Science of Computer Programming 1 (1982) 223–233
North-Holland Publishing Company

SMOOTHSORT, AN ALTERNATIVE FOR SORTING IN SITU

Edsger W. DIJKSTRA

Burroughs Corporation, Nuenen, The Netherlands

Communicated by M. Rem
Received October 1981

Abstract. Like heapsort – which inspired it – smoothsort is an algorithm for sorting in situ. It is of order $N \cdot \log N$ in the worst case, but of order N in the best case, with a smooth transition between the two. (Hence its name.)

1. Introduction

Heapsort [3, 4] is an efficient algorithm for sorting $m(i: 0 \leq i < N)$ in situ; some, however, consider it a disadvantage of heapsort that it absolutely fails to exploit the circumstances in which the sequence is initially nearly sorted. While sharing in general with heapsort its $N \cdot \log N$ characteristic, smoothsort does not share this disadvantage: for an initially (nearly) sorted sequence, smoothsort is of order N with a smooth transition between the two. Smoothsort can be viewed as a pure exchange sort that is of order $N \cdot \log N$ in the worst case. For brevity's sake we shall describe sorting the integer sequence $m(i: 0 \leq i < N)$ in ascending order.

2. General outline of smoothsort

After a preparation in its first phase, smoothsort builds up the sorted sequence from right to left, i.e. it maintains between q and m

$$P0: \quad (\forall i, j: 0 \leq i < j \wedge q \leq j < N: m(i) \leq m(j)) \wedge 1 \leq q \leq N,$$

which is vacuously true for $q = N$ and enjoys the useful property that $P0 \wedge q = 1$ implies that the sequence m is in ascending order. (Since smoothsort modifies m only by swapping two of its elements, m obviously remains a permutation of the same bag of values.)

The second relation, built up during smoothsort's first phase and maintained during its second phase, is

$P1$: the unsorted prefix $m(i: 0 \leq i < q)$ is the postorder traversal of a tree in which no son exceeds its father.

0167-6423/82/0000–0000/$02.75

Relation *P1* ensures that the rightmost element of the unsorted prefix is its maximum element and that, therefore, q can be decreased by 1 without violating *P0*. In order to maintain *P1*, however, the decrease $q := q - 1$ must, in general, be accompanied by a rebuilding of the tree. This clerical obligation has no analogue in heapsort, in which a similar tree is pruned by removing a leaf; in smoothsort the tree is pruned at its root and without precautions it would, in general, fall apart into a forest of subtrees. Smoothsort restores the tree by grafting each subtree of the forest on the root of the subtree to the right of it.

Note that relation *P1* has been inspired by the desire to leave the sequence m untouched when initially already in ascending order.

Once the shape of the tree for $q = N$ has been chosen, the grafting procedure sketched above determines the shape of the tree for all smaller values of q. Our desire to construct an algorithm that would be of order N when m is initially (nearly) sorted forced us to derive the shape of the next tree from that of the preceding one. This recurrent computation, which heavily depends on the way in which shapes of trees are represented, is responsible for much of smoothsort's apparent complexity.

3. The presentation of smoothsort

In our presentation we shall follow the principle of postponing definitions until they are needed and—as a special case—not introducing variables until they are needed. The latter leads to so-called "program projections". A program is projected on a subset of its variables by omitting the declarations of its other variables and all statements not assigning to any of the variables of the subset projected on; the remaining expressions may only depend on the variables of the subset. Each time we shall give the minimal extension of the subset projected on. In the new statements thereby introduced, the variables introduced earlier are constants.

This way of program presentation has the advantage of introducing one complication at a time. It has the disadvantage of hiding the heuristics that led to the algorithm to be presented; the general outline and later remarks have been included to overcome this disadvantage as much as possible. (I think that we shall have to learn to live with the fact that presenting the final design in the most disentangled way and giving the heuristics—perhaps even in the form of a possible design history—are not necessarily compatible goals.) Finally I beg the impatient reader to remember that a program projection—though a legal program—does not make sense in isolation: its sole purpose is to be extended to something meaningful.

When invariants are given, they precede the repetition of which they are the invariant.

3.1. The introduction of q

Projected on the variable q, smoothsort is reduced to

```
[[q: int; q := 1 {invariant: 1 ≤ q ≤ N}
 ; do q ≠ N → q := q + 1 od {invariant: 1 ≤ q ≤ N}
 ; do q ≠ 1 → q := q − 1 od
]]
```

Variable q denotes the length of the unsorted prefix; the above projection shows that smoothsort as presented here is only defined for $N \geq 1$.

3.2. The introduction of r

Projected on the variables (q, r), smoothsort is reduced to

```
[[q, r: int; q := 1; r := 0 {invariant: q − r = constant}
 ; do q ≠ N → q := q + 1; r := r − 1 od {invariant: q = r = constant}
 ; do q ≠ 1 → q := q − 1; r := r − 1 od
]]
```

Remark 0. Variable r comes in handy in two ways. Firstly because $m(r)$ is the rightmost element of the unsorted prefix, and secondly because replacing its initialization $r := 0$ by $r := X$ will cause smoothsort to sort the sequence $m(i: X \leq i < X + N)$. Smoothsort accommodates such a shift of origin a little bit more easily than heapsort.

3.3. The introduction of p, b, and c

Invariant *P1* states that the unsorted prefix $m(i: 0 \leq i < q)$ is the postorder traversal of a tree, but does not define the tree. In this section we shall begin to define the tree for the unsorted prefix of length q and show how the shape of that tree is recorded using the triple (p, b, c).

To this purpose we regard the unsorted prefix $m(i: 0 \leq i < q)$ as a so-called standard concatenation of so-called stretches.

A "stretch" is a subsequence of consecutive elements $m(i: h \leq i < h1)$ for some $h < h1$ (which we shall later identify with the postorder traversal of a binary subtree of the tree mentioned in *P1*). As we shall see later, it is desirable that the number of stretches that concatenated together constitute the unordered prefix is relatively small. Stretches, however, don't come in all possible lengths and when q is not a stretch length we need more stretches to cover $m(i: 0 \leq i < q)$. The available stretch lengths are the so-called Leonardo numbers

$$\ldots 41\ \ 25\ \ 15\ \ 9\ \ 5\ \ 3\ \ 1\ \ 1\ \ (-1)$$

given by $LP_0 = LP_1 = 1$ and $LP_{n+2} = LP_{n+1} + LP_n + 1$. (The justification for this choice of available stretch lengths is better postponed.)

The "standard concatenation" of a sequence of length $q1$ consists of the longest stretch with length $\leq q1$, followed by the standard concatenation of the remainder (when not empty).

Remark 1. We leave it as an exercise for the reader to convince himself of the fact that the standard concatenation of a sequence of given length decomposes that sequence into the minimum number of stretches.

For the sake of the recurrent stretch length computations, we introduce for each stretch length b its "companion" c, i.e. we maintain

$$(\mathbf{E}\, n: n \geq 0: b = LP_n \wedge c = LP_{n-1});$$

here LP_{-1} is to be taken $= -1$. This is achieved by modifying variables b and c using only "*up*" and "*down*", defined by

$$up: b, c := b + c + 1, b \quad \text{and} \quad down: b, c := c, b - c - 1.$$

The stretches forming a standard concatenation are given by the triple (p, b, c); more precisely, with a binary representation of p

$$\ldots p_5 p_4 p_3 p_2 p_1 p_0,$$

the triple (p, b, c) defines the set of stretches LP_{n+i} for all i such that $p_i = 1$ and n defined by $LP_n = b \wedge LP_{n-1} = c$.

Note 0. As a first result, the length of the standard concatenation given by the triple (p, b, c) can—destructively—be computed by

```
length := 0
;do p > 0 →
    if even(p) → p := p/2; up
    □ odd(p) → length := length + b; p := (p − 1)/2; up
    fi
od .
```

Note 1. The representation is not unique: the operations "$p := 2*p$; *down*" leave the standard concatenation represented by the triple (p, b, c) unchanged.

The above coding of a standard concatenation is possible because, with the exception of stretch length 1, which may occur twice in a standard concatenation—e.g. of length 2 or 7—, each stretch length occurs at most once, whereas for stretch length 1 we have LP_1 and LP_0 at our disposal. We adopt the additional convention of recording a single stretch of length 1 as LP_1.

Note 2. We leave it as an exercise for the reader to prove that, as a consequence of the stretch lengths being Leonardo numbers, in the binary representation of p

only the two least significant 1's may be adjacent. This fact will be used in our next projection.

We now extend the subset of variables projected on by adding the triple (p, b, c) satisfying the invariant

P2: the length of the standard concatenation represented by the triple (p, b, c) equals q.

```
|[q, r, p, b, c: int; q := 1; r := 0; p, b, c := 1, 1, 1{invariant: P2}
; do q ≠ N
  → if p mod 8 = 3
      → p := (p − 1)/2; up; p := (p − 1)/2; up; p := p + 1 {b ≥ 3}
    □p mod 4 = 1
      → down; p := 2*p
    ; do b ≠ 1 → down; p := 2*p od; p := p + 1 {b = 1}
    fi; q := q + 1; r := r + 1
  od {invariant: P2}
; do q ≠ 1
  → q := q − 1; r := r − 1
  ; if b = 1
      → p := p − 1; do even(p) → p := p/2; up od {p mod 4 = 1}
    □b ≥ 3
      → p := p − 1; down; p := 2*p + 1; down; p := 2*p + 1 {p mod 8 = 3}
    fi
  od
]|
```

Note 3. For the (nonempty!) standard concatenations we have chosen in the above the "normalized" representation with $odd(p)$.

Note 4. The assertions at the end of each alternative have been given in order to stress that—as it should be!—the one repeatable statement is the inverse of the other: assertions in the one reappear as guards in the other [1].

Note 5. The reader may wish to prove that p's property as described in Note 2 is an invariant of both repetitions.

Note 6. The above projection is still of order N. The argument is as follows. In the first repetition the number of "*down*'s" is bounded by the number of "*up*'s", which is certainly less than $2N$. The second repetition is merely the inverse of the first one and the conclusion follows.

3.4. The introduction of m

At last the time has come to describe how stretches and the standard concatenation define which order relations between elements of m are maintained by smoothsort. We begin with the stretches, on which the predicates "trusty" and "dubious" will be defined. In accordance with the interpretation of a stretch as the postorder traversal of a binary tree we shall refer to the rightmost element of a stretch as the "root" of that stretch.

Denoting a sequence of length LP_n by $\langle seq_n \rangle$, we parse for $n \geq 2$

$$\langle seq_n \rangle = \langle seq_{n-1} \rangle \langle seq_{n-2} \rangle \langle root \rangle$$

where $\langle root \rangle$ stands for a singleton sequence. Stretch $\langle seq_n \rangle$ is dubious means that both $\langle seq_{n-1} \rangle$ and $\langle seq_{n-2} \rangle$ are trusty. Stretch $\langle seq_n \rangle$ is trusty means that, in addition, the roots of $\langle seq_{n-1} \rangle$ and $\langle seq_{n-2} \rangle$ are at most the root of $\langle seq_n \rangle$; a stretch of length 1 is by definition both dubious and trusty. As a consequence, the root of a trusty stretch is the maximum element of that stretch.

When stretches thus parsed are viewed as postorder traversals of binary trees, trustiness means that no son exceeds its father. A dubious stretch is made into a trusty one by applying the operation "*sift*"—a direct inheritance from heapsort—to its root, where *sift* is defined as follows: *sift* applied to an element without larger son is a skip, *sift* applied to an element $m(r1)$ that is exceeded by its largest son $m(r2)$ consists of a swap of these two values, followed by an application of *sift* to $m(r2)$.

Remark 2. We can now partly justify our choice of the Leonardo numbers as available stretch lengths, i.e. justify why we have not chosen (with the same recurrence relation)

$$\ldots 33\ 20\ 12\ 7\ 4\ 2\ 1\ (0).$$

The occurrence of length 2 would have required a *sift* able to deal with fathers having one or two sons, like the *sift* required in heapsort; thanks to the Leonardo numbers a father has always two sons and, consequently, smoothsort's *sift* is simpler.

During the second repetition smoothsort maintains

P3: the stretches of the standard concatenation of the unsorted prefix $m(i: 0 \leq i < q)$ are all trusty.

During the first one it maintains the weaker

P3': of the standard concatenation of the unsorted prefix $m(i: 0 \leq i < q)$ the rightmost stretch is dubious; its other stretches are all trusty.

Remark 3. The weaker *P3'* has been introduced for reasons of efficiency which cannot be explained now; see, however, Remark 4.

So much for the order relations captured by the stretches. In addition, smoothsort maintains during the second repetition

P4: the roots of the stretches of the standard concatenation of the unordered prefix $m(i: 0 \leq i < q)$ are ascending from left to right,

a relation, which is useful since $P3 \wedge P4$ implies that $m(r)$, the rightmost element of the prefix, is a maximum element of the prefix, and this is the circumstance under which $q := q - 1$ maintains *P0*. During the first repetition smoothsort maintains the weaker

P4': the roots of the trusty stretches of the standard concatenation of the unordered prefix $m(i: 0 \leq i < q)$ that are also stretches of the standard concatenation of length N are ascending from left to right.

We now have to investigate

(1) what to add to the first repetition for the maintenance of $P3' \wedge P4'$,

(2) what to insert between the two repetitions in order to transform $P3' \wedge P4'$ into $P3 \wedge P4$,

(3) what to add to the second repetition for the maintenance of $P3 \wedge P4$.

Investigation 1. In the case p **mod** $8 = 3$, the standard concatenation ends on a dubious stretch of length b which must be made trusty before it can be combined with the preceding stretch and the following element into a new dubious rightmost stretch. This can be achieved by applying *sift* to $m(r)$. Since no new trusty stretch is added to the standard concatenation, *P4'* is maintained without further measures.

In the case p **mod** $4 = 1$, the standard concatenation ends on a dubious stretch of length b, which in this step becomes the last but one stretch of the standard concatenation and, hence, must be made trusty. In the case $q + c < N$, it suffices to apply *sift* to $m(r)$ as before, since this stretch will later disappear from the standard concatenation. In the case $q + c \geq N$, however, just applying *sift* to $m(r)$ might violate *P4'* since this stretch of length b also occurs in the standard concatenation of length N. Making a dubious stretch trusty and including its root in the sequence of ascending roots is achieved by applying "*trinkle*" to $m(r)$. (As we shall see later, *trinkle* is like *sift*, be it for a partly ternary tree.)

Investigation 2. The reader may prove that it suffices to apply *trinkle* to $m(r)$.

Investigation 3. In the cas[illegible], the standard concatenation loses its last stretch, and $P3 \wedge P4$ is maintained without further measures.

In the case $b \geq 3$, the rightmost stretch of length b is replaced by two trusty ones; hence *P3* is maintained. To restore *P4* it would suffice to apply *trinkle* first to the root of the first new stretch and then to the root of the second new stretch, but this would fail to exploit the fact that the new stretches are already trusty to start with. This is exploited by applying "*semitrinkle*" in order to those roots.

Remark 4. From a logical point of view it would be perfectly permissible to replace a call on *trinkle* by a call on *sift*, which would make the dubious stretch trusty, followed by a call on *semitrinkle*, which would include its root in the sequence of ascending roots. After this substitution, each iteration of the first repetition starts with a *sift* and the whole first repetition is immediately followed by a *sift*. Since initially the last (and only) stretch is trusty, we can transform the program by removing all calls on *sift* and inserting a single call on *sift* at the end of the repeatable statement of the first repetition. This is essentially the program transformation that would be required if we wished to replace *P3'* by *P3*. (The collection of trusty stretches being extended, *P4'* would require reformulation.)

The version resulting from the above transformation is, however, rejected because a succession of *sift* and *semitrinkle* requires in general more comparisons and swaps than *trinkle*, as will become apparent later. This can be remedied by replacing the single call on *sift* by guarded calls on either *sift* or the combination in the form of *trinkle* (and removal of the calls on *semitrinkle* from the first repetition, which have now been catered for). *P3* would still be valid, *P4'* would have to be changed. This version, however, is rejected since it would lead to a duplication of the evaluation of the guards p **mod** $8 = 3$, etc.

In order to enable the reader to check the code in which the calls on *sift*, *trinkle*, and *semitrinkle* have been inserted, we give their calling conventions. (These conventions are not to be regarded as a recommendation, they have been chosen because in this publication I did not want to make any assumptions about a parameter mechanism.)

Routine *sift* is applied to the root $m(r1)$ of a stretch of length $b1$, of which $c1$ is the companion. Routine *trinkle* is applied to the root $m(r1)$ of the last stretch of the standard concatenation represented by the triple (p, b, c); this representation need not be normalized. Routine *semitrinkle* is applied to the root $m(r)$ of a stretch of length c which is preceded by the nonempty standard concatenation represented by the triple (p, b, c); again this representation is not necessarily normalized.

Note that "$p:=(p-1)/2$; $p:=(p-1)/2$; $p:=p+1$" has been simplified to "$p:=(p+1)/4$" and that "$r:=r-b+c$; *down*; $r:=r+c$" decreases r by 1.

```
smoothsort:
|[q, r, p, b, c, r1, b1, c1: int
 ;q:=1; r:=0; p, b, c:=1, 1, 1 {invariant: P3' ∧ P4'}
 ; do q ≠ N
     → r1:=r
      ; if p mod 8 = 3
          → b1, c1:=b, c; sift; p:=(p+1)/4; up; up
       □p mod 4 = 1
          → if q+c<N → b1, c1:=b, c; sift
            □ q+c≥N → trinkle
```

```
            fi; down; p:=2*p
          ; do b ≠ 1 → down; p:=2*p od; p:=p + 1
          fi; q:=q + 1; r:=r + 1
  od {P3' ∧ P4'}; r1:=r; trinkle {invariant: P3 ∧ P4}
; do q ≠ 1
     → q:=q - 1
      ; if b = 1
          → r:=r - 1; p:=p - 1; do even(p) → p:=p/2; up od
        □ b ≥ 3
          → p:=p - 1; r:=r - b + c
          ; if p = 0 → skip □ p > 0 → semitrinkle fi
          ; down; p:=2*p + 1; r:=r + c; semitrinkle
          ; down; p:=2*p + 1
        fi
  od
]|

up1: b1, c1:=b1 + c1 + 1, b1
down1: b1, c1, :=c1, b1 - c1 - 1

sift:
do b1 ≥ 3 →
   |[r2: int; r2:=r1 - b1 + c1
   ; if m(r2) ≥ m(r1 - 1) → skip
     □ m(r2) ≤ m(r1 - 1) → r2:=r1 - 1; down1
     fi
   ; if m(r1) ≥ m(r2) → b1:=1
     □ m(r1) < m(r2) → m: swap(r1, r2); r1:=r2; down1
     fi
  ]|
od

semitrinkle:
   r1:=r - c
  ; if m(r1) ≤ m(r) → skip
    □ m(r1) > m(r) → m: swap(r, r1); trinkle
    fi
```

Trinkle is very similar to *sift* when we regard each stretch root as the stepson of the root of the stretch to its right. Applied to a root without larger son, *trinkle* is a skip; otherwise the root is swapped with its largest son, etc. The trouble with the code is that all sorts of sons may be missing. In the following, *trinkle* is eventually reduced to a sift, viz. when the stepson relation is no longer of interest.

```
trinkle:
[[p1: int; p1, b1, c1 := p, b, c
; do p1 > 0 →
  [[r3: int; do even(p1) → p1 := p1/2; up1 od; r3 := r1 − b1
  ; if p1 = 1 cor m(r3) ≤ m(r1) → p1 := 0
   □p1 > 1 cand m(r3) > m(r1)
     → p1 := p1 − 1
     ; if b1 = 1 → m: swap(r1, r3); r1 := r3
      □b1 ≥ 3 →
        [[r2: int; r2 := r1 − b1 + c1
        ; if m(r2) ≥ m(r1 − 1) → skip
          □m(r2) ≤ m(r1 − 1)
            → r2 := r1 − 1; down1; p1 := 2*p1
          fi
        ; if m(r3) ≥ m(r2)
            → m: swap(r1, r3); r1 := r3
          □m(r3) ≤ m(r2)
            → m: swap(r1, r2); r1 := r2; down1; p1 := 0
          fi
        ]]
     fi
  ]]
od
]]: sift
```

And this concludes the code, in which I have abstained from implementation dependent optimizations.

4. In retrospect

While heapsort prunes the tree leaf by leaf, smoothsort prunes the tree at the root, and immediately one of heapsort's charms is lost: while the tree in heapsort remains beautifully balanced, the tree in smoothsort can get very skew indeed. So why bother about smoothsort at all? Well, I wanted to design a sorting algorithm of order N in the best case, of order $N \cdot \log N$ in the worst case, and with a smooth transition between the two (hence its name).

This is also the answer to the question why I introduced *P4*. By dropping *P4* one can dispense with *trinkle* and the code becomes much simpler. The price to be paid is a search for the maximum stretch root in order to establish that $m(r)$ is a maximum element of the unsorted prefix. Though such a simpler sorting algorithm is quite defensible, I rejected the option because it is never of order N.

One can also raise the question why I have not chosen as available stretch lengths: ... 63 31 15 7 3 1, which seems attractive since each stretch can then be viewed as the postorder traversal of a balanced binary tree. In addition, the recurrence relation would be simpler. But I know why I chose the Leonardo numbers: with balanced binary trees the average number of stretches is 1.2559 $\{=\frac{1}{4}(5+\sqrt{5})(^2\log(1+\sqrt{5})-1)\}$ times the average number of stretches with the Leonardo numbers. (I do not present this ratio as a compelling argument.)

It is possible that others have thought of this algorithm, but have rejected it for valid reasons, as yet unknown to me. I could not find it in the literature and it is not mentioned in [2], a recent article that compares five well-known sorting algorithms when fed with initially nearly sorted sequences. (That article compares Straight Insertion Sort, Shellsort, Straight Merge Sort, Quickersort, and Heapsort.) If it has not been discovered earlier, I would like to know the reason, because all its ingredients are well known since the discovery of heapsort in 1964.

Besides the possible interest in smoothsort I had another reason for developing it to the degree I did and for writing the above. (It took me three weeks, but I consider them well-spent.) The reason was that I knew beforehand that in trying to present smoothsort in a way as disentangled as possible I would encounter considerable difficulties. I hope they have been surmounted sufficiently well.

Acknowledgment

I am greatly indebted to C.S. Scholten and to all the members of the Tuesday Afternoon Club, with whom I had the privilege of discussing the algorithm, its coding, and its presentation. They have helped me clarifying my own thoughts and have suggested several significant simplifications. I am furthermore indebted to D.E. Knuth and W.M. Turski for their comments on the previous version of this text, and to the participants of the Marktoberdorf Summer School, 1981, on whom I could try out my presentation.

References

[1] F.L. Bauer and M. Broy, Editors, *Program Construction*, Lecture Notes in Computer Science 69 (Springer, Berlin, 1979) 54–57.

[2] C. R. Cook and D.J. Kim, Best sorting algorithm for nearly sorted lists, *Comm. ACM* 23 (11) (1980) 620–624.

[3] R.W. Floyd, Algorithm 242 TREESORT 3, *Comm. ACM* 7 (12) (1964) 701.

[4] J.W.J. Willliams, Algorithm 232 HEAPSORT, *Comm. ACM* 7 (6) (1964) 347–348.