strong e or e = s) .

For all elements e such that $e \neq w$ and $e \neq s$, P2 implies (strong e); since neither the bag of values owned by the offspring of such an e nor e's own value is changed by the swap, (strong e) still holds.

For $e = w$, we observe that prior to the swap s dominates its father w on account of the guards and dominates its offspring on account of (P2 and $s \neq w$); hence, after the swap (strong e) holds.

For $e = s$, $e = s$ holds. (End of Proof.)

The algorithms to be presented sort the integer array $m(i: 0 \leq i < N)$ into ascending order.

Insertion sort

We consider relation P3 , given by

P3: the elements of $m(i: 0 \leq i < n)$ with
(A i: $1 \leq i < n$: m(i) is the father of m(i-1))
form a descending chain .

Relation P3 is of interest since it trivially holds for $n = 1$ and enjoys the property that (P3 and $n = N$) allows us to conclude that array $m(i: 0 \leq i < N)$ is in ascending order. We can apply the algorithm of the previous section, which leads to the following algorithm, known as "insertion sort".

```
|[ n: int; n:= 1 {P3: invariant}
 ; do n ≠ N →
        |[ w: int; w:= n
         ; do w > 0 cand m(w) < m(w-1)
                → m:swap(w, w-1); w:= w - 1
           od; n:= n + 1
        ]|
   od
]|   .
```

In the best case, i.e. if array m is ascending to start with, the above algorithm is of order N . In general, however, it is of order N^2 (both in number of comparisons and in number of swaps).

The operation "sift"

The notion of a chain, in which each father has exactly 1 son, can be generalized by allowing fathers to have a number of sons. The resulting structure is well-known: it is known as a rooted tree. The definition of offspring is generalized accordingly: the offspring of a leaf is empty, the offspring of a father consists of its sons together with their offsprings. A *descending tree* is defined as a rooted tree in which each element dominates its offspring.

A descending tree enjoys the property that its root dominates the entire tree and, hence, owns a maximum value; the arrangement is attractive since the average distance from the root grows logarithmically instead of linearly with the number of elements. This fact underlies the existence of sorting algorithms that are in general of order N.log N instead of of order N^2 .

The operation sift establishes for a tree t relation P4 , given by

P4: (*A* e: e *in* t: strong e)

provided initially P5 holds, where

P5: (*A* e: e *in* t: strong e *or* e = root of t) .

Situation P5 can, for instance, arise under the following circumstances:

circumstance a: the value owned by the root of a descending tree has been changed;

circumstance b: a little forest of descending trees and an additional element have been formed into a single tree with the additional element as its root and the trees of the forest as its (first-generation) subtrees;

circumstance c: from two descending trees a single tree has been formed by "grafting" one upon the other, i.e. by declaring the root of the one an additional son of the root of the other.

Given P5 , operation sift establishes P4 , i.e. makes the tree descending, by applying the algorithm of our introduction along a judiciously chosen chain starting at the root: the root of the chain is the root of the tree, the successor of an element in the chain is among that element's sons in the tree one that dominates those sons. (Consequently the leaf of the chain is one of the leaves of the tree.)

Proof. For e an off-chain element of the tree, (strong e) continues to hold since neither its own value nor the values of its offspring have been changed. For e on the chain, we separately consider its offspring on the chain and its off-chain offspring: e dominates its offspring on the chain as before and it therefore dominates its off-chain offspring because, due to the judicious choice, each element on the chain dominates its off-chain sons (if any). (End of Proof.)

The great invention embodied by sift underlies the following algorithms for sorting in situ; they are worst-case of order N.log N .

Heapsort and smoothsort

In their second phases these algorithms maintain P6 , where

P6: $(\underline{A}\ i, j: 0 \leq i < j\ \underline{\text{and}}\ q \leq j < N: m(i) \leq m(j))$,

which vacuously holds for q = N and enjoys the useful property that (P6 and q = 1) allows us to conclude that array m is ascending. Relation P6 means that $m(i: q \leq i < N)$ has its final value and that the rest of the computation can be confined to manipulating the so-called "unsorted prefix" $m(i: 0 \leq i < q)$. The purpose of these manipulations is to ensure that the "rightmost" element of the unsorted prefix dominates the elements to the "left" of it, so that q can be decreased by 1 without violating P6 . (For the elements of array m the order of increasing subscript is referred to as the order from "left" to "right".)

If nothing is known about the unsorted prefix, one needs a complete scan of it to locate its maximum value, which then can be placed in its rightmost position. The ensuing algorithm is linear in the number of swaps, but quadratic in the number of comparisons. If the unsorted prefix is known to be ascending, neither swaps nor comparisons are needed, but this would be begging the question. The moral is that for a more efficient sorting algorithm we need something in between: in order to facilitate the location of its maximum value, the unsorted prefix has to satisfy some relation, helpful to that purpose but less demanding than being completely sorted. After a first phase, which establishes this relation for the entire array, the second phase maintains it for the shrinking unsorted prefix.

It is here that the descending tree enters the picture: if the elements of the unsorted prefix are the vertices of a descending tree with a known root, we know where to find the maximum value. Two sweetly reasonable choices for the root of that descending tree present themselves: the leftmost or the rightmost element of the unsorted prefix. The first choice leads to heapsort, the second one to smoothsort. In both cases sift is used in the first phase for building up the descending tree covering the unsorted prefix of length N and in the second phase --during which the length of the unsorted prefix shrinks to 1-- for its maintenance.

Note. (To avoid temporary confusion, we warn the reader that the binary tree occurring in this Note has nothing to do with the descending trees occurring outside it.) The N! distinct possible computations can be arranged in a binary tree with the first comparison as its root and, depending on its outcome, the rest of each computation in the one subtree or the other. Each computation then corresponds to a path from the root to a leaf and the average number of comparisons is equal to th average distance of a leaf from the root. Because, independently of its shape, this binary tree has N! leaves and, hence, N!-1 "comparison nodes", this average distance is minimal if the distances differ at most 1 , i.e. if best- and worst-case behaviour (as measured by the number of comparisons) are as equal as possible. Heapsort approaches such a behaviour, which (thanks to Stirling's formula for $N!$) is of order $N.\log N$. Smoothsort is worst-case of order $N.\log N$, but best-case of order N , with a smooth transition between the two. From the above it follows that smoothsort requires on the average more comparisons than heapsort. (End of Note.)

Heapsort

In heapsort the leftmost element of the unsorted prefix is chosen as the root of the tree; it is followed by the nodes of the first generation, which are followed by the nodes of the second generation, etc.. (An example is a binary tree in which $m(i)$ has $m(2.i+1)$ and $m(2.i+2)$ as its sons.) In the second phase, in which q is decreased from N to 1 , the values of the leftmost element and the rightmost element are swapped so that the unsorted prefix can shrink by one element under invariance of P6 ; applying sift to the root $m(0)$ restores the father/son inequalities in the unsorted prefix (circumstance a).

The first phase maintains that each $m(i)$ with $i \geq p$ dominates the sons it has. This is initially established by choosing p sufficiently large: the elements of $m(i: p \leq i < N)$ can then be viewed as a forest of leaves. Then p is repeatedly decreased by 1 , each decrease being followed by an application of sift to $m(p)$ (in general circumstance b). The number of trees in the forest covering $m(i: p \leq i < N)$ eventually decreases; when $p = 0$, the forest has become one big tree and the second phase can start. Note that an initially increasing array is completely scrambled in the first phase; the second phase unscrambles it again.

Note. For simplicity's sake, the unsorted prefix is usually covered by a binary tree. A ternary tree, however, leads to smaller worst-case numbers of comparisons and swaps. (End of Note.)

Smoothsort

In smoothsort, the descending tree covering the unsorted prefix during the second phase is a so-called "leftward tree", i.e. a tree in which each son is situated to the left of its father. As a result, the rightmost element of the unsorted prefix is the root of the tree and, hence, dominates all other elements of the unsorted prefix. Note that leftward trees are the only ones permissible under the constraint that smoothsort leave an initially increasing array unchanged all through the computation. This constraint is strongly suggested by the aim that smoothsort be best-case of order N .

Its rightmost element dominating all its others, the unsorted prefix can be shortened by one element without violation of P6 , and it is here that smoothsort strongly deviates from heapsort. In the second phase, heapsort's tree is pruned leaf by leaf, i.e. each time the unsorted prefix shrinks, the set of father/son inequalities to be maintained is reduced by one. In smoothsort's second phase, however, the tree is pruned at its root: it becomes a forest of as many (descending) trees as the removed root had sons and this forest has to be rebuilt into a single descending leftward tree by the introduction of new father/son inequalities. (The obligation to keep track of the changing shape of the tree has no analogue in heapsort.)

The forest is rebuilt into a single descending leftward tree by (repeatedly) grafting the root of a tree of the forest upon such a root to the right of it (sift being applied in circumstance c). For the sake of convenience and controllable worst-case behaviour --note that sift is not of logarithmic time complexity if the number of sons with the same father is unbounded-- in the design of smoothsort it has been decided not to introduce fathers with more than 3 sons. In view of the grafting this requires that each father with 3 sons has nothing but its own offspring to its left, which is most easily achieved while keeping the unsorted prefix the tree's postorder traversal. (The postorder traversal of a tree is a special permutation of its vertices, viz. the concatenation of the postorder traversals of its first-generation subtrees followed by its root.)

The constraint on fathers with 3 sons, combined with the choice of a postorder traversal, makes the unsorted prefix a concatenation of the postorder traversals of descending binary trees whose roots have values that are ascending in the order from left to right. The only design decision left is the choice which binary trees to admit.

The binary trees admitted are the so-called Leonardo trees LT_i : LT_0 and LT_1 both consist of a single leaf, LT_{i+2} has LT_{i+1} as its left subtree and LT_i as its right subtree. To minimize the number of Leonardo trees used, the concatenation covering the unsorted prefix starts at the left with the postorder traversal of the largest possible Leonardo tree, and so for the remainder. As a result, the unsorted prefix is covered by as few Leonardo trees as possible. (Leonardo trees have been preferred to balanced binary trees because, on the average, 25 % more trees are needed for coverage by the latter.)

When, in the second phase, the unsorted prefix shrinks, two cases are distinguished. If it ended on LT_0 or LT_1 , that one-leaf Leonardo tree is removed and the values of the roots of the remaining Leonardo trees are still ascending; if it ended on LT_{i+2} , after shrinking it ends on LT_{i+1} and LT_i instead: in general, a three-tree forest has emerged and twice the left-most tree is grafted upon the root of the tree immediately to its right.

As in heapsort, smoothsort's second phase is preceded by a first phase in which the unsorted prefix of length N (i.e. covering the whole array) is prepared. In contrast to heapsort, this preparation starts from the left: the length q of the prepared prefix is initialized at 1 and repeatedly increased by 1 until $q = N$.

The first phase's main task is to see to it that after each increase of q the Leonardo trees covering the prefix are descending. If the prefix ended on LT_i preceded by LT_{i+1} , these trees and the new element are absorbed in a new LT_{i+2} , to whose root sift is applied (circumstance b); otherwise a one-node tree is added, which is descending by definition.

The first phase's second task is to see to it that eventually --i.e. when $q = N$ -- the roots of the Leonardo trees have values that are ascending in the order from left to right. To this purpose a grafting operation is inserted (circumstance c) each time the prepared prefix comes to end on the next Leonardo tree of the final coverage.

In order to arrive at an algorithm of order N in the best case, a stack records --in both phases-- which Leonardo trees cover the prefix. (Because each

Leonardo tree occurs at most once in a coverage, a bit stack, in fact, suffices; its maximum length is $(\log N)/(\log(\frac{1}{2} + \frac{1}{2}\sqrt{5}))$ bits.)

Concluding remarks

The reader will have noticed that we have presented the three algorithms in decreasing degrees of detail: for insertion sort the code has been given in full, designing from our description a nice code for heapsort is pretty straightforward, but doing the same for smoothsort will probably require a few days. (In the case of heapsort one has to design, for instance, a sift that nicely deals with the rare situation of a father with a single son; for smoothsort one has to design, for instance, an efficient merge of the first phase's two tasks.) We did so on purpose: we wanted to give descriptions that, once well-understood, can be remembered. In view of that goal we have omitted as much detail as we thought permissible. We invite the reader to accept that in smoothsort's case so much detail has been omitted: even by today's standards, smoothsort is a fairly sophisticated algorithm. We have tried to instruct the reader without snowing him under; we hope we have succeeded.

Intentionally we have given the last two algorithms a less formal treatment than the first one, thus avoiding the introduction of the machinery needed for the formal treatment of rooted trees. It should be possible to extrapolate what that machinery would look like from our fairly formal treatment of the chain, a treatment that has been included precisely for that reason.

Finally, in view of its anthropomorphism we have hesitated a long time before adopting the father/son metaphor; by way of compensation we have consistently referred to a father or a son as "it".

Acknowledgments

We owe our usual thanks to the members of the Tuesday Afternoon Club. For detailed comments on an earlier version we owe our special thanks to W.H.J.Feijen, H.Meijer, C.S.Scholten, and W.M.Turski.

References

For insertion sort:

Wirth, Niklaus, "Algorithms + Data Structures = Programs", Prentice-Hall, Inc., Englewood Cliffs, N.J., U.S.A. (1976)

For heapsort:

Williams, J.W.J., "Algorithm 232 HEAPSORT", C.A.C.M. 7, 6 (June 1964), pp. 347-348

Floyd, Robert W., "Algorithm 242 TREESORT 3", C.A.C.M. 7, 12 (Dec. 1964), p. 701

For smoothsort:

Dijkstra, Edsger W., "Smoothsort, an alternative for sorting in situ", Science of Computer Programming 1 (1982) 223-233.