Journal of Engineering and Applied Sciences 12 (Special Issue 7): 8016-8020, 2017
ISSN: 1816-949X

A Novel Sorting Algorithm using Quotient and Remainder

[1]Abul Hasnat, [1]Tanima Bhattacharyya, [2]Atanu Dey, [3]Santanu Halder and [4]Debotosh Bhattacharjee
[1]Government College of Engineering and Textile Technology, Berhampore, West Bengal, India
[2]Indian Institute of Technology, Kharagpur, West Bengal, India
[3]Government College of Engineering and Leather Technology, Kolkata, West Bengal, India
[4]Department of Computer Science and Engineering, Jadavpur University, Kolkata,West Bengal, India

Abstract: This study proposes a novel sorting algorithm to sort an input array. The sorting algorithm works in 4 steps: searching the minimum and maximum number in the input array, calculation of the quotient and remainder for each element dividing by the minimum element, generation of a matrix for the input elements based on their calculated quotient and remainder values and row-wise traversal of the matrix gives the sorted numbers in ascending order. Time complexity of the proposed algorithm is less than all the comparison based sorting algorithms and most of the existing non-comparison based sorting algorithms.

Key words: Non-comparison sorting, quotient, remainder, sorting, time complexity, sorting algorithm

INTRODUCTION

Arranging elements of an array or list is important in many applications in computational fields. A sorting algorithm is one that arranges the elements of an array in ascending or descending order. More formally, the output must satisfy two conditions: the output is in non-decreasing or non-increasing order and the output is a permutation (reordering) of the input. Sorting algorithms are classified into two categories: comparison based sorting (Lipschutz, 2002; Horowitz *et al*., 2010; Dave and Dave, 2009; Pandey, 2008; Samanta, 2009; Aho *et al*., 1974; Mahmoud, 2000; Cormen *et al*., 2012) non-comparison based sorting (Mahmoud, 2000; Cormen *et al*., 2012). Generally comparison based sorting algorithms (i.e., bubble sort, insertion sort, quick sort, etc.) are used to sort an array by comparing pair of elements. Best case time complexity (lower bound) (Cormen *et al*., 2012) for these conventional algorithms found in literature is O(nlogn). Another class of algorithms for sorting elements in an array is non-comparison based one which needs many assumptions about the input array (i.e., for count sort, elements should be integers in small range for bucket sort, elements should be uniformly distributed, etc.). The time complexity for these types of non-comparison sort algorithms is often less compared to comparison based sorting algorithms. In this study, a new sorting algorithm is proposed that runs in O(n*dlogdloglogd) time complexity and it is less than the time complexity of the existing algorithms.

Existing non-comparison based sorting algorithm: In this study, existing non-comparison based sorting algorithm are discussed because time complexity of comparison based sorting algorithms are higher (Lipschutz, 2002; Horowitz *et al*., 2010; Dave and Dave, 2009) than, that of non-comparison based sorting algorithms. Non- comparison based sorting algorithms like counting sort, radix sort, bucket sort, pigeonhole sort are found in the literature. Counting sort (Mahmoud, 2000; Cormen *et al*., 2012) is an algorithm for sorting a collection of n integers where each element $\Theta(n+k)$ lies in the range $0 \le x \le k$; k is a non-negative integer. By using array indexing as a tool for determining relative order, counting sort can sort n numbers in $\Theta(n+k)$ time (Cormen *et al*., 2012). To implement the counting sort an extra array is required to store the count value which increases the space complexity of this sorting method. Given n, d-digit numbers in which each digit can take on up to k possible values, radix sort (sometimes called bin sort) correctly sorts these numbers in $\Theta(d(n+k))$ time (Lipschutz, 2002; Horowitz *et al*., 2010; Dave and Dave, 2009; Pandey, 2008; Samanta, 2009; Aho *et al*., 1974; Mahmoud, 2000; Cormen *et al*., 2012). Bucket sort needs the knowledge of the probabilistic distribution of numbers in the input. Bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently (Mahmoud, 2000; Cormen *et al*., 2012) over the interval (0 1) and in this case the average or expected running time to sort n real numbers using bucket sort is $\Theta(n)$ time (Cormen *et al*., 2012). Bucket sort takes n-element array as input and assumes that each element A(i) in the array falls in the range $0 \le A(i) < 1$. Pigeonhole sorting which is also known as count sort is a sorting algorithm that is suitable for sorting array of size n and the

Corresponding Author: Abul Hasnat, Government College of Engineering and Textile Technology, Berhampore, West Bengal, India

number of possible key values, k are approximately the same. Time complexity of the pigeonhole sorting is O(n+k) where range of key values is (0 k).

MATERIALS AND METHODS

This novel sorting algorithm is applicable on an array containing unique elements (without repetition) and also on array containing duplicate elements (with repetition) with slight modification. It is assumed that all the elements are positive integers. Let the minimum element in the list be MIN and maximum element MAX. The algorithm works in four steps: searching of the minimum and maximum element in the input array, calculation of quotient and remainder for each element of the input array dividing it by the minimum element MIN, insertion of each element into a blank matrix. For an element if quotient is q and remainder is r, then insert it into cell of qth row and (r+1)th column of the matrix and row wise traversal of the matrix gives the sorted array in ascending order.

Proposed algorithm (all elements are unique):

Input: Array of elements (contains unique elements)
Output: Sorted array
Step 1: MIN = α, MAX = 0, k = 1
Step 2: For i = 1 to n repeat step 3 and 4
Step 3: If MIN>Array (i) then MIN = Array (i)
Step 4: If MAX<Array (i) then MAX = Array (i)
Step 5: COL = MIN, ROW = $\lfloor$MAX/MIN$\rfloor$
Step 6: Create a zero matrix, MATRIX[ROW, COL] of size ROW×COL
Step 7: For i = 1 to n repeat step 8
Step 8: q = Array(i)/MIN, r = Array(i)%MIN, MATRIX (q, r+1) = Array (i)
Step 9: For i = 1 to ROW repeat step 10 and 11
Step 10: For j = 1 to COL repeat step 11
Step 11: If MATRIX (i, j) $\neq$ 0 then sorted array (k) = MATRIX (i, j) and k = k+1

A modified algorithm is used to sort an array where multiple copies of an element exist.

Proposed algorithm (repetition of an elements allowed in the array):

Input: Array of elements (containing multiple copy of an element)
Output: Sorted array
Step 1: MIN = α, MAX = 0, k = 1
Step 2: For i = 1 to n repeat step 3 and 4
Step 3: If MIN>Array (i) then MIN = Array (i)
Step 4: If MAX<Array (i) then MAX = Array (i)
Step 5: COL = MIN; ROW = $\lfloor$MAX/MIN$\rfloor$
Step 6: Create a zero matrix, MATRIX (ROW, COL) of size ROW×COL
Step 7: For i = 1 to n repeat step 8
Step 8: q = Array (i)/MIN, r = Array (i)%MIN, MATRIX [q, r+1]+1 = Array (i)
Step 9: For i = 1 to ROW repeat step 10-13
Step 10: For j = 1 to COL repeat step 11-13
Step 11: If MATRIX (i, j) $\neq$ 0 then l = MATRIX (i, j)
Step 12: For ii = 1 to repeat step 13
Step 13: Sorted array (k) = i×COL+(j-1) and k = k+1

The initial and modified sorting algorithms application is shown using example 1 and 2, respectively in the next study.

RESULTS AND DISCUSSION

Two examples illustrate the application of the proposed sorting methods. In first example, all elements of the array are unique and in the second example repetition of an element in the input array is allowed.

Example1: Let the input array is [14, 66, 37, 47, 88, 112, 77, 100, 105, 67, 80, 18, 21, 29, 34, 82, 85, 70, 58, 46, 51, 61, 101, 99, 83, 73]. In the given array, minimum and maximum elements are MIN = 14 = COL and MAX = 112, respectively. Now, using this minimum value, quotient and remainder are calculated for each element as shown in Table 1.

Here, maximum number of rows required is 7 and maximum number of columns required is 14. From Table 1 for each element calculated quotient and remainder are considered as row number and column number, respectively in Table 2.

Finally, the elements are scanned row wise and stored in an array. Hence, the sorted elements are [14, 18, 21, 29, 34, 37, 46, 51, 58, 61, 66, 67, 70, 73, 77, 80, 82, 83, 85, 88, 99, 100, 101, 105, 112].

Example 2: Let the given array is [45, 10, 27, 15, 41, 11, 12, 24, 42, 32, 37, 45, 41, 22, 32, 24, 12, 32, 40, 29, 15, 27, 28, 17, 19, 24, 41, 45, 39, 33, 26, 33, 18, 16, 12, 41, 45, 22, 32, 39, 17, 33, 15, 27, 19, 15, 37, 34, 17, 20, 41, 42, 26, 10, 12]. Here, minimum and maximum elements are MIN = 10 = COL and MAX = 45, respectively. Using this minimum value, quotient and remainder are calculated for each element of the given array as shown in Table 3.

In this case, the maximum number of rows required is ROW = $\lfloor$45/10$\rfloor$ = 4 and maximum number of columns required is 10. Next, the quotient and remainder values of Table 3 are used as the row number and column number, respectively in Table 4 like the previous example. Only number of copies (frequency) for each element is inserted in Table 4.

The traversal of the matrix works like for (1, 1) element stored is 2 and the element is 1×COL+(1-1) = 10. As the stored value is 2, so, the element 10 should be repeated twice in the sorted array at the time of traversal. Next for (1, 2) element stored is 1 zand the element is 1×COL+(2-1) = 11. As stored value in matrix in the cell (1, 2) is 1, so, 11 will appear just once in the sorted array. Likewise, the matrix is traversed and finally we get the sorted array in ascending order as [10, 10, 11, 12, 12,

Table 1: Quotient and remainder for example 1

Elements	Quotients	Remainders (%)
14	14/14 = 1	14 (14) = 0
66	66/14 = 4	66 (14) = 10
37	37/14 = 2	37 (14) = 9
47	47/14 = 3	47 (14) = 5
88	88/14 = 6	88 (14) = 4
112	112/14 = 7	112 (14) = 13
77	77/14 = 5	77 (14) = 7
100	100/14 = 7	100 (14) = 2
105	105/14 = 7	115 (14) = 7
67	67/14 = 4	67 (14) = 11
18	18/14 = 1	18 (14) = 4
21	21/14 = 1	21 (14) = 7
29	29/14 = 2	29 (14) = 1
34	34/14 = 2	34 (14) = 6
82	82/14 = 5	82 (14) = 12
85	85/14 = 6	85 (14) = 1
70	70/14 = 5	70 (14) = 0
58	58/14 = 4	58 (14) = 4
46	46/14 = 3	46 (14) = 4
51	51/14 = 3	51 (14) = 9
61	61/14 = 4	61 (14) = 5
101	101/14 = 7	101 (14) = 3
99	99/14 = 7	99 (14) = 1
83	83/14 = 5	83 (14) = 13
73	73/14 = 5	73 (14) = 3

Table 2: Constructed matrix of example 1

	Column													
Rows	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	14	-	-	-	18	-	-	21	-	-	-	-	-	-
2	-	29	-	-	-	-	34	-	-	37	-	-	-	-
3	-	-	-	-	46	-	-	-	-	51	-	-	-	-
4	-	-	58	-	-	61	-	-	-	-	66	67	-	-
5	70	-	-	73	-	-	-	77	-	-	80	-	82	83
6	-	85	-	-	88	-	-	-	-	-	-	-	-	
7	-	99	100	101	-	-	-	105	-	-	-	-	-	112

Table 3: Quotient and remainder for example 2

Elements	Quotients	Remainders (%)
45	45/10 = 4	45 (10) = 5
10	10/10 = 1	10 (10) = 0
27	27/10 = 2	27 (10) = 7
15	15/10 = 1	15 (10) = 5
41	41/10 = 4	41 (10) = 1
11	11/10 = 1	11 (10) = 1
12	12/10 = 1	12 (10) = 2
24	24/10 = 2	24 (10) = 4
42	42/10 = 4	42 (10) = 2
32	32/10 = 3	32 (10) = 2
37	37/10 = 3	37 (10) = 7
22	22/10 = 2	22 (10) = 2
40	40/10 = 4	40 (10) = 0
29	29/10 = 2	29 (10) = 9
28	28/10 = 2	28 (10) = 8
17	17/10 = 1	17 (10) = 7
19	19/10 = 1	19 (10) = 9
39	39/10 = 3	39 (10) = 9
33	33/10 = 3	33 (10) = 3
18	18/10 = 1	18 (10) = 8
16	16/10 = 6	16 (10) = 6
34	34/10 = 3	34 (10) = 4
26	26/10 = 2	26 (10) = 6

12, 12, 15, 15, 15, 15, 16, 17, 17, 17, 18, 19, 19, 22, 22, 24, 24, 24, 26, 27, 27, 27, 28, 29, 32, 32, 32, 32, 33, 33, 33, 34, 37, 37, 39, 39, 40, 41, 41, 41, 41, 41, 42, 42, 45, 45, 45, 45].

Table 4: Constructed matrix of example 2

	Column									
Rows	1	2	3	4	5	6	7	8	9	10
1	2	2	4	-	-	4	1	3	1	2
2	-	-	2	-	3	-	1	3	1	1
3	-	-	4	3	1	-	-	2	-	2
4	1	5	2	-	-	4	-	-	-	-

This algorithm can be applied on an array where the minimum element of the array is a negative number, 0 or 1. If the minimum element is found to be 0 or 1 or a negative number then just normalize all the numbers by adding a constant value with each element of the input array to make the minimum element a significantly higher positive integer. For example, suppose in a list minimum element is 2. Then, add 20 to each element of the array to make the new minimum element 20+2 = 22 and then application of the algorithm will give the result. Once sorting is done, subtract the added constant from each element of the sorted array to get the final sorted array. Also, the above algorithm may be applied to sort floating point numbers. Initially the floating point numbers may be converted to integers by multiplying it with 10 or 100 or 1000 as required. Then, the sorting may be applied. To get the sorted array of original numbers, each element of the array must be divided by the number that was used to multiply.

Time complexity analysis: The steps of the algorithm and its time complexity are.

Step 1: Searching the minimum and maximum element of an array takes O(n) time.

Step 2: Calculation of quotient and remainder for each element. To divide a number of d digits by divisor of d digits, computational complexity of division using Newton-Rhapson division (Halder *et al*., 2013a, b, 2015, 2014; Nandhini and Kanmani, 2012) (where, multiplication is done using Schonhage-Strassen algorithm (Schonhage and Strassen, 1971) method is O(dlogdloglogd) (Borwein and Borwein, 1998; Schonhage and Strassen, 1971). Therefore, this step takes O(n*dlogdloglogd) time (Borwein and Borwein, 1998; Schonhage and Strassen, 1971) to calculate the required quotient and remainder values.

Step 3: For an element if quotient is q and remainder is r then, we require to place it into cell of qth row and (r+1)th column of the matrix. This step takes O(n) time for n elements.

Step 4: Traversal of the matrix requires at most O (ROW×COL) or O(⌊MAX/COL⌋×COL) or O(MAX) number of operations. Hence, the time T(n) to sort n number of elements can be expressed by Eq. 1:

Table 5: Time complexity of different sorting algorithm

Algorithm	Worst case running time	Average case running time
Pigeonhole sort	NA	O(n+k)
Counting sort	O(n+k)	O(n+k)
Radix sort	O(d(n+k))	O(d(n+k))
Bucket sort	$O(n^2)$	O(n)
Proposed sort	O(n*dlogdloglogd)	O(n*dlogdloglogd)

$$T(n) \le c_1 \times n + c_2 \times n \times d\log d\log\log d + c_3 \times n + c_4 \times MAX$$
$$\text{As: } c_2 \times d\log d\log\log d \times n > c_4 \times MAX \quad (1)$$
$$\text{Therefore: } T(n) = O(n * d\log d\log\log d)$$

Therefore, the complexity of the proposed sorting algorithm is O(n*dlogdloglogd). For counting sort, the items to sort are integers in the set {0, 1, ..., k}. For radix sort, each item is a d-digit number where each digit takes on k possible values. In case of bucket sort, it is assumed that the keys are real numbers uniformly distributed in the half-open interval (0, 1). Table 5 shows the time complexity of different sorting algorithms.

Bucket sort, sorts an array containing n numbers assumes that the probabilistic distribution of the numbers are uniform over the half open interval (0, 1) and for each bucket again it uses a standard comparison based sort algorithm. Also, it is assumed that insertion in a bucket takes O (1) time then, only the bucket sort algorithm could run in O(n) time. But the proposed algorithm always sorts an input array in O(n*dlogdloglogd) time. Comparison study is given only for non-comparison based sorting algorithm because time complexity of comparison based sorting algorithm is higher (Lipschutz, 2002; Horowitz *et al*., 2010; Dave and Dave, 2009) than that of non-comparison based sorting algorithms.

Space complexity analysis: Suppose an array contains unique elements. Let the minimum and maximum elements considered are MIN and MAX, respectively. To perform sorting using proposed algorithm a two dimensional matrix is needed. Then, the number of rows and number of columns of the matrix are ⌊MAX/MIN⌋ and MIN, respectively. Therefore, the size of the matrix is MIN×MAX/MIN = MAX. So, the space complexity of the proposed algorithm is O(MAX) where MAX is the maximum element in the array. When the array contains repeated elements, let the minimum and maximum elements are MIN and MAX, respectively. To perform sorting using proposed algorithm a two dimensional matrix is required where only frequency of each element is stored. Then, the number of rows and number of columns of the matrix are MAX/MIN and MIN, respectively. Hence, the space complexity of the proposed algorithm is O(MAX) where MAX is the maximum element in the array.

CONCLUSION

In this study, a novel sorting algorithm is proposed which runs in O(n*dlogdloglogd) time complexity and space complexity is O(MAX) where MAX is the maximum element in the array. Time complexity of the proposed algorithm is less than all the comparison based sorting algorithms and most of the existing non-comparison based sorting algorithms.

ACKNOWLEDGEMENT

Researchers are thankful to GCETTB, JU Kolkata for providing facilities during research.

REFERENCES

Aho, A.V., J.E. Hopcroft and J.D. Ullman, 1974. The Design and Analysis of Computer Algorithms. Pearson Education, London, UK., ISBN:978-81-317-0250-5, Pages: 471.

Borwein, J.M. and P.B. Borwein, 1998. Pi and the AGMA Study in Analytic Number Theory and Computational Complexity. John Wiley and Sons, Hoboken, New Jersey, ISBN:9780471315155, Pages: 432.

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein, 2012. Introduction to Algorithms. 3rd Edn., PHI Learning, Delhi, India, ISBN-13:9788120340077,.

Dave, P.H. and B.H. Dave, 2009. Design and Analysis of Algorithms. Pearson Education Publication, India, ISBN: 9788131740569, Pages: 832.

Halder, S., A. Hasnat, A. Hoque, D. Bhattacharjee and M. Nasipuri, 2013a. Pipelining based floating point division: Architecture and modeling. Intl. J. Innovative Technol. Exploring Eng., 3: 15-19.

Halder, S., A. Hasnat, D. Bhattacharjee and M. Nasipuri, 2013b. A novel approach for searching of a color facial image based on the similarity of complexion. Proceedings of the 3rd International Conference on Computational Intelligence and Information Technology (CIIT'13), October 18-19, 2013, Institution of Engineering and Technology, Mumbai, India, ISBN: 978-1-84919-859-2, pp: 128-133.

Halder, S., A. Hasnat, D. Bhattacharjee and M. Nasipuri, 2014. A novel low space image storing and reconstruction method by K-means clustering algorithm. J. Appl. Math., 1: 1-13.

Halder, S., A. Hasnat, D. Bhattacharjee and M. Nasipuri, 2015. A FPGA based system for measuring vector cosine angle distance between three color channels of RGB image. Proceedings of the 3rd International Conference on Image Information Processing (ICIIP'15), December 21-24, 2015, IEEE, Waknaghat, India, ISBN: 978-1-5090-0148-4, pp: 455-461.

Horowitz, E., S. Sahni and S. Rajasekaran, 2010. Fundamentals of Computer Algorithms. Galgotia Publication Private Limited, New Delhi, India, ISBN:81-7515-257-5, Pages: 769.

Lipschutz, S., 2002. Theory and Problems of Data Structures. McGraw-Hill Education, New York, USA., ISBN:0-07-038001-5, Pages: 344.

Mahmoud, H.M., 2000. Sorting: A Distribution Theory. John Wiley & Sons, New York, USA., ISBN:0-471-32710-7, Pages: 401.

Nandhini, M. and S. Kanmani, 2012. Comparison of memetic algorithm and PSO in optimizing multi job shop scheduling. J. Eng. Appl. Sci., 7: 421-427.

Pandey, H.M., 2008. Design and Analysis of Algorithms. University Science Press, New Delhi, India, Pages: 555.

Samanta, D., 2009. Classic Data Structures. 2nd Edn., PHI Learning, Delhi, India, ISBN:978-81-203-3731-2, Pages: 784.

Schonhage, A. and V. Strassen, 1971. [Quick multiplication of large numbers (In German)]. Comput., 7: 281-292.