Linear Time Complexity Sort Algorithm

M.R. Ghaeini
Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran
E-mail: mr.ghaeini@aut.ac.ir

M.R. Mirzababaei
Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran
E-mail: mirzababaei@aut.ac.ir

Abstract
In the field of Computer Science and Mathematics, sorting algorithms put elements of a list in a certain order, ascending or descending. Sorting is perhaps the most widely studied problem in computer science and is frequently used as a benchmark of a system’s performance. In this paper we present an improved stable sort algorithm based on bucket sort algorithm that statistically and in average does sorting operation with linear time complexity $O(n)$. Our algorithm is 50% faster than other comparison sorting algorithms e.g. Quick sort and Merge sort.

Keywords: Linear Time Complexity, non-Comparison Sort Algorithm, Sigmoid Function, Probability

1. Introduction
Sorting a generic list of numbers is a well-studied problem that can be efficiently solved by using generic algorithms, such as Quick Sort [1], and Shell Sort [2]. However, a generic algorithm may not be the best choice when the list which going to be sorted has some initial order e.g., when many elements in the list are already sorted. An alternative in these cases is to use adaptive algorithms, which take the advantage of the partial order of the list to accelerate the sorting process [3]. A sorting algorithm is called adaptive if it sorts the sequences that are close to fully sorted faster than the random sequences. It sorts without knowing how far the list is from the sorted sequence [4].

An alternative to reduce the sorting time and complexity of sorting algorithm is to change the model that used to determine the key order.

Most of the classic sorting algorithms works under the comparison based model. They sort the list exclusively through pair wise comparison. However, there are also alternative sorting methods, which the content of their keys are used to obtain the position without any need to compare them to each other. They can obtain better results, because real machines allow many other operations besides the comparison [5].

Examples of non-comparison methods are Radix Sort [6], and Group Sort [7]. The number of comparisons that a comparison sort algorithm requires increases at least in proportion to $n \log n$, where $n$ is the number of elements of list. According to Knuth’s theoretical lower bound theorem for general sorting algorithms in comparison sorting algorithms minimum of the time complexity is $O(n!)$ equals to $O(n(n \log n))$ [6]:

$$\log(n!) = n \log(n) - n \log(e) + \theta(\log(n)) = n \log(n) - 1.442695n$$ (1)

Thus, the $O(n(n \log n))$ bound is asymptotically tight. We attended to the Importance of sorting issue and ascending humans requirements to sorting data, to prepare that for processing, its so valuable to design a kind of sorting algorithm that can sort data in linear time complexity in average and dominant case.

One of the non-comparison based sorting algorithms is the bucket sort. Input numbers to the bucket sort algorithm is considered to be between 0 and 1. This range divides to arbitrary number of buckets, and
supposed to have a suitable dispersion in our data. The algorithm works with linear time complexity, although, time complexity can increase to quadratic time complexity $O(n^2)$.

In this algorithm we use a statistical perspective, according to reality and the future usage of an algorithm to sort the data that we expect in real usage, can be said that these data are reasonably uniform (normal model and dispersion) and we have little data with very high dispersion and a very irrelevant relationship with each other. Also, when we increase the number of sort list element, uniformity improves to suitable case. Thus, we respect the efficiency of our sort algorithm in real and huge usages.

2. Related Works

Sorting is one of the most widely studied problem in computer science and is frequently used as a benchmark of a systems performance [8].

We can classify Sorting algorithm by some viewpoints, and their classification not similar; therefore, we mention some their specifics. We spot $(x, y, z)$, refer to best time complexity $(x)$, median time complexity $(y)$, and worst time complexity $(z)$.

- Quick Sort [1]: ($\log n$, $n\log n$, $n^2$), Comparison, often unstable.
- Efficient Quick Sort [9]: ($\log n$, $n\log n$, $n^2$), Comparison, often unstable.
- Partition Sort [10]: ($\log n$, $n\log n$, $n\log n^{\frac{3}{2}}$), Comparison.
- Merge Sort [11]: ($\log n$, $n$, $n\log n$), Comparison, stable.
- Average Sort [12]: ($\log n$, $n\log n$, $n\log n$), Comparison, unstable.
- Heap Sort [13]: ($\log n$, $n\log n$, $n\log n$, Comparison, unstable.
- Modified Heap Sort [8]: ($\log n$, $n\log n$, $n\log n$), Comparison, unstable.
- Insertion Sort [11]: (n, $n^2$, $n^2$), Comparison, stable.
- Selection Sort [11]: ($n^2$, $n^2$, $n^2$), Comparison, unstable.
- Optimized Selection Sort [14]: ($n^2$, $n^2$, $n^2$), Comparison, unstable.
- Bubble Sort [15]: (n, $n^2$, $n^2$), Comparison, stable.
- Freezing Sort [16]: (n, n, $n^2$), Comparison, stable.
- Shell Sort [2]: (n, $n\log n^2$, $n\log n^2$), Comparison, unstable.
- Bucket Sort [11]: (n, n, $n^2$), non-Comparison, stable.
- Radix Sort [6]: (d(n + k), d(n + k), d(n + k)), non-Comparison, stable.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case running time</th>
<th>Average-case running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n\log n)$</td>
<td>-</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$O(n^2)$</td>
<td>$O(n\log n)$</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>$O(k+n)$</td>
<td>$O(k+n)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(d(k+n))$</td>
<td>$O(d(k+n))$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
- Counting Sort [11]: (n + k, n + k, n + k), non-Comparison, stable.
- Skewed Distribution Sort [17]: (n, n, −), non-Comparison, stable.

In this area, somebody worked in parallel sort algorithm that we can mention some of these.
- Parallel Merge Sort [18],[11]: O( n/(log n)^2 )
- Parallel Bucket Sort [19]: O(log n)
- Parallel tree-Sort [19]: O(n)

Table 2, Processor Number & Run Time Required by Parallel Sort Algorithms [19]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Processors</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odd-even transposition</td>
<td>n</td>
<td>O(n)</td>
</tr>
<tr>
<td>Batchers bitonic</td>
<td>O(nlog’n)</td>
<td>O(nlog’n)</td>
</tr>
<tr>
<td>Stones bitonic</td>
<td>n/2</td>
<td>O(nlog’n)</td>
</tr>
<tr>
<td>Mesh-bitonic</td>
<td>n</td>
<td>O(√n)</td>
</tr>
<tr>
<td>Muller-Preparata</td>
<td>n^2</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>Hirschberg (1)</td>
<td>n</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>Hirschberg (2)</td>
<td>n^{1+1/k}</td>
<td>O(klog(n))</td>
</tr>
<tr>
<td>Preparata (1)</td>
<td>nlog(n)</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>Preparata (2)</td>
<td>n^{1+1/k}</td>
<td>O(klog(n))</td>
</tr>
<tr>
<td>Ajtai et al</td>
<td>nlog(n)</td>
<td>O(log(n))</td>
</tr>
</tbody>
</table>

3. Linear Time Complexity Sort Algorithm

In this algorithm we use a function to help us in data classification. The function name is Sigmoid function (Equation 2)

\[ y(x) = \frac{1}{1 + e^{-x}} \quad (2) \]

This function is Injective, Extender and derivable in all of real number range (R), and we can calculate y by x, and x by y (Equation 3). This function maps all of real numbers into range of 0 and 1, (0,1). But out of a symmetrical range, function output tend to 0 by being x less than minimum of that range and function tend to 1 by being x bigger than maximum of that range. We consider another form of sigmoid function for our problem and implementation.

\[ y(x) = \frac{x}{- \frac{(x-m)}{w} + 1} \quad \text{x(y) = } -w \ln\left(\frac{1-y}{y}\right) + m \quad (3) \]

In formulas (Equation 3), m refers to median point, and w refers to function weight. According to w we can increase or decrease function acceptable rage. This function is symmetrical, but our data may not be symmetrical e.g. all of them are positive, therefore, we lose half of mapping area, so that, at first we find data median and spot that for m value, therefore, our diagram moves on x axis and put into our data median data. We must mention that we can use average of data for m value too, but median value is more robust than outlier data. If we use median value, we can warranty that half of data are in half of
function acceptable area and another half of data are in another area.

\[ y'(x) = \frac{e^{-(x-m)}}{w(1+e^{-w(x-m)})^2} \]  

(4)

We suppose to define the function acceptable area. The function acceptable area is an area, in which derivation of function is bigger than \( \varepsilon \). \( \forall x \in (x_{\min}, x_{\max}); y'(x) \geq \varepsilon \)

And the area is in range of \((x_{\min}, x_{\max})\).

As all of our parameters affect \( x \), we can solve the problem for main sigmoid function and generalize that for our problem.

\[ y'(x) \geq \varepsilon \Rightarrow \frac{e^{-x}}{(1+e^{-w})^2} \geq \varepsilon \Rightarrow \frac{e^{-x}}{(1+e^{-w})^2} = \varepsilon \Rightarrow \varepsilon e^{w} + (2\varepsilon - 1)e^{-x} + \varepsilon = 0 \Rightarrow X = -X = e^{-x} \Rightarrow X = \frac{-X - \pm \sqrt{(2\varepsilon - 1)^2 - 4\varepsilon^2}}{2\varepsilon} \]  

(6)

\( \varepsilon X^2 + (2\varepsilon - 1)X + \varepsilon = 0 \Rightarrow X = \frac{-(2\varepsilon - 1) \mp \sqrt{(2\varepsilon - 1)^2 - 4\varepsilon^2}}{2\varepsilon} \Rightarrow X = \frac{-(2\varepsilon - 1) \mp \sqrt{1 - 4\varepsilon}}{2\varepsilon} \)
For example we assume $\varepsilon = 0.01 \rightarrow x_{\min} = -4.5849$, $x_{\max} = 4.5849$

If we want to generalize function, we must set function derivation bigger than $\varepsilon w$ but for reaching to our area, we ignore $w$. ($x'$ equals to $x_{\min}$ or $x_{\max}$).

$$x' = \beta \Rightarrow \frac{x - m}{w} = \beta \Rightarrow x = w\beta + m \quad w = \frac{2d}{|\beta|} \quad (7)$$

### 3.1 Algorithm

At first we find median value (based on Selection Algorithm [20]), next we find efficient data distance ($d$), and by this value we can calculate the acceptable area, this area is in range of $(-d, +d)$.

Next, by $d$, we calculate $w$ (Equation 7), so that we can calculate $y$ values for our data and sort $y$ values by Bucket Sort Algorithm.

If number of data in a bucket get more than an arbitrary constant ($\alpha$), a Boolean variable is set to be True that shows we must call our sort for bucket data, but if Boolean variable is False, we sort buckets data in $O(1)$.

### 3.2 Efficient data distance ($d$)

First we find $\max_1$, $\max_2$, $\min_1$, $\min_2$, and delete $\max_1$ and $\min_1$ form our data and calculate new the average, certainly median has the same value (median didn’t change).

Obviously our data are in range of $[\min_2$, $\max_2]$, now we want to find the best range that the median is in its center (Equation 8).

$$d = (m - \min_2) + (\max_2 + \min_2 - 2m)C(average, m, \min_2, \max_2, a, b) \quad (8)$$

$$C(x, m, \min_2, \max_2, a, b) = \begin{cases} 
0 & : x \leq m - a \\
\frac{x - m}{b} & : m - a < x < m - a + b \\
1 & : x \geq m + a + b 
\end{cases}$$

$$a = \begin{cases} 
0 & m < \frac{\min_2 + \max_2}{2} \\
b & m \geq \frac{\min_2 + \max_2}{2}
\end{cases}$$

$$b = \begin{cases} 
\frac{\max_2 - m}{m - \min_2} & m < \frac{\min_2 + \max_2}{2} \\
\frac{\min_2 + \max_2}{2} & m \geq \frac{\min_2 + \max_2}{2}
\end{cases}$$

### 3.3 $\alpha$ Determination

We can set this value arbitrarily, but if we set a big value, algorithm efficiency will be decreased, although time complexity wouldnt change.

Must not have any relation to $n$, because if we have any relationship between and $n$, algorithm time complexity will be changed.

While Boolean value is False, we must check that the number is new or not when adding it to bucket, if its new, increase counter and if counter is bigger than $\alpha$ set Boolean value True.

### 3.4 Pseudo Code

```
Sort(list, a, e):
    Median ← FindMedian(list)
    Range ← FindBaseRange(e)
    (Min1, Min2, Max2, Max1) ← FindMin&Max(list)
    d ← FindD(list, median, Min1, Min2, Max2, Max1)
    W ← 2d/Range
    For i ← 0 : list.size
        tempId ← [ExpRe.sult(list[i], median, w)*list.size]
```
Bucket[tempId].add(list[i], α)
For i ← 0 : list.size+1
   If (Bucket[i].isSortAgain=True) then
      Sort(Bucket[i].list, α, ε)
   Else
      makeSortArray()

Range ← FindBaseRange(ε):

Range ← \( \left\lbrack \frac{-\ln(2ε-1) - \sqrt{1 - 4ε}}{2ε} \right\rbrack \)

(Min1, Min2, Max2, Max1) ← FindMin&Max(list):
Min1 ← list[0]
Min2 ← list[0]
Max1 ← list[0]
Max2 ← list[0]
For i = 1 : list.size
   If (list[i]>Max2) then
      If (list[i]>Max1) then
         Max1 ← list[i]
      Else
         Max2 ← list[i]
   Else
      If (list[i]<Min2) then
         If (list[i]<Min1) then
            Min1 ← list[i]
         Else
            Min2 ← list[i]
   Else
      d ← FindD(list, median, Min1, Min2, Max2, Max1):
      Sum ← 0
      Number ← 0
      For i ← 0 : list.size
         If (Min1 < list[i] < Max1) then
            Sum ← Sum + list[i]
            Number ← Number + 1
      Average ← Sum/Number
      c ← C(Average, median, Min2, Max2, a, b)
      d ← (median-Min2)+(Min2+Max2-2median)*c
      c ← C(x, median, Min2, Max2, a, b) :
center ← (Min2+Max2)/2
If (median-center ≥ 0) then
    b ← median – Min2
    a ← b
Else
    b ← Max2 – median
    a ← 0
If (x – median + a ≤ 0) then
    c ← 0
Else
    If (x – median + a – b < 0) then
        c ← \[\frac{x - median}{b}\]
    Else
        c ← 1
result ← ExpResult(number, median, w):
result ← \[\frac{1}{1 + e^{\left(\frac{number-median}{w}\right)}}\]

Bucket:
isSortAgain ← False
add(element, a):
if (!isSortAgain && !list.hasElement(element))
    DiffNum ← DiffNum + 1
    If (DiffNum > α) then
        isSortAgain ← True
    list.add(element)

makeSortedArray:
    For i ← 0 : list.size+1
        tempList ← SortBucketList(Bucket[i].list)
        mainArray.add(tempList)

4. Implementation and Simulation

We implement our algorithm by JAVA. The system properties are:
Processor: Intel Core 2 Duo E4600, 2.40GHz / Memory: 2*1GB DDR2 800.
We tried to make our inputs similar to real world data and real usage. To decrease system errors, we tested each dataset 10 times, and accepted the minimum; we can use some optimization methods in our implementation, e.g.: Using Lookup Table (LUT), or we sort bucket data while Boolean variable is false.
We can also do some optimization in algorithm, e.g.: Using average of data (except min1 and max1) instead of median, therefore, d equals to \[(\max_2 - \min_2)/2\].
and acceptable area equals to \[(\max_2 - \min_2)\], we can even substitute sigmoid function with combination of three linear functions. All of these optimizations depend on usage and situation of problem.
As shown in Figure 3, and by a statistical view, our algorithm is more efficient than merge sort and quick sort, and its gradient is less than others.

Although, our algorithm have linear time complexity, but those algorithms have \(O(n \log n)\) time complexity.

5. Comparison

First we must calculate algorithm time complexity. One of the most important algorithm features is its time complexity; because it will have heavy cost if the data grow up and algorithm time complexity isn’t efficient.

\[
\begin{align*}
\text{Execution Time Comparison} \\
\text{Figure 3, Comparison result}
\end{align*}
\]

5.1 Time complexity determination

In this algorithm best case and worst case don’t need to be assumed for below calculations.

1- median calculation: we can find data median by using selection algorithm in \(O(n)\) \([20]\).

2- calculation y values and determination bucket of numbers and put numbers in their buckets. All of these acts are done in \(O(n)\), because we can calculate \(\exp(x)\) in \(O(1)\) and we can put numbers in their bucket in \(O(1)\) (we have \(\alpha\) comparison in worst case in every addition but \(\alpha\) doesn’t have any relationship with \(n\)), so, we have done all of them in \(O(n)\).

3- min, max, d and acceptable area calculation: we can find min and max in \(O(n)\), and calculate d and acceptable area in \(O(1)\) because we don’t need any traversal.

All of them are done in linear time complexity.

5.1.1 Best Case

Best case of our algorithm is similar to best case of bucket sort, which by normal or uniform distribution of numbers, Sorting is done in \(O(n)\). Number of bucket data is not depended to \(n\).

5.1.2 Worst Case

If the data distribution is so irrelevant, and our data is classified into two classes in each step and it is continued until the end of sort, we must call our function \(\log_2 n\) times. In this case we can prove that algorithm time complexity equals to \(O(n \log n)\) (by decision tree or master theorem \([11]\)).

\[
T(n) = 2T(n/2) + \theta(n) \xrightarrow{\text{Master Theorem}} T(n) = \theta(n \log n) \quad (9)
\]
5.1.3 Average Case

The input distribution of worst case is so rare, and statistically, we can ignore that data situation and accept linear time complexity for this algorithm in average case (you can see average time complexity proof in appendix).

In reality the data follows normal or uniform models and it is the inverse of worst-case data model. And random dataset can be a good dataset for sort algorithm comparison.

Another noticeable matter in this algorithm is independency of data positions and input sequences. Statistically, each input sequences can occur and some algorithms (e.g. Quick sort) are sensitive to input sequences. Different input sequences has different efficiencies, therefore, we have lower reliability in these algorithms. But our algorithm only depends on data not sequences of them; therefore, it is more reliable.

Our algorithm is stable because we traverse input and store them respectively, therefore, sequence of data remains.

One of the algorithm benefits is presenting a good data classification and this algorithm is more than a sorting algorithm and we can use this algorithm in other usages.

Best case of our algorithm is similar to best case of bucket sort, which by normal or uniform distribution of numbers, Sorting is done in $O(n)$. Number of bucket data is not depended to $n$.

6. Conclusion

We see that statistically, we can present a linear sorting algorithm without any limitations in input, and this is an important matter because by growing data size we need lower time complexity algorithms to be able to process data in a reasonable time.

Also in worst case situation, we have $O(n\log n)$ time complexity which is similar to average case of other comparison sort algorithms.

This algorithm supports other abilities of sorting algorithms, e.g. stability.

At last, we can say that this algorithm has many advantages and we can use it instead of other sorting algorithms.

References

Appendix: Average Time Complexity Proof

Let $n_i$ be the random variable denoting the number of elements placed in bucket $B[i]$, so in time complexity equals to

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} T(n_i) < \theta(n) + \sum_{i=0}^{n-1} O(n_i \log(n_i)) < \theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

(10)

We now analyze the average case running time of our sort, by computing the expected value of the running time, where we take the expectation over the input distribution. Taking expectations of both sides and using linearity of expectation, we have

$$E[T(n)] = E[\theta(n) + \sum_{i=0}^{n-1} O(n_i^2)] = \theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] = \theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])$$

We claim that $E[n_i^2] = 2 - 1/n$ for $i=0, 1, \ldots, n-1$.

It is no surprise that each bucket $i$ has the same value of $E[n_i^2]$, since each value in the input array $A$ is equally likely to fall in any bucket. To prove our claim, we define indicator random variables $X_{ij}[A[j]] \text{ falls in bucket } i$ for $i=0, 1, \ldots, n-1$ and $j=0, 1, \ldots, n$. Thus $n_i = \sum_{j=0}^{n-1} X_{ij}$.

To compute $E[n_i^2]$, we expand the square and regroup terms:

$$E[n_i^2] = E[(\sum_{j=0}^{n-1} X_{ij})^2] = E[\sum_{j=0}^{n-1} \sum_{k=0}^{n-1} X_{ij} X_{ik}] = E[\sum_{j=0}^{n-1} (X_{ij})^2 + \sum_{i_j \neq k} \sum_{i,k \neq j} X_{ij} X_{ik}] =$$

$$\sum_{j=0}^{n-1} E[(X_{ij})^2] + \sum_{i,j \neq k} \sum_{i,k \neq j} E[X_{ij} X_{ik}]$$

Where the last line follows by linearity of expectation. We evaluate the two summations separately. Indicator random variable $X_{ij}$ is 1 with probability $1/n$ and 0 otherwise, and therefore

$$E[X_{ij}^2] = 1^2 \cdot 1/n + 0^2 \cdot (1-1/n) = 1/n$$

When $k \neq j$, the variables $X_{ij}$ and $X_{ik}$ are independent, and hence

$$E[X_{ij} X_{ik}] = E[X_{ij}]E[X_{ik}] = (1/n)(1/n) = 1/n^2$$
so

\[ E[n_i^2] = \sum_{j=1}^{n} E[(X_{ij})^2] + \sum_{i<j} \sum_{k} E[X_{ij}X_{ik}] = \sum_{j=1}^{n} \frac{1}{n} + \frac{1}{n^2} - \frac{1}{n^2} = n \left( \frac{1}{n} \right) + n(n-1) \left( \frac{1}{n} \right) = n - \frac{1}{n} \]

This proves our claim.

At last, we conclude that the average case running time for bucket sort is (less than)

\[ \theta(n) + n \cdot O(2 - \frac{1}{n}) = \theta(n) \Rightarrow T(n) = \theta(n) \]

Even if the input is not drawn from a uniform distribution, our sort may still run in linear time. As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, \( \theta(n) + \sum_{d=0}^{n-1} O(E[n_i]) \) tells us that our sort will run in linear time. Also, since we call our algorithm for elements placed in buckets, if we avoid worst case input distribution, we reach to linear time complexity (elements placed in buckets will have good distribution (maybe after some steps)).