Implementation of the technique of Space minimization for Counting Sort algorithm

Sanjeev Kumar Sharma
Professor and Dean
JP Institute of Engineering & Technology, Meerut
dean.ar@jpiet.com

Prem Sagar Sharma
Research Scholar
Premsagar1987@rediffmail.com

Abstract

Today we consider the question of whether it is possible to sort without the use of comparisons. They answer is yes, but only under very restrictive circumstances. Many applications involve sorting small integers (e.g. sorting the employees according employee Id, sorting the list of students according roll numbers of the student etc.). We present an algorithm based on the idea of speeding up sorting in special cases, by not making comparisons.

Key words: Introduction, Literature review, TPS Algorithm, complexity of TPS sort, Advantages, Disadvantages, Applications, sorting algorithm Implementation

1. Introduction

Sorting is always carried out technically. In computer science and mathematics; we can formulate a procedure for sorting unordered array or a file. Such procedure is always governed by an algorithm; called Sorting Algorithm. Algorithms are paramount in computer programming, but an algorithm could be of no use even though it is correct and gives a desired output if the resources like storage and time it needs to run to completion are intolerable. This paper presents a Linear sorting algorithm which sorts the element in O (n) time.

Generally, Computational complexity (worst, average and best behavior) of element comparisons in terms of the size of the unsorted list is considered for the analysis of the efficiency of sorting algorithm. The complexity notational terminology is covered in [2]. If the size of unsorted list is (n), then for typical sorting algorithms, good behavior is O (n log n) and bad behavior is (n^2). The Ideal behavior is O (n). Sort algorithms which only use an abstract key comparison operation always need (n log n) comparisons in the worst case. An important criterion used to rate sorting algorithms is their running time. Running time is measured by the number of computational steps it takes the sorting algorithm to terminate when sorting n records. We say that an algorithm is O (n^2) ("of order n-squared") if the number of computational steps needed to terminate as n tends to infinity increases in proportion to n^2.

Literature review carried out in indicates the man’s longing efforts to improve efficiency of sorting algorithm with respect to time.

2. Literature review

Counting sort assumes that each input is an integer in the range from 1 to *k*. The algorithm sorts in Θ (*n* + *k*) time. If *k* is known to be *O* (*n*), then this implies that the resulting sorting algorithm is Θ (*n*) time. The basic idea is to determine, for each element in the input array, its *rank* in the final sorted array. Recall that the rank of a item is the number of elements in the array that are less than or equal to it. Notice that once you know the rank of every element, you sort by simply copying each element to the appropriate location of the final sorted output array. ***The question is how to find the rank of an element without comparing it to the other elements of the array?*** Counting sort uses the following ***three*** arrays. As usual A[1…n] is the input array. Recall that although we usually think of *A* as just being a list of numbers, it is actually a list of records, and the numeric value is the *key* on which the list is being sorted. In this algorithm we will be a little more careful to distinguish the entire record *A*[*j*] from the key A[j].key.

Counting Sort

```
CountingSort(int n, int k, array A, array B) {        // sort A[1..n] to B[1..n]
for x = 1 to k do R[x] = 0                              // initialize R
for j = 1 to n do R[A[j].key]++                          // R[x] = #(A[j] == x)
for x = 2 to k do R[x] += R[x-1]                         // R[x] = rank of x
for j = n downto 1 do {                                  // move each element of A to B
x = A[j].key                                             // x = key value
B[R[x]] = A[j]                                   // R[x] is where to put it
R[x]--                                                   // leave space for duplicates
}
}
```

We use three arrays:

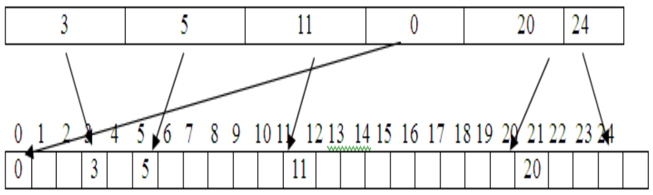
A[1..n]	Holds the initial input. *A*[*j*] is a record. *A*[*j*].*key* is the integer key value on which to sort.
B[1..n]	Array of records which holds the sorted output.
R[1..k]	An array of integers. *R*[*x*] is the rank of *x* in *A*, where *x* □ [1..*k*].

The algorithm is remarkably simple, but deceptively clever. The algorithm operates by first constructing *R*. We do this in two steps. First we set *R*[*x*] to be the number of elements of *A*[*j*] whose key is equal to *x*. We can do this initializing *R* to zero, and then for each *j*, from 1 to *n*, we increment *R*[*A*[*j*].*key*] by 1. Thus, if *A*[*j*].*key* = 5, then the 5th element of *R* is incremented, indicating that we have seen one more 5. To determine the number of elements that are less than or equal to *x*, we replace *R*[*x*] with the sum of elements in the sub array *R*[1..*x*]. This is done by just keeping a running total of the elements of *R*. Now *R*[*x*] now contains the rank of *x*. This means that if *x* = *A*[*j*].*key* then the final position of *A*[*j*] should be at position *R*[*x*] in the final sorted array. Thus, we set *B*[*R*[*x*]] = *A*[*j*]. Notice that this copies the entire record, not just the key value. There is a subtlety here however. We need to be careful if there are duplicates, since we do not want them to overwrite the same location of *B*. To do this, we decrement *R*[*i*] after copying. There are four (unnested) loops, executed *k* times, *n* times, *k* − 1 times, and *n* times, respectively, so the total running time is Θ (*n* + *k*) time. If *k* = *O* (*n*), then the total running time is Θ (*n*).

This paper is important as Sorting algorithms are often prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts. Sorting algorithms illustrate a number of important principles of algorithm design; some of them are also counterintuitive. Efficient sorting is important to optimizing the use of other algorithms such as Binary search and merge algorithms that require sorted lists to work correctly. We can not deploy binary search if data is not pre sorted otherwise the search process may get trapped into a blind alley thereby exhibiting worst case complexity.

3. TPS Sort:

The basic idea of this sort is to place the element in the position (index) what is its own value. That is an element ‘x’ would be placed in the x’th position of the array.



So if there is no repetition of the values, the elements will be well sorted very effortlessly. This **sort runs in O (n)** time.

In this sort we assume that the input is an array **arr**[0,1,…n], and the maximum of the all elements is **max.** A temporary storage (array) **temp []** provides the intermediate working storage that holds the sorted elements with some possibly empty (NULL) entries.

TPS Sorting Algorithm:

```
TPS_SORT (arr[], max)
    Step1.  temp[max]        //allocating array of size equal to the maximum of input
    Step2.  For every element of arr []

            //if repetition allowed *******

            IF temp [arr []] is FREE
               temp [arr []] =arr []
                 ELSE
                  Maintain a linked list to store arr [] //

    Step3.  temp [arr[]] =arr[]
    Step4.  For every non-NULL values of temp []
    Step5. Return temp[]
```

SORT ALGORITHMS REVIEWS

	Keys Type	**Average run-time**	**Worst case run-time**	**Extra Space**	**In Place**	**Stable**
Insertion Sort	Any	O(n2)	O(n2)	O(1)	√	√
Merge Sort	Any	O(nlogn)	O(nlogn)	O(n)	X	√
Heap Sort	Any	O(nlogn)	O(nlogn)	O(1)	√	X
Quick Sort	Any	O(nlogn)	O(n2)	O(1)	√	X
Counting Sort	integers [1..k]	O(n+k)	O(n+k)	O(n+k)	X	√
TPS Sort	integers [1…n]	O(n)	O(n)	O(n)	X	√
Radix Sort	d digits in base b	O(d(b+n))	O(d(b+n))	Depends on the stable sort used	Depends on the stable sort used	√
Bucket sort	[0,1)	O(n)	O(n2)	O(n)	X	√

4. Algorithm Analysis

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the output array, iterate at most k + 1 times and therefore take O(k) time. The other for loop, and the initialization of the output array, it take O(n) time. Therefore the time for the whole algorithm is the sum of the times for these steps, O (n + k). Because it uses array of length k (**max_element - min_element**), the total space usage of the algorithm is also O (k). For problem instances in which the maximum key value is significantly less than equal to the number of items , TPS sort can be highly space-

Efficient, as the only storage it uses other than its input array is the output array which uses space O(k)

```
for i=1 to k do
        *(ptr+i)  =32767        /*Initialize Output array 32767 such as Null*/
   for j=1 to n do
            *(ptr+arr[i]) =arr[i]     /*putting at the index according to it's value*/
         if(*(ptr+i)!=32767)          /*Place the elements in output array ptr[]*/
        printf("%d\n",*(ptr+i));
```

1. *The loop of lines 1 takes O(k) time*
2. *The loop of lines 2 takes O(n) time*

Therefore, the overall time of the TPS sort is O (k) + O (n) = O (k + n). In practice, we usually use TPS sort algorithm when have k = O (n), in which case running time is O (n).
The TPS sort is a stable sort i.e., multiple keys with the same value are placed in the sorted array in the same order that they appear in the input array.

Note: - That TPS sort beats the lower bound of Ω (n log n), because it is not a comparison sort. There is no comparison between elements. Counting sort uses the actual values of the elements to index into an array. [5]

ADVANTAGES-

1) This sort can be proved one the fastest and simplest as well.
2) Very simple and Easy algorithm.
3) Complexity of RUNTIME is proportional to the array size(i.e. O(n))
4) This algorithm is even applicable for **NEGATIVE NUMBERS using pointer**
5) This algorithm is better suited for small numbers.
6) This algorithm is very effective where there is no much differences(gTPS)
 among the numbers to be sorted.

DISADVANTAGES-

1) Without dynamic memory allocation facility….it may cause memory wastage (Hardware being in trend of getting cheaper day by day, it's won't be a great issue, until and unless the difference is very big.).

APPLICATION AREAS-

Where the data come serially. For example, sorting the employees according employee Id, sorting the list of students according roll numbers of the student etc.

Sorting Algorithm Implementation

IMPLEMENTATION IN ‘C’LANGUAGE

```
#include<stdio.h>
#include<conio.h>
int max(int []);
int size;
void main()
  int i,j,*ptr,s_len,min_len,max_len;
  int num[10],sorted[10];
  clrscr();
```

ENTER THE SIZE OF INPUT PROBLEM (UNSORTED PROBLEM

```
  printf("Enter the number of inputs:");
  scanf("%d",&size);
  printf("\nEnter the numbers :\n");
  for(i=0;i<size;i++)
     scanf("%d",&num[i]);
```

FIND THE MAXIMUM KEY VALUE(S_LEN) AND MINIMUM KEY VALUE(MIN_LEN)

```
  s_len=max(num);
  min_len=min(num);
```

FIND TOTAL SIZE OF TEMPARARY ARRAY(MAX_LEN)

```
max_len=max(num)-min(num);
ptr=(int*)calloc(max_len,sizeof(int));
            /*dynamic memory allocate to *ptr*/
 for(i=min_len;i<s_len;i++)
         *(ptr+i)=32767;
      /*Initializing 32767,This could be NULL */
    if(min_len<1)
     {
      ptr=ptr-min_len;
 /*Set value of pointer to handel nagetive value*/
     }

 printf("The sorted array is:\n");
  for(i=min_len;i<=s_len;i++)
  printf("Initialize starting location of ptr : %u\n",ptr);
                 for(i=0;i<size;i++)
                 {
                        j=num[i];
/*putting at the index according to its value*/
                        *(ptr+j)=j;
                 }
                }
/*Collect only not null value as output*/
if(*(ptr+i)!=32767)
                    { printf("%d\t%u\n",*(ptr+i),(ptr+i));}
                }
  getch();
}
```

DEFINATION OF max(int) FUNCTION

```
int max(int a[])
{
  int i,max=0;
  for(i=0;i<size;i++)
     {
                          if(a[i]>max)
                            max=a[i];
     }
  return max;
}
```

FUNCTION THE SIZE OF OUTPUT ARRAY

```
int min(int a[]){
  int i,min=32767;
  for(i=0;i<size;i++)
     {
                         if(a[i]<min)
                         min=a[i];
     }
  return min;}
```

//The program here is implemented more effectively (to save memory) by introducing Minimum calculating variable ‘min’.

References-

[1]. Darlington. J, A synthesis of several sorting algorithms, Acta Inf. II, 1978, pp 1-30.

[2]. V. Havran, “Heuristic ray shooting algorithms,” Ph.D. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Nov.2000.

[3]. D. E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Boston, MA: Addison- Wesley, 1998.

[4]. Horowitz, E.,Sahni. S, Fundamentals of Computer Algorithms, Computer Science Press, Rockville. Md.

[5]. IJCSI International Journal of Computer Science Issues, Vol. 4, No. 2, 2009[6]

[6].E.M. McCreight. A space-economic suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[7].E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[8].*A. Andersson and S. Nilsson. A new efficient radix sort. In Proc. 35th Annual IEEE Symp. on Foundations of Computer Science, pp. 714-721, 1994.*