MASTERS THESIS

DESIGN AND IMPLEMENTATION OF AN EFFICIENT SORTING ALGORITHM

Wasi Haider Butt

In The Name Of ALLAH, The Most Beneficial, The Most Merciful.

ACKNOWLEDGEMENTS

I am thankful to Allah Almighty for giving me the ability to finish this research work and allow me to acquire knowledge from it.

I am thankful to my supervisor Brig. Dr. Muhammad Younus Javed for his guidance and help and all the committee members for their time and diligence.

Finally, I want to thank my parents for the continuous encouragement, understanding and moral support.

Dedicated to my Parents and Teachers

ABSTRACT

A Sorting Algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical orders and lexicographical orders. There are two basic requirements that a sorting algorithm must fulfill, the first one is that the output should be any permutation or reordering of the input and the second is that all the elements in the output should be in the desired order, which is increasing or decreasing. Sorting is of significant importance as we live in a world obsessed in keeping information. In order to efficiently search for required information we must keep that information in a sensible (logically appealing) order. So for our convenience computers spend a considerable amount of time on keeping data in order. Basically sorting is the rearranging of given items on the basis of some well defined ordering rules. From the very start of computer science the sorting problem, due to its immense usefulness, has invited the interest of researchers. The aim is to reduce the cost and complexity of the algorithm and to make them achieve efficiency levels untouched in the past. In the thesis, a new sorting method has been proposed, its algorithm has been derived and performance has been compared with well known existing methods. It was discovered that the algorithm proposed in this thesis is relatively simpler and efficient.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES **PAGE NO.**

LIST OF TABLES

TABLES **PAGE NO.**

CHAPTER 1

INTRODUCTION

1.1. Sorting

English language dictionaries define "sorting" as the process of separating or arranging things according to class or kind, it is traditional for computer programmers to use the word in the much more special sense of sorting things into ascending or descending order. It is more likely to be said "the ordering" but saying it ordering may be confusing. Some people also suggest the word "sequencing" as an appropriate name for ordering process, but this word often seems to lack the right suggestion, especially when equal elements are present, and it occasionally conflicts with other terminology. It is quite true that sorting is itself an overused word but it has become firmly established in computing. Therefore the word sorting is used for all the techniques used to arrange data in the desired order.

1.2. Sorting Algorithms

A Sorting Algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical orders and lexicographical orders. There are two basic requirements that a sorting algorithm must fulfill, the first one is that the output should be any permutation or reordering of the input and the second is that all the elements in the output should be in the desired order, which is increasing or decreasing. Sorting is of significant importance as we live in a world obsessed in keeping information. In order to efficiently search for required information we must keep that information in a sensible (logically appealing) order [1]. So for our convenience computers spend a considerable amount of time on keeping data in order [2]. Basically sorting is the rearranging of given items on the basis of some well defined ordering rules [3]. From the very start of computer science the sorting problem, due to its immense usefulness, has invited the interest of researchers. The aim is to reduce the cost and complexity of the algorithm and to make them achieve efficiency levels untouched in the past. Efficient sorting is

important to optimize the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful to canonicalize data and for producing output readable to humans.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

1.3. History and Importance

Sorting is the elemental algorithmic problem in computer science, being the first step in solving many other algorithmic problems. Donald Knuth, author of The Art of Computer Programming, wrote [4]: “I believe that virtually every important aspect of programming arises somewhere in the context of searching or sorting”. The origins of sorting algorithms, their roots are found in discrete mathematics. The starting point is the book by Knuth, which discusses the history of sorting from the nineteenth century, when the first machines for sorting were invented. In the thesis main references from that book have also been used. The book also explains the importance of sorting by quoting some sayings from the ancient times. Some interesting among them are as follows:

“There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things”.

Niccolo Machiavelli (The Prince, 1513) [5]

"But you can't look up all those license numbers in time," Drake objected. "We don't have to, Paul. We merely arrange a list and look for duplications".

Perry Mason (The Case of Angry Mourner, 1951) [5]

"Tree Sort " Computer, with this new computer-approach to nature study you can quickly identify over260 different trees of U.S., Alaska, and Canada, even palms, desert trees, and other exotics. To sort, you simply insert the needle"

Catalog of Edmund Scientific Company (1964) [5]

1.4. Some Important Applications

- Solving the togetherness problem in which all the elements with same identification are brought together. Suppose that we have 10,000 items in random order containing duplicate values and we have to rearrange them in the order that equal values appear in consecutive positions.
- If more than one file have been sorted in same order, it is probable to find all of the matching entries in one sequential pass through them without backing up. It is generally much more reasonable to access a list of information in sequence from beginning to end, instead of skipping around at random in the list, unless the list is small enough to be stored in a high speed RAM. Sorting makes it possible to use sequential accessing on large files, as a feasible substitute for direct addressing.
- Sorting helps the searching process and so helps to make computer output more suitable for human consumption.

Although sorting has conventionally been used frequently for business data processing, it is in fact a basic tool which a programmer should keep in mind for use in a broad range of situations.

1.5. Sorting Algorithms Classification Criteria

Sorting algorithms are generally classified on the basis of:

- **Computational complexity** (worst, average and best behavior) of element comparisons in terms of the size of the list 'n'. For typical sorting algorithms good behavior is O (n log n) and bad behavior is $\Omega(n^2)$. Ideal behavior for a sort is O (n). Comparison sorts, sort algorithms which only access the list via an abstract key comparison operation, always need Ω(n log n) comparisons in the worst case.
- Computational complexity of swaps (for **"in place"** algorithms).
- **Memory usage** (and use of other computer resources). In particular, some sorting algorithms are "in place", such that only O(1) or O(log n) memory is needed beyond the items being sorted, while others need to create auxiliary locations for data to be temporarily stored.
- **Recursion.** Some algorithms are either recursive or non recursive, while others may be both (e.g., merge sort).
- **Stability:** stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a **comparison sort**. A comparison sort examines the data only by comparing two elements with a comparison operator.
- **General method**: insertion, exchange, selection, merging, *etc*. Exchange sorts include bubble sort and quick sort. Selection sorts include shaker sort and heap sort.

1.6. Memory usage patterns

When the size of the array to be sorted approaches or exceeds the available primary memory, so that (much slower) disk or swap space must be employed, the memory usage pattern of a sorting algorithm becomes important, and an algorithm that might have been fairly efficient when the array fit easily in RAM may become impractical. In this scenario, the total number of comparisons becomes (relatively) less important, and the number of times sections of memory must be copied or swapped to and from the disk can

dominate the performance characteristics of an algorithm. Thus, the number of passes and the localization of comparisons can be more important than the raw number of comparisons, since comparisons of nearby elements to one another happen at system bus speed (or, with caching, even at CPU speed), which, compared to disk speed, is virtually instantaneous.

For example, the popular recursive quick sort algorithm provides quite reasonable performance with adequate RAM, but due to the recursive way that it copies portions of the array it becomes much less practical when the array does not fit in RAM, because it may cause a number of slow copy or move operations to and from disk. In that scenario, another algorithm may be preferable even if it requires more total comparisons.

One way to work around this problem, which works well when complex records (such as in a relational database) are being sorted by a relatively small key field, is to create an index into the array and then sort the index, rather than the entire array. (A sorted version of the entire array can then be produced with one pass, reading from the index, but often even that is unnecessary, as having the sorted index is adequate.) Because the index is much smaller than the entire array, it may fit easily in memory where the entire array would not, effectively eliminate the disk-swapping problem. This procedure is sometimes called "tag sort" [6].

Another technique for overcoming the memory-size problem is to combine two algorithms in a way that takes advantages of the strength of each to improve overall performance. For instance, the array might be subdivided into chunks of a size that will fit easily in RAM (say, a few thousand elements), the chunks sorted using an efficient algorithm (such as quick sort or heap sort), and the results merged as per merge sort. This is less efficient than just doing merge sort in the first place, but it requires less physical RAM (to be practical) than a full quick sort on the whole array.

Techniques can also be combined. For sorting very large sets of data that vastly exceed system memory, even the index may need to be sorted using an algorithm or

combination of algorithms designed to perform reasonably with virtual memory, i.e., to reduce the amount of swapping required.

1.7. Scope of The Thesis

In this humble effort, it has been tried to cover all aspects of sorting. Initially the methodologies upon which different sorting algorithms depend have been discussed, and then existing sorting algorithms have been discussed along with necessary details. After that a new algorithm has been proposed that has been discussed in detail. Implementation of the proposed algorithm and comparison made after experimenting are also been provided in order to contrast the performance of the new and the existing methods.

1.8. Problem Statement

The objective was to study the existing slow category of sorting algorithms then to design and implement an efficient sorting algorithm in order to improve data sorting efficiency and to obtain comprehensive simulation results of the developed sorting algorithm then compare performance of the developed algorithm with existing algorithms of the same category.

Chapter 2

LITERATURE SURVEY

This chapter discusses the main methodologies leading which several sorting algorithms are based. Discussion on various existing sorting algorithm is also given in order to understand the various methodologies and the algorithms based on them.

2.1 Sorting Methodologies

While analyzing the existing sorting techniques, we come to know that the core ideas behind all techniques can be classified remaining at a broader level. Donald Knuth in his book "The art of computer programming" discusses the major classification of the ideas upon which almost all known techniques are based. These groups and their further details are discussed below as in the upcoming discussions it will be helpful. Following are these classifications, their detailed explanation and their further classifications if there are any.

2.1.1 Sorting by Insertion

A very important sorting methodology is sorting by insertion. Before considering an element, it is assumed that the preceding elements have already been sorted and then the considered element is placed at its proper place among the previously sorted elements. Several variations of this methodology have lead to several sorting algorithms. Proposed "Relative Sort" that will be discussed in the coming chapters is also based on this methodology. A few variations of this methodology are as under: -

a. Straight Insertion

The most understandable insertion sort is the simplest insertion sort. An example is presented to illustrate the simplest insertion sort. Suppose that $1<j<=N$ and that records $R_1,\ldots..,\ R_{j-1}$ have been rearranged so that $K_1<=K_2<=\ldots..K_{j-1}$. Assume that R throughout denotes the record portion of the element i.e. the actual data and K represents the key portion i.e. on the basis of which the sort is being done.

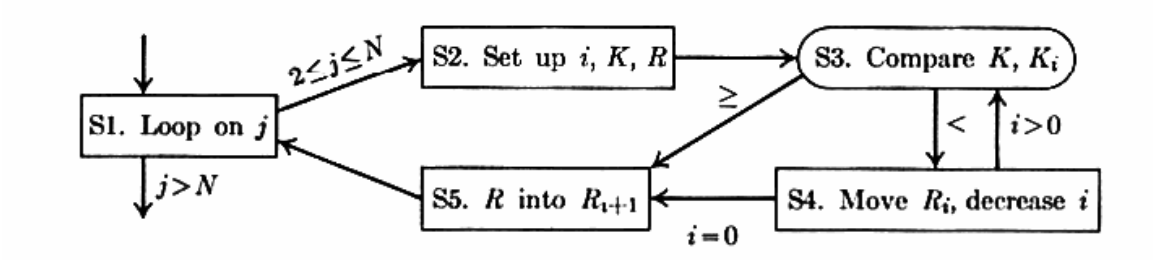


Figure 2.1 : Straight Insertion Algorithm

In each pass, the key K_j being processed is compared with keys K_{j-1} ,K_{j-2} ,……., until assuring that R_j should be inserted between records R_i and R_{i+1} then the records R_{i+1},…..,R_{j-1} are moved up one space and the new record is put at position i+1. It is convenient to combine the comparison and moving operations, interleaving them. Since R_j is set to its proper level so this method is also often called *shifting* or *sinking* technique.

^503:087
087 503^:512
^087 503 512:061
061 087 503 512^:908
061 087^ 503 512 908:170
061 087 170 503 512^ 908:897
. . . . .
061 087 154 170 275 426 503 509 512 612 653 677^ 765 897 908:703
061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

Table 2.1: Example of Straight Insertion

b. Binary insertion and two way Insertion

When j^{th} record is being processed, its key can be compared by with about ½ j of the previously sorted keys. It is some what similar to binary search. For example when inserting 128^{th} record, we can start comparing K_{128} with K_{64}, if it is less, we

can compare it with K_{32} and else if it is greater, we can compare it with K_{96} and so on so that the proper place to insert 128^{th} record will be known after making 12 comparisons. This method is called ***binary insertion*** and it was mentioned by John Mauchly in 1964 [7].

It looks better as it solved the problem of comparison but again the second part of the problem still remained unsolved. As it is found that where to place R_i but how it is to be inserted i.e. how to create room for R_i is still unsolved.

Another alternative was proposed in early 1950's. In this alternate, the first item is placed in the center of the list and space is made for subsequent items by moving to the right or to the left, whatever is more expedient. This works better and saves almost half the running time of ordinary binary insertion but also having an expense of a fairly more complex program. Following table illustrates this method more visibly: -

			^503				
		087	503_^				
		^087	503	512			
	061	087	503	512_^			
		061	087_^	503	512	908	
	061	087	170	503	512	^908	
		061	087	170_^	503	512	897	908
061	087	170	275	503	512	897	908	

Table 2.2: Example of Two way insertion

This seems to work better but there are still some more interesting insertion methods. Let us have a look on them.

c. Diminishing Increment sort

Another variant of insertion was proposed by Donald L. Shell in 1959 [8]. His presented algorithm "Shell Sort" hence belongs to this family. Following table illustrates this idea more clearly. In the given example, first the 16 records to be sorted are divided into 8 groups of two each namely (R_1, R_9), (R_2, R_{10}), ……, (R_8,R_{16}). In the first pass, each group is sorted and the output can be seen in the following table. In the second pass, the records are divided into four groups namely (R_1,R_5,R_9,R_{13}) ,….., (R_4,R_8,R_{12},R_{16}) and again each group is sorted separately. In the third sort, two groups are made at the same pattern and sorted again and the fourth pass makes a single group and sorts all 16 elements. In each pass, we move closer to the sorted list so that straight insertion can be made easily.

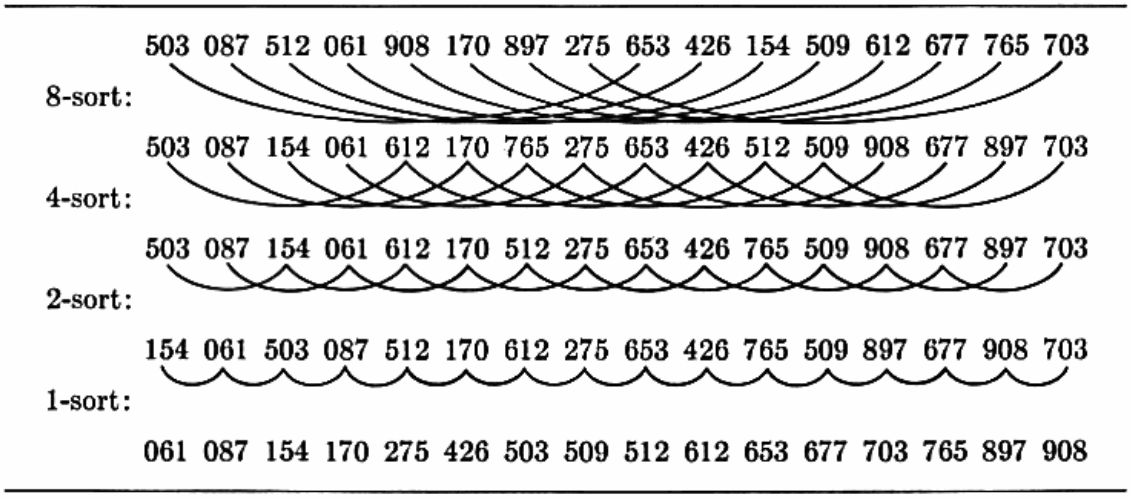


Table 2.3: Diminishing increment Sort

The sequence of the increments is not revered. Any sequence can be used.

2.1.2 Sorting by Exchanging

The second family of sorting algorithms is sorting using "exchange" or "transposition" methods which semantically exchange pairs of elements that are out of order until no more such unordered pairs exist.

The process of straight insertion that has been discussed earlier can be viewed as almost an exchange method as we take new record R_i and exchange it with its neighbors unless it is inserted at its proper place. From this it is also clear that classification of sorting methods into various families is not always clear cut.

This exchange method family is can be further classified into following families.

a. Exchange Selection

The most obvious way to sort by exchange is to compare K_1 with K_2, interchanging R_1 and R_2 if the keys are not in sorted order then doing same for R_2, R_3 and so on. A very common algorithm "Bubble Sort" which will be discussed in detail in the upcoming section belongs to this family. As a result of these operations, the records with larger key values tend to move to the right of the list and the largest value reaches to nth position. Repetition of these operations will eventually place all records at their proper places and the data will become sorted. Following flow chart illustrates this operation more clearly:

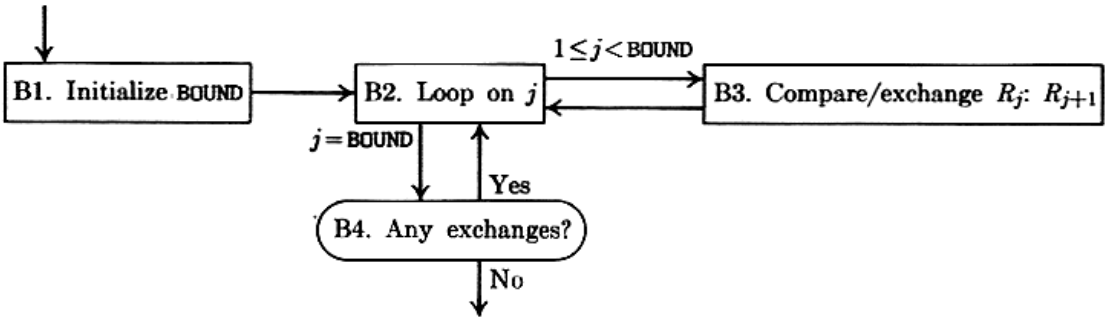


Figure 2. 2: Flow chart of Exchange Selection

Following is an example of this methodology; it illustrates the picture more evidently:

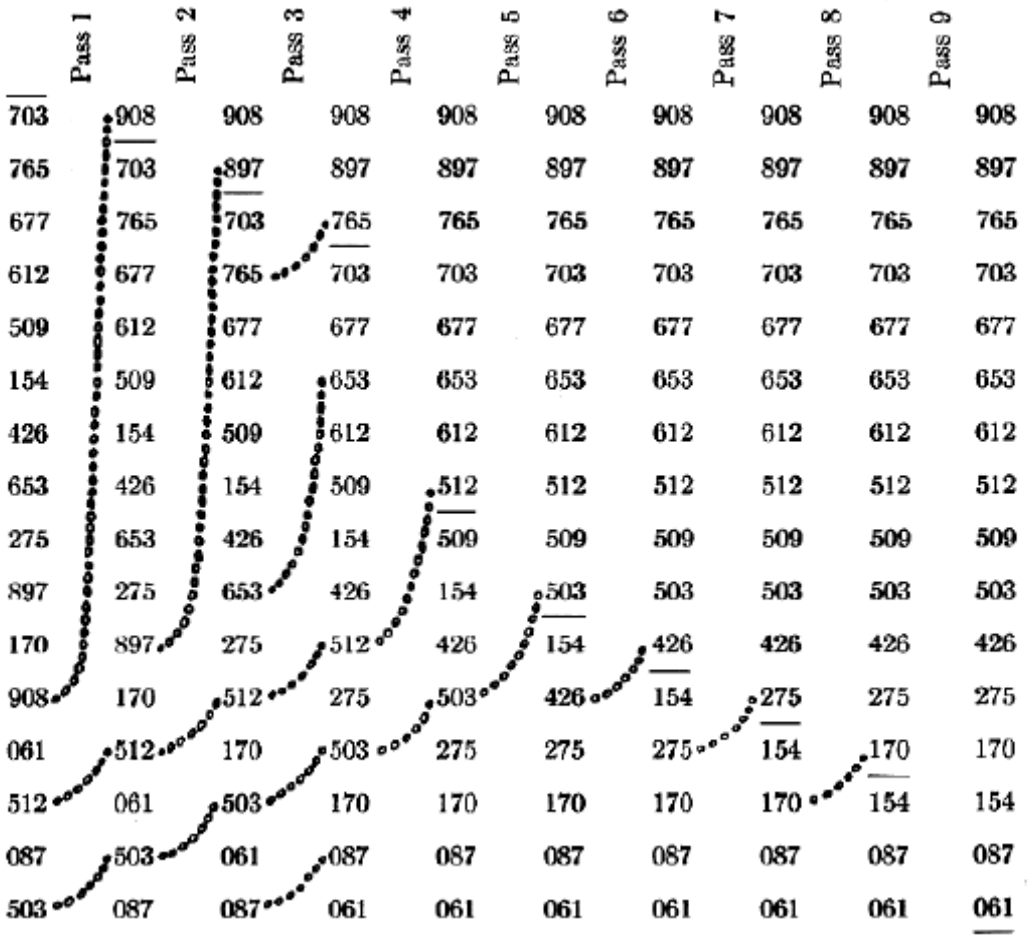


Table 2.4: Exchange Selection in action

b. Merge Exchange

Another disparity of sorting by exchanging is Merge Exchange first presented by K.E. Batcher in 1964 [9] as algorithm known as Batcher's Parallel Algorithm. The method presented is not apparent; in fact; it requires a fairly complicated proof just to show that it is valid, since comparatively few comparisons are made.

Batcher's sorting scheme is almost like Shell's sort discussed earlier, but the comparisons are done in a new way so that no propagation of exchange is necessary.

Following is the flow chart representing operation of this technique graphically:

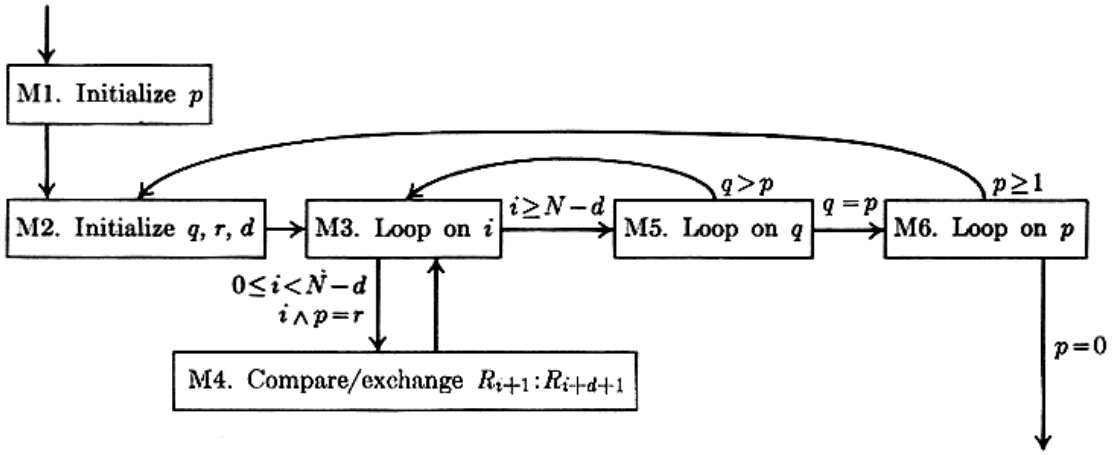


Figure 2.3: Merge Exchange Sort flow chart

Since Batcher's algorithm merges pairs of sorted subsequences so the family is referred to as "Merge Exchange".

Following table illustrates Batcher's method with the help of an example:

p	q	r	d	
				503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703
8	8	0	8	
				503 087 154 061 612 170 765 275 653 426 512 509 908 677 897 703
4	8	0	4	
				503 087 154 061 612 170 765 275 653 426 512 509 908 677 897 703
4	4	4	4	
				503 087 154 061 612 170 512 275 653 426 765 509 908 677 897 703
2	8	0	2	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
2	4	2	6	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
2	2	2	2	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
1	8	0	1	

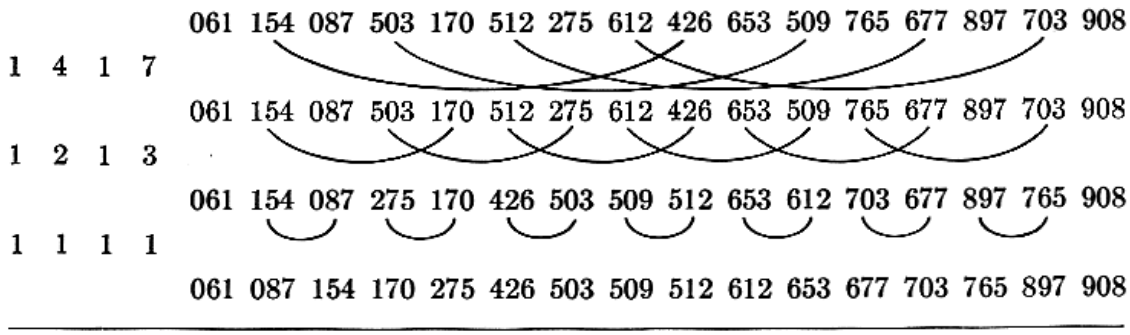


Table 2.5: Batcher's Sort in action

c. Partition Exchange Method

In this new strategy, the result of each comparison is utilized to determine which keys are to be compared next. This strategy is not good for parallel computations, but on serially execution machines, it is quite fruitful.

Consider the following comparison/ exchange scheme. Keep two pointers, i and j, with i=1 and j=N initially. Compare K_i:K_j, and if no exchange is necessary decrease j by 1 and repeat the process. After an exchange first occurs, increase i by 1 and continue comparing and increasing i until another exchange occurs. Then decrease j again and so on. This process goes on until i approaches j or j approaches i.

For example we have the same list as used in earlier examples; this method will work in the following order:

Given:	**503** 087 512 061 908 170 897 275 653 426 154 509 612 677 765 **703**	
		Decrease *j*
1st exchange:	**154** 087 512 061 908 170 897 275 653 426 **503** 509 612 677 765 703	
		Increase *i*
2nd exchange:	154 087 **503** 061 908 170 897 275 653 426 **512** 509 612 677 765 703	
		Decrease *j*
3rd exchange:	154 087 **426** 061 908 170 897 275 653 **503** 512 509 612 677 765 703	
		Increase *i*
4th exchange:	154 087 426 061 **503** 170 897 275 653 **908** 512 509 612 677 765 703	
		Decrease *j*
5th exchange:	154 087 426 061 **275** 170 897 **503** 653 908 512 509 612 677 765 703	
		Increase *i*
6th exchange:	154 087 426 061 275 170 **503** **897** 653 908 512 509 612 677 765 703	
		Decrease *j*

Table 2.6: Partition Exchange Method in action

Following flowchart represent process graphically:

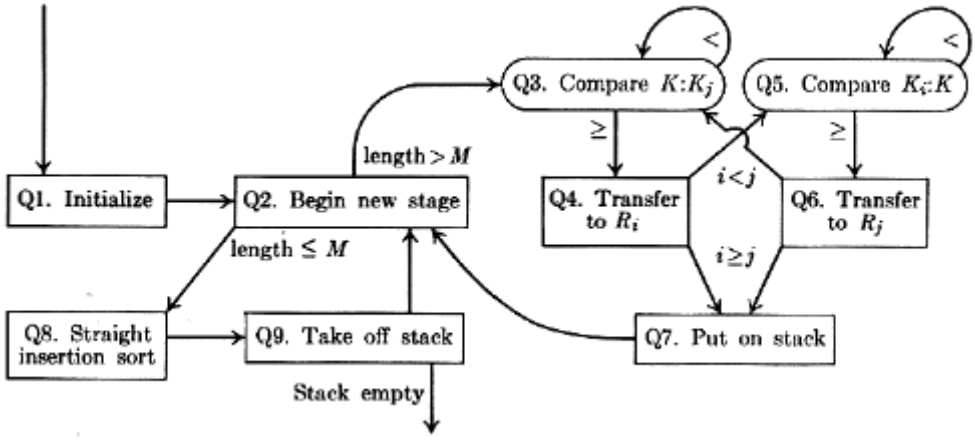


Figure 2.4: Partition Exchange Method flow chart

The most common member of this family is the Quick Sort presented by C.A.R. Hoare in 1962 [10] that will be discussed in detail in the upcoming section.

d. Radix Exchange

Radix exchange, another variant of sorting by exchanging is quite different from almost all the previously discussed methods. It makes use of the binary representations of the keys. As a replacement for of comparing two keys, the method checks the individual bits of the keys. Other respects are nearly like quick sort methodology. As the method depends upon radix 2 representations so it is termed to as "radix exchange method".

First the sequence is sorted for the most significant bit so that all keys having a leading 0 come before all keys containing a leading 1. Sorting is done by finding the left most key K_i having a leading 1 and the rightmost key K_j having a leading 0. Then records are exchanged and the process continues until i approaches j. Some process is repeated for all bits.

2.1.3 Sorting by Selection

Another important family of sorting algorithms is sorting by Selection based on the idea of repeated selection. The simplest method is to find out the smallest key from the data and to transfer it to the output area and then replacing its key with value "∞" assuming it higher than the actual key and then repeating this step finding the second highest key, transferring and so on. The process continues until the last value has been transferred to the output.

The method can be viewed as opposite of insertion as it generates the final outputs one by one in sequence while in insertion the inputs are received sequentially but we do not know any of the final outputs until sorting is completed.

Following flow chart shows the simplest selection method:

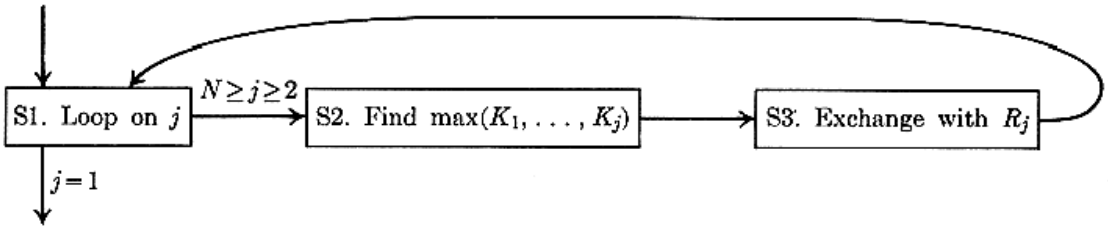


Figure 2.5: Simple Selection Sort flow chart.

Following table shows an example which further elaborates this technique.

503	087	512	061	**908**	170	**897**	275	653	426	154	509	612	677	**765**	**703**
503	087	512	061	703	170	**897**	275	653	426	154	509	612	677	765	908
503	087	512	061	703	170	**765**	275	653	426	154	509	612	**677**	897	908
503	087	512	061	**703**	170	**677**	275	**653**	426	154	509	**612**	765	897	908
503	087	512	061	612	170	**677**	275	**653**	426	154	**509**	703	765	897	908
503	087	512	061	612	170	509	275	**653**	**426**	**154**	677	703	765	897	908
...															
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Table 2.7: Selection by Selection

2.1.4 Sorting by Merging

Sorting by merging has the core idea of combining two unordered collections of records into a single ordered collection. For example we can merge two collection 2,1,0 and 5,4,6 into a single collection 0,1,2,4,5,6. To achieve this, first we need to find out the smallest elements from both collection and then placing the smaller of them in the ordered file and finding again smallest from the unordered collection whose element has been placed and again comparing it to the already found smallest element from the second collection. The process continues till the ends of both unordered collections.

Following flow chart graphically depicts the overall process: -

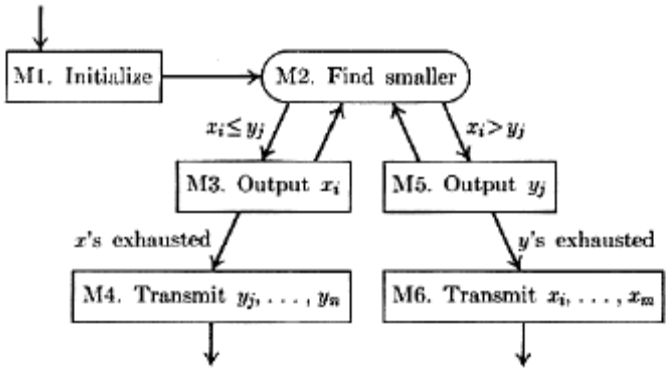


Figure 2.6: Sorting by merging flow chart

In the next section we will see Merge Sort which is a very efficient sorting method belonging to this family.

Following table represents the general merging process of two unordered collections of records.

$$\begin{Bmatrix} 503 & 703 & 765 \\ 087 & 512 & 677 \end{Bmatrix}$$

we obtain

$$087 \begin{Bmatrix} 503 & 703 & 765 \\ 512 & 677 \end{Bmatrix},$$

then

$$087 \quad 503 \begin{Bmatrix} 703 & 765 \\ 512 & 677 \end{Bmatrix},$$

Table 2.8: Example of merging

2.1.5 Sorting by Distributing

Now a new family is presented. It can be viewed as a totally opposite of sorting by merging family. To explain this methodology, an example of cards is presented. Suppose we want to sort 53 card deck of playing cards. We may define "A<2<3<4<5<6<7<8<9<10<J<Q<K", as ordering of the display values for the suits and for ordering of the suits we may define: ♣ < ♦ < ♥ < ♠. A card is placed before a card if its suit is lesser than the suit of the next card or if both are from the same suit; its display value is lesser. So the sorted order will be: A♣ < 2♣ <…..< K♣ <A ♦<…..< K ♠.

The trick to sort the deck in this order is first deal the cards face up into 13 piles, one for each face value. Then collect there piles by putting the aces on the bottom, the 2's face up on the top of them then 3's etc., finally putting the kings (face up) on top. Then turn the deck face down and deal again, this time into four piles of four suits. By putting the resulting piles together, with clubs on the bottom, then diamonds, hearts, and spades, the deck will be in desired order.

The same idea works for sorting numbers and alphabetic data. A much known algorithm belonging to this family is Radix sorting. Following table illustrates the Radix sort operation to sort numeric data. Radix sort will be discussed in detail in the next section.

Input area contents:	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
Counts for units digit distribution:				1	1	2	3	1	2	1	3	1	1			
Storage allocations based on these counts:				1	2	4	7	8	10	11	14	15	16			
Auxiliary area contents:	170	061	512	612	503	653	703	154	275	765	426	087	897	677	908	509
Counts for tens digit distribution:				4	2	1	0	0	2	2	3	1	1			
Storage allocations based on these counts:				4	6	7	7	7	9	11	14	15	16			
Input area contents:	503	703	908	509	512	612	426	653	154	061	765	170	275	677	087	897
Counts for hundreds digit distribution:				2	2	1	0	1	3	3	2	1	1			
Storage allocations based on these counts:				2	4	5	5	6	9	12	14	15	16			
Auxiliary area contents:	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Table 2.9: Radix sort in action

Following flow chat shows the Radix sort overall process:

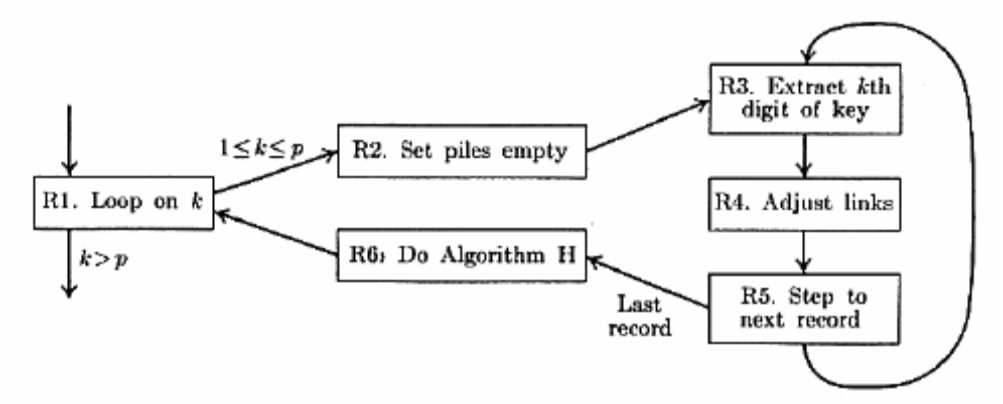


Figure 2.7: Radix Sort flow chart

Along with this we have reached at the end of this section. Now we will see different existing sorting algorithms, their complexities, examples, families etc.

2.2 Existing Well Known Sorting Algorithms

In this section, we will have a look on various well known existing sorting algorithms, how they work, what is he core idea behind them, how much they have contributed in solving the problem, to which sorting family do they belong, what is their complexity and several other issues. Following are their details:

2.2.1 Bubble Sort

Bubble sort is I think the simplest known sorting algorithm. It works by iterating on the input list, comparing two items at a time and swapping them if they are unordered. Iterations are repeated until no more swaps are needed, indicating that the list is sorted. The algorithm got its name as the largest elements "bubbles" to the end of the list in each pass [11]. It belongs to the family “Exchange Sort” as discussed earlier.

Bubble sort has worst-case and average complexity both O (n^2), where *n* is the number of items being sorted.

Following is the pseudo code for Bubble sort:

```
procedure bubbleSort( A : list of sortable items ) defined as:
  do
    swapped := false
    for each i in 0 to length(A) - 2 inclusive do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
    end for
  while swapped
end procedure
```

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in **bold** are being compared.

First Pass:

(**5 1** 4 2 8) ⟶ (**1 5** 4 2 8) Here, algorithm compares the first two elements, and swaps them.

(1 **5 4** 2 8) ⟶ (1 **4 5** 2 8)

(1 4 **5 2** 8) ⟶ (1 4 **2 5** 8)

(1 4 2 **5 8**) ⟶ (1 4 2 **5 8**) Now, since these elements are already in order, algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) ⟶ (**1 4** 2 5 8)

(1 **4 2** 5 8) ⟶ (1 **2 4** 5 8)

(1 2 **4 5** 8) ⟶ (1 2 **4 5** 8)

(1 2 4 **5 8**) ⟶ (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. Algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) ⟶ (**1 2** 4 5 8)

(1 **2 4** 5 8) ⟶ (1 **2 4** 5 8)

(1 2 **4 5** 8) ⟶ (1 2 **4 5** 8)

(1 2 4 **5 8**) ⟶ (1 2 4 **5 8**)

Finally, the array is sorted, and the algorithm can terminate.

Although bubble sort is one of the simplest sorting algorithms to understand and implement, its $O(n^2)$ complexity means it is far too inefficient for use on lists having more than a few elements. Even among simple $O(n^2)$ sorting algorithms, algorithms like insertion sort are usually significantly more efficient. Due to its plainness, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory computer science students.

2.2.2 Cocktail Sort

Cocktail sort, also known as bidirectional bubble sort, cocktail shaker sort, shaker sort (which can also refer to a variant of selection sort), ripple sort, shuttle sort or happy hour sort, is a variant of bubble sort that is a comparison sort [12]. The algorithm differs from bubble sort in that sorts in both directions each pass through the list. This sorting algorithm is only slightly more difficult than bubble sort to implement, and shows itself a bit more efficient than bubble sort. As it's a variant of bubble sort so it also belongs to the family "Exchange Sort".

Following is the simplest pseudo of cocktail sort: -

```
procedure cocktailSort( A : list of sortable items ) defined as:
  do
    swapped := false
    for each i in 0 to length( A ) - 2 do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
    end for
    if swapped = false then
      break do-while loop
    end if
    swapped := false
    for each i in length( A ) - 2 to 0 do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
    end for
  while swapped
end procedure
```

From the above code, it can be observed that the first rightward pass will shift the largest element to its correct place at the end, and the following leftward pass will shift the smallest element to its correct place at the beginning. The second complete pass

will shift the second largest and second smallest elements to their correct places, and so on. After *i* passes, the first *i* and the last *i* elements in the list are in their correct positions, and do not need to be checked.

The complexity of cocktail is $O(n^2)$ for both the worst and the average case, but it becomes closer to $O(n)$ if the list is mostly ordered before sorting.

2.2.3 Combo Sort

Comb sort is a relatively simplistic sorting algorithm originally designed by Wlodek Dobosiewicz in 1980 and later rediscovered and popularized by Stephen Lacey and Richard Box, who described it in Byte Magazine in April 1991 [13]. It also belongs to the family 'Exchange Sort". Combo sort improves on bubble sort, and rivals in speed more complex algorithms like Quick sort. The basic idea is to eliminate small values near the end of the list, since in a bubble sort these slow the sorting down enormously. In bubble sort, when any two elements are compared, they always have a *gap* (distance from each other) of 1. The basic idea of combo sort is that the gap can be much more than one. The gap starts out as the length of the list being sorted divided by the *shrink factor* that is generally taken 1.3, and the list is sorted with that value (rounded down to an integer if needed) for the gap. Then the gap is divided by the shrink factor again, the list is sorted with this new gap, and the process repeats until the gap is 1. At this point, comb sort continues using a gap of 1 until the list is fully sorted. The final stage of the sort is thus equal to a bubble sort, but by this time most small values have been dealt with, so a bubble sort will be efficient. Following is the pseudo code of combo sort.

```
function combsort11(array input)
    gap := input.size
    loop until gap <= 1 and swaps = 0
        if gap > 1
            gap := gap / 1.3
            if gap = 10 or gap = 9
                gap := 11
            end if
        end if

        i := 0
        swaps := 0
        loop until i + gap >= input.size
            if input[i] > input[i+gap]
```

```
                swap(input[i], input[i+gap])
                swaps := 1
            end if
            i := i + 1
        end loop

    end loop
end function
```

2.2.4 Gnome Sort

Gnome sort is a sorting algorithm like insertion sort (insertion sort will be discussed in detail in the coming section), except that moving an element to its proper place is done by a series of swaps like in bubble sort. It got its name from the supposed behavior of the Dutch garden gnome in sorting a line of flowerpots.

Gnome Sort is based on the technique used by the standard Dutch Garden Gnome [13]. Here is how a garden gnome sorts a line of flower pots. Basically, he looks at the flower pot next to him and the previous one; if they are in the right order he steps one pot forward, otherwise he swaps them and steps one pot backwards. If there is no previous pot, he steps forwards; if there is no pot next to him, he is done.

It is theoretically simple, requiring no nested loops having running time $O(n^2)$. In practice the algorithm can run as fast as Insertion sort. The algorithm always finds the first place where two contiguous elements are in the wrong order, and swaps them. It takes benefit of the fact that performing a swap can introduce a new out-of-order contiguous pair only right before or after the two swapped elements. It does not suppose that elements forward of the current position are sorted, so it only needs to check the position directly before the swapped elements.

Following is the pseudo code:

```
function gnomeSort(a[0..size-1]) {
  i := 1
  j := 2
  while i < size
    if a[i-1] <= a[i]
        i := j
        j := j + 1
    else
        swap a[i-1] and a[i]
        i := i - 1
        if i = 0
          i := 1
 }
```

Following is an example which will help in understanding it more clearly.

Let suppose we want to sort an array with values 4, 2, 7, 3 from highest to lowest, here is what would happen with each iteration of the while loop:

- 4, 2, 7, 3 (initial state. i is 1 and j is 2.)
- 4, 2, 7, 3 (did nothing, but now i is 2 and j is 3.)
- 4, 7, 2, 3 (swapped a[2] and a[1]. now i is 1 and j is still 3.)
- 7, 4, 2, 3 (swapped a[1] and a[0]. now i is 1 and j is still 3.)
- 7, 4, 2, 3 (did nothing, but now i is 3 and j is 4.)
- 7, 4, 3, 2 (swapped a[3] and a[2]. now i is 2 and j is 4.)
- 7, 4, 3, 2 (did nothing, but now i is 4 and j is 5.)
- At this point the loop ends because i isn't < 4.

2.2.5 Odd-Even Sort

Odd-even sort is also a simple sorting algorithm. It also belongs to the family "Exchange Sort" as it is also based on bubble sort with which it shares many characteristics. It works by comparing all (odd, even)-indexed pairs of contiguous elements in the list and, if a pair is in the wrong order the elements are swapped. The next step repeats this for (even, odd)-indexed pairs of contiguous elements. Then it alternates between (odd, even) and (even, odd) steps until the list is sorted. It can be considered as using parallel processors, each using bubble sort but starting at different points in the list all odd indices for the first step. This sorting algorithm is only a bit more difficult than bubble sort to implement.

Following is the pseudo code describing the operation of the technique:

```
sorted = false;
     while not sorted
       sorted = true;
           // odd-even
           for ( x = 1; x < list.length-1; x += 2)
              if list[x] > list[x+1]
                 swap list[x] and  list[x+1]
                 sorted = false;
           // even-odd
           for ( x = 0; x < list.length-1; x += 2)
              if list[x] > list[x+1]
                 swap list[x] and  list[x+1]
                 sorted = false;
```

2.2.6 Quick Sort

Quick sort is a well-known sorting algorithm presented by C. A. R. Hoare [16] that, on average, makes $O(n \log n)$ comparisons to sort n items. However, in the worst case, it makes $O(n^2)$ comparisons. Typically, quick sort is significantly faster in practice than other $O(n \log n)$ algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the probability of requiring quadratic time [16]. Quick sort belongs to the family "Partition Exchange".

Following pseudo code defines the operation of Quick Sort.

```
function quicksort(array)
     var list less, greater
     if length(array) ≤ 1
         return array
     select and remove a pivot value pivot from array
     for each x in array
         if x ≤ pivot then append x to less
         else append x to greater
     return concatenate(quicksort(less), pivot, quicksort(greater))
```

2.2.7 Heap Sort

Heap sort is also part of the selection sort family. Although it is somewhat slower in practice on most machines than a good execution of quick sort but it has the advantage of a worst-case Θ $(n \log n)$ runtime [17]. It works as - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Basic implementations require two arrays - one to hold the heap and the other to hold the sorted elements. Following is the typical pseudo code:

2.2.8 Selection Sort

Selection sort is the most common algorithm of Sorting by Selection family. It has $O(n^2)$ complexity, making it inefficient on large lists, and generally performs inferior than the similar insertion sort. Selection sort is noted for its simplicity, and also has

performance compensation over more complex algorithms in certain situations. The general operation is: find the minimum element from the list, swap it with the element at first location, next time find second minimum and swap with element at second location and so on. The operation continues until end of list reaches. In this way we get a sorted list. Following is the typical pseudo code:

```
for i ← 0 to n-2 do
    min ← i
    for j ← (i + 1) to n-1 do
        if A[j] < A[min]
            min ← j
    swap A[i] and A[min]
```

2.2.9 Insertion Sort

Insertion sort is a most simple sorting algorithm of insertion sort family in which the sorted array is built one entry at a time. It is much less efficient on large lists than more advanced algorithms however, insertion sort provides several advantages, for example, its implementation is simpler, it's efficient for quite small lists, it's also efficient for lists where the data is mostly in sorted order and it's more efficient than other quadratic sorting algorithms.

In each iteration, an element is removed from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain.

Following is the simple pseudo code:

```
insertionSort(array A)
begin
    for i := 1 to length[A]-1 do
    begin
        value := A[i];
        j := i-1;
        while j ≥ 0 and A[j] > value do
        begin
            A[j + 1] := A[j];
            j := j-1;
        end;
        A[j+1] := value;
    end;
end;
```

2.2.10 Library Sort

Library sort, or gapped insertion sort is a sorting algorithm that uses an insertion sort, but with gaps in the array to speed up subsequent insertions. The algorithm was proposed by Michael A. Bender, Martín Farach-Colton, and Miguel Mosteiro in 2004 [18]. As it can be seen as a variant of insertion so it belongs to the family "Sorting by Insertion". Its implementation is very similar to a skip list. The drawback of library sort is that it requires extra space for its gaps.

2.2.11 Shell Sort

Shell sort also belongs to the family "Sorting by Insertion". It is a generalization of simple insertion sort. It got its name after its presenter, Donald Shell. Following algorithms shows the procedure used to perform sorting:

```
input: an array a of length n

inc ← round(n/2)
while inc > 0 do:
    for i = inc .. n – 1 do:
        temp ← a[i]
        j ← i
        while j ≥ inc and a[j – inc] > temp do:
            a[j] ← a[j – inc]
            j ← j – inc
        a[j] ← temp
    inc ← round(inc / 2.2)
```

2.2.12 Tree Sort

Tree sort is builds a binary search tree from the keys to be sorted, and then traverses the tree (in-order) so that the keys come out in sorted order. Adding items to a binary search tree is an O(log n) process, so, therefore, adding n items becomes an O(log n) process, making tree sort a fast sort. Its ideal use is when sorting the elements of a stream from a file. Other sorts would be a slower O(n log n) because of additional time to load the elements to a temporary data structure, whereas in a tree sort the act of loading the input into a data structure is sorting it.

2.2.13 Merge Sort

Merge sort is an $O(n \log n)$ sorting algorithm. It belongs to the family "Sorting by Merging". It is an example of the divide and conquer algorithmic paradigm. It was invented by John von Neumann in 1945.

Conceptually, a merge sort works as follows:

1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
2. Divide the unsorted list into two sub lists of about half the size.
3. Sort each sub list recursively by re-applying merge sort.
4. Merge the two sub lists back into one sorted list.

Following is the commonly used pseudo code:

```
function merge_sort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m

    // This calculation is for 1-based arrays. For 0-based, use
length(m)/2 - 1.
    var middle = length(m) / 2
    for each x in m up to middle
         add x to left
    for each x in m after middle
         add x to right
    left = merge_sort(left)
    right = merge_sort(right)
    result = merge(left, right)
    return result
```

2.2.14 Strand Sort

Strand sort is variant of merge sort. It works by repeatedly pulling sorted sub lists out of the list to be sorted and merging them with a result array. Each iteration through the unsorted list pulls out a series of elements which were already sorted, and merges those series together. The name of the algorithm comes from the "strands" of sorted data within the unsorted list which are removed one at a time. 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. Following is the algorithm:

```
procedure strandSort( A : list of sortable items ) defined as:
  while length( A ) > 0
    clear sublist
    sublist[ 0 ] := A[ 0 ]
    remove A[ 0 ]
    for each i in 0 to length( A ) do:
      if A[ i ] > sublist[ last ] then
        append A[ i ] to sublist
        remove A[ i ]
      end if
    end for
    merge sublist into results
  end while
  return results
end procedure
```

2.2.15 Bead Sort

Bead sort is a natural sorting algorithm which was developed by Joshua J. Arulanandham, Cristian S. Calude and Michael J. Dinneen in 2002, and published in The Bulletin of the European Association for Theoretical Computer Science. Both digital and analog hardware implementations of bead sort can achieve a sorting time of $O(n)$; however, the implementation of this algorithm tends to be significantly slower in software and can only be used to sort lists of positive integers. Also, it would seem that even in the best case, the algorithm requires $O(n^2)$ space. The bead sort operation is like the manner in which beads slide on parallel poles, such as on an abacus. However, each pole may have a different number of beads. Initially, it may be helpful to imagine the beads suspended on vertical poles. In Step 1, such an arrangement is displayed using $n=5$ rows of beads on $m=4$ vertical poles. The numbers to the right of each row indicate the number that the row in question represents; rows 1 and 2 are representing the positive integer 3 (because they each contain three beads) while the top row represents the positive integer 2 (as it only contains two beads) [19]. If we then allow the beads to fall, the rows now represent the same integers in sorted order. Row 1 contains the largest number in the set, while row n contains the smallest. If the above-mentioned convention of rows containing a series of beads on poles 1..k and leaving

poles *k*+1..*m* empty has been followed, it will continue to be the case here. The action of allowing the beads to "fall" in our physical example has allowed the larger values from the higher rows to propagate to the lower rows. If the value represented by row *a* is smaller than the value contained in row *a+1*, some of the beads from row *a+1* will fall into row *a*; this is certain to happen, as row *a* does not contain beads in those positions to stop the beads from row *a+1* from falling.

2.2.16 Bucket Sort

Bucket sort, or bin sort works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. Bucket sort sets up an array of empty buckets, scatters i.e. goes over the original array putting each element in its bucket, sorts each non empty bucket, and then gathers i.e. visits the buckets in order and put all elements back into the original array. Following is the pseudo code.

```
function bucket-sort(array, n) is
  buckets ← new array of n empty lists
  for i = 0 to (length(array)-1) do
    insert array[i] into buckets[msbits(array[i], k)]
  for i = 0 to n - 1 do
    next-sort(buckets[i])
  return the concatenation of buckets[0], ..., buckets[n-1]
```

2.2.17 Counting Sort

Counting sort takes advantage of knowing the range of the numbers in the input array 'A'. It uses this range to create an array *C* of this length. Each index *i* in array *C* is then used to count how many elements in *A* have the value *i*. The counts stored in *C* can then be used to put the elements in *A* into their right position in the resulting sorted array.

2.2.18 Pigeon-Hole Sort

Pigeonhole sorting, also known as count sort is suitable for sorting lists of elements where the number of elements (*n*) and the number of possible key values (*N*) are approximately the same [20].

The pigeonhole algorithm works as follows:

1. Set up an array of initially empty "pigeonholes", one pigeonhole for each value in the range of keys. Each pigeonhole will contain a list of values having that key.
2. Go over the original array, adding each object to the list in its pigeonhole.
3. Iterate over the pigeonhole array in order, and put elements from non-empty holes back into the original array.

2.2.19 Radix Sort

Radix sort sorts integers by processing individual digits. It belongs to the family "Radix Exchange". As integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers, radix sort is not limited to integers. Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines.

2.3 Summary

In this chapter we have took an overview of sorting methodologies and the well known existing sorting algorithms and their necessary details. It's important to mention here that the proposed "Relative Sort" that will be discussed thoroughly in the next chapter belongs to the family "Sorting by Insertion" and in and improvement in the simple insertion sort as experimental results have proved. The results are also included and will be discussed in detail in the next chapters.

Chapter 3

PROPOSED RELATIVE SORT

In this chapter a novel sorting algorithm is presented based on comparing the arithmetic mean with each item in the list and deciding position of that element in the new list based on that comparison. Details, core methodology and running cost analysis are discussed in this chapter.

3.1 Introduction

Our proposed Relative sort is based on comparing each element by the arithmetic mean of the given unsorted list. Consequently we then have an idea of the location of that corresponding element. The core of Relative sort is the insertion sort so obviously it belongs to the "Sorting by insertion" family.

As mentioned earlier, Knuth wrote in his book that a clever programmer can think of various ways to reduce the amount of moving the data, the most expensive operation of insertion sort. According to Knuth, the first such trick was proposed in 1950 and was named as binary insertion. I looked this technique after I had made my algorithm, here there was no concrete methodology proposed that where to start, where to go, where to end. However while proposing my algorithm, this technique was not known to me.

The idea of relative sort came into my mind from relative grading as in our institute, there is relative grading and the students are given grades on the basis of the arithmetic mean of the class. Once I was thinking over that and suddenly this idea came into my mind.

The reason of using arithmetic mean is that arithmetic mean of a series will lie most probably in the middle of the given series if the data is evenly distributed or near the middle even if it is unevenly distributed so if we compare a value with arithmetic mean, we can have idea whether the element will come near the starting side of the list or at the closing side. So keeping this idea in mind I made and proposed the new relative sort algorithm. More details are presented in the next section.

3.2 Explanation:

While reading this section, it is requested to keep the straight insertion sort methodology in mind. The insertion sort overview is discussed first. As we have seen that insertion sort is a most simple sorting algorithm of insertion sort family in which the sorted array is built one entry at a time. It is much less efficient on large lists than more advanced algorithms however, insertion sort provides several advantages, for example, its implementation is simpler, it's efficient for quite small lists, it's also efficient for lists where the data is mostly in sorted order and it's more efficient than other quadratic sorting algorithms.

During each iteration an element is removed from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The processing becomes slower when new elements go on added to the new list. Let suppose we are sorting a list of one hundred thousand elements. Ninety nine thousand, nine hundred and ninety nine elements have been placed in sorted order as placed by the algorithm. When the last element was picked up, it may be the smallest element. So now placing it to its appropriate location needs comparing it with ninety nine thousand, nine hundred and ninety nine elements and then the most costly shifting these all ninety nine thousand nine hundred and ninety nine elements on location ahead in order to create room for the new item as we will to know in the start that the current item is the smallest one, rather after every comparison we will come to know that its smaller than the compared element then that element will be shifted and then the next element will be checked and so on until the first location reaches. The whole cost of algorithm is hidden in this comparison and shifting.

The analysis made by Knuth is really impressive. He discusses the two major problems with this simple algorithm. The first one, finding the space for the new element, i.e. the appropriate space and the second one is that once we have found the space, now creating room for the new element, the most expensive task.

Solution to the first problem was proposed namely binary insertion by John Mauchly in 1946. The theme was to use binary insertion. For example if you have placed hundred elements, now for hundred first element if you use simple method, you will have to make one hundred comparisons but if you use the above mentioned method i.e. compare the

element with the middle element of that hundred elements, if the current element is larger, it means that its place lies in the upper fifty have so go straightly to upper have bi passing fifty elements, similarly if the element is smaller, go to smaller half again bi passing fifty elements. In the next pass again go to the middle element, you will again bi pass twenty five elements and so on. Now rather than comparing with each element, the number of comparisons has been decreased exponentially but only half of the problem has been solved. The bigger problem is to shift all the bigger elements one location ahead that is not possible without shifting them one by one, I mean there is no short cut for this as John has found for searching.

To solve the problem of this room creation, a new sort was proposed in 2004 name library sort by Michael A. Bender, Martín Farach-Colton, and Miguel Mosteiro. The idea was to keep gaps to speed up subsequent insertions. The implementation is very similar to skip list. The drawback is of space. It uses a lot of space for the operation.

The proposed Relative sort breaks the burden of insertion sort in two halves hence reducing the 50% burden. Which is not accurate 50% but may be near this? Experimental results have proved that relative sort really has improved the performance of the insertion sort which is presented in discussed in the next chapters.

As in simple insertion, we follow one side only hence building the sorted list from one side only. Comparable to this approach, relative sort starts building the new sorted list from both sides. The arithmetic mean helps the algorithm to decide the appropriate side for the item under consideration. In the following sections, the steps and the pseudo code of the proposed algorithm is proposed and then for better understanding, a small example is presented which depicts the actual working more clearly.

3.3 Steps

- Take a new list having size equal to the size of given list
- Take Arithmetic mean of the given list
- Iterate through the given list and perform the following steps for each item:
    - Compare the picked item with Arithmetic Mean
    - If Item is less than mean:

        - If list is empty, place item at first location
        - Else start from the location where last element was inserted in this direction and compare element with that
        - Iterating downwards and shifting elements one location ahead (if required) place the element to its exact location

    - Else if item is greater than or equal to mean

        - If list empty, place item at last location
        - Else start from the location where last item was inserted in this direction and compare that element with that
        - Iterating upwards and shifting elements one location back (if required), place the element to its exact location

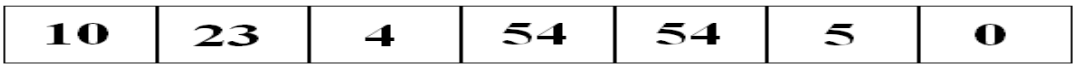
- New list will be in sorted order

3.4 Pseudo Code:

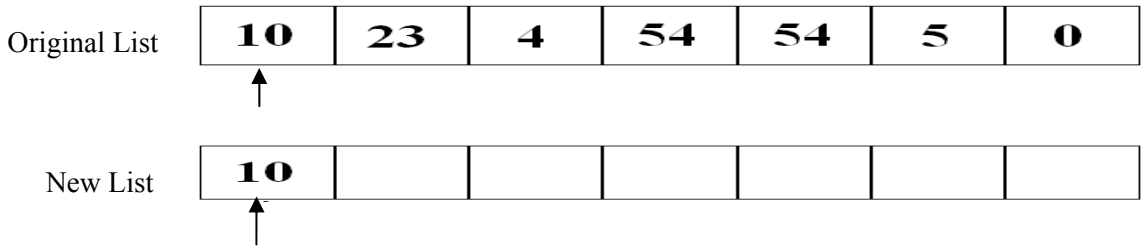
```
Average:= AVERAGE[List]
First:=0
Last:=LENGTH[List]
For i← 1 to LENGTH[List]
 if (First < Last)
  if (List[i] < Average)
   For j←First down to 1
    if (List[i] > NewList[j-1])
     break
    else
   NewList[j] = NewList[j - 1];
   NewList[j] = List[i];
   First++
  else
  For j←Last to LENGTH[List]
    if (List[i] < NewList[j+1])
     break
  else
  NewList[j] = NewList[j + 1]
  NewList[j] = List[i]
  Last--
 else
  NewList[first] = List[i]
```

3.5 Example:

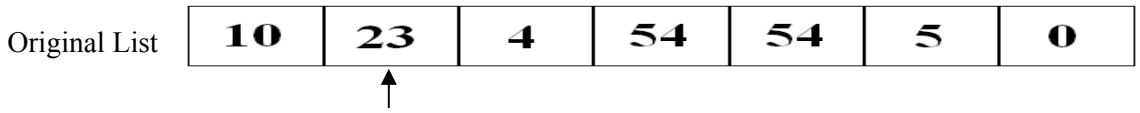
Assume the following given list.



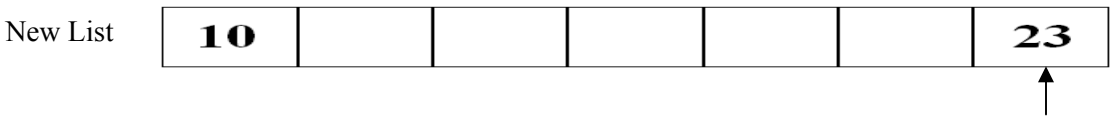
- Total Elements=7
- Sum of Elements=150
- Arithmetic Mean=150/7=21
- Outer Loop Pass 1:
- As i=1 and at first location in given list there is 10 and 10<21 so in the new list it will be put at first location as starting end of new list is empty: new list will become:



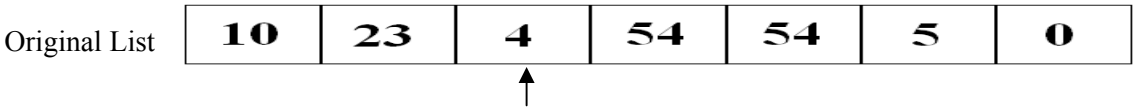
Outer loop pass=2



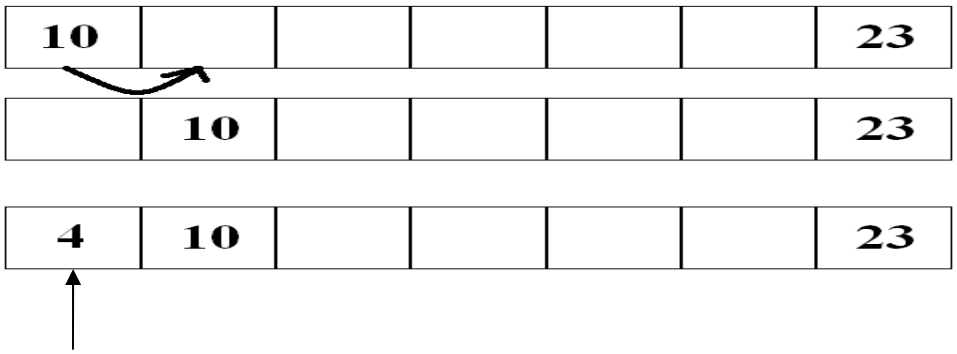
Now i=2 and at second location in given list there is 23 and 23>21 so in the new list it will be put at last location as end is empty: new list will become:



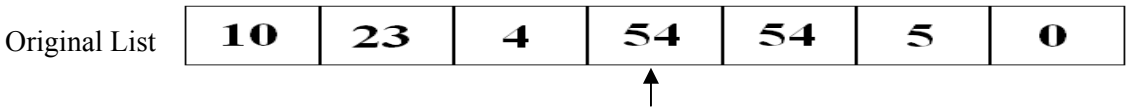
Outer loop pass=3



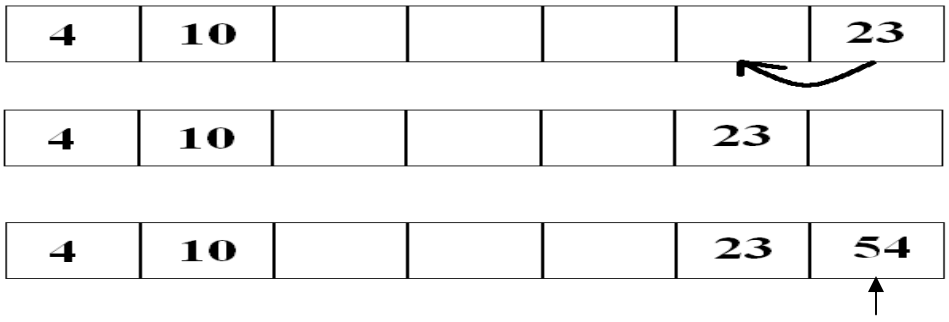
Now i=3 and at third location in given list there is 4 and 4<21 so we will follow the lower end. Here the available location is second. Now inner loop will check if the element before available location in greater than 4, and as 10 >4 so inner loop will be executed and 10 will be shifted on location towards available location. Now again end has reached so 4 will be placed here:



Outer loop pass=4

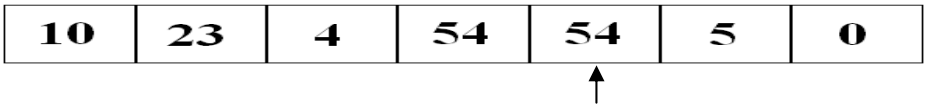


Now i=4 and at forth location in given list there is 54 and 54>21 so we will follow the higher end. Here the available location is sixth. Now inner loop will check if the element after available location in smaller than 54, and as 23 < 54 so inner loop will be executed and 23 will be shifted on location towards available location. Now again end has reached so 54 will be placed here:

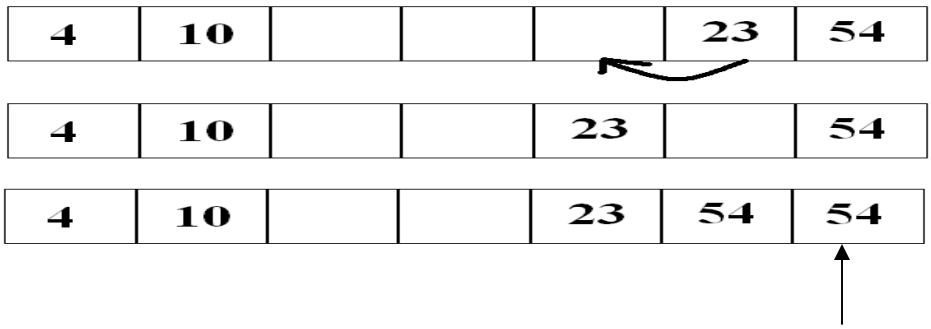


Outer loop pass=5

Original List

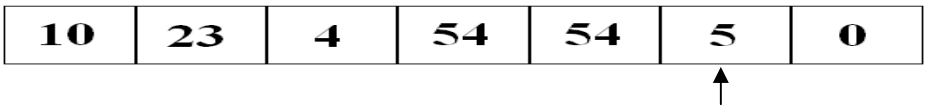


Now i=5 and at fifth location in given list there is 54 and 54>21 so we will follow the higher end. Now the available location is fifth. Now inner loop will check if the element after available location in smaller than 54, and as 23 < 54 so inner loop will be executed and 23 will be shifted on location towards available location. Again 54 will be compared with the next element, now 54 = 54 so 54 will simply be placed here:

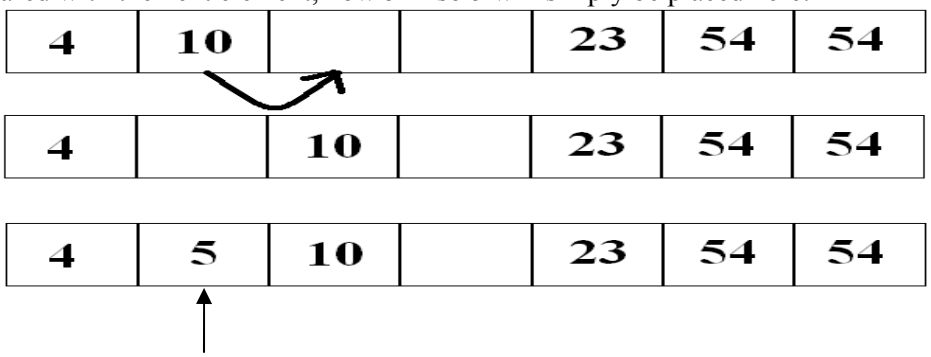


Outer loop pass=6

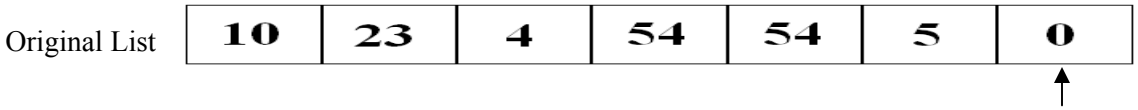
Original List



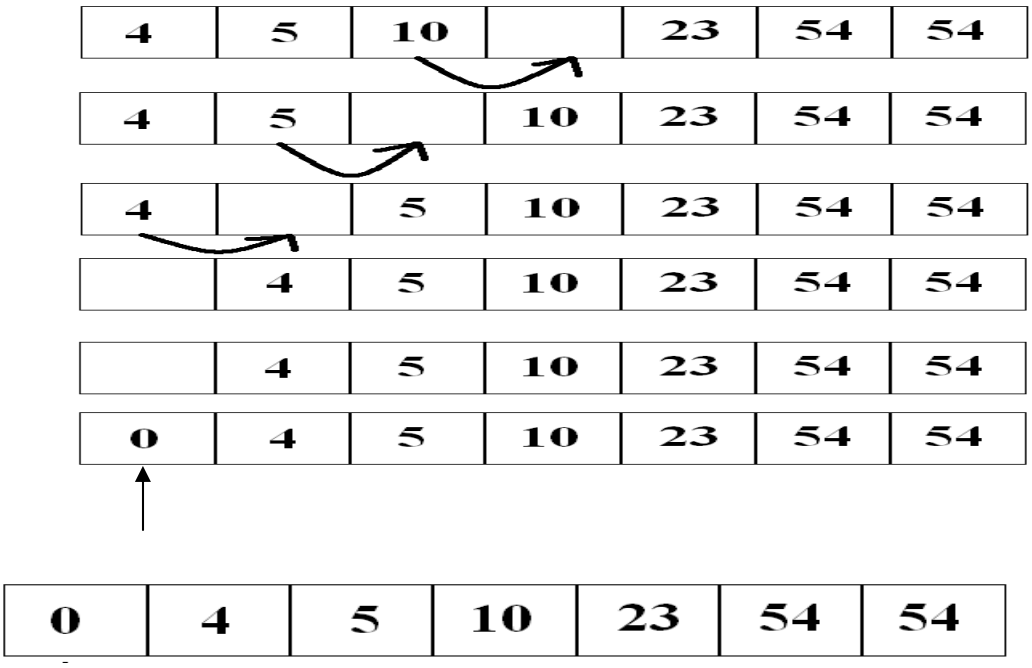
Now i=6 and at sixth location in given list there is 5 and 5< 21 so we will follow the lower end. Here the available location is third. Now inner loop will check if the element before available location in greater than 5, and as 10 > 5 so inner loop will be executed and 10 will be shifted on location towards available location. Again 5 will be compared with the next element, now 5>4 so 5 will simply be placed here:



Outer loop pass=7



Now i=7 and at seventh location in given list there is 0 and 0< 21 so we will follow the lower end. Here the available location is forth. Now inner loop will check if the element before available location in greater than 0, and as 10 > 0 so inner loop will be executed and 10 will be shifted on location towards available location. Again 0 will be compared with the next element, now 0<5 so 5 will be shifted on location towards available location. Now 0 will be compare with 4, again 0<4 so 4 will be shifted on location towards available location. Now end has reached so 0 will simply be placed here:



Outer loop has been terminated as ‘i=7’ now which is the length of the list required to be sorted and now the new list has been sorted.

To compare and contrast proposed Relative sort performance, we implemented it in C#, for more convenience, that code is also presented: -

3.6 C# Code:

```
protected void RelativeSort()
        {
            try
            {
                long inner = 0, outer = 0;
                int sum = 0, avg = 0;
                int length = inputlist.Length;
                outputlist = new int[length];
                for (int i = 0; i < length; i++)
                {
                    sum = sum + inputlist[i];
                }
                avg = sum / length;
                int first = 0, last = inputlist.Length - 1;
                for (int i = 0; i < length; i++)
                {
                    // outer++;
                    if (first < last)
                    {
                        if (inputlist[i] <= avg)
                        {
                            int j;
                            for (j = first; j > 0; j--)
                            {
                                //inner++;
                                if (inputlist[i] > outputlist[j - 1])
                                    break;
                                else
                                    outputlist[j] = outputlist[j - 1];
                            }
                            outputlist[j] = inputlist[i];
                            first++;
                        }
                        else
                        {
                            int j;
                            for (j = last; j < length - 1; j++)
                            {
                                //  inner++;
                                if (inputlist[i] < outputlist[j + 1])
                                    break;
                                else
                                    outputlist[j] = outputlist[j + 1];
                            }
                            outputlist[j] = inputlist[i];
                            last--;
                        }
                    }
                    else
                        outputlist[first] = inputlist[i];
                }
```

After observing the structure of the algorithm, the following running cost analysis was done and now is presented here:

3.7 Running Cost Analysis:

A careful observer will note that there is one main loop within which lie two nested loops. The outer loop will be executed in every case while the execution of inner loops depends on the item taken within an iteration of the list. As the item is compared with the average value, it will traverse in one direction, so out of these two inner loops one loop will be executed. The best case will depict *O(n)* behavior when the given list is arranged in such a way that from the start of the data structure to the location holding the average value the elements are in ascending order and after that in descending order. The worst case will be involve the inverse scenario i.e. the given list is in descending order from the start to the average value and is in ascending order from the average value to the last, in this case cost will be $O(n^2)$. In the average case, the running cost will be $O(n^2/2)$ i.e. n^2 like the insertion sort as the properties are the same as insertion sort other than following two paths at a time and divided by 2 because of dividing the path into two which can also be seen from algorithm structure where inner two loops are there and in each pass of outer loop only one inner loop is executed and the decision that which loop to follow will be made on the basis of comparison with arithmetic mean. The algorithm has been implemented and tested a multiple times. After a thorough analysis we discovered that the sorting will depict average behavior in all cases except then the list is arranged for achieving the best or worst case. In the upcoming section, several comparisons of proposed algorithm with present day, highly used sorting algorithms are included.

3.8 Summary:

The proposed novel relative sort algorithm is a new n^2 algorithm which can be viewed as an enhancement of the simple insertion sort. It is a simple algorithm which uses arithmetic mean to decide the side to be followed for inserting new element. Running cost analysis has been given in order to estimate the cost. The strengths and weaknesses have also been discussed to present an overall picture.

Chapter 4

IMPLEMENTATION AND RESULTS

As discussed earlier, to compare and contrast the performance of the proposed relative sort, we developed a small software named Sort Performance Comparator and checked the live performance of the proposed methodology. In the upcoming sections the detail and the interesting results along with observations are presented.

4.1 Introduction

The software named Software Algorithms Performance Comparator was implemented using C#.Net. To run the developed software, the system requirements are:

a. Windows XP or later
b. Dot Net Framework 2.0 or later

All well known sorting algorithms have been implemented are integrated in this software. The software generates a list of random numbers. The size of list is required to be input by user. Also the starting and closing indices of the list are required to be provided by users. After that users can select any of the two implemented algorithms. The software sorts the same list by both of the provided algorithms. Displays the sorted list by both algorithms and also gives the execution time taken by both of the algorithms in milliseconds. Following section presents the details about the software.

4.2 Software Details

The software has a very user friendly interface. Following figure shows the main page of the software:

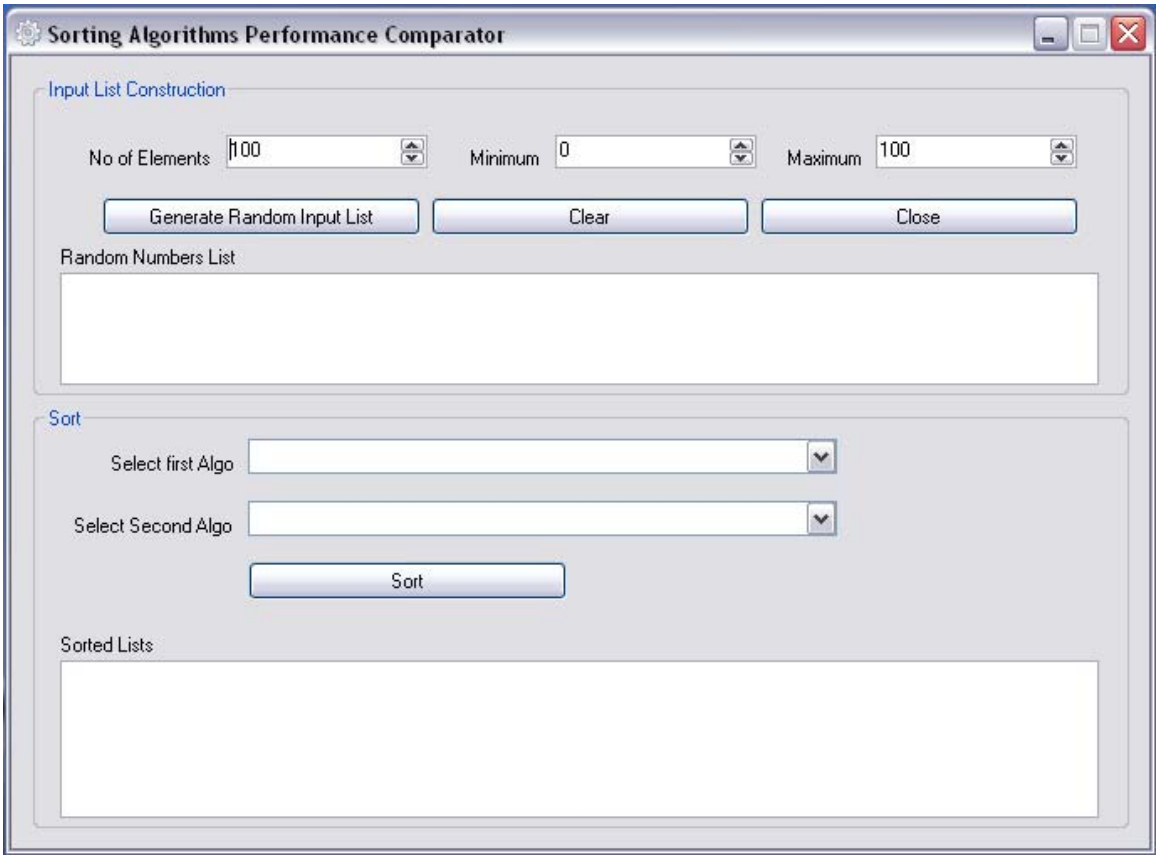


Figure 4.1: Sorting Algorithms Performance Calculator

First of all, the user provides the number of elements, i.e. the desired size of the input list he wants. Then user provides the minimum number, i.e. the starting index of the random list and then maximum i.e. the upper index of the list. The input list is generated by clicking "Generate Random Input List" key present at the form. The desired list is generated and displayed in a panel of random numbers as a result. After this process, user needs to select two algorithms from the two drop down lists as can be seen from the figure above. Sort button needs to be clicked in order to sort the generated list. The input generated list of random numbers is sorted by the two selected algorithms. Two sorted output lists are generated by both selected algorithms respectively and are displayed in the output panel. The execution times are represented in two message boxes respectively in milliseconds. If size of the list is greater then 1000, then the complete list is not displayed on both panels. Values after a regular interval are displayed. Following figures represent an exemplary execution to compare performances of the proposed Relative sort

and Bubble sort. The number of input elements is 5000, starting index is zero and the maximum number is 1000.

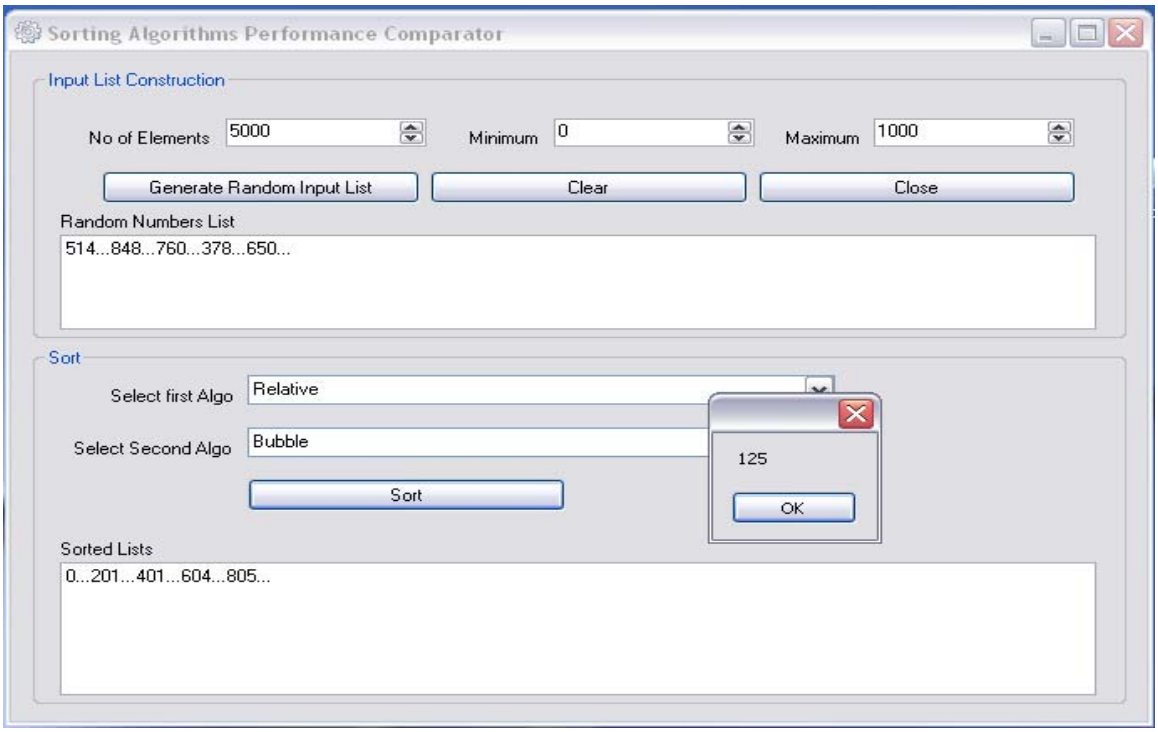


Figure 4.2: Sorted by Relative Sort

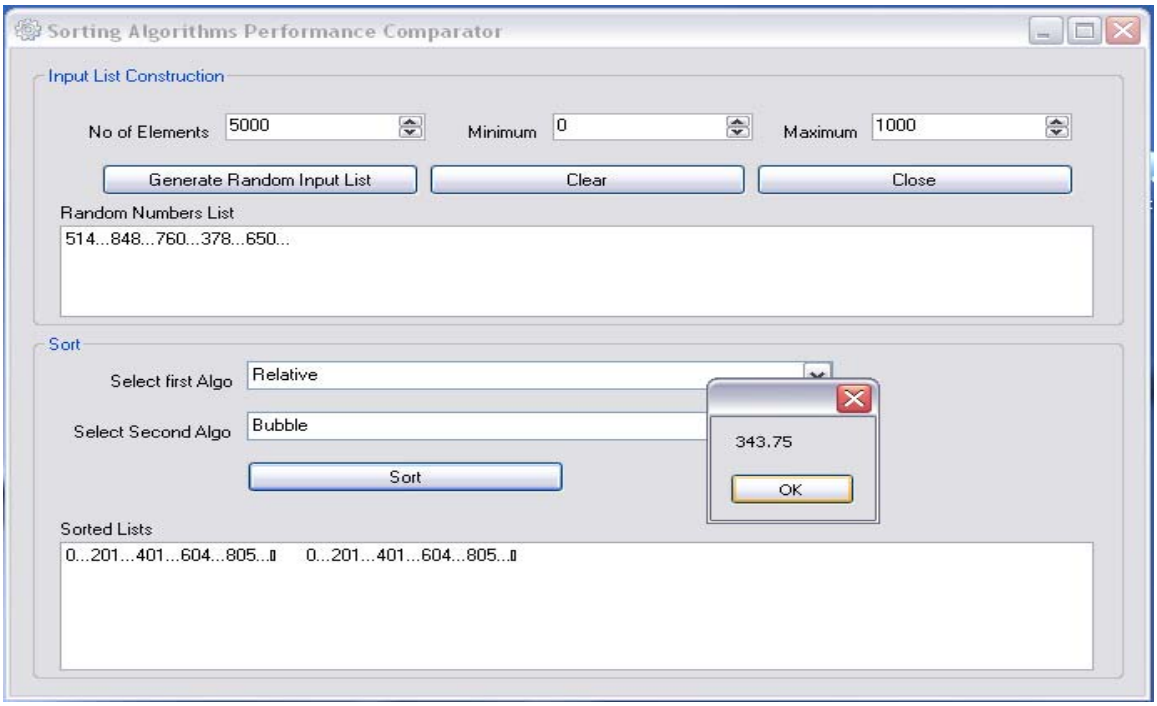


Figure 4.3: Sorted by Bubble Sort

4.3 Experiments:

For live performance checking of the proposed Relative sort, developed software was used and simulation was run on a computer with following specifications:

a. 17. GHz Intel Centrino Processor
b. 512 MB of RAM
c. 2MB Built-in Cache

4.4 Results:

With all implemented algorithms, relative sort was compared. For each comparison, lists of different sizes were generated and sorted. The sizes were 100, 1000, 5000, 10000, 15000, 20000, 30000, 40000, 50000, 70000, 90000 and 100000. Minimum number was kept zero and the maximum was kept 10000 always. Following are the results of these experiments. Graphical as well as textual description of the results is presented for convenience.

4.4.1 Comparison with Bubble Sort

As discussed earlier, Bubble sort is the most old and traditional algorithm used for sorting. So we start our comparison from bubble sort.

	100	1000	5000	10000	15000	20000	30000	40000	50000	70000	90000	100000
Bubble Sort	0	15.625	328.125	1078.125	2359.375	4015.625	8968.75	15656.25	25656.23	48125	86250	100687.5
Relative Sort	0	0	109.375	343.75	578.125	968.75	1984.375	3281.25	9875	10031.25	16750	20953.13

Table 4.1: Comparison of Bubble Sort and Relative Sort

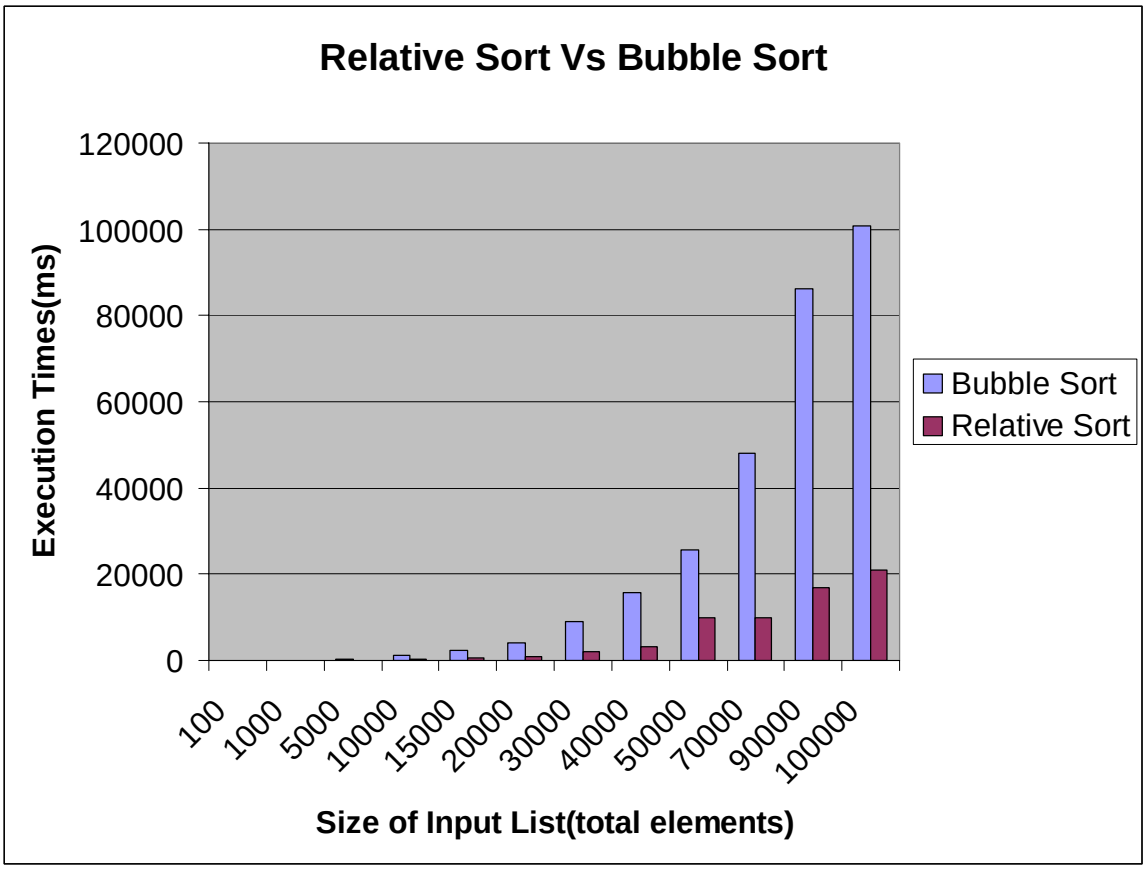


Figure 4.4: Graph (Relative sort Vs Bubble Sort)

In the above graph, at x-axis we have placed number of items in the list to be sorted and at y-axis we have placed the time taken by program for execution in milliseconds. It can be seen clearly that up to 5000 elements the difference is not much but when the size starts increasing, execution time of bubble sort also increases rapidly while relative sort and much more better performance that is obvious from the graph.

4.4.2 Comparison with Cocktail Sort

Cocktail sort as discussed earlier is an enhancement of bubble sort. After comparing performance of relative with bubble sort, we compared its performance with cocktail sort. Following table and graph is presented to analyze the contrast between the performances.

	100	1000	5000	10000	15000	20000	30000	40000	50000	70000	90000	100000
Cocktail Sort	0	15.625	343.75	765.625	1687.5	2828.125	6234.375	11062.5	17312.5	33468.75	55328.13	68468.75
Relative Sort	0	0	109.375	296.875	531.25	906.25	1953.125	3343.75	5078.125	9828.125	16359.38	20109.38

Table 4.2: Comparison of Cocktail Sort and Relative Sort

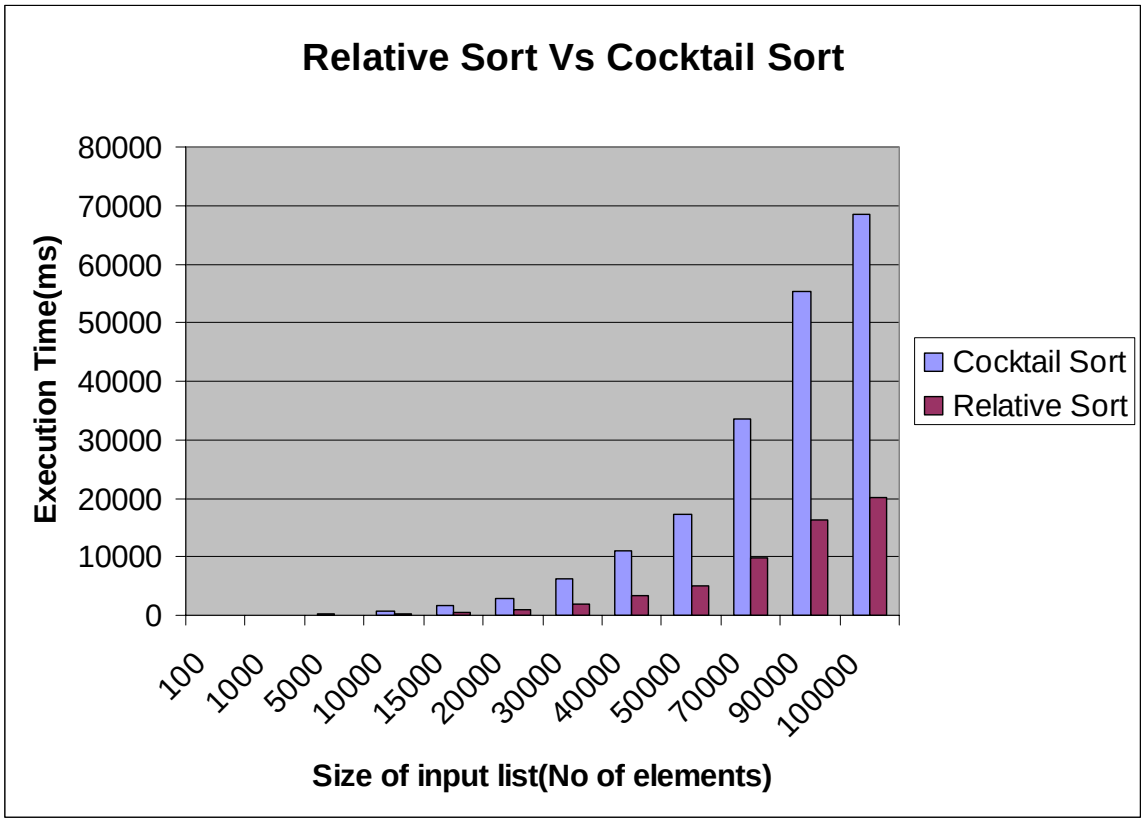


Figure 4.5: Graph (Relative sort Vs Cocktail Sort)

Again at x-axis are the numbers of items and along y-axis are the execution times in milliseconds. Like bubble sort the change in execution times is almost same but performance difference starts emerging when we move to larger lists.

4.4.3 Comparison with Selection Sort

Next performance was compared with selection sort. Following table and graph depicts the performance difference between selection sort and the proposed one.

	100	1000	5000	10000	15000	20000	30000	40000	50000	70000	90000	100000
Cocktail Sort	0	15.625	343.75	765.625	1687.5	2828.125	6234.375	11062.5	17312.5	33468.75	55328.13	68468.75
Relative Sort	0	0	109.375	296.875	531.25	906.25	1953.125	3343.75	5078.125	9828.125	16359.38	20109.38

Table 4.3: Comparison of Selection Sort and Relative Sort

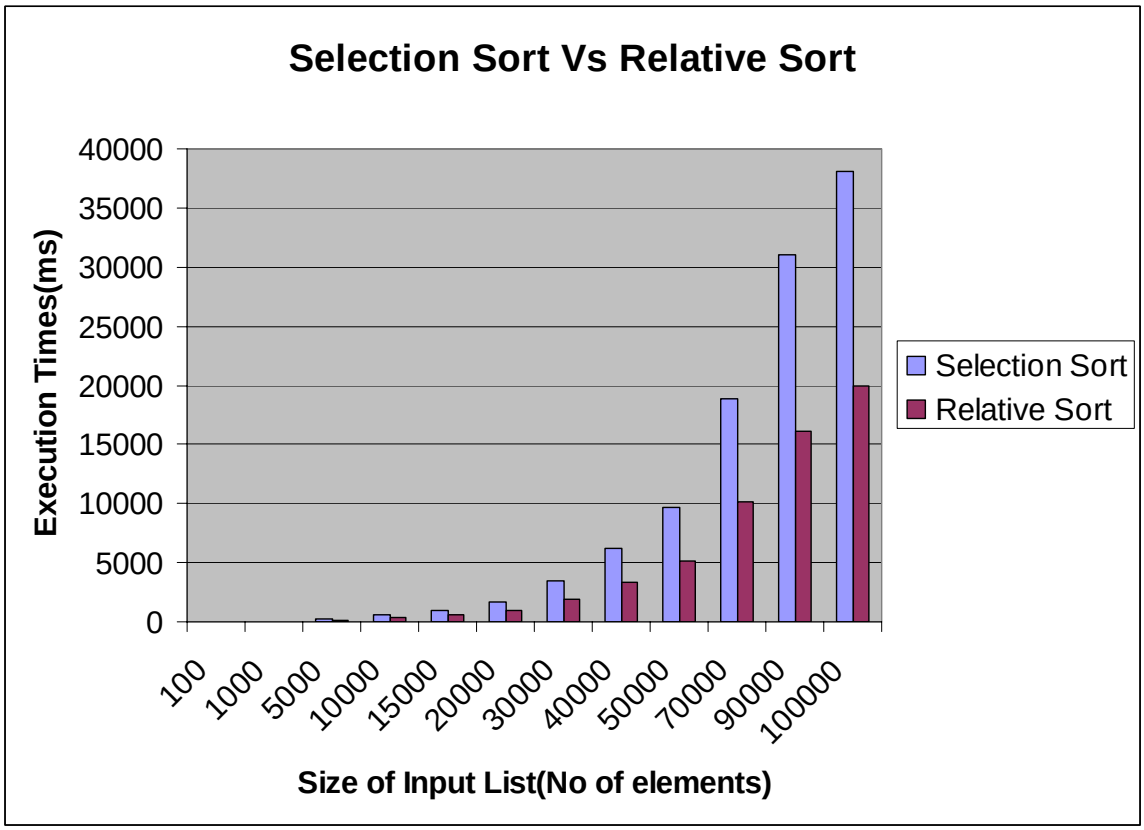


Figure 4.6: Graph (Relative sort Vs Selection Sort)

Above graph depicts the performance difference of Selection and the Relative sort. Along x-axis are the number of items in the input list while along y are the execution times. Relative sort is giving much lesser time of execution for lists almost larger than 1000 elements.

4.4.4 Comparison with Insertion Sort

Next we compared the performance of insertion sort with our algorithm. As discussed earlier that Relative sort belongs to the same family of sorting algorithms, so this comparison is important. Following table and graph shows the difference between the performances of both algorithms. From the table and graph, Relative sort is clearly proved as an improvement in the old insertion sort.

	100	1000	5000	10000	15000	20000	30000	40000	50000	70000	90000	100000
Insertion Sort	0	15.625	156.25	453.125	890.625	1453.125	3109.375	5390.625	8296.875	16250	26609.38	32796.88
Relative Sort	0	0	93.75	375	531.25	937.5	1968.75	3296.875	5140.625	10015.63	16234.38	20000

Table 4.4: Comparison of Insertion Sort and Relative Sort

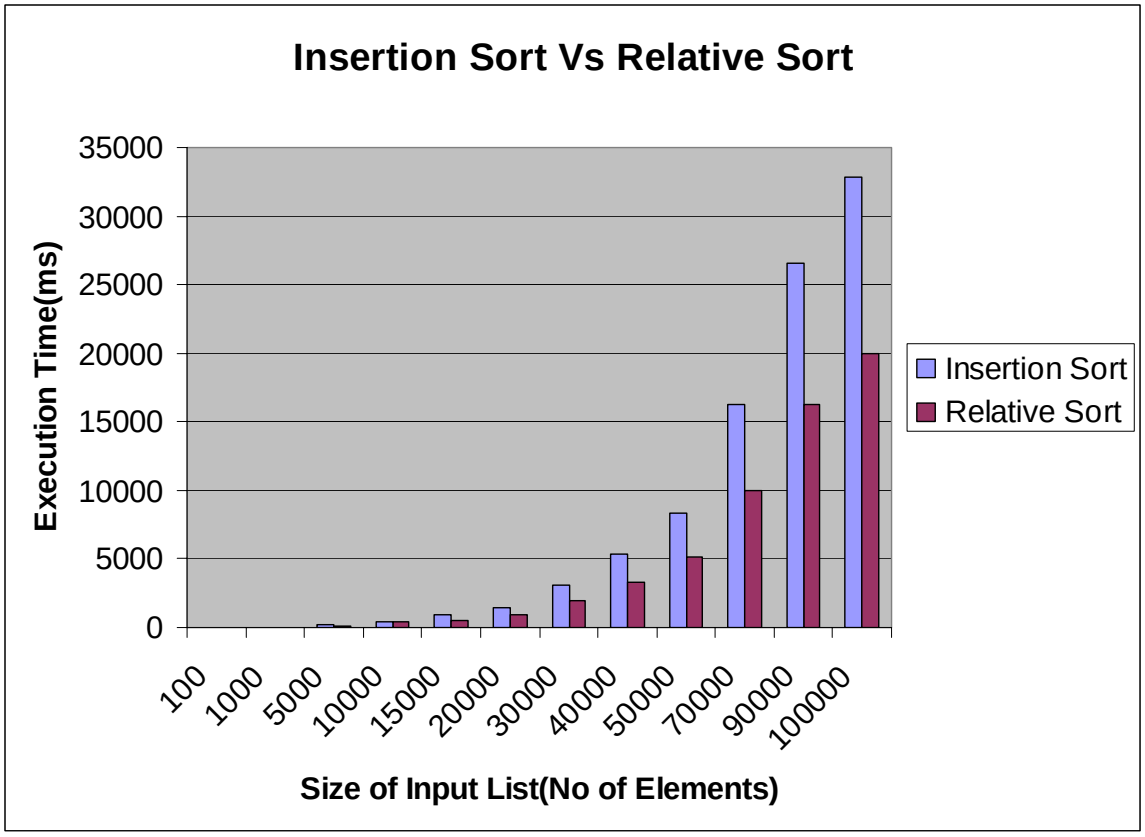


Figure 4.7: Graph (Relative sort Vs Insertion Sort)

Horizontally, numbers of input list elements are kept and along y is the execution time. Almost up to 10000 elements, the difference between the execution times is ignorable but graph presents a valuable difference when we keep on increasing input list size.

4.4.5 Comparison with Odd Even Sort

Next we compare the performance of Odd even sort with our algorithm. Following table and graph shows the difference between the performances of both algorithms. From the table and graph, Relative sort is clearly viewed much more efficient than Odd Even Sort.

	100	1000	5000	10000	15000	20000	30000	40000	50000	70000	90000	100000
Relative Sort	0	15.625	46.875	298.875	609.375	906.25	1921.875	3359.375	5156.25	9968.75	16343.75	20171.88
Odd Even Sort	0	15.625	328.125	1203.825	2500	4359.375	9640.625	16921.88	26500	52703.13	84796.88	106890.6

Table 4.5: Comparison of Insertion Sort and Relative Sort

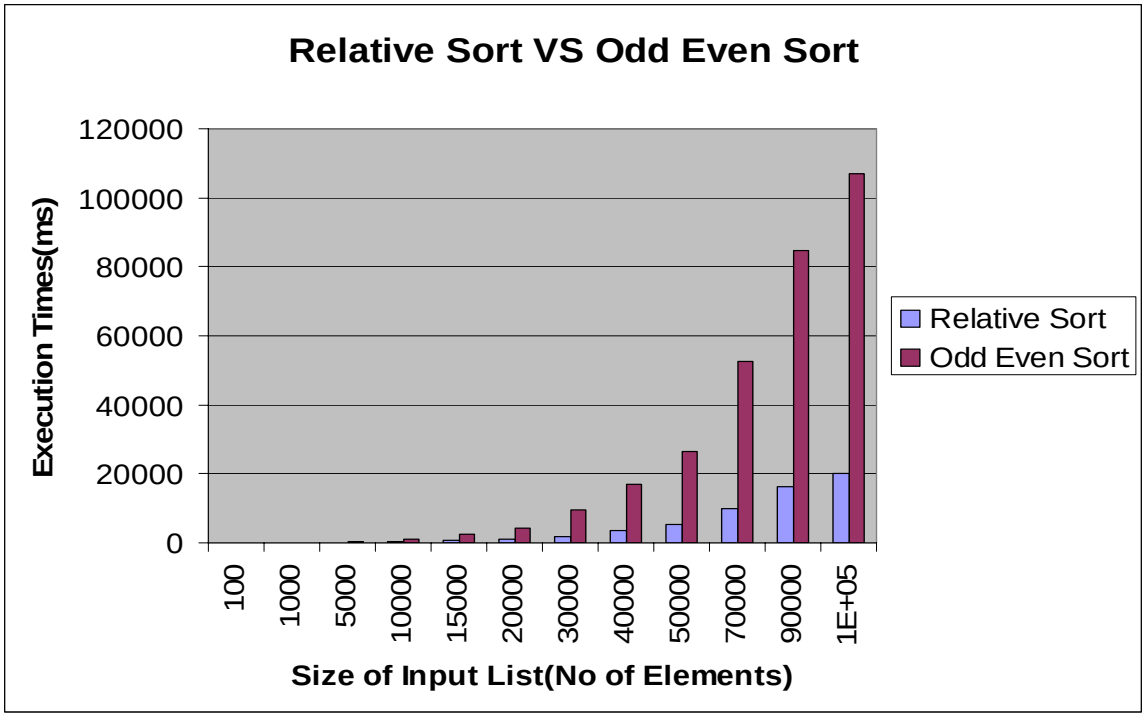


Figure 4.8: Graph (Relative sort Vs Odd Even Sort)

Horizontally, numbers of input list elements are kept and along y is the execution time. Almost up to 1000 elements, the difference between the execution times is ignorable but graph presents a valuable difference when we keep on increasing input list size.

4.4.6 Comparison with GNome Sort

Now the comparison of the performances of GNome sort and our algorithm are contrasted. Following table and graph shows the difference between the performances of both algorithms. From the table and graph, Relative sort is clearly viewed much more efficient than GNome Sort.

	100	1000	5000	10000	15000	20000	30000	40000	50000	70000	90000	100000
Relative Sort	0	15.625	109.375	312.5	609.375	921.875	1937.5	3343.75	5171.875	9968.75	16531.25	20296.88
GNome Sort	0	15.625	328.125	765.625	1625	2890.625	6234.375	11109.38	17531.25	34171.88	55593.75	71203.13

Table 4.6: Comparison of GNome Sort and Relative Sort

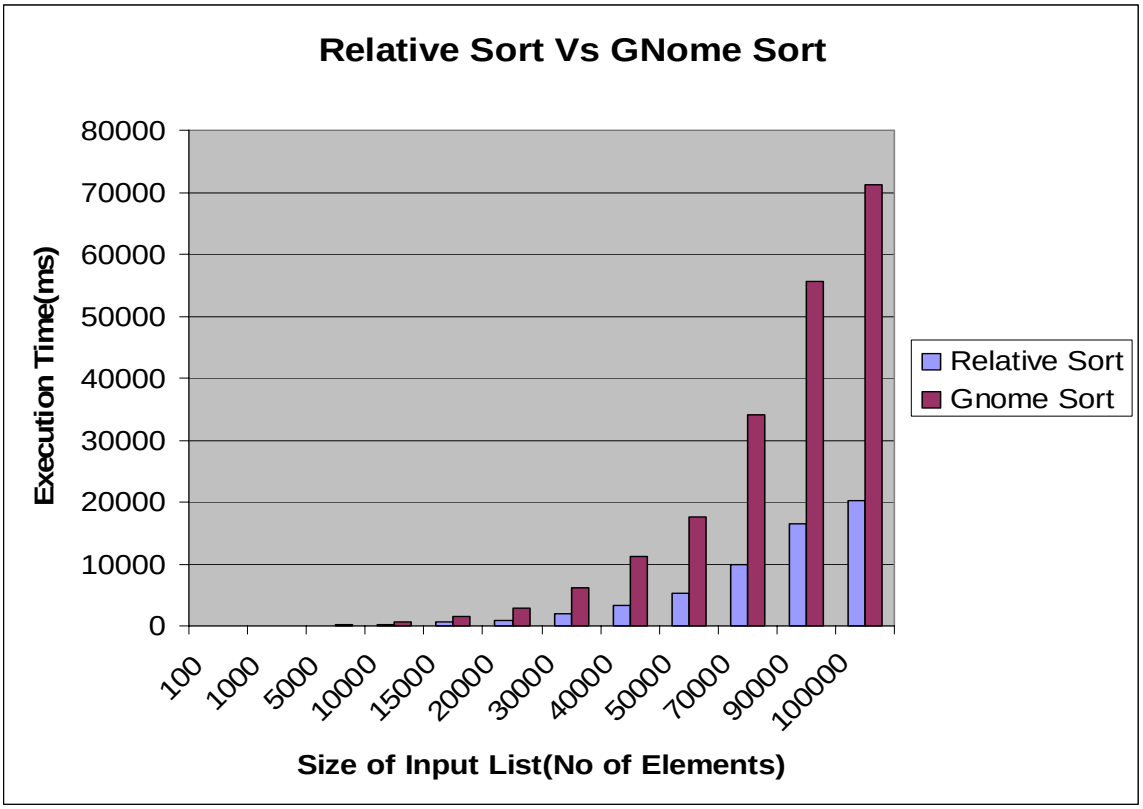


Figure 4.9: Graph (Relative sort Vs GNome Sort)

Horizontally, numbers of input list elements are kept and along y is the execution time. Almost up to 5000 elements, the difference between the execution times is ignorable but graph presents a valuable difference when we keep on increasing input list size.

4.4.7 Comparison with Shell Sort

Now the comparison of the performances of Shell sort and our algorithm are contrasted. Following table and graph shows the difference between the performances of both algorithms. From the table and graph, Relative sort is not as much efficient as Shell but here the fact should also be kept in mind that not always Shell sort guarantee that the output will be correctly sorted.

	100	1000	5000	10000	15000	20000	30000	40000	50000	70000	90000	100000
Shell Sort	0	0	140.625	281	484.375	437	640.625	984.375	1046.875	1625	2187.5	2406.25
Relative Sort	0	0	109.375	359	593.75	921.75	1937.5	3218.75	5156.25	10015.63	16328.25	20031.25

Table 4.7: Comparison of Shell Sort and Relative Sort

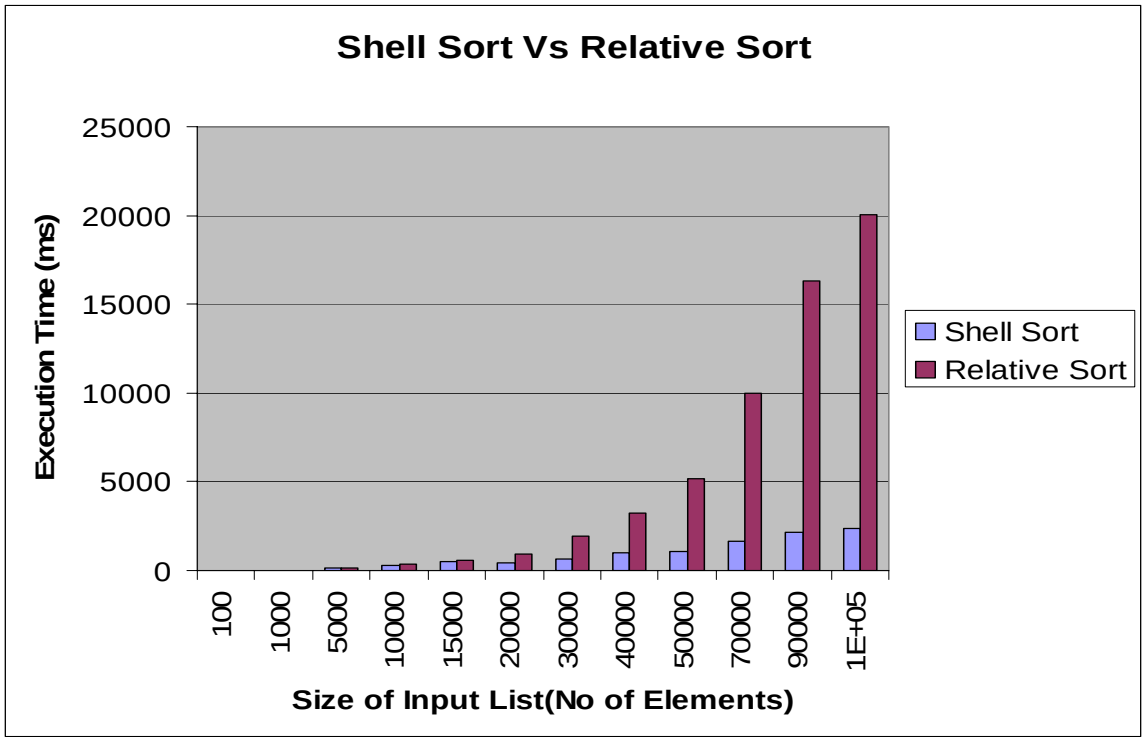


Figure 4.10: Graph (Relative sort Vs Shell Sort)

Horizontally, numbers of input list elements are kept and along y is the execution time. Almost up to 15000 elements, the difference between the execution times is ignorable but in case of larger lists, shell sort takes lesser time.

4.5 Summary:

To compare and contrast performance of the proposed Relative sort with existing well known sorting algorithms, all those algorithms were implemented along with proposed one. The purpose of these experiments was also to conclude where relative sort should be placed in the whole community of sorting algorithms. In the next chapter the whole discussion in concluded with further detail.

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

As discussed earlier, sorting is the elemental algorithmic problem in computer sciences. It is obvious that in any computational problem, searching and sorting is involved in any way and in any form. As also discussed earlier Donald Knuth, author of The Art of Computer Programming, wrote: “I believe that virtually every important aspect of programming arises somewhere in the context of searching or sorting”. So also from our experiences while working in the field of computer sciences, the importance of searching and sorting is obvious as awareness starts from very first day when one places his first step in the world of computer sciences and carries on like I am writing my MS Thesis on the same topic. The topic was chosen keeping this importance in mind. Basically I believe that if have a grip over these basic elemental ideas, you mind gets more creative, your way of analyzing problems, especially algorithmic problems improves in a very positive way.

The proposed idea is a new creation. It is a totally new idea. No relevance to any existing algorithms is there. The aim behind all this effort was to present a very efficient algorithm that may overcome limitations of existing algorithms. To some extent, the aim was achieved; a way has also been introduced, following which can cause new ideas to emerge.

The sort comparator developed can be further enhanced by plugging new algorithms and can be used as required. Execution time calculation technique is also quiet good and efficient.

Now coming towards the proposed methodology. From chapter 3, i.e. about the proposed algorithm, it can be viewed that the proposed algorithm is quiet simple.

Algorithm is very simple to understand, analyze and implement. A sample implementation has also been provided with intention to understand if any body wants.

From fourth chapter i.e. Implementation and Results, the performance and position of the proposed Relative Sort can be seen. The performance is not very good as the most efficient algorithms like Merge and Quick have beaten our algorithm with a very huge margin but all the others have been beaten by the proposed algorithm with a very large margin. As the proposed algorithm belongs to Insertion family so actual comparison of proposed algorithm was with the straight insertion sort and from the results provided, it can be clearly seen that the proposed method is really a good enhancement.

Summarizing the whole discussion, it is clear from the whole results that the proposed algorithm has got its position almost above than the middle order algorithms. As it has beaten the old algorithms and has been beaten by the most efficient algorithms. As it is an n^2 algorithm so if we only compare it with n^2 algorithms, we will see that it is almost among the best n^2 algorithms. It is first effort that can be viewed as a guide line and can be worked upon.

5.2 Future Work

As in the proposed idea, we take only arithmetic mean once and then fit elements on the base of comparison of that. This methodology can be made dynamic i.e. recursion may be involved so that in every call again arithmetic mean may be calculated and comparisons may be made. I hope this will be a more efficient way and will give a very good performance.

Also other ways also exist. The algorithm can be enhanced further in a number of ways. Any other existing methodology can be merged with this to get more efficient results. Any enhancement is appreciated and encouraged.

REFERENCES

[1] R. L.Kruse and A. J. Ryba, *Data Structures and Program Design inC++*, 2nd ed. Pearson Education, 1999.

[2] D. A. Bailey, *Java Structure:Data Structure in Java for Principled Programmer*, 2nd ed. McGraw-Hill, 2003.

[3] R. Sedgewick, *Algorithms in C++*. Reading, Massachusetts: Addison-Wesley, 1992.

[4] Knuth, D.E. The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, Reading Massachusetts, 1973.

[5] Knuth, D.E. The Art of Computer Programming, Volume 3: Sorting and Searching, First Edition, Page No. 14.

[6] http://www.pcmag.com/encyclopedia_term/0,2542,t=tag+sort&i=52532,00.asp

[7] Knuth, D.E. The Art of Computer Programming, Volume 3: Sorting and Searching, First Edition, Page No. 84.

[8] CACM 2 (July, 1959) 30-32

[9] Proc. AFIPCS Spring joint computer conference 32 (1968), 307-314.

[10] Comp.j.5 (1962) 10-15.

[11] http://en.wikipedia.org/wiki/Bubble_sort

[12] http://en.wikipedia.org/wiki/Cocktail_sort.htm

[13] http://en.wikipedia.org/wiki/Comb_sort

[14] http://en.wikipedia.org/wiki/Gnome_sort.htm

[15] Hoare, C. A. R. "Partition: Algorithm 63," "Quicksort: Algorithm 64," and "Find: Algorithm 65." Comm. ACM 4(7), 321-322, 1961

[16] http://en.wikipedia.org/wiki/Quicksort.htm

[17] http://en.wikipedia.org/wiki/Heapsort.htm

[18] http://en.wikipedia.org/wiki/ Library_sort.htm

[19] http://en.wikipedia.org/wiki/ Bead_sort.htm

[20] NIST's Dictionary of Algorithms and Data Structures: pigeonhole sort