Optimization of Merge Based Sort Algorithms on Nearly Sorted Lists

ORHAN CAN ÖZALP
Galatasaray University
Graduate School of Science and Engineering
Çırağan Cad. No:36 Ortakoy, Istanbul
TURKEY
ozalp.orhan@gmail.com

MURAT AKIN
Galatasaray University
Graduate School of Science and Engineering
Çırağan Cad. No:36 Ortakoy, İstanbul
TURKEY
murakin@gsu.edu.tr

Abstract: The Merge Sort is a well-known comparison-based sort algorithm that has the same time complexity for all cases and is not adaptive. In this paper we made some improvements on merging method of Merge Sort and present a new adaptive and merge based sort algorithm for nearly sorted lists that has a time complexity $O(n+m)$, where $n$ is the size of the list and $m$ is the count of comparisons made in merge operations which is related to $\text{Rem}$ and $\text{Run}$ parameters.

Key–Words: Sorting Algorithm, Adaptive, Nearly Sorted, Merge

1 Introduction

Sorting is the computational process of rearrangement of items into ascending or descending order [1]. It is one of the fundamental techniques in computer science, because a lot of other techniques and algorithms are based on sorting.

A sorting algorithm is adaptive if it runs faster when the input list is nearly in order (or nearly sorted) [1]. Straight Insertion-Sort is a well-known example for adaptive sorting algorithms. In straight insertion sort, we start to choose values from the left hand side to the right and try to find suitable position for all values. This algorithm has a time complexity $O(n + d)$, where $d$ is the total number of exchange operations. For a sorted list no exchange operation will occur and the complexity will be $O(n)$.

In this paper, we are going to introduce a new approach for already known sorting algorithms merge-sort and Cook-Kim Sort [2] and combine them for better results for nearly sorted lists. Our new approach is based on finding already sorted sub-lists and merging them.

2 New Approach

2.1 Presortedness

There are many measures of presortedness (or disorder[3] [4]). As our algorithm takes advantage of already sorted sub-lists, we are going to focus the criteria those are related to sorted sub-lists.

2.1.1 Rem

For nearly sorted lists, it is possible to get a long (related to the length of the original list) sub-list with removing the elements which breaks the non-decreasing subsequence. Rem (abbreviation for Remove), is the number of removal operations in the original list to get the longest non-decreasing sub list[3].

2.1.2 Runs

Runs is the count of non-decreasing subsequences in the original list[4]. It also can be defined as a function;

$$\text{runs}(X) = |i|1 \leq i < n, a_{i+1} < a_{i}| + 1$$ (1)

 Runs value for a sorted list is 1 therefore for each Rem this value is incremented. Our algorithm uses these lists and with the increase of Runs value count of the merge operations will also increase. Optimizations in merge and sort algorithms are made for this issue.

2.2 Merge

Merging is the combination of two sorted lists into one sorted list [1]. Popular algorithms like Merge Sort uses merging technique.

Generic merge method is suitable for all sorted lists and has a time complexity $O(n)$. But since we are working with nearly sorted lists, a different assumption must be made.
Suppose

\[ S = k_1, k_2, k_3, \ldots, k_n \]  

(2)

is a nearly sorted list with \( n \) elements,

\[ \alpha = k_a, \ldots, k_b \]  

(3)

\[ \beta = k_c, \ldots, k_d \]  

(4)

are two sorted sub-lists of \( S \) which \( b < c \).

In a nearly sorted list, we assume that \( \alpha \) and \( \beta \) are more likely to be consecutive thus before using the classical merge method, we determine maximum and minimum value blocks and do not include them in classical merge method.

Assume that \( k_a < k_c \). We need to find an element in \( \alpha \) with index \( x \) that \( k_x \) is the smallest element that satisfies the condition \( k_x > k_c \). After that we can say that the subset of \( \alpha \),

\[ \alpha_{\text{min}} = k_a, \ldots, k_x \]  

(5)

definitely contains the minimum elements of two lists.

Assume that \( k_b < k_d \). To find the maximum elements, we need to find an element in \( \beta \) with index \( y \) that \( k_y \) is the smallest element that satisfies the condition \( k_y > k_b \). After that we can say that the subset of \( \beta \),

\[ \beta_{\text{max}} = k_y, \ldots, k_d \]  

(6)

definitely contains the maximum elements of two lists.

After finding the maximum and minimum sets, we find \( S_{\text{merge}} \) using these two subsets;

\[ \alpha_{\text{mid}} = k_{x+1}, \ldots, k_b \]  

(7)

\[ \beta_{\text{mid}} = k_c, \ldots, k_{y-1} \]  

(8)

and the result list \( R \) is

\[ R = \alpha_{\text{min}} + S_{\text{merge}} + \beta_{\text{max}} \]  

(9)

where ‘+’ operation means appending the second list to the first list.

### 2.3 Algorithm

Because of the presortedness of two sub arrays of the set \( S \) reduces the time complexity of merge operation, our algorithm is basically based on finding two non-decreasing sub arrays and merging them. While performing these operations, it uses \( 2n \)-sized buffer arrays; one for storing a non-decreasing sub-array and the other one for merging.

The algorithm starts iterating the elements from the first element and continues iterating while \( i \)th element is not greater than \( i+1 \)th element in other words, the algorithm always iterates in non-decreasing sequences.

When \( i+1 \)th element is smaller than \( i \)th element, elements starting from \( i+2 \) are controlled and appended to the first buffer if the element is not smaller than the last element of the first buffer. After the append operation, the non-decreasing sequence and the first buffer is merged into the second buffer. After the merge operation, references of buffers are swapped because the first buffer is always used for storing sorted elements and the second buffer is used for storing the result of the next merge operation.
Def. 1 Pseudocode

define buffer1, buffer2
define start := 0
define bufferlength := 0
define max := -∞

for index := 1 to negin{align*}
&\text{if } S_i \geq \text{max} \\
&\text{begin} \\
&\text{max} := S_i \\
&\text{increase } i \\
&\text{end} \\
&\text{else} \\
&\text{begin} \\
&\text{len} := i - \text{start} \\
&\text{if } \text{buffer1 has elements} \\
&\text{begin} \\
&\text{last} := \text{buffer1} \text{bufferlength} \\
&\text{while } S_i \geq \text{last} \\
&\text{begin} \\
&\text{inc } \text{bufferlength} \\
&\text{bufferlength} := i \\
&\text{inc } i \\
&\text{last} := \text{buffer1} \text{bufferlength} \\
&\text{end} \\
&\text{buffer2} := \text{merge(buffer1, } S_{\text{start, start+len}}) \\
&\text{end} \\
&\text{else} \\
&\text{buffer2} := \text{merge(buffer1, } S_{\text{start, start+len}}) \\
&\text{swap references of buffer1 and buffer2} \\
&\text{start} := i \\
&\text{bufferlength} := i \\
&\text{end} \\
&\text{end} \\
&\text{end}
\end{align*}

3 Conclusion

The performance of our algorithm are compared with other related algorithms with using low Rem arrays in Table-1 and Table-2. All results are in microseconds.

Algorithms are implemented in C and compiled with gcc under Ubuntu Linux 11.04.

<table>
<thead>
<tr>
<th>List Size</th>
<th>Merge Sort</th>
<th>Splitsort</th>
<th>Straight Insertion Sort</th>
<th>New Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.091</td>
<td>0.002</td>
<td>0.001</td>
<td>0.002</td>
</tr>
<tr>
<td>200</td>
<td>0.167</td>
<td>0.002</td>
<td>0.002</td>
<td>0.003</td>
</tr>
<tr>
<td>500</td>
<td>0.448</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>1000</td>
<td>0.902</td>
<td>0.007</td>
<td>0.008</td>
<td>0.009</td>
</tr>
<tr>
<td>2000</td>
<td>1.764</td>
<td>0.013</td>
<td>0.017</td>
<td>0.014</td>
</tr>
<tr>
<td>5000</td>
<td>4.434</td>
<td>0.031</td>
<td>0.043</td>
<td>0.037</td>
</tr>
</tbody>
</table>

Table 1: Results for best case

<table>
<thead>
<tr>
<th>List Size</th>
<th>Merge Sort</th>
<th>Splitsort</th>
<th>Straight Insertion Sort</th>
<th>New Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.083</td>
<td>0.005</td>
<td>0.002</td>
<td>0.004</td>
</tr>
<tr>
<td>200</td>
<td>0.168</td>
<td>0.016</td>
<td>0.005</td>
<td>0.010</td>
</tr>
<tr>
<td>500</td>
<td>0.428</td>
<td>0.085</td>
<td>0.023</td>
<td>0.038</td>
</tr>
<tr>
<td>1000</td>
<td>0.887</td>
<td>0.326</td>
<td>0.087</td>
<td>0.129</td>
</tr>
<tr>
<td>2000</td>
<td>1.788</td>
<td>1.252</td>
<td>0.334</td>
<td>0.438</td>
</tr>
<tr>
<td>5000</td>
<td>4.461</td>
<td>7.825</td>
<td>2.013</td>
<td>1.971</td>
</tr>
</tbody>
</table>

Table 2: Results for Rem=2%

Regarding Table-1 and Table-2, we’ve seen that our algorithm gives better results comparing to other split or merge based algorithms with low Rem values.

As we see in results and experiments, we saw the key part of our algorithm is finding a long sub-
sequence in linear time. It is not possible to use a Longest Non-Decreasing Subsequence (LNS) algorithm because LNS algorithms work in $O(n^2)$ time so if a better algorithm could be developed for finding optimal non-decreasing subsequences in linear time, performance of the algorithm will be better.

**Acknowledgements:** This work has completed in Galatasaray University Computer Science Department for master thesis. The author would like to thank Asst. Prof. Dr. Murat Akin for his supervision and helpful advices.

**References:**


