International Journal of Scientific & Engineering Research, Volume 4, Issue ł,)Ü0à 2013
ISSN 2229-5518

Geo-Mean Sort

Safal Pandita

Abstract

Sorting is an algorithm used to arrange all elements of a list in an order. It can be arranged in increasing as well as decreasing order. Several sorting algorithms with various time and space complexities exist already due to the very important applications of sorting .

This paper proposes an algorithm which is a modification of counting sort and bucket sort to produce a new stable sorting algorithm having better space complexity than counting sort without affecting it's time complexity and better worst case time complexity than generic bucket sort. It also has better time complexity than any comparison based sorting algorithm as it is not bound by O(n*log(n)) .It's best case complexity is O(n) whereas worst case complexity is O(m+n). The space complexity is ($\sqrt{mn}$ + n). Hence it is concluded from experimental and theoretical observations that this algorithm can replace counting sort and is also better than the generic bucket sort implementation .

Key Words: Algorithm, Counting Sort, Bucket Sort, Non-Comparison Sorting, Rapid Sort, Stable Sort

INTRODUCTION

A comparison sort is a type of sorting algorithm that only reads the list elements through a single abstract comparison operation (often a "less than or equal to" operator) that determines which of two elements should occur first in the final sorted list. [1]

There are fundamental limits on the performance of comparison sorts. A comparison sort must have a lower bound of Ω(n log n) comparison operations.[2]
A Non-Comparison based sorting algorithm on the other hand does not use comparisons to sort a list and is thus not bound by any lower limit. Most popular sorts include radix sort , counting sort and bucket sort.

Efficient sorting techniques (in terms of processing time) are necessary when we have to sort large number of data. Extensive research works are being conducted to find
out better techniques.

In this paper we sort the elements by using an array of arbitrary size ranging from number of elements to the maximum element.
Let number of elements be 'n' and the maximum element in the list be 'm'.
Then-

n<=size-1<=m (for m>n) ...(a)

size-1=m(for m<=n)

(a) will later be changed to-

size=$\sqrt{\mathbf{mn}}$

This is done to keep the time complexity optimum. More insights about this are in later sections.
For m<=n, this sorting algorithm behaves exactly like counting sort and hence only the case where m>n will be explained throughout this paper.

A variable 'ratio' is declared which is the number of times the input array will have to be traversed. In the first pass the elements in the range from '0' to 'size-1' are marked by raising the counter in the arbitrary array "Y" at the same position as the value of that element. This is also called counting sort with the difference that counting sort uses an array of size 'm' where 'm' is the maximum element and does the job in a single pass. After the marking phase we traverse the array Y for values other than 0 and write them in some "X" array. Similarly second pass takes place where we mark the elements from 'size' to '2*size-1' and again traverse Y and write the non-zero valued array elements in X. Then the same happens from '2*size' to '3*size-1'.This happens till we have put every element in X. Now we have a sorted list in X.

- *Safal Pandita is currently pursuing a bachelor's degree in Computer Science in Maharaja Surajmal Institute of Technology in IP University . E-mail : safalpandita@gmail.com*

International Journal of Scientific & Engineering Research, Volume 4, Issue ł0)Ü0à 2013
ISSN 2229-5518

COUNTING SORT

Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. However, it is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.[3][4][5][6]

Because counting sort uses key values as indexes into an array, it is not a comparison sort, and the Ω(n log n) lower bound for comparison sorting does not apply to it.[1] Bucket sort may be used for many of the same tasks as counting sort, with a similar time analysis; however, compared to counting sort, bucket sort requires linked lists, dynamic arrays or a large amount of preallocated memory to hold the sets of items within each bucket, whereas counting sort instead stores a single number (the count of items) per bucket.

EXAMPLE

12 18 13 6 8 4 9

Here n=7,m=18

This algorithm takes an array of size m+1.

Index	0	1	2	3	4	5	6	7	8	9
Freq	0	0	0	0	1	0	1	0	1	1
Index	10	11	12	13	14	15	16	17	18	
Freq	0	0	1	1	0	0	0	0	1	

Then we display the non zero frequency elements

PROPOSED ALGORITHM

This is a new algorithm which uses bucket sort, counting sort and a bit of mathematics to sort elements efficiently.

Here-

- arr[] is the input array
- n is the number of elements
- m is the maximum element in the array
- sp is used only when there is a memory constraint otherwise it uses default value.
- size is the geometric mean of m and n if m>n else it is m+1.
- up(upper limit) and low(lower limit) define the range in which elements will be looked for.
- temp[size] is the arbitrary array used in every pass and ans[length] is the array used to store the final sorted list.
- change is a boolean variable which remains false if no number from the input array was put in the temp array.
- ratio remains constant .

```
int* sorting (int arr[],int n,int m,int sp=-1)
{
  if(sp!=-1&&sp< sqrt(n*m)&&sp>n)
        size=sp;
  else if(m<=n)
        size=m+1;
  else
        size=sqrt(n*m);

  int temp[size],ans[n];
  int up=size,low=0,counter=0,ratio;
 bool change=false;

  ratio=ceiling((float)(m-1)/size);

  for(int i=0;i<size;i++)
    temp[i]=0;

  for( a=0;a<ratio;a++)
    {
      for(int i=0; i<n; i++)
          if(arr[i]<up&&arr[i]>=low)
            {
              temp[arr[i]%size]++;
              change=true;
                }

       up+=size,low+=size;

      if(!change)continue;
      else change=false;

        for(int i=0; i<size; i++)
       if(temp[i]!=0)
          while(temp[i]!=0)
          {
            ans[counter++]=a*size+i;
            temp[i]--;
          }
    }
    return ans;
```

International Journal of Scientific & Engineering Research, Volume 4, Issue Ł0)Ü0à 2013
ISSN 2229-5518

EXAMPLE

Let the array be arr[]:

11 14 7 2 14

12 7 13 5 7

Here n=10 , m=14

Now since m>n
size=floor($\sqrt{m * n}$)
size=11

ratio= ceiling((float)(m-1)/size)
ratio=2
This determines the number of passes.

Initialize temp[size] with 0 for all elements

First Pass-

We check for elements from 0 to size-1
and increment temp[arr[i]%size]

temp[] becomes as follows:

Index	0	1	2	3	4	5	6	7	8	9	10
Freq	0	0	1	0	0	1	0	3	0	0	0

Freq=Frequency

Since there is atleast one element which has been inserted in temp[] , the value of change will be true.

Now we traverse temp[] and for every non-zero frequency we write the elements in ans[] according to their frequencies.

ans[] becomes as follows:

Index	0	1	2	3	4
Element	2	5	7	7	7

Now we increment up and low by size.
up+=size
low+=size

Second Pass-

We check for elements from size to 2*size-1
and increment temp[arr[i]%size]

temp[] becomes as follows:

Index	0	1	2	3	4	5	6	7	8	9	10
Freq	1	1	1	2	0	0	0	0	0	0	0

Freq=Frequency

Since there is atleast one element which has been inserted in temp[] , the value of change will be true.

Now we traverse temp[] and for every non-zero frequency we write the elements in ans[] according to their frequencies.

ans[] becomes as follows:

Index	0	1	2	3	4	5	6	7	8	9
Element	2	5	7	7	7	11	12	13	14	14

This is our sorted list.

COMPLEXITY ANALYSIS

The time complexity of this algorithm in it's worst case is-

$$O(size + ratio*n + ratio*size + n)$$

Simplifying further,we get

$$O(size + \frac{(m * n)}{size} + m + n)$$

Since m and n cannot be changed for an input we can optimise the complexity by minimising the term :

$$size + \frac{(m * n)}{size}$$

This term is minimised by taking the value of size equal to the geometric mean of m and n.

$$\sqrt{m * n} + \frac{(m * n)}{\sqrt{m * n}}$$

or

$$2 * \sqrt{m * n}$$

Hence the time complexity becomes:

International Journal of Scientific & Engineering Research, Volume 4, Issue ł0)Ü0à 2013
ISSN 2229-5518

$$O(m+n+2*\sqrt{\mathbf{m}*\mathbf{n}})$$
or
$$O((\sqrt{\mathbf{m}}+\sqrt{\mathbf{n}})2)$$

Since $\sqrt{m*n}$ will be less than m when m>n , we can replace it by m and change the complexity to :

$$O(m+n)$$

Now we know that the time complexity reaches optimum level at $\sqrt{m*n}$, so size-1 does not need to have a value more than it.

$$n<=size-1<\sqrt{\mathbf{m}*\mathbf{n}}$$

The best case of this algorithm is when m<=n.
Hence we can replace m by n.
The complexity becomes :

$$\Omega(m+n)$$
or
$$\Omega(n+n)$$
or
$$\Omega(n)$$

The space complexity of this algorithm is $O(n+\sqrt{m*n})$ because of the two arrays used.One of size $\sqrt{m*n}$ and the other of size n.

SPACE-TIME TRADEOFF

This algorithm has a big advantage over other sorting algorithms. In situations where memory is a constraint and enough memory is not available for this algorithm to make an array of size ($\sqrt{m*n}$) ,then this algorithm can take an arbitrary value of size that can be chosen in the range from n to $\sqrt{m*n}$.
$n \leq size-1 \leq \sqrt{m*n}$

This value will solely depend on the available amount of memory. The time complexity gets better as we move from n to $\sqrt{m*n}$ and starts getting worse as we move from $\sqrt{m*n}$ to m. Thus the maximum and optimum value of size has been set to $\sqrt{m*n}$ .The advantage is that the time complexity remains almost identical even if size=n+1 which is the minimum possible value for size.

$$O(size+ {}^{(\mathbf{m}*\mathbf{n})}\big/_{\mathbf{size}} + m + n)$$

will become O(n+m+m+n)

or

$$O(m+n)$$

This is equal to the time complexity when size=$\sqrt{m*n}$.

The difference is that now the exact time complexity is actually O(2*m+2*n) and earlier it was $O(m+n+2\sqrt{m*n})$. Though it slightly loses speed but it definetely saves space.

Counting sort and Bucket sort will fail and won't be able to execute if memory is a constraint as they all need a fixed amount of memory which cannot be reduced but this algorithm can still sort correctly by a simple space-time tradeoff.

COMPARISON ANALYSIS

In this section , the algorithm is compared to the already existing sorting algorithms. It is done in two parts.

First it is compared with all the popular

Comparison based sorting algorithms and then it is compared with the non-comparison based sorting algorithms.

The algorithms used are-

1. Comparison Based

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort

2. Non-Comparison Based
- Counting Sort
- Generic Bucket Sort
- Radix Sort

Standard Implementations of these algorithms are used.They may not be the most efficient implentation of these algorithms.

International Journal of Scientific & Engineering Research, Volume 4, Issue ł0)Ü0à2013
ISSN 2229-5518

As we can see from the tables and graph on the next page, when m=n or rather m<=n then Geo Mean sort does the sorting in the least time as compared to all other sorting algorithms which are compared.Counting Sort also does it in the same time but when the value of m gets high, the algorithm fails whereas Geo Mean Sort still sorts by sacrificing it's optimum speed to save space.Radix Sort is slower when m<=n but gets considerably faster as compared to Geo Mean when the value of m is raised.

A graph has been plotted for m=n, between Quick sort,Merge sort,Radix sort and Geo Mean sort.It shows that Geo Mean is the fastest in this case.

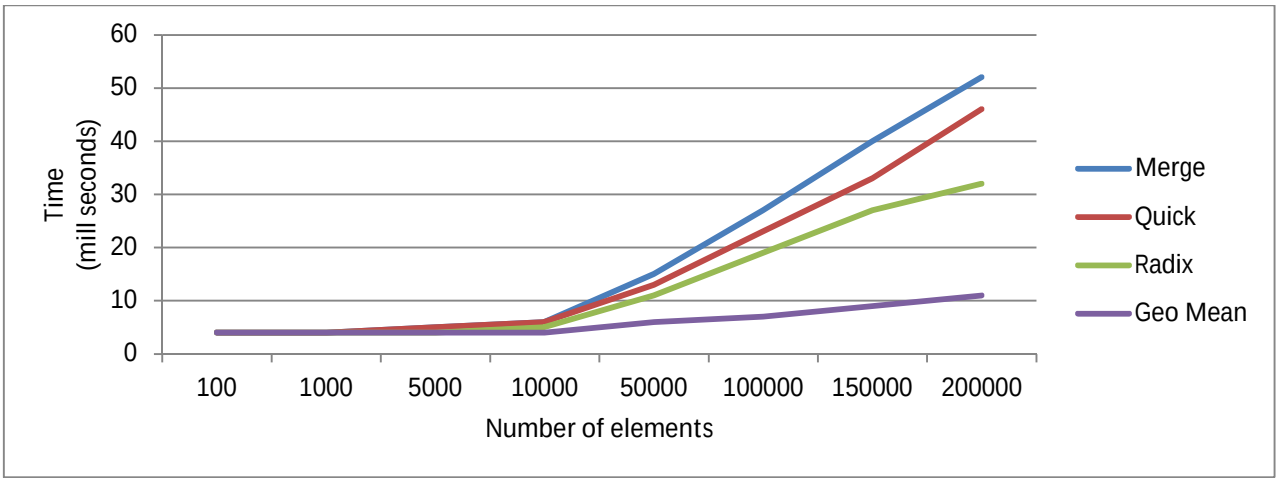
Sort / n	Bubble m=n	Selection m=n	Insertion m=n	Merge m=n	Quick m=n	Geo Mean m=n
100	4	4	4	4	4	4
1000	7	6	5	4	4	4
5000	80	35	22	5	5	4
10000	321	128	73	6	6	4
50000	8120	2908	1600	15	13	6
100000	33649	11556	6380	27	23	7
150000	**	26211	14304	40	33	9
200000	**	46410	25481	52	46	11*
250000	**	**	39714	65	55	20*

Sort / n	Counting		Generic Bucket Sort		Radix		Geo Mean	
	m=n	m=n* 10^3	m=n	m=n*10^3	m=n	m=n* 10^3	m=n	m=n*10^3
100	4	4	4	4	4	4	4	4
1000	4	-	4	-	4	4	4	4
5000	4	-	22	-	4	4	4	5
10000	4	-	72	-	5	5	4	7
50000	6	-	1608	-	11	11	6	25*
100000	7	-	6308	-	19	19	7	95*
150000	9	-	1427 1	-	27	28	9	281*
200000	-	-	-	-	32	33	11*	871*

The tables represent the time taken by sorting algorithms (in milli seconds) for different values of n.
n=number of elements m=maximum element
* signifies that optimum required memory could not be allocated and the algorithm had to sacrifice speed to make up for lesser memory
** signifies that algorithm required > 1 min to finish.
- signifies that required memory could not be allocated and hence the algorithm did not execute.

For m=n,the graph between number of elements and time taken in milli seconds is as follows:

International Journal of Scientific & Engineering Research, Volume 4, Issue Ł0)Ü0à 2013
ISSN 2229-5518



CONCLUSION

Hence we can conclude that for m<=n this sorting algorithm is better than every sorting algorithm tested which include the fastest comparison based sortings like merge sort and quick sort and also all the popular non-comparison based sorting algorithms like radix sort and counting sort.

When the value of m is too high as compared to n , even then this algorithm can sort the elements quickly by using the geometric mean of m and n as the size of the array. It can also work in cases where memory is a constraint as it can sacrifice speed for memory.Overall it can be widely used for sorting data and replace many existing sorting algorithms in various situations.

REFERENCES

1. http://en.wikipedia.org/wiki/Comparison_sort
2.
3. Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 5.3.1: Minimum-Comparison Sorting, pp. 180–197.
4.
5. 3. http://en.wikipedia.org/wiki/Counting_sort
6.
7. 4. (Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein,
8.
9. Clifford (2001) [1990]. Introduction to Algorithms [1])
10.
11. 5. Edmonds, Jeff (2008), "5.2 Counting Sort (a Stable Sort)", How to Think about Algorithms, Cambridge University Press, pp. 72–75
12.
13. 6. Sedgewick, Robert (2003), "6.10 Key-Indexed Counting", Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, and Searching (3rd ed.), Addison-Wesley