BIT Sorting: New Technique for Sorting

Hemkumar D
GITAM University, Bengaluru

Abstract-- **In field of computer science, there are various applications of sorting algorithm. Sorting is an operation to arrange the elements of a data structure in some logical order Sorting is data arranged in a particular fashion either in ascending or descending form. As we know that, there existing many sorting algorithms with different complexities. In this study, I am proposing a new sorting algorithm (BIT Sorting) and compared with existing algorithms in terms of complexities. Results obtained after implementation are described in a graphical form with an objective to compare the efficiency of the proposed algorithm with standard algorithm method**.

Keywords: *Sorting, Complexity analysis*

1. INTRODUCTION

In computer science, a sorting algorithm is an algorithm that puts numbers or elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for suitable data and for producing human-readable output. More formally, the output must satisfy two constraints: first constraint is that output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order). Second constraint is that output is a permutation, or reordering, of the input. Since the dawn of computing, the sorting problem has a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Sorting algorithms are 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, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds. The ultimate goal of sorting techniques is reduction in cost and complexity of the algorithm. In this paper, I propose a new sorting technique i.e., BIT Sorting derive its algorithm and compare it with well-known standard techniques. This sort is more efficient than bubble sort and it also proves to be efficient than insertion sort. All the analysis count and graphs have been provided for the researchers to check its efficiency.

In order to find which algorithm is better than other algorithms, to compare the complexity of these algorithms need to be calculated. There are two types of complexity. They are:

i) Space Complexity: Total amount of memory requires performing the algorithm.
ii) Time Complexity: Total amount of time requires performing the algorithm.

Now-a-days, when we deal about the complexity, we concentrate more on time complexity compared to space complexity. One way of comparing is based on the exact running time of all algorithms but it depends upon processor and language used. Even if the language and processor are same, calculating the exact time is more difficult as it would require CPU utilization may be different. The time complexity is dependent on the number of input elements. So, it is expressed in term of number of input size or element size. Two algorithms can be compared by using the rate of growth function f(n) of the algorithms expressed in term of number of input n. The growth function of algorithm with lesser rate is better than the other algorithm. If the rate of growth function is high when the numbers of input size or element size increase the number of operation also increase.

1.1 APPLICATIONS OF SORTING

Two important usage of sorting in terms of
1) Searching
2) Matching entries in lists.

Sorting also finds solution in various applications in order to solve many other more complex problems from areas such as optimization, graph theory and job scheduling. Next section, we have discussed some existing algorithms a like selection sort, Bubble sort, quick sort, merge sort used for sorting the elements of an array. In this paper, we have considered proper utilization of memory and also the simplicity of the algorithm. By taking these two factors, we compared our proposed algorithm with existing algorithms.

2. EXISTING SORTING TECHNIQUES

There are various many sorting algorithms. Some of the common sorting algorithms are given here.

Selection sort: The idea of selection sort is simple; we repeatedly find out the next largest element in the array and move it to its final position in the array (sorted). Assume that we wish to sort the array in increasing order, i.e. the smallest number or element at the beginning of the array and the largest element or number at the end. We begin by selecting the largest element or number and moving it to the highest index position. We can do this by swapping the element or number at the highest index and the largest element or number. We then reduce the size of the array by one element and repeat the process on the sub array. The process stops when the size of the array becomes 1

Bubble sort: The main idea of bubble sort is similar to the idea of selection sort: on each step through the algorithm, we place at least one item in its proper location. The differences between bubble sort and selection sort lie in how many times data is exchange or swap and when the algorithm terminates. Bubble sort performs more swaps in each pass or step, in the hopes that it will finish sorting the list sooner than selection sort will. Like selection sort, bubble sort works by comparing two items in the list at a time. Unlike selection sort, bubble sort will always compare two consecutive items in the list of an array, and swap or exchange them if they are out of order. If we assume that we start at the beginning of the list, this means that at each pass or step through the algorithm, the largest remaining item in the list will be placed at its proper location in the list of an array.

Quick sort: Quick sort is a very fast sorting algorithm. The algorithm itself is a bit tricky to understand, but it works very well. The basic idea of quick sort is that, choose random an element in the list of an array as a “pivot” element. Then, go through all of the elements in the list, swap items that are smaller than the pivot that are placed on the right side of the pivot , items that are larger than the pivot that are placed on the left side of the pivot. Once you’ve done all possible swaps, move the pivot to wherever it belongs in the list of an array. Now we can ignore the pivot, since it’s in position, and repeat the process for the two halves of the list (on each side of the pivot). We repeat this until all of the items or elements in the list have been sorted. Quick sort is an example of a divide and conquer algorithm. Quick sort sorts a list effectively by dividing the list into smaller and smaller lists, and sorting the smaller lists in turn faster.

Merge sort: Merge sort is a best algorithm for specific application, because it’s the “sort that sorts itself”. This means that merge sort algorithm requires very few comparisons and swaps; it instead relies on a “divide and conquer” strategy that’s slightly different from the one that quick sort uses. Merge sort starts by dividing the list to be sorted in half. Then, it divides each of these halves in half. The algorithm repeats until all of these “sublists” have exactly one element in them. At that point, each sublist of on array is sorted. In the next step of the algorithm, the sublists are gradually merged back together , until we get our sorted list of an array.

3. BIT SORTING

Consider list of elements that are stored in an array

7	1	11	14	13	9	5	6

Convert all elements into binary form

0111	0001	1011	1110	1101	1001	0101	0110

Find elements whose first leftmost bit is 1

0111	0001	**1**011	**1**110	**1**101	**1**001	0101	0110

Compare consequence bits to all selected elements until get single BIT sequence number

0111	0001	1011	1**1**10	1**1**01	1001	0101	0110

0111	0001	1011	11**1**0	1101	1001	0101	0110

Swap selected BIT sequence with last element of an array, If selected BIT sequence is last element of an array then no need to exchange. So that next iteration only checks remaining bits of an element.

0111	0001	1011	0110	1101	1001	0101	~~1110~~

Repeat step2, step3 and step4 (proposed algorithm) until all elements are sorted

2nd Iteration:

0111	0001	**1**011	0110	**1**101	**1**001	0101	~~1110~~

0111	0001	1011	0110	1**1**01	1001	0101	~~1110~~

0111	0001	1011	0110	0101	1001	~~1101~~	~~1110~~

3rd Iteration:

0111	0001	**1**011	0110	0101	**1**001	~~1101~~	~~1110~~

0111	0001	1**0**11	0110	0101	1**0**01	~~1101~~	~~1110~~

0111	0001	10**1**1	0110	0101	1001	~~1101~~	~~1110~~

0111	0001	1001	0110	0101	~~1011~~	~~1101~~	~~1110~~

4th Iteration:

0111	0001	**1**001	0110	0101	~~1011~~	~~1101~~	~~1110~~

0111	0001	0101	0110	~~1001~~	~~1011~~	~~1101~~	~~1110~~

5th iteration:

0111	0001	0101	0110	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0111	0001	0101	0110	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0111	0001	0101	0110	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0111	0001	0101	0110	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0110	0001	0101	~~0111~~	~~1001~~	~~1011~~	~~1101~~	~~1110~~

6th Iteration:

0110	0001	0101	~~0111~~	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0110	0001	0101	~~0111~~	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0110	0001	0101	~~0111~~	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0101	0001	~~0110~~	~~0111~~	~~1001~~	~~1011~~	~~1101~~	~~1110~~

7th Iteration:

0101	0001	~~0110~~	~~0111~~	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0101	0001	~~0110~~	~~0111~~	~~1001~~	~~1011~~	~~1101~~	~~1110~~

0001	~~0101~~	~~0110~~	~~0111~~	~~1001~~	~~1011~~	~~1101~~	~~1110~~

Sorted array is

0001	0101	0110	0111	1001	1011	1101	1110

Convert sorted BIT sequence elements into decimal number

1	**5**	**6**	**7**	**9**	**11**	**13**	**14**

Proposed Algorithm

```
List of elements that are stored in array
begin
        Step 1: convert all elements into binary form but converted elements are in same BIT
        sequence form.
        Step 2: find elements whose first leftmost bit is 1
        Step 3: repeat step 2 to all selected elements until get single BIT sequence number
        Step 4: swap selected BIT sequence with last element of an array
        Step 5: repeat step2, step3 and step4 until all elements are sorted
        Step 6: convert sorted BIT sequence elements into decimal number
End
```

4. Results

The behaviour of the BIT Sorting algorithm in the best case will be *O(n)*, depicting that the elements in the list are in sorted form. Similarly the average case of the running cost will be *O(n)* depending upon the number of elements in the list. There lies a significant difference between the average case running cost of BIT Sort and the other algorithms that cannot be overlooked. The results are showed in Figure 1. i.e., analysis of BIT Sorting in terms of number of comparison and number of input elements. The experiment was performed using the linear data structure array and datasets is generated using the C in-built function called rand(). This BIT Sorting algorithm is compared with the common existing algorithm like Bubble sort, Insertion sort Quick sort, Merge sort. The complexity of the algorithm depends upon the number of comparisons of respective algorithms. Based on this analysis (Figure 1), we conclude that Bit sorting is better efficiency compared to bubble sort and selection sort.

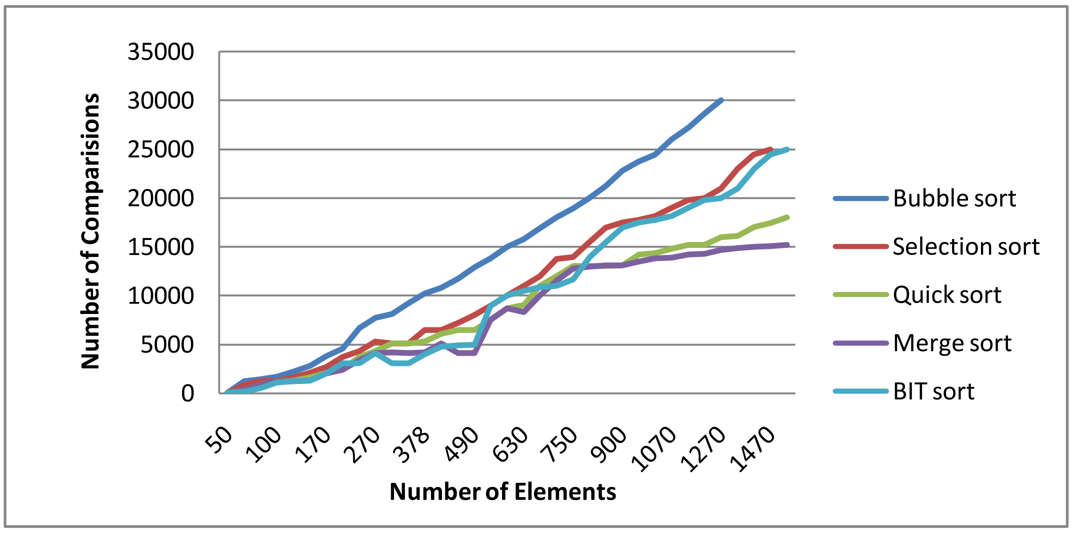

Figure 1: analysis of BIT Sorting in terms of number of comparison and number of input elements

4. Conclusion

The above figure showed that, number of comparisons with the number of inputs of different sorting. It has been found that BIT sorting algorithm is better than the existing sorting algorithm (bubble sort and selection sort) as the rate of growth is much slower than the other algorithms. By analysing the graph above, it can be easily prove that BIT Sort is clearly efficient than well-known standard sorts like Bubble sort and Selection sort. It can also be prove from the graph that the rate of growth of BIT sort over large input sizes is very steady as compared to Bubble Sort and Selection Sort.

References

[1]. Yedidyah Langsam, Moshe J. Augenstein, Aron M. Tenenbaum, *Data Structures using C and C++, Pearson Prentice Hall*, 2007,pp 355-358.

[2] Seymour Lipschutz. *Theory and Problems of Data Structures, Schaum's Outline Series*: International Edition, McGraw- Hill, 1986. ISBN 0-07-099130-8., pp. 322–323, of Section 9.3: Insertion Sort.

[3] Robert Sedgewick, *Algorithms*, Addison-Wesley 1983 (chapter 8 p 95)

[4] S.K. Srivastava, Deepali Srivastava, *Data Structures through C in depth*, BPB publication, 2011, pp 421- 424.

[5] T.H.Cormen , C.E.Leiserson, R.L.Rivest, C Stein, *Introduction to Algorithms*, 2nd ed.,PHI,pp 145-148.

[6] S.K. Srivastava, Deepali Srivastava, *Data Structures through C in depth*, BPB publication, 2011, pp 473-476.

[7] Owen Astrachan. Bubble Sort: *An Archaeological Algorithmic Analysis*. SIGCSE 2003 Hannan Akhtar. Available at: http://www.cs.duke.edu/~ola/papers/bubble.pdf.

[8] Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, *Fundamentals of Computer Algorithms*, Uiversities Press, 2009, pp 159-167.

[9] Liu Jing, *An Introduction to computer algorithms –techniques of design and analysis[M]*. Beijing: science press 2003

[10] deng Xiangyang, wan tingtin *Design and analysis of algorithms[m]* .Beijing : metallurgical Industry press , 2006.

[11] Mriganka Sarmah , Heisnam Rohen Singh, *The Rapid Sort* International Journal of Advances in Computer Science & Its Applications – IJCSIA Volume 4: Issue 3 September,2014