A Comparative Analysis of Sorting Algorithms with focus on Merge Sort

Ashim Kumar Sikder

Student ID: 012131021

A Thesis

in

The Department

of

Computer Science and Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Science in Computer Science and Engineering

United International University

Dhaka, Bangladesh

February 2018

Declaration

This is to certify that the work entitled "**A Comparative Analysis of Sorting Algorithms with focus on Merge Sort**" is the outcome of the research carried out by me under the supervision of Dr. Mohammad Nurul Huda, Professor and MSCSE Coordinator, United International University (UIU), Dhaka, Bangladesh.

_____

Ashim Kumar Sikder
Department of Computer Science and Engineering
MSCSE Program
Student ID: 012131021
United International University (UIU)
Dhaka-1209, Bangladesh.

In my capacity as supervisor of the candidate's thesis, I certify that the above statements are true to the best of my knowledge.

_____

Dr. Mohammad Nurul Huda
Professor and MSCSE Coordinator
Department of Computer Science and Engineering
United International University (UIU)
Dhaka-1209, Bangladesh.

Abstract

In our thesis work, we try to find out the efficiency of several sorting algorithms and generate a comparative report according to performance, based on experimental data size and data order for all algorithm. To do this we have researched, analyzed about 9 different types of well-known sorting algorithms. We convert the algorithms to programable code, compile and run with different set of data. We keep the sorting algorithm's description as it is. We give focus on, how the algorithms work, considering their total operation count (assignment count, comparison count) and complexity based on our same data set for all algorithm. We write programming code for each sorting algorithm in C programming language. In our investigation, we have also worked with big and small data for different cases (ordered, preordered, random, disordered) and put their result in different tables. We show the increasing ratio to compare the result. we also show the data in graphical chart for view comparative report in same window. We mark their efficiency with point and ranked them. At last we discussed their result of efficiency in a single table.

We modify the merge sort and try to make an improved tri-merge sorting algorithm that is more efficient than marge sort. Theoretically if we divide and conquer with higher number its result is better, some paper exists on it, but to manage the algorithm, there cost lot of operations count. Like, if we consider quadratic divide-conquer, its manage complexly is huge than binary divide-conquer that why we generally use binary merge. We found tri-merge is theoretically and practically true based on investigation data set. Tri-marge take some more compare operation for manage and sort when data remain 1 or 2 at last stage, whereas binary merge don't need such compare. But for big data size tri-merge gain lot of operation count that give significant result that declare tri-merge is more efficient than merge sort algorithm. We also experiment with penta-merge algorithm which give more better result but algorithm and implementation is too complex.

We shall try to define the tri-merge algorithm so that it can be used to implement in any programming language. It will help students, researchers to use the algorithm, as like we got the various algorithm structure over the internet.

Acknowledgement

This thesis titled "**A Comparative Analysis of Sorting Algorithms with focus on Merge Sort**" has been prepared to fulfill the requirement of MSCSE degree. I am very much fortunate that I have received sincere guidance, supervision and co-operation from various persons. I would like to express my heartiest gratitude to my supervisor, **Dr. Mohammad Nurul Huda**, Professor and MSCSE Coordinator, United International University, for his continuous guidance, encouragement, and patience, and for giving me the opportunity to do this work. His valuable suggestions and strict guidance made it possible to prepare a well-organized thesis report.

Finally, my deepest gratitude and love to my parents and well-wisher for their support, encouragement, and endless love.

Table of Contents

Chapter 1

1.1. Introduction

The usefulness of an ordered set of elements is important that has impact on our daily life. For example, the finding a telephone number from a telephone directory. This process, called a **search,** is simplified considerably by the fact that the names in the directory are listed in the alphabetical order. Consider the trouble we might have in attempting to locate a telephone number if the names were listed in the order in which the customers placed their phone orders with the telephone company. In such a case, the names might as well have been entered in random order. Since the entries are sorted in alphabetical rather than in chronological order, the process of searching is simplified. Hence ordering the elements of a list is a problem that occurs in many contexts. **Sorting** refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabetically, with character data. Suppose that we have a list of elements of a set. Furthermore, suppose that we have a way to order elements of the set. A **sorting** is putting these elements into a list in which the elements are in increasing order or decreasing order. For instance, sorting the list

$$27, 12, 31, 65, 15, 93, 7$$

produces the list

$$7, 12, 15, 27, 31, 65, 93.$$

Suppose that we have a list of elements of a set. Furthermore, suppose that we have a way to order elements of the set. A sorting is putting these elements into a list in which the elements are in increasing (or decreasing) order.

Let A be a list of n elements $A_1, A_2, \ldots, A_n$ in memory. Sorting A refers to the operation of rearranging the contents of A so that they are increasing in order (numerically or lexicographically), that is, so that

$$A_1 \leq A_2 \leq A_3 \leq \ldots \leq A_n .$$

Since A has n elements, there are n! ways that the contents can appear in A. These ways correspond precisely to the n! permutations of 1, 2,…, n. Accordingly, each sorting algorithm must take care of these n! possibilities.

There are various sorting algorithms such as:

- Insertion sort
- Bubble sort
- Replacement sort
- Selection sort
- Quick sort
- Merge sort
- Shell sort
- Binary Tree sort
- Tournament sort
- Heap sort
- Radix sort
- Address calculation sort

1.2. Classification of Sorting:

A sort can be classified as being **internal** if the records that it is sorting are in main memory, or **external** if some of the records that it is sorting are in auxiliary storage. e.g. bubble sort is internal and merge sort is external type. In our thesis work we discuss the following type sorting based on internal and external.

Internal type Sort: Bubble, Insertion, Selection, Replacement, Heap, Quick, Shell.
External type Sort: Radix, Merge, Tri-merge (New)

A sort also can be classified as **recursive type** if recursive function (the function which calls itself in the function) is used in the implementation of this sorting algorithm and otherwise **non-recursive type** sort e.g. insertion sort is non-recursive and quick is recursive type sort.

In our analysis, we discuss the following type sorting based on Recursive type and Non-recursive type

Non-recursive type Sort: Bubble, Insertion, Selection, Replacement, Heap, Radix, Shell

Recursive type Sort: Quick, Merge, Tri-merge.

In our thesis work we have tried to present all these types of sorting algorithms in an easy implementation.

In our thesis work we analysis around 9 types of different sorting algorithms, their implementations, how it works, total operation counts (assignment count, comparison count), its complexity and how their work can be developed. Among these one of the sorting algorithm out come from our own analysis. We named it **Tri-merge** sorting algorithm.

1.3. Complexity of Sorting Algorithm:

The complexity of a sorting algorithm measures the running time as a function of the number n of items to be sorted. We note that each sorting algorithm will be made up of the following operations, where A_1, A_2, ... , A_n contain the items to be sorted and B is an auxiliary location:

Comparisons, which test whether $A_i < A_j$ or test whether $A_i < B$.

Interchanges, which switch the contents of A_i and A_j or of A_i and B.

Assignments, which set $B_i = A_i$ and then set $A_j = B$ or $A_j = A_i$

Normally, the complexity function measures only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

There are two main cases whose complexity we will consider; the worst case and the average case. In studying the average case, we make the probabilistic assumption that all the n! permutations of the given n times are equally likely. For different sorting algorithms, the approximate number of comparisons and the order of complexity of these algorithms are summarized in the following table:

Table of Complexity of the algorithm

Algorithm	Worst Case	Average case
Bubble Sort	$N(N-1)/2 = O(N^2)$	$N(N-1)/2 = O(N^2)$
Selection Sort	$N(N-1)/2 = O(N^2)$	$N(N-1)/2 = O(N^2)$
Insertion Sort	$N(N-1)/2 = O(N^2)$	$N(N-1)/4 = O(N^2)$
Replacement Sort	$N(N-1)/2 = O(N^2)$	$N(N-1)/2 = O(N^2)$
Shell Sort	$N*(\text{Log } N)^2 = O(N*(\text{Log } N)^2)$	$N*(\text{Log } N)^2 = O(N*(\text{Log } N)^2)$
Heap Sort	$3(N*\text{Log } N) = O(N*\text{Log } N)$	$3(N*\text{Log } N) = O(N*\text{Log } N)$
Radix Sort	$O(N^2)$	$O(N*\text{Log } N)$
Quick Sort	$N(N+3)/2 = O(N^2)$	$1.4(N*\text{Log } N) = O(N*\text{Log } N)$
Merge Sort	$N*\text{Log } N = O(n \log_2 n)$	$N*\text{Log } N = O(n \log_2 n)$
Tri-Merge Sort	$N*\text{Log } N = O(n \log_3 n)$	$N*\text{Log } N = O(n \log_3 n)$
Penta-Merge Sort	$N*\text{Log } N = O(n \log_5 n)$	$N*\text{Log } N = O(n \log_5 n)$

Table-1

Chapter 2

In this chapter, we derive about 9 numbers of sorting algorithms. To illustrate the sorting algorithms, we try to use exactly same set of data, which will be easy to understand and compare each other.

For experimental data, we have shown total number of operation count for 600 data with ordered, preordered, random type then we also discuss their merits and drawbacks as well as complexity. For all sorting algorithm, we sort data ascending ordered.

2.1.1. Bubble Sort:

2.1.1.1. Description

The bubble sort is simple sorting algorithms, It is less efficient. It puts a list into increasing order by successively comparing adjacent elements, interchanging them if they are in wrong order. To carry out the bubble sort, we perform the basic operation, which is interchanging a larger element with a smaller one following it, starting at the beginning of the list for a full pass. We iterate this procedure until the sort is complete. In the bubble sort, the smaller element 'bubble' to the top as they are interchanged with larger elements. The larger elements 'sink' to the bottom. We can imagine the elements in the list placed in a column.

Suppose the list of numbers A[1], A[2], ..., A[n] is in memory. The bubble sort algorithm works as follows:

Step 1:

Compare A[1] and A[2] arrange them in their desired order, so that A[1] < A[2]. Then compare A[2] and A[3] and arrange them so that A[2] < A[3].Then compare A[3] and A[4] and arrange them so that A[3] < A[4]. Continue until we compare A[n-1] with A[n] and arrange them so that A[n-1] < A[n].

Observe that Step-1 involves n-1 comparisons (During Step-1, the largest element is 'bubbled up' to the n-th position or 'sinks' to the n-th position) when Step-1 is complicated, A[n] with contain the largest element.

Step 2:

Repeat Step-1 with one less comparison; that is, now we stop after we compare and possibly rearrange A[n-2] and A[n-1]. (Step-2 involves n-2 comparisons and when Step-2 is completed, the second largest element will occupy A[n-1]).

Step 3:

Repeat Step-1 with two fewer comparisons, that is we stop after we compare and possible rearrange A[n-3] and A[n-2].

… …

… …

Step n-1:

Compare A[1] with A[2] and rearrange them so that A[1] < A[2].

After n-1 steps the list will be sorted in increasing order.

The process of sequentially traversal through all or part of a list is frequently called a ‘Pass’, so each of the above steps is called a pass. Accordingly, the bubble sort algorithm requires n-1 passes when n is the number of input items.

2.1.1.2. Example:

Suppose we have an array A[1,…12] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58

Now we apply the Insertion algorithm to sort the array in ascending order. Each pass of the sorting algorithm are shown below:

```
Input :
 46 30 82 90 56 17 95 15 48 26 4 58

Pass-1:  30  46  82  56  17  90  15  48  26  4  58  95
Pass-2:  30  46  56  17  82  15  48  26  4  58  90  95
Pass-3:  30  46  17  56  15  48  26  4  58  82  90  95
Pass-4:  30  17  46  15  48  26  4  56  58  82  90  95
Pass-5:  17  30  15  46  26  4  48  56  58  82  90  95
Pass-6:  17  15  30  26  4  46  48  56  58  82  90  95
Pass-7:  15  17  26  4  30  46  48  56  58  82  90  95
Pass-8:  15  17  4  26  30  46  48  56  58  82  90  95
Pass-9:  15  4  17  26  30  46  48  56  58  82  90  95
Pass-10:  4  15  17  26  30  46  48  56  58  82  90  95
Pass-11:  4  15  17  26  30  46  48  56  58  82  90  95
Pass-12:  4  15  17  26  30  46  48  56  58  82  90  95

Output:
 4 15 17 26 30 46 48 56 58 82 90 95
```

Every pass of Bubble sort

2.1.1.3. Algorithm:

```
BubbleSort( a[0,....,n-1])
{
        FOR( i = n-1 TO 0 STEP -1 )
        {
                FOR( j = 1 TO i STEP 1 )
                {
                        IF( a[j-1] > a[j] ) THEN SWAP( a[j-1],a[j] )
                }
        }
}
```

For **600** Data total operation of compare and assignment count for various cases of **Bubble** Sort:

Bubble Sort	100% Order	80% Ordered	10% Order	Random	Completely Disorder
Total operation Count	563169	601914	713247	820536	882461

Table-2

After analysis Bubble sort for various cases we can conclude that:

1. Algorithm is very easy to understand but not so efficient.
2. It needs nearly same number of operations for all four cases random, partially ordered, partially disorder and completely disordered data.
3. Rationally total number of operations increases rapidly.

2.1.2. Insertion Sort:

2.1.2.1. Description

Insertion sort is a simple sorting algorithm, but it is usually not very efficient. To sort a list with n elements the insertion sort begins with the second element. The insertion sort compares the second element with the first element and inserts it before the first element, if it does not exceed the first element and after the first element if it exceeds the first element. At this point the first two elements are in the correct order. The third element is then compared with the second element and if it is smaller than second element, it is compare with the first element; it is inserted into the corrected position among the first three elements. In general, the j-th step of the insertion sort, the j-th element of the list is inserted into the correct position in the list of the previously sorted (j-1) elements. The algorithm continues until the last element is placed in the correct position relative to the already sorted list of the first (n-1) element.

Suppose an array A with n elements A[1], A[2], ..., A[n] is in memory. The insertion algorithm scans A from A[1] to A[n], inserting each element A[k] into its proper position in the previously sorted sub array A[1], A[2],..., A[k-1], i.e.

Step-1:

A[1] by itself is trivially sorted.

Step-2:

A[2] is inserted either before or after A[1] so that: A[1], A[2] is sorted.

Step-3:

A[3] is inserted into its proper in A[1], A[2] that is before A[1], between A[1] & A[2] or after A[2] so that: A[1], A[2], A[3] are sorted.

Step-4:

A[4] is inserted in A[1], A[2], A[3] so that: A[1], A[2], A[3], A[4] is sorted.

… …

… …

Step-n:

A[n] is inserted into its proper place in A[1], A[2], A[3],..., A[n-1] so that A[1], A[2],..., A[n] is sorted.

The sorting algorithm is frequently used when n is small. For example, this algorithm is very popular with bridge players when they are first sorting their cards.

There remains only the problem of deciding how to insert A[k] in its proper place in the sorted sub array A[1], A[2], ..., A[k-1]. This can be accomplished by comparing A[k], with A[k-1], comparing A[k] with A[k-2], comparing A[k] with A[k-3], and so on, until the first meeting an element A[j] such that A[j] ≤ A[k]. Then each of the elements A[k-1], A[k-2], ..., A[j+1] is moved forward one location, and A[k] is then inserted in the (j+1)-th position in the array.

2.1.2.2. Example:

Suppose we have an array A[1,…12] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58

Now we apply the Insertion algorithm to sort the array in ascending order. Each pass of the sorting algorithm are shown below:

```
Input :
 46 30 82 90 56 17 95 15 48 26 4 58

Pass-1:   30 46 82 90 56 17 95 15 48 26 4 58
Pass-2:   30 46 82 90 56 17 95 15 48 26 4 58
Pass-3:   30 46 82 90 56 17 95 15 48 26 4 58
Pass-4:   30 46 56 82 90 17 95 15 48 26 4 58
Pass-5:   17 30 46 56 82 90 95 15 48 26 4 58
Pass-6:   17 30 46 56 82 90 95 15 48 26 4 58
Pass-7:   15 17 30 46 56 82 90 95 48 26 4 58
Pass-8:   15 17 30 46 48 56 82 90 95 26 4 58
Pass-9:   15 17 26 30 46 48 56 82 90 95 4 58
Pass-10:   4 15 17 26 30 46 48 56 82 90 95 58
Pass-11:   4 15 17 26 30 46 48 56 58 82 90 95

Output:
 4 15 17 26 30 46 48 56 58 82 90 95
```

Every pass of Insertion sort

2.1.2.3. Algorithm:

```
InsertionSort( a[0,....,n-1]
{
        FOR( i =1 TO n-1 STEP -1 )
        {
                index = a[i]
                j = i
                WHILE( j > 0 AND a[j-1] > index )
                {
                        a[j] = a[j-1]
                        j = j-1
                }
                a[j] = index
```

```
        }
}
```

For **600 Data** total operation of compare and assignment count for various cases of **Insertion Sort:**

Insertion Sort	**100% Order**	**80% Ordered**	**10% Order**	**Random**	**Completely Disorder**
Total operation Count	33487	85147	233591	376643	706742

Table-3

After the analysis of insertion sort we can conclude that:

1. Its algorithm is fairly easy to understand.
2. It works very well for ordered and partially ordered data.
3. It does not work well for Random and completely disordered data.
4. Rationally total number of operations increases rapidly.

2.1.3. Selection Sort:

2.1.3.1. Description

Selection sort is one in which successive elements are selected in order and placed into their proper sorted positions. The elements of the input may have to be preprocessed to make the ordered selection possible. Any selection sort can be described as the following general algorithm that uses a descending order.

Suppose, an array A with n elements A[1], A[2],.....,A[n] is in memory. The selection sort algorithm for sorting works as follows. First find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it in the second position and so on more precisely.

Pass 1:

Find the location (= mPos) of the smallest in the list of n elements A[1], A[2], ... , A[n] and then interchange, A[mPos], and A[1]. Then, A [1] is sorted.

Pass 2:

Find the location mPos of the smallest in the sub list of n-1 elements A[2], A[3], ... , A[n] and the interchange A[mPos] and A[2]. Then A[1], A[2] is sorted, since A[1] ≤ A[2]

Pass 3:

Find the location mPos of the smallest in the sub list of n-2 elements A[3], A[4],....., A[n] and then interchanging A[mPos] and A[3]. Then A[1], A[2], A[3] are sorted , since A[2] ≤ A[3]

... ...

... ...

Pass n-1:

Find the location mPos of the smallest of the elements A[n-1], A[n], and then interchange A[mPos] and A[n-1] then A[1], A[2] ,....., A[n] is sorted , since A[n-1] ≤ A[n] . Then A is sorted after n-1 passes.

2.1.3.2. Example:

Suppose we have an array A[1,…,12] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58

Now we apply the Selection algorithm to sort the array in ascending order. Each pass of the sorting algorithm is shown below:

```
Input :
 46 30 82 90 56 17 95 15 48 26 4 58

Selection
Pass-1:   4 30 82 90 56 17 95 15 48 26 46 58
Pass-2:   4 15 82 90 56 17 95 30 48 26 46 58
Pass-3:   4 15 17 90 56 82 95 30 48 26 46 58
Pass-4:   4 15 17 26 56 82 95 30 48 90 46 58
Pass-5:   4 15 17 26 30 82 95 56 48 90 46 58
Pass-6:   4 15 17 26 30 46 95 56 48 90 82 58
Pass-7:   4 15 17 26 30 46 48 56 95 90 82 58
Pass-8:   4 15 17 26 30 46 48 56 95 90 82 58
Pass-9:   4 15 17 26 30 46 48 56 58 90 82 95
Pass-10:   4 15 17 26 30 46 48 56 58 82 90 95
Pass-11:   4 15 17 26 30 46 48 56 58 82 90 95
Pass-12:   4 15 17 26 30 46 48 56 58 82 90 95

Output:
 4 15 17 26 30 46 48 56 58 82 90 95
```

Every Pass of Selection Sort

2.1.3.3. Algorithm:

```
SelectionSort( a[0,....,n-1]
{
        FOR( i = 0 TO n-1 STEP 1 )
        {
                min = i
                FOR( j = i +1 TO n-1 STEP 1 )
                {
                        IF( a[j] < a[min] ) THEN min = j
                }
                SWAP( a[min], a[i] )
        }
}
```

Applying the selection sort algorithm to A yields the data. Observe that mPos gives the location of the smallest among A[k], A[k+1],..., A[n] during pass k. The bolded elements indicated the elements, which are to be interchanged. The Process is shown in the table as follows:

For **600** Data total operation of compare and assignment count for various cases of **Selection** Sort:

Selection Sort	**100% Order**	**80% Ordered**	**10% Order**	**Random**	**Completely Disorder**
Total operation Count	542976	543359	544263	545218	619878

Table-4

After the analysis of Selection Sort we can conclude that:

1. Number of operation count is about same for random, preordered data.
2. Rationally total number of operations increases rapidly.

2.1.4. Replacement sort:

2.1.4.1. Description:

In replacement sort, we find the smallest element and put it into the first element of the array. Then the second smallest element has to be found and place it into the 2nd position of the array. The procedure is going on until the full array is sorted.

Suppose an array A with n elements A[1], A[2], A[3], …, A[n] is in memory. Replacement sort algorithm for sorting A works as follow: first find the smallest element in the list and put it in the first position, then find the second element in the list and put it in the second position, and so on.

Pass-1:

The first element of the array has to be compared from the 2nd element to last element of the array. When the element is greater than another element then they will interchange their position. So that we have to find such j that is A[1] > A[j] then they swap themselves.

Pass-2:

Now the 2nd element of the array has to be compared with the 2nd, 3rd …. n-th element. When 2nd element greater than another element then they will interchange their position.

Pass-3:

In this way the process is going on up to the last element of the array.

2.1.4.2. Example:

Suppose we have an array A[1,…12] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58

Now we apply the Selection algorithm to sort the array in ascending order. Each pass of the sorting algorithm are shown below:

```
Input :
 46 30 82 90 56 17 95 15 48 26 4 58

Replacement
Pass-1:   4 46 82 90 56 30 95 17 48 26 15 58
Pass-2:   4 15 82 90 56 46 95 30 48 26 17 58
Pass-3:   4 15 17 90 82 56 95 46 48 30 26 58
Pass-4:   4 15 17 26 90 82 95 56 48 46 30 58
Pass-5:   4 15 17 26 30 90 95 82 56 48 46 58
Pass-6:   4 15 17 26 30 46 95 90 82 56 48 58
Pass-7:   4 15 17 26 30 46 48 95 90 82 56 58
Pass-8:   4 15 17 26 30 46 48 56 95 90 82 58
Pass-9:   4 15 17 26 30 46 48 56 58 95 90 82
Pass-10:   4 15 17 26 30 46 48 56 58 82 95 90
Pass-11:   4 15 17 26 30 46 48 56 58 82 90 95
Pass-12:   4 15 17 26 30 46 48 56 58 82 90 95

Output:
 4 15 17 26 30 46 48 56 58 82 90 95
```

Every Pass of Replacement Sort

2.1.4.3. Algorithm:

```
ReplacementSort( a[0,....,n-1]
{
        FOR( i = 0 TO n-1 STEP 1 )
        {
                FOR( j = i +1 TO n-1 STEP 1 )
                {
                        IF( a[j] < a[i] ) THEN SWAP( a[j],a[i] )
                }
        }
}
```

For **600** Data total operation of compare and assignment count for various cases of **Replacement** Sort:

Replacement Sort	**100% Order**	**80% Ordered**	**10% Order**	**Random**	**Completely Disorder**
Total operation Count	551178	566412	600645	615657	653454

Table-5

After the analysis of Replacement sort we can conclude that:

1. The number of total operation count is about same for random and preordered data.
2. Rationally total number of operations increases rapidly.

2.1.5. Shell Sort:

2.1.5.1. Description:

In this method elements are sort among themselves of sub-files. Sub-files are created taking the elements with a fixed interval of the original file. Firstly interval is taken 5 and sort among themselves. Again interval is taken 3 and sort among themselves. Finally taking interval 1 sub-file are sorted and we get fully ordered list.

For example, if the original file is

46 30 82 90 56 17 95 15 48 26 4 58

and the sequence interval(span) is chosen 5,3,1 the following sub-files are sorted on each iteration.

First iteration (interval = 5)

(x[0], x[5], x[10])

(x[1], x[6] , x[11])

(x[2], x[7] , x[12])

(x[3] , x[8])

(x[4] , x[9])

Second iteration (interval =3)

(x[0], x[3], x[6] , x[9] , x[12])

(x[1], x[4], x[7] , x[10])

(x[2], x[5] , x[8] , x[11])

Third iteration (interval =1)

(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7] , x[8], x[9], x[10], x[11], x[12])

The above-discussed sub-files are sorted after each iteration which is shown below in the figure-1

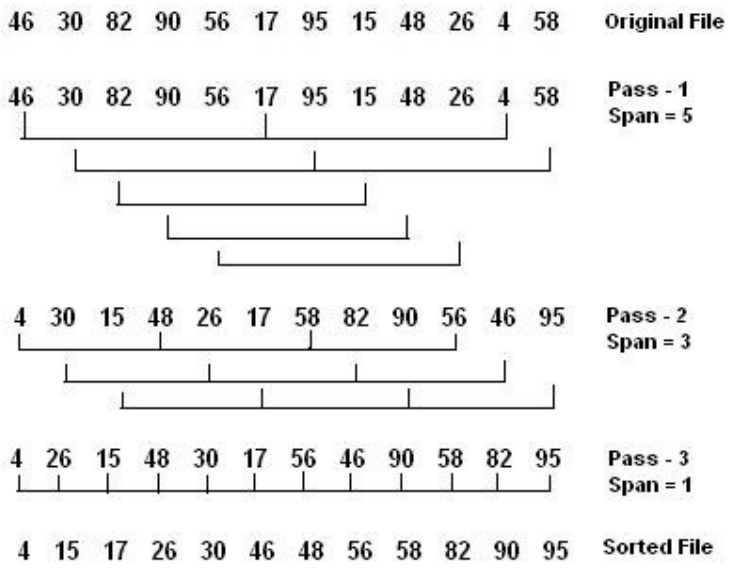


Figure – 1

Figure – 1 illustrates the Shell sort on this sample file. The lines underneath each array join individual elements of the separate subfiles. Each of the subfiles is sorted using the simple insertion sort.

2.1.5.2. Example:

Suppose we have an array A[1,…,9] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48

Now we apply the Shell algorithm to sort the array in ascending order. Each pass of the sorting algorithm are shown below:

```
 46 30 82 90 56 17 95 15 48

Shell Sort
Span-5      17 30 82 90 56 46 95 15 48
Span-5      17 30 82 90 56 46 95 15 48
Span-5      17 30 15 90 56 46 95 82 48
Span-5      17 30 15 48 56 46 95 82 90
Span-3      17 30 15 48 56 46 95 82 90
Span-3      17 30 15 48 56 46 95 82 90
Span-3      17 30 15 48 56 46 95 82 90
Span-3      17 30 15 48 56 46 95 82 90
Span-3      17 30 15 48 56 46 95 82 90
Span-3      17 30 15 48 56 46 95 82 90
Span-1      17 30 15 48 56 46 95 82 90
Span-1      15 17 30 48 56 46 95 82 90
Span-1      15 17 30 48 56 46 95 82 90
Span-1      15 17 30 48 56 46 95 82 90
Span-1      15 17 30 46 48 56 95 82 90
Span-1      15 17 30 46 48 56 95 82 90
Span-1      15 17 30 46 48 56 82 95 90
Span-1      15 17 30 46 48 56 82 90 95

Output:
 15 17 30 46 48 56 82 90 95
```

Every Pass of Shell Sort

2.1.5.3. Algorithm:

```
ShellSort( a[0,….,n-1] )
{
        set number of increment and span value in array incrs[i]
        FOR( in = 0 TO numinc –1 )
        {
                span = incrs[in];
                FOR( j = span TO n-1 )
                {
                        y = a[j];
                        FOR( k = j-span TO 0 AND y < a[k] STEP -span)
                        {
                                a[k+span]=a[k]
                        }
```

```
                    a[k+span] = y
                }
            }
        }
```

For **600** Data total operation of compare and assignment count for various cases of **Shell** Sort:

Shell Sort	**100% Order**	**80% Ordered**	**10% Order**	**Random**	**Completely Disorder**
Total operational Count	15847	28447	60851	89331	910844

Table-6

After the analysis of Shell sort, we can conclude that

1. It works very good for order and partially ordered data (i.e. For random and disordered data it needs more operations.
2. It works well for small data.

2.1.6. Heap Sort:

2.1.6.1. Description:

Heap sort uses binary tree structure technique. In tree structure first element of array is called parent node and every parent node can contain two elements, which is called child node. Similarly, each child node also can contain two elements and behave as parent node. In this way all the elements of array a[0,…….,n] are arranged in a tree structure like in Figure-2

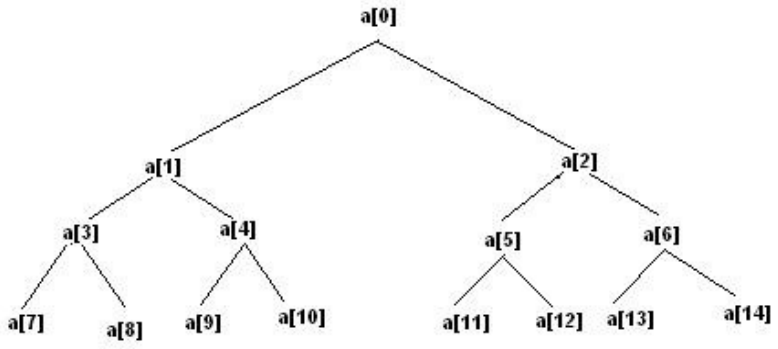
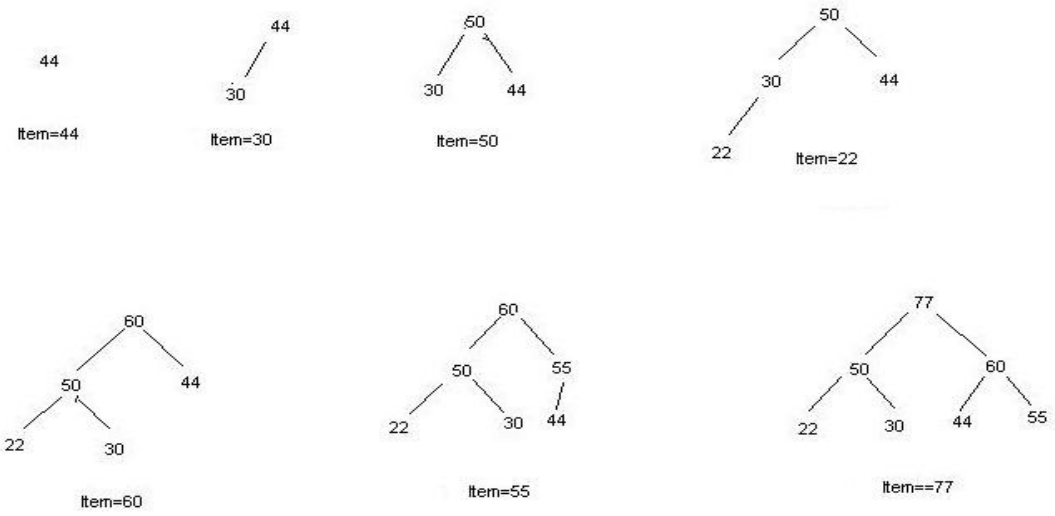


Figure-2

Heap sort works in two phases:

Phase –1:

In this phase the tree structure maintain heap property. Heap property means every parent element will be greater or smaller than child elements according ascending or descending order. Maintaining heap property we get maximum/minimum value of all elements at position index 0. The figure shows that how the elements arrange in the tree structure and maintain heap property.



At the last stage we will get the elements in the array like:

A[0] = 77 a[1] = 50 a[2] = 60 a[3] = 22

a[4] = 30 a[5] = 44 a[6] = 55

Phase –2:

In this phase heap technique replace extreme value from position 0 to correct position. In this example at position 6, and set last element 55 to position 0. This time system further maintain heap property up to 1,….,5 position and get second extreme element at position 0. Similarly second extreme value replace with position 0 and 5. This way phase –2 sort all the data.

2.1.6.2. Example:

Suppose we have an array A[1,…12] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58

Now we apply the Heap algorithm to sort the array in ascending order. Each pass of the sorting algorithm are shown below:

```
Input :
 46 30 82 90 56 17 95 15 48 26 4 58

After Maintaing Heap property
 95 82 90 48 56 58 46 15 30 26 4 17
pass-1  :   90 82 58 48 56 17 46 15 30 26 4 95
pass-2  :   82 56 58 48 26 17 46 15 30 4 90 95
pass-3  :   58 56 46 48 26 17 4 15 30 82 90 95
pass-4  :   56 48 46 30 26 17 4 15 58 82 90 95
pass-5  :   48 30 46 15 26 17 4 56 58 82 90 95
pass-6  :   46 30 17 15 26 4 48 56 58 82 90 95
pass-7  :   30 26 17 15 4 46 48 56 58 82 90 95
pass-8  :   26 15 17 4 30 46 48 56 58 82 90 95
pass-9  :   17 15 4 26 30 46 48 56 58 82 90 95
pass-10  :   15 4 17 26 30 46 48 56 58 82 90 95
pass-11  :   4 15 17 26 30 46 48 56 58 82 90 95

Output:
 4 15 17 26 30 46 48 56 58 82 90 95
```

Every Pass of Heap Sort

2.1.6.3. Algorithm:

HeapSort(a[0,....,n-1]

{

FOR(i = 0 TO n-1 STEP 1)

{

cValue = a[i]

ptr = I

head = (ptr – 1)/2

WHILE(ptr > 0 AND a[head] <cValue)

{

a[ptr] = a[head]

ptr = head

head = (ptr –1)/2

}

a[ptr] = cValue

```
        }

        FOR( i = n-1 TO 1 STEP -1 )
        {
                iValue = a[i]
                a[i] = a[0]
                head = 0
                IF (i =1 ) THEN
                        prt = -1
                ELSE
                        prt = 1

                IF (i > 2 AND a[2] >a[1] ) THEN prt = 2

                WHILE( ptr >= 0 AND iValue < a[ptr] )
                {
                        a[head]=a[ptr]
                        head=ptr
                        ptr=2*head+1
                        IF( ptr +1 <= i-1 AND a[ptr] < a[ptr+1] ) THEN ptr = ptr +1
                        IF( ptr > i-1 ) THEN ptr = -1
                }
                a[head]=iValue
        }
}
```

For **600** Data total operation of compare and assignment count for various cases of **Heap** Sort:

Heap Sort	**100% Order**	**80% Ordered**	**10% Order**	**Random**	**Completely Disorder**
Total operation Count	53348	51977	48836	46987	432346

Table-7

After the analysis of Heap sort, we can conclude that:

1. It takes nearly same operations for random, partially ordered, partially disorder and completely disordered.
2. It works good for random than ordered data.
3. The Algorithm of this sorting hard for understand.

2.1.7. Radix Sort:

2.1.7.1. Description

Radix Sort is interesting sorting algorithm among all sort. It takes less number of operation and take less time among all sort. We assume that all the numbers have the same number of digits, by padding with leading zeros, if necessary.

In this method first make 10 pockets indexing form 0 to 9. The process completes the sorting after maximum number of digit passes.

At first pass each number is pick into indexed pocket according unit position value of the number. Now all the numbers are collected to an array list starting from 0 indexed pocket. At second pass again, each number is pick into indexed pocked according tens position value of the number. Further all the numbers are collected to an array list similar way as earlier. Third pass is complete according hundreds position. Similarly rest of the pass will continue up to maximum number of digits. At last pass we will get sorted list in the array.

For clear understand how radix sort works, let us suppose we have an array with 9 elements of 3 digits number like:

348, 143, 361, 423, 538, 128, 321, 543, 366

Every pass is shown in the table. Where first column is treated as array list and next following columns are pocket with indexed from 0 to 9.

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

First pass

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366							366			
348					348					
538				538						
128			128							

Second pass

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538										
143		143								
543						543				
348				348						
361				361		538				
366				366						

Third pass

In the third and final pass, the hundred digits are sorted into pockets. When the numbers are collected after the third pass, the numbers are in the following order:

128, 143, 321, 348, 361, 366, 423, 538, 543

Thus, the numbers are now sorted.

2.1.7.2. Example:

Suppose we have an array A[1,…12] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58

Now we apply the Radix algorithm to sort the array in ascending order. Each pass of the sorting algorithm are shown below:

```
Input :
 46 30 82 90 56 17 95 15 48 26 4 58

Radix Sort
Busket-0:  4
Busket-1:  15 17
Busket-2:  26
Busket-3:  30
Busket-4:  46 48
Busket-5:  56 58
Busket-6:
Busket-7:
Busket-8:  82
Busket-9:  90 95

Output:
 4 15 17 26 30 46 48 56 58 82 90 95
```

Every Pass of Radix Sort

2.1.7.3. Algorithm:

```
RadixSort ( a[0,….n-1], maxNoDigit )
{
        Declare: tempArray[10][n/5], busketNo[10]
        FOR( m = 1 TO maxNoDigit STEP 1 )
        {
                set all busket[k] = 0
                FOR( i = 0 TO n-1 STEP 1 )
                {
                        dig = (a[i]/POW(10,m-1)) MOD 10
                        busketNo[dig] = busketNo[dig] + 1
                        tempArray[dig][busketNo[dig]] = a[i]
                }
                ind = 0
                FOR( j = 0 TO 9 STEP 1 )
                {
                        FOR( k = 0 TO busketNo[j] STEP 1 )
                        {
                                ind = ind + 1
                                a[ind] = tempArray[j][k]
                        }
                }
        }
}
```

For **600** Data total operation of compare and assignment count for various cases of **Radix** Sort:

Radix Sort	**100% Order**	**80% Ordered**	**10% Order**	**Random**	**Completely Disorder**
Total operation Count	7306	7306	7306	7306	7306

Table-8

After the analysis of Radix sort, we can conclude that:

1. It takes exactly same operations for random, partially ordered, partially disorder and completely disordered data
2. Maximum digit number of values should be known firstly.
3. It works very well for same number of digit.
4. It works comparatively fast than any other sort.
5. Algorithm is easy to understand

2.1.8. Quick sort:

2.1.8.1. Description:

Quick sort algorithm is divide and conquer type algorithm. Let x be an array, and n is the number of elements in the array to be sorted. Choose an element a from a specific position within the array (for example, pivot can be chosen as the first element of the array so that pivot = x[0]). Now all the elements are divide in two parts by setting elements less than pivot in one side and greater than pivot in another side. This way we get pivot element in right position in the array. This process again applies to each part recursively excluding pivot element until one element at each side.

Quick sort example are describe with bellow example:

Let us say A is the following list of 8 numbers 44 33 11 55 77 90 40 60. Here 44 is pivot element which divide into two parts 33 11 40 and 60 90 77 55 according less than 44 and greater than 44. Here we get right position for 44 as index 3. Again part-1 divides as 11 and 40 accordingly. Similarly other part divides and finally gets them as sort order.

44	33	11	55	77	90	40	60

33 11 40 **44** 60 90 77 55

11 **33** 40 **44** 55 **60** 77 90

11 33 40 44 55 60 77 90

11 33 40 44 55 60 77 90

2.1.8.2. Example:

Suppose we have an array A[1,…,12] which are given below

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58

Now we apply the Quick algorithm to sort the array in ascending order. Each pass of the sorting algorithm are shown below:

```
Input :
 46 30 82 90 56 17 95 15 48 26 4 58

Pivot:46   4 30 26 15 17 46 95 56 48 90 82 58
Pivot:4   4 30 26 15 17
Pivot:30   17 26 15 30
Pivot:17   15 17 26
Pivot:15   15
Pivot:26   26
Pivot:95   58 56 48 90 82 95
Pivot:58   48 56 58 90 82
Pivot:48   48 56
Pivot:56   56
Pivot:90   82 90
Pivot:82   82

Output:
 4 15 17 26 30 46 48 56 58 82 90 95
```

Every Pass of Quick Sort

2.1.8.3. Algorithm:

```
QuickSort( a[L,....,R], L, R )
{
        set pivot = a[L]
        set all values less than pivot to Left side of the array
        set all values greater than pivot to Right side of the array
        find pivot position index
        IF( L < index ) THEN
        {
                QuickSort( a[L,....,index-1], L, index-1 )

        }
        IF( R > index ) THEN
        {
                QuickSort( a[index+1,....,R], index+1, R )

        }
}
```

For **600** Data total operation of compare and assignment count for various cases of **Quick** Sort:

Quick Sort	**100% Order**	**80% Ordered**	**10% Order**	**Random**	**Completely Disorder**
Total operation Count	80090	52127	29874	29405	80546

Table-9

After the analysis of Quick Sort, we can conclude that:

1. It works very well for Random data.
2. For ordered and disordered data it works bad
3. Rationally total number of operations increases slowly.
4. This algorithm is very difficult to understand

2.1.9. Merge Sort:

2.1.9.1. Description

Merge sort is used as divide and conquer technique. As for example, A[0,…,n] is an array list with n elements. The merge sort algorithm uses recursive function technique to sort a list of elements. Let we have two sorted sub-files. Now comparing the first element from two sub-files take smallest or biggest one to a temporary file. Again, comparing remaining first element from two sub-files take it to temporary file. Similarly, all the elements will come to temporary file as a sorted list. The process is called merge. The Complete process illustrate by an example given below:

Suppose the array A contains 14 elements as follows:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58, 47, 2

The procedure of Merge sort complete in two phases.
Phase-1: Split in to two parts recursively
Phase-2: Merge

Split:

In this stage total data of the given array split into 2 parts. Each part consequently split 2 parts recursively until 1 elements remain. The split is shown below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58, 47, 2

46, 30, 82, 90, 56 17, 95 15, 48, 26 4, 58, 47, 2

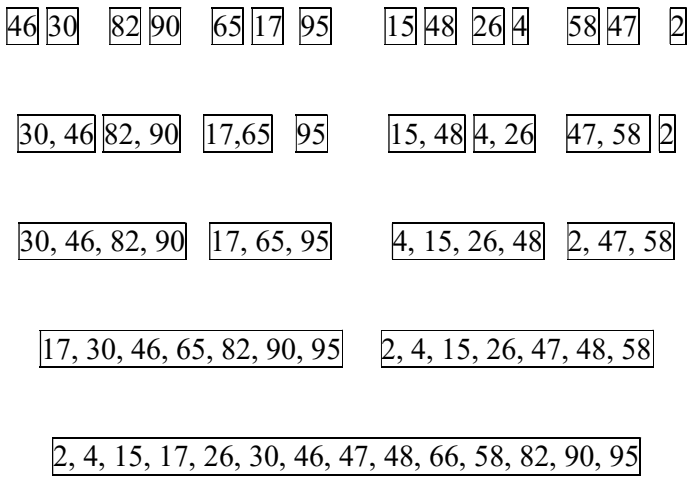
46, 30, 82, 90 65, 17, 95 15, 48, 26, 4 58, 47, 2

46, 30 82, 90 65, 17 95 15, 48 26, 4 58, 47 2

46 30 82 90 65 17 95 15 48 26 4 58 47 2

Merge:

In this phase every 2 split parts become 1 part and sort among themselves. This process continues until all data become one part and finally we get completely sorting data. The Merge process are given bellow:



2.1.9.2. Example:

Suppose we have an array A[1,…12] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58

Now we apply the Merge algorithm to sort the array in ascending order. Each pass of the sorting algorithm are shown below:

:

```
Input :
 46 30 82 90 56 17 95 15 48

Pre Merge       : 46 30
After Merge     : 30 46
Pre Merge       : 30 46 82
After Merge     : 30 46 82
Pre Merge       : 90 56
After Merge     : 56 90
Pre Merge       : 30 46 82 56 90
After Merge     : 30 46 56 82 90
Pre Merge       : 17 95
After Merge     : 17 95
Pre Merge       : 15 48
After Merge     : 15 48
Pre Merge       : 17 95 15 48
After Merge     : 15 17 48 95
Pre Merge       : 30 46 56 82 90 15 17 48 95
After Merge     : 15 17 30 46 48 56 82 90 95

Output:
 15 17 30 46 48 56 82 90 95
```

Every Pass of Merge Sort

2.1.9.3. Algorithm:

MergeSort(a[L,....,R], L, R)

{

IF (L<R) THEN

{

mid = (F+L)/2

MergeSort(a[L,….,mid], L, mid)

MergeSort(a[mid+1,….,R], mid+1, R)

Merge(a[L,….,R], L, mid+1, R)

}

}

Merge(a[L,….,R], L, mid , R)

{

```
        part1 = a[L,…,mid-1]
        part2 = a[mid,…,R]
        TempArray[L,…,R]
        WHILE( part1 and part2 has elements )
        {
                Comparing from 2 parts find minimum and set to Temporary array.
        }

        WHILE( part1 has elements )
        {
                set to Temporary array.
        }
        WHILE( part2 has elements )
        {
                set to Temporary array.
        }

        FOR ( i = L TO R STEP 1)
        {
                Set all TempArray[i] value to a[i]
        }
        RETURN
}
```

For **600** data total operation of compare and assignment count for various cases of **Merge** Sort:

Merge Sort	**100% Order**	**80% Ordered**	**10% Order**	**Random**	**Completely Disorder**
Total operation Count	58799	59565	60017	60129	59234

Table-10

After the analysis of Merge sort we can conclude that:

1. It takes nearly same operations for random, partially ordered, partially disorder and completely disordered.
2. It works well for ordered data.
3. Rationally total number of operations increases slowly.
4. Algorithm is difficult to understand.

Chapter 3

3.1. Analysis of efficiency of various sorting algorithms

In this chapter, we try to find out efficiency of sorting algorithm. Efficiency of sorting algorithm depends on how quickly (how much time) data can sort. And time depends on how many operation (assignment and compare) count require for the algorithm. That is why we find out assignment and compare count for each algorithm and arrange it in the table after summation. We find out operation count for 100% ordered, 80% ordered, 10% ordered, random, completely disordered data.

To test the algorithm technique and get operation count we use C programming language. We use Mathematica Software to generate graph (data-operation count).

For 50 Data total operation count with different ordered:

Sort Name	100% Order	80% Ordered	10% Order	Random	Completely Disorder
Bubble	3775	4153	5305	5335	7405
Insertion	245	749	2285	2325	5085
Selection	3975	4010	4071	4095	4445
Replacement	3775	4108	5086	5026	6154
Heap	2899	2773	2532	2479	2145
Shell	582	838	1186	1270	1622
Radix	706	706	706	706	706
Quick	4172	1615	1355	1204	4147
Merge	3163	3195	3205	3209	3145
Tri-Merge (New)	2078	2441	2560	2596	2443

Table-11

For 200 Data total operation count with different ordered:

Sort Name	100% Order	80% Ordered	10% Order	Random	Completely Disorder
Bubble	60601	68011	81640	91915	117584
Insertion	1663	11543	29715	43415	80344
Selection	61067	61114	61316	61600	67631
Replacement	60601	64906	72451	76552	81537
Heap	15622	15098	13521	13008	12093
Shell	2827	5859	10423	13727	14420
Radix	2506	2506	2506	2506	2506
Quick	33568	13833	8227	6337	33045
Merge	16437	16633	16825	16885	16233
Tri-Merge (New)	12458	13445	14536	14744	13990

Table-12

For 400 Data total operation count with different ordered:

Sort Name	100% Ordered	80% Ordered	10% Order	Random	Completely Disorder
Bubble	242900	267395	316460	365282	400252
Insertion	5595	38255	103675	168771	304534
Selection	242046	242336	242718	243444	274562
Replacement	241997	255140	274163	285164	349343
Heap	35271	32321	30898	29379	270642
Shell	6347	14535	29331	42971	448428
Radix	4906	4906	4906	4906	4906
Quick	76852	20832	19554	14717	75951
Merge	36385	37171	37659	37767	36007
Tri-Merge (New)	29967	32804	34667	34852	30182

Table-13

For 600 Data total operation count with different ordered:

Sort Name	100% Order	80% Ordered	10% Order	Random	Completely Disorder

Bubble	563169	601914	713247	820536	882461
Insertion	33487	85147	233591	376643	706742
Selection	542976	543359	544263	545218	619878
Replacement	551178	566412	600645	615657	653454
Heap	53348	51977	48836	46987	432346
Shell	15847	28447	60851	89331	910844
Radix	7306	7306	7306	7306	7306
Quick	80090	52127	29874	29405	80546
Merge	58799	59565	60017	60129	59234
Tri-Merge (New)	50107	53015	54521	54805	52015

Table-14

For 800 Data total operation count with different ordered:

Sort Name	100% Order	80% Ordered	10% Order	Random	Completely Disorder
Bubble	9226764	1063453	1259566	1447693	2756473
Insertion	509393	141399	402883	653719	619815
Selection	964759	964760	965474	966869	967801
Replacement	823641	1005562	1049428	1065151	1076530
Heap	73640	70835	68372	65851	64712
Shell	18494	41607	100366	148267	152850
Radix	9706	9706	9706	9706	9706
Quick	82110	54429	46866	37285	38002
Merge	80945	82205	83143	83455	84013
Tri-Merge (New)	72984	75943	79611	79993	79732

Table-15

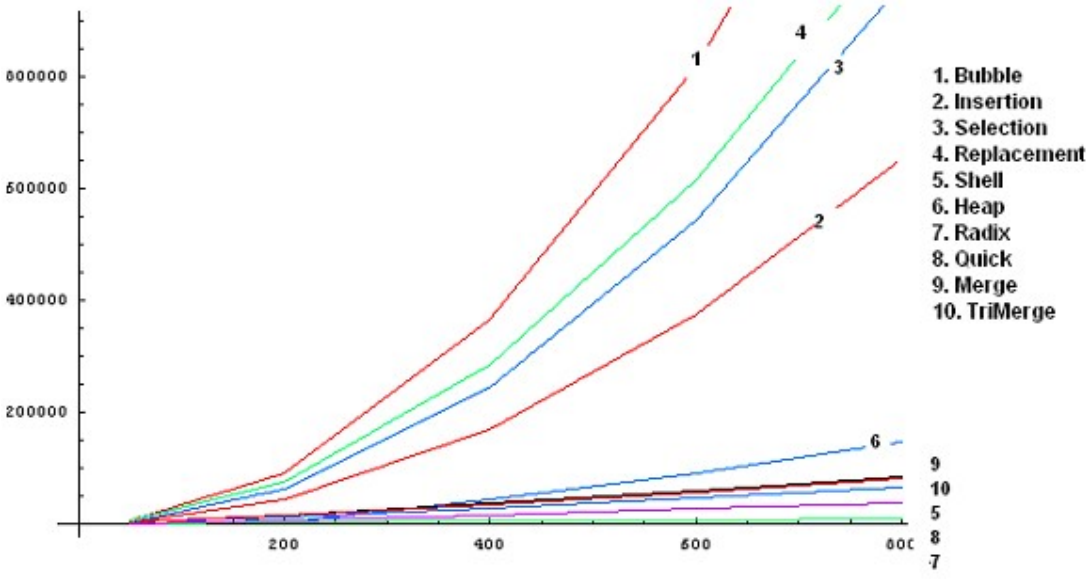
3.2. Comparative result in one table

Increasing Ratio of operation count for increasing data with random order data.

Sort Name	**50**	**200**	**400**	**600**	**800**	**increasing ratio**
Bubble	5335	91915	365282	820536	1447693	
		17.22	3.97	2.24	1.76	Ratio
Insertion	2325	43415	168771	376643	653719	
		18.67	3.88	2.23	1.73	Ratio
Selection	4095	61600	243444	545218	966869	
		15.04	3.95	2.23	1.77	Ratio
Replacement	5026	76552	285164	615657	1065151	
		15.23	3.72	2.15	1.73	Ratio
Heap	2479	13008	29379	46987	65851	
		5.24	2.25	1.59	1.40	Ratio
Shell	1270	13727	42971	89331	148267	
		10.80	3.13	2.07	1.65	Ratio
Radix	706	2506	4906	7306	9706	
		3.54	1.95	1.48	1.32	Ratio
Quick	1204	6337	14717	29405	37285	
		5.26	2.32	1.99	1.26	Ratio
Merge	3209	16885	37767	60129	83445	
		5.26	2.23	1.59	1.38	Ratio
Tri-Merge (New)	2596	14744	34852	54805	79993	
		5.67	2.36	1.57	1.45	Ratio

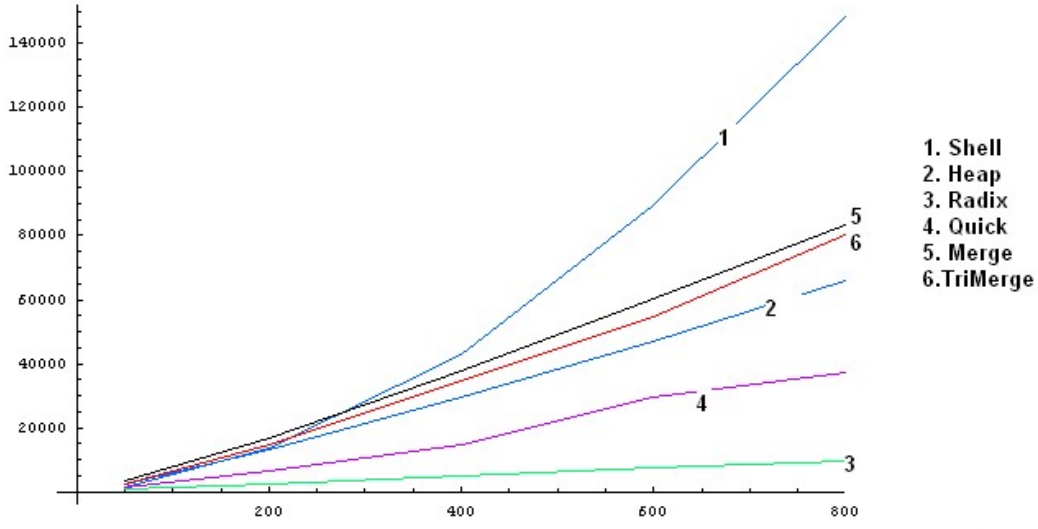
Table-16

3.3. Comparative result in graph



Graph-1

The comparative Graph of Number of Data and operation count for 10 Sort



Graph-2

The comparative Graph of Number of Data and operation count for 6 efficient Sort

3.4. Comparative Analysis:

From our analysis, we observed that:

1. Bubble sort is a very slow way of sorting. Its main advantage is the simplicity of the algorithm.
2. Insertion is much efficient for preordered and small data. For disorder and big data it is less efficient.
3. Quick sort is bad for fully ordered and fully disordered data. It works very well for random data.
4. Bubble, Selection, Replacement, Merge, Heap, has fairly effect for order or random data.
5. Radix has no effect for order or random data. For all type data its efficiency is remarkable. If number of digit goes very high, there need very high computing power.
6. Insertion and Shell sort are good for semi ordered data and very bad for disorder data

3.5. Efficiency Ranking:

As per our observation we find out the ranking according efficiency among the type of sorting we used.

Rank Name	Rank	Description
Very Good	1	Radix sort give very good result. It is number one in ranking. It gives comparatively less number of operations among all sorts but it takes extra memory for process.
Good	2	Quick sort is second rank
Fairly Good	3	Shell sort is third rank
Fairly Bad	4	In ranking Tri-Merge is fourth and Merge sort is in fifth position
Bad	5	Then heap, insertion, selection, replacement sort respectively
Very bad	6	Bubble sort is very bad

Table-17

3.6. Computation & Resource Limitation:

1. We use earlier version of C compiler.
2. We use Limited memory size computer.
3. Due to one or both of the above did not support sort for big data.

Chapter 4

We modify the merge sort and try to make an improved tri-merge sorting algorithm that is more efficient than marge sort. Theoretically if we divide and conquer with higher number its result is better, some paper exists on it (ref: https://doi.org/10.1016/S0898-1221(98)00158-8), but more than 3 divides are more costing (operation count) to manage the algorithm. If we consider quadratic divide-conquer, its manage complexly is huge than binary divide-conquer that is why we generally use binary merge. We found tri-merge is theoretically and practically better based on investigation data set. Tri-marge take some more compare operation for manage and sort when data remain 1 or 2 at last stage where binary merge don't need such compare. But for big data size tri-merge gain lot of operation count that give significant result that declare tri-merge is more efficient than merge sort algorithm.

Our aim is to make the tri-merge algorithm so that it can be used to implement in any programming language.

4.0. Improved Sorting Algorithm Tri-Merge & Penta-Merge

4.1. Background

In this paper, we design a new merge sorting algorithm named Ternary-Merge sort, which is competitive with the fastest sorting algorithms, especially when the number of elements to be sorted is too large. Compared with the traditional Marge sort algorithm, our Ternary-Merge sort algorithm is more robust, which indicates that it avoids extreme slowdowns on plausible inputs. From this research work it is shown that the time complexity of our Ternary-Merge sort requires O($n\log_3 n$) and Penta-Merge sort require O($n\log_5 n$) in comparison with the traditional Merge sort, which requires O($\mathrm{n}\log_2 \mathrm{n}$). The empirical results show that our simple algorithm is robust and faster than the traditional Merge sort algorithm. For a large input data, we implement this algorithm in well-known programming language C and Java.

Keywords: Ternary-Merge sort, Sorting algorithms, Quick sort.

4.2. Introduction

In this study, we have proposed a merge sorting algorithm with different implementation named Ternary-Merge sort, which is comparable with the fastest sorting algorithm than merge, especially when the number of input elements to be sorted is large. Comparing with the traditional Marge sort algorithm, our Ternary-Merge sort algorithm is better, which indicates that it avoids extreme slowdowns on plausible inputs. From this research it is observed that the time complexity of our Ternary-Merge sort requires O($n \log_3 n$) and Penta-Merge sort require O($n \log_5 n$) in comparison with the traditional Merge sort, which requires O($\mathrm{n} \log_2 \mathrm{n}$). The experimental results show that our ternary-merge algorithm faster than the traditional binary merge sorting algorithm. For a large input data, we implement this algorithm in well-known programming language C and Java.

4.3. Traditional Binary Merge Sorting Algorithm

Suppose we have n elements. The traditional Merge sort algorithm uses a recursive algorithm to sort a list of these n elements. This algorithm divides the list of elements until each smaller portion of the list gets the number of element one and then merges these smaller portions to get a solution for the whole list. Let we have two sorted sub-files. Now comparing the first element from two sub-files take smallest or biggest one to a temporary file. Again, comparing remaining first element from two sub-files it is taken to the temporary file. All the elements will come to the temporary file as a sorted file. The process is called merge. Now instead of two sub-files we make three sub-files and apply the merging technique on them. Strictly speaking, we worked with ternary three structures and were able to construct structure, which proved to be more efficient than the existing one. Since the algorithm sort data by merging from three files we named it Ternary-Merge sorting algorithm.

4.4. Proposed Ternary-Merge Sorting Algorithm

Section 4.4.1 and Section 4.4.2 show the formulation of our proposed Ternary Merge Sorting algorithm and its algorithm, respectively. The total number of elements is (high-low+1) in this algorithm. It is also based on "divide and conquer" strategy. Just like as traditional method it also divides the whole list of n elements into smaller parts with one element each, but merging is done with the three smaller parts at each step.

4.4.1. Formulation

Ternary-Merge is a new proposed technique for sorting data. It uses the attractive features of the sorting methods so far discussed: use of recursive functions and efficiency of merge sorting. Most of the sorting technique generally uses binary tree structure whereas Ternary-Merge uses ternary tree structure, which proves to be more effective than the binary one.

The procedure of Ternary-Merge sort is completed in two phases.
Phase-1: Split in 3 parts recursively.
Phase-2: Merge to 1 part from 3 parts

Split:

In this stage total data of the given array split into 3 parts. Each part consequently split 3 parts recursively until 1 or 2 elements remain. When two data remain in a part they are sorted among themselves.

Merge:

After split merge step will start. In this phase every 3 split parts become 1 part and are sorted among themselves. This process continues until all data become one part and finally we get completely sorted data.

Example:

Suppose we have an array A [1, … , 12] which are given below:

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58, 47, 2

Now we apply the Ternary-Merge algorithm to sort the array in ascending order. Here split and merge step are described in the list

Split **(Phase-1)**

46, 30, 82, 90, 56, 17, 95, 15, 48, 26, 4, 58, 47, 2

46, 30, 82, 90, 56 | 17, 95, 15, 48, 26 | 4, 58, 47, 2

46, 30 | 82, 90 | 56 | 17, 95 | 15, 48 | 26 | 4, 58 | 47 | 2

Merge (Phase-2)

30, 46 | 82, 90 | 56 | 17, 95 | 15, 48 | 26 | 4, 58 | 47 | 2

30, 46, 56, 82, 90 | 15, 17, 26, 48, 95 | 2, 4, 47, 58

2, 4, 15, 17, 26, 30, 46, 47, 48, 66, 58, 82, 90, 95

4.4.2. Tri-Merge Algorithm

Here is the structural algorithm for easy implementation to the programming language.

```
TernaryMergeSort (a [L,.., R], L, R)
{
        n=R - L+1
        IF (n > 2) THEN
        {
                m1 = n/3
                m2 = 2*m1
                TernaryMergeSort (a [L, …., m1], L, m1)
                TernaryMergeSort (a [m1+1, …., m2], m1+1, m2)
                TernaryMergeSort (a [m2+1, …., R], m2+1, R)
                TernaryMerge (a [L, …., R], L, m1+1, m2+1, R)
        }
        ELSE IF (n = 2) THEN
        {
                IF (a[L] > a[R])
                {
                        temp = a[L]
                        a[L] = a[R]
                        a[R] = temp
                }
        }
}
```

```
TernaryMerge (a [L, …., R], L, m1, m2, R)
{
        part1 = a [L, …, m1-1]
        part2 = a [m1, …, m2-1]
        part3 = a [m2, …, R]
        TempArray[L, …, R]
        n = R - L+1
```

```
IF ( n > 2)
{
        WHILE (part1, part2 and part3 has elements)
        {
                Comparing from 3 parts find minimum and set to Temporary array.
        }
        WHILE (part1 and part2 has elements)
        {
                Comparing from 2 parts find minimum and set to Temporary array.
        }
        WHILE (part2 and part3 has elements)
        {
                Comparing from 2 parts find minimum and set to Temporary array.
        }
        WHILE (part1 and part3 has elements)
        {
                Comparing from 2 parts find minimum and set to Temporary array.
        }
        WHILE (part1 has elements)
        {
                set to Temporary array.
        }
        WHILE (part2 has elements)
        {
                set to Temporary array.
        }
        WHILE (part3 has elements)
        {
                set to Temporary array.
        }
        FOR ( i = L TO R STEP 1)
        {
                Set all TempArray[ i ] value to a[ i ]
```

```
                }
                RETURN
        }
}
```

4.4.3. Time Complexity Analysis

For n elements input data and divide 3 parts, constant time C is required if list contains only one or two elements, but 3T(n/3) + Dn time required dividing the list of n elements and merging the lists. The recurrence relation for time and its solution are given below. From the solution it is observed that the algorithm requires O($n \log_3 n$).

Recurrence relation for time:

$$T(n) = \begin{cases} C, \ when\ n \leq 2 \\ 3T\left(\frac{n}{3}\right) + Dn, \ when\ n \geq 3 \end{cases}$$

Solution:

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{3}\right) + Dn \\ &= 3\left[3T\left(\frac{n}{3^2}\right) + D\frac{n}{3}\right] + Dn \\ &= 3^2T\left(\frac{n}{3^2}\right) + Dn + Dn \\ &= 3^2T\left(\frac{n}{3^2}\right) + 2\,Dn \end{aligned}$$

.
.
.

$$\begin{aligned} &= 3^KT\left(\frac{n}{3^K}\right) + KDn && [Let, \frac{n}{3^K} = 2 \\ &= \frac{n}{2}T(2) + Dn(log_3n - log_32) && \Rightarrow 3^K = \frac{n}{2} \\ &= \frac{n}{2}C + Dnlog_3n - Dnlog_32 && \Rightarrow log_33^K = log_3(\frac{n}{2}) \\ &= O(nlog_3n) && \Rightarrow K \\ & && = log_3n - log_32] \end{aligned}$$

Time Complexity Analysis Penta-Merge & Hepta-Merge

For n elements input data, the complexity will be O($n \log_2 n$), O($n \log_3 n$), O($n \log_5 n$), O($n \log_7 n$), O($n \log_x n$). for binary-merge, tri-merge, penta-merge, hepta-merge and x-merge accordingly. The complexity and variable operation constraint will take part to be efficient merging algorithm.

4.4.4. Experimental Results on Merge and Ternary-Merge and Penta-Merge

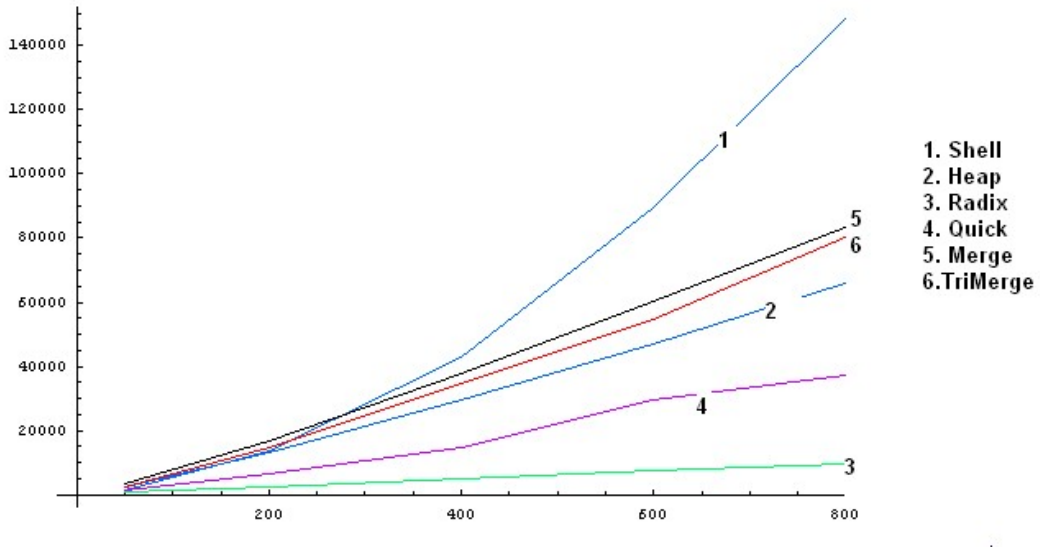
To find out comparative execution time we tried to use high memory sized computing that can sort big size data. In that case we used java programming language where memory is managed auto. In the bellow table, comparative operation counts are given (assignment and compare). It is observed that Ternary-Merge sort efficient than traditional Merge sort for larger data size. Penta-Merge Sort is more efficient than Tri-Marge. But algorithm is very difficult to understand implement. The experiment result shows that Ternary Merge take less time as well as less operation count and Penta-Merge is better for large data size..

The execution time and operation count of two sorting algorithm for various input data

Number of Data with random disorder	Merge Sort		Tri-Merge		Penta-Merge		Processing Capacity
	Number of Operation	Time in mili-second	Number of Operation	Time in mili-second	Number of Operation	Time in mili-second	
25,000,000			4,868,858,941	3,970	4,590,214,841	2,830	High CPU
2,500,000			407,055,697	380	401,730,466	580	High CPU
1,500,000	371,865,175	21,782	364,556,046	15,484			Low CPU
1,000,000	240,311,019	13,219	236,506,615	9,875			Low CPU
500,000	114,153,811	6,031	111,678,441	4,500			Low CPU
100,000	20,096,619	985	20,059,506	797			Low CPU
10,000	1,610,743	94	1,565,278	94			Low CPU
1,000	120,579	35	112,956	47			Low CPU
100	8,179	0	7,626	0			Low CPU

Table-18

Time and Operation count comparison reports



Graph-3

Comparison of number of data and operations count for 6 sorting algorithms.

From Graph-3 it is observed that Ternary-Merge sort give comparatively remarkable good result than merge sort for a large data. It has internal mechanism that causes some extra time/operation. For small data it takes an effect on the time/operation. But for large data its benefit overcomes the time/operations.

Conclusion

In this study, we modify the merge sort and try to make an improved tri-merge sort algorithm that is more efficient than marge sort. We also try with Penta-Merge but its result is good but implementation in recursive manner is difficult. Hepta-Merge is more complex to implement. The result is practically true based on investigation data set, in which merging technique is done among three parts. The paper concludes the following:

i) For n elements list, the traditional method (Merge) and our proposed Tri-Merge and Penta-Merge sort algorithm require O($n \log_2 n$), O($n \log_3 n$) and O($n \log_5 n$) time, respectively.

ii) For the larger number of input elements, the real time for our proposed method is less in comparison with the traditional method.

iii) At lowest stage when data remain 1 or 2 there need additional compare for manage ternary structure that is why it cost little bit higher than binary marge sort. But when data size is large, it gains lot of operation count which gives significant result compare than merge sort

iv) Number of operations for our proposed method is fewer at higher dimensions of input elements that declare tri-merge is more efficient than merge sort algorithm

Limitation

At the time of experiment, we unable to use high computing power and high memory size. For better result and comparison, we shall have to use high capacity computing power.

Future Work

In future we have plan to work with high performance CPU with high memory that work with huge data size to experiment the result. There is lot of opportunity to improve the algorithm that can be remarkable contribution in the sorting algorithms.

References

[1] Knuth D.E., The Art of Computer Programming, vol 3: Sorting and Searching Algorithm, Addison-Wesley; Reading MA; 1973.

[2] Weiss M.A., Data Structures and Algorithm Analysis in C; Addison-Wesley; Reading MA; 1993.

[3] C.A.R. Hoare C.A.R., Algorithm 63 (partition) and algorithm 65 (find), Comm. ACM 4(7) (1961) 321-322.

[4] Sedgewick R.., Quick sort, Garland Publishing, New York, 1980.

[5] Sedgewick R.., Implementing Quick sort programs, Comm. ACM 21 (10) (1978) 847-857.

[6] Williams J.W., Heap sort (alg. 232); Comm. ACM 7 (1964) 347-348.

[7] Rosen, Kenneth H, Discrete Mathematics and its applications, 5th Edition, pp. 120 – 135 and 274 – 283, McGraw-Hill, New York, 2003.

[8] Grimaldi R.P., Discrete and combinatorial Mathematics, 3rd Edition, pp. 634 – 638, Addison-Wesley; Reading MA; 1994.

[9] Heap sort. http://info.mcip.ro/?t=ts&p=7, March 2014.

[10] Arne Andersson and Stefan Nilsson. A new efficient radix sort. In FOCS, pages 714–721. IEEE Computer Society, 1994.

[11] Kenneth E. Batcher. Sorting networks and their applications. Spring Joint Computer Conference, AFIPS Proc, 32:307–314, 1968.

[12] Jon Louis Bentley and M. Douglas McIlroy. Engineering a sort function. Softw., Pract. Exper., 23(11):1249–1265, 1993.

[13] Coenraad Bron. Merge sort algorithm [m1] (algorithm 426). Commun. ACM, 15(5):357–358, 1972.

[14] M.S. Garai Canaan.C and M. Daya. Popular sorting algorithms. World Applied Programming, 1:62–71, April 2011.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 2009.

[16] Ian J. Davis. A fast radix sort. Comput. J., 35(6):636–642, 1992.

[17] E.H.Friend. Sorting on electronic computers. JACM 3(2), pages 34–168, 1956.

[18] Donald E.Knuth. The Art of Computer Programming Second Edition, volume 3. ADDISON-WESLEY, 1998.

[19] C.A.R. Hoare. Quicksort. Commun.ACM, 4:321, 1961.

[20] Omar khan Durrani, Shreelakshmi V, Sushma Shetty, and Vinutha D C. Analysis and determination of asymptotic behavior range for popular sorting algorithms. Special Issue of International Journal of Computer Science Informatics(IJCSI), ISSN(PRINT):2231-5292, Vol-2, Issue-1,2

[21] N. Santoro M.D. Atkinson, J.-R. Sack and T.Strothotte. Min-max heaps and generalized priority queues. Communications of the ACM, October 1986.

[22] A.D. Mishra and D. Garg. Selection of Best Sorting Algorithm. International Journal of Intelligent Processing, 2(2):363–368, July-December 2008.

[23] Samanta Debasis Samanta. Classic Data Structure, Second Edition. PHI, 2009.

[24] Pankaj Sareen. Comparison of sorting algorithms(on the basis of average case). IJARCSSE, 3:522–532, March 2013.

[25] Harold H Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, M.I.T, 1954.

[26] Clifford A. Shaffer. A Practical Introduction to Data Structures and Algorithm Analysis Third Edition. Prentice Hall, April 2009.

[27] D. L. Shell. A high-speed sorting procedure. Commun. ACM, 2(7):30–32, July 1959.

[28] Ingo Wegener. Bottom-up-heap sort, a new variant of heap sort beating on average quick sort (if n is not very small). In Branislav Rovan, editor, MFCS, volume 452 of Lecture Notes in Computer Science, pages 516–522. Springer, 1990.

[29] Mark A. Weiss. Data Structures Algorithms Analysis in C++ (Third Edition). Prentice Hall, March 2006.

[30] J. W. J. Williams. Algorithm 232: Heapsort. Communications of the ACM, 7(6):347–348, 1964.

[31] Niklaus Wirth. Algorithms+DataStructures=Programs. Prentice Hall.

[32] Ray Wisman. The fundamentals: Algorithms, the integers, and matrices, June 1998.

[33] M. KAYKOBAD, MD. M. ISLAM AND M. E. AMYEEN. 3 is a More Promising Algorithmic Parameter than 2, 1998

Appendix A

```
#include<stdio.h>
#include<stdlib.h>
//#include <sys\timeb.h>
#include <math.h>

//http://rextester.com/l/c_online_compiler_gcc

#define N 250000
#define DIGNO 5 //for Radix Sort
#define D 100  //for Radix Sort

void TriMergeSort( int A[],  int L,  int R, long int asscount[], long int comcount[]);
void TriMerge( int a[], int L, int m1,int m2, int R,long int asscount[], long int comcount[]);

void PentaMergeSort( int A[],  int L,  int R, long int asscount[], long int comcount[]);
void PentaMerge( int a[], int L, int m1, int m2,int m3,int m4, int m5, int R, long int asscount[], long
int comcount[]);

void Swap(int k1, int k2);



long int comcount[1], asscount[1];
        int i, a[N], b[N],temp[N], k;

int  main (void)
{
        for (i=0; i < N; i++)
        {
                a[i] = rand() % 100;
                temp[i] = -1;
        }
```

```
            printf ("\n Sort Name\t   Assignment Count\tCompare Count\tTotal Count\n");
            printf (" =========\t   ================\t=============\t===========");

/*          comcount[0] = 0;
            asscount[0] = 0;

            PentaMergeSort(a, 0, N-1, asscount, comcount);
//          printf("\nPenta-Merge");
            printf("\n\t\t%15ld%15ld%15ld", asscount[0], comcount[0], asscount[0] + comcount[0]);
*/
            comcount[0] = 0;
            asscount[0] = 0;
            TriMergeSort(a, 0, N-1, asscount, comcount);
            printf("\nTriMerge");
            printf("%12ld%15ld%15ld", asscount[0], comcount[0], asscount[0] + comcount[0]);


    printf ("\nAfter Merge\n");
      for(i = 0; i < N; i++)
           {
          printf("(%d - %d), ", i, a[i]);
           }
}

void Swap(int k1, int k2)
{
   int t = k1;
   k1 = k2;
   k2 = t;

}

void TriMergeSort( int a[], int L,  int R, long int asscount[],long int comcount[])
{
            int noOfEle=R-L+1;
            int m1,m2,part,t;
```

```
        comcount[0]++;
        if(noOfEle > 2)
        {
                part=(R-L+1)/3;
                m1 = L+part-1;
                m2 = L+2*part-1;
                asscount[0]+=3;

                TriMergeSort(a, L, m1, asscount, comcount);
                TriMergeSort(a, m1+1, m2, asscount, comcount);
                TriMergeSort(a, m2+1, R, asscount, comcount);
                TriMerge(a, L, m1+1, m2+1, R, asscount, comcount);
        }
        else if(noOfEle == 2)
        {
                comcount[0]++;
                if(a[L] > a[R])
                {
                        asscount[0] += 3;
                        t = a[L];
                        a[L] = a[R];
                        a[R] = t;
                }
   }
}


void PentaMergeSort( int a[], int L,  int R, long int asscount[],long int comcount[])
{
        int noOfEle = R-L+1;
        int m1,m2,m3,m4, m5, part;
        comcount[0]++;


        if(noOfEle>=5)
        {
```

```
            part = (R-L+1)/5;

            m1 = L + 0*part;
            m2 = L+1*part;
            m3 = L+2*part;
            m4 = L+3*part;
            m5 = L+4*part;
                asscount[0]+=5;


                PentaMergeSort(a, m1, m2-1, asscount, comcount);
                PentaMergeSort(a, m2, m3-1, asscount, comcount);
                PentaMergeSort(a, m3, m4-1, asscount, comcount);
                PentaMergeSort(a, m4, m5-1, asscount, comcount);
                PentaMergeSort(a, m5, R, asscount, comcount);

                PentaMerge(a, L,m1, m2, m3, m4, m5, R, asscount, comcount);

    }

    else if(noOfEle > 2 &&  noOfEle <5  )
        {
                part = (R-L+1)/3;
                m1 = L+part-1;
                m2 = L+2*part-1;
                asscount[0] += 3;

                TriMergeSort(a,L,m1,asscount,comcount);
                TriMergeSort(a, m1+1,m2,asscount,comcount);
                TriMergeSort(a, m2+1,R,asscount,comcount);

                TriMerge(a,L,m1+1,m2+1,R,asscount,comcount);
        }

}
```

```
void PentaMerge( int a[], int L, int m1, int m2,int m3,int m4, int m5, int R, long int asscount[], long int comcount[])
{
//          int temp[N],t; // Temporary array
            int i, part;
           int LL = L, RR = R;
           int t;
            int end1 = m2;
            int end2 = m3;
            int end3 = m4;
            int end4 = m5;
            int end5 = R+1;
            int no_ele = R-L+1;
            int index = L;
            int minval = a[L];
           int min = L,pos = 1;

           asscount[0]+=4;
           comcount[0]++;

//          if (no_ele>=5)
//    {

// start 5 part

                    while((m1 < end1) && (m2 < end2) && (m3 < end3) && (m4 < end4) && (m5 < end5))
                    {
                    comcount[0]+=5;
                    min = m1;
                    pos = 1;
                    asscount[0]+=2;

                            if(a[m1] < a[index] )
                            {
```

```
            min = m1;
            pos = 1;
            asscount[0]+=2;
}

 if(a[m2] < a[index] )
{
            min = m2;
            pos = 2;
            asscount[0]+=2;
}
 if(a[m3] < a[index] )
{
            min=m3;
            pos=3;
            asscount[0]+=2;
}
 if(a[m4] < a[index] )
{
            min=m4;
            pos=4;
            asscount[0]+=2;
}
 if(a[m5] < a[index] )
{
            min=m5;
            pos=5;
            asscount[0]+=2;
}

comcount[0] += 5;

temp[index] = a[min];
index++;
```

```
                    if(pos==1)
                            m1++;
                     if(pos==2)
                            m2++;
                     if(pos==3)
                            m3++;
                     if(pos==4)
                            m4++;
                     if(pos==5)
                            m5++;


                    comcount[0] += 5;
                    asscount[0] += 7;



            }

// start 4-5 part

   while((m1 < end1) && (m2 < end2) && (m3 < end3) && (m4 < end4) )
            {
                    comcount[0]+=4;
                   min = m1;
                    pos = 1;
                    asscount[0]+=2;

                    if(a[m1] < a[index] )
                    {
                            min = m1;
                            pos = 1;
                            asscount[0]+=2;
                    }

                     if(a[m2] < a[index] )
```

```
{
        min=m2;
        pos=2;
        asscount[0]+=2;
}
 if(a[m3] < a[index] )
{
        min=m3;
        pos=3;
        asscount[0]+=2;
}
 if(a[m4] < a[index] )
{
        min=m4;
        pos=4;
        asscount[0]+=2;
}

comcount[0] += 4;

temp[index]=a[min];
//printf ("\n index: %d,a: %d) ",index, temp[index]);
index++;

if(pos==1)
        m1++;
 if(pos==2)
        m2++;
 if(pos==3)
        m3++;
 if(pos==4)
        m4++;


comcount[0] += 4;
asscount[0] += 6;
```

```
                }
// end 4-5 part



// start 4-4 part
                while((m1 < end1) && (m2 < end2) && (m3 < end3) && (m5 < end5) )
                {
                        comcount[0]+=4;

                       min = m1;
                        pos = 1;

                        asscount[0]+=2;

                        if(a[m1] < a[index] )
                        {
                                min=m1;
                                pos=1;
                                asscount[0]+=2;
                        }
                         if(a[m2] < a[index] )
                        {
                                min=m2;
                                pos=2;
                                asscount[0]+=2;
                        }
                         if(a[m3] < a[index] )
                        {
                                min=m2;
                                pos=3;
                                asscount[0]+=2;
                        }
                         if(a[m5] < a[index] )
                        {
                                min=m5;
```

```
                pos=5;
                asscount[0]+=2;
            }

            comcount[0] += 4;

            temp[index]=a[min];
            ////printf ("\n index: %d,a: %d) ",index, temp[index]);
            index++;

           if(pos==1)
                m1++;
            if(pos==2)
                m2++;
            if(pos==3)
                m3++;
            if(pos==5)
                m5++;

            comcount[0] += 4;
            asscount[0] += 6;
        }
// end 4-4 part



// start 4-3 part
        while((m1 < end1) && (m2 < end2) && (m4 < end4) && (m5 < end5) )
        {
            comcount[0]+=4;
            min = m1;
            pos = 1;
            asscount[0]+=2;

            if(a[m1] < a[index] )
            {
```

```
                min=m1;
                pos=1;
                asscount[0]+=3;
        }
        if(a[m2] < a[index] )
        {
                min=m2;
                pos=2;
                asscount[0]+=3;
        }
         if(a[m4] < a[index] )
        {
                min=m4;
                pos=4;
                asscount[0]+=3;
        }
         if(a[m5] < a[index] )
        {
                min=m5;
                pos=5;
                asscount[0]+=3;
        }

        comcount[0] += 4;

        temp[index]=a[min];
////printf ("\n index: %d,a: %d) ",index, temp[index]);
        index++;

        if(pos==1)
                m1++;
         if(pos==2)
                m2++;
         if(pos==4)
                m4++;
         if(pos==5)
```

```
                    m5++;


                comcount[0] += 4;
                asscount[0] += 6;
            }
// end 4-3 part


// start 4-2 part
            while((m1 < end1) && (m3 < end3) && (m4< end4) && (m5< end5) )
            {
                comcount[0]+=4;
                min = m1;
                pos = 1;

                if(a[m1] < a[index] )
                {
                    min=m1;
                    pos=1;
                    asscount[0]+=3;
                }


                 if(a[m3] < a[index] )
                {
                    min=m3;
                    pos=3;
                    asscount[0]+=3;
                }
                 if(a[m4] < a[index] )
                {
                    min=m4;
                    pos=4;
                    asscount[0]+=3;
                }
```

```
                        if(a[m5] < a[index] )
                        {
                                min=m5;
                                pos=5;
                                asscount[0]+=3;
                        }

                        comcount[0] += 3;

                        temp[index]=a[min];
        ////printf ("\n index: %d,a: %d) ",index, temp[index]);
                        index++;

                        if(pos==1)
                                m1++;
                        if(pos==3)
                                m3++;
                        if(pos==4)
                                m4++;
                        if(pos==5)
                                m5++;


                        comcount[0] += 4;
                        asscount[0] += 6;
                }
// end 4-2 part

// start 4-1 part
                while((m2 < end2) && (m3 < end3) && (m4 < end4) && (m5 < end5) )
                {
                        comcount[0]+=4;
                        min = m2;
                        pos = 2;

                        if(a[m2] <= a[index] )
```

```
{
        min=m2;
        pos=2;
        asscount[0]+=3;
}

 if(a[m3] < a[index] )
{
        min=m3;
        pos=3;
        asscount[0]+=3;
}
 if(a[m4] < a[index] )
{
        min=m4;
        pos=4;
        asscount[0]+=3;

}
 if(a[m5] < a[index] )
{
        min=m5;
        pos=5;
        asscount[0]+=3;
}

comcount[0] += 3;

temp[index]=a[min];
////printf ("\n index: %d,a: %d) ",index, temp[index]);
index++;

if(pos==2)
        m2++;
 if(pos==3)
        m3++;
```

```
                    if(pos==4)
                            m4++;
                    if(pos==5)
                            m5++;

                    comcount[0] += 4;
                    asscount[0] += 6;
            }
// end 4-1 part



// start 3-4,5 part
            while((m1 < end1) && (m2 < end2) && (m3 < end3)  )
            {
                    comcount[0]+=3;
                    min = m1;
                    pos = 1;

                    if(a[m1] < a[index] )
                    {
                            min=m1;
                            pos=1;
                            asscount[0]+=3;
                    }
                    if(a[m2] < a[index] )
                    {
                            min = m2;
                            pos = 2;
                            asscount[0]+=3;
                    }
                    if(a[m3] < a[index] )
                    {
                            min = m3;
                            pos = 3;
                            asscount[0]+=3;
```

```
                    }

                    comcount[0] += 2;

                    temp[index]=a[min];
        ////printf ("\n index: %d,a: %d) ",index, temp[index]);
                    index++;

                    if(pos==1)
                            m1++;
                     if(pos==2)
                            m2++;
                     if(pos==3)
                            m3++;

                    comcount[0] += 3;
                    asscount[0] += 5;
                }
// end 3-4,5 part


// start 3-3,5 part
                while((m1 < end1) && (m2 < end2) && (m4 < end4)  )
                {
                    comcount[0]+=3;
                    min = m1;
                    pos = 1;

                    if(a[m1] < a[index] )
                    {
                            min=m1;
                            pos=1;
                            asscount[0]+=3;
                    }
                     if(a[m2] < a[index] )
                    {
```

```
                    min = m2;
                    pos = 2;
                    asscount[0]+=3;
                }
                 if(a[m4] < a[index] )
                {
                    min = m4;
                    pos = 4;
                    asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

                if(pos==1)
                    m1++;
                 if(pos==2)
                    m2++;
                 if(pos==4)
                    m4++;

                comcount[0] += 3;
                asscount[0] += 5;
            }
// end 3-3,5 part


// start 3-3,4 part
            while((m1 < end1) && (m2 < end2) && (m5 < end5)  )
            {
                comcount[0]+=3;
                min = m1;
                pos = 1;
```

```
                    if(a[m1] < a[index] )
                    {
                            min=m1;
                            pos=1;
                            asscount[0]+=3;
                    }
                     if(a[m2] < a[index] )
                    {
                            min = m2;
                            pos = 2;
                            asscount[0]+=3;
                    }
                     if(a[m5] < a[index] )
                    {
                            min = m5;
                            pos = 5;
                            asscount[0]+=3;
                    }

                    comcount[0] += 2;

                    temp[index]=a[min];
//printf ("\n index: %d,a: %d) ",index, temp[index]);
                    index++;

                    if(pos==1)
                            m1++;
                     if(pos==2)
                            m2++;
                     if(pos==5)
                            m5++;

                    comcount[0] += 3;
                    asscount[0] += 5;
            }
```

```
// end 3-3,4 part


// start 3-2,5 part
                while((m1 < end1) && (m3 < end3) && (m4 < end4)  )
                {
                        comcount[0]+=3;
                        min = m1;
                        pos = 1;

                        if(a[m1] < a[index] )
                        {
                                min=m1;
                                pos=1;
                                asscount[0]+=3;
                        }
                         if(a[m3] < a[index] )
                        {
                                min = m3;
                                pos = 3;
                                asscount[0]+=3;
                        }
                         if(a[m4] < a[index] )
                        {
                                min = m4;
                                pos = 4;
                                asscount[0]+=3;
                        }

                        comcount[0] += 2;

                        temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                        index++;
```

```
                    if(pos==1)
                            m1++;
                     if(pos==3)
                            m3++;
                     if(pos==4)
                            m4++;

                    comcount[0] += 3;
                    asscount[0] += 5;
            }

// end 3-2,5 part


// start 3-2,4 part
            while((m1 < end1) && (m3 < end3) && (m5 < end5)  )
            {
                    comcount[0]+=3;
                    min = m1;
                    pos = 1;

                    if(a[m1] < a[index] )
                    {
                            min=m1;
                            pos=1;
                            asscount[0]+=3;
                    }
                     if(a[m3] < a[index] )
                    {
                            min = m3;
                            pos = 3;
                            asscount[0]+=3;
                    }
                     if(a[m5] < a[index] )
                    {
```

```
                        min = m5;
                        pos = 5;
                        asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

                if(pos==1)
                        m1++;
                 if(pos==3)
                        m3++;
                 if(pos==5)
                        m5++;

                comcount[0] += 3;
                asscount[0] += 5;
        }


// end 3-2,4 part


// start 3-2,3 part
                while((m1 < end1) && (m4 < end4) && (m5 < end5)  )
                {
                        comcount[0]+=3;
        min = m1;
        pos = 1;

                        if(a[m1] < a[index] )
                        {
                                min=m1;
```

```
                    pos=1;
                    asscount[0]+=3;
                }
                 if(a[m4] < a[index] )
                {
                    min = m4;
                    pos = 4;
                    asscount[0]+=3;
                }
                 if(a[m5] < a[index] )
                {
                    min = m5;
                    pos = 5;
                    asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

        if(pos==1)
                    m1++;
                 if(pos==4)
                    m4++;
                 if(pos==5)
                    m5++;

                comcount[0] += 3;
                asscount[0] += 5;
            }

// end 3-2,3 part
```

```
// start 3-2,1 part
                while((m3 < end3) && (m4 < end4) && (m5 < end5)  )
                {
                        comcount[0]+=3;
                         min = m3;
                        pos = 3;

                        if(a[m3] < a[index] )
                        {
                                min=m3;
                                pos=3;
                                asscount[0]+=3;
                        }
                         if(a[m4] < a[index] )
                        {
                                min = m4;
                                pos = 4;
                                asscount[0]+=3;
                        }
                         if(a[m5] < a[index] )
                        {
                                min = m5;
                                pos = 5;
                                asscount[0]+=3;
                        }

                        comcount[0] += 2;

                        temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                        index++;

                        if(pos==3)
                                m3++;
                         if(pos==4)
```

```
                    m4++;
                if(pos==5)
                    m5++;

                comcount[0] += 3;
                asscount[0] += 5;
            }

// end 3-2,1 part

// start 3-3,1 part
            while((m2 < end2) && (m4 < end4) && (m5 < end5)  )
            {
                comcount[0]+=3;
                min = m2;
                pos = 2;

                if(a[m2] < a[index] )
                {
                    min=m2;
                    pos=2;
                    asscount[0]+=3;
                }
                if(a[m4] < a[index] )
                {
                    min = m4;
                    pos = 4;
                    asscount[0]+=3;
                }
                if(a[m5] < a[index] )
                {
                    min = m5;
                    pos = 5;
                    asscount[0]+=3;
                }
```

```
                        comcount[0] += 2;

                        temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                        index++;

                        if(pos==2)
                                m2++;
                         if(pos==4)
                                m4++;
                         if(pos==5)
                                m5++;

                        comcount[0] += 3;
                        asscount[0] += 5;
                }

// end 3-3,1 part


// start 3-5,1 part
                while((m2 < end2) && (m3 < end3) && (m4 < end4)  )
                {
                        comcount[0]+=3;
                        min = m2;
                        pos = 2;

                        if(a[m2] < a[index] )
                        {
                                min=m2;
                                pos=2;
                                asscount[0]+=3;
                        }
                         if(a[m3] < a[index] )
                        {
                                min = m3;
```

```
                    pos = 3;
                    asscount[0]+=3;
                }
                 if(a[m4] < a[index] )
                {
                    min = m4;
                    pos = 4;
                    asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

                if(pos==2)
                    m2++;
                 if(pos==3)
                    m3++;
                 if(pos==4)
                    m4++;

                comcount[0] += 3;
                asscount[0] += 5;
            }


// end 3-5,1 part


// start 3-4,1 part
            while((m2 < end2) && (m3 < end3) && (m5 < end5)  )
            {
                comcount[0]+=3;
                min = m2;
```

```
                pos = 2;

                if(a[m2] < a[index] )
                {
                        min=m2;
                        pos=2;
                        asscount[0]+=3;
                }
                 if(a[m3] < a[index] )
                {
                        min = m3;
                        pos = 3;
                        asscount[0]+=3;
                }
                 if(a[m5] < a[index] )
                {
                        min = m5;
                        pos = 5;
                        asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
//printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

                if(pos==2)
                        m2++;
                 if(pos==3)
                        m3++;
                 if(pos==5)
                        m5++;

                comcount[0] += 3;
                asscount[0] += 5;
```

```
                }

    // end 3-4,1 part


// start 2/1,2 part
                while((m1 < end1) && (m2 < end2)  )
                {
                        comcount[0]+=3;
                        min = m1;
                        pos = 1;

                        if(a[m1] < a[index] )
                        {
                                min=m1;
                                pos=1;
                                asscount[0]+=3;
                        }
                         if(a[m2] < a[index] )
                        {
                                min = m2;
                                pos = 2;
                                asscount[0]+=3;
                        }

                        comcount[0] += 2;

                        temp[index]=a[min];
          //printf ("\n index: %d,a: %d) ",index, temp[index]);
                        index++;

                        if(pos==1)
                                m1++;
                         if(pos==2)
                                m2++;
```

```
                    comcount[0] += 3;
                    asscount[0] += 5;
                }

// end 2/1,2 part


// start 2/1,3 part
                while((m1 < end1) && (m3 < end3)  )
                {
                    comcount[0]+=3;
                    min = m1;
                    pos = 1;

                    if(a[m1] < a[index] )
                    {
                        min=m1;
                        pos=1;
                        asscount[0]+=3;
                    }
                     if(a[m3] < a[index] )
                    {
                        min = m3;
                        pos = 3;
                        asscount[0]+=3;
                    }

                    comcount[0] += 2;

                    temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                    index++;

                    if(pos==1)
                        m1++;
                     if(pos==3)
```

```
                    m3++;

                comcount[0] += 3;
                asscount[0] += 5;
            }

// end 2/1,3 part


// start 2/1,4 part
            while((m1 < end1) && (m4 < end4)  )
            {
                comcount[0]+=3;
                min = m1;
                pos = 1;

                if(a[m1] < a[index] )
                {
                    min=m1;
                    pos=1;
                    asscount[0]+=3;
                }
                 if(a[m4] < a[index] )
                {
                    min = m4;
                    pos = 4;
                    asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

                if(pos==1)
```

```
                    m1++;
                if(pos==4)
                    m4++;

                comcount[0] += 3;
                asscount[0] += 5;
            }
// end 2/1,4 part


// start 2/1,5 part
            while((m1 < end1) && (m5 < end5)  )
            {
                comcount[0]+=3;
                min = m1;
                pos = 1;

                if(a[m1] < a[index] )
                {
            min=m1;
                    pos=1;
                    asscount[0]+=3;
                }
                 if(a[m5] < a[index] )
                {
                    min = m5;
                    pos = 5;
                    asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;
```

```
                    if(pos==1)
                            m1++;
                     if(pos==5)
                            m5++;

                    comcount[0] += 3;
                    asscount[0] += 5;
                }

// end 2/1,5 part


// start 2/2,3 part
                while((m2 < end2) && (m3 < end3)  )
                {
                    comcount[0]+=3;
                    min = m2;
                    pos = 2;

                    if(a[m2] < a[index] )
                    {
                            min=m2;
                            pos=2;
                            asscount[0]+=3;
                    }
                     if(a[m3] < a[index] )
                    {
                            min = m3;
                            pos = 3;
                            asscount[0]+=3;
                    }

                    comcount[0] += 2;

                    temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
```

```
                index++;

                if(pos==2)
                        m2++;
                 if(pos==3)
                        m3++;

                comcount[0] += 3;
                asscount[0] += 5;
        }

// end 2/2,3 part


// start 2/2,4 part
        while((m2 < end2) && (m4 < end4)  )
        {
                comcount[0]+=3;
                min = m2;
                pos = 2;

                if(a[m2] < a[index] )
                {
                        min=m2;
                        pos=2;
                        asscount[0]+=3;
                }
                 if(a[m4] < a[index] )
                {
                        min = m4;
                        pos = 4;
                        asscount[0]+=3;
                }

                comcount[0] += 2;
```

```
                        temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                        index++;

                        if(pos==2)
                                m2++;
                         if(pos==4)
                                m4++;

                        comcount[0] += 3;
                        asscount[0] += 5;
                }



// end 2/2,4 part


// start 2/2,5 part
                while((m2 < end2) && (m5 < end5)  )
                {
                        comcount[0]+=3;
                        min = m2;
                        pos = 2;

                        if(a[m2] < a[index] )
                        {
                                min=m2;
                                pos=2;
                                asscount[0]+=3;
                        }
                        if(a[m5] < a[index] )
                        {
                                min = m5;
                                pos = 5;
                                asscount[0]+=3;
```

```
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

                if(pos==2)
                        m2++;
                 if(pos==5)
                        m5++;

                comcount[0] += 3;
                asscount[0] += 5;
        }

// end 2/2,5 part


// start 2/3,4 part
        while((m3 < end3) && (m4 < end4)  )
        {
                comcount[0]+=3;
                min = m3;
                pos = 3;

                if(a[m3] < a[index] )
                {
                        min=m3;
                        pos=3;
                        asscount[0]+=3;
                }
                 if(a[m4] < a[index] )
                {
                        min = m4;
```

```
                    pos = 4;
                    asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

                if(pos==3)
                    m3++;
                 if(pos==4)
                    m4++;

                comcount[0] += 3;
                asscount[0] += 5;
            }
// end 2/3,4 part


// start 2/3,5 part
            while((m3 < end3) && (m5 < end5)  )
            {
                comcount[0]+=3;
                min = m3;
                pos = 3;

                if(a[m3] < a[index] )
                {
            min=m3;
                    pos=3;
                    asscount[0]+=3;
                }
                 if(a[m5] < a[index] )
                {
```

```
                    min = m5;
                    pos = 5;
                    asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

        if(pos==3)
                    m3++;
                 if(pos==5)
                    m5++;

                comcount[0] += 3;
                asscount[0] += 5;
            }
// end 2/3,5 part


// start 2/4,5 part
            while((m4 < end4) && (m5 < end5)  )
            {
                comcount[0]+=3;
                min = m4;
                pos = 4;

                if(a[m4] < a[index] )
                {
                    min=m4;
                    pos=4;
                    asscount[0]+=3;
                }
                 if(a[m5] < a[index] )
```

```
                {
                        min = m5;
                        pos = 5;
                        asscount[0]+=3;
                }

                comcount[0] += 2;

                temp[index]=a[min];
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;

                if(pos==4)
                        m4++;
                 if(pos==5)
                        m5++;

                comcount[0] += 3;
                asscount[0] += 5;
        }
// end 2/4,5 part

// start 1 part
        while(m1 < end1)
        {
                comcount[0]++;
                asscount[0]+=3;
                temp[index]=a[m1];
                m1++;
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;
        }



// end 1 part
```

```
// start 2 part
                while(m2 < end2)
                {
                        comcount[0]++;
                        asscount[0]+=3;
                        temp[index] = a[m2];
                        m2++;
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                        index++;
                }

// end 2 part

// start 3 part
                while(m3 < end3)
                {
                        comcount[0]++;
                        asscount[0]+=3;
                        temp[index] = a[m3];
                        m3++;
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
                        index++;
                }

// end 3 part

// start 4 part
                while(m4 < end4)
                {
                        comcount[0]++;
                        asscount[0]+=3;
                        temp[index] = a[m4];
                        m4++;
        //printf ("\n index: %d,a: %d) ",index, temp[index]);
```

```
                index++;
        }

// end 4 part

// start 5 part
        while(m5 < end5)
        {
                comcount[0]++;
                asscount[0]+=3;
                temp[index] = a[m5];
                m5++;
    //printf ("\n index: %d,a: %d) ",index, temp[index]);
                index++;
        }

// end 5 part

 //   }

        for(i=0;i<no_ele;i++)
        {
                comcount[0]++;
                asscount[0]+=3;
                a[R]=temp[R];
              R--;
        }

}


void TriMerge( int a[], int L, int m1, int m2, int R, long int asscount[], long int comcount[])
{
    int temp[N]; // Temporary array
    int i;
    int end1 = m1-1;
```

```
 int end2 = m2-1;
 int no_ele = R-L+1;
 int index = L;
asscount[0]+=4;
comcount[0]++;
 if(no_ele>2)
 {
        while((L<=end1)&&(m1<=end2)&&(m2<=R))
        {
                comcount[0]+=4;
                int min,pos;
                if(a[L]<a[m1])
                {
                        min=L;
                        pos=1;
                        asscount[0]+=2;
                }
                else
                {
                        min=m1;
                        pos=2;
                        asscount[0]+=2;
                }
                comcount[0]++;

if(a[min]<a[m2])
                {
                        min=min;
                        asscount[0]++;
}
                else
                {
                        min=m2;
                        pos=3;
                        asscount[0]+=2;
 }
```

```
		asscount[0]+=3;
		comcount[0]+=2;
		temp[index]=a[min];
		index++;

		if(pos==1)
			L++;
		else if(pos==2)
			m1++;
		else if(pos==3)
			m2++;
	}
	while((L<=end1)&&(m1<=end2))
	{
		comcount[0]+=3;
		if (a[L]<=a[m1])
		{
			temp[index]=a[L];
			index++;
			L++;
			asscount[0]+=3;
		}
		else
		{
			temp[index]=a[m1];
			index++;
			m1++;
			asscount[0]+=3;

		}
	}
	while((m1<=end2)&&(m2<=R))
	{
		comcount[0]+=3;
		if (a[m1]<=a[m2])
		{
```

```
                temp[index]=a[m1];
                index++;
                m1++;
                asscount[0]+=3;
        }
        else
        {
                temp[index]=a[m2];
                index++;
                m2++;
                asscount[0]+=3;
        }
}
while((L<=end1)&&(m2<=R))
{
        comcount[0]+=3;
        if (a[L]<=a[m2])
        {
                temp[index]=a[L];
                index++;
                L++;
        asscount[0]+=3;
        }
        else
        {
                temp[index]=a[m2];
                index++;
                m2++;
                asscount[0]+=3;
        }
}
while(L<=end1)
{
        comcount[0]++;
        asscount[0]+=3;
        temp[index]=a[L];
```

```
                index++;
                L++;
        }
        while(m1<=end2)
        {
                comcount[0]++;
                asscount[0]+=3;
                temp[index]=a[m1];
                index++;
                m1++;
        }
        while(m2<=R)
        {
                comcount[0]++;
                asscount[0]+=3;
                temp[index]=a[m2];
                index++;
                m2++;
        }

    }


    for(i=0;i<no_ele;i++)
        {
                comcount[0]++;
                asscount[0]+=3;
                a[R]=temp[R];
                R--;
        }
}
```

-End-