Two Way Counting Position Sort

Nitin Arora  
Dept. of Computer Sci. &  
Engineering  
Uttarakhand Technical  
University, Dehradun

Anil Kumar  
Dept. of Computer Sci. &  
Engineering  
Uttarakhand Technical  
University, Dehradun

Pramod Mehra  
Dept. of Computer Sci. &  
Engineering  
Uttarakhand Technical  
University, Dehradun

ABSTRACT
Sorting is an algorithm that arranges all elements of an array, orderly. Sorting Technique is frequently used in a large variety of important applications to arrange the data in ascending or descending order. Several Sorting Algorithms of different time and space complexity are exist and used. This paper provides a novel sorting algorithm Two Way Counting Position sort, which is a modified version of Counting Position Sort Algorithm and is based on counting the position of each element in array from both the ends. We have also compared the Two Way Counting Position sort algorithm with Counting Position Sort, Bubble Sort and Selection Sort. We have used the MATLAB 8.0 for implementation and Analysis of CPU time taken by all the four Sorting Algorithms. We have checked the algorithms with random input sequence of length 10, 100, 1000, 5000, 10000, 50000. Result shows that for the small length of input sequence the performance all the three techniques is all most same, but for the large input sequence Selection sort is faster than all the three sorting techniques. Results show that Two Way Counting Position Sort is better than Counting Position Sort for all lengths of inputs.

Keywords
Bubble Sort; Position Sort; Selection Sort; Two Way Counting Position Sort.

1. INTRODUCTION
Sorting algorithms can be categories in two ways: Internal Sort and External Sort. In Internal sort the data can fit entirely into the main memory and in External Sort the data cannot fit in main memory all at once but reside in secondary storage (e.g. disk). Sorting is a technique for arrangement of objects according to some ordering criteria. Assume we have a collection of information concerning some set of objects. Assume this collection of information is organized in records. Within a record, information is structured into a number of units called fields. The data structure of a record depends on the application. There are many sorting algorithms. No single sorting technique is “best” for all applications. The sorting problems are to find a permutation such that if the ordering relations is “>” then Key(i-1) > Key(i). A sorting technique is called stable if Key(i) = Key(j), element ‘i’ precedes element ‘j’ then in the sorted list element ‘i’ also precedes element ‘j’.

Rest of the paper is organized as follows: section 2 describes the Related work in this we have discussed many sorting algorithms. Section 3 describes our new sorting algorithm. Performance analysis and comparison is described in section 4. Followed by conclusion and future scope in section 5 and used references are described in section 6.

2. RELATED WORK
2.1. Bubble Sort
The bubble sort is an exchange sort. It involves the repeated comparison and, if necessary, the exchange of adjacent elements. The elements are like bubbles in a tank of water—each seeks its own level [1, 3, 4]. For example, if the Bubble Sort were used on the array, 9, 1, 10, 7, 3, 11, 2, 4, each pass would be like as shown in Table 1.

Table 1. Bubble Sort for the input values 9, 1, 10, 7, 3, 11, 2, 4

<table>
<thead>
<tr>
<th>Initial</th>
<th>9</th>
<th>1</th>
<th>10</th>
<th>7</th>
<th>3</th>
<th>11</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass1</td>
<td>1</td>
<td>9</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>Pass2</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Pass3</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Pass4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Pass5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

With the Bubble Sort, the number of comparisons is always the same because the two for loops repeat the specified number of times whether the list is initially ordered or not. This means that the bubble sort always performs \( \frac{1}{2}(n^2 - n) \) Comparisons, where n is the number of elements to be sorted [4, 5].

2.2. Selection Sort
Input: a list of records: \( R_0, R_1, R_2, \ldots, R_n \).
Output: an order list of records: \( R_0, R_1, R_2, \ldots, R_n \).

Algorithm:
Step 1: \( i=0 \);
Step 2: find the largest item \( R_i \) from list \( R_0, \ldots, R_n \).
Step 3: swap \( R_i \) and \( R_{i+1} \) to produce a sequence of ordered records \( R_0, R_1, R_3, \ldots, R_{i+1} \).
Step 4: increment \( i \), repeat step 2 until \( i = n - 1 \).

For example, if the selection method were used on the array, 9, 1, 10, 7, 3, 11, 2, 4, each pass would be like as shown in Table 2.
Table 2. Selection Sort for the input values 9, 1, 10, 7, 3, 11, 2, 4

<table>
<thead>
<tr>
<th>Initial</th>
<th>9</th>
<th>1</th>
<th>10</th>
<th>7</th>
<th>3</th>
<th>11</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pass2</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>11</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Pass3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Pass4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Pass5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Pass6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Pass7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

The selection sort requires \( \frac{1}{2} (n^2 - n) \) Comparisons, where \( n \) is the number of elements to be sorted [4, 5].

2.3. Counting Position Sort

Counting Position sort is a new sorting algorithm. It is based on counting the smaller elements in the array and fixes the position of the element. Counting Position sort uses the following algorithm:

Algorithm: Counting_Position_SORT(array, n-1)
/* array is set of total n input elements */
for(i=1; i<=n; )
{
  int count = 0; j = i+1;
  while(j<=n)
  {
    if(array[i]>array[j]) then
      count++;
      j++;
  } /* end while*/
  if(count>0) then
    swap(array[i] and array[i + count]);
  else
    i++;
} /*end for loop*/

Table 3. Counting Position Sort for the input values 9, 1, 10, 7, 3, 11, 2, 4

<table>
<thead>
<tr>
<th>Initial</th>
<th>9</th>
<th>1</th>
<th>10</th>
<th>7</th>
<th>3</th>
<th>11</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pass2</td>
<td>4</td>
<td>1</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Pass3</td>
<td>7</td>
<td>1</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Pass4</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Pass5</td>
<td>10</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Pass6</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Pass7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

3. OUR MODIFIED TWO WAY COUNTING POSITION SORT ALGORITHM

Two ways counting Position Sort Algorithm is based on counting the smaller elements from both the side i.e. forward and backward and fixing the position of the elements.

Size of input array=MAX
Lower_Index=1;
Upper_Index=MAX;
count1=0;
count2=0;
while(Lower_Index<Upper_Index)
z = 0;
  for j=Lower_Index+1:Upper_Index
    if(array(Lower_Index)>array(j))
      count1=count1+1;
    end // end if
  end // end for loop
  if(count1>0)
    t=array(Lower_Index);
    array(Lower_Index)=array(count1+Lower_Index);
    array(count1+Lower_Index)=t;
    count1=0;
    z = 1;
  else
    Lower_Index=Lower_Index+1;
  end // end if else
  if(z==0)
    break;
  end
end // end outer while loop

Table 4: Two Way Counting Position Sort for the input values 9, 1, 10, 7, 3, 11, 2, 4

<table>
<thead>
<tr>
<th>Initial</th>
<th>9</th>
<th>1</th>
<th>10</th>
<th>7</th>
<th>3</th>
<th>11</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pass2</td>
<td>7</td>
<td>1</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Pass3</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>Pass4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
4. PERFORMANCE ANALYSIS
Bubble Sort, Selection Sort, CountingPosition Sort and our TwoWayCountingPosition Sort were implemented in MATLAB 8.0 and tested for the random sequence input of length 10, 100, 1000, 10000, 50000. All the three sorting algorithms were executed on machine with 32-bit Operating System having Intel(R) Pentium (R) CPU P6200 @ 2.13 GHz, 2.13 GHz and installed memory (RAM) 3.00 GB. The time taken by the CPU at execution for different inputs is shown in the table 5. The Plot of length of input and CPU time taken (msec) is shown in figure 1. Result shows that for the small length of input sequence the performance all the four techniques is almost same, but for the large input sequence Selection sort is faster than Bubble sort, CountingPosition sort and TwoWayCountingPosition Sort.

<table>
<thead>
<tr>
<th>Sorting Technique/# of Nodes</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>5,000</th>
<th>10,000</th>
<th>50,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwoWayCountingPositionSort</td>
<td>0.00468</td>
<td>1.1232</td>
<td>4.5396</td>
<td>102.8543</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountingPositionSort</td>
<td>0.0624</td>
<td>1.4196</td>
<td>5.1480</td>
<td>110.3211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BubbleSort</td>
<td>0.0390</td>
<td>1.0980</td>
<td>4.2563</td>
<td>95.1534</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SelectionSort</td>
<td>0.0156</td>
<td>0.7644</td>
<td>3.1044</td>
<td>77.3141</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. CONCLUSION AND FUTURE SCOPE
From the results it can be concluded that TwoWayCountingPosition Sorting algorithm is working well for all length of input values. It takes lesser CPU time than Counting position sort and larger CPU time than Bubble sort and Selection sort. In the future work more effective sorting algorithm can be proposed.

6. REFERENCES