STUDY AND IMPROVEMENT OVER INSERTION SORT ALONG WITH COMPARATIVE ANALYSIS

Thesis

Submitted to the

G. B. PANT UNIVERSITY OF AGRICULTURE AND TECHNOLOGY
PANTNAGAR-263145, U.S. NAGAR, UTTARAKHAND, INDIA

By

ARUN SAINI
B. Tech. (Computer Science and Engineering)

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
(COMPUTER ENGINEERING)

July, 2015

ACKNOWLEDGEMENT

In this world, I come in contact with so many other people. Among these people, there are some good ones who help or support me. Acknowledgement for those might be just a trifle thing written on a piece of paper. However, I can say that it gives me a great opportunity to memorise and express my heartily feelings to those, whom I love, respect, revere and share my secrets. Now, here I get a great chance to express my token of thanks to people who in a way helped and supported me to complete this work.

It is my sublime duty to express my deepest sense of gratitude and veneration to my advisor Prof. **B.K Singh,** Associate Professor, Department of Computer Engineering, for his sincere exhortation, indelible inspiration, constant encouragement and constructive criticism. To him I owe a great debt of gratitude for his patience, supporting attitude throughout the entire work, helping me to shape my interest and ideas and preparation of this manuscript.

I express my deepest sense of reverence and indebtedness to the esteemed members of my Advisory Committee, Dr. **S.B. Singh,** Professor, Department of Mathematics, Statistics and Computer Science, Prof. **Sunita Jalal,** Assistant Professor, Department of Computer Engineering and Prof. **Shriprakash Dwivedi,** Assistant Professor, Department of Information Technology for their valuable suggestions and eternal encouragement at various stages of the experiment & thesis writing. I would also like to thank Prof. **S. D. Samantaray,** Professor and Head, Department of Computer Engineering for his able guidance, Prof. **Jalaj Sharma,** Associate Professor, Department of Computer Engineering and Dr. **Rajeev Singh,** Assistant Professor, Department of Computer Engineering for supporting and helping us and Prof. **Chetan Singh Negi** and Prof. **P.K. Mishra,** Assistant Professor, Department of Computer Engineering for giving valuable suggestions during course of my work.

I am thankful to Dr. **D. S. Murthy,** Dean, Student Welfare, **Dr. N. S. Murthy,** Dean, College of Post Graduate Studies and Dr. **H. C. Sharma,** Dean, College of Technology, G. B. Pant University of Agriculture and Technology, Pantnagar for providing necessary facilities to carry out the study.

The other teaching and non-teaching staffs, Department of Computer Engineering, and my friends Garima Chaudhary, Isha Bhardwaj, Ashwani, Shivam Saini, Sunil Singh, Daljeet Singh, Mayank Gupta, Shubham Sharma and all my class mates who made the friendly environment for working definitely deserve a special word of thanks for always being there to support and encourage me.

At this juncture of time my heart is full and I feel short of words at my command to express my respect to my beloved parents, my father Mr. Harish Chandra Saini, my mother Mrs. Saroj Saini, my elder brothers Mr. Raghuveer Singh, Mr. Tejbhadur Saini and my elder sisters Mrs. Seema Saini, Mrs. Santosh Saini and Ms. Pushpa Saini whose invaluable love and support have brought me to this position.

This list is obviously incomplete but allow me to submit that the omissions are inadvertent and I once again record my heartfelt gratitude to all those who cooperated with me in this endeavour.

Pantnagar
July, 2015

Arun Saini
(Author)

CERTIFICATE

This is to certify that the thesis entitled **"Study and Improvement over Insertion Sort along with Comparative Analysis"** submitted in partial fulfillment of the requirements for the degree of **Master of Technology** with major in **Computer Engineering** of the College of Post-Graduate Studies, G. B. Pant University of Agriculture and Technology, Pantnagar, is a record of bonafide research carried out by Mr. **Arun Saini,** Id. No. **45619** under my supervision and no part of the thesis has been submitted for any other degree or diploma.

The assistance and help received during the course of this investigation and source of literature have been duly acknowledged.

Pantnagar
July, 2015

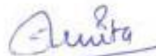
(B.K. SINGH)
Chairman
Advisory Committee

CERTIFICATE

We, the undersigned, members of the Advisory Committee of Mr. **Arun Saini,** Id. No. **45619,** a candidate for the degree of **Master of Technology in Computer Engineering** with major in **Computer Engineering,** agree that the thesis entitled ***"*Study and Improvement over Insertion Sort along with Comparative Analysis*"*,** may be submitted in partial fulfillment of the requirements for the degree.

(B.K. SINGH)
Chairman
Advisory Committee

(S.B. SINGH)
Member



(SUNITA JALAL)
Member

(SHRIPRAKASH DWIVEDI)
Member

TABLE OF CONTENTS

Acknowledgement

Certificate from Chairman, Advisory Committee

Certificate from Advisory Committee

Contents

List of Tables

List of Figures

List of Abbreviations

S. NO.	CHAPTER	PAGE NO.
1.	**INTRODUCTION**	
	1.1 Application of Sorting	
	1.1.1 Searching	
	1.1.2 Closest Pair	
	1.1.3 Element's Uniqueness	
	1.1.4 Frequency Distribution	
	1.1.5 Selection	
	1.2 Issues in Sorting	
	1.2.1 Equal Keys	
	1.2.2 Non Numerical Data	
	1.2.3 Increasing or Decreasing order	
	1.3 Categorization of Sorting	
	1.3.1 Basic operation	
	1.3.2 Stability	
	1.3.3 Requirement of additional placeholder	
	1.3.4 Comparison, stable and in-place sort	
	1.3.5 Comparison, stable and out-place sort	
	1.3.6 Comparison, unstable and in-place sort	
	1.3.7 Comparison, unstable and out-place sort	
	1.3.8 Non-comparison, stable and in-place sort	
	1.3.9 Non-comparison, stable and out-place sort	

1.3.10 Non-comparison, unstable and in-place sort
1.3.11 Non-comparison, unstable and out-place sort
1.3.12 Wild sort methods
1.4 Methods of Sorting
1.4.1 Exchange sort
1.4.2 Selection sort
1.4.3 Insertion sort
1.4.4 Merge sort
1.4.5 Distribution sort
1.4.6 Non-comparison sort
1.5 Motivation
1.6 Objective
1.7 Thesis Outline
2. REVIEW OF LITERATURE
2.1 Insertion Sort
2.2 Some other Sorting Algorithms
2.2.1 Radix sort
2.2.2 Merge sort
2.2.3 Counting sort
2.2.4 Bubble sort
2.2.5 Shell sort
2.2.6 Quick sort
2.2.7 Tree sort
2.2.8 Heap sort
2.2.9 Comb sort
2.2.10 Gnome sort
2.2.11 Pancake sort
2.2.12 Sleep sort
2.2.13 Bucket sort
2.2.14 Pigeonhole sort
2.2.15 Odd even sort
2.2.16 Bogo sort
2.2.17 Bozo sort
2.2.18 Permutation sort

2.2.19 Stooge sort
2.2.20 Taco sort
2.2.21 Strand sort
2.2.22 Cocktail sort
2.2.23 Selection sort
2.3 Chronological Order of Sorting Algorithms
3. MATERIALS AND METHODS
3.1 Software Tools Used
3.2 Dataset Used
3.3 Methodology
3.3.1 Proposed sorting approach(PSA)
3.4 Pseudo Code for Proposed Sorting Approach
3.5 Flowchart for Proposed Sorting Approach
4. RESULTS AND DISCUSSION
4.1 Performance Factors
4.1.1 Execution time
4.1.2 Operation performed
4.2 Design and Implementation
4.3 Different Case Scenarios
4.4 Experimental Results
4.5 Experimental Comparison of Sorting Methods
4.6 Discussion
4.7 Cases of Failure
5. SUMMARY AND CONCLUSIONS
5.1 Conclusion
5.2 Future Work
Literature Cited
Vita
Abstract (English)
Abstract (Hindi)

LIST OF TABLES

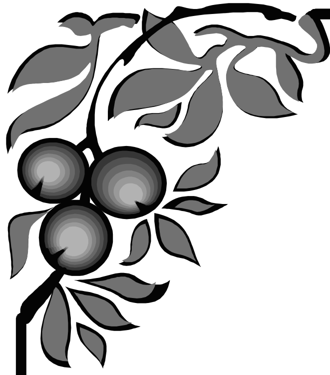
TABLE NO.	TITLE	PAGE NO.
Table 4.1	Execution time, comparisons and swaps taken by proposed sorting approach	
Table 4.2	Execution time, comparisons and swaps taken by insertion sort	
Table 4.3	Execution time, comparisons and swaps taken by bubble sort	
Table 4.4	Execution time, comparisons and swaps taken by selection sort	
Table 4.5	Number of comparisons needed by propsed, insertion, bubble and selection sort	
Table 4.6	Number of swaps needed by propsed, insertion, bubble and selection sort	
Table 4.7	Execution time for sorting techniques	
Table 4.8	Execution time	
Table 4.9	Execution time/N for sorting techniques	
Table 4.10	Execution time/N	
Table 4.11	Execution time/NlogN for sorting techniques	
Table 4.12	Execution time/NlogN	
Table 4.13	Execution time/N^2 for sorting techniques	
Table 4.14	Execution time/N^2	
Table 4.15	Execution time/logN for sorting techniques	
Table 4.16	Execution time/logN	

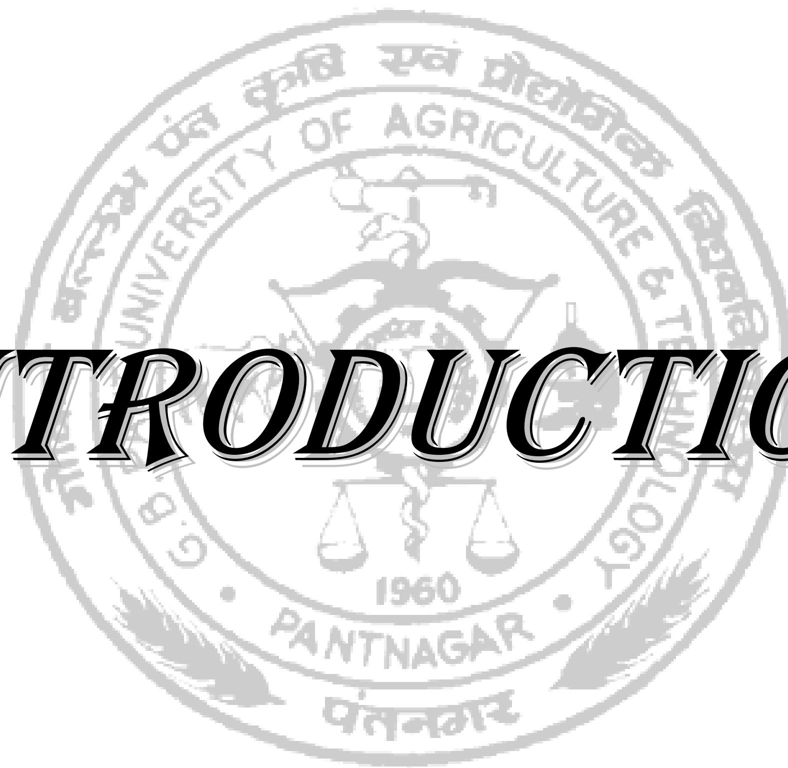
LIST OF FIGURES

FIGURE NO.	TITLE	PAGE NO.
Figure 3.1	Flowchart for Proposed Sorting Approach	
Figure 3.2	Flowchart for step 3 of Proposed Sorting Approach	
Figure 4.1	Snapshot of final result	
Figure 4.2	Execution time taken by proposed sorting approach, insertion, bubble and selection sort	
Figure 4.3	Execution time taken by proposed sorting approach, insertion, bubble and selection sort	
Figure 4.4	Number of comparisons needed by proposed sorting approach, insertion, bubble and selection sort	
Figure 4.5	Number of comparisons needed by proposed sorting approach, insertion, bubble and selection sort	
Figure 4.6	Number of comparisons needed by proposed sorting approach, insertion, bubble and selection sort	
Figure 4.7	Number of swaps needed by proposed sorting approach, insertion, bubble and selection sort	
Figure 4.8	Number of swaps needed by proposed sorting approach, insertion, bubble and selection sort	
Figure 4.9	Number of swaps needed by proposed sorting approach, insertion, bubble and selection sort	
Figure 4.10	Sum of comparisons and swaps required by different sorting methods	
Figure 4.11	Sum of comparisons and swaps required by different sorting methods	
Figure 4.12	Comparison on the basis of execution time for sorting techniques	
Figure 4.13	Comparison on the basis of time/N for sorting techniques	
Figure 4.14	Comparison on the basis of time/NlogN for sorting techniques	
Figure 4.15	Comparison on the basis of time/N^2 for sorting techniques	
Figure 4.16	Comparison on the basis of time/logN for sorting techniques	

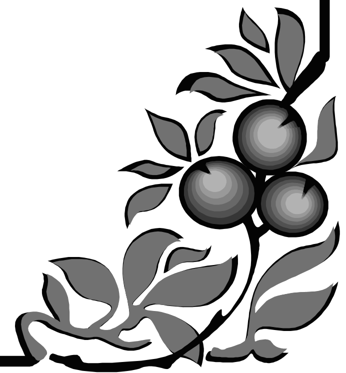
LIST OF ABBREVIATIONS

BS	Bubble Sort
BST	Binary Search Tree
e.g.	For example
Et al.	And others
Etc.	Et cetera
GB	Giga Byte
GHz	Giga Hertz
HDD	Hard Disk Drive
i.e.	That is
IS	Insertion Sort
Log	Logarithm
MAX	Maximum
MS	Milliseconds
N	Number of Elements or Problem Size
O	Big-oh
PRO	Professional
PSA	Proposed Sorting Approach
QS	Quick Sort
RQS	Randomized Quick Sort
SS	Selection Sort
WWW	World Wide Web
@	At the Rate
*	Multiplication
/	Per





INTRODUCTION



CHAPTER 1 INTRODUCTION

This chapter contains seven sections: first section presents applications of sorting, second section describes issues of sorting, section three discusses categorization of sorting, section five includes motivation, section six deals with the objectives of the proposed work in the thesis, and last section presents the thesis outline.

The concept of order is important to mankind. Order allows man to better understand the world around him. Imagine trying to find a name in a phone book if the names were not in alphabetical order. Computer science is one area where order is especially important. Much of it depends on the concept of order. Many commercial and non-commercial programming applications involve sorting of items into order.

Sorting Problem: Computing domain was basically invented to deal with data storage where sorting is needed for tasks like organizing and managing data. Sorting activity plays a vital role in computing context as it makes many questions easier to answer. A significant amount of research has already been made in the domain of sorting. However, sorting is still called an unsolved problem, which leads to further study of sorting. While looking back in the history of sorting there was no confirmation which could tell us about exact confinement of sorting problem or sorting algorithm.

Sorting is the process of arranging items according to a certain sequence or in different sets, and it has two common, but distinct meanings.
Ordering: arranging items of the same kind, class or nature, in some ordered sequence.
Categorizing: grouping and labeling items with similar properties together (by sorts).

The basic premise behind sorting an array is that its elements are in some random order and need to be arranged from lowest to highest. If the number of items to be sorted is small, a human reader may be able to tell at a glance what the correct order ought to be. If there are a large number of items, a more systematic approach is required. To do this, it is necessary to think about what it means for an array to be sorted. It is easy to see that the list
10, 15, 26, 29, 33, 45, 56, 89, 120, 325
is sorted, whereas the list

24, 11, 15, 24, 120, 45, 30, 72, 54, 400, 345

is not. The second list is not sorted because some of the adjacent elements in the list are not in order. The first item is greater than the second instead of being smaller, and likewise the fifth is greater than the sixth and so on. After knowing this observation, it is easy to find a sorting method that checks whether the adjacent elements are in order, and swap them if they are not.

1.1 Applications of Sorting

Sorting is the most fundamental algorithmic problem. Supposedly, 25% of all CPU cycles are spent for sorting by mainframes **(Wegner, 2014)**. Many different approaches lead to useful sorting algorithms, and these ideas can be used to solve many other problems. One reason why sorting is so important is that once a set of items is sorted, many other problems become easier.

1.1.1 Searching

Searching is the most important application of sorting. Binary search takes **O(logN)** time to check whether an item is in a given list or not.

1.1.2 Closest pair

Sorting is applicable to find the solution of closest pair problem that is finding the pair of numbers which are closest to each other in a given number list. If we have many data sets similar to each other, then sorting is the only technique that will help you to find the closest pair.

1.1.3 Element's uniqueness

Sorting is the only solution to element uniqueness problem which is to check whether an element is unique or not in a given list. For this we need to first sort the list and do a linear search to check all adjacent elements.

1.1.4 Frequency distribution

This problem is easily solved by sorting algorithms. We need to sort the list of elements and do a sequential search to measure the length of all adjacent elements.

1.1.5 Selection

Selection of a particular element like smallest, largest and k^{th} largest from a list is easily done by sorting algorithm. Once the keys are placed in sorted order in an array, the kth largest can be found in constant time by simply looking at k^{th} position of the array.

1.2 Issues in Sorting

There are three issues in sorting, equal keys, element's uniqueness and increasing or decreasing order as given below.

1.2.1 Equal keys

Relative order of elements matters if there are two or more elements with same values. Many stable sorting techniques are available to sort out this issue. Stable sort algorithms maintain the relative order of records having the same key values. Detailed explanation is provided in section 1.3.2 below.

1.2.2 Non-numerical data

Sorting of non-numeric data i.e. characters and text strings is called as Alphabetizing or lexicographic order. It is a challenging task as libraries have very complicated rules concerning punctuation etc. occurring in text strings.

1.2.3 Increasing or decreasing order

To find the increasing order or decreasing order of elements we need to change the comparison operator only i.e. change <= to >= in the comparison function as we desire.

1.3 Categorization of Sorting

Sorting algorithms are classified on the basis of some criteria. Some of them are as follows:

1.3.1 Basic operation

The basic operation in many sorting algorithms is to compare two elements at a time. There are still some sorting algorithms which do not use this basic operation.

1.3.2 Stability

Stable sorting algorithms are those algorithms that retain the order of occurrence of elements with same values in the list **(Wegner, 2014)**. When there are two or more than two elements with same keys in the list, then a stable sorting algorithm should be used to sort them. Stability means that if there are two records X and Y with the same key and X appearing before Y in original list, X will also appear before Y in sorted list.

1.3.3 Requirement of additional placeholder

One of the large classes of sorting algorithm is in-place sorting. In-place sorting algorithms are those algorithms that do not use additional memory or extra space and thus are generally slower than algorithms that use additional memory. Most fast stable algorithms use additional memory.

We can classify the sorting algorithms in different categories based on the above categorization criteria.

1.3.4 Comparison, stable and in-place sort

Sorting algorithms which uses comparison sort maintain the relative order of records with equal keys and not required extra memory are fall in this category. Quick sort, Gnome sort, Insertion sort, Bubble sort, Cocktail sort.

1.3.5 Comparison, stable and out-place sort

Sorting algorithms which uses comparison sort maintain the relative order of records with equal keys. These algorithms also require extra memory used to sort elements. Some of the sorting algorithms which lies in this category are Merge sort, Tree sort, Strand sort etc.

1.3.6 Comparison, unstable and in-place sort

Sorting algorithms which uses comparison sort, do not maintain the relative order of records with equal keys and not required extra memory are fall in this category. Selection sort, Shell sort, Comb sort, Pancake sort etc.

1.3.7 Comparison, unstable and out-place sort

Sorting algorithms which uses comparison sort, do not maintain the relative order of records with equal keys and required extra memory are fall in this category. Heap sort, etc.

1.3.8 Non-comparison, stable and in-place sort

Sorting algorithms which uses non comparison sort, maintain the relative order of records with equal keys and not required extra memory are fall in this category. Sleep sort etc.

1.3.9 Non-comparison, stable and out-place sort

Sorting algorithms which uses non comparison sort, maintain the relative order of records with equal keys and required extra memory are fall in this category. Bucket sort, Counting, Radix sort, Pigeonhole sort etc.

1.3.10 Non-comparison, unstable and in-place sort

Sorting algorithms which uses non comparison sort, do not maintain the relative order of records with equal keys and not required extra memory are fall in this category.

1.3.11 Non-comparison, unstable and out-place sort

Sorting algorithms which uses non comparison sort, do not maintain the relative order of records with equal keys and required extra memory to sort the data elements are fall in this category.

1.3.12 Wild sort methods

Some of the sorting algorithms based on wild approach are given below.

1. Bogo sort
2. Bozo sort
3. Permutation sort
4. Lucky sort

1.4 Methods of Sorting

There are several methods of sorting. Some of them are given below.

1.4.1 Exchange sort

If two items are found to be out of order, they are interchanged. This process is repeated until no more exchanges are necessary. Bubble sort, Cocktail sort, Comb Sort, Gnome sort, Quick sort are examples of exchange sorting methods.

1.4.2 Selection sort

In this method of sorting a particular item (largest or smallest) is to be finding. Item is separated from other items in the list after selection. This process is repeated until whole list is sorted. Selection sort, Heap sort, Smooth sort, Strand sort are examples of this type of sorting method.

1.4.3 Insertion sort

Insertion sort method takes one element at a time and place that element at its correct place in the list. In this method whole list is divided in two parts. One is sorted and other is unsorted. This method is repeated until unordered part emptied. Examples of these sorting methods are Insertion sort, Shell sort, Tree sort etc.

1.4.4 Merge sort

In this method, merging of two unsorted array into one sorted array is carried on. Merge sort is an example of this kind of sorting method.

1.4.5 Distribution sort

Segregate data elements into different lots and then sort each lot separately and concatenate all the lots.

1.4.6 Non-comparison sort

In this method of sorting number of comparisons occurred to sort the given list of elements is zero. Sorting is done without comparing any of the two elements. Radix sort, Bucket sort, Counting sort, pigeonhole sort are the sorting algorithms which belong to this type of sorting methods.

1.5 Motivation

There is an ever-increasing demand for more computing power than today's machines can deliver with the growing number of areas in which computers are being used. Usage of computers is increasing greatly in our daily life due to which applications are required to process large quantities of data in reasonable amounts of time now a days. In addition to this, the capacity of storage devices and memories was increased rapidly. Thus, there is a need for extremely fast computers. It is becoming evident very soon that it will be impossible to achieve significant increases in speed by simply using faster electronic devices, as was done in the past three decades.

Hence, we need to improve our reckoning methods for fast computations and processing on computers. As sorting is widely used in almost all computing activities, various solutions to sorting problem are available in the form of sorting algorithms. Every sorting algorithm has some merits and demerits and they are applicable to some specific kind of problem. Insertion sort is applicable to sort small data elements and quick sort is suitable for sorting large number of data elements while counting sort has its own limitations like it can be applicable to integer numbers only not characters or strings. All these drawbacks motivate us to improve performance of sorting algorithm and in this study we are trying to improve insertion sort whose performance is not so good till today.

1.6 Objective

The objective of this thesis is to study various sorting algorithms based on a literature survey and their possible implementations. The literature survey will include different approaches and improvements of the sorting algorithms, and their reported computational complexities and other properties like stability and memory requirement will be summarized and discussed. The goal is to analyze the performance of proposed sorting approach and other sorting algorithms with respect to a common data set.

The goals of my proposed work are:

1. To study the various sorting algorithms.
2. To minimize the total number of comparisons, total number of swaps and execution time required to sort the given numbers.

3. To compare the total number of comparisons, the total number of swaps and the execution time of proposed sorting approach with that of standard insertion sort.

1.7 Thesis Outline

The Thesis is organized as follows:

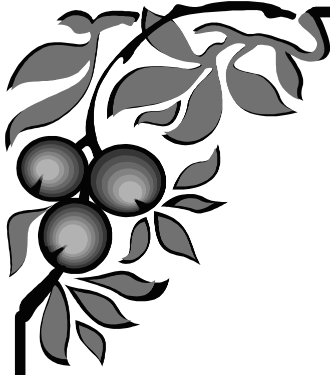
Chapter 1-This chapter contains introduction of sorting problem, research in sorting, application of sorting, motivation and objective of thesis.

Chapter 2-This chapter provides a detailed literature survey of the various sorting algorithms and work in domain.

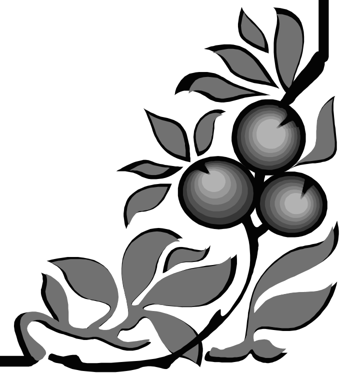
Chapter 3-This chapter deals with the solution of the problem along with the flow chart. The description of the tools used in the thesis is also provided in this chapter.

Chapter 4- This chapter describes the implementation details and experimental result, and snapshots of the output.

Chapter 5-This chapter concludes the work done and future work possible.



REVIEW OF LITERATURE



Sorting might been present in human activities but become prevalent since 19th century. Search for sorting algorithm or sorting techniques starts from the era when there was a need to store and process huge amount of data and many difficulties were faced during maintenance and organization of that data. Large data storage was required which resulted into difficulties in handling volume of data. Meanwhile an efficient technique to handle this data is invented, resulted into solution to sorting and searching problem.

Herman Hollerith, a statistician for census department of United States, invented 100 machines consisting of punch, tabulator and sorters toprocess 1890 census data. These machines were based on the idea that data could be encoded by the locations of holes in a card. Pursuing to same he developed an approach of sorting as well in 1880.

After the development of computing devices the requirement of sortingmethod was felt and hence sorting become a prominent problem in computers domain. K. Zuse was a scientist working for the commercial purpose computer named Z4. He wrote a program similar to Insertion sort **(Kadam, 2014)**.

Some of the previous work related to evolution of sorting algorithms is discussed in the following sections. First section described the insertion sort and related material. Second section included description of some other sorting algorithms. A chronological order of sorting algorithms is also presented.

2.1 Insertion Sort

John Mauchly (1946) introduced a solution to sorting problem and named it Insertion sort. Insertion sort is a simple sorting algorithm that makes the final sorted array as one item at a time.Understanding its working is very simple as it works similar to playing cards, it start with an empty left hand then remove one card at a time from the table, insert it into its correct position in the left hand. Repeat this process for each card on the table. In this way, the cards in the left hand are sorted at each iteration **(Cormen *et al.*, 2001)**. Insertion sort is simple, relatively efficient for small lists and for mostly sorted lists, and often is used as a sub-part of other sophisticated algorithms. It takes elements from the list one by one and inserts them with their correct position into sorted list.Insertion sort is expensive because it requires shifting all following elements over by one **(Nenwani *et al.*, 2014)**.

Algorithm:

Insertion-Sort (A, N)

```
1. for j = 2 to N do
2.     key = A[ j ]
3.     //Insert A[j] into the sorted sequence A[1 . . j – 1].
4.     i = j – 1
5. while i >0 and A[i ] > key
6.         A[i + 1] = A[i ]
7.         i = i – 1
8.     A[i + 1] = key
```

Performance Indices

Worst case performance	$O(N^2)$
Best case performance	$O(N)$
Average case performance	$O(N^2)$

Some of the previous work related to insertion sort is discussed below.

Srivastava *et al.* (2009) proposed bidirectional expansion to insertion sort. In this paper authors device a sorting algorithm, in which sorting starts from the middle of the array and expands both ways i.e. left and right. Since the number of iterations required is half than that in case of insertion sort, the overhead is reduced thus decreasing the running time.

Kumar *et al.* (2012) introduced a two way sorting algorithm and compared performance with existing algorithms.They called this algorithm as An End to-End Bi-directional Sorting (EEBS) Algorithm. The authors proved in the results that EEBS is much more efficient than the other algorithms having $O(N^2)$ complexity, like bubble, selection and insertion sort.

Arora *et al.* (2012) proposed a novel sorting algorithm Counting Position sort which is based on counting the position of each element in array. In this sorting algorithm, authors counted the smaller elements in the array and fix the position of the element. Result showed that for the small length of input sequence the performance of bubble sort,

selection sort and counting positions sort was almost same, but for the large input sequence Selection sort was faster than Bubble sort and Counting Position sort.

Dutta (2013) presented an approach to improve the performance of insertion sort.He showed a way to improve the performance of insertion sort technique by implementing the algorithm using a new approach of implementation. He compared this new approach with the original version of insertion sort algorithm and bubble sort and showed that the proposed approach performed better in the worst case scenario.

Chhajed *et al.* (2013) analyzed four different sorting algorithms,quick, heap, insertion and merge sort and also explained their performance analysis with respect to time complexity. These four algorithms have been an area of focus for a long time but his works attempt to explain which sorting algorithm to use in what condition.

Khairullah (2013) made enhancement to bubble sort, selection sort and insertion sort. Hisimprovement in insertion sort involves reducing shift operations with the aid of a double sized temporary array. Itsubstantially avoids unnecessary comparisons, the performance bottleneck in searching or sorting and data shifts, needing costly memory writes. He showed that avoiding or minimizing either certain comparisons or certain swaps or data movements can enhance typical inefficient algorithms.

Umar *et al.* (2014) focused on an enhancement to selection, bubble and insertion sort algorithm. In this paper a set of improved sorting algorithms are proposed which gives better performance and design idea. In this study five new sorting algorithms (bi-directional selection sort, bi-directional bubble sort, mid-bidirectional selection sort, mid-bidirectional bubble sort and linear insertion sort are described. The authors run this algorithm on different size data (integers) and experimental results showed that these enhanced algorithms are much better in terms of order of growth as well as machine running time by using Intel core i3 processor.

Bansal (2014) identified a new method of sorting known as AVI Sort. In this paper, he evaluated the time complexity $O(N^2)$ of AVI sort theoretically. His results showed a one more different way to sort the data elements in quadratic time complexity.

Nenwani *et al.* (2014) have explored the idea ofinsertion sort. They partitioned the list as in insertion sort and inserts the element from unordered list into ordered list as done

in conventional insertion sort. They used binary search for locating the position of element from unordered list in ordered list. AIS also requires additional space of size O(N) which makes the list to extend in both the sides and results in reduced number of shift operation.

2.2 Some other Sorting Algorithms

Sorting is fundamental operation used in computing domain. There are many sorting algorithms available till now. Each sorting algorithm is suitable for a specific problem. Some of them are described here.

2.2.1 Radix sort

Herman Hollerith (1880) proposed first solution to sorting problem and named it radix sort. He sorts N numbersconsisting of K digitsby processing the individual digits. Radix sort is a non-comparative integer sorting algorithm. He processed digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD) to sort those **(Cormen *et al.*, 2001)**.

The algorithm first sorts the list by its least significant digit (LSD) and keeps their relative order using a stable sort. Then it sorts them by the next digit, and this whole process goes on from the least significant to the most significant, and results of which left with a sorted list.

The LSD radix sort requires the use of a stable sort, but the MSD radix sort algorithm does not (unless stable sorting is desired) require. Hybrid sorting approach improves the performance of radix sort (**Shukla *et al.*, 2012**).

Algorithm:

Radix_sort(A, K)

1. for i = 1 to K do
2. Use a stable sorting algorithm to sort array A on digit i.
3. return

Performance Indices

Best case performance	O(d(N+K))
Average case performance	O(d(N+K))
Worst case performance	O(d(N+K))

2.2.2 Merge sort

John von Neumann (1945) invented Merge sort which takes advantage of the ease of merging already sorted lists into a new sorted list. It is a sorting algorithm based on divide and conquers approach. Merge sort starts by comparing every two elements and swap them if the first element is greater than the second **(Cormen *et al*, 2001)**. It then merges each of the resulting lists of two elements into lists of four elements, and then merges those lists, and so on; until at last two lists are merged into the final sorted list.

Merge sort has seen a relatively recent surge in its popularity for the practical implementations. It is commonly used for the standard sort in the programming languages.

Algorithm:

MERGE-SORT (A, p, r)

```
1.  if p < r
2.          q = (p + r)/2
3.          MERGE-SORT(A, p, q)
4.          MERGE-SORT(A, q + 1, r )
5.          MERGE (A, p, q, r)
```

MERGE (A, p, q, r)

```
1.  n1 =  q - p + 1
2.  n2 = r - q
3.  create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
4.  for i = 1 to n1 do
5.        L[i ] = A[p + i –1]
6.  for j = 1 to n2 do
7.        R[ j ] = A[q + j ]
8.  L[n1 + 1] = ∞
9.  R[n2 + 1] = ∞
10. i = 1
11. j = 1
12. for k = p to r do
13.       if L[i ] ≤ R[ j ]
```

14. A[k] = L[i]
15. i = i + 1
16. **else** A[k] = R[j]
17. j = j + 1

Performance Indices

Worst case performance	O(NlogN)
Best case performance	O(NlogN)
Average case performance	O(NlogN)

2.2.3 Counting sort

Harold H. Seward (1954) first introduced counting sort applicable only when each input is known to belong to a particular set, K, of possibilities. It is used as an internal part of radix sort to sort numbers. The algorithm runs in O(|K|+N) time where O(|K|) is memory used and N is the length of the input. It works by creating an integer array of size | K| and using the ith bin to count the occurrences of the ith member of K in the input.

Each input is then counted by incrementing the value of its corresponding bin. This sorting algorithm cannot often be used because to be efficient 'K' needs to bereasonably small, but the algorithm is extremely fast and demonstrates great asymptotic behavior as 'N' increases **(Beniwal *et al.,* 2013)**.

Algorithm:

Counting_sort (A,N,K)

1. Repeat for i = 1 to K do
2. C[i] = 0
3. Repeat for j = 1 to N do
4. C[A[j]] = C[A[j]] + 1
5. Repeat for i = 2 to k do
6. C[i] = C[i] + C[i–1]
7. Repeat for j = Ndown to 1 do
8. B[C[A[j]]] = A[j]
9. C[A[j]] = C[A[j]] – 1
10. Return

Performance Indices

Worst case performance	O(N+K)
Best case performance	O(N+K)
Average case performance	O(N+K)

2.2.4 Bubble sort

Iverson (1956) suggested a very simple solution to sorting problem based on exchange sort. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set.

It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average and worst case performance is $O(N^2)$, so it is rarely used to sort large, unordered, data sets **(Cormen *et al.*, 2001)**.

Algorithm:

Bubble_sort(a,n)

1. Repeat for i=0 to n-1
2. Repeat for j=0 to n-i-1
3. If a[j]>a[j+1] then exchange a[j] and [j+1]

Performance Indices

Worst case performance	$O(N^2)$
Best case performance	O(N)
Average case performance	$O(N^2)$

2.2.5 Shell sort

Donald Shell (1959) made improvement in insertion sort by moving out of order elements more than one position at a time. Shell sort can be implemented byorganizing the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort. The performance of Shell sort depends on the increment sequence (gap). After D. Shell, Robert Sedgewick did improvement to shell sort. The operation of moving larger elements one position to the right is equivalentto doing a sequence of compare-exchange operations. A[j-gap] is compared with A[j] and exchange them if necessary to put the smaller of the two items on the left.

Algorithm:

Shell_sort(A, N)

1. gap = N/2
2. Repeat while gap > 0
3. Repeat for p = gap to N-1
4. temp = A[p]
5. j = p
6. Repeat while j >= gap && temp < A[j- gap]
7. A[j] = A[j-gap]
8. j = j – gap
9. A[j] = temp
10. gap = gap /2

Performance Indices

Worst case performance	$O(N^2)$
Best case performance	depends on the gap taken
Average case performance	$O(N(\log N)^2)$

2.2.6 Quick sort

C. A. R. Hoare (1960) invented quicksort algorithm based on partitioning. To partition an array, an element called a pivot, is selected. All elements smaller than the pivot, are moved before it and all elements greater than the pivot are moved after it. In this way, whole array is divided in two lists. Both the sub lists are then sorted recursively **(Cormen *et al.*, 2001)**.

Quicksort is one of the most popular sorting algorithms and is predefined in libraries of most of the standard programming languages. Implementations of quicksort are very complex but it is very efficient and the fastest sorting algorithms in practice.

Algorithm:

Quicksort (A, p, r)

1. if p < r
2. then q partition (A, p, r)
3. Quicksort (A, p, q-1)
4. Quicksort (A, q+1, r)

To sort an entire array A, the initial call is Quick sort (A, 1, length [A]) where length of array is N.

Partition of array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray A[p...r] in place.

Partition (A, p, r)

1. x = A[r]
2. i = p-1
3. for j= p to r-1 do
4. if A[j] <=x
5. i = i+1
6. Exchange A[i] and A[j]
7. Exchange A[i+1] and A[r]
8. Return i+1

Performance Indices

Worst case performance	$O(N^2)$
Best case performance	O(NlogN)
Average case performance	O(NlogN)

The main problem in quicksort is to choose a good pivot element. Poor choices of pivots may result in slower $O(N^2)$ performance, further if at each step the middle element is chosen as the pivot then the algorithm works in O(NlogN). A quick sort algorithm gives worse performance when data is already in sorted order. This problem is solved by randomized quick sort. Randomized quick sort algorithm selects randomly any of the elements as pivot element.

2.2.7 Tree sort

K. E. Iversion (1962) identified a sorting algorithm based on the binary search tree (BST). Firstly it builds BST from the data elements to be sorted then traverses the tree in inorder traversal so that the data elements come out in sorted order. Several other sorting approaches would have to store the elements to a temporary data structure, whereas in a tree sort the act of loading the input into a data structure is actually, sorting it.

Tree sort takes NlogN running time to sort N numbers as each element is inserted in binary search tree in logN time.

Algorithm:

Tree_Sort(A, N)

1. Repeat for i = 0 to N do
2. Insert A[i] in binary search tree.
3. Traverse the tree by inorder traversal.

Performance Indices

Best case performance	O(NlogN)
Average case performance	O(NlogN)
Worst case performance	O(NlogN)

2.2.8 Heap sort

J. W. J. Williams (1964) invented Heap sort, a much more efficient version of selection sort. It also works similar to selection sort. First, determine the largest element of the list. After finding largest element, placed it at the end of the list, then continuing with the rest of the items in list. This task is efficiently accomplished by using a data structure called a heap **(Cormen *et al.*, 2001)**.

First, heap is created by using N elements to be sorted; the root node is guaranteed to be the largest (or smallest) element. It is removed and placed at the end of the list and heap is rearranged so the largest element moves to the root. This process is repeated until there are no elements left in heap.

Finding of an element using heap takes only O(logN) time instead of a linear search takes O(N) time in selection sort. This allows Heap sort to run in O(NlogN) time.

Algorithm:

Heap_sort(A,N)

1. Repeat for i = 0 to N do
2. Insert A[i] in heap
3. Repeat for i = N down to 1
4. Swap A[i] and A[0].
5. Apply Min_heapify on the tree with A[i-1] elements.

Performance Indices

Worst case performance	O(NlogN)
Best case performance	O(NlogN)
Average case performance	O(NlogN)

2.2.9 Comb sort

Wlodzimierz Dobosiewicz (1980) designed a relatively simple sorting algorithm known as Comb sort that improves the bubble sort. The basic idea is to eliminate turtles, or small values near the end of the list, since in a bubble sort these slow down the sorting tremendously **(Abbaslo, 2013)**.

Algorithm:

Comb_sort(a,n)

1. gap =n;
2. While gap >1 and swap !=0
3. Repeat for i=0 to n
4. If a[i]>a[i+gap] then swap them
5. gap =gap/1.3

Performance Indices

Worst case performance	$O(N^2)$
Best case performance	O(N)
Average case performance	$O(N^2)$

2.2.10 Gnome sort

Hamid Sarbazi-Azad (2000) a Professor of Computer Engineering at Sharif University of Technology, discovered Gnome sort which is also known as stupid sort. It is a sorting algorithm which is similar to insertion sort, except that moving an element to its proper place is accomplished by a series of swaps, as in bubble sort. It is simple, requires no nested loops. The running time is $O(N^2)$, but if the list is initially almost sorted it tends towards O(N).Itplaces the items in order by comparing the current item with its previous item. If they are in order- then move to the next item (or stop if the end is reached) else if they are out of order- then swaps them and goes back to the previous item. If there is no previous item, move to the next item.

Algorithm:

Gnome_sort(A, N)

1. Repeat for I = 0 to N do
2. if A[i] > A[i-1]
3. theni++, move to next item.
4. Else
5. swap A[i] and A[i-1]
6. i--, move to previous item and if no item then move to next item.

Performance Indices

Worst case performance	$O(N^2)$
Best case performance	O(N)
Average case performance	$O(N^2)$

2.2.11 Pancake sort

Hal Sudborough (2008) invented the pancake sort algorithm with the objective of minimizing reversal operation. Name of this sorting approach came from its working of sorting the number in the similar way as a cook makes pancake. Every time, he found the index of maximum element and then swaps this maximum element with very first element of the array. After it, Swap first element with last element and reduce the size of array by 1.

Pancake sort is a variation of the sorting approach in which the only allowed operation is to reverse the elements of some prefix of the sequence. Unlike a traditional sorting algorithm, which attempts to sort with the fewest comparisons possible, the goal is to sort the sequence in as few reversals as possible.

Algorithm:

Pancake_sort(a,n)

1. Find index of the maximum element in a[0…n]. Let the index be 'mi'.
2. swap(a[0], a[mi])
3. swap(a[0],a[n-1])
4. n = n-1
5. if(n>1)
6. pancake_sort(a,n)

Performance Indices

Best case performance	$O(N^2)$
Average case performance	$O(N^2)$
Worst case performance	$O(N^2)$

2.2.12 Sleep sort

In general, sleep sort works by starting a separate task for each item to be sorted, where each task sleeps for an interval corresponding to the item's value, after the interval value of item printed. Items are then collected sequentially in time. What it does is that sleep sort spawns off one process for each argument. Each process waits for n seconds, and then prints out n, meaning it takes 1 second to print out "1", 2 seconds to print out "2", 100 seconds to print out "100". This means that for the most part, the numbers are printed out in the order of their size, thus "sorting" the arguments in increasing order.

The complexity of this algorithm in a perfect world is O(max(argument)), as it takes max(argument)seconds to print the biggest argument. In reality, the complexity is $O(N^2 + \max(\text{argument}))$, because maintaining multiple background processes relies on the operating system to manage the context switching.

Algorithm:

Sleep_sort(a, n)

1. Repeat for i=0 to n
2. Sleep(a[i])
3. Print a[i]

Performance Indices

Best case performance	O(max(argument))
Average case performance	O(max(argument))
Worst case performance	O(max(argument))

2.2.13 Bucket sort

Bucket sort is a divide and conquer sorting algorithm that generalizes Counting sort by partitioning an array into a finite number of buckets. The idea of bucket sort is to divide the interval [0,1] into k equal-sized subinterval or buckets and then distribute the N elements into the buckets. Each bucket is then sorted individually, either using a different

sorting algorithm, or by recursively applying the bucket sort **(Cormen *et al.*, 2001)**. Bucket sort is a linear time sorting algorithm i.e. it sort the elements of a list in linear time. Bucket sort takes O(N+K) time on average to sort elements of a given list.

Algorithm:

Bucket_Sort (A, N)

1. Repeat for i= 1 to n do
2. insert A[i] into list B[NA[i]]
3. for i = 0 to N-1 do
4. Sort lists B with insertion sort
5. Concatenate all sorted buckets B[0], B[1],...B[N-1] together in order.

Performance Indices

Worst case performance	$O(N^2)$
Best case performance	O(N+K)
Average case performance	O(N+K)

2.2.14 Pigeonhole sort

Pigeonhole sort is also known as count sort. In this approach for sorting, elements are partitioned into different empty place called as pigeonholes. The number of elements (N) and the number of possible key values (K) are same.

In this sorting approach, first an empty array (pigeonholes) of length K is created. Then each element is placed in the pigeonhole whose index corresponds to its key value. After that the values from non-empty pigeonhole array are copied into the original array A. It takes O(N+ K) time to sort a given list of elements.

Algorithm:

Pigeonhole_sort(A, N, max,min)

1. Set j=0, NUM= max - min +1
2. Repeat for i = 0 to NUM-1 do
3. Pigeon[i] = 0
4. Repeat fori = 0 to N-1do
5. pigeon[A[i]-min] += 1
6. increment counter i++.
7. Repeat for i = 0 to NUM-1 do

8. Repeat while pigeon[i]-- > 0 do
9. A[j] = min + i.
10. Increment counter k++ and j++.
11. Return A.

Performance Indices

Best case performance	O(N + K)
Average case performance	O(N + K)
Worst case performance	O(N + K)

2.2.15 Odd even sort

Habermann (1972) introduced one more solution to sorting problem known as odd-even sort. In the sorting domain, an odd–even sort or odd–even transposition sort is a relatively simple sorting algorithm, developed for use on parallel processors. This is a comparison sort similar to bubble sort, with which it shares many characteristics.

Odd-Even sorting technique sorts numbers by exchanging two number at a time like in bubble sort.It functions by comparing all (odd, even)-indexed pairs of adjacent elements in the list and, if a pair is not in order then elements are exchanged. The next step repeats this for (even, odd)-indexed pairs (of adjacent elements). Then it alternates between (odd, even) and (even, odd) steps until the list is sorted.

Algorithm:

Odd_Even_sort (A, N)

1. Repeat for i= 0 to N-1 do.
2. If A[i] > A[i+2] or A[i+1] >A[i+3]
3. Swap (A[i], A[i+2]) and (A[i+1], A[i+3])
4. Repeat above two steps until items are sorted
5. Return A.

Performance Indices

Worst case performance	$O(N^2)$
Best case performance	O(N)
Average case performance	$O(N^2)$

2.2.16 Bogo sort

In computer science, bogo sort is a particularly ineffective sorting algorithm. Its only use is for educational purposes and to contrast it with other practical sorting algorithms. When bogo sort is used to sort a deck of cards or numbers, first it check whether the deck or numbersare already in order or not, and if they are not, throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted. And in case of numbers, find next permutation of the numbersuntil they are sorted.

Algorithm:

Bogo_sort(a,N)

1. while not in order(a) do
2. Randomly permute a[1 . . . N]

Performance Indices

Worst case performance	O(N!)
Best case performance	O(N)
Average case performance	O(N!)

2.2.17 Bozo sort

Bozo sort, based on random numbers is similar to bogo sort. In this sorting algorithm, firstly list is checked to see whetherit is in sorted order or not. If not, it takes two items at random and swaps them, then checks to see if the list is sorted. The worst case and average case time for bozo sort is infinite.

Algorithm:

Bozosort (A, N)

1. while !ordered(array)
2. swap(random(A), random(A))
3. return

Performance Indices

Best case performance	O(N)
Average case performance	Infinite
Worst case performance	Infinite

2.2.18 Permutation sort

Permutation sort is based on principle of permutation. Firstly, list is checked to see whether it is in order or not. If not, find next permutation and check again.

Algorithm:

Permutation_sort(A, N)

1. while notInOrder(A) do
2.   nextPermutation()
3. retrun

Performance Indices:

Worst case performance	O(N!)
Average case performance	O(N!)
Best case performance	O(N)

2.2.19 Stooge sort

Howard *et al.*proposed a new sorting technique called stooge sort, a recursive sorting algorithm with a time complexity of $O(N^3)$. The running time of the algorithm is slower as compared to efficient sorting algorithms, and is even slower than Bubble sort, a fairly inefficient and simple sort. Stooge sort algorithm sorts first 2/3 elements ofarray, then last 2/3 elements and then again first 2/3 elements. Repeat this process until the array is sorted.

Algorithm:

Stooge_sort(A, N)

1. i=0 and j=N-1
2. if a[j] < a[i]
3. exchange A[i] and A[j]
4. if j – i > 1
5.   t := (j – i + 1)/3
6.   Stooge_sort(A, i , j-t)
7.   Stooge_sort(A, i+t, j)
8.   Stooge_sort(A, i , j-t)

Performance Indices

Worst case performance	$O(N^3)$

2.2.20 Taco sort

Michael Bernard created an algorithm whose performance is less efficient than bogo sort. It is a veryineffective sorting algorithm.It repeatedly swaps a random item by a random amount until a sorted permutation occurs. Taco sort does not guarantee whether it sort the list or not.

Algorithm:

Taco_sort(A, N)

1. while not in order(A) do
2. Exchange (random(A) , A(random(n))

Performance Indices

Best case performance	O(N)
Worst case performance	infinite
Average case performance	infinite

2.2.21 Strand sort

A sorting algorithm that works well when many items are already in order. First, create an empty sub list. Move the first item from the original list to the sub list. Now start from second element in the original list, if it is greater than the last item of the sub list, remove it from the original list and append it to the sub list. Merge the sub list into a final, sorted list. Repeat this process for rest of the items in original list.

Algorithm:

Strand_sort (A, N)

1. Repeat for i = 0 to N
2. Take first item A[0], say m in a sub list.
3. If A[i+1] > m
4. Remove A[i+1] from original list and add it to sub list in ascending order.
5. Parse the original list again, again taking out relatively sorted numbers in sublist in ascending order.
6. Merge the sub lists into sorted list and repeat steps 3 to 5 until original list is empty.

Performance Indices

Worst case performance $O(N^2)$

Best case performance $O(N)$

Average case performance $O(N^2)$

Strand sort is most useful for data which is stored in a linked list, due to the frequent insertions and removals of data. Using another data structure, such as an array, would greatly increase the running time and complexity of the algorithm due to lengthy insertions and deletions. Strand sort is also useful for data which already has large amounts of sorted data, because such data can be removed in a single strand.

2.2.22 Cocktail sort

Cocktail sort (also known as bidirectional bubble sort) unlike bubble sort orders the array in both directions. Each iteration of the algorithm consists of two phases. In the first one the lightest bubble ascends to the end of the array, in the second phase the heaviest bubble descends to the beginning of the array.

This way an imperfection of bubble sort, the problem of rabbits and turtles is mitigated. The problem of rabbits and turtles is a situation in bubble sort, when a heavy bubble is placed at the end of the array.

Algorithm:

Cocktail_sort(a,n)

1. Repeat for i=0 to n/2
2. Repeat for j=i to n-i-1
3. If a[j]<a[j+1] then swap them and j++
4. Repeat for j=n-i-2 while j>i
5. If a[n-j]>a[n-j+1] then swap them and j--
6. Return

Performance Indices

Worst case performance $O(N^2)$

Best case performance $O(N)$

Average case performance $O(N^2)$

2.2.23 Selection sort

Selection sort is an in-place comparison sort. It is a basic sorting algorithm found in many text books. It has $O(N^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list.It does no more than n swaps, and thus is useful where swapping is very expensive.

Algorithm:

Selection-Sort (A, n)

```
1.  N = length[A]
2.  for j = 1 to N– 1 do
3.          smallest = j
4.          for i = j + 1 to N do
5.                  if A[i] < A[smallest]
6.                          smallest = i
7.          Exchange A[j] and A[smallest].
```

Performance Indices

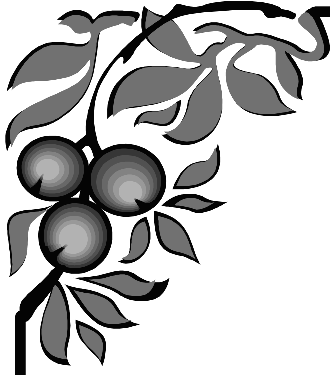
Worst case performance	$O(N^2)$
Best case performance	$O(N^2)$
Average case performance	$O(N^2)$

2.3 Chronological Order of Sorting Algorithms (Kadam, 2014)

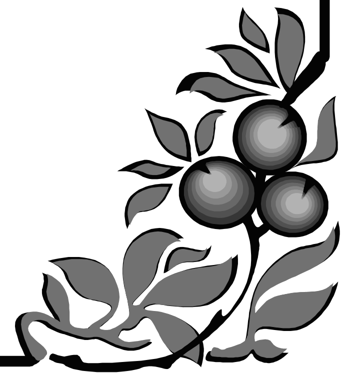
Sr. No.	Sorting Algorithm	Inventors Name	Invention Year
1	Radix Sort	Herman Hollerith	1880
2	Merge Sort	John Van Neumann	1945
3	Insertion Sort	John Mauchly	1946
4	Counting Sort	Harold H. Seward	1954
5	Digital Sorting	NA	1954
6	Key Sort	NA	1954
7	Distribution sort	H.Seward	1954
8	Bubble Sort	Iverson	1956
9	Address calculation sorting	Issac and singleton	1956
10	Comparison Sort	E.H.Friend	1956
11	Radix list sort	E.H.Friend	1956
12	Two way insertion sort	D.J.Wheeler	1957
13	Radix Sort (Modifed)	P. Hildebrandt, H Rising, J Scwartz	1959
14	New Merge Sort	B.K. Betz & W.C. Carter	1959
15	Shell Sort	Donald L Shell	1959
16	Cascade Merge Sort	R.L. GiIstad	1960
17	Poly-Phase Merge Sort	R.L. GiIstad	1960
18	Math sort	W. Feurzeig	1960
19	Quick Sort	CAR Hoare	1961
20	Patience Sort	C. L. Mallow	1962
21	Selection Sort	NA	1962
22	Topological Sort	Kahn	1962
23	Tournament Sort(tree sort)	K.E. Iversion	1962
24	Tree Sort(Modified)	K.E. Iversion	1962
25	Shuttle Sort	NA	1963
26	Bitonic Merge sort	K.E. Batcher	1964
27	Heap Sort	J.W.J Williams	1964
28	Batcher Odd-Even	Ken Batcher	1968

29	List sort	L.J.Woodrum & A.D. Woodall	1969
30	Improved Quick sort	Singleton	1969
31	Find:The Program	CAR Hoare	1971
32	Odd Even sort/Brick sort/ Oddeven Transport	Habermann	1972
33	Brick sort	Habermann	1972
34	Gyrating sort	R.M. Karp	1972
35	Binary Merge sort	F.K. Hawang & D.N. Deutsh	1973
36	Binary Merge sort	C. Christen	1978
37	Binary Merge sort	G.K. Manacher	1979
38	Comb Sort	Wlodzimierz Dobosiewicz	1980
39	Proxmap Sort	Thomas A. Standish	1980
40	Smooth Sort	Edsger Dijkstra	1981
41	B Sort	Wainright	1985
42	Unshuffle Sort	Art S. Kagel	1985
43	Qsort	Wainright	1987
44	American Flag Sort	NA	1993
45	Qsort	Benteley & Mcilroy	1993
46	Self Indexed sort	Yingxu Wang	1996
47	Splaysort	Moggat, Eddy & Petersson	1996
48	Flash Sort	Karl-Dietrich Neubert	1997
49	Introsort	David Musser	1997
50	Gnome Sort	Hamid Sarbazi Azad	2000
51	Tim sort	Tim Peters	2002
52	Spread sort	Steven J. Ross	2002
53	Bead Sort	Joshua J. Arulanandham, Cristian S. Calude and Michael J. Dinneen	2002
54	Burst Sort	Ranjan Sinha	2004
55	Libarary Sort/ Gapped Insertion	Michael A. Bender, Martín Farach Colton, and Miguel Mosteiro	2004
56	Cycle Sort	B.K. Haddon	2005
57	Pancake sorting	Hal Sudborough	2008

58	Qureshi Sort	M. A. Qureshi	2009
59	A New Friends Sort	S. Z. Iqbal, H. Gull & A. Muzaffar	2009
60	U Sort	Upendra Singh Aswal	2011
61	Counting Position Sort	Nitin Arora	2012
62	Novel Sorting Algorithm	R.Shriniwas & A. Raga Deepthi	2013
63	Bogo Sort	NA	NA
64	Bucket Sort	NA	NA
65	J Sort	Jason Morrison	NA
66	SS06 Sort	K.K. Sudharajan& S. Chakraborty	NA
67	Stooge Sort	Howard Fine and Howard	NA
68	Strand Sort	NA	NA
69	Trim Sort	NA	NA
70	Punch Card Sorter	A. S. C. Ross	NA



MATERIALS AND METHODS



This chapter elaborates proposed sorting approach and tools that have been employed in the development of the proposed sorting approach. First section presents materials and the software tools that have been used for the realization of the proposed approach. Section two focuses on the methodology used for proposed approach. Section three presents the proposed methodology block diagram.

3.1 Software Tools Used

Visual Studio: The main software that is used in formalizing the proposed work.Visual studio support many programming languages including C, C++, visual basic, C# and many others. I used C# programming language to implement my proposed work. To calculate the execution time of my proposed sorting approach, Stopwacth() function defined in System.Diagnostics namespace will be used.

Syntax:
```
Stopwatch mytimer = new Stopwatch();
mytimer.Start();
  //implementation code
mytimer.Stop();
```

Windows .NET Framework: Windows .NET Framework is required for Visual Studio.

3.2 Dataset Used

Numbers are generated randomly using random generator and stored in array.This generated numbers are input to the proposed sorting approach and other sorting algorithms. Each time different size input is generated and stored in array. I have referred a book titled “Design and Analysis of Algorithms” by **(Dave, 2008)** for experimental comparison of sorting methods.

3.3 Methodology

Sorting, a mandatory data structure operation is used in almost all activities performed by a computer. There are many sorting algorithm we have studied till now and it shows that every sorting technique has some merits and demerits. The algorithms we have studied till now generally fall in two categories. First is numerical sort and second is alphabetical or lexicographical sort. In numerical sorting, the data elements are arranged inascending or descending order and in alphabetical sort, the records are

arranged in alphabetical order. Insertion sort is well suited for small number of data while heap sort can work efficiently with large number of data. The limitations of sorting algorithms like bubble, insertion, selection and quick sort etc. have been reduced to some extent over the years but further improvements are still possible for getting a more efficient solution.

The result of work includes major reduction in the upper bound for worst case complexity of insertion sort, reduction in number of comparisons, swaps and execution time needed for sorting of numbers. The proposed sorting approach sorts numbers faster than the insertion sort and number of swaps required by proposed sorting approach are less than that of insertion sort. It needs few comparisons while insertion sort takes very large number of comparisons to sort numbers and it increases very fast as number of elements increases.

3.3.1 Proposed sorting approach (PSA)

The PSA is more efficient than insertion sort and is explained as follows:

Step 1: Input the total number of data elements, n, to be sorted.

Step 2: Read the number of data elements generated by random generator in array A.

Step 3: Compare first element with last element. If first element is greater than last element, swap them otherwise do nothing. Repeat this step for rest of the elements, comparing second element with second-last element, third with third-last and so on.

Step 4: If no swap occurs in Step 3, then apply Step 3 for first (n/2) elements and last (n/2) or (n/2+1) elements separately.

Step 5: Repeat for i = 1 to n-1

Compare A[i] with A[i-1], if A[i] is greater than or equal to A[i-1] then do nothing else Compare A[i] with A[0], if A[i] is smaller than or equal to A[0] then A[i] is inserted at position 0 else do

Step 5.1: Find middle index, m, of the list of elements from 0 to i.

Step 5.2: Find the correct position of A[i].

Compare A[i] with A[m]. If these two are equal then store position m+1.

Else if A[i] is smaller than A[m] then apply Step 5.1 for elements ranging from 0 to m, otherwise apply it for elements from m+1 to i.

Step 5.3: Repeat Step 5.2 till correct position of A[i] is found.

Step 6: Insert A[i] at position returned by Step 5.

3.4 Pseudo Code for Proposed Sorting Approach

Proposed_sort(A,N)

```
1.  Repeat for i = 0 to N/2 do
2.          If(A[i]>=A[n-i-1])
3.                  exchange A[i] and A[N-i-1].
4.  Repeat for i = 0 to N do
5.          If A[i] >= A[i-1]
6.                  Return
7.          else if(A[i] <= A[0])
8.                  Repeat for j = i down to 1 do
9.                          A[j]=A[j-1]
10.                         j= j -1
11.                 A[0]= A[i] and retrun.
12.         else call loc= locfind(A ,1,i,A[i]) //function using binary search.
13.                 Repeat for j = i down to loc+1 do
14.                         A[j] = A[j-1]
15.                         j = j-1
16.                 A[loc] = A[i] and return.
```

locfind(A, left, right, key)

```
1.  mid = left + (right - left) / 2
2.  if (left = right)
3.          return left
4.  if (key < numarray[mid])
5.          return locfind(numarray, left, mid, key)
6.  else if (key > numarray[mid])
7.          return locfind(numarray, mid + 1, right, key)
8.  else    return mid
```

3.5 Flowchart for Proposed Sorting Approach

Flowchart is divided in two parts. First part shows the flowchart for step 3 of proposed sorting approach. Second part shows the flowchart of complete proposed sorting approach.

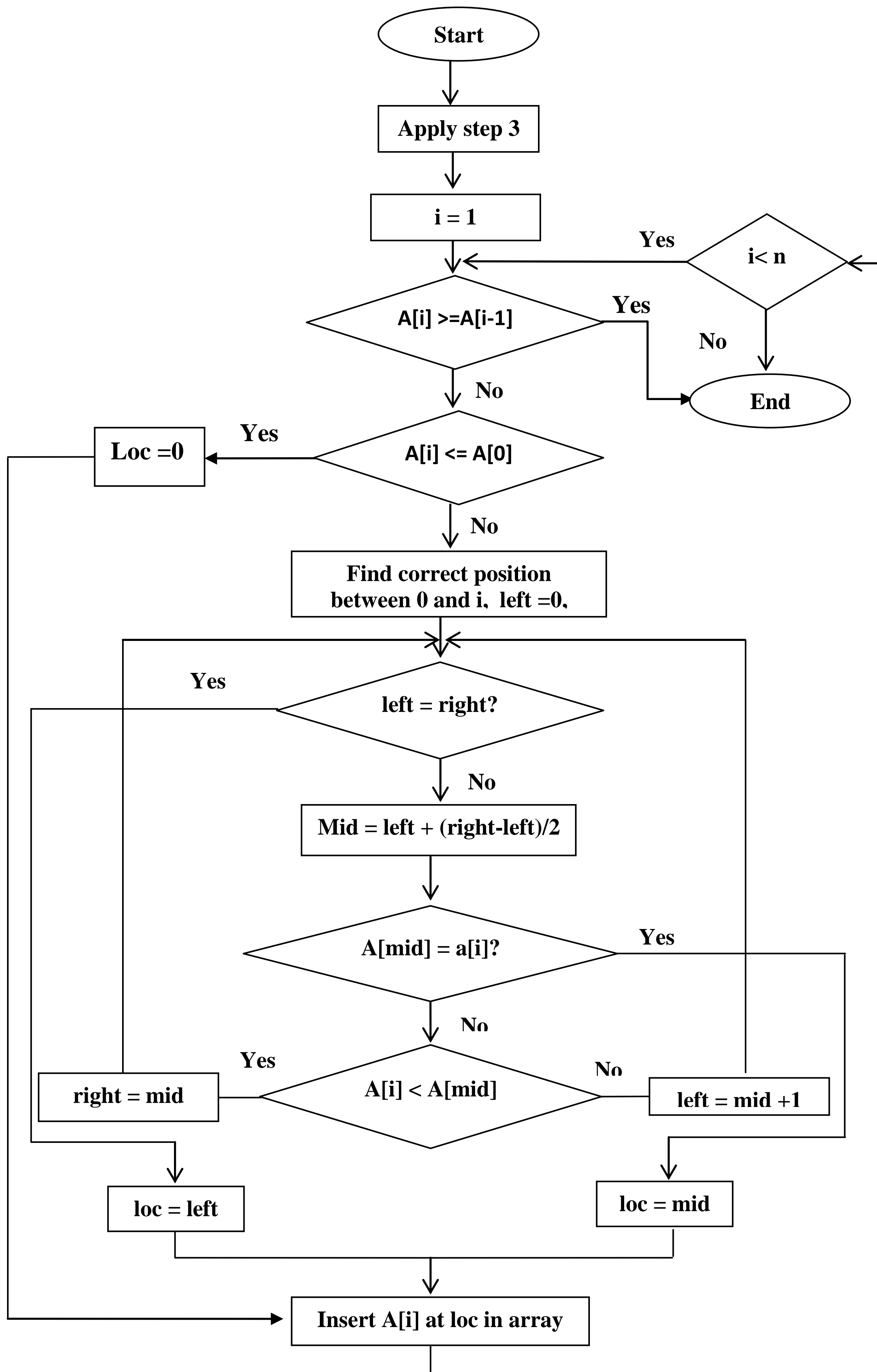


Figure 3.1: Flowchart for proposed sorting approach

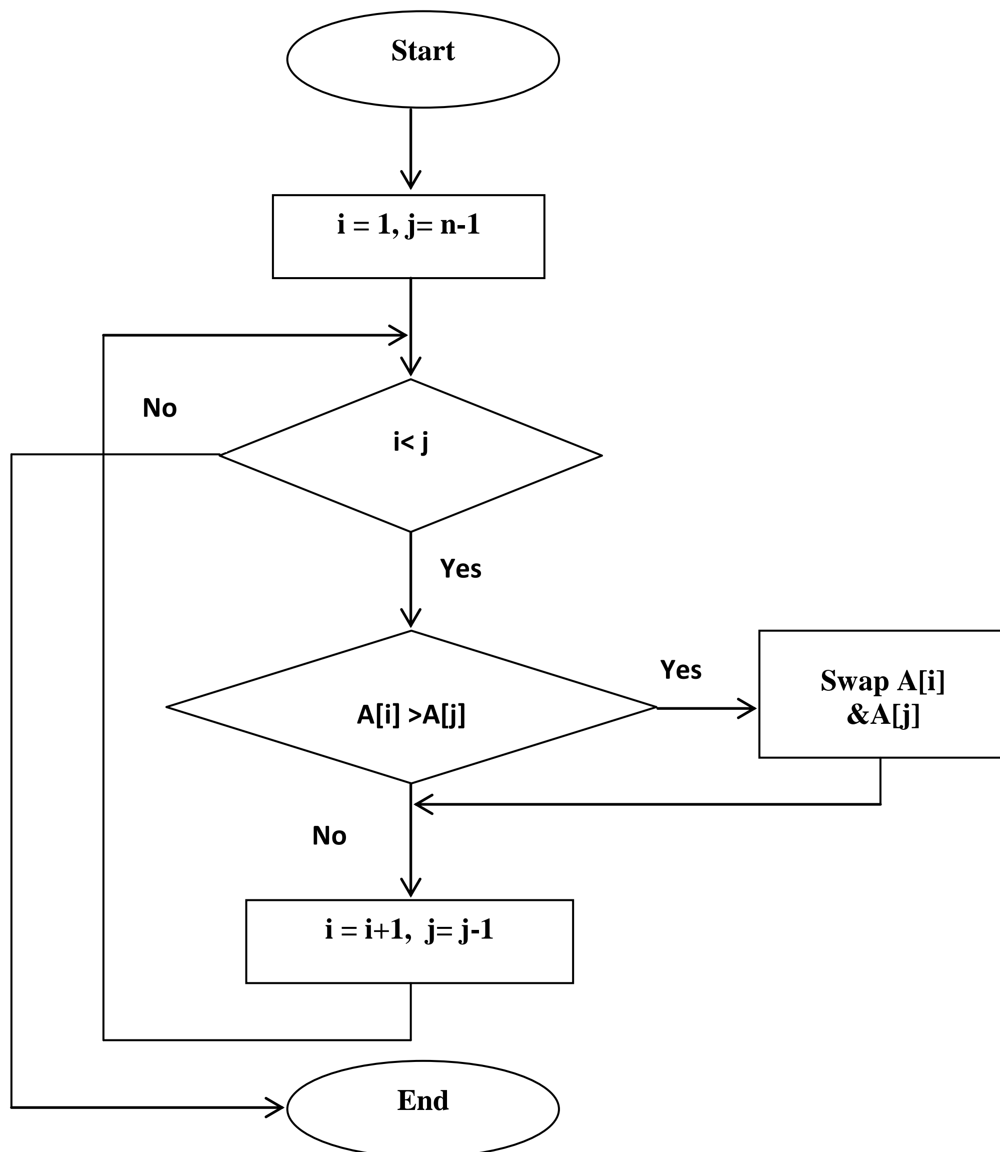
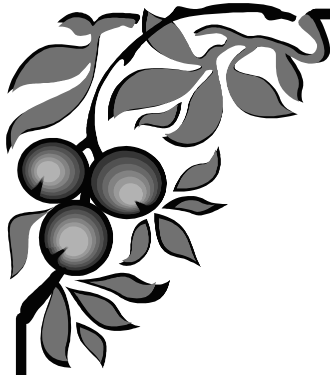
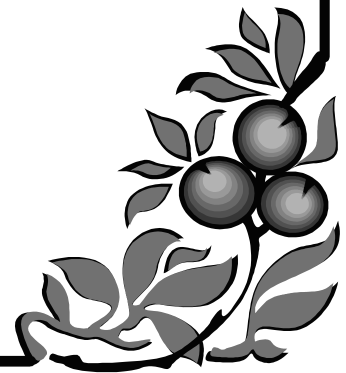


Figure 3.2: Flowchart for step 3 of proposed sorting approach



RESULTS AND DISCUSSION



This chapter presents the resultsthat have been obtained using proposed approach and shows the comparison with existing approach.This chapter is divided into three sections. In first section, describes the factors included in final result, dataset description and in second section results are included. Third section showed the comparison and discussion part.

The efficiency of an algorithm, in general, primarily relies upon three factors and sometimes it also considered how effectively the algorithm implemented under what circumstances and what type of machines used. The analysis should be organized associated with some of the below mentioned factors.

1. What machines used
2. Programming style and
3. Running environment

However these three primary factors are considered to ensure the efficiency of an algorithm such as:

1. Overall running time of an algorithm
2. Number of comparison and
3. Assignment operations performed

These are the computational resources needed for the algorithm to implement. The necessity of an analysis of the algorithm is to make sure that the algorithm is requiring fewer resources to solve a given problem.

4.1 Performance Factors

One of the main objectives of this thesis is to compare the sorting algorithm from the practical perspective. Keeping in mind the three factors that interested the whole period of study, they are defined and explained in the following sections.

4.1.1 Execution time

The overall running time or the time complexity of an algorithm states the amount of time taken by an algorithm regarding to the amount of input data supplied to the algorithm.

Usually the time taken includes the following three factors.

1. Number of memory accesses performed
2. Number of comparisons between integers and
3. Number of times some inner loop is executed.

Likewise there are other factors not related to algorithms can also affect the running time.

4.1.2 Operation performed

In general, there are the two different types of operations that are performed in most of the sorting algorithms. Proposed work counts following two operations for each sorting algorithm.

4.1.2.1 Comparison operation

Let's say, to sort an array of integers the algorithm needs to compare each element in that array to find out the appropriate position in the final sorted array, this method is called as comparison operation.

It affects the overall running time of an algorithm, because of the fact that if the input size grows, the number of comparisons will also increase proportionally. Most of the sorting algorithms perform this operation to sort the elements of a list.

4.1.2.2 Assignment operation

In assignment operation, each time when the desired position found, the algorithm need to swap the element by using a temporary variable. This also affect the running time of an algorithm, however when compared to comparison operation, the influence of an assignment operation in the running time is comparatively less as that of comparison operation. The running time of insertion sort depends on the number of shifts or swaps operations performed to sort the elements.

4.2 Design and Implementation

To implement the algorithm - I have chosen C# as a base language so as to provide the working environment for the same.

The design of the Algorithm includes the various sorting algorithms along with the collection of the desired data while executing the program. The designed program will be

capable of creating random numbers of integers. The size of the array elements will be varied from 500 to 7500 and 10000 to 50000 in the results and change automatically once each iterationof algorithm completed its execution with certain array size. The data collected during the experiment will be:

1. Running time of the algorithm
2. Number of comparisons and
3. Number of swap operations

All the collected data were noted down in a file after each algorithm completed its execution for further evaluation. The experiment was repeated for various test case scenarios such as- different types of input data, for instance arrays of numbers of different sizes. Data collected during the execution of the program later used for comparing various sorting algorithms and their efficiency against various experiments. All the experiments were conducted on the model machine described below as our test bed.

Test bed: Intel Core i5-3470 CPU @ 3.20 GHz, 4GB RAM, 360 GB HDD, Windows 8.1 Pro, 64-bit Operating System, Microsoft visual studio 8.

4.3 Different Case Scenarios

Proposed work focus on the following six cases in the whole study.

4.3.1 Case 1

In this experiment, we took randomly generated array with the array size varied from 500 to 7500 with an interval of 10000 for each of the sorting algorithm (values were integers) and the data collected during the experiment was the running time of each sorting algorithm.

4.3.2 Case 2

In this experiment, we took randomly generated array with the array size varied from 10000 to 50000 with an interval of 10000 for each of the sorting algorithm (values were integers) and the data collected during the experiment was the running time of each sorting algorithm.

4.3.3 Case 3

In this experiment the randomly generated array values used, were integers and the data collected during the experiment was the number of comparison operations performed

for each sorting algorithm. The array size varied from 500 to 7500 with an interval of 1000 for each of the sorting algorithm.

4.3.4 Case 4

In this experiment the randomly generated array with the array size varied from 10000 to 50000 with an interval of 10000 for each of the sorting algorithm (values were integers) were used and the data collected during the experiment was the number of comparison operations performed for each sorting algorithm.

4.3.5 Case 5

In this experiment, we took randomly generated array with the array size varied from 500 to 7500 with an interval of 1000 for each of the sorting algorithm (values were integers) and the data collected during the experiment was the number of swap performed for each sorting algorithm.

4.3.6 Case 6

In this experiment the randomly generated array with the array size varied from 10000 to 50000 with an interval of 10000 for each of the sorting algorithm (values were integers) were used and the data collected during the experiment was the number of swap performed for each sorting algorithm.

4.4 Experimental Results

The following figure shows the final result of proposed sorting approach along with output of other sorting algorithms like Insertion sort, Bubble sort and Selection sort. The figure contains the execution time in seconds, total number of comaprisons and total number of swaps performed in each sorting approach.

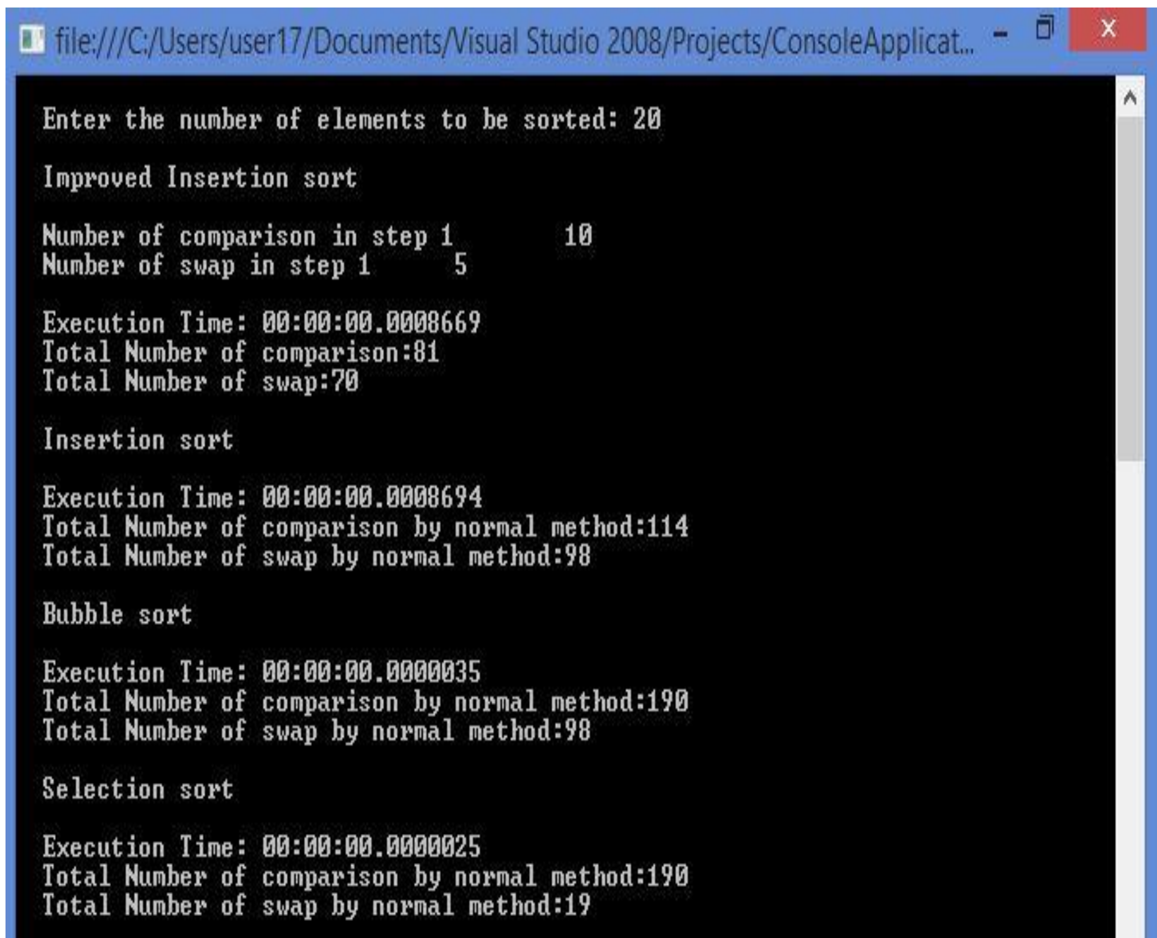


Figure 4.1: Snapshot of final result

The following sections lists the results obtained for each sorting algorithms i.e. proposed sorting approach, insertion sort, bubble sort and selection sort on a common data set. Results for each sorting algorithms are placed in different tables. The tables contains the execution time taken by each sorting algorithm, number of comparisons needed, number of swaps performed and sum of comparisons and swaps for each sorting techniques. These all rersults are noted down with respect to problem size or number of elements to be sorted in a list. Problem size is varied according to interval size dicussed before.

4.4.1 Proposed sorting approach

Table 4. 1: Execution time, comparisons and swaps taken by proposed sorting approach

Number of Elements	Execution Time(Seconds)	Number of Comparisons	Number of Swaps	Sum of Comparisons and Swaps
500	0.001046	4989	40055	45044
1500	0.00336	17350	376937	394287
2500	0.007642	30906	1048282	1079188
3500	0.013939	44977	2028943	2073920
4500	0.022095	59357	3449429	3508786
5500	0.032549	74093	5070285	5144378
6500	0.043012	89200	6959990	7049190
7500	0.055613	104450	9399316	9503766
10000	0.096206	143312	16739713	16883025
20000	0.37545	305705	66669318	66975023
30000	0.833399	475082	1.5E+08	1.5E+08
40000	1.468213	647539	2.67E+08	2.67E+08
50000	2.311659	824178	4.15E+08	4.16E+08

4.4.2 Insertion sort

Table 4. 2: Execution time, comparisons and swaps taken by insertion sort

Number of Elements	Execution Time(Seconds)	Number of Comparisons	Number of Swaps	Sum of Comparisons and Swaps
500	0.000963	60247	59751	119998
1500	0.008241	570450	568959	1139409
2500	0.023409	1550013	1547522	3097535
3500	0.042998	3073149	3069655	6142804
4500	0.068119	5093652	5089164	10182816
5500	0.103559	7502139	7496646	14998785
6500	0.141172	10610825	10604334	21215159
7500	0.18756	14039898	14032405	28072303
10000	0.334929	25284754	25274762	50559516
20000	1.349471	99893663	99873675	2E+08
30000	3.050421	2.23E+08	2.23E+08	4.47E+08
40000	5.437587	3.98E+08	3.98E+08	7.96E+08
50000	8.442214	6.23E+08	6.23E+08	1.25E+09

4.4.3 Bubble sort

Table 4. 3: Execution time, comparisons and swaps taken by bubble sort

Number of Elements	Execution Time(Seconds)	Number of Comparisons	Number of Swaps	Sum of Comparisons and Swaps
500	0.001573	124750	59751	184501
1500	0.014356	1124250	568959	1693209
2500	0.039811	3123750	1547522	4671272
3500	0.077411	6123250	3069655	9192905
4500	0.130329	10122750	5089164	15211914
5500	0.197144	15122250	7496646	22618896
6500	0.274245	21121750	10604334	31726084
7500	0.368083	28121250	14032405	42153655
10000	0.682073	49995000	25274762	75269762
20000	2.718656	2E+08	99873675	3E+08
30000	6.140273	4.5E+08	2.23E+08	6.73E+08
40000	10.8093	8E+08	3.98E+08	1.2E+09
50000	16.9997	1.25E+09	6.23E+08	1.87E+09

4.4.4 Selection sort

Table 4. 4: Execution time, comparisons and swaps taken by selection sort

Number of Elements	Execution Time(Seconds)	Number of Comparisons	Number of Swaps	Sum of Comparisons and Swaps
500	0.000844	124750	499	125249
1500	0.006549	1124250	1499	1125749
2500	0.018152	3123750	2499	3126249
3500	0.035595	6123250	3499	6126749
4500	0.05852	10122750	4499	10127249
5500	0.087217	15122250	5499	15127749
6500	0.123979	21121750	6499	21128249
7500	0.16563	28121250	7499	28128749
10000	0.292614	49995000	9999	50004999
20000	1.176331	2E+08	19999	2E+08
30000	2.586666	4.5E+08	29999	4.5E+08
40000	4.606029	8E+08	39999	8E+08
50000	7.187063	1.25E+09	49999	1.25E+09

Table 4. 5: Number of comparisons needed by propsed, insertion, bubble and selection sort

Problem Size(N)	Proposed Approach	Insertion Sort	Bubble Sort	Selection Sort
4	7	3	6	6
8	26	16	28	28
16	70	83	120	120
30	156	285	435	435
40	236	406	780	780
50	285	654	1225	1225
60	392	982	1770	1770
70	457	1309	2415	2415
80	565	1731	3160	3160
90	661	2245	4005	4005
100	717	2405	4950	4950

Table 4. 6: Number of swaps needed by propsed, insertion, bubble and selection sort

Problem Size(N)	Proposed Approach	Insertion Sort	Bubble Sort	Selection Sort
4	3	3	3	3
8	13	13	13	7
16	47	65	65	15
30	162	210	210	31
40	280	312	312	39
50	386	746	746	49
60	670	878	878	59
70	861	1123	1123	69
80	989	1673	1673	79
90	1334	1720	1720	89
100	1687	2670	2670	99

4.5 Experimental Comparison of Sorting Methods

The pattern is taken to match my results with experimental comparison of sorting procedures described by **(Dave, 2008)**. Some of the sorting methods were tested on the computer discussed in section 4.2 with varying sizes of array, doubling the sizes from 4 to 65536. I timed each run and also computed the time divided by N, NlogN, N^2 and logN, in an effort to ascertain the performance and compare it in each case. The results in Tables 4.7- 4.16. shows time, time divided by N, time Divided by NlogN, time divided by N^2 and time Divided by logN for proposed sorting approach, insertion sort, bubble sort, selection sort and randomized quick sort respectively executed on my test bed and corresponding

tables presented in the book titled “Design and Analysis of Algorithms ” by **(Dave, 2008)** in are also included.

Table 4.7 shows the execution time with respect to problem size (number of elements to be sorted) for proposed sorting approach, insertion sort, bubble sort, selection sort and randomized quick sort. For each run, problem size gets doubled.

Table 4. 7: Execution time for sorting techniques

Problem Size(N)	PSA	IS	BS	SS	RQS
4	0.0074	0.0017	0.5718	0.5194	1.5024
8	0.6169	0.0022	0.5449	0.5103	1.5177
16	0.6306	0.0039	0.5616	0.5696	1.5833
32	0.6311	0.0114	0.5661	0.5245	1.7628
64	0.7167	0.0342	6.283	0.5616	1.7909
128	0.7577	0.1117	0.7788	0.6579	2.0457
256	0.9972	0.447	1.3974	1.0154	2.6005
512	1.6859	1.8872	4.0271	2.4545	3.7272
1024	3.9832	7.1471	14.642	8.1084	5.9052
2048	11.8538	28.411	61.5466	29.4607	10.5002
4096	39.9376	113.5833	246.9175	101.7477	19.4519
8192	147.5986	407.7521	1031.99	458.636	41.1128

Table 4.8 presents the execution time with respect to problem size (number of elements to be sorted) for insertion sort, bubble sort, selection sort and quick sort taken from the book titled “Design and Analysis of Algorithms ” by **(Dave, 2008)**.

Table 4. 8: Execution time (Dave, 2008)

Problem Size(N)	IS	BS	SS	QS
4	0.013	0.013	0.011	0.018
8	0.015	0.022	0.016	0.021
16	0.037	0.07	0.034	0.037
32	0.109	0.26	0.104	0.082
64	0.361	0.984	0.393	0.17
128	1.023	4.144	1.301	0.352
256	4.853	15.254	4.972	0.775
512	19.763	76.353	19.471	1.662
1024	74.036	243.894	76.942	3.801
2048	294.659	989.817	306.675	8.046
4096	1184.282	4081.95	1271.62	17.69
8192	4799.96	16237.81	5165.823	38.495

Table 4.9 gives the execution time/N with respect to problem size (number of elements to be sorted) for proposed sorting approach, insertion sort, bubble sort, selection sort and randomized quick sort.

Table 4. 9: Execution time/N for sorting techniques

Problem Size (N)	PSA	IS	BS	SS	RQS
4	0.00185	0.000425	0.14295	0.12985	0.3756
8	0.0771125	0.000275	0.0681125	0.0637875	0.1897125
16	0.0394125	0.00024375	0.0351	0.0356	0.09895625
32	0.019721875	0.00035625	0.017690625	0.016390625	0.0550875
64	0.011198438	0.000534375	0.098171875	0.008775	0.027982813
128	0.005919531	0.000872656	0.006084375	0.005139844	0.015982031
256	0.003895313	0.001746094	0.005458594	0.003966406	0.010158203
512	0.003292773	0.003685938	0.00786543	0.004793945	0.007279688
1024	0.003889844	0.00697959	0.014298828	0.007918359	0.005766797
2048	0.005787988	0.013872559	0.030052051	0.014385107	0.005127051
4096	0.009750391	0.027730298	0.060282593	0.024840747	0.004748999
8192	0.018017407	0.049774426	0.125975342	0.05598584	0.005018652

Table 4.10 shows the execution time/N with respect to problem size (number of elements to be sorted) for insertion sort, bubble sort, selection sort and quick sort adapted from the book titled "Design and Analysis of Algorithms " by **(Dave, 2008)**.

Table 4. 10: Execution time/N (Dave, 2008)

Problem Size(N)	IS	BS	SS	QS
4	0.00325	0.00325	0.00275	0.0045
8	0.001875	0.00275	0.002	0.002625
16	0.002313	0.004375	0.002125	0.002313
32	0.003406	0.008125	0.00325	0.002563
64	0.005641	0.015375	0.006141	0.002656
128	0.007992	0.032375	0.010164	0.00275
256	0.018957	0.059586	0.019422	0.003027
512	0.0386	0.149127	0.038029	0.003246
1024	0.072301	0.238178	0.075139	0.003712
2048	0.143876	0.483309	0.149744	0.003929
4096	0.289131	0.99657	0.310454	0.004319
8192	0.585933	1.982155	0.630594	0.004699

Table 4.11 presents the execution time/NlogN with respect to problem size (number of elements to be sorted) for proposed sorting approach, insertion sort, bubble sort, selection sort and randomized quick sort.

Table 4. 11: Execution time/NlogN for sorting techniques

Problem Size (N)	PSA	IS	BS	SS	RQS
4	0.003072783	0.00070591	0.23743481	0.215676182	0.6238581
8	0.085387393	0.00030451	0.07542161	0.070632496	0.21007043
16	0.032731373	0.00020243	0.02914992	0.02956516	0.08218139
32	0.01310293	0.000236687	0.0117534	0.010889696	0.03659934
64	0.006200067	0.000295859	0.05435332	0.00485832	0.01549282
128	0.00280918	0.000414129	0.00288741	0.00243917	0.00758445
256	0.001617494	0.00072505	0.00226663	0.001647015	0.0042181
512	0.001215373	0.001360491	0.00290315	0.00176946	0.00268696
1024	0.001292178	0.00231857	0.00474997	0.002630422	0.00191569
2048	0.001747935	0.004189422	0.00907552	0.004344208	0.00154834
4096	0.002699175	0.007676505	0.01668787	0.006876598	0.00131465
8192	0.004604041	0.012719005	0.03219085	0.014306226	0.00128243

Table 4.12 gives the Execution Time/NlogN with respect to problem size (number of elements to be sorted) for insertion sort, bubble sort, selection sort and quick sort taken from the book titled "Design and Analysis of Algorithms " by **(Dave, 2008)**.

Table 4. 12: Execution time/NlogN (Dave, 2008)

Problem Size(N)	IS	BS	SS	QS
4	0.005398	0.005398	0.004568	0.007474
8	0.002076	0.003045	0.002215	0.002907
16	0.00192	0.003633	0.001765	0.00192
32	0.002263	0.005398	0.002159	0.001702
64	0.003123	0.008512	0.0034	0.001471
128	0.003793	0.015364	0.004823	0.001305
256	0.007872	0.024743	0.008065	0.001257
512	0.014247	0.055043	0.014037	0.001198
1024	0.024018	0.079121	0.024961	0.001233
2048	0.04345	0.145956	0.045222	0.001186
4096	0.080039	0.275878	0.085942	0.001196
8192	0.149725	0.506506	0.161137	0.001201

Table 4.13 shows the execution time/N^2 with respect to problem size (number of elements to be sorted) for proposed sorting approach, insertion sort, bubble sort, selection sort and randomized quick sort.

Table 4. 13: Execution time/N^2 for sorting techniques

Problem Size (N)	PSA	IS	BS	SS	RQS
4	0.0004625	0.00010625	0.0357375	0.0324625	0.0939
8	0.009639063	0.000034375	0.008514063	0.007973438	0.0237141
16	0.002463281	1.52344E-05	0.00219375	0.002225	0.0061848
32	0.000616309	1.11328E-05	0.000552832	0.000512207	0.0017215
64	0.000174976	8.34961E-06	0.001533936	0.000137109	0.0004372
128	4.62463E-05	6.81763E-06	4.75342E-05	4.0155E-05	0.0001249
256	1.52161E-05	6.82068E-06	2.13226E-05	1.54938E-05	3.968E-05
512	6.4312E-06	7.1991E-06	1.53622E-05	9.36317E-06	1.422E-05
1024	3.79868E-06	6.81601E-06	1.39637E-05	7.73277E-06	5.632E-06
2048	2.82617E-06	6.77371E-06	1.46739E-05	7.02398E-06	2.503E-06
4096	2.38047E-06	6.77009E-06	1.47174E-05	6.06464E-06	1.159E-06
8192	2.19939E-06	6.07598E-06	1.53778E-05	6.83421E-06	6.126E-07

Table 4.14 gives the execution time/N^2 with respect to problem size (number of elements to be sorted) for insertion sort, bubble sort, selection sort and quick sort taken from the book titled "Design and Analysis of Algorithms " by **(Dave, 2008)**.

Table 4. 14: Execution time/N^2 (Dave, 2008)

Problem Size(N)	IS	BS	SS	QS
4	0.000813	0.000813	0.000688	0.001125
8	0.000234	0.000344	0.00025	0.000328
16	0.000145	0.000273	0.000133	0.000145
32	0.000106	0.000254	0.000102	8.01E-05
64	8.81E-05	0.00024	9.59E-05	4.15E-05
128	6.24E-05	0.000253	7.94E-05	2.15E-05
256	7.41E-05	0.000233	7.59E-05	1.18E-05
512	7.54E-05	0.000291	7.43E-05	6.34E-06
1024	7.06E-05	0.000233	7.34E-05	3.62E-06
2048	7.03E-05	0.000236	7.31E-05	1.92E-06
4096	7.06E-05	0.000243	7.58E-05	1.05E-06
8192	7.15E-05	0.000242	7.7E-05	5.74E-07

Table 4.15 presents the execution time/logN with respect to problem size (number of elements to be sorted) for proposed sorting approach, insertion sort, bubble sort, selection sort and randomized quick sort.

Table 4. 15: Execution time/logN for sorting techniques

Problem Size (N)	PSA	IS	BS	SS	RQS
4	0.012291	0.002824	0.949739	0.862705	2.495432
8	0.683099	0.002436	0.603373	0.56506	1.680563
16	0.523702	0.003239	0.466399	0.473043	1.314902
32	0.419294	0.007574	0.376109	0.34847	1.171179
64	0.396804	0.018935	3.478612	0.310932	0.99154
128	0.359575	0.053008	0.369588	0.312214	0.97081
256	0.414078	0.185613	0.580258	0.421636	1.079834
512	0.622271	0.696571	1.486415	0.905964	1.375721
1024	1.32319	2.374215	4.863967	2.693552	1.961665
2048	3.57977	8.579936	18.58667	8.896939	3.170992
4096	11.05582	31.44296	68.35352	28.16655	5.384818
8192	37.7163	104.1941	263.7074	117.1966	10.50567

Table 4.16 shows the execution time/logN with respect to problem size (number of elements to be sorted) for insertion sort, bubble sort, selection sort and quick sort adapted from the book titled “Design and Analysis of Algorithms ” by **(Dave, 2008)**.

Table 4. 16: Execution time/logN (Dave, 2008)

Problem Size(N)	IS	BS	SS	QS
4	0.021593	0.021593	0.018271	0.029897
8	0.01661	0.024361	0.017717	0.023253
16	0.030728	0.058134	0.028236	0.030728
32	0.072418	0.17274	0.069096	0.05448
64	0.199869	0.544796	0.217586	0.094121
128	0.485476	1.966581	0.617404	0.167046
256	2.015165	6.334086	2.064578	0.321812
512	7.294585	28.18213	7.186807	0.613449
1024	24.59423	81.01983	25.55958	1.262665
2048	88.98509	298.9183	92.61385	2.429839
4096	327.8416	1129.995	352.0192	4.897076
8192	1226.548	4149.295	1320.038	9.83674

4.6 Discussion

In the Figure 4.2-4.3, graphs show the comparison of execution time taken by proposed sorting approach with other sorting algorithms. In the graph, x-axis represents numbers of elements in the list to be sorted and y-axis represents the time taken by program for execution in seconds.

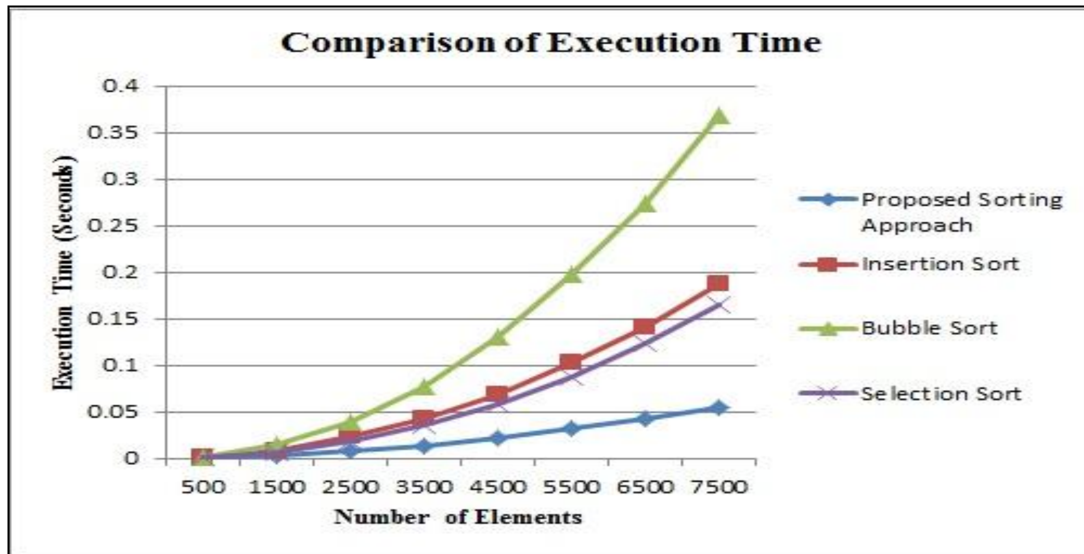


Figure 4.2: Execution time taken by proposed sorting approach, insertion, bubble and selection sort

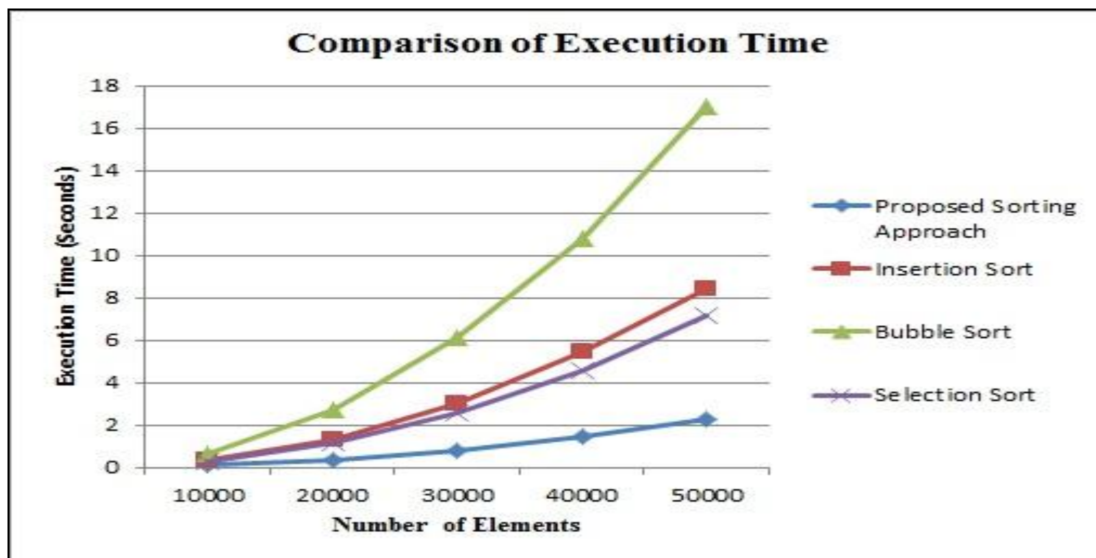


Figure 4.3: Execution time taken by proposed sorting approach, insertion, bubble and selection sort

In the Figures 4.2-4.3, graphs show the comparison of proposed sorting approach with insertion sort, bubble sort and selection sort. It is clear from the given graphs that time taken by proposed approach is less than that of insertion, bubble and selection sort.

In the Figure 4.4-4.6, graphs show the comparison of number of comparisons performed by proposed sorting approach with other sorting algorithms. In the graph, x-axis represents numbers of elements in the list to be sorted and y-axis represents the numbers of comparisons needed for each sorting methods.

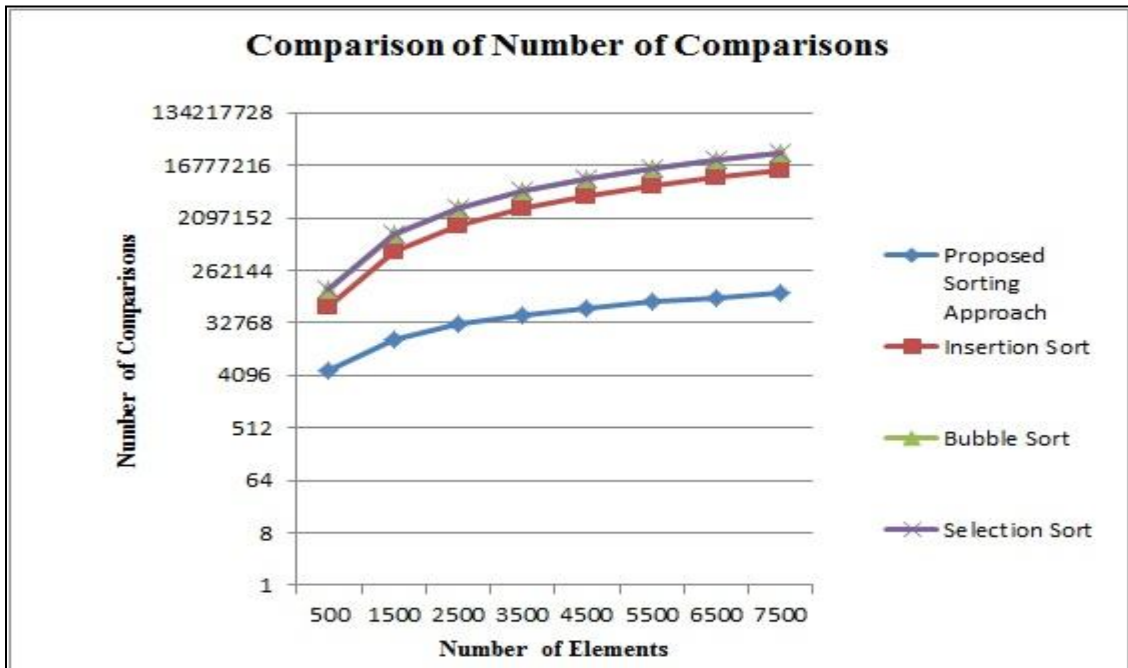


Figure 4.4: Number of comparisons needed by proposed sorting approach, insertion, bubble and selection sort

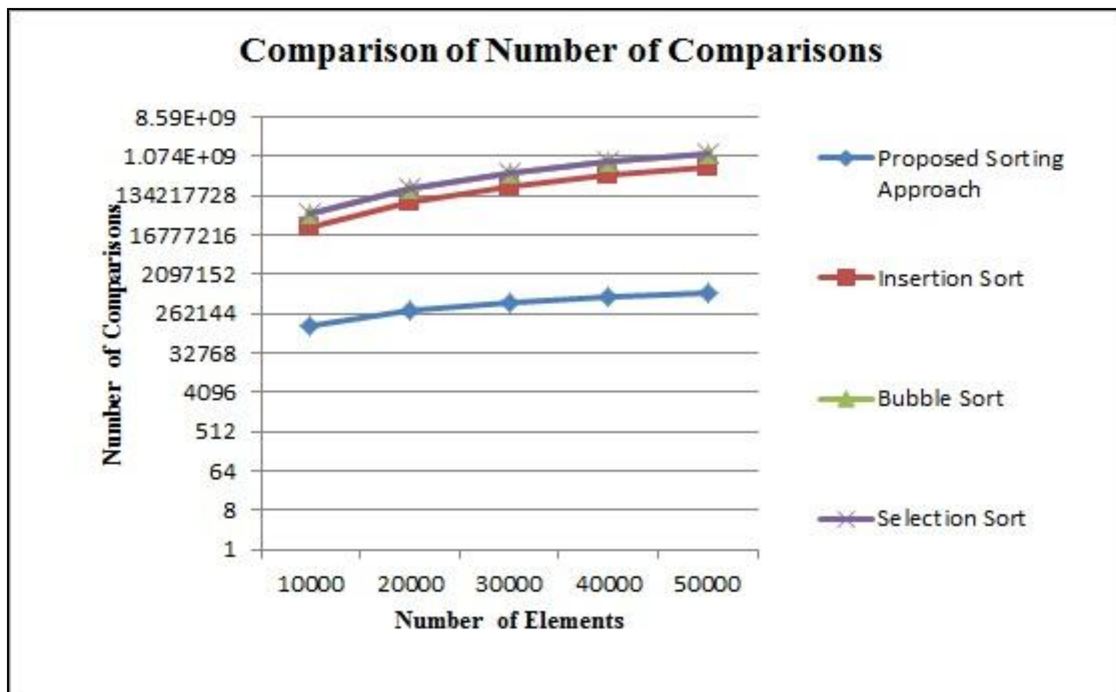


Figure 4.5: Number of comparisons needed by proposed sorting approach, insertion, bubble and selection sort

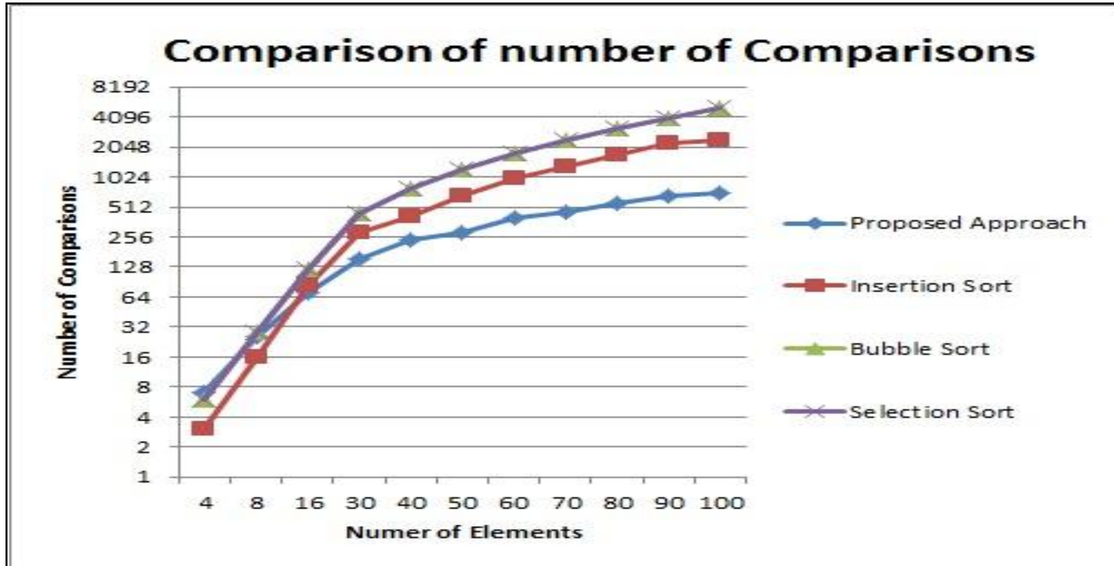


Figure 4.6: Number of comparisons needed by proposed sorting approach, insertion, bubble and selection sort

In the Figure 4.4-4.6, graphs show the comparison of proposed sorting approach with insertion sort, bubble sort and selection sort. It is clear from the given graphs that number of comparisons needed by proposed sorting approach is less than that of insertion sort, bubble sort and selection sort.

In the Figure 4.7-4.9, graphs show the comparison of number of swaps performed by proposed sorting approach with other sorting algorithms. In the graph, x-axis represents numbers of elements in the list to be sorted and y-axis represents the numbers of swaps required for each sorting methods.

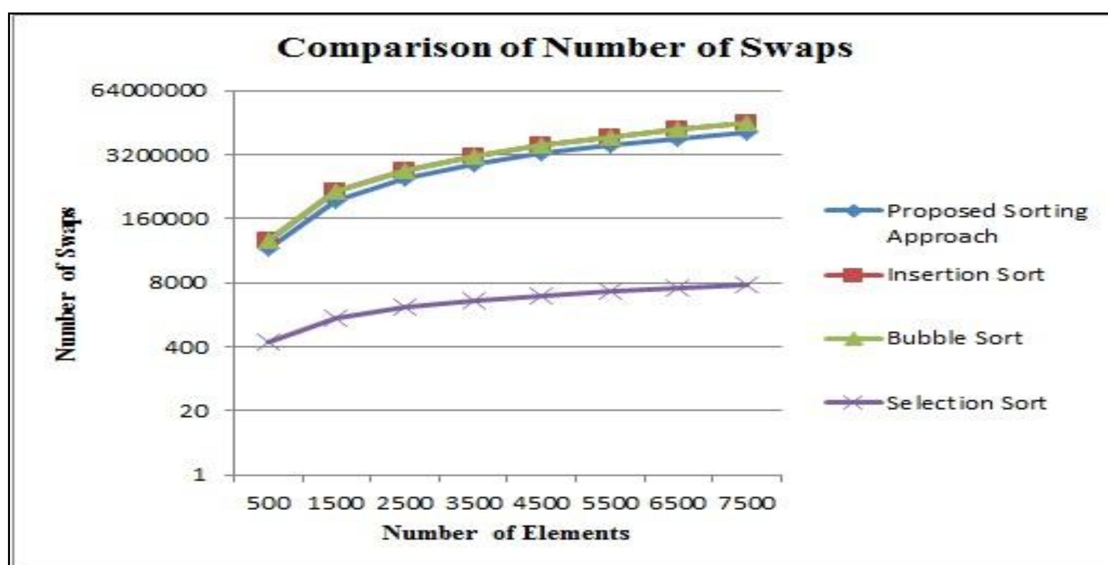


Figure 4.7: Number of swaps needed by proposed sorting approach, insertion, bubble and selection sort

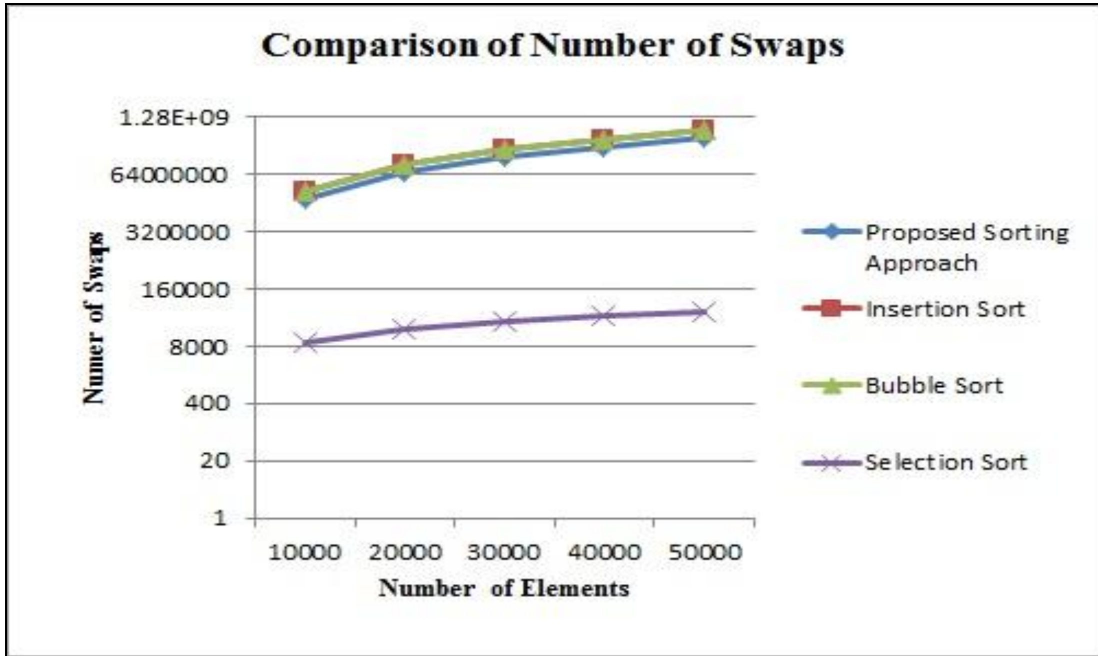


Figure 4.8: Number of swaps needed by proposed sorting approach, insertion, bubble and selection sort

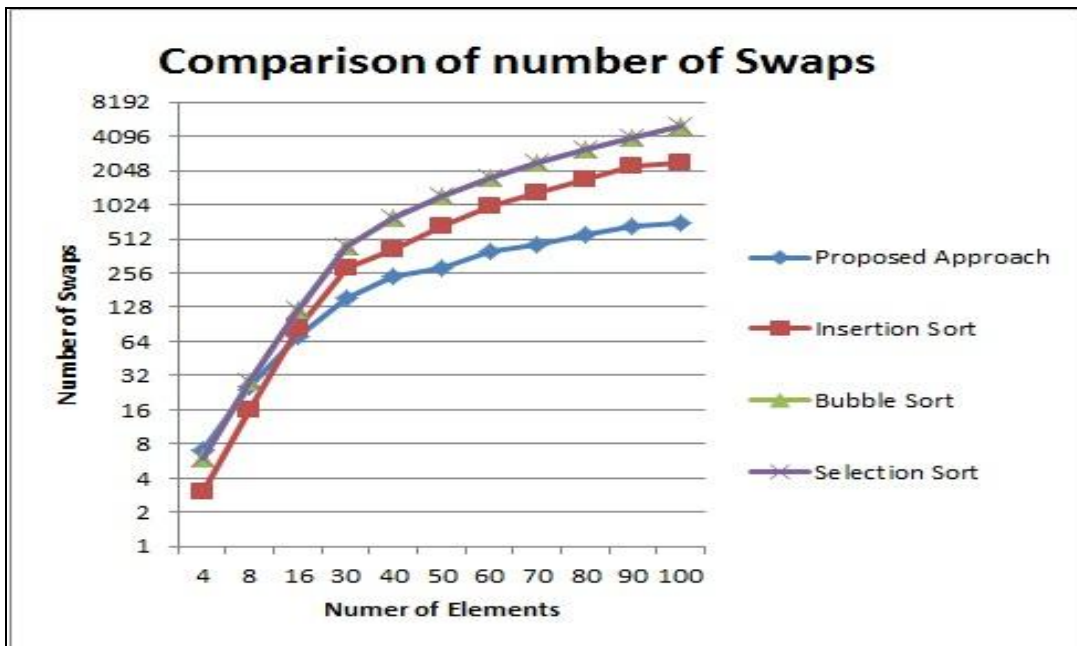


Figure 4.9: Number of swaps needed by proposed sorting approach, insertion, bubble and selection sort

In the Figures 4.7-4.9, graphs show the comparison of proposed sorting approach with insertion sort, bubble sort and selection sort. It is clear from the given graphs that number of swaps needed by proposed sorting approach is less than that of insertion sort, bubble sort and selection sort.

In the Figure 4.10-4.11, graphs show the sum of comparisons and swaps needed in different sorting methods. In the graph, x-axis represents numbers of elements in the list to be sorted and y-axis represents the sums of numbers of comparisons and swaps needed for each sorting methods.

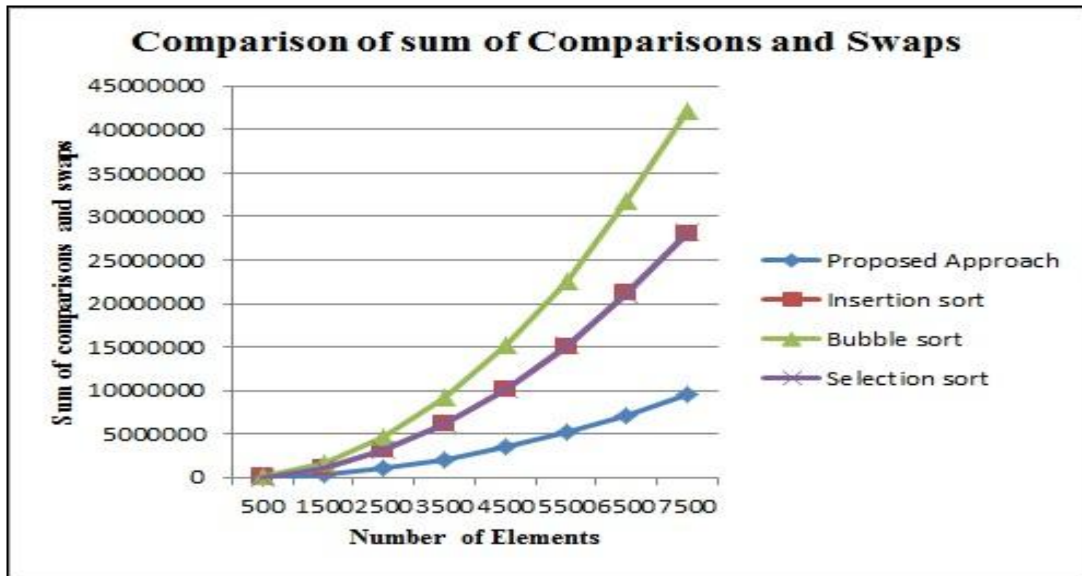


Figure 4.10: Sum of comparisons and swaps required by different sorting methods

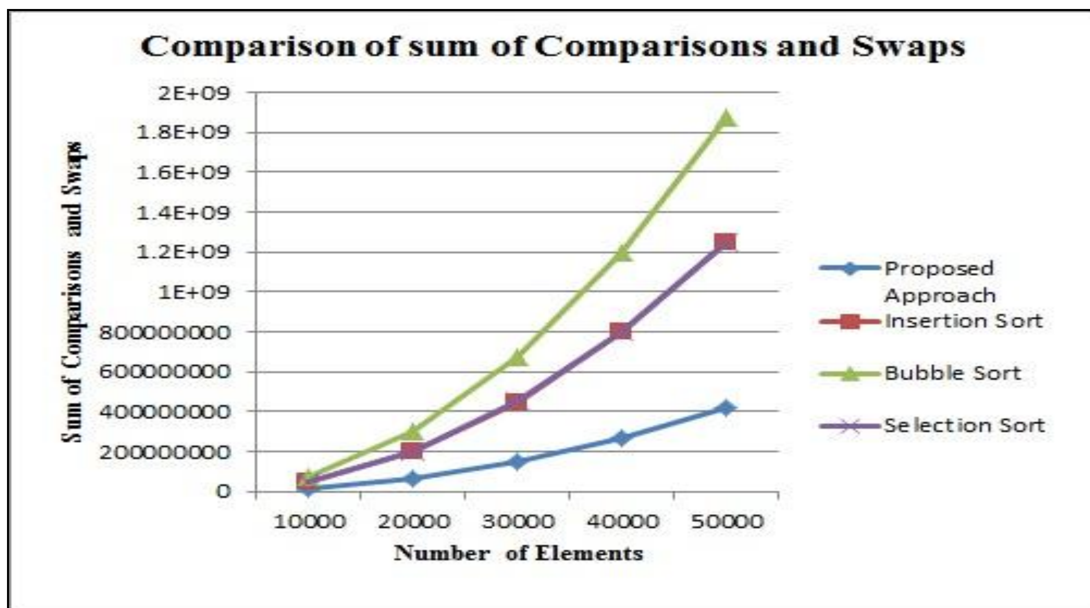


Figure 4.11: Sum of comparisons and swaps required by different sorting methods

In the Figure 4.10-4.11, graphs show the comparison of proposed sorting approach with insertion sort, bubble sort and selection sort. It is clear from the given graphs that sum of number of comparisons and swaps needed by proposed approach is less than that of insertion sort, bubble sort and selection sort.

The Figure 4.12 shows the execution time taken by different sorting algorithms.In the graph, x-axis represents numbers of elements (problem size say N) in the list to be sorted and y-axis represents the time taken by program for execution in milliseconds. The Figure 4.12 shows the comparison of proposed sorting approach (PSA) with insertion sort (IS), bubble sort (BS), selection sort (SS) and quick sort (QS). The results are taken from Test Bed defined in section 4.2 and from book titled "Design and Analysis of Algorithms " by **(Dave, 2008)**.

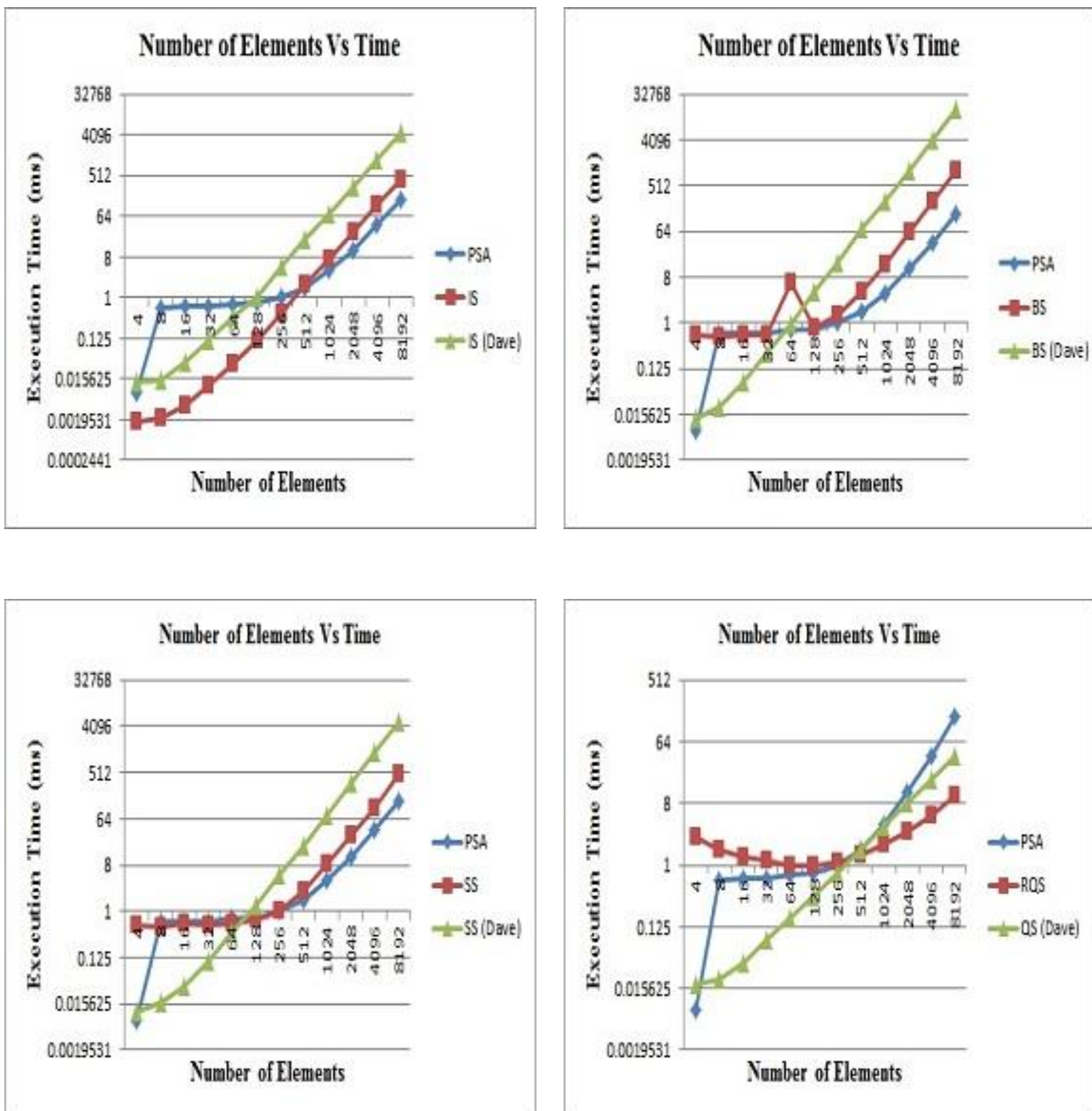


Figure 4.12 Comparison on the basis of execution time for sorting techniques

It is clear from the given graphs that when comparing time, PSA is better than the insertion, bubble and selection sort for problem size greater than 500. In comparison with QS, PSA performs better when problem size is less than 500.

The Figure 4.13 shows the execution time/ N for different sorting algorithms. In the graph, x-axis represents numbers of elements in the list to be sorted and y-axis represents the time divided by problem size. The Figure shows the comparison of proposed sorting approach (PSA) with insertion sort (IS), bubble sort (BS), selection sort (SS) and quick sort (QS). The results are taken from Test Bed defined in section 4.2 and from book titled "Design and Analysis of Algorithms " by **(Dave, 2008)**.

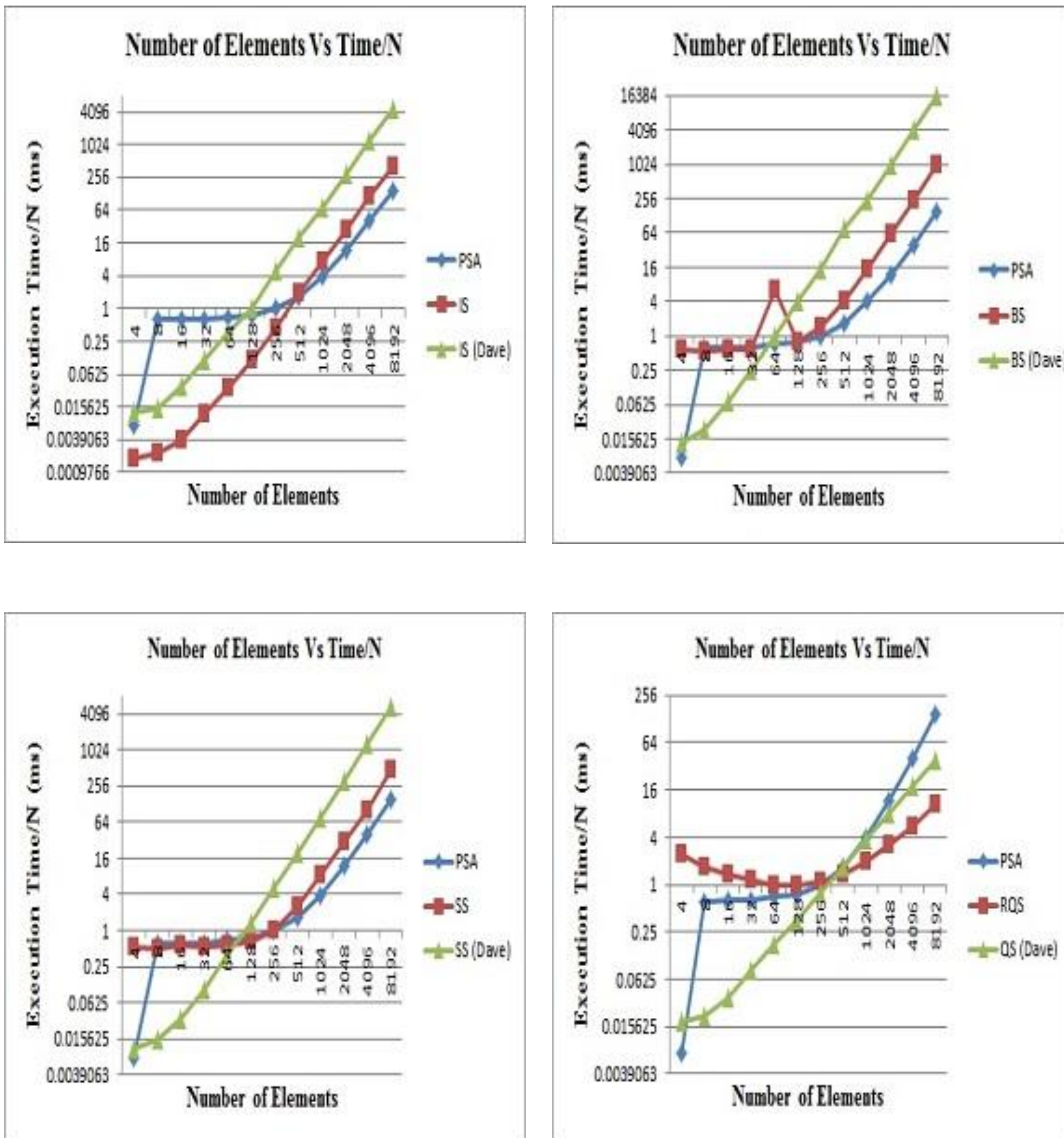


Figure 4.13: Comparison on the basis of time/N for sorting techniques

It is clear from the given graphs that when comparing time/N, PSA is better than the insertion, bubble and selection sort for problem size greater than 500. In comparison with QS, PSA performs better when problem size is less than 500.

In the Figure 4.14, graphs show the execution time/ NlogN taken by different sorting algorithms. In the graph, x-axis represents the numbers of elements and y-axis represents the time divided by (problem_size*log(problem size)). The Figure shows the comparison of proposed sorting approach (PSA) with insertion sort (IS), bubble sort (BS), selection sort (SS) and quick sort (QS). The results are taken from Test Bed defined in section 4.2 and from book titled “Design and Analysis of Algorithms ” by **(Dave, 2008)**.

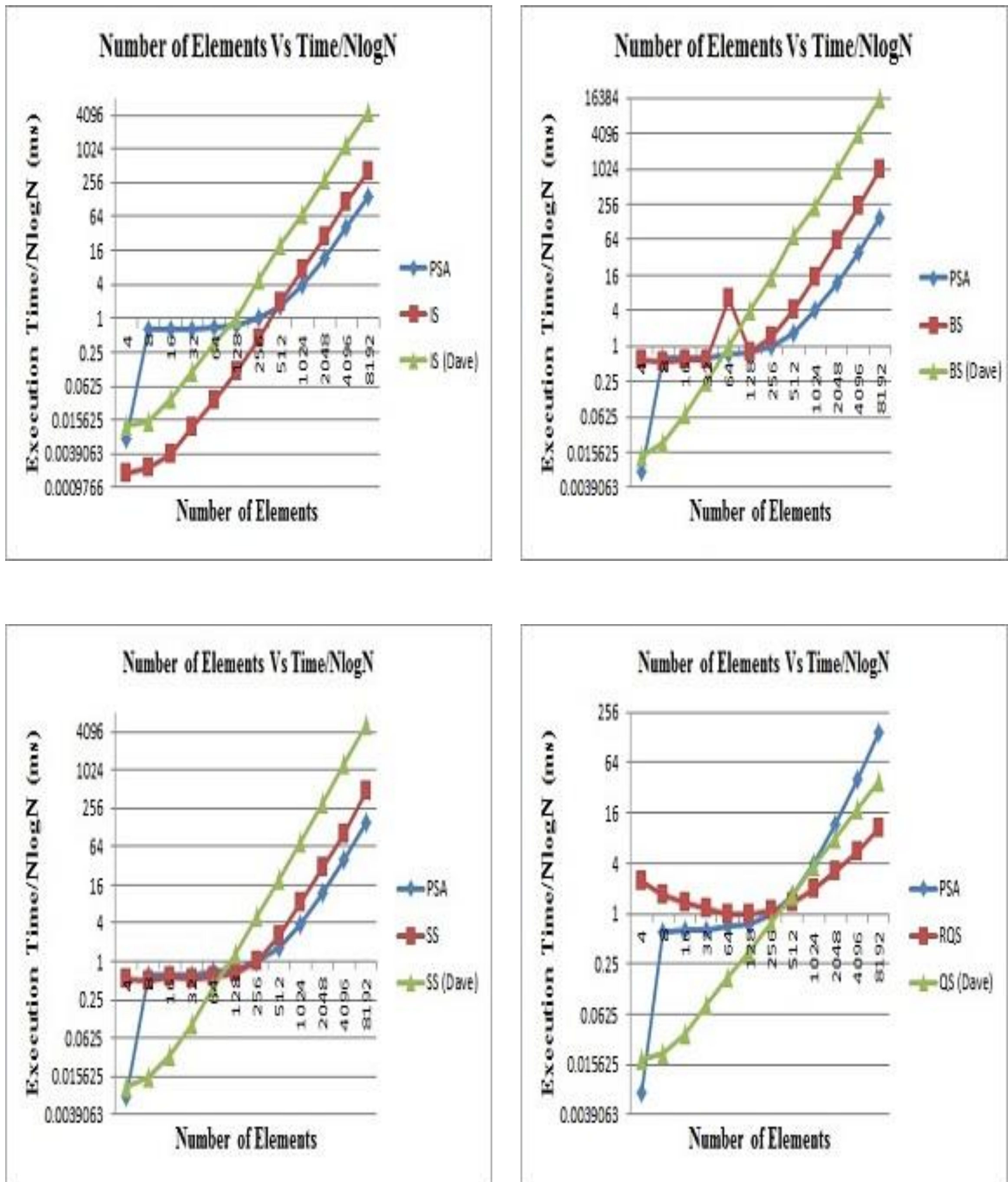


Figure 4.14: Comparison on the basis of time/NlogN for sorting techniques

It is clear from the given graphs that when comparing time/NlogN, PSA is better than the insertion, bubble and selection sort for problem size greater than 500. In comparison with QS, PSA performs better when problem size is less than 500.

The Figure 4.15 shows execution time/ N^2 for all sorting algorithms. In the graph, x-axis is numbers of elements in the list to be sorted and y-axis is the time divided by square of problem size.

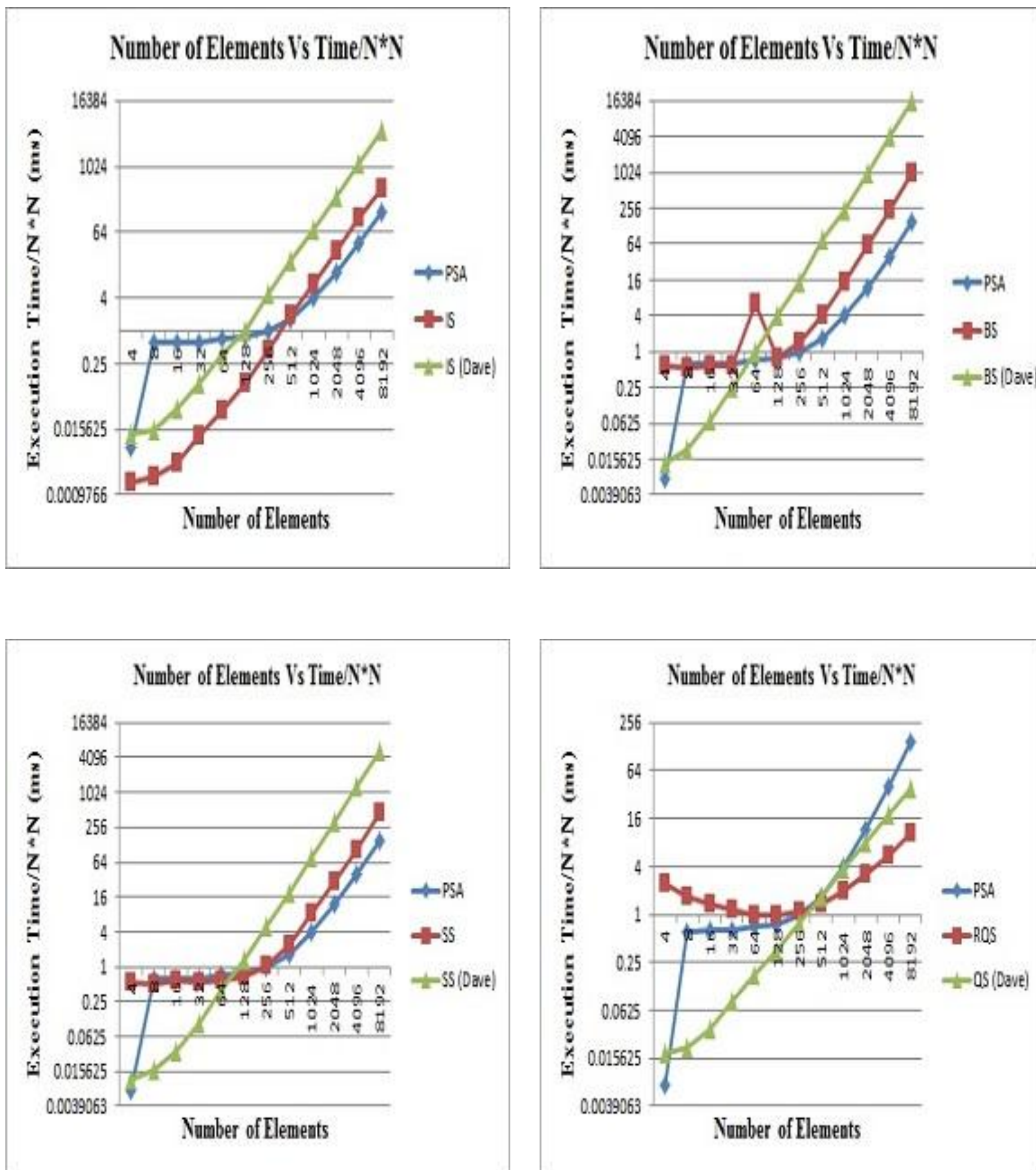


Figure 4.15: Comparison on the basis of time/N^2 for sorting techniques

The Figure 4.15 shows the comparison of proposed sorting approach (PSA) with insertion sort (IS), bubble sort (BS), selection sort (SS) and quick sort (QS). The results are taken from Test Bed defined in section 4.2 and from book titled "Design and Analysis of Algorithms " by **(Dave, 2008)**. It is clear from the given graphs that when comparing time/N^2, PSA is better than the insertion, bubble and selection sort for problem size greater than 500. In comparison with QS, PSA performs better when problem size is less than 500.

The Figure 4.16 shows the execution time/logN for all sorting algorithms. In the graph, x-axis represents the numbers of elements and y-axis represents the time divided by logarithm of problem size.

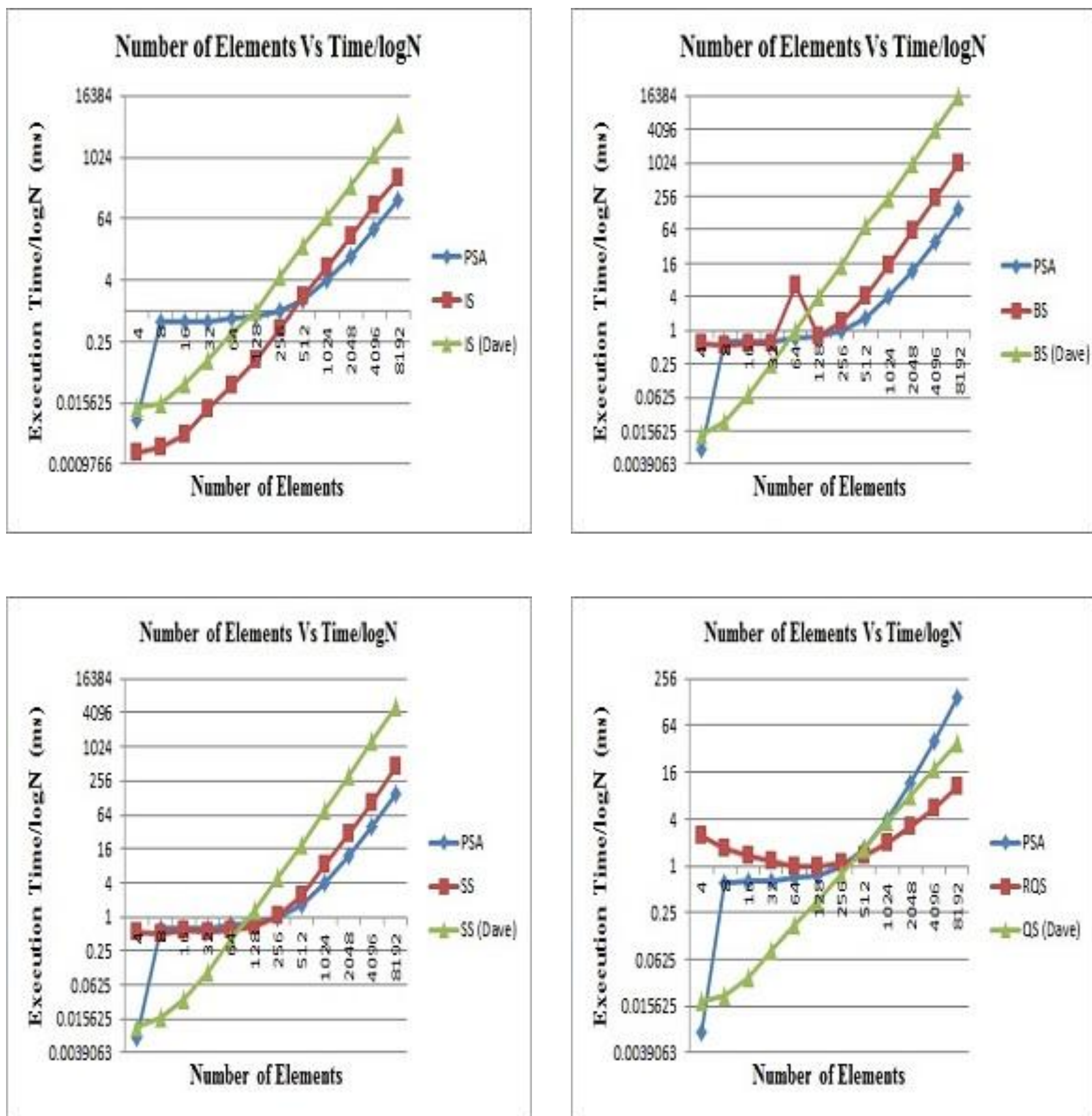


Figure 4.16: Comparison on the basis of time/logN for sorting techniques

The Figure 4.16 shows the comparison of proposed sorting approach (PSA) with insertion sort (IS), bubble sort (BS), selection sort (SS) and quick sort (QS). The results are taken from Test Bed defined in section 4.2 and from book titled “Design and Analysis of Algorithms ” by **(Dave, 2008)**. It is clear from the given graphs that when comparing time/logN, PSA is better than the insertion, bubble and selection sort for problem size greater than 500. In comparison with QS, PSA performs better when problem size is less than 500.

The Figure 4.2-4.11 shows that proposed sorting approach takes less number of comparisons and swaps than that of insertion, bubble and selection sort. The Figure 4.12-4.16 shows that proposed sorting approach takes more time than the insertion sort when problem size is small i.e. up to 500. When problem size is more than 500, proposed sorting approach takes less time than insertion sort. Hence proposed sorting approach is more fast and efficient than insertion, bubble and selection sort for sorting large elements but is slower than quick sort.

4.7 Cases of Failure

The proposed sorting approach takes more number of comparisons than the insertion sort in the following input scenarios.

Case 1: The proposed sorting approach performed well on large lists. It took very less number of comparisons with large lists. While in case of very smaller lists it took more number of comparisons than that of insertion sort.

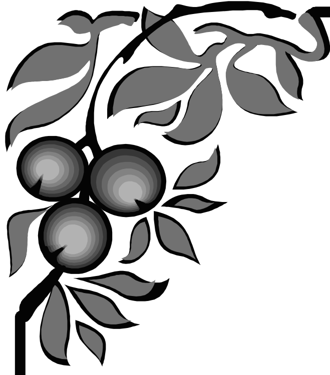
Case 2: When the numbers are already sorted in increasing order. The proposed approach takes 2*N comparisons instead of N comparisons taken by insertion sort.
For e.g. 1 2 3 4 5 6 7 8

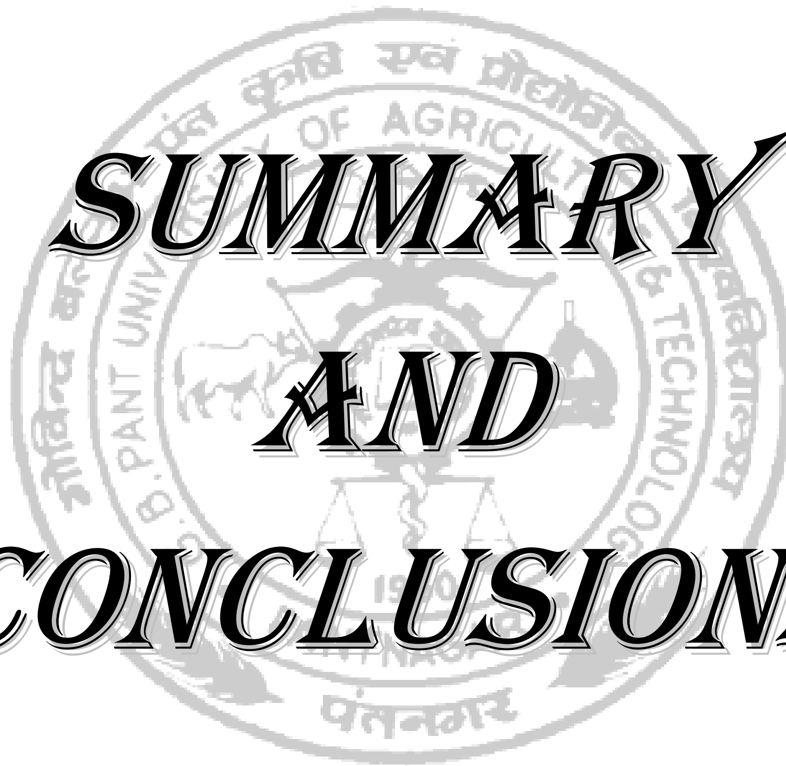
Case 3: When the numbers are arranged in a sequence i.e. largest value comes at first position and second largest value comes at last position. This sequence is repeated like third largest comes at second position and fourth largest at second last and so on.
For e.g. 8 6 4 2 1 3 5 7
In this case also the number of comparisons required by proposed approach is more than that of insertion sort.

Case 4: When the numbers are nearly sorted as shown by example.

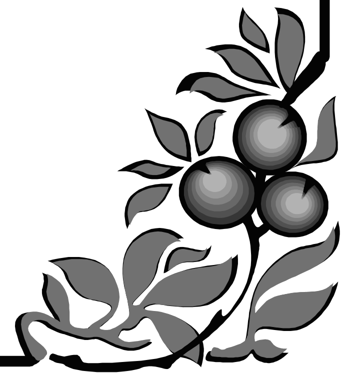
For e.g. 1 2 3 5 4 6 7 8

In this case, the number of comparisons taken by proposed sorting approach is more than that of needed by insertion sort.





SUMMARY AND CONCLUSIONS



This chapter discusses the conclusion of the work presented in this thesis and future work possible.

5.1 Conclusion

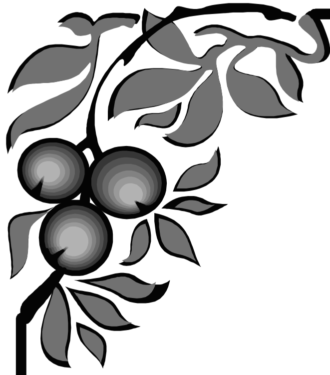
The Data availability requirement is increasing with the advancement of new technology and the vast increased usage of the Internet along with the various applications which require large amount of data for processing. So as to overcome the problem of maintaining and organizing huge amount of data, a fast and efficient sorting algorithm is needed.

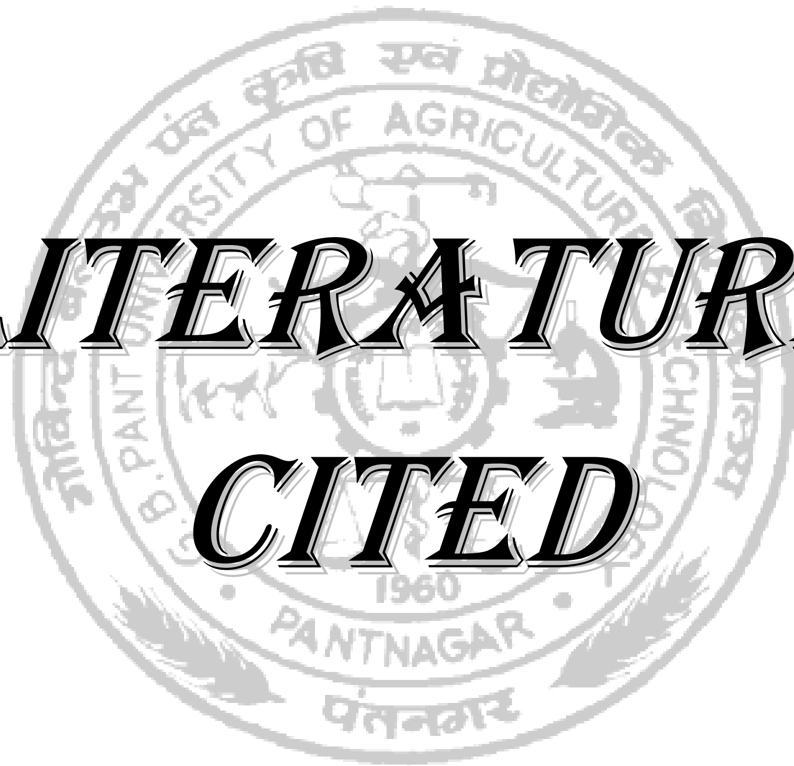
For this study, we have chosen the Insertion sort and this study focuses on providing some advancement and enhancement to the insertion sort algorithm and also makes comparison of proposed sorting approach with the rest of sorting algorithms - bubble sort, insertion sort and selection sort algorithms, along with the results of its practical performance. There are basically three main factors which come during this study - running time, number of comparisons and number of swaps performed, as all these are very acute for the efficiency of any sorting algorithm. Further, in the study, the Experimental results also clearly show that the theoretical performance behavior is in relevance to the practical performance of each sorting algorithm.

The results show that in most of the cases there is huge reduction in number of comparisons required to sort large number of input by using proposed sorting approach compared to that of required in bubble sort, selection sort and insertion sort, it also shows reduction in the number of swaps performed to sort large numbers. In addition to this, the experimental results also proved that the algorithm's behavior will change according to the size of the elements to be sorted and type of its input elements.

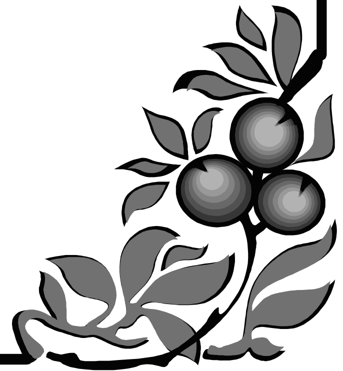
5.2 Future Work

In the future, proposed sorting approach will be used with another sorting algorithm as a hybrid sorting method and the number of swaps taken by proposed sorting approach can be reduced and compared with other existing sorting algorithms.





LITERATURE CITED



Agarwal, A.; Pardesi, V. and Agarwal, N. 2013. A New Approach to Sorting: Min-Max Sorting Algorithm. *International Journal of Engineering Research & Technology,* Vol. 2.

Aho, A.; Hopcroft, J. and Ullman, J. 2004. Data Structures and Algorithms. *Pearson India,* Reprint.

Alnihoud, J. and Mansi, R. 2010. An Enhancement of Major Sorting. *International Arab Journal of Information Technology,* Vol. 7, No. 1, pp.55 – 62.

Alsuwaiyel, M. H. 1999. Algorithms: Design Techniques and Analysis. *King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia*.

Arora, N.; Tamta, V. K. and Kumar, S. 2012. A Novel Sorting Algorithm and Comparison with Bubble sort and Insertion sort. *International Journal of Computer Applications,* Vol. 45, pp. 31-32.

Bell, T. and Aspvall, B. 2011. Sorting Algorithms as Special Cases of a Priority Queue Sort. *ACM*, USA.

Beniwal, S. and Grover, D. 2013. Comparison of Various Sorting Algorithms: A review. *International Journal of Emerging Research in Management & Technology*, Vol. 2, Issue- 5, pp. 83 – 86.

Chhajed, N.; Uddin, I. and Bhatia, S. S. 2013. A Comparison Based Analysis of Four Different Types of Sorting Algorithms in Data Structures with Their Performances. *International Journal of Advanced Research in Computer Science and Software Engineering*, Vol. 3, pp. 373-381.

Chhatwani, P. K. and Somani, J. S. 2013. Comparative Analysis & Performance of Different Sorting Algorithm in Data Structure. *International Journal of Advanced Research in Computer Science and Software Engineering*, Vol. No. 3.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. and Stein, C. 2001. Introduction to Algorithms.2nd edition. *MIT Press*, Cambridge.

Dave, P. H. and Dave H. B. 2008. Design and Analysis of Algorithms. 1st edition, *Pearsom Education*.

Huang, B. C. and Langston, M. A. 1988. Practical In-Place Merging. *Communication of the ACM,* Vol. No. 31, pp. 348-352.

Iqbal, S. Z.; Gull, H. and Muzaffar, A. W. 2009. A New Friends Sort Algorithm. *IEEE,* pp. 326-329.

Kadam, P. and Kadam, S. 2014. Root to Fruit (1): An Evolutionary Approach for Sorting Algorithms. *Oriental Journal of Computer Science & Technology,* Vol. 7, No. 01, pp. 111-116.

Kadam, P. and Kadam, S. 2014. Root to Fruit (2): An Evolutionary Approach for Sorting Algorithms. *Oriental Journal of Computer Science & Technology,* Vol. 7, No. 03, pp. 369-376.

Kapur, E.; Kumar, P. and Gupta, S. 2012. Proposal of a two way sorting algorithm and performance comparison with existing algorithms. *International Journal of Computer Science, Engineering and Applications*, Vol.2, No.3.

Khairullah, M. 2013. Enhancing Worst Sorting Algorithms. *International Journal of Advanced Science and Technology*, Vol. 56.

Mishra, A. D. and Garg, D. 2008. Selection of Best Sorting Algorithm. *International Journal of Intelligent Information Processing,* pp.363-368.

Moses, O. O. 2009. Improving the performance of bubble sort using a modified diminishing increment sorting. *Academic Journals - Scientific Research and Essay*, Vol. 4, pp. 740-744.

Nenwani, K.; Mane, V. and Bhame, S. 2014. Enhancing Adaptability of Insertion Sort through 2-Way Expansion. *In:* 5th International Conference- Confluence The Next Generation Information Technology Summit (Confluence) IEEE. pp. 843-847.

Qureshi, M. A. 2009. Qureshi Sort: A new Sorting Algorithm. *IEEE.*

Seymour Lipschutz. 2011. Data Structures with C. *Tata McGraw-Hill*, Inc., New York.

Shell D. 1959. A High Speed Sorting Procedure. *Computer Journal of Communications of ACM*, vol. 2, no. 7, pp. 30-32.

Shukla, A. and Saxena, A. K. 2012. Review of Radix Sort & Proposed Modified Radix Sort for Heterogeneous Data Set in Distributed Computing Environment. *International Journal of Engineering Research and Application,* Vol. No. 2, pp.555-560.

Srinivas, R. and Deepthi, A. R.2013. Novel Sorting Algorithm. *International Journal on Computer Science and Engineering,* Vol. 5, No. 01, pp. 43-46.

Srivastava, R.; Tiwari, T. and Singh, S. 2013. Bidirectional Expansion—Insertion Algorithm for Sorting. *In:* Second International Conference on Emerging Trends in Engineering and Technology, ICETET-13.

Umar, M.F.; Munir, E. U., Shad, S. A. and Nisar, M. W. 2014. Enhancement of Selection, Bubble and Insertion Sorting Algorithm. *Research Journal of Applied Sciences, Engineering and Technology,* pp. 263-271.

Wegner, L. M. 2014. Sorting- The Turku Lectures. *TUCS Lecture Notes, TUCS, Springer.* pp. 1-157.

www.cs.virginia.edu/~luebke/cs332. Date of visit: 14th January, 2015.

www3.cs.stonybrook.edu/~skiena/214/lectures/lect16/lect16.html. Date of visit: 14th January, 2015.

www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html. Date of visit: 14th January, 2015.

www.sorting-algorithms.com. Date of visit: 5th February, 2015.

www.cs.nthu.edu.tw/~wkhon/algo08.html. Date of visit: 5th February, 2015.

www.algs4.cs.princeton.edu/25applications. Date of visit: 22th February, 2015.

www.ics.uci.edu/~eppstein/161/960116.html. Date of visit: 25th February, 2015.

www.xlinux.nist.gov/dads. Date of visit: 4th March, 2015.

www.rosettacode.org/wiki/Sorting_algorithms/Permutation_sort. Date of visit: 4th March, 2015.

www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting.countingSort.html. Date of visit: 4th March, 2015.

Yadav, R.; Varshney, K. and Verma, N. K. 2013. Analysis of Recursive and Non-recursive Merge Sort Algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering,* Vol. No. 3, pp. 977-981.

VITA

*The author, **Arun Saini** was born in village-Chandra Nagar in district-Moradabad (U.P) on 8th September 1990. He passed his High School and intermediate examinations with first division, from Parker Intermediate College, Moradabad affiliated to Uttar Pradesh State Board, Allahabad. He earned his Bachelor's degree in Computer Science and Engineering from Moradabad Institute of Technology, Moradabad, affiliated to U.P.T.U., Lucknow in 2012 with first division and then, he took admission in the college of Post Graduate Studies in G. B. Pant University of Agriculture and Technology, Pantnagar, (U.S. Nagar) Uttarakhand in July 2013, for Master's Degree in Computer Engineering.*

Permanent Address:

Arun Saini
S/O Harish Chandra Saini
195, Bhogpur Mithoni,
Chandra Nagar,
District- Moradabad, Pin -244001
Uttar Pradesh (India)
Email: arun.mit.08@gmail.com
Mobile No. +91-8791150199

ABSTRACT

Name	:	Arun Saini	**Id. No.**	:	45619
Semester & Year of admission	:	I, 2013-14	**Degree**	:	Master of Technology
Major	:	Computer Engineering	**Dept.**	:	Computer Engineering
Thesis Title	:	**Study and Improvement over Insertion Sort along with Comparative Analysis**			
Advisor	:	Prof. B.K. Singh			

Sorting is most widely used activity in the computer world without which computer cannot complete a single task efficiently. Sorting is used in many applications like searching, real-time systems and operating systems. Although there is a large number of sorting algorithms available but there is no single sorting algorithm that fulfills all the requirements. In this work I proposed a sorting technique; a type of hybrid sort method is based on the idea similar to insertion sort and efficiently sorts the elements of a given list. This sorting approach uses an additional step and binary search is used to find the exact location at which element is to be inserted in the list. A comprehensive empirical analysis has been done on the proposed approach and it has been compared with the traditional sorting methods- bubble, insertion, and selection sort.

The results showed that proposed sorting approach is fast and effective than the sorting algorithms belong to $O(N^2)$ complexity class like insertion sort, selection sort and bubble sort in most of the cases as it required very less number of comparisons and takes less time and swaps to sort the elements of a list.

(B.K. Singh)
Advisor

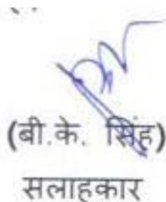
(Arun Saini)
Author

सारांश

नाम	: अरुण सैनी	**परिचयांक**	:	45619
छमाही एवं प्रवेश का वर्ष	: I, 2013-14	**उपाधि**	:	स्नातकोत्तर
प्रमुख	: संगणक अभियांत्रिकी	**विभाग**	:	संगणक अभियांत्रिकी
शोधशीर्षक	: अध्ययन एवं तुलनात्मक विश्लेषण के साथ इन्सरशन सॉर्ट के ऊपर सुधार			
सलाहकार	: प्रो. बी.के. सिंह			

सॉर्टिंग, कंप्यूटर जगत मे सर्वाधिक उपयोग की जाने वाली गतिविधि है जिसके बिना कंप्यूटर एक भी कार्य कुशलतापूर्वक समाप्त नही कर सकता है। सॉर्टिंग का उपयोग विभिन्न अनुप्रयोगो जैसे कि सर्चिंग, सेलेक्शन, रियल सिस्टम और ऑपरेटिंग सिस्टम आदि मे किया जाता है। हालांकि, बहुत सारे सॉर्टिंग एल्गोरिथ्म्स उपलब्ध है लेकिन किसी भी एक सॉर्टिंग एल्गोरिथ्म से सभी ज़रूरतों को पूरा नही किया जा सकता। इस कार्य मे मैं एक सॉर्टिंग तकनीक प्रस्तावित करता हूँ जो एक प्रकार का हाइब्रिड सॉर्ट विधि है जो इन्सरशन सॉर्ट के विचार पर आधारित है एवं सूची के एलिमेंट्स को कुशलतापूर्वक सॉर्ट करती है। इस विधि मे एक अतिरिक्त चरण का उपयोग किया गया है एवं बाइनरी सर्च का प्रयोग एलिमेंटका सही स्थान खोजने मे किया जाता है। प्रस्तावित एप्रोच का अनुभवजन्य विश्लेषण किया गया है एवं इसकी तुलना पारंपरिक सॉर्टिंग विधियो जैसे कि ब्बलसॉर्ट, इन्सरशन सॉर्ट और सेलेक्शन सॉर्ट से की गई है।

परिणामों से पता चलता है कि प्रस्तावित सॉर्टिंग तकनीक $O(N^2)$ जटिलता श्रेणी के सॉर्टिंग एल्गोरिथ्म्स जैसे- ब्बलसॉर्ट, इन्सरशन सॉर्ट और सेलेक्शन सॉर्ट से अधिकतर स्तिथियो मे अधिक तेज़ एवं प्रभावी है क्योंकि इस सॉर्टिंग तकनीक मे दी गई बड़ी सूची के एलिमेंट्स को सॉर्ट करने के लिए कम कमपेरीसन्स की संख्या तथा कम समय लगता है।



(बी.के. सिंह)
सलाहकार

(अरुण सैनी)
लेखक