Smoothsort, an alternative for sorting in situ.

Heapsort [0][1] 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 with heapsort its $N \cdot \log N$ characteristic, smoothsort does not share this disadvantage : when the sequence is sorted to start with, no rearrangement whatsoever takes place. For brevity's sake we shall describe sorting the integer sequence $m(i: X \leq i < X+N)$ in ascending order.

Projected on variables q and r, smoothsort is

```
|[ q,r : int ; q,r := 1,X
; do q ≠ N → q,r := q+1,r+1 od
  {invariant P0:
      (A i,j: X ≤ i < j ∧ X+q ≤ j < X+N : m(i) ≤ m(j))}
; do q ≠ 1 → q,r := q-1,r-1 od
]|
```

The rest of smoothsort is devoted to ensuring that, for the sake of the invariance of P0,

R1: $(\underline{A}\, i : X \leq i \leq r : m(i) \leq m(r))$

holds prior to q's decrementation. To that purpose, smoothsort maintains a further invariant concerning the sequence $m(i: X \leq i < r)$ of length q:

P1: the sequence $m(i: X \leq i < r)$ is 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 $h < h1$) of a special kind with, when "trusty", the property that none of its elements exceeds its last element, which is called its "root". The maximum of $m(i: X \leq i < r)$ is thus the maximum stretch root of the standard concatenation for $m(i: X \leq i < r)$, and relation P1 is helpful for establishing R1 when the number of stretches that together constitute the sequence $m(i: X \leq i < r)$ 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 $m(i: X \leq i < r)$. The available stretch lengths are the so-called Leonardo numbers

... 25 15 9 5 3 1 1 (-1)

given by $L_0 = L_1 = 1$ and $L_{n+2} = L_{n+1} + L_n + 1$. The "standard concatenation" of a sequence of

given length consists of the largest possible stretch, followed by the standard concatenation of the remainder.

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

$$(\underline{E}n:\ b=L_n \wedge c=L_{n-1})\ ;$$

this is achieved by only modifying variables b and c using "up" and "down", defined by

up: $b,c := b+c+1, b$ and down: $b,c := c, b-c-1$.

The stretches that form a standard concatenation are given by a triple p,b,c, where the 1's in the binary representation of p correspond to stretches that do occur in this standard concatenation, with the units position of p corresponding to the stretch of length b with companion c. More precisely, the length of the standard concatenation given by the triple p,b,c can be computed by

```
length := 0
; do p>0 ∧ even(p) → p := p/2 ; up
  [] p>0 ∧ odd(p) → length := length + b
                  ; p := (p-1)/2 ; up
od .
```

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 L_1 and L_0 at our disposal. We adopt the additional convention of recording a single stretch of length 1 in p as L_1. We furthermore observe that in p only the two least significant 1's may be adjacent.

We now add to our program

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

```
[[ q,r,p,b,c: int ; q,r := 1,X ; p,b,c := 1,1,1
; 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,r := q+1, r+1
 od
; do q≠1 → q,r := q-1, 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
     fi ; down ; p:= 2·p+1 ; down; p:= 2·p+1 {p mod 8=3}
od
]]
```

Note 1. For the (nonempty!) standard concatenations we have chosen in the above the "normalized" representation with odd(p). (End of Note 1.)

Note 2. 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: the assertions in the one reappear as guards in the other [2]. (End of Note 2.)

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, of which we shall encounter two types: "trusty stretches" and "dubious stretches".

Denoting a sequence of length L_n by $<seq_n>$, we parse for $n \geq 2$

$$<seq_n> = <seq_{n-1}><seq_{n-2}><r_n>$$

where $<r_n>$ stands for a singleton sequence. For $<seq_n>$ to be a dubious stretch it is required that both $<seq_{n-1}>$ and $<seq_{n-2}>$ are trusty stretches. For $<seq_n>$ to be a trusty stretch it is in addition required that no element of

$\langle seq_n \rangle$ exceeds its root r_n. A stretch of length 1 is by definition trusty.

In other words, we may view a stretch as the postorder traversal of a rooted binary tree and it is trusty when no son exceeds its father. It is dubious when no son, except possibly sons of the root, 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 sons 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 $m(r2)$.

During the second repetition smoothsort maintains

P_3: the stretches of the standard concatenation for $m(i: X \leq i \leq r)$ are all trusty.

During the first one it maintains the weaker

P_3': of the standard concatenation for $m(i: X \leq i \leq r)$ the last stretch is dubious; its other stretches are all trusty.

So much for the order relations captured by the

stretches. In addition, smoothsort maintains during the second repetition

P_4: the roots of the stretches of the standard concatenation are ascending ,

a relation which is useful since $(P_3 \wedge P_4) \Rightarrow R_1$. During the first repetition smoothsort maintains the weaker

P_4': the roots of the trusty stretches of the standard concatenation that are also stretches of the standard concatenation of length N are ascending.

We now have to investigate

1) what to add to the first repetition for the maintenance of $P_3' \wedge P_4'$
2) what to insert between the two repetitions in order to transform $P_3' \wedge P_4'$ into $P_3 \wedge P_4$
3) what to add to the second repetition for the maintenance of $P_3 \wedge P_4$.

Investigation 1. In the case $p \bmod 8 = 3$ 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 last 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 remains in the standard concatenation; in view of the extension, however, it must be made trusty. In the case q+c < N it suffices to apply sift to m(r) as before. In the case q+c ≥ N, however, just applying sift to m(r) would 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 will see later, trinkle is like sift, but for a partly ternary tree.) (End of Investigation 1.)

Investigation 2. It suffices to apply trinkle to m(r). (End of Investigation 2.)

Investigation 3. In the case b=1 the standard concatenation looses its last stretch, and P3 ∧ P4 remains invariant without additional measures. In the case b ≥ 3, the last stretch 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 "pretrinkle" in order to those roots. (End of Investigation 3.)

In order to enable the reader to check the code in which the calls on sift, trinkle, and pretrinkle have been inserted, we give the calling conventions.

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 pretrinkle is applied to 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" reduces 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,r := 1,X ; p,b,c := 1,1,1
; 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,r := q+1, r+1
  od ; r1 := r ; trinkle
; 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 → pretrinkle fi
        ; down ; p := 2·p+1 ; r := r+c ; pretrinkle
        ; 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

pretrinkle:
  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 stepson of the root of the next stretch. Applied to a root without larger sons, 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.

```
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
    fi
   ]|
 od
]| ; sift
```

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

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, in general of order $N \cdot \log N$, that would be of order N when the array is initially sorted.

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 $R1$. Though such a sorting algorithm is quite defensible, I rejected the option because it is never of the order N.

One can also raise the question why I have chosen the Leonardo numbers for my stretch lengths: ... 63 31 15 7 3 1 is also a possibility, which seems attractive since each stretch can now be viewed as the postorder traversal of a completely balanced tree. I know why I chose the Leonardo numbers: with the balanced binary trees the average number of stretches

is 1.2559 ($= \frac{1}{4}\cdot(5+\sqrt{5})\cdot(^2\log(1+\sqrt{5}) - 1)$) 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. It is not mentioned in [3], a recent article that compares five well-known sorting algorithms when fed with initially nearly sorted sequences. (It compares Straight Insertion Sort, Shellsort, Straight Merge Sort, Quickersort, and Heapsort.)

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 an intelligible manner I would encounter considerable difficulties. I hope they have been surmounted sufficiently well.

Acknowledgements.

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.

References.

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

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

[2] Bauer, F.L. and Broy, M. (Ed.), Program Construction, Lecture Notes in Computer Science 69, Berlin, Heidelberg, New York, Springer Verlag, 1979, pp. 54 - 57

[3] Cook, Curtis R. and Kim, Do Jin, Best Sorting Algorithm for Nearly Sorted Lists, C.A.C.M., 23,11 (Nov. 1980) pp. 620-624

Plataanstraat 5
5671 AL NUENEN
The Netherlands

12 July 1981
prof. dr. Edsger W. Dijkstra
Burroughs Research Fellow