I B.Tech II Semester CSE (R-20)

DATA STRUCTURES

SORTING TECHNIQUES

****

Basic concepts, Sorting by: insertion (insertion sort), Selection sort(heap sort), exchange sort(bubble sort, quick sort), distribution sort(radix sort) and merging (merge sort) algorithms

****

SORTING: Arranging the elements either in ascending order or in descending order is called Sorting. Sorting techniques are broadly categorized into two.

1. **Internal Sorting**: All the records that are to be sorted are in main memory.

2. **External Sorting**: Some sorts that cannot be performed in main memory and must be done on disk or tape. This type of sorting is known as External Sorting.

Efficiency: One of the major issues in the sorting algorithms is its efficiency. If we can efficiently sort the records then, that adds value to the sorting algorithm. We usually denote the efficiency of sorting algorithm in terms of time complexity. The time complexities are given in terms of big-oh notation.

Commonly there are $O(n^2)$ and O(n log n) time complexities for various algorithms. Quick sort is the fastest algorithm and bubble sort is the slowest one.

Sorting Method	Efficiency	
Bubble sort/Selection sort/Insertion sort	$\mathbf{0(n^2)}$	
Shell sort	$\mathbf{0(n^2)}$	$\mathbf{0(n^{7/6})}$
Heap / Quick / Merge/ Radix	**0(n log n)**	

Various Sorting Techniques are

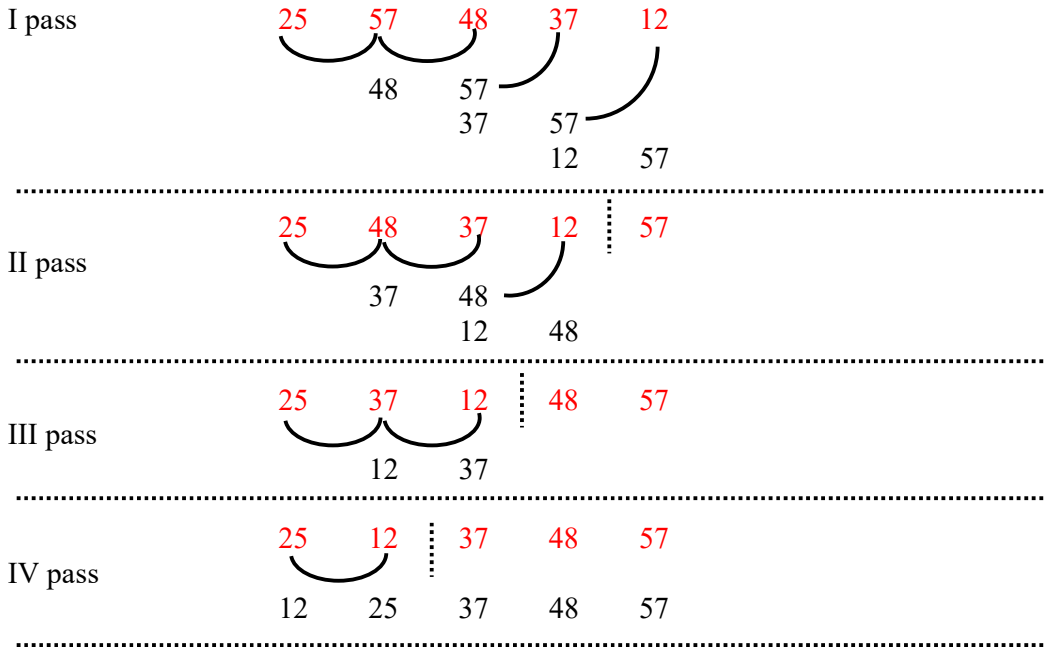
1. **Bubble sort**
2. **Insertion sort**
3. **Selection sort**
4. **Quick sort**
5. **Merge sort**
6. **Heap sort**
7. **Radix sort**

Bubble Sort:

A well known algorithm called Bubble Sort, is easy to understand. Probably this is the least efficient.

The basic idea underlying the bubble sort is to pass through the array left to right several times. Each pass consists of comparing each element in the array with its successor and interchanging the two elements if they are not in proper order. At the end of each pass we can observe that the largest element of the list is moved to its final position.

Consider the following array.



After 4th pass the array contains

12 25 37 48 57

The array contains 5 elements and after 4 passes the array is sorted. An array of n elements requires not more than n-1 iterations / passes.

Efficiency of Bubble Sort:

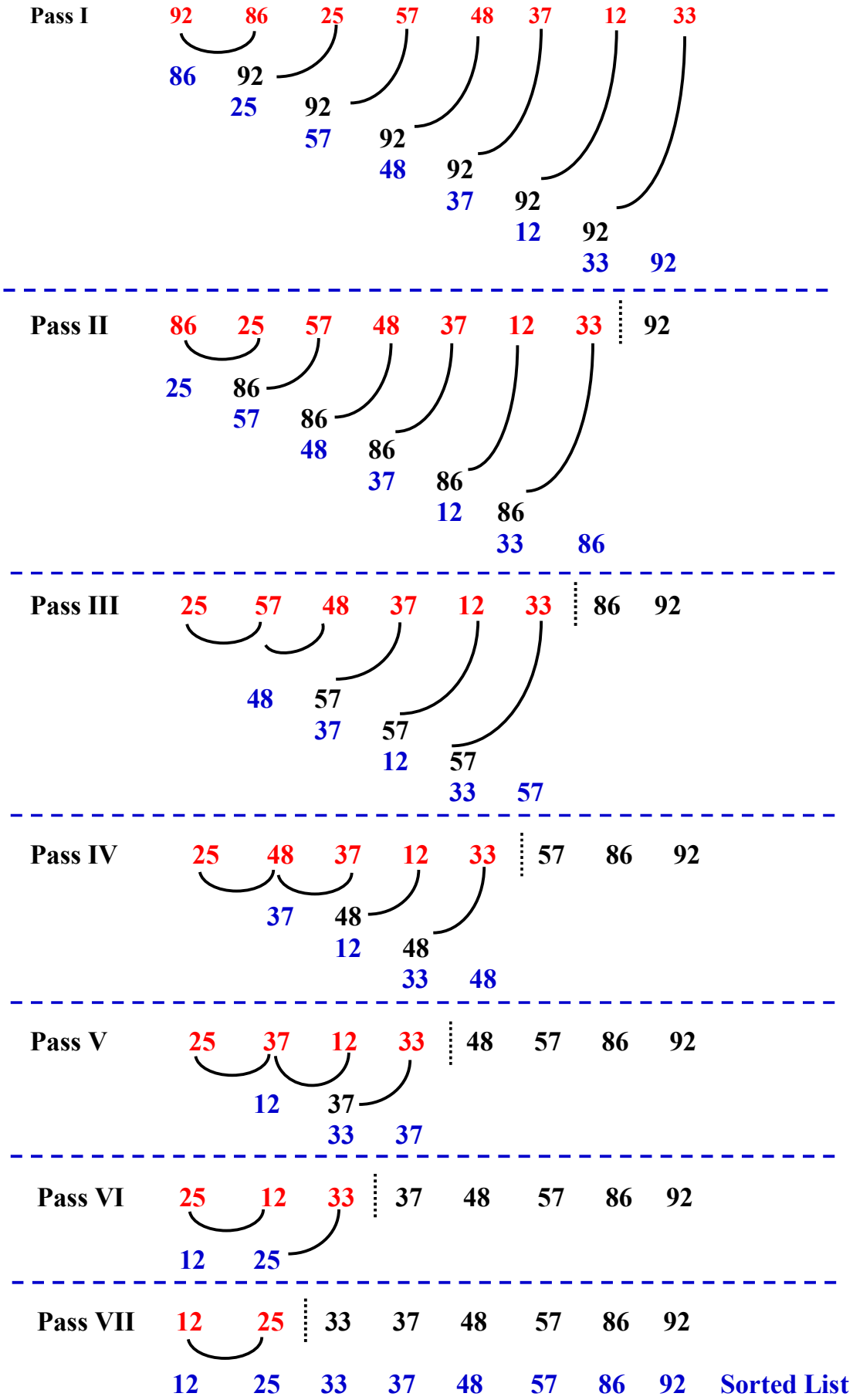
There are n-1 passes and n-1 comparisons on each pass. Thus the total no of comparisons is (n-1) X (n-1) = $\mathbf{n^2 - 2n + 1}$ which is $\mathbf{0(n^2).}$

```
/*-----------------------------------------
 Program to implement Bubble sort technique
-----------------------------------------*/
#include<stdio.h>
#include<conio.h>

bubblesort(a,n)
int a[],n;
{
  int i,j,temp;

  for(i=0;i<n;i++)
  {
    for(j=0;j<n-1-i;j++)
     if(a[j]>a[j+1])
     {
       temp=a[j];
       a[j]=a[j+1];
       a[j+1]=temp;
     }
   }
}

void main()
{
  int i,t,j,n,a[20];

  clrscr();

  printf("enter size of array ");
  scanf("%d",&n);

  printf("enter array elements:");
  for(i=0;i<n;i++)
  scanf("%d",&a[i]);

  bubblesort(a,n);

  printf("the sorted array is:");
  for(i=0;i<n;i++)
    printf("%d ",a[i]);

  getch();
}
```

OUTPUT

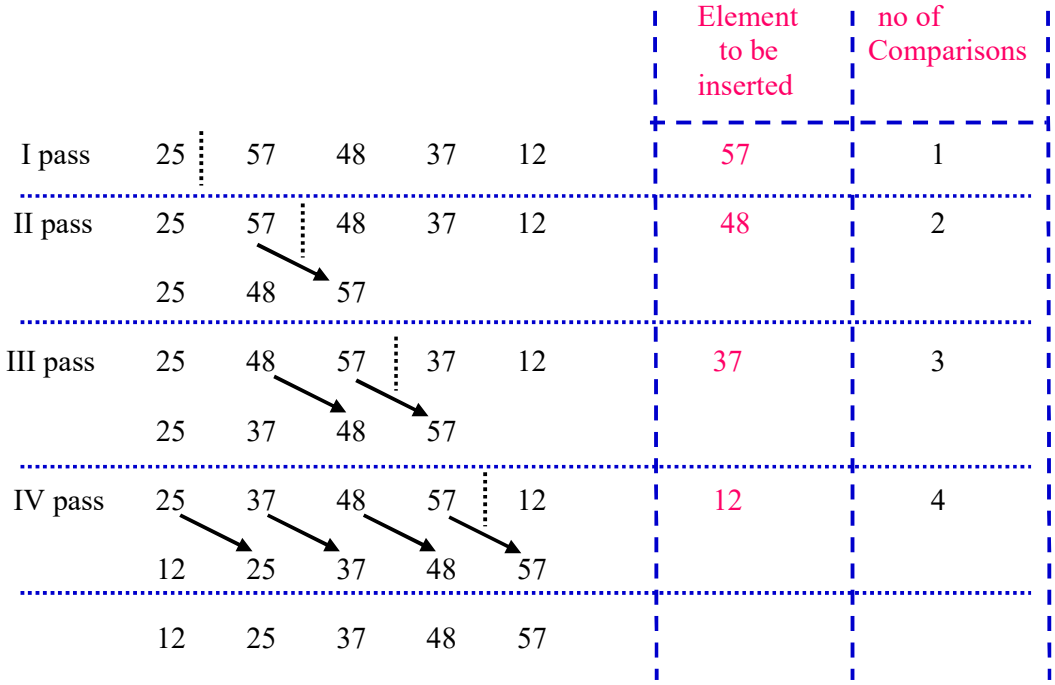
enter size of the array : **5**
enter array elements : **25 57 48 37 12**
The sorted array is : **12 25 37 48 57**

Ex2) sort the following list into ascending order using bubble sort



Insertion Sort:

The Insertion sort is one that sorts a set of elements by inserting elements into an existing sorted array. In this sorting, each time one element is inserted in its position into an existing sorted list. The simple Insertion sort is effective, if the list is small in size. This method uses very small memory and hence it is effective.

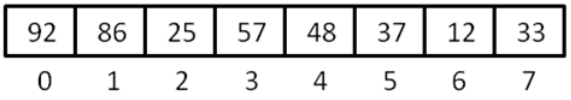


Total no of comparisons = 1 + 2 + 3 + ------- + (n-1)
= (n-1) n/2
= $(n^2-n)/2$
= **$O(n^2)$**

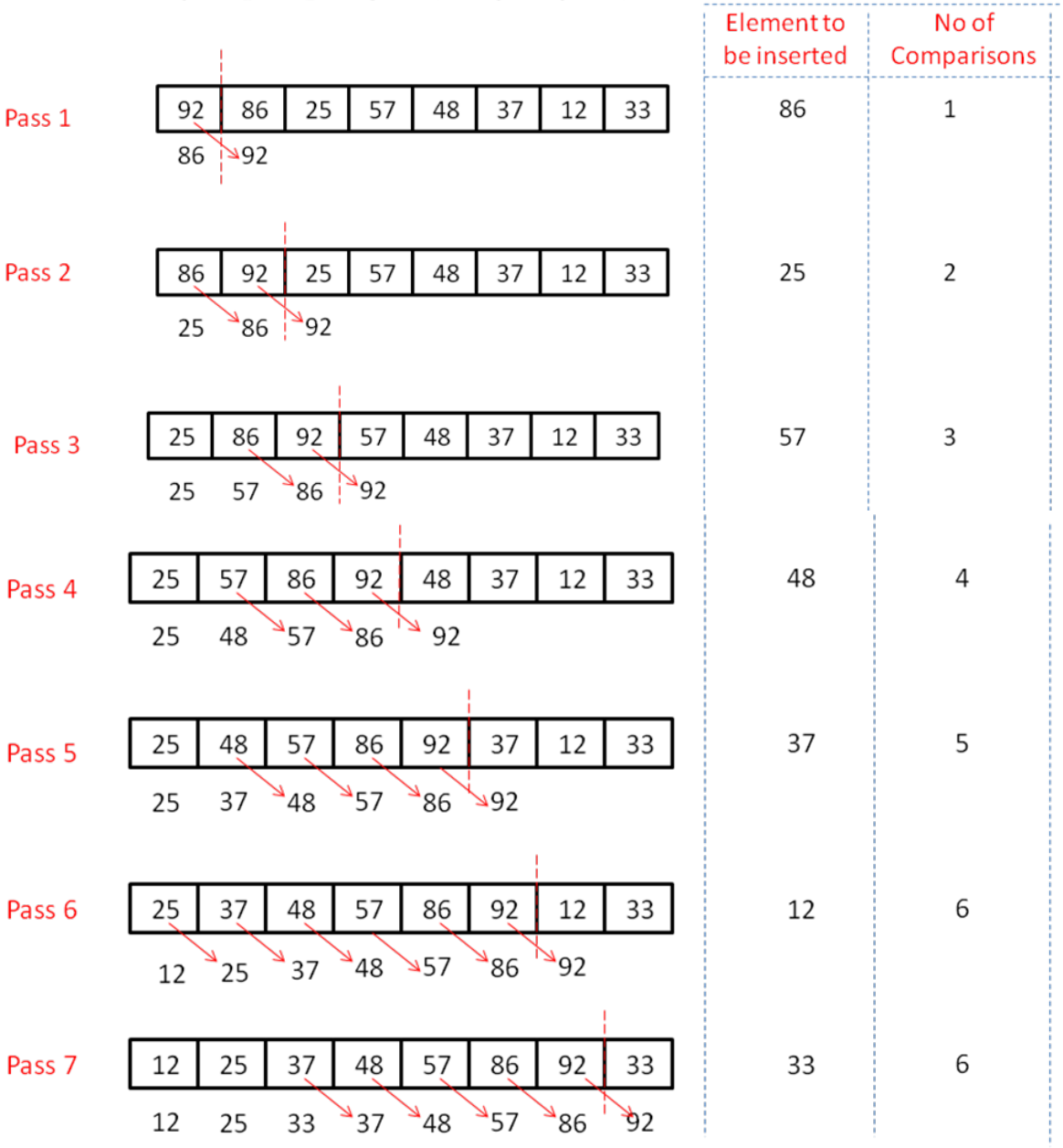
However the simple insertion sort is still better than the bubble sort. The average number of comparisons in the simple insertion sort is also

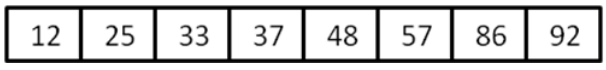
$O(n^2)$

Example 2) Sort the following array using Insertion sort technique



Un Sorted Array





Sorted Array

```
/*-----------------------------------------------
 Program to implement Insertion sort technique
-----------------------------------------------*/
# include <stdio.h>
#include <conio.h>
insertionsort (a,n)
int a[10];
int n;
{
  int i,k,y,found;
        for (k=1;k<n;k++)
        {
            y = a[k];
            i = k-1;
            found = 0;

            while ((i>=0) && (found !=1))
            {
             if (y<a[i])
              {
                    a[i+1] = a[i];
                     i --;
              }
             else
                  found = 1;
            }                          /*----------------------------end of while ---------*/
           a[i+1] = y;
        }                             /*-----------------------------------end of for ---------*/
}                                      /*-----------------------------------end of function--*/
void main()
{
        int a[10];
        int i, n;
        clrscr();
        printf("Enter n");
        scanf("%d",&n);

        printf("Enter array elements\n");
        for (i=0;i<n;i++)
           scanf("%d",&a[i]);

        printf("array before insertion sort\n");
        for (i=0;i<n;i++)
            printf("%d",a[i]);
        printf("\n");

        insertsort(a,n);

        printf("Array after insertion sort\n");
        for(i=0;i<n;i++)
        printf("%d",a[i]);
        printf("\n");
        getch();
}
```

OUTPUT
enter n **5**
enter array elements
25 57 48 37 12
Array before insertion sort
25 57 48 37 12
array after insertion sort
12 25 37 48 57

Selection Sort:

Selection sort minimizes data movement by putting one entry in its final position at every pass. This algorithm is primarily useful for contiguous lists with large entries for which movement of entries is expensive.

This algorithm uses a function called MAXKEY, which finds the maximum key on the list. The procedure swap simply swaps the two entries given as parameters.

The primary advantage of selection sort regards data movement. If an entry is in its correct final position, then it will never be moved. Every time any pair of entries is swapped, then at least one of them moves into its final position, and therefore n-1 swaps are done altogether in sorting a list of n elements.

At the first stage we scan the list to find the largest number. 57 is the largest number. Then exchange this number with the number in the last position. Now repeat the process on the shorter list by omitting the last entry. Again the largest number in the shorter list is exchanged with the last number in the shorter list and this process continues until the length of the unsorted list is 1.

Unsorted elements	25	57	48	37	12	
Pass I	25	(57)	48	37	12	interchange 57 and 12

57 is the largest element and it is interchanged with the last element of the array. 57 and 12 are swapped.

Pass II	25	12	(48)	37	57	interchange 48 and 37

48 is the largest element and it is interchanged with the last element of the shorter array. 48 and 37 will be interchanged.

Pass III	25	12	(37)	48	57	interchange 37 and 37

37 is the largest element and it is interchanged with the last element of the shorter array . 37 and 37 will be interchanged

Pass IV	(25)	12	37	48	57	interchange 25 and 12

25 is the largest element and it is interchanged with the last element of the shorter array. 25 and 12 will be interchanged

12	25	37	48	57	**Sorted list**

Note: The numbers in circles are the larget numbers of the array and they will be inter changed with the last elements of that list.

Analysis of Selection sort:

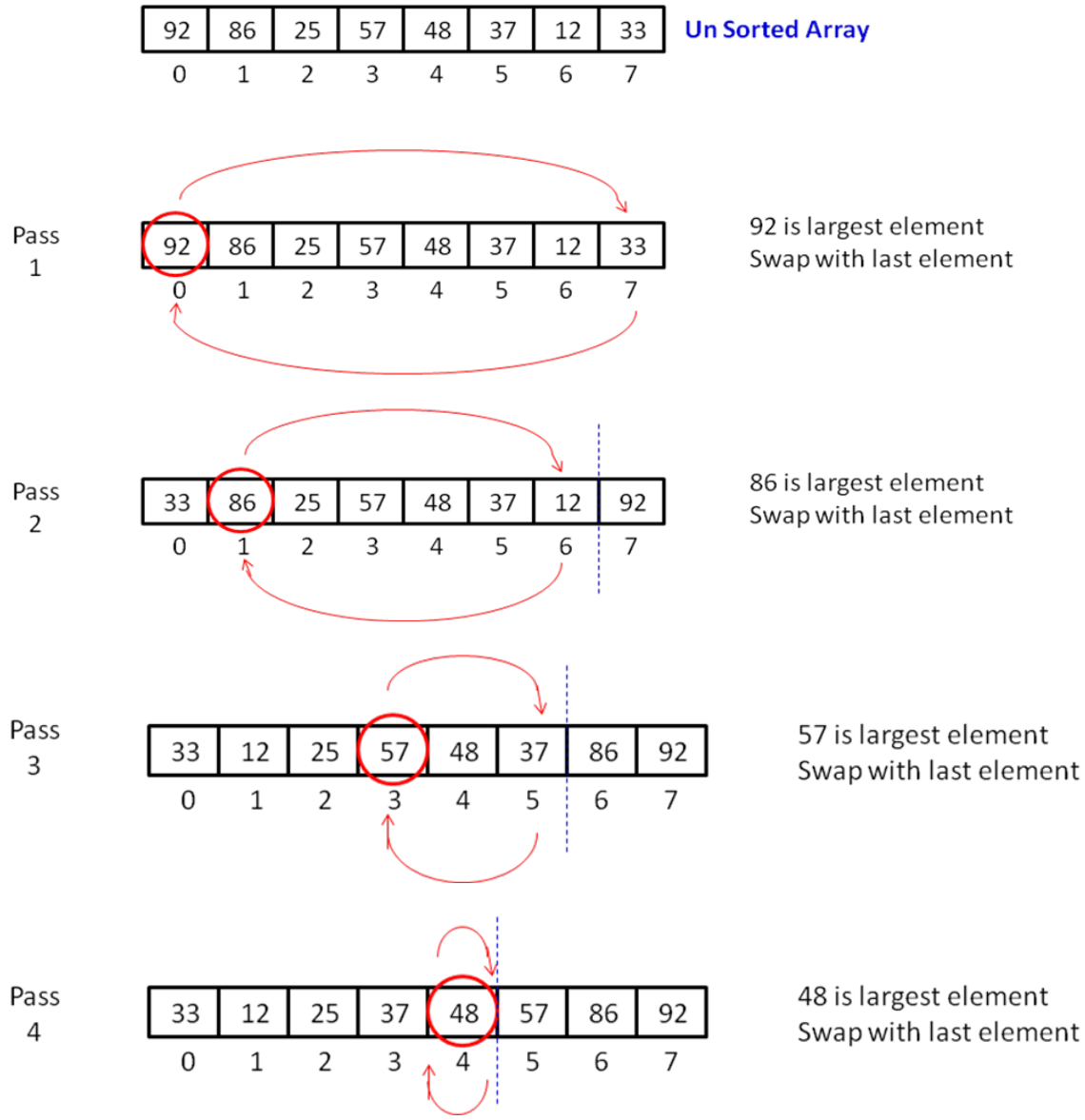
The first pass makes n-1 comparisons,The second pass makes n-2 and so on….Therefore, there is a total of

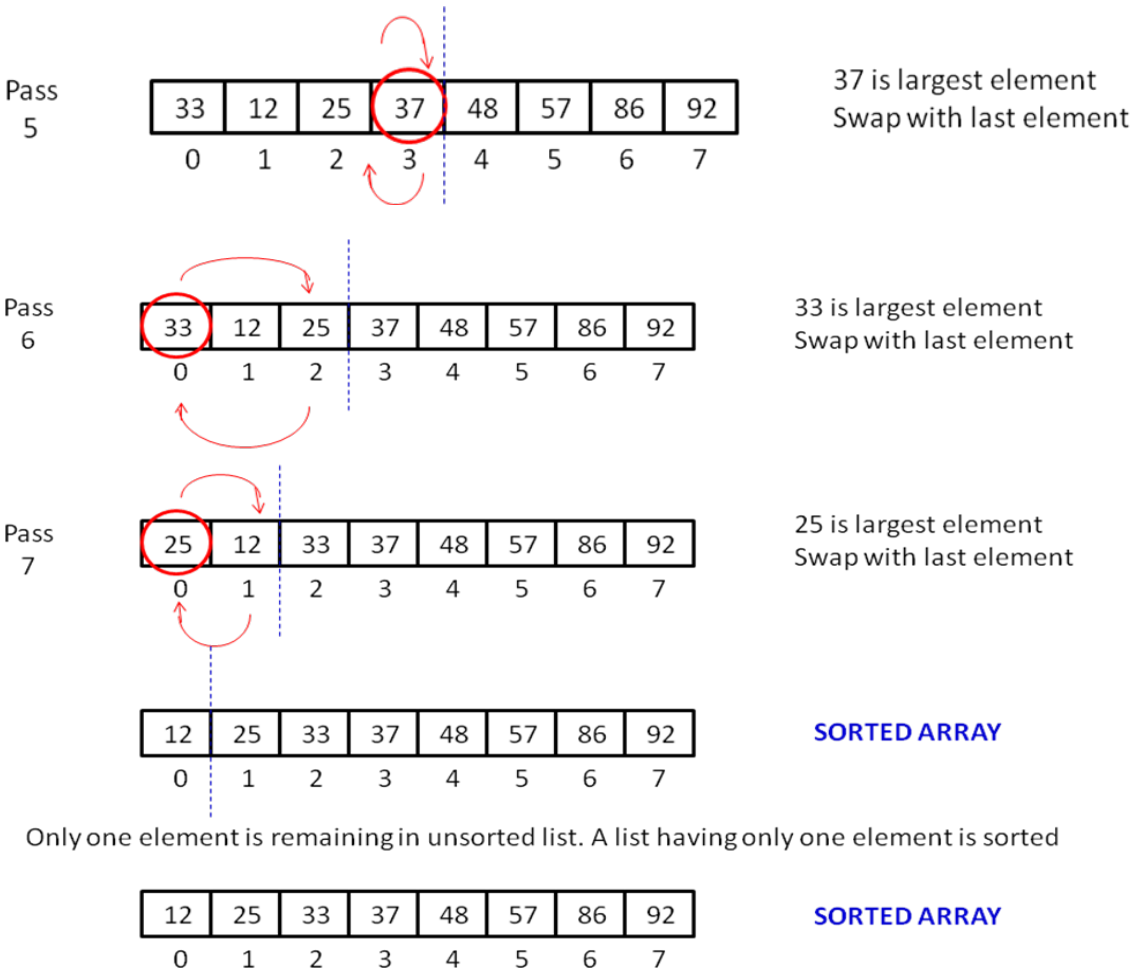
(n-1) + (n-2) + (n-3) + ……+1 = $\frac{n(n-1)}{2}$ comparisons, which is **$0(n^2)$.**

The number of interchanges is always (n-1). The sort therefore be categorized as

$$0(n^2)$$

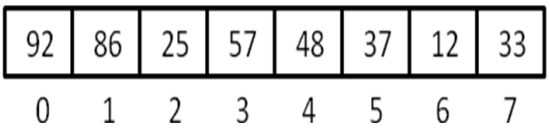
Example 2) Sort the following array using Selection sort technique

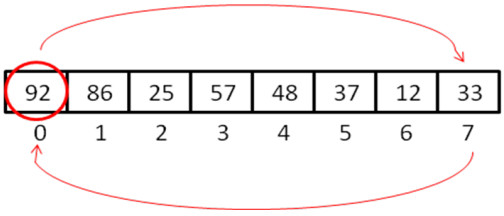


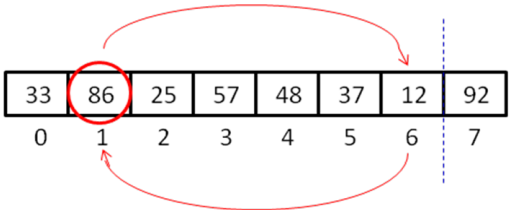


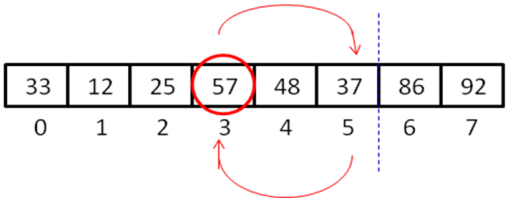
Only one element is remaining in unsorted list. A list having only one element is sorted

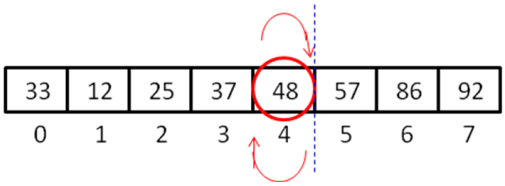
```
/*-------------------------------------------------------------------------------------
      Program to sort elements of an array using Selection Sort Technique
-------------------------------------------------------------------------------------*/
#include <stdio.h>
#include <conio.h>

void swap (int a[],int i, int j)
{
        int temp;

        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
}

int maxkey(int a[],int low, int high)
{
        int large, largepos, i;

        large = a[low];
        largepos = low;

        for (i=low+1; i<=high;i++)
           if (a[i] > large)
            {
                large  = a[i];
                largepos = i;
            }
        return (largepos);
}


void selectionsort(int a[], int n)
{
    int current, maxpos;

    for (current = n-1;current >=0;current--)
      {
        maxpos = maxkey(a,0, current);
        swap (a,current,maxpos);
      }
}
```

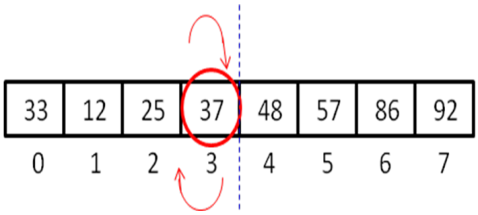


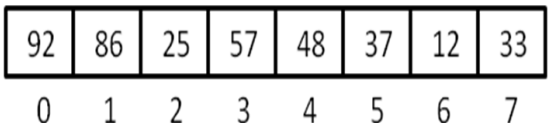




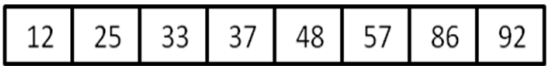









```
void main()
{
      int a[10];
      int i, n;

      clrscr();

      printf("Enter n");
      scanf("%d", &n);

      printf("Enter array elements\n");
      for(i=0;i<n;i++)
         scanf("%d",&a[i]);

      printf("array before  sorting\n");
      for (i=0;i<n;i++)
         printf("%d",a[i]);
      printf("\n");

      selectionsort(a,n);

      printf("Array after  sorting\n");
      for(i=0;i<n;i++)
         printf("%d ",a[i]);
      printf("\n");

      getch();
}
```



OUTPUT

```
enter n  8

enter array elements
92  86  25  57  48  37  12  33

Array before sorting
92  86  25  57  48  37  12  33

array after sorting
12  25  33  37  48  57  86  92
```

Quick Sort

The quick sort scheme developed by C.A.R.Hoare, has the best average behavior among all the sorting methods.

Let 'A' be an array and 'n' the number of elements in the array to be sorted. Choose an element 'x' from a specific position within the array i.e., x = a[0]

Suppose the elements of array 'a' are partitioned, such that 'x' is placed into position 'j' and the following conditions hold.

1. Each of the elements in positions '0' through 'j-1' is less than or equal to 'x'.
2. Each of the elements in positions 'j+1' through 'n-1' is greater than or equal to 'x'.

Notice that if these two conditions hold for a particular 'x' and 'j', 'x' is the j^{th} smallest element of array 'a', so that 'x' remains in position 'j' when the array is completely sorted. If the foregoing process is repeated with the sub arrays a[0] through a[j-1] and a[j+1] through a[n-1] and any sub arrays created by the process in successive iterations, the final result is a sorted array.

Let us illustrate the quick sort with an example.

25 57 48 37 12 92 86 33

First element is placed in its proper position.

12 | 25 | 57 48 37 92 86 33

At this point 25 is in its proper position in the array a[1], each element below that position is less than or equal to 25 and each element above that position is greater than or equal to 25 Since 25 is in its final position the original problem is decomposed into the problem of sorting two arrays.

(12) and (57 48 37 92 86 33)

The array of one element is already sorted. To sort the second sub array the process is repeated and the sub array is further sub divided. The entire array is now be viewed as

12 25 (57 48 37 92 86 33)

Where parentheses enclose the sub arrays that are yet to be sorted.

12 25 (48 37 33) | 57 | (92 86)

and further representations are

12 25 (37 33) 48 57 (92 86)

12 25 (33) 37 48 57 86 (92)

12 25 33 37 48 57 86 92

Note that the final array is sorted. By this time you should noticed that the quick sort may be defined most conveniently as a recursive procedure.

Efficiency of Quick Sort is **0(n log n).**

Assume that the file size is n and n is a power of 2 say

$n = 2^m$

apply log both sides

$\log n = \log_2 2^m$

$m = \log_2 n$ can be read as m = log n base 2.

for example 8 is 2^3. m=log n $m=\log_2 2^3$ $m= 3 \log_2 2$ m=3

And also assume that the proper position for pivot always points the exact middle element of the sub array. In that case there will be approximately n comparisons. Actually n-1 comparisons.

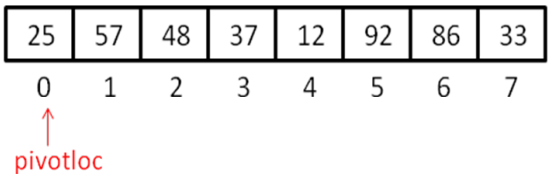
The file of size n will be splitted into 2 subfiles each of size n/2.
The file of size n/2 will be splitted into 2 subfiles each of size n/4.
The file of size n/4 will be splitted into 2 subfiles each of size n/8.
The file of size n/8 will be splitted into 2 subfiles each of size n/16.
And so on until the subfile having size 1.

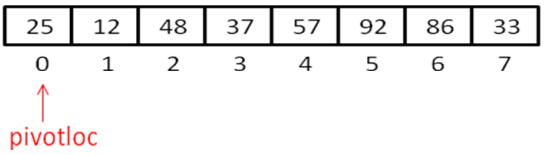
After having the subfile m times there are n files of size 1. Thus the total no of comparisions for the entire sort is approximately

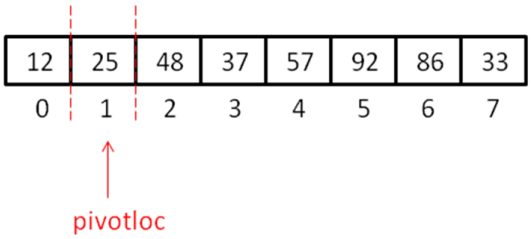
= n + 2*n/2 + 4 * n/4 + 8*n/8 + ……n*n/n upto m terms
= n * m = n*log n

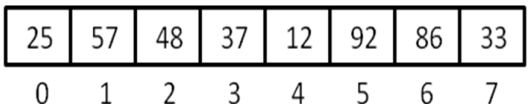
Time Complexity is **O(n logn)**

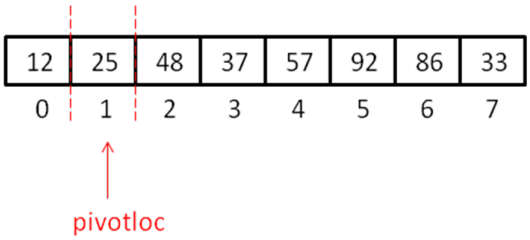
```
/*-------------------------------------------------------------------------------------
        Program to sort elements of an array using Quick Sort Technique
-------------------------------------------------------------------------------------*/

#include<stdio.h>
#include<conio.h>

void swap(a,i,j)

int a[],i,j;
{
   int temp;
   temp=a[i];
   a[i]=a[j];
   a[j]=temp;
}

int partition(a,low,high)
int a[],low,high;
{
  int pivotloc,pivotkey,i;
  pivotkey=a[low];
  pivotloc=low;
  for(i=low+1; i<=high; i++)
   {
     if (a[i]<pivotkey)
     {
           swap(a,pivotloc+1,i);
           swap(a,pivotloc,pivotloc+1);
           pivotloc++;
     }
   }
   return(pivotloc);
}

void quicksort(a,low,high)
int a[],low,high;
{
 int pivotloc;

 if (low<high)
   {
    pivotloc=partition(a,low,high);
    quicksort(a,low,pivotloc-1);
    quicksort(a,pivotloc+1,high);
   }
}
```











```
void main()
{
    int a[20],n,i;

    clrscr();

    printf("enter size of array A");
    scanf("%d",&n);

    printf("enter elements of array A \n");
    for(i=0;i<n;i++)
      scanf("%d",&a[i]);

    quicksort(a,0,n-1);       /*---------------------------------- function call */

    printf("the sorted array is:");
    for(i=0;i<n;i++)
       printf("%d ",a[i]);
    printf("\n");

    getch();
  }
```

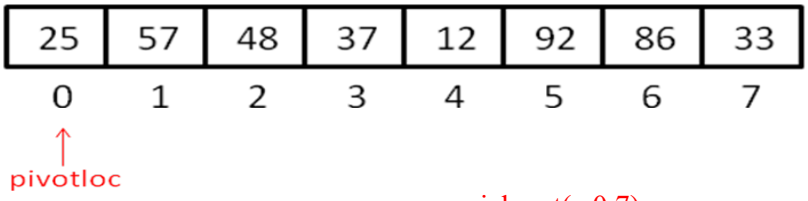
OUTPUT

enter size of the array A **8**
enter elements of array A
25 57 48 37 12 92 86 33

The sorted array is
12 25 33 37 48 57 86 92

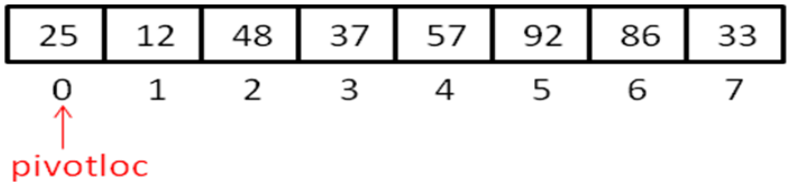
Example: sorting 25 57 48 37 12 92 86 33 using quick sort method. diagrammatic explanation at each stage of quicksort

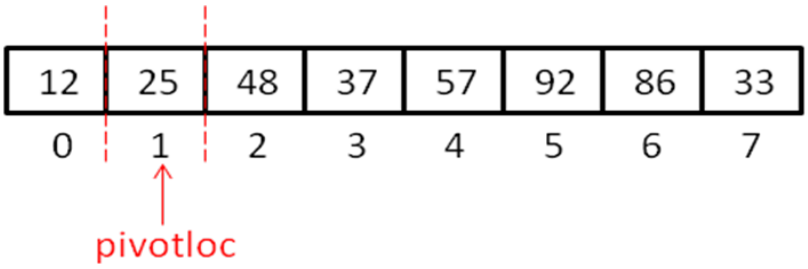
Un Sorted Array



quicksort(a,0,7)

is 57 < 25 no
is 48 < 25 no
is 37<25 no
is 12 < 25 yes -- swap(a,pivotloc+1,i); 12 and 57 will be swapped
swap(a,pivotloc,pivotloc+1); 12 and 25 will be swapped



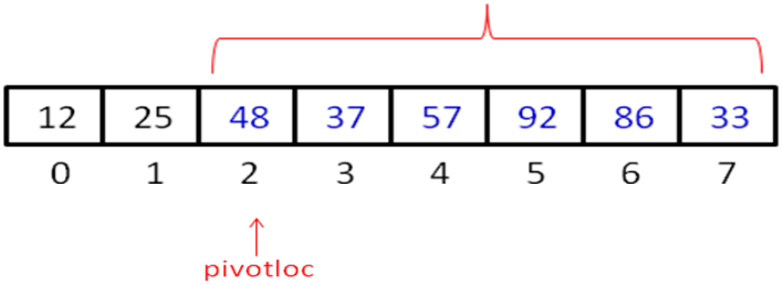


25 is fixed in its location such that it divides the array into 2 partitions

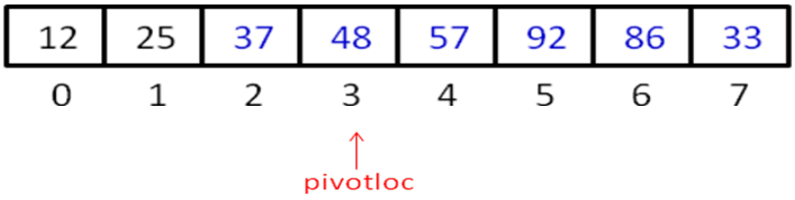
one partition contains 12 quicksort(a,0,0)

other partition contains 48 37 57 92 86 33 quicksort(a,2,7)

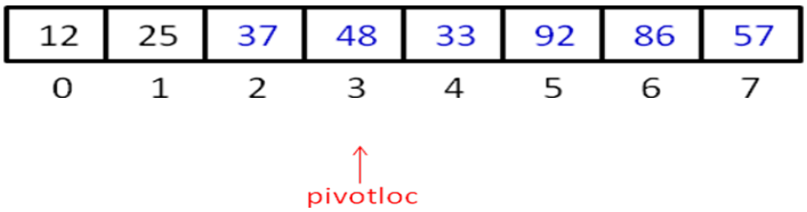
apply quick sort function on both the partitions.
remember that the partition contains only one element need not be sorted as it is already sorted. Apply quicksort on second partition.

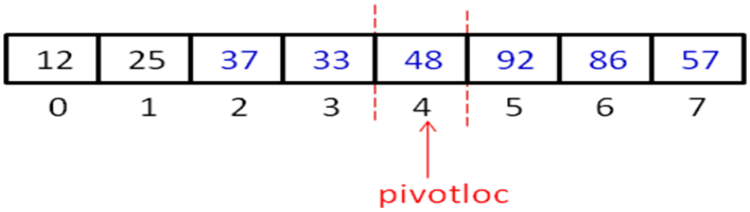


is 37 <48 yes swap(a,pivotloc+1,i); 37 and 37 will be swapped
swap(a,pivotloc,pivotloc+1); 48 and 37 will be swapped

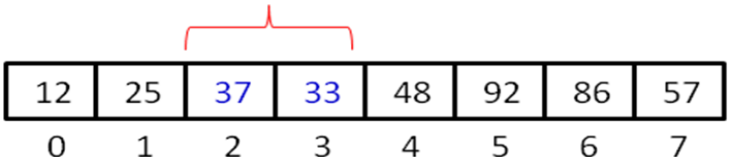


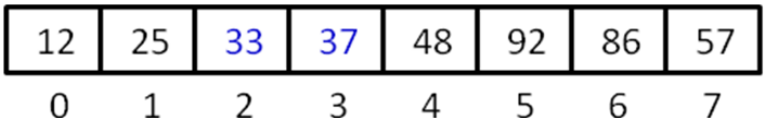
is 57 < 48 no
is 92 < 48 no
is 86 < 48 no
is 33 < 48 yes swap(a,pivotloc+1,i); 57 and 33 will be swapped
swap(a,pivotloc,pivotloc+1); 48 and 57 will be swapped

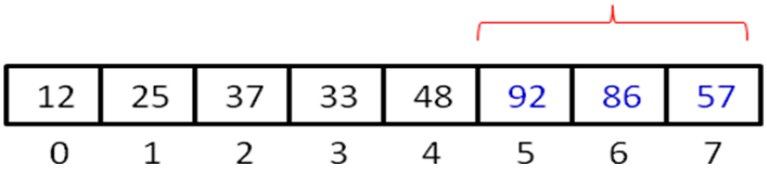


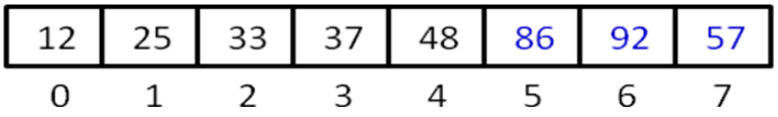


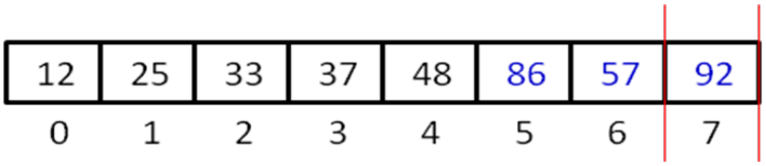
48 is placed in its proper position such that it divides the array into two partitions.
first partition contains 37 , 33, **quicksort(a,2,3)**
Second partition contains 92, 86, 57. **quicksort(a,5,7)**
Apply quick sort function on both partitions.

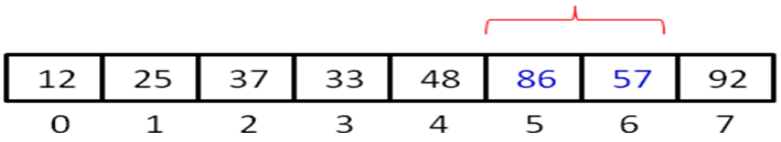




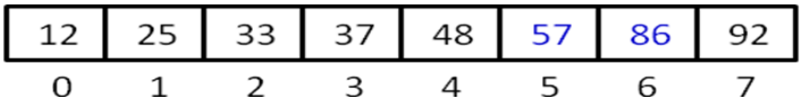








quicksort(a,5,6)



Sorted Array

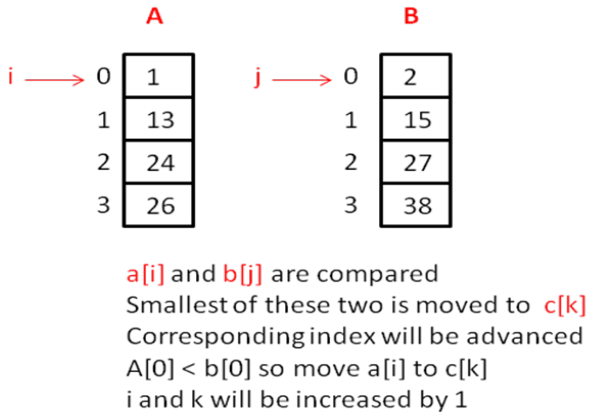
Merging two sorted arrays into one Sorted Array

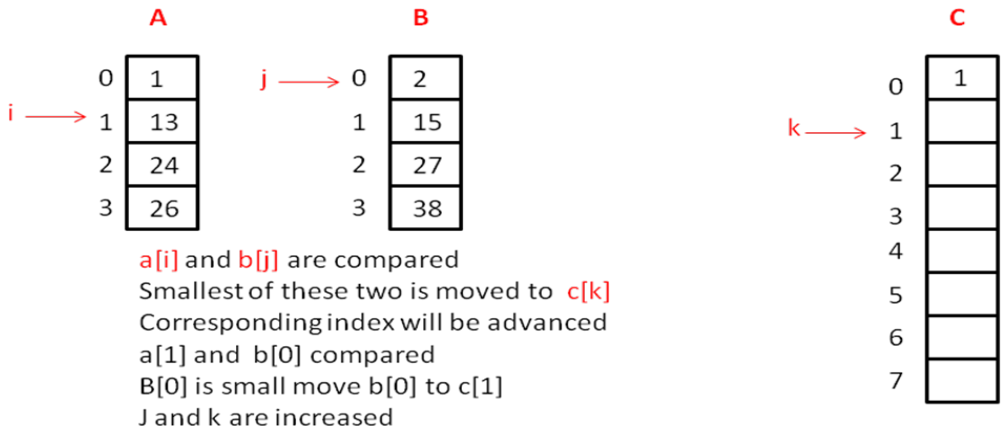
The fundamental operation in this algorithm is merging two sorted lists. The basic merging algorithm takes two input arrays A , B and an output array C and three counters Aptr, Bptr and Cptr which are initially set to the beginning of their respective arrays. The smaller of A[Aptr] and B[Bptr] is copied to the next entry in C, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to C.

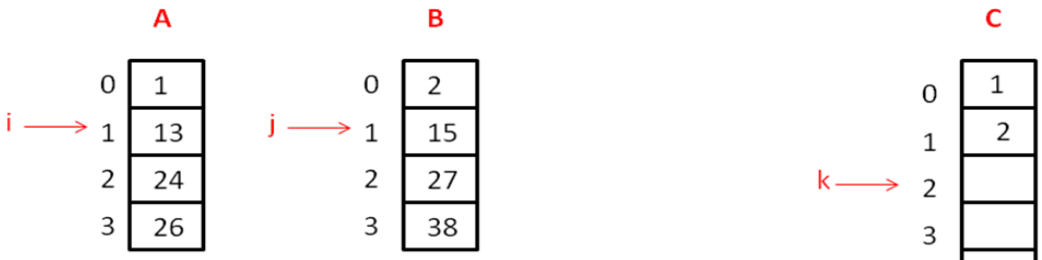
A	B	C
→1	→ 2	→1
13	15	2
24	27	13
26	38	15
		24
		26
		27
		38

At most N-1 comparisons are made, where N is the total number of elements. Every comparison adds an element to C, except the last comparison, which adds at least two. Merge sort running time = **0(n log n).**

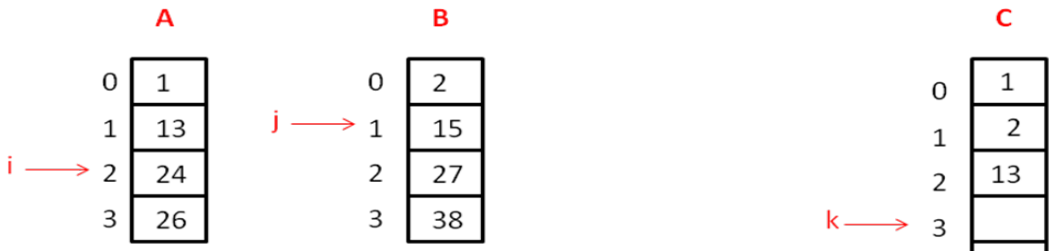
Merge contents of array A and array B into array C in ascending order



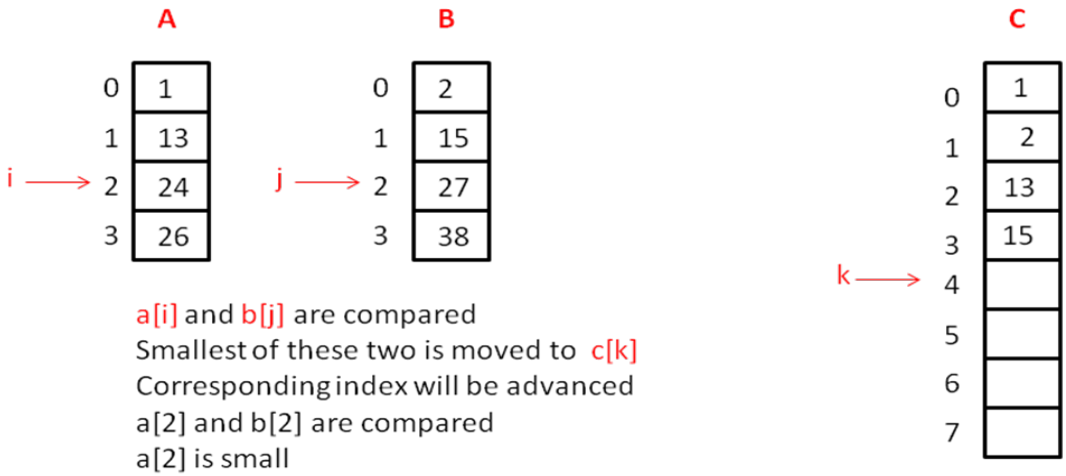




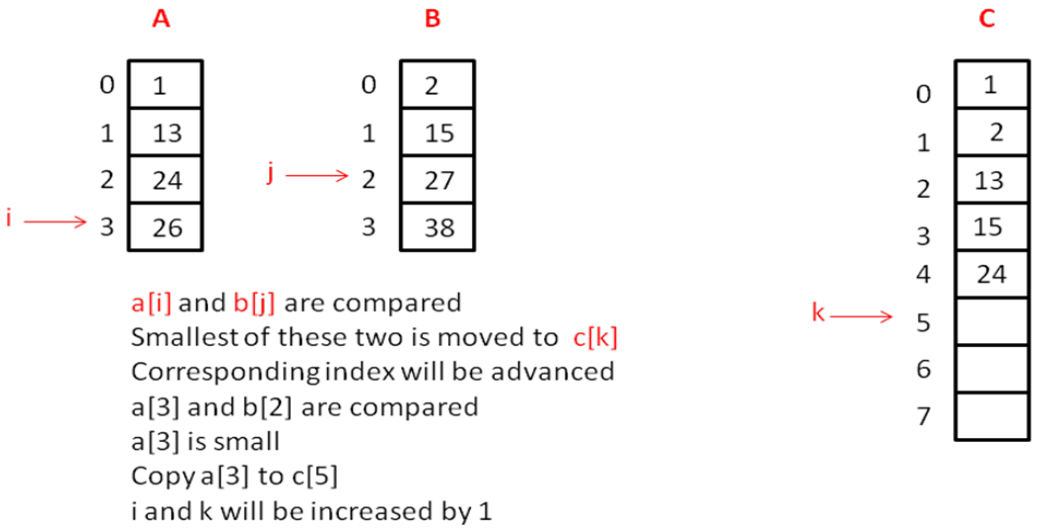
a[i] and b[j] are compared
Smallest of these two is moved to c[k]
Corresponding index will be advanced
a[1] and b[1] are compared
A[1] is small
Copy a[1] to c[2]
i and k will be increased by 1



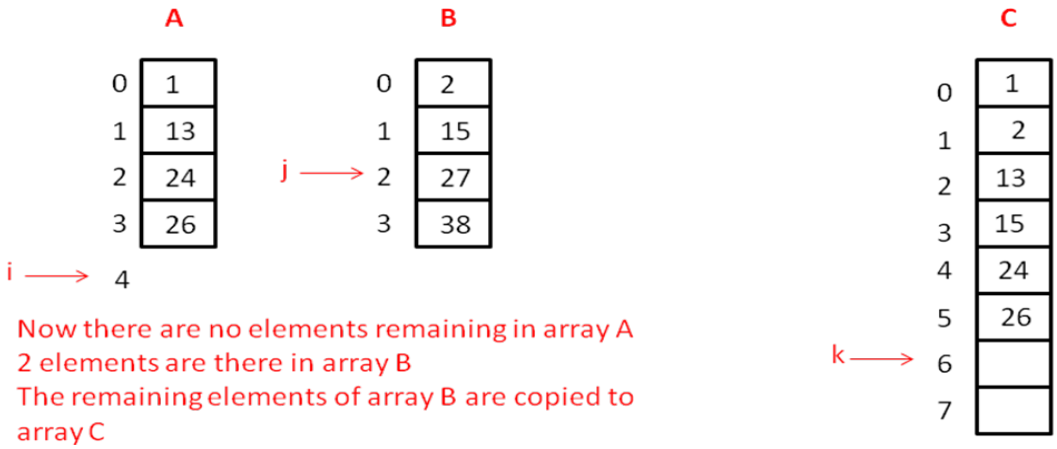
a[i] and b[j] are compared
Smallest of these two is moved to c[k]
Corresponding index will be advanced
a[2] and b[1] are compared
b[1] is small
Copy b[1] to c[3]
j and k will be increased by 1



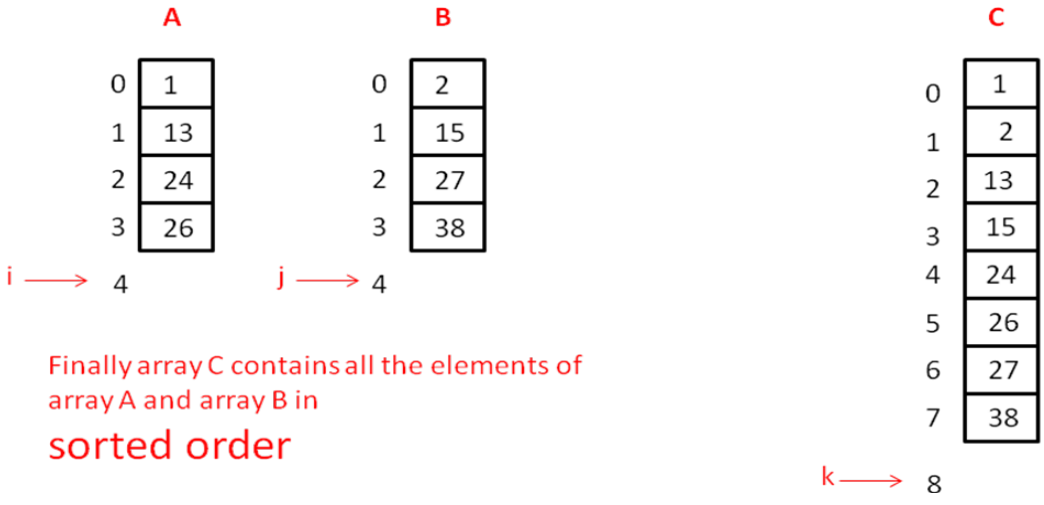
a[i] and b[j] are compared
Smallest of these two is moved to c[k]
Corresponding index will be advanced
a[2] and b[2] are compared
a[2] is small
Copy a[2] to c[4]
i and k will be increased by 1



a[i] and b[j] are compared
Smallest of these two is moved to c[k]
Corresponding index will be advanced
a[3] and b[2] are compared
a[3] is small
Copy a[3] to c[5]
i and k will be increased by 1



Now there are no elements remaining in array A
2 elements are there in array B
The remaining elements of array B are copied to array C



Finally array C contains all the elements of array A and array B in

sorted order

```
/*-------------------------------------------------------
Program to Merge two sorted arrays in to one sorted array.
Destination is a sorted array
-------------------------------------------------------*/
#include<stdio.h>
#include<conio.h>
void merge(int a[],int n,int b[],int m,int c[])
{
  int i,j,k;

  i=0;
  j=0;
  k=0;

  while ((i<n)&&(j<m))
   {
     if (a[i] < b[j])
       {
          c[k]=a[i];
          i++;
          k++;
       }
     else
      {
         c[k]=b[j];
         j++;
         k++;
      }
   }



   while (i<n)
    {
      c[k]=a[i];
      i++;
      k++;
    }

   while (j<m)
    {
      c[k]=b[j];
      j++;
      k++;
    }
}
```

```
void main()
{
   int a[20],n,b[20],m,c[20],i;

   clrscr();
   printf("enter size of array A");
   scanf("%d",&n);

   printf("enter elements of array A in sorted order : \n");
   for(i=0;i<n;i++)
     scanf("%d",&a[i]);

   printf("enter size of array B");
   scanf("%d",&m);

   printf("enter elements of array B in sorted order: \n");
   for(i=0;i<m;i++)
      scanf("%d",&b[i]);

   merge(a,n,b,m,c); /* function call */

   printf("the merged array is:");
   for(i=0;i<m+n;i++)
       printf("%d ",c[i]);
   getch();
 }
```

OUTPUT

```
enter size of array A  4
enter elements of array A in sorted order
1  13  24  26

enter size of array B  4
enter elements of array B in sorted order
2  15  27  38

The merged array is
1  2  13  15  24  26  27  38
```

MERGE SORT

Merge Sort is an efficient, general-purpose, and comparison based sorting algorithm. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

In sorting n objects, merge sort has an average and worst-case performance of **O(n log n)**.

If the running time of merge sort for a list of length n is $T(n)$, then the recurrence relation

$T(n) = 2T(n/2) + n$

follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists).

The number of comparisons made by merge sort in the worst case is given by the sorting numbers. These numbers are equal to or slightly smaller than

$(n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1)$, which is between

$(n \log n - n + 1)$ and $(n \log n + n + O(\log n))$.

Merge sort's best case takes about half as many iterations as its worst case

Worst Case time complexity **O(n log(n))**

Assume that the elements are to be sorted in non decreasing order.

Assume that the Array contains n elements a[1], a[2],……a[n]. Imagine them split in to two sub arrays.

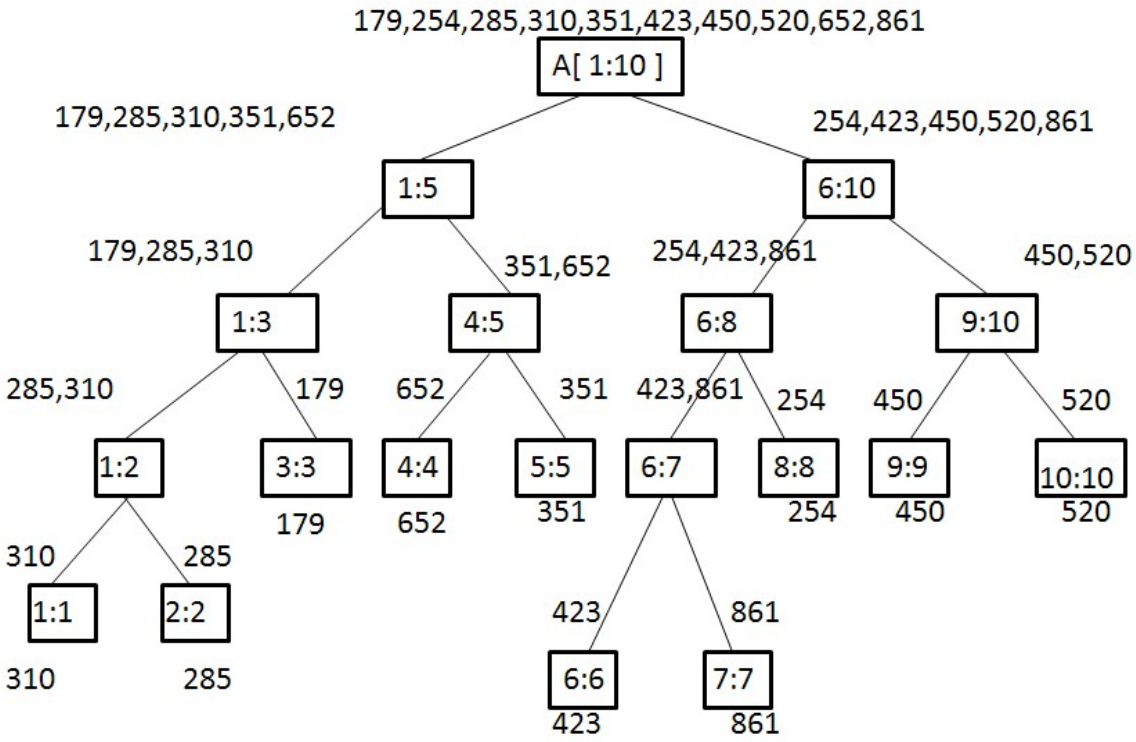
a[1], a[2] ….a[n/2]

a[n/2+1]……a[n]

Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elements. Thus we have another ideal example of divide and conquer strategy in which the splitting is into two equal sized sets and the combining operation is the merging of two sorted sets into one.

Example
A[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)
A[1:5]=(310, 285, 179, 652, 351)
A[6:10]=(423, 861, 254, 450, 520)

Tree of calls of Merge Sort

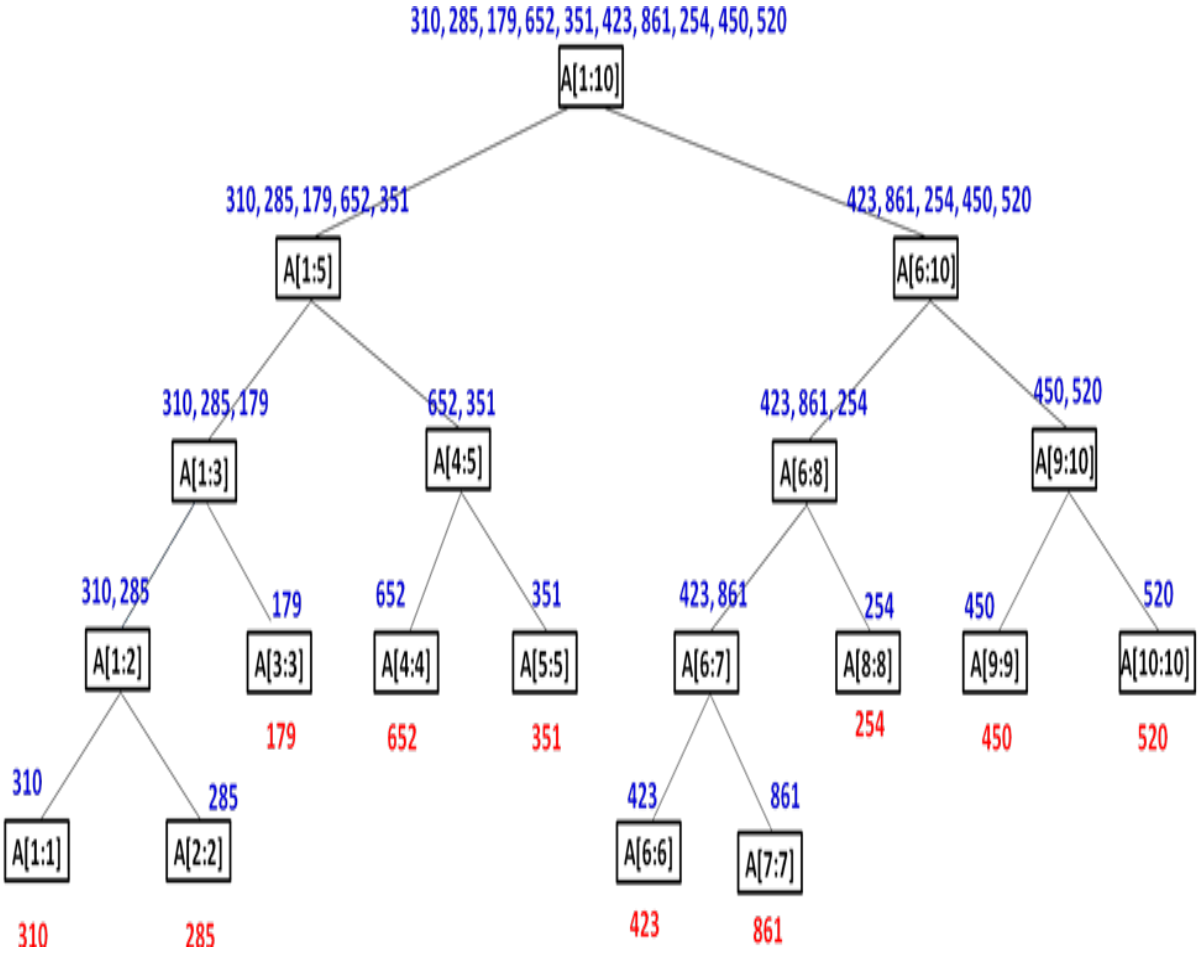


Array is divided into 10 arrays such that each array contains only one element each. An array consisting of only one element is sorted array.

now combine these sorted arrays to form bigger sorted array.
start merging from last array.

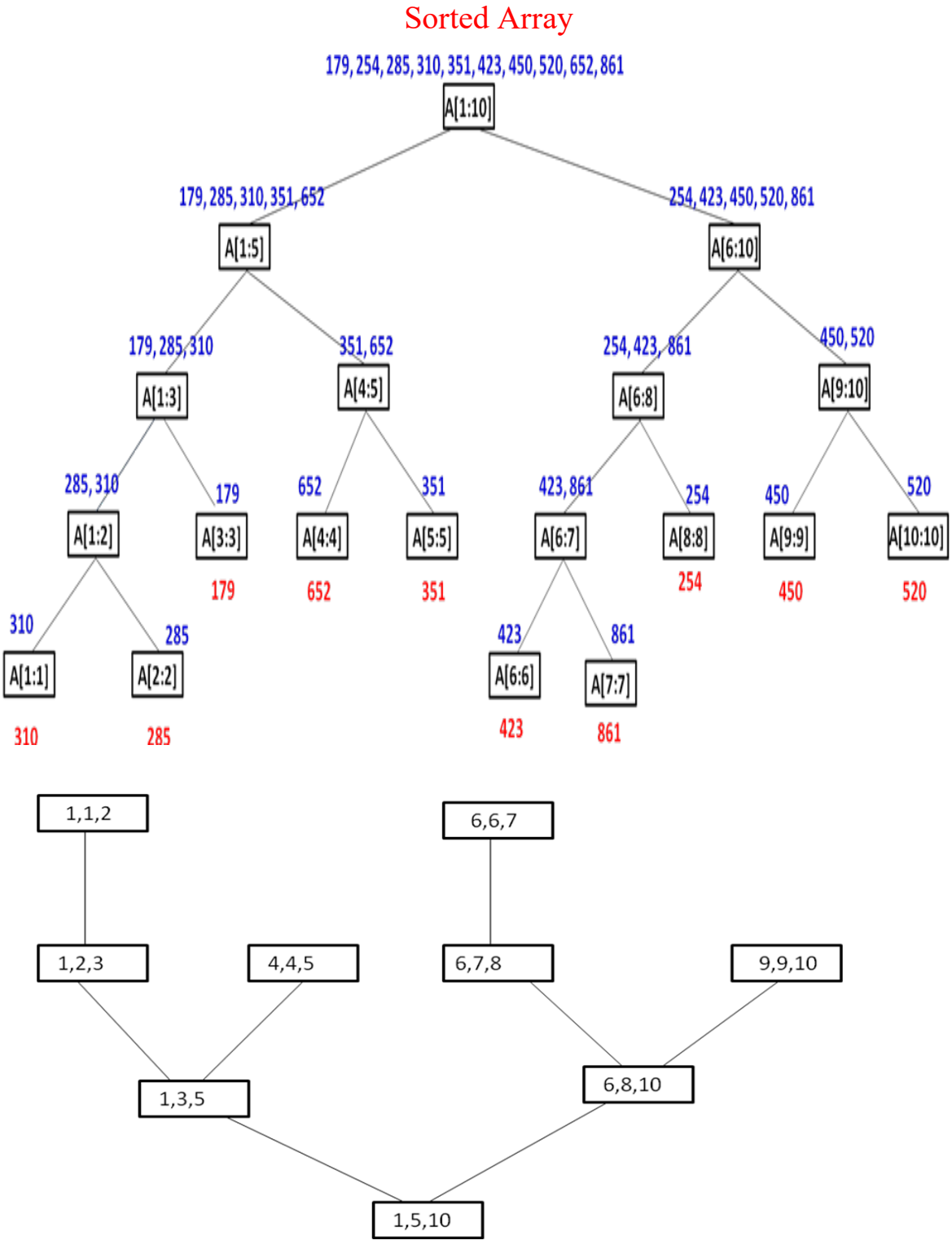
array	310	285	179	652	351	423	861	254	450	520
index	0	1	2	3	4	5	6	7	8	9

Unsorted Array

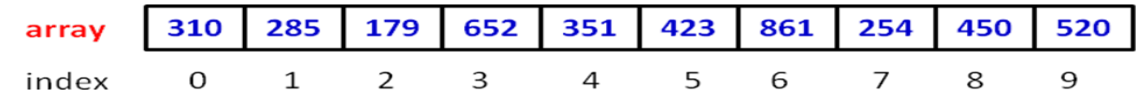


Now the merge begins. Note that no movement of data has yet taken place. A record of the sub arrays is implicitly maintained by the recursive mechanism.
(310| 285| 179|, 652, 351| 423, 861, 254, 450, 520)
(285, 310| 179|, 652, 351 | 423, 861, 254, 450, 520)
(179, 285, 310| 652, 351 | 423, 861, 254, 450, 520)
(179, 285, 310| 351, 652 | 423, 861, 254, 450, 520)
(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)
(179, 285, 310, 351, 652 | 423| 861| 254| 450, 520)
(179, 285, 310, 351, 652 | 423, 861| 254| 450, 520)
(179, 285, 310, 351, 652 | 254, 423, 861| 450, 520)
(179, 285, 310, 351, 652 | 254, 423, 450, 520, 861)
Final
(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

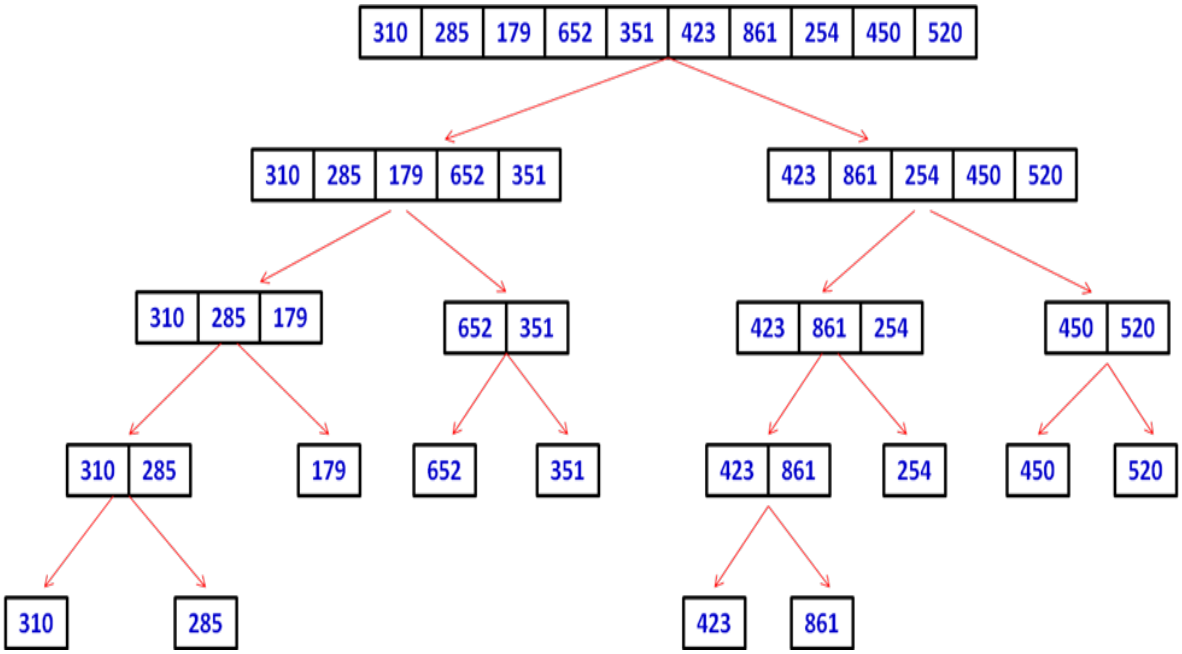
Back substitution
follow from bottom to top
last nodes contain one element each
merge starts from last elements
merge from bottom to top



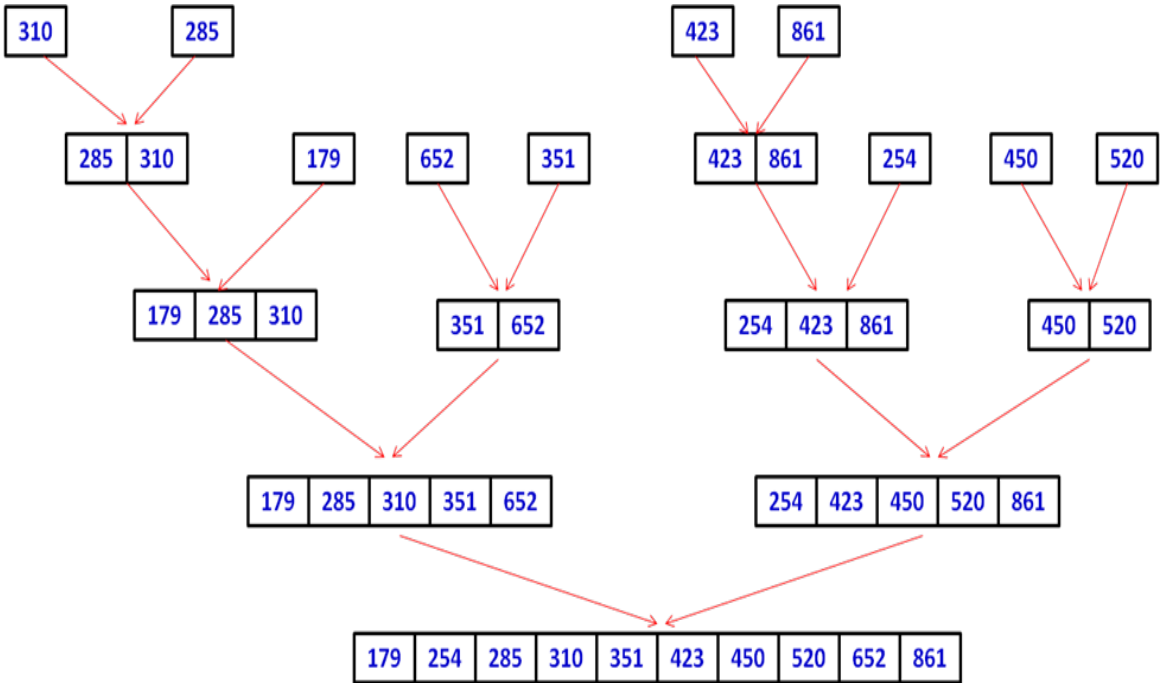
Tree of calls of Merge



To sort the elements of an array using merge sort technique, first split the array into number of arrays such that the final smallest array contains sorted elements. We know that an array contains 1 element is sorted.



The array has been splitted into 10 arrays. Each array contains exactly one element. An array contains one element is a sorted array. Now apply merge technique on the multiple sorted arrays to merge into one sorted array. Go back from lowest level to highest level by merging of elements.



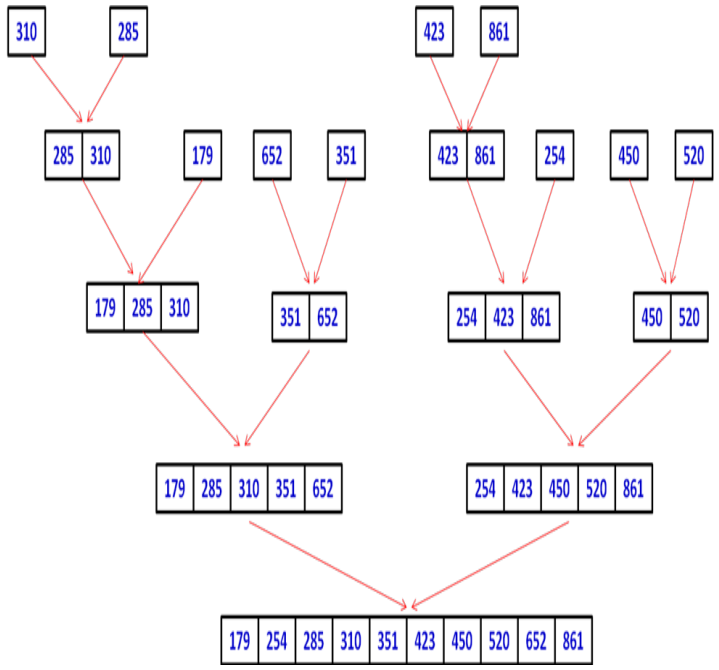
Final sorted array using merge sort technique

Algorithm for Merge Sort

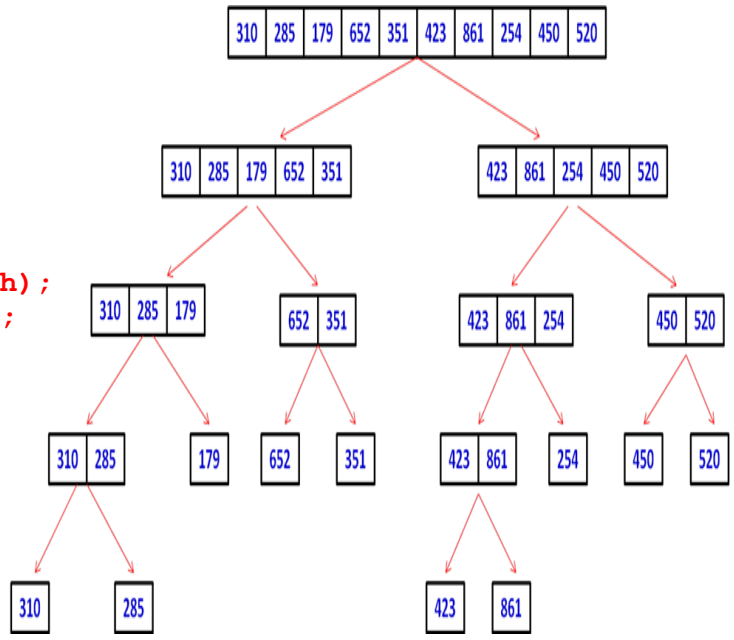
```
Algorithm mergesort(low,high)
//a[low:high] is a global array
{
    if (low<high) then
     {
          mid:= (low+high)/2;
          mergesort(low,mid);
          mergesort(mid+1,high);
                    //combine  the solutions
         merge(low,mid,high);
     }
}
```

```
Algorithm merge(low,mid,high)
{
     h:=low;
     i:= low;
     j:= mid+1;
     while ((h<=mid)and(j<=high)) do
      {
          if (a[h] <= a[j]) then
            {
                b[i]:=a[h];
                h:=h+1;
            }
          else
             {
                b[i]:=a[j];
                h:=h+1;
            }
          i:=i+1;
      }
  if (h>mid) then
     for k:= j to high do
      {
          b[i]:=a[k];
            i:=i+1;
      }
  else
       for k:= h to mid do
      {
            b[i]:=a[k];
            i:=i+1;
      }
    for k:= low to high do
        a[k]:=b[k];
}
```

```
/*************************************************************
        Sorting an array using Merge Sort Technique
*************************************************************/
#include <stdio.h>
#include <conio.h>

void merge(a,low,mid,high)
int a[50],low,mid,high;
{
      int i,j,k;
      int temp[50];

      k=low;
      i=low;
      j=mid+1;
      while ((i<=mid)&&(j<=high))
      {
        if (a[i]<=a[j])
        {
            temp[k]=a[i];
            i++;
            k++;
        }
        else
        {
            temp[k]=a[j];
            j++;
            k++;
        }
      }

      while(i<=mid)
      {
        temp[k]=a[i];
        i++;
        k++;
      }

      while(j<=high)
      {
        temp[k]=a[j];
        j++;
        k++;
      }

      for(k=low;k<=high;k++)
      a[k]=temp[k];
}
```



```
void mergesort(a,low,high)
int a[50],low,high;
{
  int mid;

  if (low<high)
  {
    mid=(low+high)/2;
    mergesort(a,low,mid);
    mergesort(a,mid+1,high);
    merge(a,low,mid,high);
  }
}

void main()
{
   int a[50], i,n;

   clrscr();
   printf("enter n \n");
   scanf("%d",&n);

   printf("enter array elements \n");
   for(i=0;i<n;i++)
      scanf("%d",&a[i]);

   mergesort(a,0,n-1);

   printf("The sorted array is\n");
   for(i=0;i<n;i++)
      printf("%d\n",a[i]);

   getch();
}
```



OUTPUT

```
Enter n
10

enter array elements
310  285  179  652  351  423  861  254  450  520

The sorted array is

179  254  285  310  351  423  450  520  652  861
```

Time Complexity of Merge Sort

If the time for merging operation is 4n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & \text{if } n=1 \\ 2T(n/2)+cn & \text{if } n>1 \end{cases}$$

Where n is a power of 2 i.e $n=2^k$. We can solve this equation by successive substitutions

$T(n) = 2T(n/2)+cn$ ----------1

Replace n with n/2

$T(n/2) = 2T(n/4)+cn/2$ ----------2

Substituting T(n/2) in equation 1

$T(n) = 2(2T(n/4)+cn/2)+cn$

$T(n) = 4T(n/4)+2cn$ ----------3

Replace n with n/4 in eq 1

$T(n/4) = 2T(n/8)+cn/4$ ----------4

Substituting T(n/4) in equation 3

$T(n) = 4(2T(n/8)+cn/4)+2cn$

$T(n) = 8T(n/8)+4cn$ ----------5

:

:

:

:

$T(n) = 2^k(1)+k.c.n$ $\quad n=2^k$

$= n.a + \log n \,.\, c.\, n$ $\quad k=\log n$

$= an + c\, n \log n$

It is easy to see that is $2^k <= n <= 2^{k+1}$ then

$T(n) <= T(2^{k+1}) >= T(N)$

$= \mathbf{O(n \log n)}$

RADIX SORT

Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However, the process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit.

To appreciate Radix Sort, consider the following analogy:

Suppose that we wish to sort a deck of 52 playing cards (the different suits can be given suitable values, for example 1 for Diamonds, 2 for Clubs, 3 for Hearts and 4 for Spades). The 'natural' thing to do would be to first sort the cards according to suits, then sort each of the four separate piles, and finally combine the four in order. This approach, however, has an inherent disadvantage. When each of the piles is being sorted, the other piles have to be kept aside and kept track of. If, instead, we follow the 'counterintuitive' approach of first sorting the cards by value, this problem is eliminated. After the first step, the four separate piles are combined in order and then sorted by suit. If a stable sorting algorithm (i.e. one which resolves a tie by keeping the number obtained first in the input as the first in the output) it can be easily seen that correct final results are obtained.

The sorting of numbers proceeds by sorting the least significant to most significant digit. For sorting each of these digit groups, a stable sorting algorithm is needed. Also, the elements in this group to be sorted are in the fixed range of 0 to 9. Both of these characteristics point towards the use of Counting Sort as the sorting algorithm of choice for sorting on each digit.

The time complexity of the algorithm is as follows:

Suppose that the n input numbers have maximum k digits. Then the Counting Sort procedure is called a total of k times. Counting Sort is a linear, or O(n) algorithm. So the entire Radix Sort procedure takes **O(k.n) time**.

Definition:

Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into. There is a one-to-one correspondence between the number of buckets and the number of values that can be represented by a digit. Then, the process repeats with the next neighboring digit until there are no more digits to process. In other words:

1. Take the least significant digit (or group of bits, both being examples of radices) of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys. (This is what makes the LSD radix sort a stable sort).
3. Repeat the grouping process with each more significant digit.

The sort in step 2 is usually done using bucket sort or counting sort, which are efficient in this case since there are usually only a small number of digits.

An example: Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives:

170, 90, 802, 2, 24, 45, 75, 66

Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.

Sorting by next digit (10s place) gives:

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

It is important to realize that each of the above steps requires just a single pass over the data, since each item can be placed in its correct bucket without having to be compared with other items.

Some LSD radix sort implementations allocate space for buckets by first counting the number of keys that belong in each bucket before moving keys into those buckets. The number of times that each digit occurs is stored in an array. Consider the previous list of keys viewed in a different way:

170, 045, 075,090, 002, 024, 802, 066

The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:

2 (bucket size for digits of 0: 170, 090)
2 (bucket size for digits of 2: 002, 802)
1 (bucket size for digits of 4: 024)
2 (bucket size for digits of 5: 045, 075)
1 (bucket size for digits of 6: 066)

A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:

2 (bucket size for digits of 0: 002, 802)
1 (bucket size for digits of 2: 024)
1 (bucket size for digits of 4: 045)
1 (bucket size for digits of 6: 066)
2 (bucket size for digits of 7: 170, 075)
1 (bucket size for digits of 9: 090)

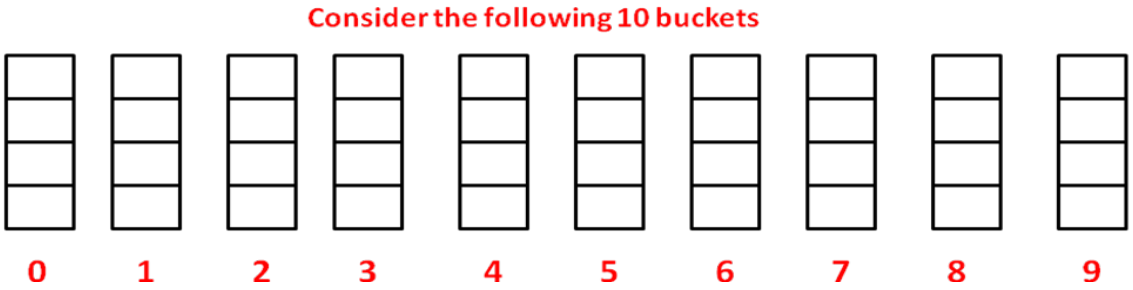
A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:

6 (bucket size for digits of 0: 002, 024, 045, 066, 075, 090)
1 (bucket size for digits of 1: 170)
1 (bucket size for digits of 8: 802)

At least one LSD radix sort implementation now counts the number of times that each digit occurs in each column for all columns in a single counting pass. Other LSD radix sort implementations allocate space for buckets dynamically as the space is needed.

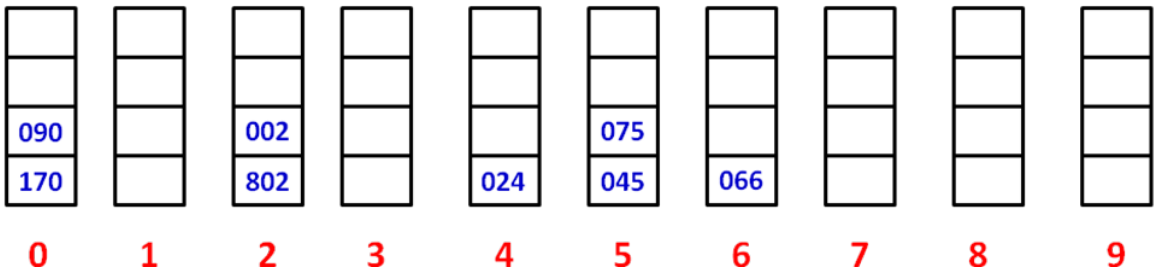
Example 1) Sort the list of numbers using radix sort technique (bucket sort) the Original, unsorted list is

170, 045, 075, 090, 802, 024, 002, 066



Sorting the following list by least significant digit (1's place) gives:
170, 045, 075, 090, 802, 024, 002, 066

```
Consider least significant digit of every number
for example 170 will be placed in bucket 0
            045 will be placed in bucket 5
            075 will be placed in bucket 5
            090 will be placed in bucket 0
            802 will be placed in bucket 2
            024 will be placed in bucket 4
            002 will be placed in bucket 2
            066 will be placed in bucket 6
```



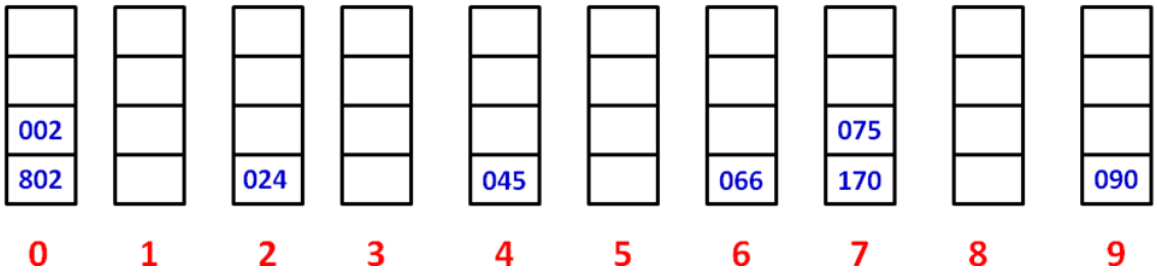
Collect numbers sequentially from bucket 0 to bucket 9

170, 090, 802, 002, 024, 045, 075, 066

Sorting the following list by next digit (10's place) gives:

170, 090, 802, 002, 024, 045, 075, 066

```
Consider digit at radix 1 i.e. second digit of every number
for example 170 will be placed in bucket 7
            090 will be placed in bucket 9
            802 will be placed in bucket 0
            002 will be placed in bucket 0
            024 will be placed in bucket 2
            045 will be placed in bucket 4
            075 will be placed in bucket 7
            066 will be placed in bucket 6
```

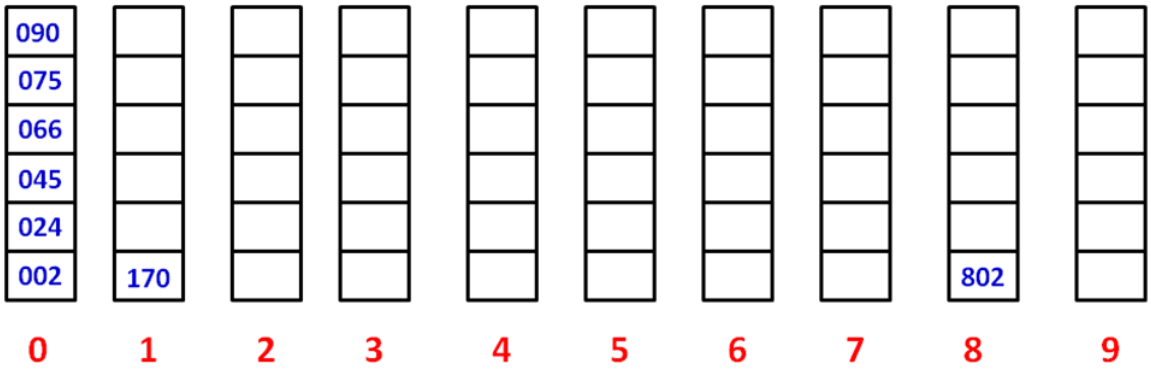


Collect numbers sequentially from bucket 0 to bucket 9

802, 002, 024, 045, 066, 170, 075, 090

now Sorting by most significant digit (100s place) gives:

802, 002, 024, 045, 066, 170, 075, 090



collect elements from first bucket to last bucket sequentially

002 024 045 066 075 090 170 802

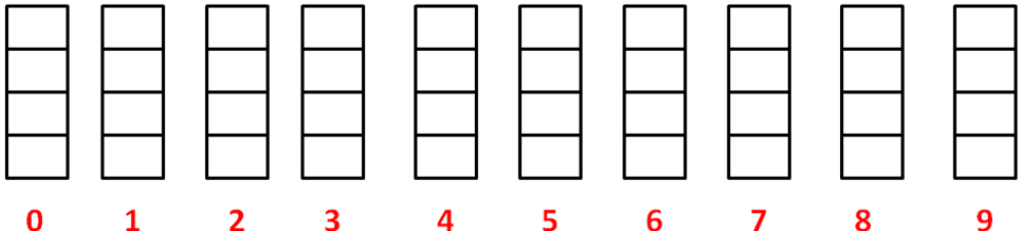
Final sorted list

002, 024, 045, 066, 075, 090, 170, 802

Example 2) sort the elements of the following list using Radix Sort / bucket sort technique.

unsorted list: 523, 153, 088, 554, 235

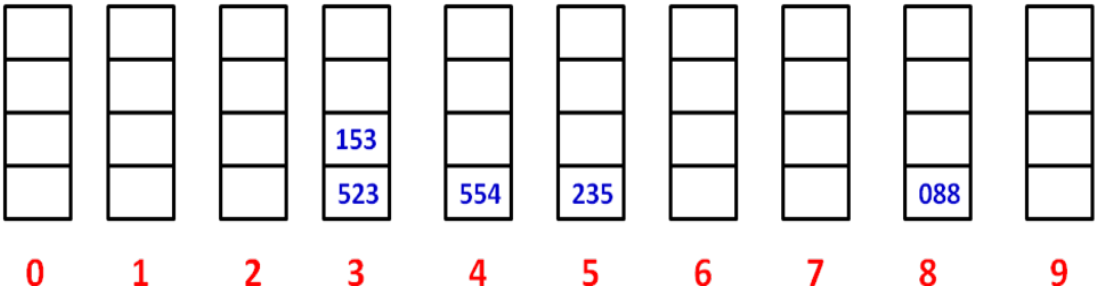
Consider the following 10 buckets



Sorting for Radix 0 (least significant digit)

523, 153, 088, 554, 235

Consider least significant digit of every number
for example 523 will be placed in bucket 3
153 will be placed in bucket 3
088 will be placed in bucket 8
554 will be placed in bucket 4
235 will be placed in bucket 5



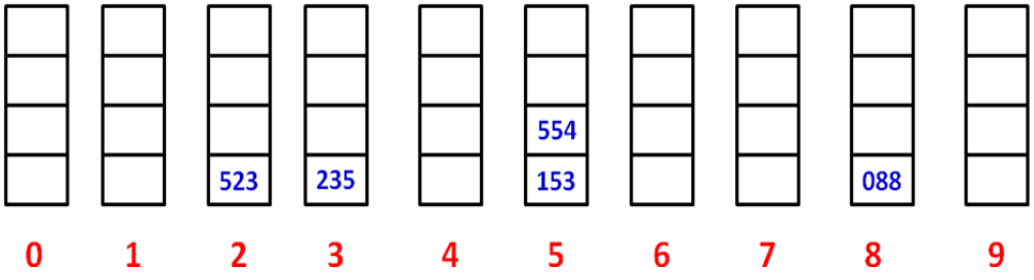
collect the numbers from buckets sequentially

523, 153, 554, 235, 088

Sorting for Radix 1 (2nd. significant digit)

523, 153, 554, 235, 088

Consider the second digit of every number
for example 523 will be placed in bucket 2
153 will be placed in bucket 5
554 will be placed in bucket 5
235 will be placed in bucket 3
088 will be placed in bucket 8



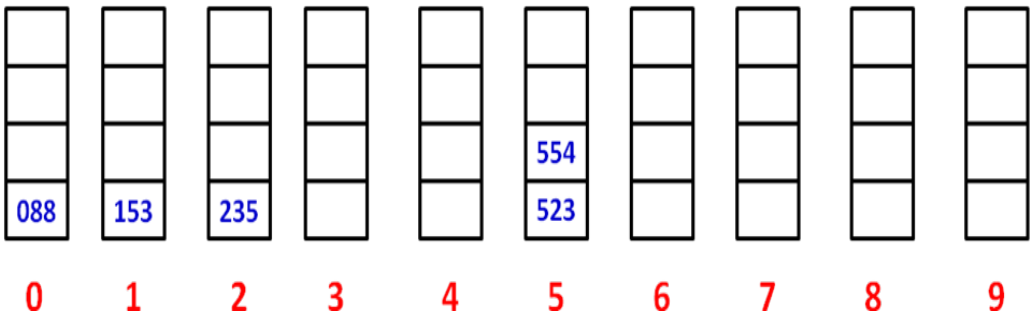
Collect the numbers from buckets sequentially

523, 235, 153, 554, 088

Sorting for Radix 2 (most. significant digit)

523, 235, 153, 554, 088

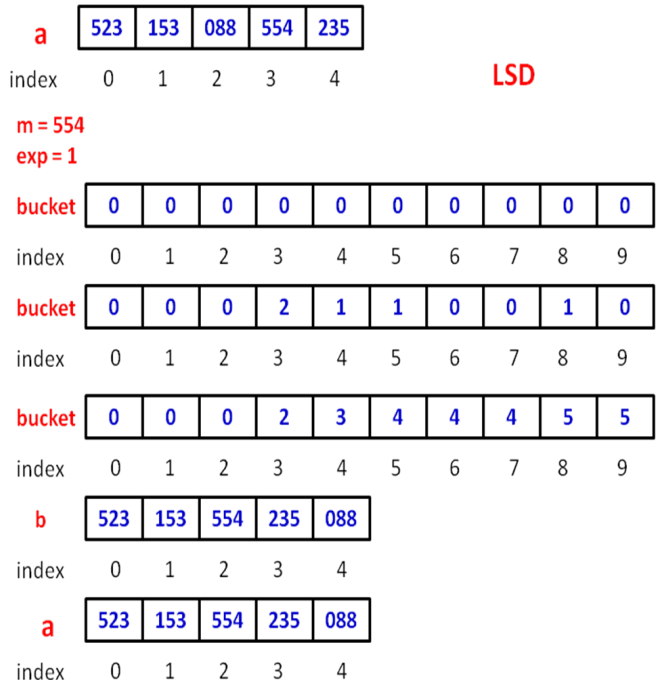
Consider the third digit of every number
for example 523 will be placed in bucket 5
235 will be placed in bucket 2
153 will be placed in bucket 1
554 will be placed in bucket 5
088 will be placed in bucket 0



Collect the numbers from buckets sequentially
088, 153, 235, 523, 554

Final sorted List
088, 153, 235, 523, 554

```
/*-------------------------------------------------------
    Program to sort a list using Radix Sort Technique
-------------------------------------------------------*/
#include <stdio.h>
#include <conio.h>

void radixsort(int a[],int n)
{
   int i,b[20],m=0,exp=1;
   int bucket[10];

   for(i=0;i<n;i++)
      if(a[i]>m)
         m=a[i];

   while(m/exp>0)
   {
      for(i=0;i<10;i++)
         bucket[i]=0;

      for(i=0;i<n;i++)
        bucket[a[i]/exp%10]++;

      for(i=1;i<10;i++)
        bucket[i]+=bucket[i-1];

      for(i=n-1;i>=0;i--)
        b[--bucket[a[i]/exp%10]]=a[i];

      for(i=0;i<n;i++)
         a[i]=b[i];

      exp*=10;
   }
}

void main()
{
   int a[20],i,n;

   clrscr();
   printf("enter n ");
   scanf("%d",&n);
   printf("Enter Array Elements \n");
   for (i=0;i<n;i++)
      scanf("%d",&a[i]);

   radixsort(a,n);

   printf("The sorted array\n");
   for(i=0;i<n;i++)
      printf("%d\n",a[i]);

   getch();
}
```



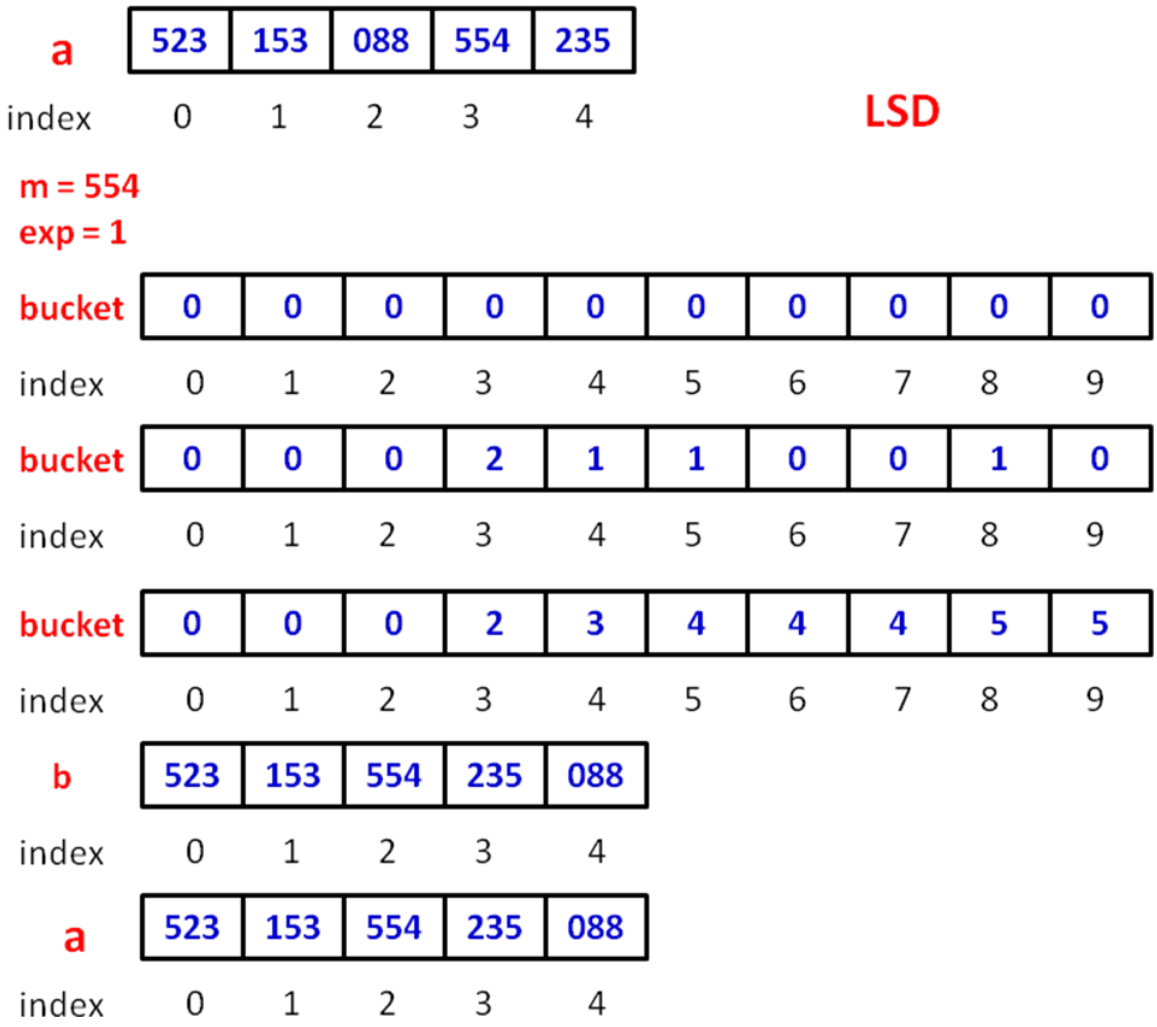
OUTPUT

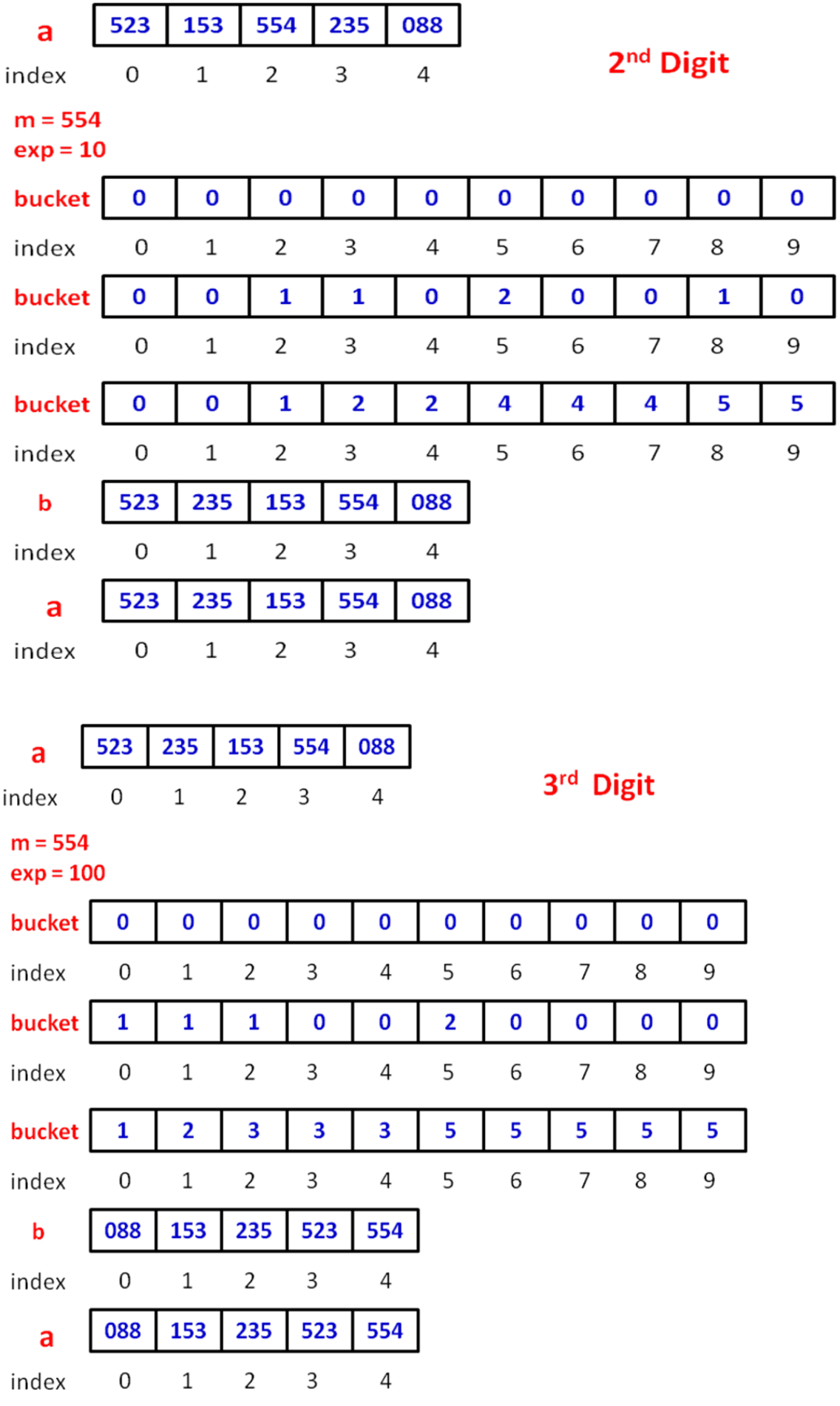
```
enter n 8
Enter array elements
170   045   075   090   802   024   002   066
The sorted array
002   024   045   066   075   090   170   802
```

OUTPUT

```
enter n 5
Enter array elements
523   153   088   554   235
The sorted array
088   153   235   523   554
```

Explanation of radix function



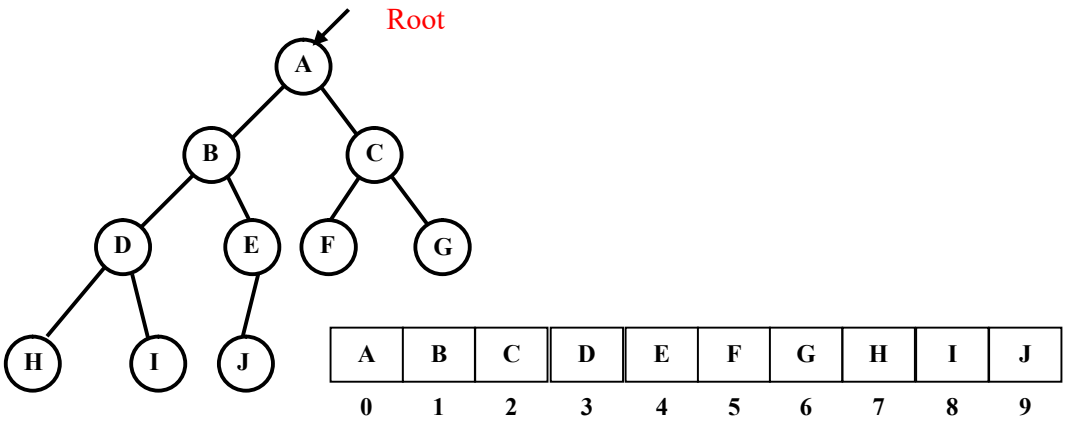

a
523 153 554 235 088
index 0 1 2 3 4
2nd Digit
m = 554
exp = 10
bucket 0 0 0 0 0 0 0 0 0 0
index 0 1 2 3 4 5 6 7 8 9
bucket 0 0 1 1 0 2 0 0 1 0
index 0 1 2 3 4 5 6 7 8 9
bucket 0 0 1 2 2 4 4 4 5 5
index 0 1 2 3 4 5 6 7 8 9
b 523 235 153 554 088
index 0 1 2 3 4
a 523 235 153 554 088
index 0 1 2 3 4
a 523 235 153 554 088
index 0 1 2 3 4
3rd Digit
m = 554
exp = 100
bucket 0 0 0 0 0 0 0 0 0 0
index 0 1 2 3 4 5 6 7 8 9
bucket 1 1 1 0 0 2 0 0 0 0
index 0 1 2 3 4 5 6 7 8 9
bucket 1 2 3 3 3 5 5 5 5 5
index 0 1 2 3 4 5 6 7 8 9
b 088 153 235 523 554
index 0 1 2 3 4
a 088 153 235 523 554
index 0 1 2 3 4

Binary Heap

A heap is a binary tree that is completely filled with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a complete binary tree.

It is easy to show that a complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes.

An important observation is that because a complete binary tree is so regular, if can be represented in an array and no pointers are necessary.



For any element in array position i, the left child is in position 2i + 1, the right child is in the cell after the left child **2i+2** and parent is in position **i/2 –1**.

Heap Sort :

Heap Sort is simply an implementation of the general selection Sort using the input array x as a heap representing a descending priority Queue. The preprocessing phase creates a heap of size n, using the sift-up operation, and the selection phase redistributes the elements of the heap in order as it deletes elements from the priority Queue using the sift-down operation.

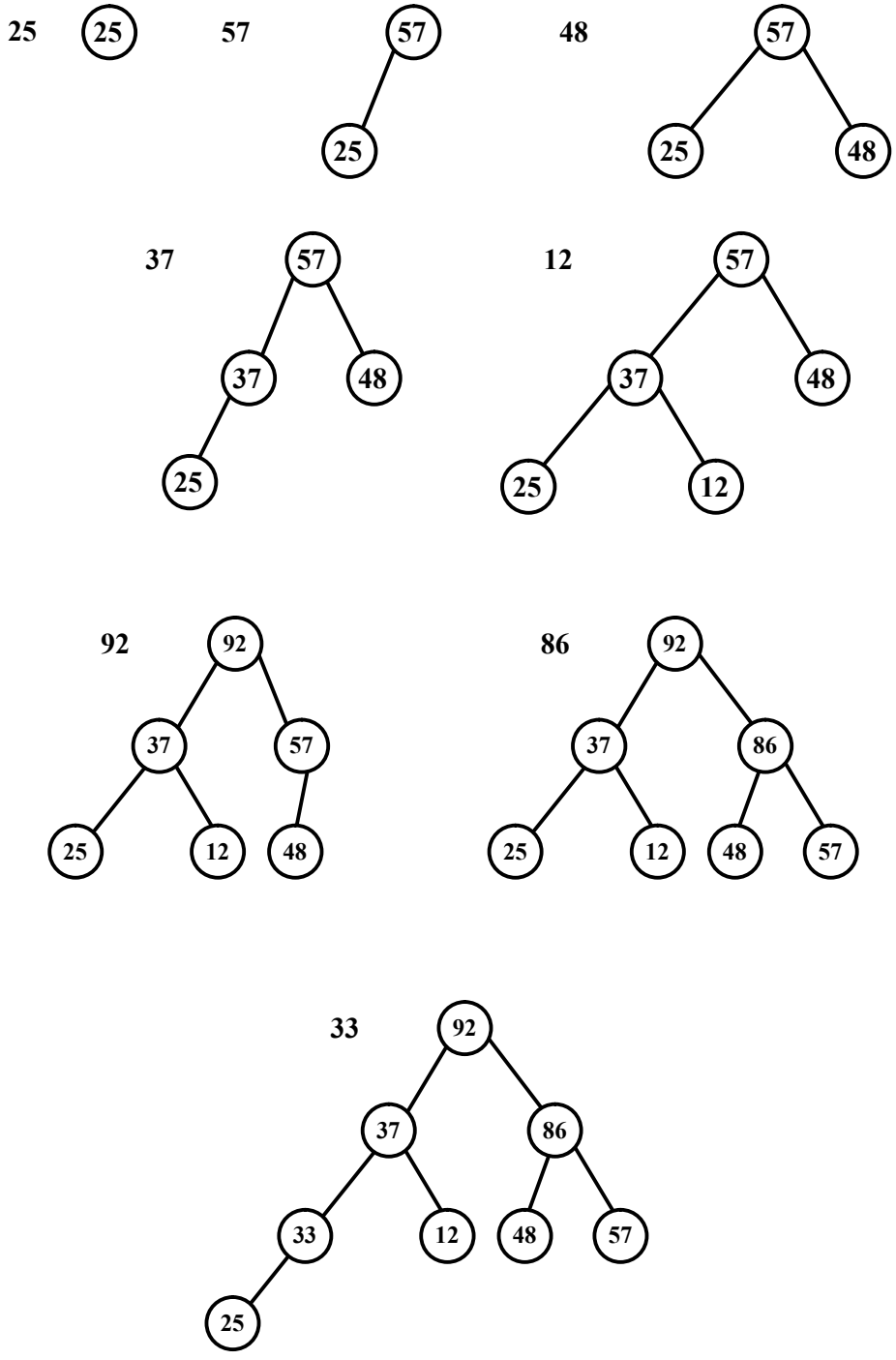
Phase1	---	Creating a heap.
Phase2	---	Illustrates the adjustment of the heap as x[0] is

Repeatedly selected and placed into its proper position in the array and the heap is readjusted, Until all the heap elements are processed. Note that after an element has been deleted from the heap, it remains in the array, it is merely ignored in subsequent processing.

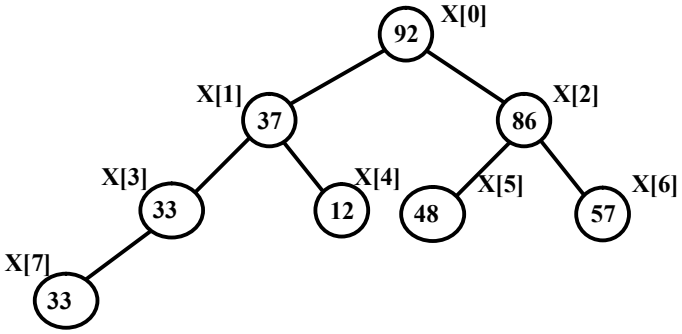
Running time of heap sort O(n log n)

Heap Sort

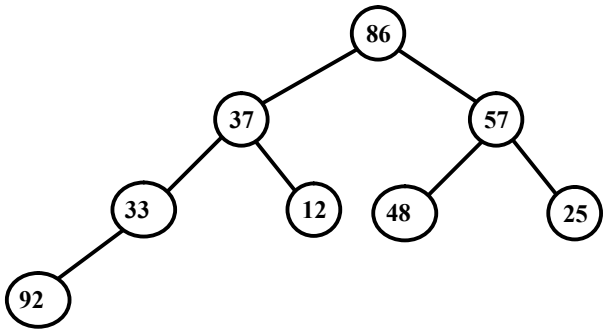
Constructing a heap with the following list of elements and arrange them in ascending order using Heap sort. **25 57 48 37 12 92 86 33**



Creating a heap of the size 8

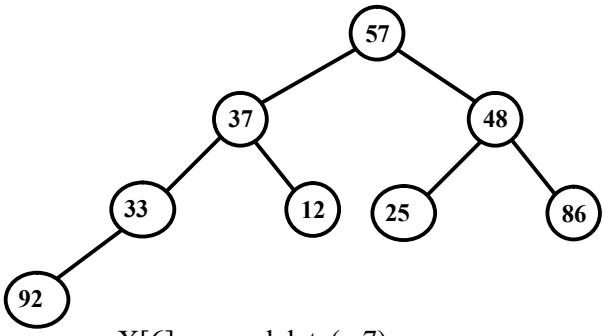


92	37	86	33	12	48	57	25



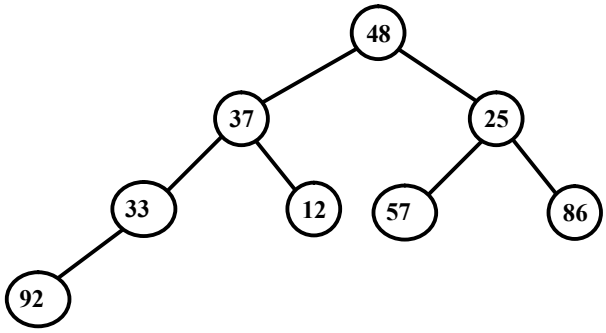
X[7] = max(x,8)

86 37 57 33 12 48 25 92

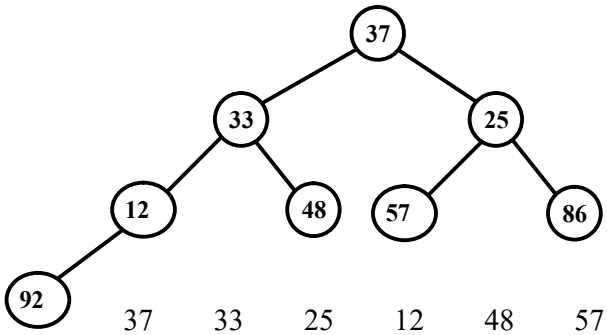


X[6] = maxdelete(x,7)

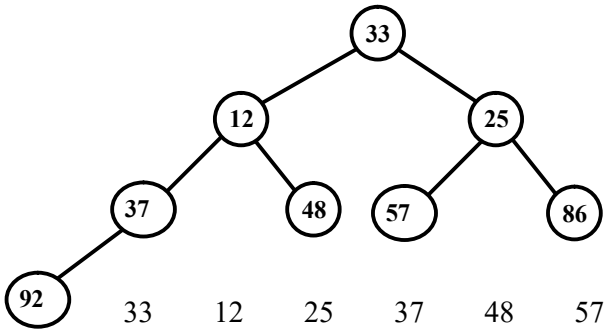
57 37 48 33 12 25 86 92



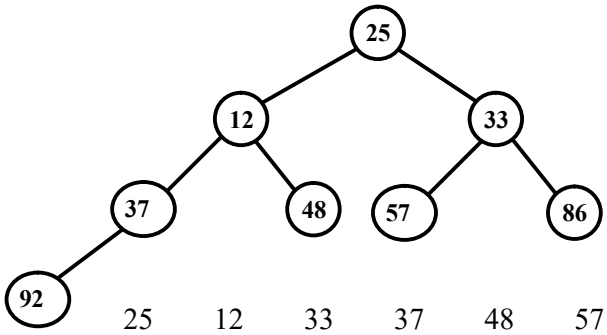
48 37 25 33 12 57 86 92



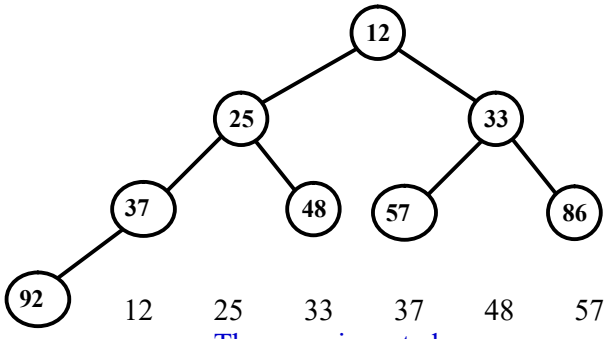
37 33 25 12 48 57 86 92



33 12 25 37 48 57 86 92



25 12 33 37 48 57 86 92



12 25 33 37 48 57 86 92

The array is sorted.

```
/*****************************************************************
                        Heap Sort Technique
 *****************************************************************/
#include<stdio.h>
#include<conio.h>
display(a,n)
int a[],n;
{
        int i;
        for(i=0;i<n;i++)
        printf("%d ",a[i]);
        printf("\n");
}

createheap(x,n)             /*-------------------- Function to create heap */
int x[],n;
{
  int i,ele,s,f;
  for (i=1;i<n;i++)
  {
    ele = x[i];
    s = i;
    f = (s-1) / 2;
    while (s>0 && x[f]<ele)
     {
        x[s] = x[f];
        s = f;
        f = (s-1) / 2;
     }
     x[s] = ele;
  }

}

  swap(x,i,j)
  int x[],i,j;
  {
   int temp;
   temp=x[i];
   x[i]=x[j];
   x[j]=temp;
  }

  heapsort(x,n)          /* Repeatedly remove x[0] and insert  it in proper */
  int x[],n;             /*   position and readjust the remaining heap */
  {
   int i;
   for (i=n-1;i>0;i--)
    {
        display(x,n);
        swap(x,0,i);
        createheap(x,i);
    }
  }
```

```
void main()
{
   static int a[20] = { 25, 57, 48, 37, 12, 92, 86, 33 };
   int n,i;
   clrscr();
   createheap(a,8);
   heapsort(a,8);                    /*------------------------- function call */
   printf("the sorted array is:");
   display(a,8);
   getch();
}
```

```
/*****************************************************
                Heap Sort another version
*****************************************************/
#include <stdio.h>
#include <conio.h>
void makeheap(a,n)
int a[50],n;
{
  int i,j,val,father;
  for (i=1;i<n;i++)
    {
       val=a[i];
       j=i;
       father=(j-1)/2;
       while (j>0 && a[father]<val)
       {
      a[j]=a[father];
      j=father;
      father=(j-1)/2;
       }
       a[j]=val;
    }
}



void display(a,n)
int a[50],n;
{
   int i;
   for(i=0;i<n;i++)
      printf("%d\n",a[i]);
}
```

```
void heapsort(a,n)
int a[50],n;
{
   int i,k,temp,j;
   for(i=n-1;i>0;i--)
   {
      temp=a[i];
      a[i]=a[0];
      k=0;
      if ( i==1)
      j=-1;
      else
      j=1;
      if (i>2 && a[2]>a[1])
      j=2;
      while(j>0 && temp<a[j])
      {
      a[k]=a[j];
      k=j;
      j=2*k+1;
      if (j+1<= i-1 && a[j]<a[j+1])
         j++;
      if (j>i-1)
         j=-1;
      }
      a[k]=temp;
   }
}

void main()
{
   int a[50], i,n;
   clrscr();
   printf("enter n ");
   scanf("%d",&n);
   for(i=0;i<n;i++)
   {
     printf("enter array element ");
     scanf("%d",&a[i]);
   }
   printf("The Un sorted array is\n");
   display(a,n);

   makeheap(a,n);
   printf("The heapfied array is\n");
   display(a,n);

   heapsort(a,n);
   printf("The sorted array is\n");
   display(a,n);

   getch();
}
```