Smoothsort's Behavior on Presorted Sequences

by

Stefan Hertel

Fachbereich 10
Universität des Saarlandes
6600 Saarbrücken
West Germany

A 82/11

July 1982

Abstract: In [5], Mehlhorn presented an algorithm for sorting nearly sorted sequences of length n in time $O(n(1+\log(F/n)))$ where F is the number of initial inversions. More recently, Dijkstra[3] presented a new algorithm for sorting in situ. Without giving much evidence of it, he claims that his algorithm works well on nearly sorted sequences. In this note we show that smoothsort compares unfavorably to Mehlhorn's algorithm. We present a sequence of length n with $O(n\log n)$ inversions which forces smoothsort to use time $\Omega(n\log n)$, contrasting to the time $O(n\log\log n)$ Mehlhorn's algorithm would need.

0. Introduction

Sorting is a task in the very heart of computer science, and efficient algorithms for it were developed early. Several of them achieve the O(nlogn) lower bound for sorting n elements by comparison that can be found in Knuth[4]. In many applications, however, the lists to be sorted do not consist of randomly distributed elements, they are already partially sorted. Most classic O(nlogn) algorithms - most notably mergesort and heapsort (see [4]) - do not take the presortedness of their inputs into account (cmp. [2]). Therefore, in recent years, the interest in sorting focused on algorithms that exploit the degree of sortedness of the respective input.

No generally accepted measure of sortedness of a list has evolved so far. Cook and Kim[2] use the minimum number of elements after the removal of which the remaining portion of the list is sorted - on this basis, they compared five well-known sorting algorithms experimentally. Mehlhorn[5], on the other hand, uses F, the number of inversions, i.e. the number of pairs of elements such that the bigger element precedes the smaller one in the sequence. He developed a new algorithm based on sorting by insertion and analysed its running time to be $O(n(1 + \log(F/n)))$.

Without indicating his measure of sortedness, Dijkstra[3] now presented a new algorithm for sorting in situ. He claims that it is well-behaved: "I wanted to design a sorting algorithm of order n in the best case, of order

nlogn in the worst case, and with a smooth transition between the two (hence its name)." However, not much evidence is given to support that claim.

We use the number-of-inversions-measure to show that smoothsort compares unfavorably to Mehlhorn's algorithm. In particular, a sequence of length n with O(nlogn) inversions is presented which forces smoothsort to use time Ω(nlogn) where Mehlhorn's algorithm would need only O(nloglogn) time.

The following section gives a high-level presentation of smoothsort. In section 2, then, some simple results concerning smoothsort's running time are proved. In section 3, we exhibit a small example where inversions caused by a single exchange of two elements are only removed one by one. The same one-by-one removal of inversions is the key observation that proves our main result in section 4.

1. The algorithm

Smoothsort is a two-pass algorithm for sorting an array A[1..n] in situ. In the original paper, a forest of recursively defined unbalanced binary trees is constructed - the sizes of the trees are given by the so-called Leonardo numbers which stand in a simple relation with Fibonacci numbers.

It is much easier, however, to describe and analyse smoothsort if one uses complete binary trees instead. The asymptotic behavior stays the same, and one can use binary

logarithms that are more comfortable to handle than those to the base $(\sqrt{5}+1)/2$.

This way, an outline of smoothsort is as follows:
In the first, the forward pass, a forest of complete heaps with the root being the last element of the array portion considered is constructed such that each tree has maximal size with respect to the remainder of the sequence. The roots of the heaps are sorted in ascending order. The second, the backward pass, consists of a repetition of the two operations: Remove the root of the respective last tree, and rebuild the structure.

Algorithm: Smoothsort

Input: Array A[1..n] of elements

Output: A, sorted into nondecreasing order

Method: We make use of Aho/Hopcroft/Ullman's procedure HEAPIFY [1] that makes a new heap out of two heaps of equal size and a new root. In addition, we use

```
proc MAKEHEAP(l,r: int):
   for i:=l to r do HEAPIFY(l,i)
corp
```

that makes A[l..r] into a heap with root r.

We define a root-sorted forest of heaps to be an ordered forest of heaps of decreasing size (the last two heaps may have equal size) such that the respective roots form a nondecreasing sequence.

RESTRUCT(r) basically transforms a root-sorted forest of

r-1 heaps and an additional heap with size(heap$_r$) ≤ size(heap$_{r-1}$) into a root-sorted forest of r heaps. It is described in more detail later.

With these procedures, the algorithm is as follows.

```
begin
  array R[1..⌈log(n+1)⌉] of int;      {indices of the heap
                                       roots in the current
                                       forest}
  m:= 0;          {A[1..m] currently under consideration}
  r:= 0;          {current number of heaps over A[1..m]}

  while m < n     {invariant 1: A[1..m] is a root-sorted
  do               forest of r heaps}
    k:= ⌊log(n-m+1)⌋;      {k-1 is height of next tree}
    MAKEHEAP(m+1, m+2^k-1);
    m:= m+2^k-1;
    r:= r+1;
    R[r]:= m;
    RESTRUCT(r)
  od;       {m = n}

  while m > 1     {invariant 2: inv.1 & A[m+1..n] contains
  do               the n-m biggest elements of A in
                   ascending order}
    size:= R[r] - R[r-1];      {size of last heap}
    if size = 1  →  r:= r-1
     □ size > 1  →  {split last heap}
                    R[r+1]:= R[r] - 1;
                    R[r]:= R[r+1] - (size-1)/2;
                    r:= r+1;
```

```
                    RESTRUCT(r-1);
                    RESTRUCT(r)
    fi;
    m:= m-1
  od
end
```

We still have to describe RESTRUCT.

RESTRUCT(r) takes ROOT, the root of the last heap,and swaps it with its left root neighbor as long as the left neighbor of ROOT is bigger than the top three elements of ROOT's current tree. Then ROOT is sifted down into its current tree to restore the heap property.

```
proc RESTRUCT(r: int):
  r':= r;
  while r' > 1  cand
        A[R[r'-1]] > max of A[R[r']] and its current
                     two children (if they exist)
  do
    swap(A[R[r'-1]], A[R[r']]);
    r':= r'-1
  od;
  if r' = 1  →  HEAPIFY(R[r'], 1)
   □ r' > 1  →  HEAPIFY(R[r'], R[r'-1]+1)
  fi
corp
```

2. Simple running time results

Lemma 1: The forward pass is of time complexity O(n).

Proof: We have to check the two procedures MAKEHEAP and RESTRUCT.

MAKEHEAP(l,r) is a parameterized version of BUILDHEAP in [1] and thus is known to make A[l..r] into a heap in linear time. Since the sum of the sizes of all heaps constructed is n, all calls to MAKEHEAP take total time O(n).

RESTRUCT is called once for every new heap constructed, i.e., as is easily seen, at most $\lceil \log(n+1) \rceil$ times.
Consider one call to RESTRUCT:
The loop can be executed at most O(logn) times (R[r] moves all the way to the leftmost tree), and HEAPIFY(i,j) is known from [1] to need time $O(\log|i-j|) \leq O(\log n)$.
Hence the total time for RESTRUCTuring in pass one is $O((\log n)^2)$.

Everything else in pass one takes constant time per loop iteration. Thus

time for pass one

$= O(n + (\log n)^2 + c\log n) = O(n)$. □

Lemma 2: Smoothsort is of order O(n) for sequences that are initially sorted.

Proof: The forward pass is handled by lemma 1.
Not any structure rebuilding is necessary in the backward pass, i.e., per iterative step, the loop in RESTRUCT is never executed, and HEAPIFY is not called recursively. □

Lemma 3: Smoothsort always runs in time O(nlogn).

Proof: By lemma 1 it suffices to consider the backward pass. From the proof of lemma 1 we know that one execution of RESTRUCT takes time O(logn). Since RESTRUCT is executed not more than n times (in more than half of the cases, the size of the last heap is 1), a total time of O(nlogn) results. □

3. Slow removal of inversions

It is clear that if given an input sequence with the (2i-1)st and the 2i-th elements interchanged, i = 1, ..., $\lfloor n/2 \rfloor$, i.e. a permutation with $\lfloor n/2 \rfloor$ inversions, smoothsort cannot do better than using one swap each to remove one inversion. Can we force smoothsort to act likewise in nontrivial cases? A first answer gives the following theorem.

Theorem 1: An otherwise perfectly sorted sequence with a single interchange of two elements of logarithmic distance may force smoothsort to use one swap per inversion.

Proof: Suppose w.l.o.g. that $n = 2^k-1$ for a positive k. Exchange s, the rightmost leaf of the left subtree in the perfectly sorted heap of size n with b, the leftmost leaf in the right subtree, as shown in figure 1.

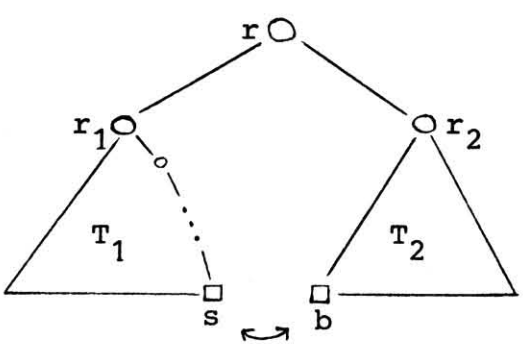


figure 1

Notice b is smallest in T_2, and precisely those elements on the path from s to r_1 are greater than s in T_1. Thus, T_1 having a height of k-2, s and b are distance k-1 = O(logn) apart, and their interchange is responsible for a total of 2k-3 inversions.

The given algorithm now works as follows:

In pass one, b moves up to r_1's place in k-2 swaps.
In pass two, upon uncovering s, schanges its place with b, and subsequently sinks down to its correct position in k-2 swaps. Thus 2k-3 swaps are performed. □

Of course, this behavior is not crucial in such a small example since the overall running time stays O(n). However, we shall see in the next section that much larger examples exhibiting that same behavior can be constructed.

4. The central example

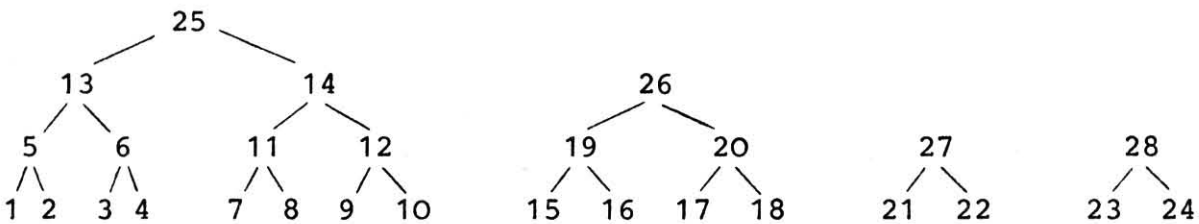
In this section, we show that smoothsort does not achieve running time O(n(1 + log(F/n))). More precisely, we show that there are input sequences with only O(nlogn) inversions which force smoothsort to use time Ω(nlogn).

Theorem 2: Sorting a permutation with O(nlogn) inversions may require Θ(nlogn) swaps.

Proof: Consider the following initial permutation of [1..n] in an array of length n:
Let p be the number of trees in the forest induced by n. Let the trees successively have the roots n-p+1, n-p+2, ..., n-1, n. The descending sequence n-p, n-p-1, ..., 1 is distributed as follows:

Fill the last tree by placing n-p at the root of its right subtree, n-p-1 at the root of its left subtree. Fill the right, then the left subtree recursively. Continue in the same manner with the remainig trees of increasing size.

Example: n = 28, p = 4



Let us count the number of inversions occurring in such a permutation. For simplicity, consider just the case of a single tree, i.e. $n = 2^k-1$ for a positive k. Obviously, this is no relevant restriction. If k = 1 or k = 2, there is no inversion at all. Otherwise, there is one inversion between the root of the left subtree and each non-root element in the right subtree, in addition to the inversions within the subtrees.

This gives rise to the following recurrence equation:

$$inv_1 = inv_2 = 0,$$
$$inv_k = 2 \cdot inv_{k-1} + 2^{k-1} - 2$$

Solving this equation yields

$$inv_k = (k-3) \cdot 2^{k-1} + 2 .$$

Since $k = \lceil \log(n+1) \rceil$, this means that

$$inv_k = O(n \log n).$$

How does smoothsort perform on such a permutation?

The first pass does not rearrange anything. For the second pass, we need a few definitions.

If we have removed m elements, we call the forest corresponding to n-m the current forest. The elements at roots of the heaps in the current forest are called visible, all other elements of the current forest covered.

As can be seen by induction on the number of elements removed, the following holds true for our permutation:

At any time during the second pass, all visible elements are greater than the covered ones, and the two elements uncovered next are following in size.

Thus, upon uncovering two elements, these two have to migrate down to the roots of the two largest trees. Consider any one swap between roots of neighboring trees: Removed are the inversions between the greater of the two roots and all elements of the right tree, created anew are inversions between the smaller of the two roots and all non-root elements of the right tree. There are no other kinds of swaps, sinking down of an element within one heap never occurs. Thus, the total number of inversions is reduced by exactly one per swap.

Hence the total number of swaps performed by smoothsort equals the total number of inversions in the initial permutation. □

5. Conclusion

We have seen that smoothsort's running time both for totally sorted sequences and for random sequences is optimal to within a constant factor. However, the transition in-between is not very smooth. Namely, we notice from theorem 2 that, for presorted sequences, it does not outperform Mehlhorn's algorithm [5] which has a running time of $O(n(1 + \log(F/n)))$ where F is the number of inversions. Still, it should be noted that this comparison is not necessarily fair since the latter algorithm does not sort in situ.

References:

[1] A.V.Aho/J.E.Hopcroft/J.D.Ullman, "Heapsort - an O(nlogn) comparison sort", in "The Design and Analysis of Computer Algorithms", Addison-Wesley 1974, pp. 87-92

[2] C.R.Cook/D.J.Kim, "Best sorting algorithm for nearly sorted lists", CACM 23, 11 (Nov. 1980), 620-624

[3] E.W.Dijkstra, "Smoothsort, an alternative for sorting in situ", Sc. of Comp. Prog. 1, 3 (May 1982), pp. 223-233

[4] D.E.Knuth, "The Art of Computer Programming", vol. 3, Sorting and Searching, Addison-Wesley 1973

[5] K.Mehlhorn, "Sorting presorted files", in Th. CSc. 4th GI Conf., LNCS 67, Springer-Verlag 1979, pp. 199-212