# An introductory essay on three algorithms for sorting in situ

This note is primarily an experiment in explanation. Without giving any detailed code, we have tried to describe the main ideas underlying a few related algorithms. Our guiding principle has been to confine ourselves to what the Careful Reader can be expected to remember for the rest of his life. The incentive for choosing this guiding principle came from EWD's observation — sampled from international audiences adding up to a thousand people — that, 17 years after its publication, heapsort is known by only 4% of the computing community. Heapsort being a very beautiful algorithm, we thought this figure disappointingly low and interpreted it as an indication that something is still lacking in the traditional style of presenting algorithms. In the following we try to contribute (to the best of our ability) to the improvement of this situation.

In more than one respect we are anxious to receive feedback from you. As usual all technical or linguistic comments will be most welcome, but besides that we need your reaction to assist us in deciding whether (in some future version) this essay should be submitted for publication. (For instance, if it has enlightened you or has given you some novel appreciation, we would like to know!)

* * *

Terminology. We define a chain to be a sequence of elements (having integer values). Each element with a successor in the chain is the father of that successor; each element with a

predecessor in the chain is 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.) <u>Offspring</u> is defined recursively: the offspring of a leaf is empty, the offspring of a father is its son together with that son's offspring. A chain is <u>descending</u> means that each father has a value that is at least the value of its son. For an element e of the chain, the predicate <u>dominant</u> means that no element of e's offspring has a value exceeding e's value. (End of Terminology.)

We leave to the reader to convince himself that "chain c is descending" can now be expressed by

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

We are interested in making a chain c descending — i.e. establishing P0 — without changing the bag of values owned by its elements. The latter requirement will be met by the usual technique: values owned will only be changed by swapping element values. We are particularly interested in this task when the chain c has been formed by prefixing an already descending chain with a further element.

Since after prefixing we can assert

(<u>A</u>e: e <u>in</u> c: dominant e <u>or</u> e = root of c)

we introduce the relation

$P_1$: $(\underline{A}e:\ e\ \underline{in}\ c:\ \text{dominant } e\ \ \underline{or}\ \ e = w)$

which can be established by the initialization w := root of c . Relation $P_1$ is of interest since we are allowed to conclude $P_0$ from

$P_1$ and "w has no son with a value larger than its own value".

Note. "w has no son with a value larger than its own value" can occur in a variety of ways: w may have no son at all (either by being a leaf or by not occurring in c ) or w may be a father, but of a son whose value is small enough. (End of Note.)

The program establishing $P_0$ is

```
w := root of c   {P1: invariant}
; do "w has a son s with a value larger than its own value"
   → "swap the values of elements w and s"
     ; w := s
od  {P0}
```

Proof. Relation $P_1$ is invariant provided the swap establishes

$$(\underline{A}e:\ e\ \underline{in}\ c:\ \text{dominant } e\ \ \underline{or}\ \ e = s).$$

For all elements e such that $e \neq w$ and $e \neq s$, $P_1$ implies (dominant 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, (dominant e) still holds.

For $e = w$, we observe that prior to the swap s dominates

its father $w$ on account of the guard and dominates its offspring on account of ($P_1$ and $s \neq w$) ; hence, after the swap (dominant $e$) holds.

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

## Insertion Sort

The sequence $m(i: 0 \leq i < N)$ is in ascending order means that its elements form a descending chain in which

$$(\underline{A}i: 1 \leq i < N: \ m(i) \text{ is the father of } m(i-1)) .$$

We can apply the algorithm of the previous section by building up a descending chain for the first $n$ elements of the sequence and increasing $n$ until $n = N$. This leads to the following algorithm, known as Insertion Sort.

```
|[ n: int ; n:= 1    {the chain defined by
     (Ai: 1 ≤ i < n:  m(i) is the father of m(i-1))
                  is descending  :  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. when sequence 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"

An attractive generalization of a chain is a rooted tree. Analogously, a descending tree is one in which each element dominates its entire offspring. A descending tree enjoys the property that the maximum value occurring in it can be found at its root; 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 algorithms that are in general of order $N \cdot \log N$ instead of of order $N^2$.

The operation sift establishes for a tree t

$P_2$: (Ae: e in t: dominant e)

provided initially

$P_3$: (Ae: e in t: dominant e or e = root of t)

holds. Situation $P_3$ can, for instance, arise under the following circumstances:

circumstance a: the root of a descending tree has been decreased;

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. making the root of the one an additional son of the root of the other.

A tree satisfying $P_3$ can be made to satisfy $P_2$, i.e. made into a descending tree, by applying the algorithm of the previous section 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 that element's largest son in the tree.

Proof. For e an off-chain element of the tree, dominant e continues to hold since neither its own value nor the values owned by its offspring have been changed. For e on the chain, we separately consider its offspring along the chain and its off-chain offspring: e dominates its offspring along 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 $N \cdot \log N$.

## Heapsort and smoothsort

The main pattern of these algorithms is the maintenance of

$P_4$: $$(\underline{A}\, i,j : 0 \leq i < j \ \underline{\text{and}}\ q \leq j < N : m(i) \leq m(j))$$

which vacuously holds for $q = N$ and enjoys the useful property that $P_4$ and $(q = 1)$ allows us to conclude that sequence $m$ is ascending. Relation $P_4$ 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 unsorted prefix has its maximum element at its rightmost position, so that $q$ can be decreased by 1 without violating $P_4$. If nothing is known about the prefix one needs a scan of the entire unsorted prefix to locate its maximum value, which then can be placed at its rightmost position. The ensuing algorithm is linear in the number of swaps, but quadratic in the number of comparisons. Neither swaps nor comparisons would be needed were the prefix ascending; this, however, would be begging the question. We conclude that for the construction of a more efficient sorting algorithm we have to do something in between: without completely sorting it, we have to prepare — in one way or another — the unsorted prefix so as to facilitate locating its maximum value. It's 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 firstly for building up the descending tree covering the unsorted prefix of length $N$ and secondly for its maintenance when subsequently the length of the unsorted prefix shrinks to 1.

Note. The $N!$ different possible computations can be arranged in a binary tree with the first comparison as its root and, depending

on how that comparison came out , 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 the average distance of a leaf from the root . This average distance is minimal if the tree is as well-balanced as possible , i.e. with a minimal difference between best- and worst-case behaviour as measured by the number of comparisons . Heapsort approaches such behaviour , which is of order $N \cdot \log N$ (thanks to Stirling's formula for $N!$). Smoothsort is worst-case of order $N \cdot \log N$ , 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.)

* *
*

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 $m(2 \cdot i + 1)$ and $m(2 \cdot i + 2)$ as its sons .) In the final sorting phase , in which $q$ has to be decreased from $N$ to $1$ , the values of the first and last element of the unsorted prefix are swapped so that the unsorted prefix can shrink by one element under invariance of P4 ; applying sift to the root $m(0)$ restores the father-son inequalities in the unsorted prefix (circumstance a). The final sorting phase is preceded by the first phase in which the whole sequence is prepared to act as an unsorted prefix: to begin with , $p$ is chosen high enough so that no father with index $\geq p$ has a son with index $< N - m(i : p \leq i < N)$ 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 $m(p)$ (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 final sorting phase can start. Note that an initially increasing sequence 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.)

* *
*

In smoothsort the unsorted prefix is the postorder traversal of the tree its elements are arranged in. (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.) As a result its rightmost element dominates all the others and the unsorted prefix can be shortened by one element without violation of $P_4$. In contrast to heapsort's tree, which is pruned leaf by leaf, smoothsort's tree is pruned in the second phase at its root: it becomes a forest of as many trees as the removed root had sons. These trees are binary, except the leftmost one, which may contain fathers with three sons. By grafting -as many times as possible- the leftmost tree upon the tree to the right of it, the forest is rebuilt into a tree (sift being applied in circumstance c) of which the shortened prefix is again the postorder traversal. As a result, no fathers with more than three sons are introduced.

Alternatively we can -and shall do so in the sequel- view the

unsorted prefix as the concatenation of the postorder traversals of one or more binary trees such that, firstly, in each binary tree each father dominates its offspring and, secondly, the roots of the binary trees are ascending in the order in which they occur. 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. The concatenation is a so-called standard concatenation, i.e. it consists of the postorder traversal of the largest possible Leonardo tree followed by the standard concatenation for the remainder of the unsorted prefix. Thus we achieve that the unsorted prefix is covered by the minimum number of Leonardo trees. (Leonardo trees have been preferred to perfectly balanced binary trees because, on the average, 25 % more trees are needed for coverage by the latter.)

As in heapsort, smoothsort's second phase - in which the sorted sequence is built up from the right - is preceded by a first phase in which the unsorted prefix of length $N$ (i.e. covering the whole sequence) is prepared. In contrast to heapsort this preparation starts from the left; $q$ - in the first phase the length 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 at each increase of $q$ the binary trees covering the prefix are such that each (binary) father dominates its (binary) offspring. (There are two cases. If increasing $q$ by 1 boils down to extending the standard concatenation by a one-node Leonardo tree, this obligation is empty; otherwise $LT_{i+1}$, $LT_i$, and the new element are combined into $LT_{i+2}$, to whose root sift is applied (circumstance b).) The first phase's second obligation

is to insert a grafting operation (circumstance c) each time a Leonardo tree is formed that will not subsequently be absorbed in a larger Leonardo tree; as a result, at the end of the first phase the roots are in ascending order.

In order to arrive at an algorithm of order N in the best case, a stack records which Leonardo trees cover the unsorted prefix. (Because each Leonardo tree occurs at most once in a standard concatenation, a bit stack in fact suffices.) For the same reason the number of sons of a single father had to be bounded.

18 January 1982

drs. A.J.M. van Gasteren
BP Venture Research Fellow
Dept. of Mathematics and
Computing Science
University of Technology
5600 MB EINDHOVEN
The Netherlands

prof. dr. Edsger W. Dijkstra
Burroughs Research Fellow
Plataanstraat 5
5671 AL NUENEN
The Netherlands