International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 4 Issue: 3 51 - 54

The Index Sorting Algorithm

Anubhav Chaturvedi, Kshitij Gupte, Anjali Jivani*
Department of Computer Science & Engineering
The M. S. University of Baroda
Vadodara, India
acanubhav@gmail.com, kugupte@yahoo.com, anjali_jivani@yahoo.com

***Abstract*—**Efficient sorting and searching are corner-stones in algorithm design. In computer science it has become a deep-rooted habit to use comparison-based methods to solve these problems. In this research paper we have come up with a new algorithm which we have named 'The Index Sorting Algorithm', that sorts given list of elements in the array in O(n) time and O(n) space complexity in worst case, better than any other sorting algorithm. The algorithm is very easy to implement and understand.

Keywords*-sorting algorithm; sorting technique; time complexity; space complexity*

*****

I. INTRODUCTION

Sorting algorithms, which arrange the elements of a list in a certain order (either ascending or descending), are an important category in Computer Science. We use sorting as a technique in a number of applications related to Computer Science and Engineering. Vast amount of research has taken place in this category of algorithms, and many techniques have been developed till now. In fact the existing ones are so robust and efficient that perhaps the need to develop a new one needs courage! We have developed a new sorting algorithm which is simple and more time efficient in an environment where the data values are close to each other and the total number of elements is not very large.

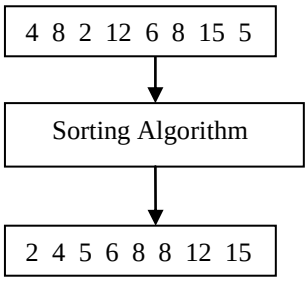

Figure 1. Sorting Technique.

II. SORTING ALGORITHMS

The sorting algorithms are generally classified into different categories based on various factors like number of comparisons, memory usage, recursion, space stacks built and many more.

Existing sorting algorithms include Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Bucket Sort and Radix sort. All the listed algorithms have their own advantages and disadvantages. Each one behaves in a different way with different data types and size of element list.

Algorithms like Bubble, Selection and Insertion sort take $O(n^2)$ time and Merge ,Heap and Quick sorts take O(nlog(n)) time and Merge Sort also takes a Space Complexity of O(n). Radix and Bucket Sorts take O(n) time ,but they ask for input restrictions(only integers allowed as input) and cannot sort all the input given. But our algorithm is able to sort any input (not asking for any input restrictions) in O(n) time complexity and O(n) space complexity. Any of the above mentioned algorithms are not capable to throw the desired result in O(n) time. In order to implement this algorithm we have used an array mapping with the input numbers, and only by doing three scans we are able to sort any kind of input (no restriction) provided to us.

III. THE INDEX SORTING ALGORITHM

This new algorithm is basically very simple to understand as well as implement. We have called it the Index Sorting Algorithm as the sorting is done as per the content of the index of the array that is being created. The Index Sorting Algorithm works as follows:

1. Take as input the unsorted elements which are to be sorted e.g. 4, 8, 2, 12, 6, 8, 15, 5.
2. Find the maximum from the input – 15
3. Create an array say A of the maximum size initializing each value with zero. – over here we create an array of size 15 i.e. A(15) with value zero for each.
4. Now read the elements one by one and considering that value as the index value of the array, increase the content of the array element at that position by one i.e. when the first element 4 is read, go to A(4) and change the content from 0 to 0+1. If the same element is repeated i.e. if 4 appears again in our element list, A(4) will become 2.
5. Keep on repeating the above till all unsorted elements are completed.
6. Now simply scan the array from the first value to last value in the array A and for every non-zero value, display the index which is actually an element of the list. If the array contains more than 1 in its value that index is to be displayed that many times. Figure 2 displays the execution of the algorithm.

The above sorting technique is very easy to implement and very fast too. We need to scan the list once to find the

maximum, once to read the elements one by one and change the array content accordingly and once to read the array and display the elements i.e. three scans of n elements – O(3n) which is O(n).

The comparison between the Index Sorting Algorithm and other algorithms is given in the next section.

Unsorted Elements
4
8
2
12
6
8
15
5

Array Index	Array Value	Sorted Elements
1	0	
2	1	2
3	0	
4	1	4
5	1	5
6	1	6
7	0	
8	2	8, 8
9	0	
10	0	
11	0	
12	1	12
13	0	
14	0	
15	1	15

Figure 2. The Index Sorting Technique.

IV. IMPLEMENTATION OF THE INDEX SORT ALGORITHM

The code for the implementation of Merge Sort, Selection Sort and or Index Sort is given below. We have executed the above three algorithms with the same dataset and have compared the execution time for each one of them.
All codes are compiled in the gcc compiler version: gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04).

A. The Index Sort Algorithm

Given below is the code for the Index Sort Algorithm. As shown in the Figure 2 above, the array value is changed i.e. incremented by one if an element with that index value is present in the list. As can be seen since the element 8 occurs twice, our array A(8) = 2.
The time complexity of the above algorithms is O(n) and with only three scans ,we are able to arrive at the results. Its space complexity is O(n), as it requires auxiliary space (almost equal to the maximum element in the array) to get the result.

The Index Sort Algorithm:

```
#include<iostream>
#include<stdlib.h>
using namespace std;
void indexSort(int a[],int n)
{    int max=a[0],min=a[0];
        int j,i,temp;
        for(i=1;i<n;i++)
        {        if (min>a[i])
                {min=a[i];}
  if (max<a[i])
                {max=a[i];}
        }int*index=(int*)calloc((max+1),sizeof(int));
  //for accomodating 0 also max+1
  for (i=0;i<n;i++)
        {index[a[i]]++;}
  //store the count of occurances of the element
        cout<<endl;
  for (i=0;i<max+1;i++)
        {if (index[i]>0)
                {cout<<i<<'\t';}
                j=index[i];
  //used to print duplicated element more than once.
                if (j==1 || j==0)continue;
                while(j!=1)
        {cout<<i<<'\t';//print the duplicates
                j--;}}}
int main()
{int a[]={4,1,0,5,3,2,34, 8, 10, 3, 2, 80, 30, 33, 1,9, 2, 3, 4, 5, 6,
7, 8};
int s=sizeof(a)/sizeof(int);
//size of the input array
                indexSort(a,s);
//function call to Sorting function
                Return0;}
```

B. The Selection Sort Algorithm

Given below is the code for the Selection Sort Algorithm.

```
#include <stdio.h>
void swap(int *xp, int *yp)
{int temp = *xp;
   *xp = *yp;
   *yp = temp;}
 void selectionSort(int arr[], int n)
{ int i, j, min_idx;
 // One by one move boundary of unsorted subarray
   for (i = 0; i < n-1; i++)
   {// Find the minimum element in unsorted array
      min_idx = i;
      for (j = i+1; j < n; j++)
      if (arr[j] < arr[min_idx])
      min_idx = j;
  // Swap the found minimum element with the first
element
      swap(&arr[min_idx], &arr[i]);}}
/* Function to print an array */
void printArray(int arr[], int size)
{   int i;
   for (i=0; i < size; i++)
   printf("%d ", arr[i]);
   printf("\n");}
 // Driver program to test above functions
int main()
{ int arr[] ={4,1,0,5,3,2,34, 8, 10, 3, 2, 80, 30, 33, 1,9, 2, 3, 4,
5, 6, 7, 8,};
 int n = sizeof(arr)/sizeof(arr[0]);
   selectionSort(arr, n);
```

```
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;}
```

The Index Sort Algorithm is much faster as compared to selection sort algorithm. It is faster by ".100" seconds for an input of 24 array elements. This difference increases as the number of elements increase.

C. *The Merge Sort Algorithm*

Given below is the code for the Merge Sort Algorithm.

```
#include<iostream
using namespace std;
void merge(int arr[], int l, int m, int r)
{ int i, j, k;
 int n1 = m - l + 1;
  int n2 =  r - m;
   /* create temp arrays */
 int L[n1], R[n2];
  /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
  /* Merge the temp arrays back into arr[l..r]*/
    i = 0;  j = 0; k = l;
    while (i < n1 && j < n2)
    {if (L[i] <= R[j])
        {  arr[k] = L[i]; i++;  }
        else{arr[k] = R[j];  j++;}
        k++;}
  /* Copy the remaining elements of L[], if there are any */
    while (i < n1)
    {arr[k] = L[i];
        i++;  k++;}
   /* Copy the remaining elements of R[], if there are any */
    while (j < n2)
    {arr[k] = R[j];
        j++; k++;}}
void msort(int a[],int i,int j)
{int m;
        if(i<j)
        {m=i+((j-i)/2);
msort(a,i,m);
        msort(a,m+1,j);
        merge(a,i,m,j);}}
int main()
{int k;
int arr[] ={4,1,0,5,3,2,34, 8, 10, 3, 2, 80, 30, 33, 1,9, 2, 3, 4,
5, 6, 7, 8};
int s=sizeof(arr)/sizeof(int);
                msort(arr,0,s);
        cout<<"The sorted array is";
        cout<<endl;
        for (k=0;k<s;k++)
       {cout<<arr[k]<<"-";}
cout<<endl;}
```

We observe from the output that our algorithm is faster than merge sort algorithm by 0.008 seconds and this difference will increase with the number of input. Both Merge sort [O(nlog(n))] and Quick Sort [O(nlog(n)) -best case] use recursion to sort the input, also, the running time of quick sort in the worst case is $O(n^2)$. There are always certain overheads involved while calling the function. Time has to be spent on passing values, passing control, returning values and returning control. Recursion also takes a lot of stack space. Every time a function calls itself memory has to be allocated, and recursion also affects the efficiency of the time complexity.

V. COMPARISON OF SORTING ALGORITHMS

In our sorting algorithm we are eliminating all the issues of recursion and reducing the time complexity. In comparison with Merge Sort and Quick Sort we are almost cutting the time by log(n) and making algorithm to run in O(n) time.

The time outputs of all the three implemented sorting algorithms are as under:

linux:/home/cse/programs# time -o indexsort

a. real 0m0.179s
b. user 0m0.135s
c. sys 0m0.044s

linux:/home/cse/programs# time -o mergesort

a. real 0m0.187s
b. user 0m0.132s
c. sys 0m0.044s

linux:/home/cse/programs# time -o selectionsort

a. real 0m0.270s
b. user 0m0.131s
c. sys 0m0.045s

Table 1 below depicts the time and space complexity comparison between the popular sorting algorithms. The table clearly shows that the Index Sort Algorithm is very efficient in terms of time complexity but has drawbacks in space complexity.

VI. CONCLUSION

The advantages and disadvantages of the Index Sorting Algorithm can be briefly described point-wise as given below. The only disadvantage is when the elements are less but the difference in their values is very high, unnecessarily a big array needs to be created.

Advantages:

1. It is free from recursion and extra overhead, and requires only three scans to get the result.
2. Time efficient about O(n) in even the worst case.
3. Simple code which is easy to understand.

Disadvantages:

1. It requires extra space as per input provided.

Wastage of space if the input is short and has large number. For example {1,890,56,345} – unnecessarily an array of 890 size needs to be created to sort just four elements.

REFERENCES

[1] Dennis Ritchie, "The C Programming Language
[2] Tremblay, Sorenson, "Introduction to Data Structures"
[3] Lipschultz, " Data Structures", Schaum Series
[4] Robert Kruse, "Data Structures using C", PHI
[5] D. Samantha, " Classic Data Structures, PHI
[6] Cormen, "Introduction to Algorithms"

TABLE I. COMPARISON OF SORTING ALGORITHMS

Algorithm	**Complexity**			
	Space	***Time***		
	Worst	***Best***	***Average***	***Worst***
Bubble sort	O(1)	O(n)	$O(n^2)$	$O(n^2)$
Bucket sort	O(n)	O(n+k)	O(n+k)	$O(n^2)$
Merge sort	O(n)	O(nlog(n))	O(nlog(n))	O(nlog(n))
Quick sort	O(log(n))	O(nlog(n))	O(nlog(n))	$O(n^2)$
Heap sort	O(1)	O(nlog(n))	O(nlog(n))	O(nlog(n))
Insertion sort	O(1)	O(n)	$O(n^2)$	$O(n^2)$
Radix sort	O(n+k)	O(nk)	O(nk)	O(nk)
Selection sort	O(1)	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bucket sort	O(1)	O(n)	$O(n^2)$	$O(n^2)$
Index sort	O(n)	O(n)	O(n)	O(n)