Proceedings of the 5th National Conference; INDIACom-2011
Computing For Nation Development, March 10 – 11, 2011
Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi

Implementing Bubble Sort Using a New Approach

V. Mansotra[1] and **Kr. Sourabh**[2]
[1,2] Department of Computer Science and IT, University of Jammu, Jammu -180006
[1]vibhakar20@yahoo.co.in and [2]Kumar9211_sourabh@gmail.com

ABSTRACT
Sorting is an important data structure which finds its place in many real life applications. A number of sorting algorithms are in existence till date and ample of research is still going on to make an extensive analysis of the existing algorithms to reduce their time complexity up to the extent possible. Research is also going on for finding the algorithms which are fast enough than the existing ones.
In this paper the authors have tried to improve upon the Bubble Sort technique by implementing the algorithm using a new approach of implementation_.An extensive analysis has been done by the authors on the new approach and the approach has been compared with the traditional method of "Bubble Sort" along with its popular variation "Bi-Directional Sort".
Interesting observations have been obtained on comparing this new approach with the existing approaches of Bubble Sort. The approach was tested for Average Case analysis, Best Case analysis and Worst case analysis. It has been observed that the new approach has given wonderful results on Average Case and Worst Case analysis. Hence the authors have reached to the conclusion through the experimental observations that the new approach as suggested in this paper is better than the traditional Bubble Sort and its Bi-Directional variation.

1. INTRODUCTION

In mathematics, computing, linguistics, and related disciplines, an **algorithm** is a finite list of well-defined instructions for accomplishing some task that, given an initial state, will proceed through a well-defined series of successive states, possibly eventually terminating in an end-state. No generally accepted *formal* definition of "algorithm" exists yet. We can, however, derive clues to the issues involved and an informal meaning of the word from the following quotation given by **Boolos and Jeffrey (1974, 1999) [1]**:
"No human being can write fast enough, or long enough, or small enough to list all members of an innumerably infinite set by writing out their names, one after another, in some notation. But humans can do something equally useful, in the case of certain innumerably infinite sets: They can give **explicit instructions for determining the nth member of the set**, for arbitrary finite n. Such instructions are to be given quite explicitly, in a form in which **they could be followed by a computing machine**, or by a **human who is capable of carrying out only very elementary operations on symbols**".
Searching and sorting algorithms have gained their importance due to their heavy use in commercial market like telephone companies, job portals, search engines etc. In computer science and mathematics, a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. Efficient sorting is important to optimize the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:
1. The output is in nondecreasing order each element is no smaller than the previous element according to the desired total order;
2. The output is a permutation, or reordering of the input.
Since a significant portion of commercial data processing involves sorting large quantity of data, efficient sorting algorithms are of considerable economic importance.
There is a good collection of algorithms on sortings techniques categorized under their execution behaviour which is known as complexity. Some algorithms like Bubble Sort, Selection Sort, Insertion Sort have complexity O(n^2) where as other algorithms like Quick Sort, Heap Sort have complexty O(nlogn). Authors have proposed a slight variation to Bubble sort by introducing a new appraoch for implementing the bubble sort

1.1 BUBBLE SORT

The bubble sort is the oldest and simplest sorting method in use. Unfortunately, it's also the slowest. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. The total number of comparisons, is $(n - 1) + (n - 2)...(2) + (1) = n(n - 1)/2$ or $O(n^2)$. The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant $O(n)$ level of complexity. General-case is an abysmal $O(n^2)$. While the insertion, selection, and shell sorts also have $O(n^2)$ complexities, they are significantly more efficient than the bubble sort**. [1].** Don Knuth, in his famous *The Art of Computer Programming*, concluded that "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems", some of which he discusses therein. Bubble sort is asymptotically equivalent in running time to insertion sort in the worst case, but the two algorithms differ greatly in the number of swaps necessary. Insertion sort needs only $O(n)$

 ISSN 0973-7529 ISBN 978-93-80544-00-7

operations if the list is already sorted, whereas naïve implementations of bubble sort (like the pseudocode below) require $O(n^2)$ operations. (This can be reduced to $O(n)$ if code is added to stop the outer loop when the inner loop performs no swaps.) **[2].** For example, in **[6]** we find: "The bubble sort is worse than selection sort for a jumbled array—it will require many more component exchanges—but it's just as good as insertion sort for a pretty well-ordered array. More important, it's usually the easiest one to write correctly." Owen says that Bubble sort's prime virtue is that it is easy to implement, but whether it is actually easier to implement than insertion or selection sort is arguable **[5].** Authors have tried to bring the Bubble Sort closer to other sorts by using a new variation.

2.PSEUDO-CODE [3]

```
function bubblesort (A : list[1..n]) {
   var int i, j;
   for i from n downto 1 {
      for j from 1 to i-1 {
         if (A[j] > A[j+1])
            swap(A[j], A[j+1])
      }
   }
}
```

2.1.1 Cocktail sort a slight variation of bubble sort

Another optimization is to alternate the direction of bubbling each time the inner loop iterates. This is *shaker sort* or *cocktail shaker sort* (see, e.g., [7, 8]). Cocktail sort, also known as bidirectional bubble sort, cocktail shaker sort, shaker sort which can also refer to a variant of selection sort), ripple sort, shuttle sort or happy hour sort, is a variation of bubble sort that is both a stable sorting algorithm and a comparison sort. The algorithm differs from bubble sort in that sorts in both directions each pass through the list. This sorting algorithm is only marginally more difficult than bubble sort to implement, and solves the problem with so-called turtles in bubble sort**.**

2.1.2 Differences from bubble sort

It differs in that instead of repeatedly passing through the list from bottom to top, it passes alternately from bottom to top and then from top to bottom. It can achieve slightly better performance than a standard bubble sort. The reason for this is that bubble sort only passes through the list in one direction and therefore can only move items backward one step each iteration.The complexity of cocktail sort in big O notation is $O(n^2)$ for both the worst case and the average case, but it becomes closer to $O(n)$ if the list is mostly ordered before applying the sorting algorithm**.[4]**

2.2 PERFORMANCE CHARACTERISTICS

Here we show that by several measures bubble sort is not simpler to code than other sorts and that its performance is terrible. We worry about bubble sort because of its inexplicable popularity. We view the use of bubble sort as an instance of a larger problem: choosing examples that are not exemplars of accepted best-practices. Such examples invariably must be unlearned later which is often an impossible task. **[5]**

Bubble sort is an indivisible operation of comparisons and interchanges where interchange of adjacent elements for successful comparisons adds to it's cost of performance.

For worst case :-> for N elements there are (n-1) comparisons for ith pass and for each successful comparison there is an interchange of adjacent element.

3. A VARIATION TO BUBBLE SORT

There is another variation presented which when tested for Average Best and Worst Cases has yielded fruitful results. A variation of bubble sort:-> Keeping the idea of moving the smallest element at its best place intact and unchanged this method works as follows. Let's divide the whole operation into three indivisible steps independent of each other.

Step 1 Find the smallest element in an array and save it in a variable say (temp) also save its location in variable say (loc).

Step 2 Replace element at (loc) by element at (loc-1) , element at (loc-) by element at (loc-2) and so on up to N where N is size of array.

Step 3 Place the saved variable at Nth position and decrement N by 1. Repeat above 3 steps until n goes to 0.

Even though bubble sort is quite efficient in its execution the authors still believe there is a possibility that the algorithm can be made more efficient. In this direction authors have incorporated a new approach of implementation of the bubble sort method. The new approach works on the principle that rather than swapping two variables using third variable a shift and replace procedure should be followed which takes less time as compared to later.

Consider an array of 5 elements.

Array elements [58 1 0 98 48]

After step 1 Temp= 0; Loc=2;

After step 2 [58 1 98 48]

Array elements

After step 3 [58 1 98 48 0]

Array elements. Array [N]=Temp;

N=N-1.Saved 0 reaches at it's position. Repeat the process for new N which is 4 that is now search and shift area reduces by 1 Finally when N reaches 0 array becomes

[98 58 48 1 0]

There are no adjacent replacements only one complete shift that takes element to its destination. Although performance of this variation can be increased by introducing a check or flag variable by setting or resetting it based on following behaviour.

If smallest element lies on the boundary i.e. Nth element then no shift and placement i.e. smallest element is at its best position

There is no need of going through 2nd and 3rd step. Example [98 108 512 85 0 1]

For N=6 after three operations array becomes

[98 108 512 85 1 0]

 ISSN 0973-7529 ISBN 978-93-80544-00-7

Loc 5 contains an element and is smallest among others from (0 to 5)
Boundary index 5 has smallest element 1 and it need no new position to be adjusted it is placed at its best position so Idea of going through next two steps can be dropped. Only N need one decrement and search area again reduces by one.

3.1 PROPOSED ALGORITHM

Here N (size of array) Loc (location of smallest element found) and flag are global variables. Array is assumed as a collection of random numbers.

Step1:a) Temp=Call Procedure find_smallest_number.
(b) Set Loc = Location of smallest number.
(c) Set flag=0 if smallest number lies on boundary of array.
Else set flag=1
Step2 If flag=1 then
Call procedure shift_ array(Loc)
Step 3 Set array[N]=Temp.
Decrement N by 1.
Repeat from 1 to 3 until N becomes 0.

4.EXPERIMENTAL RESULTS

In order to test our proposed algorithm for its efficiency the algorithm was implemented in C language on a Pentium (iv) machine with the following configuration :-

- Pentium(R) D CPU 280 GHZ 512 MB of RAM .
- Windows Operating System (XP) Professional 2002 Service Pack 2 .
- Compiler Turbo c++ (memory model huge).

In order to make a comparison of the proposed variation of bubble sort with the existing bubble sort and its bi-directional approach; the two approaches were also implemented for

1. Average case
2. Best case
3. Worst case

A number of tests were conducted for small as well as large data files ranging from 10K to 200Kdata elements. In each case the time taken by Bubble, Bi-directional and proposed variation is recorded and the experimental results as investigated by authors are demonstrated through bar charts for the average, best and worst cases.

4.1 OBSERVATIONS

As depicted in table 1 it can be observed in case of proposed variation that there is a tremendous decrease of the order of almost 50 % in reduction of time as compared to Bubble and Bi-directional sort. This is also well depicted in fig (1) for Proposed Algorithm.

In case of Best Case analysis, the algorithm works well with almost the same efficiency as Bubble sort. In this case an interesting observation has been met by the authors regarding the time taken by Bi-directional Bubble Sort .It has been observed in this case that time taken by Bi-directional Bubble Sort for all range of data as shown in fig (ii) is almost non record able.

In case of worst case analysis the proposed algorithm is again much more efficient as compared to the other two algorithms. The efficiency in this case has also been observed almost 50% better in comparison to other two algorithms.

CONCLUSION

By going through all the experimental results and their analysis one can easily conclude that the proposed algorithm is better for the data elements generated which are randomly ordered. In case of best case we may conclude that Bi-directional Bubble sort algorithm is the best algorithm to be used with regard to worst case analysis where the data elements are in reverse order it may be concluded that the proposed algorithm can be very effective for the small as well as large data files. One more interesting conclusion can be that proposed algorithm requires only comparisons of data elements but no exchange of elements where as Bubble and Bi-directional sorting techniques require comparison and exchanges. It has been observed that complexity of the algorithm does not change in any case (best, average, worse),the efficiency is due to change in some constant (n). In short authors have reached the conclusion that algorithm as proposed in this paper is efficient in comparison to the other two techniques discussed.

REFERENCES

[1]Boolos, George & Jeffrey, Richard (1974, 1980, 1989, 1999), *Computability and Logic* (4th ed.), Cambridge University Press, London, ISBN 0-521-20402-X: cf. Chapter 3 *Turing machines* where they discuss "certain enumerable sets not effectively (mechanically) enumerable".

[2] Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 106–110 of section 5.2.2: Sorting by Exchanging.

[3]Source:- http://www.codecodex.com/wiki/index.php?title=Bubble_sort# Performance .

[4] Source:- http://en.wikipedia.org/wiki/Cocktail_sort

[5] Owen Astrachan Bubble Sort: An Archaeological Algorithmic Analysis, *SIGCSE '03*, February 19-23, Reno, Nevada, USA. Copyright 2003 ACM 1-58113-648-X/03/0002.

[6] Cooper, D. *Oh My! Modula-2!* W.W. Norton, 1990.

[7] Knuth, D. *The Art of Computer Programming: Sorting and Searching*, 2 ed., vol. 3. Addison-Wesley, 1998.

[8] Iverson, K. *A Programming Language*. John Wiley,1962.

[9] http://linux.wku.edu/~lamonml/algor/sort/bubble.html

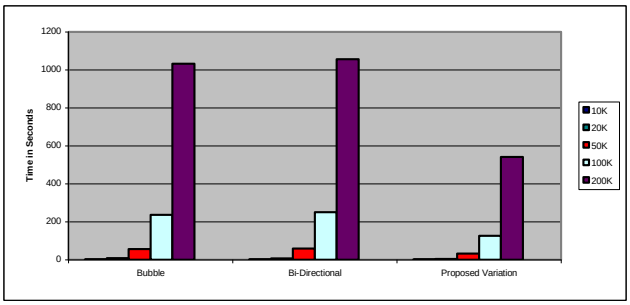

Fig-1: Graph showing Average Case analysis

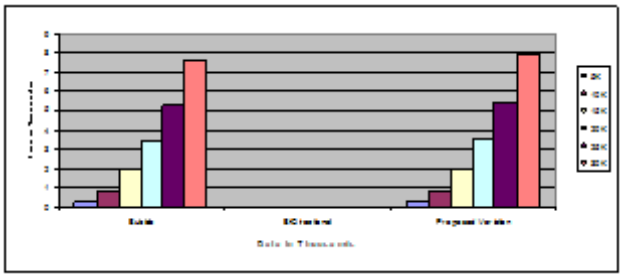

Fig-2: Graph showing Best Case analysis

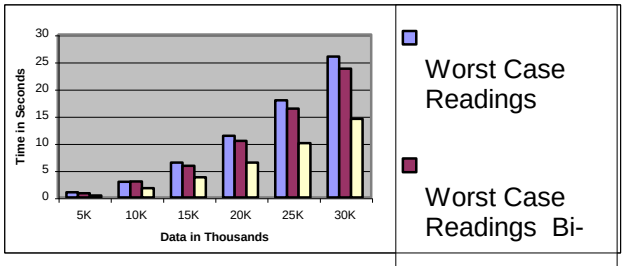

Fig-3: Graph showing Worst Case analysis

 ISSN 0973-7529 ISBN 978-93-80544-00-7

Data	**Time Taken in Seconds**		
K Stands for Thousand	Bubble Sort	Bi-directional Sort	Proposed Variation
10K	2	3	1
20K	8	10	4
50K	55	65	31
100K	236	268	125
200K	1032	1055	541

Table-1: Test results for Average Case analysis

Data	**Time Taken in Seconds**		
K Stands for Thousand	Bubble Sort	Bi-directional Sort	Proposed Variation
5K	0.21	0	0.21
10K	0.82	0	0.82
15K	1.95	0	1.97
20K	3.47	0	3.51
25K	5.32	0	5.49
30K	7.69	0	7.96

Table-2: Test results for Best Case analysis

Data	**Time Taken in Seconds**		
K Stands for Thousand	Bubble Sort	Bi-directional Sort	Proposed Variation
5K	0.93	0.76	0.38
10K	2.91	2.96	1.68
15K	6.42	5.82	3.68
20K	11.37	10.43	6.42
25K	17.91	16.37	10.00
30K	25.93	23.68	14.45

Table-3: Test results for Worst Case analysis

 ISSN 0973-7529 ISBN 978-93-80544-00-7