Index Sort

Hari Krishna Gurram
M.Tech (CS),
Ucek, JNTU Kakinada.

Gera Jaideep
M.Tech(CS),
Ucek, JNTU Kakinada.

Abstract

One of the fundamental issues in computer science is ordering a list of items. Although there is a number of sorting algorithms, sorting problem has attracted a great deal of research, because efficient sorting is important to optimize the use of other algorithms. This paper presents a new sorting algorithm (Index Sort) which runs based on the previously sorted elements. This algorithm was analyzed, implemented and tested and the results are promising for a random data.

Keywords: Index Sort, Binary Search, Position.

1. INTRODUCTION

Today real world getting tremendous amounts of data from various sources like data warehouse, data marts etc. To search for particular information we need to arrange this data in a sensible order. Many years ago, it was estimated that more than half the time on commercial computers was spent in sorting. Fortunately variety of sorting algorithms came into existence with different techniques [1].

Many algorithms are well known for sorting the unordered lists. Most important of them are merge sort, heap sort, shell sort and quick sort etc. [2]. As stated in [3], sorting has been considered as a fundamental problem in the study of algorithms, that due to many reasons:

- The need to sort the information is inherent in many applications.
- Algorithms often use sorting as a key subroutine.
- Many engineering issues come to the fore when implementing sorting algorithms.
- In algorithm design, there are many essential techniques represented in the body of sorting algorithms.

Sorting algorithms plays a vital role in various indexing techniques used in data warehousing, and daily transactions in online Transactionalprocessing (OLTP). Efficient sorting is important to optimize the use of other sorting algorithms that require sorted lists correctly. Sorting algorithms can be classified by:

1. Computational complexity (best, average and worst behavior) of element comparisons in terms list size n. For a typical sorting algorithm best case, average case and worst case is O(n log n), example merge sort.
2. Number of swaps
3. Stability: A sorting algorithm is stable if whenever there are two records X and Y, with the same key and X appearing before Y in original list, X will be appear before Y in the sorted list.
4. Usage of memory

In this paper, a new sorting algorithm (Index sort) is proposed; here the basic idea is first we are going to sort the 2 elements and these 2 elements acts as an index to next 4 elements to sort, so totally 6 elements sorted, next these 6 elements are used as an index to sort 12 elements so totally 18 elements sorted, this process continues until all elements are sorted.
Section 2 presents the concept of Index sorting algorithm and its pseudo code. Section 3 shows the implementation results for various sizes of input. Finally, the conclusion was presented in section 4.

2. INDEX SORT

2.1 Concept
The main logic presented here is initially we sort forst two elements in the given input. An index is created by using these two elements which is used to sort twice the elements in the index, i.e., if index has ‘n’ number of elements, then ‘2n’ elements are sorted by this index. This process repeated until all the elements sorted. To get better results we are using binary search to search for the position of the element.

2.2 Pseudocode

function indexSort( input, noEle )
1. initialSort(input)
2. var noEle := 2
3. sort( input, noEle )
4. end indexSort

Line 1 of indexSort( input, noEle ), calls the method initialSort(input), which sort the first two elements in the given input. The pseudo code for initialSort(input) is given below.

function initialSort(input)
Sort the first two elements in the given input.
end initialSort

Line 2 of indexSort( input, noEle ), calls the sort method which sort the remaining all elements by using this index. The pseudo code for sort method is given below.

Global variables:
var length := input.length
var flag := 0
var position
function sort( input, noEle )
1. Node index[noEle]
2. copyFromArrayToNode( input, index, noEle )
3. var sortPos := 3*noEle
4. var length1 := index.length
5. if( sortPos > len )
6.    sortPos := len
7.    flag := 1
8. end if
9. for i in noEle to sortPos
10.   if ( input[i] < index[1].element )
11.     position := 1
12.   else if ( input[i] > index[len].element )
13.     position := len
14.   else
15.     Position := binarySearch( index, 1, len, input[i] )
16.     once the position is found, then compare the element input[i] with the index[position] and place input[i] in proper place by making single linked list, since index[] is of type Node.
17. end LOOP
18. copyFromNodeToArray( index, input )
19. if ( flag = 0 )
20. Sort( input, sortPos )

Line 1 of sort(input, noEle) creates an array index of type Node, Node is of structure type, the definition of Node is given below.

```
Struct Node
    int element
    Node next
end Node
```

Line 2 of sort(input, noEle) calls the method copyFromArrayToNode(input, index, noEle), this method creates an index which is used to sort the elements in the given input, the index is created by using first 'noEle' elements of the input array. The pseudo code for copyFromArrayToNode(input, index, noEle) is given below.

```
function copyFromArrayToNode(input, index, noEle)
    for i 1 to noEle
        in[i].element := input[i]
        in[i].next := NULL
    end LOOP
end copyFromArrayToNode
```

Line 4 of sort method has below type of declaration:

```
var length1 := index.length
```

here index.length returns the length of the array index, then this value is assigned to length1, same case for global variable length.

Lines 5 - 8 has a condition, which checks when to stop the sorting procedure, and limit the number of elements to the size of given input data.

Lines 9-17 sorts the given elements from noEle to 3*noEle, since all the elements up to noEle in the input array are already sorted, so by using this as index, 2*noEle elements of input array are sorted. To make this algorithm efficient we are using binary search to find out the position of the given input element in the index. The pseudocode for binarySearch is given below.

```
function binarySearch(Node array[], var low, var high, var element)
    find the position such that element is greater than or equal to array[position] and less than array[position+1].
    return position
end binarySearch.
```

Line 18 of sort method calls copyFromNodeToArray method which copy the sorted elements in the index to the input array, the pseudo code for copyFromNodeToArray is given below.

```
Function copyFromNodeToArray(Node in[], var input[])
    var length := in.length
    var count := 1
    for i in 1 to length
        while ( in[i] != NULL )
            Input[count] := in[i].element
            count := count +1
            in[i] := in[i].next
        end LOOP
    end LOOP
end copyFromNodeToArray
```

Line 20 calls the sort method recursively until all the elements in the given array are sorted. The index creation and the number of elements sorted by this index are explained by using the below table.
<table>
<thead>
<tr>
<th>S.No</th>
<th>Index created by number of elements</th>
<th>Number of elements sorted by using the index</th>
<th>Total sorted elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>36</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>54</td>
<td>108</td>
<td>162</td>
</tr>
<tr>
<td>5</td>
<td>162</td>
<td>324</td>
<td>486</td>
</tr>
<tr>
<td>6</td>
<td>486</td>
<td>972</td>
<td>1458</td>
</tr>
<tr>
<td>7</td>
<td>1458</td>
<td>2916</td>
<td>4374</td>
</tr>
<tr>
<td>8</td>
<td>4374</td>
<td>8748</td>
<td>13122</td>
</tr>
<tr>
<td>9</td>
<td>13122</td>
<td>26244</td>
<td>39366</td>
</tr>
<tr>
<td>10</td>
<td>39366</td>
<td>78732</td>
<td>118098</td>
</tr>
<tr>
<td>11</td>
<td>118098</td>
<td>236198</td>
<td>354294</td>
</tr>
<tr>
<td>12</td>
<td>354294</td>
<td>708588</td>
<td>1062882</td>
</tr>
<tr>
<td>13</td>
<td>1062882</td>
<td>2125764</td>
<td>3188646</td>
</tr>
<tr>
<td>14</td>
<td>3188646</td>
<td>6399292</td>
<td>9564938</td>
</tr>
<tr>
<td>15</td>
<td>9565938</td>
<td>10434062</td>
<td>20000000 (Actually the value is 19131876 but it exceeded the input size 2 cease)</td>
</tr>
</tbody>
</table>

**TABLE 1:** Shows how the index is created and elements are sorting.

3. IMPLEMENTATION RESULTS
Below graphs shows the implementation results of Index Sort.

**NUMBER OF ELEMENTS VS TIME (FOR RANDOM DATA)**

![Graph showing the relationship between number of elements and time](image)

**FIGURE 1:** Sorting the random elements (Number of elements Vs Time)
FIGURE 2: Sorting the random elements (Number of elements Vs Comparisons)

FIGURE 3: Sorting the reversely sorted elements (Number of elements Vs Time)
FIGURE 4: Sorting the reversely sorted elements (Number of elements Vs Comparisons)
When elements are in normal sort order (i.e. not reverse sorted), then this algorithm has time complexity of O(n*n), when elements are in reverse sorted order then this algorithm works in best case.

4. CONCLUSION
The index sorting algorithm sort the elements by creating the index recursively and sort the elements which are in reverse order very efficiently and sort the random input also in an efficient manner. For all random inputs of data the implementation results shows promising results.

REFERENCES


