SieveSort: Yet Another Sorting Algorithm [1]

Rajat K. Pal

Department of Computer Science and Engineering
University of Calcutta
92, A. P. C. Road
Kolkata – 700 009, India

Abstract. *Sorting is a well-known computational problem. Sorting means arranging a set of records (or a list of keys) in some (increasing or decreasing) order. In this paper, we propose a new comparison sorting algorithm SieveSort, based on the technique of divide-and-conquer, that takes time $O(n^2)$ in the worst-case, where n is the number of records in the given list to be sorted. This sorting algorithm could be treated as a multi-way Quicksort, which can be used to identify a desired key without sorting the entire sequence.*

Keywords. Sorting, Comparison sort, Divide-and-conquer, Record, Satellite data, Algorithm, Complexity.

1. INTRODUCTION

There are several algorithms that solve the following ***sorting problem*** [1, 2, 3, 4, 5]:

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (or reordering) $\langle a_1', a_2', \dots, a_n' \rangle$ of the input sequence such that $a_1' \le a_2' \le \dots \le a_n'$.

The input sequence is usually an n-element array, although it may be represented in some other fashion, such as linked list. In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a *record*. Each record contains a *key*, which is the value to be sorted, and the remainder of the record consists of *satellite data*, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well.

As for example, let us assume that a given unsorted list is as follows.

6 4 1 9 5 8 3 7 2

The subsequent sorted sequence of the elements in this list is given below, in non-decreasing fashion.

1 2 3 4 5 6 7 8 9

Here the satellite information may be to identify whether two keys a_i and a_j in their respective positions i and j in the given list are in order, in computing the sequence in some sorted fashion.

Now there is a lot of sorting algorithms in the present day world. Out of which, *Insertion sort* takes $\Theta(n^2)$ time in the worst case. *Merge sort* has a better asymptotic running time, $\Theta(n \lg n)$, in the worst-case. *Heapsort* sorts n numbers in $O(n \lg n)$ time, whereas the worst case running time of *Quicksort* is $\Theta(n^2)$. The average case running time of Quicksort is $\Theta(n \lg n)$, though, it generally outperforms Heapsort in practice [3].

Insertion sort, Merge sort, Heapsort, and Quicksort are all comparison sorts: they determine the sorted order of an input array by comparing elements, and it has been proved that Heapsort and Merge sort are asymptotically optimal comparison sorts [1, 2, 3, 4, 5]. In other words, Merge sort and Quicksort follow the algorithmic technique of divide-and-conquer. In this technique, a problem is divided into several smaller but similar subproblems, and after having the solutions of those subproblems, we combine them in order to obtain the solution of the given problem. In this paper, we have developed one more comparison sorting algorithm based on the technique of divide-and-conquer.

Though $\Omega(n \lg n)$ is a lower bound on the worst-case running time of any comparison sort of n inputs, there are a few mechanical sorting algorithms, viz., Counting sort, Radix sort, Bucket sort, etc. that run in linear time, if the size of each of the numbers to be sorted is bounded by a constant [3].

The sorting algorithm developed in this paper is designated as *SieveSort*. *SieveSort* sorts the elements in the given list by sieving them in an appropriate fashion. The worst-case and best-case running times of the sorting algorithm are $O(n^2)$ and $O(n)$, respectively, where n is the number of records in the given list to be sorted.

The paper is organized as follows. In Section 2, we formulate the sorting algorithm developed in this paper. In this section, we also compute the computational complexity of the algorithm, and study the stability of the same. In Section 3, we conclude the paper with a few remarks.

2. THE SORTING ALGORITHM

Let $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$ be the given sequence (or list) of n *unsorted* elements (or keys). The elements are numbered 1 through n, from left to right of the list, to label their positions in Π. That is, Π_1 is the first element in Π, Π_2 is the second element in Π, and so on. The position i, for some element Π_i in Π, is important in comparing other elements in their respective positions with Π_i in Π.

In this section, we formulate the sorting algorithm *SieveSort*, developed in this paper. This algorithm sorts the elements in Π in non-decreasing order. In the formulation of the algorithm, we divide the elements in Π and enlist them into p subsequences, $1 \le p \le n$, where the value of each of the elements in a subsequence is smaller than that of the first element of the subsequence. Moreover, the values of the first elements in successive subsequences are arranged in non-decreasing fashion. This work is formally realized as stated below.

For a list of n elements in Π, we assume a header list H[] of size n, where H[1] contains the first element in Π, i.e., Π_1. We append Π_2 after Π_1 (in the form of a linked list), only if the value of Π_2 is less than the value of Π_1. Otherwise, we update the header list where H[2] contains Π_2. For the third element in Π, i.e., for Π_3, if the value of Π_3 is less than the value of Π_1, we append Π_3 in the list of Π_1, i.e., along H[1]. If the value of Π_3 is greater than (or equal to) the value of Π_1, we judge whether it could be linked along H[2]. If yes, then the value of Π_3 must not be greater than that of H[2]. Otherwise, Π_3 is assigned to H[3], and so on. These phenomena are exemplified with three distinct numbers, 1, 5, and 9, for their different possible permutations, as follows.

Clearly, the permutations are (i) 1 5 9, (ii) 1 9 5, (iii) 5 1 9, (iv) 5 9 1, (v) 9 1 5, and (vi) 9 5 1. Consider the cases one after another. For the first case, 1 is assigned to H[1]. Since 5 is greater than 1, 5 is assigned to H[2]. Again, since 9 is greater than 1 as well as greater than 5, it is assigned to H[3]. Hence we have the linked information as shown in Figure 1(a).

For the second permutation, 1 is assigned to H[1]. Since 9 is greater than 1, 9 is assigned to H[2]. Again, since 5 is greater than 1, we compare it with the content of H[2], i.e., 9. Since, 5 is less than 9, we append 5 in the list of H[2]. Hence we have the linked information as shown in Figure 1(b). In a similar way, the other cases are considered in order to distribute the elements into p subsequences led by p header nodes; the linked information are shown in Figures 1(c) through 1(f), for the remaining permutations.

In all these linked information for different permutations as shown in Figure 1, the interesting observation is as follows. None of the elements in the list of $H[i]$ is greater than any element in the list of $H[i+1]$, where $1 \le i < p$, if we have at least two of such subsequences. So, if H[1] contains q elements, $1 \le q \le n$, then all these q elements must occupy the first q positions in the resulting sorted sequence. Moreover, if $q = n$, then the number of subsequences is just one as H[1] contains the largest element, or Π_1 is the largest element in the given sequence. We may generalize the fact in the form of a lemma, as stated below.

Lemma 1. *If the subsequences (or sublists) led by H[1] through H[i−1] contain a total of r elements and the subsequence led by H[i] contains q elements, then in the resulting sorted sequence (or list) these q elements are to be available in positions r+1 through r+q.*

Proof. In our sorting algorithm, we distribute the elements in a given sequence of n elements among p subsequences led by p header nodes H[1] through $H[p]$. In this process of distribution, a new element k is appended in the list led by header node $H[i]$, only if k is greater than or equal to the content of $H[i-1]$ but less than the content of $H[i]$, where $1 < i \le p$. If $H[i]$ is the last header node when $i < p$, such that the content of $H[i]$ is the largest amongst the elements considered so far, and k is either greater than or equal to the content of $H[i]$, then the header list is updated assigning the content of $H[i+1]$ same as k. So, in two consecutive subsequences led by $H[i]$ and $H[i+1]$, the later subsequence may contain an element which is as small as the content of $H[i]$. In other words, the content of $H[i]$ may be as large as the least element belonging to the subsequence led by $H[i+1]$. ♣

Now we consider each of the computed subsequences (for a given sequence) one after another in reverse form, and follow the same algorithm developed above. The process of sieving is continued till we have the subsequences of single elements' only led by corresponding header nodes. In other words, ultimately, we have to have a header list of n header nodes, where for each such node the link field is *nil*. Our claim is that the elements in successive header nodes H[1] through $H[n]$ are arranged in non-decreasing fashion.

To exemplify the phenomenon stated here, we may consider the last permutation in the above example, i.e., 9 5 1. The corresponding subsequence obtained is shown in Figure 1(f), which is same as the given sequence. Next, we consider this subsequence in a reverse manner, i.e., as 1 5 9. Needless to mention that this is same as the first permutation for which the subsequences obtained are shown in Figure 1(a).

Let us consider another sequence, say the last but one permutation of the above example, i.e., 9 1 5. Here also the subsequence obtained is same as the given sequence, as shown in Figure 1(e). In the next step of our algorithm, we consider the subsequence in the reverse manner, i.e., as 5 1 9, which is same as the third permutation. For this sequence, the distribution of elements is shown in Figure 1(c). Here we have two subsequences; in the first subsequence we have 5 and 1, and in the second subsequence we have 9 only. So, again for the first subsequence, we consider it in the reverse manner, i.e., as 1 5, and follow the same algorithm for it. In this step, we assign 1 to $H[1]$ and 5 to $H[2]$, as 5 is greater than 1, and for the second subsequence, we assign 9 to $H[1]$.

Note that for different subsequences, header lists are also different, and they are naturally arranged as they evolved in the process of forming the subsequences. Hence, $H[k]$ for subsequence i and $H[k]$ for subsequence j, for any fixed k and $i \neq j$, are different. Therefore, if $i < j$, $H[k]$ for subsequence i must be less than and appear earlier than $H[k]$ for subsequence j, in a level of hierarchy. So, in the last example, element 9 is actually assigned to $H[3]$ of the assumed header list. In this way, ultimately we obtain the sorted subsequences of elements, and the desired sorted sequence of the given (unsorted) sequence is computed by placing them one after another as they are generated in the process of dividing the sequence. See Figure 2, for making the algorithm explicitly understandable, for the sequence of 9 elements given in the introductory section of this paper.

So in different dividing and distributing steps, we sieve the elements based on their key values in the form of subsequences, and in the way of making these subsequences, we reduce their sizes into unit element only. In addition, in making the subsequences, we maintain the relative ordering of the subsequences, and eventually a sorted sequence is computed.

The algorithm developed in this paper is in general true if the presence of an element is either once or at most twice in the given sequence. Otherwise, to make the algorithm stable, we do the following modification on the algorithm developed here. (A sorting algorithm is *stable* if elements with equal keys are left in the same order as they occur in the input sequence [1, 2, 3, 4, 5].) For a new element k that is to be included in a subsequence, if k is same as the content of $H[i]$, where $1 \leq i < p$, such that the content of $H[i+1]$ is greater than k, we update $H[i+1]$ etc. as $H[i+2]$ onwards, and assign the content of $H[i+1]$ same as k. This is all about algorithm *SieveSort*. Hence we conclude the following.

Lemma 2. *Algorithm SieveSort is a stable sorting algorithm.*

From Lemma 1 we have an important observation as follows. Following algorithm *SieveSort*, the elements in a given sequence are divided in such a way so that each of the elements in the subsequence led by $H[i]$ must come later (earlier) than each of the elements in the subsequence led by $H[i-1]$ ($H[i+1]$), for $1 < i \leq p$ ($1 \leq i < p$). Hence, in order to compute the kth smallest or kth largest element in the given sequence, it is not required to compute the sorted sequence anyways. Rather, to identify the desired key, the proper subsequence could easily be differentiated, and in a similar way, recursively the desired key is obtained without computing the sorted sequence as a whole. Quicksort does the same thing, and in this algorithm a (sub)sequence is always divided into at most two, whereas algorithm *SieveSort* may divide a given (sub)sequence into at most n subsequences. We summarize this result as a theorem given below.

Theorem 1. *Algorithm SieveSort is as good as Quicksort in the worst-case.*

Now we study the computational complexity of algorithm *SieveSort*, developed in this paper. The worst-case complexity of the algorithm is stated in the following theorem.

Theorem 2. *Algorithm SieveSort runs in time $O(n^2)$ in the worst-case, where n is the number of records in the given list to be sorted.*

Proof. During execution of algorithm *SieveSort*, a maximum of $O(n)$ elements in a sequence are to be divided and distributed over $O(n)$ subsequences in the worst-case, and a sequence could be recursively divided into subsequences $O(n)$ times. ♣

Note that the best-case complexity of the sorting algorithm is $O(n)$, where n is the number of elements in the given sequence, which may occur when the given sequence is either in non-decreasing or in non-increasing order.

3. CONCLUSION

In this paper, we have developed a new comparison sorting algorithm *SieveSort*, based on the technique of divide-and-conquer, that sorts a given list in non-decreasing order and takes time $O(n^2)$ in the worst-case, where n is the number of records to be sorted in the given list. For an n-element sequence, the best-case running time of the algorithm is $O(n)$, when the elements are arranged in some fashion. The *SieveSort* is a stable sorting algorithm. If it is not required to make the algorithm stable, we could straightway implement the initial version of the algorithm. The novelty of the sorting algorithm developed in this paper is that *SieveSort* is as good as Quicksort in the worst-case. Even for a situation when Quicksort may work worst, *SieveSort* works better. In general, we could conclude that *SieveSort* is nothing but a multi-way Quicksort, which can be used to identify a desired key without sorting the entire sequence.

REFERENCES

[1] Aho A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley: An Imprint of Addison Wesley Longman, Inc., Reading Massachusetts, 1999.

[2] Brassard G. and P. Bratley, *Fundamentals of Algorithmics*, Prentice-Hall of India Pvt. Ltd., New Delhi, 1999.

[3] Cormen T. H., C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, Prentice-Hall of India Pvt. Ltd., New Delhi, 2001.

[4] Knuth D. E., *The Art of Computer Programming: Sorting and Searching*, Vol. 3, Second Edition, Addison-Wesley: An Imprint of Pearson Education Asia, New Delhi, 2000.

[5] Weiss M. A., *Data Structures and Algorithm Analysis in C*, Second Edition, Pearson Education Asia, New Delhi, 2002.

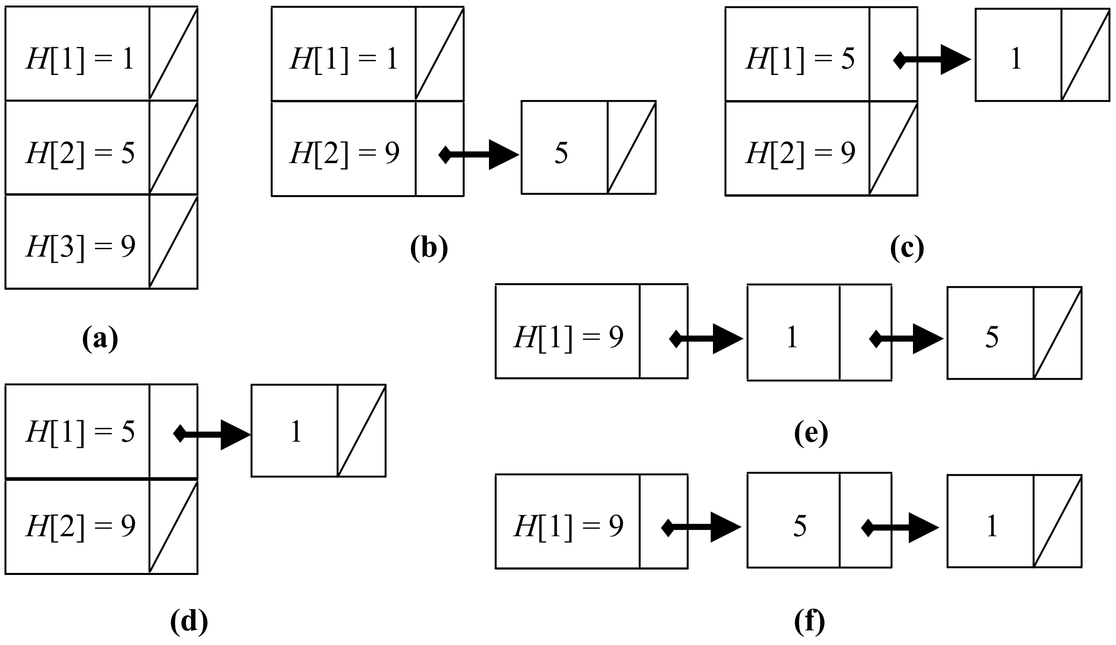

Figure 1: Assignment of elements in a given sequence into different subsequences.

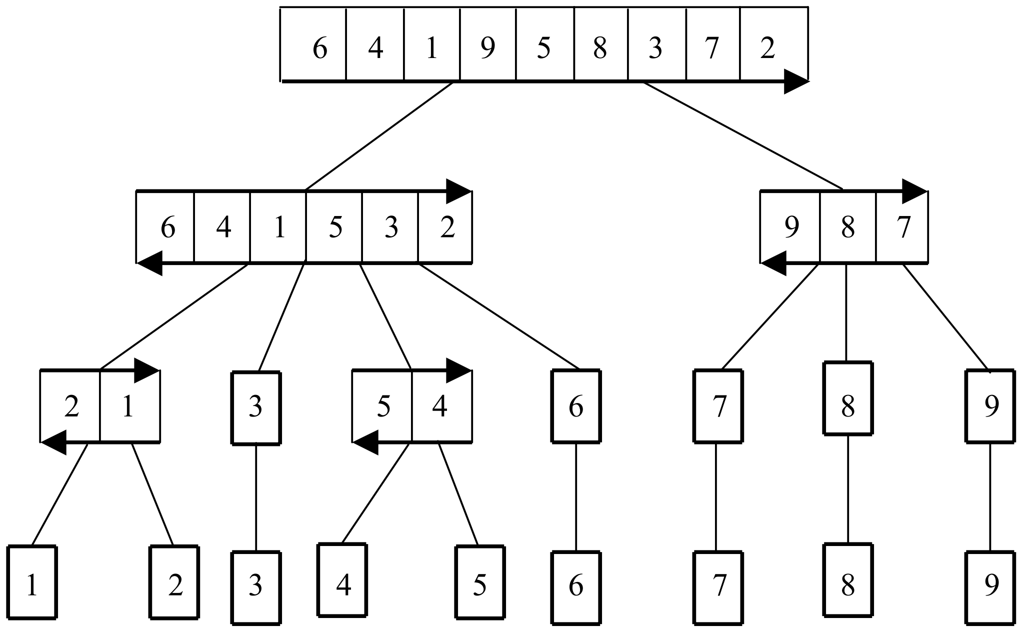

Figure 2: An example of the sorting algorithm *SieveSort*. Here each arrow on the top of a node indicates the scanned subsequence obtained from its parent sequence, and each arrow at the bottom of a node indicates the direction of scanning of the elements in the corresponding (sub)sequence.