Sorting is a well-known process in computer science, in which elements in a list are organized in a specific manner. Sorting algorithms are used to organize data and facilitate the application of downstream procedures that may be more complex, such as search algorithms. While research in this field has been ongoing for several decades, newer and more efficient sorting algorithms are being developed to this day. In this paper, Shreya Moudgalya presents “linear convection” sorting and walks through the logic behind the algorithm. Shreya demonstrates that his sorting process works efficiently on small data sets and is comparable to traditional algorithms such as the bubble sort.

Manuscript edited by Karren Yang and TianYi Ma

Linear Convection Sort
Shreya S Moudgalya
RV College, Bangalore, India

DOI: 10.13034/cysj-2013-006

Sorting is one of the most fundamental computational problems that can never be permanently resolved. It has become a stand in need of our daily life activities. Sorting helps to organize data in proper order. In turn it helps to improve the ability to find data in the data structures. Each algorithm has its own restrictions as well as its advantages. Efficiency of the code depends on various parameters such as the CPU (time) usage, memory usage, disk usage and network usage. But time complexity is considered to be the most important deciding parameter of a give piece of algorithm. The analysis of efficiency of algorithm may also depend considerably on the nature of the data. For example, if the original data set already is almost ordered, a sorting algorithm may behave rather differently than if the data set originally contains random data or is ordered in the reverse direction.

Similarly, for many sorting algorithms, it is difficult to analyse the average-case complexity. Generally speaking, the difficulty comes from the fact that one has to analyse the time complexity for all inputs of a given length and then compute the average. This paper proposes a new algorithm which is called as the linear convection sort that is more efficient for sorting a comparatively small set of data much faster than other sorting techniques.
Introduction

Time complexity is considered to be one of the most important deciding parameters for any given algorithm. The efficiency of a given piece of code and its behavior in handling data is dependent on the amount of data it handles. Moreover, if the original data set is almost already in order, a sorting algorithm may behave rather differently than if the data set is completely randomized. Thus, various sorting techniques must be developed for different types of data sets. Most algorithms have been coded to sort large volumes of data, leaving few procedures that can handle small amounts of data as efficiently. This paper proposes a new algorithm that is more effective for sorting a comparatively small set of data than most traditional sorting techniques.

Take into consideration the convection currents within a closed water heater; the water molecules on the surface cool faster and settle to the bottom of the container, whereas the warmer and lighter molecules rise to the top. This very idea has been exploited in writing this algorithm.

It is not possible to say that a particular algorithm is the best algorithm. In certain cases, an algorithm may be easy to implement but it may take maximum time to execute, whereas the other algorithm may be hard to implement but it may save execution time. Thus, when time complexity results are analysed, the best that has been achieved for n records, is a minimum $O(n \log n)$ time. We cannot sort faster than $O(n \log n)$. Any comparison algorithm (which doesn’t incorporate divide and sort technique) will have its worst case as $O(n^2)$.

Proposed Algorithm

In this paper, the proposed algorithm compares elements of $1^{st}$ & $(n-1)^{th}$ position. Then it compares the nth element with the $2^{nd}$ element of the array. The $1^{st}$ element is then compared with $(n-2)^{th}$ element and so on. After all comparisons have been made in the first pass, we have fixed the positions of the first and the last element. Then we move on to the second pass, and the iteration continues. Numbers are swapped whenever the required condition is met. Thus, after each pass, the loop is reduced by 2 units. Given below is the pseudo code for the proposed algorithm.

Pseudo Code

Declare integer variables a[], n = length of array a, i, k, b, m, p and z

Initialize i, temp and m to 0
Initialize k to n
Initialize p and z to n-1

do
  for(i=0; i<k; i++)
    { if a[m] is lesser than a[p]
      swap(a[m],a[p])
    if a[z] is greater than a[b]
      swap(a[z],a[b])
      increase value of b by 1
      decrease value of p by 1
    }
  decrease value of k by 2
  increase value of m by 1
  set value of p to (n-m-1)
  decrease value of z by 1
  set value of b to (m+1)
while (value of k is greater than or equal to 1)

Step-By-Step Example

In this example, we sort the array of numbers “5 1 4 2 8 3” in descending order. In each step, elements written in bold are being compared. There are 3 passes here.

1st Pass:
(5 1 4 2 8 3) → (8 1 4 2 5 3), Swap
(8 1 4 2 5 3) → (8 3 4 2 5 1), Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap

2nd Pass:
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap

3rd Pass:
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap
(8 3 4 2 5 1) → (8 3 4 2 5 1), No Swap

14
2nd Pass:
(8 [3 4 2 5] 1) → (8 [3 5 2 4] 1), Swap
(8 [3 5 2 4] 1) → (8 [5 3 2 4] 1), Swap
(8 [5 3 2 4] 1) → (8 [5 3 4 2] 1), Swap

3rd Pass:
(8 5 [3 4] 2 1) → (8 5 4 3 1), Swap

Figure 1. Array Exchanges. ([] shows the array elements used during sort)

There are 13 comparisons for 6 elements to be sorted. Notice that the element at the end gets fixed after 1st pass. Thus after each one full iteration, the terminal elements are fixed.

Equivalent Python Program Code for the Algorithm

```
a = []  # put Your values here
n = a.length()

I = 0
k = n
b = 1
m = 0
p = n-1
z = z-1

while True:

    for (I in range(k)):

        if a[m] < a[p]:
            swap(a[m],a[p])

        if a[z] > a[b]:
            swap(a[z],a[b])

    b = b + 1
    p = p -1
    k = k - 2
    m = m + 1
    p = n - m - 1
    z = z - 1
    b = m + 1

    if k>=1:
        break
```

Implementing the Space–Time Trade-off in the algorithm, a larger code size is traded for a higher program speed when loop unrolling is applied. Thus, it saves computational time required for jumping back to the beginning of the loop at the end of each iteration.

Complexity Analysis

It is difficult to analyse the average-case complexity for many sorting algorithms because we must analyze the time complexity for all inputs of a given length and then compute their average.

The average complexity of this sorting technique is $O(n^2)$. When sorting arrays that are nearly ordered, linear convection sorting behaves similarly to when sorting a completely random array. The best case performance of this algorithm does not make much of a difference in its complexity analysis, as the probability of receiving the data in which the elements are almost in order is very low. For 8 elements to be sorted, the probability reduces to 1 in 8!

Taking into consideration practical application, the asymptotic time complexity of this procedure is $O(n^2)$. Thus, this sorting technique is the best when it comes to sorting small data sets (from 10 to 25 elements). This convection sort outperforms most quadratic sorting algorithms, such as selection or bubble sorting, both in run time and in number of comparisons.

Figure 2. Plot of Number of Elements v/s Number of Comparisons.
The number of comparisons is given by the function: \( f(x) = \frac{n^2 - 2n}{2} \) where 'n' is the number of elements. Thus, it can be easily deduced that the average case performance is \( O(n^2) \). Here, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps.

Comparision with Other Sorting Techniques

<table>
<thead>
<tr>
<th>Sort Name</th>
<th>Number of cycles to sort 'n' elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>((3n^2 - 3))</td>
</tr>
<tr>
<td>Selection sort</td>
<td>((5n^2 + 5)(5n - 7))</td>
</tr>
<tr>
<td>Gnome sort</td>
<td>((5n^2 - 3)(5n))</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>((5n^2 + 5)(5n - 7))</td>
</tr>
<tr>
<td>Convection sort</td>
<td>((n-1)(2n)/4)</td>
</tr>
</tbody>
</table>

**Conclusion**

A novel, bidirectional algorithm has been proposed to sort elements. Linear convection sorting is very simple to understand, facilitating its implementation in comparison to other simple quadratic (i.e., \( O(n^2) \)) algorithms. Furthermore, it is much faster and more efficient than traditional algorithms when sorting small data sets.

**Keywords**

Time Complexity; Efficiency; Data Structures; Linear Convection Sort; Complexité Temps; Efficacité; Structures de Données; Linear Convection Trier.

**References**

Review of *Linear Convection Sort*

"Linear Convection Sort", by S. Moudgalya, shows that even after 110+ years of work on sorting algorithms, there are always new variations to be found.

The linear convection sort described in this manuscript can be thought of as a variation of the "Cocktail Shaker Sort" (http://en.wikipedia.org/wiki/Cocktail_sort) which does a traditional bubble sort - passing through the array, swapping neighbours that are out of order - alternating sweeps so that it is sorted from both ends of the array in alternate passes. This bi-directional approach avoids a problem with "turtles" - elements at the far end of the array that are very out of place - taking a long time to sort, as the far end of the array is sorted as often as the near end of the array. The sorting algorithm here inherits that advantage. The main difference is that this approach looks on the face of it a bit more like a selection sort - it is expressly picking out the first and last elements of the sorted array region at each point - but this distinction is less important than it might appear, as the Cocktail Shaker Sort behaves similarly, even though this isn't necessarily obvious. The other difference is that the two directions of the pass are interleaved within one phase, rather than there being a distinct pass in one direction followed by another distinct pass in the other direction, but this is a small point.

I'd be interested in seeing a more detailed complexity analysis; for instance, the derivation of the number of comparisons. It's worth noting that the step by step example skips a couple of steps - for instance, in first pass, 5 should be compared to 3 right away. As implemented in the code, this is actually done twice, although it would be possible to skip this and bring the comparison complexity down to \((n^2+n)/2\).

Confusingly, this same expression is used for number of "cycles" - clock cycles? - in the table. It's not clear what is meant here, but clearly some of the results quoted in the table are wrong; for instance, it simply isn't the case that selection sort requires \((5n^2+5)(5n-7) = 25n^3 - 7n^2 - 25n - 35\) comparisons (or any operations) to sort; the table itself makes it clear that this isn't right, there's no way of stepping through a selection sort that it would take 10 operations to sort 2 numbers, or that would take its complexity to \(n^3\), particularly when the worst case is correctly listed as \(n^2\). References and explanations for these results must be given. Also, dismissing best case complexity - for instance, the performance of insertion sort on already-sorted arrays - is done too readily; while it's true that a nearly-sorted list is very unlikely to be generated randomly, not all sorting is done on random lists. It is common for instance to update maintained data structures, and it is convenient to take advantage of this. Thus updates to a sorted list, if they need to be re-sorted, would best take advantage of the likely nearly-sorted structure given.

More detailed comparisons to the recursive/hierarchical sorts, which give \(n\ln n\) complexity, rather than \(n^2\) complexity, would also be needed.

*Jonathan Dursi, Ph.D., Advanced Research & Development Computing Consultant, SciNet HPC, Toronto, Ontario*