Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.

Types

1.Insertion sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

Algorithm

```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

Ex:

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Programe:

```
classInsertionSort{

 voidsort(intarr[])

    {
        intn = arr.length;
        for(inti=1; i<n; ++i)
        {
            intkey = arr[i];
            intj = i-1;
  while(j>=0&&arr[j] > key)
            {
```

```
                arr[j+1] = arr[j];
                j = j-1;
            }
            arr[j+1] = key;
        }
    }
 staticvoidprintArray(intarr[])
    {
        intn = arr.length;
        for(inti=0; i<n; ++i)
            System.out.print(arr[i] + " ");
  System.out.println();
    }
     publicstaticvoidmain(String args[])
    {
        intarr[] = {12, 11, 13, 5, 6};
 InsertionSortob = newInsertionSort();
        ob.sort(arr);
         printArray(arr);
    }
}
```

Out put:

```
5 6 11 12 13
```

2.Merge sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

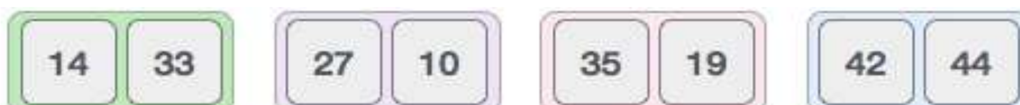
Algorithm

```
Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.
```

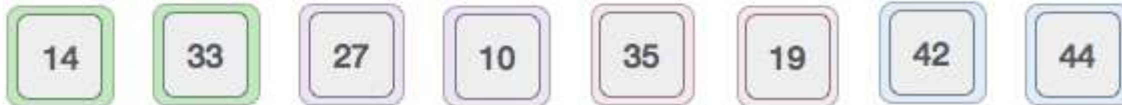
Ex:To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

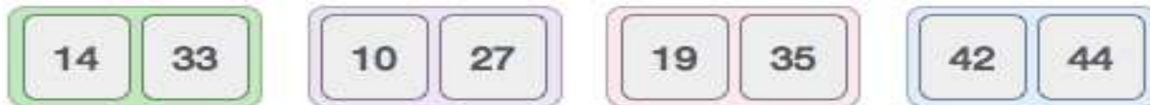


We further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this –

Now we should learn some programming aspects of merge sorting.

Program:

```
classMergeSort

{
    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    voidmerge(intarr[], intl, intm, intr)
    {
        // Find sizes of two subarrays to be merged
        intn1 = m - l + 1;
        intn2 = r - m;

        /* Create temp arrays */
        intL[] = newint[n1];
        intR[] = newint[n2];

        /*Copy data to temp arrays*/
        for(inti=0; i<n1; ++i)
```

```
        L[i] = arr[l + i];
    for(intj=0; j<n2; ++j)
        R[j] = arr[m + 1+ j];


    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    inti = 0, j = 0;

    // Initial index of merged subarry array
    intk = l;
    while(i< n1 && j < n2)
    {
        if(L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy remaining elements of L[] if any */
    while(i< n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy remaining elements of R[] if any */
    while(j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
voidsort(intarr[], intl, intr)
{
    if(l < r)
    {
        // Find the middle point
        intm = (l+r)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr , m+1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
```

```
    }

    /* A utility function to print array of size n */
    staticvoidprintArray(intarr[])
    {
        intn = arr.length;
        for(inti=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver method
    publicstaticvoidmain(String args[])
    {
        intarr[] = {12, 11, 13, 5, 6, 7};

        System.out.println("Given Array");
        printArray(arr);

        MergeSortob = newMergeSort();
        ob.sort(arr, 0, arr.length-1);

        System.out.println("\nSorted array");
        printArray(arr);
    }}
```

Out put:

```
Given array is
12 11 13 5 6 7


Sorted array is
5 6 7 11 12 13
```

3.Quick sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

Algorithm

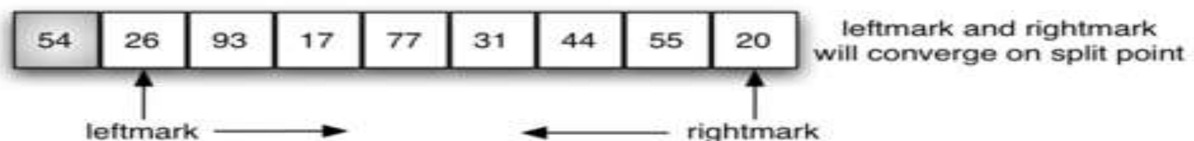
```
Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
```

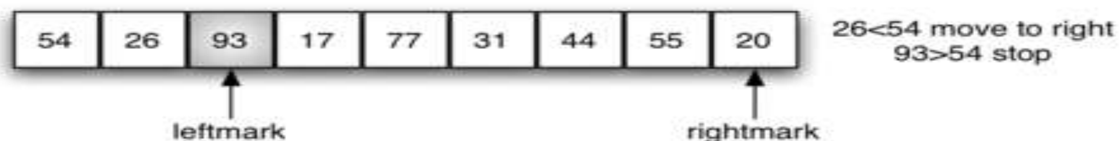
```
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot
```

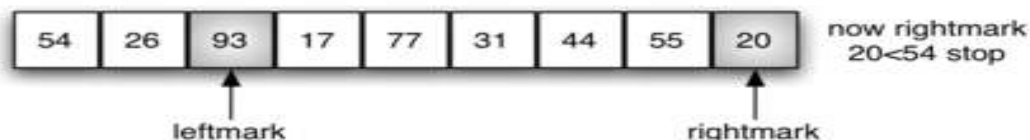
Ex:

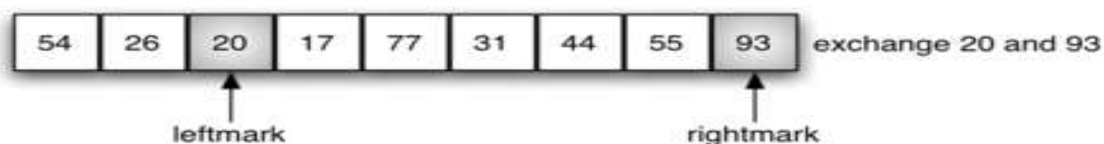
shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31.
The **partition**process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

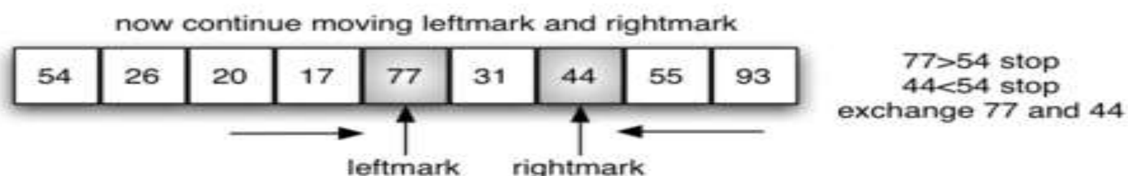
Partitioning begins by locating two position markers—let’s call them `leftmark` and `rightmark`—at the beginning and end of the remaining items in the list (positions 1 and 8 in Figure). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure shows this process as we locate the position of 54.

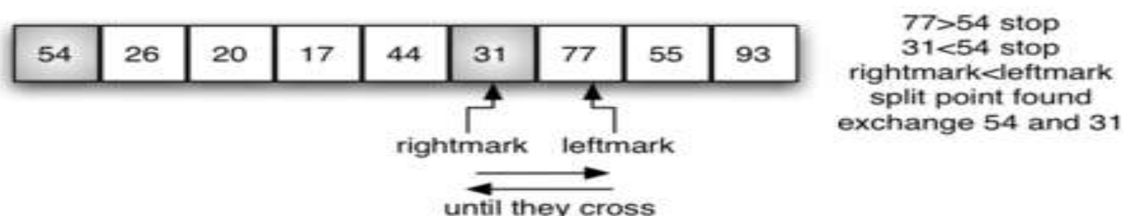






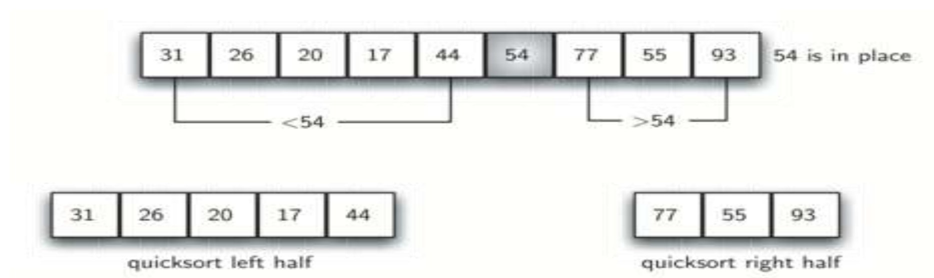






We begin by incrementing `leftmark` until we locate a value that is greater than the pivot value. We then decrement `rightmark` until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where `rightmark` becomes less than `leftmark`, we stop. The position of `rightmark` is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place (Figure 14). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



Program

```
classQuickSort
{
    intpartition(intarr[], intlow, inthigh)
    {
        intpivot = arr[high];
        inti = (low-1);
        for(intj=low; j<high; j++)
        {

            if(arr[j] <= pivot)
            {
                i++;

                // swap arr[i] and arr[j]
                inttemp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

             inttemp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        returni+1;
    }

    voidsort(intarr[], intlow, inthigh)
    {
        if(low < high)
```

```
        {
            intpi = partition(arr, low, high);

            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }

    staticvoidprintArray(intarr[])
    {
        intn = arr.length;
        for(inti=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    publicstaticvoidmain(String args[])
    {
        intarr[] = {10, 7, 8, 9, 1, 5};
        intn = arr.length;

        QuickSortob = newQuickSort();
        ob.sort(arr, 0, n-1);

        System.out.println("sorted array");
        printArray(arr);
    }
}
```

Out put:

```
Sorted array:
1 5 7 8 9 10
```

4.Heap sort:

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

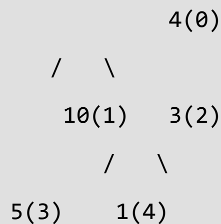
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

How to build the heap?

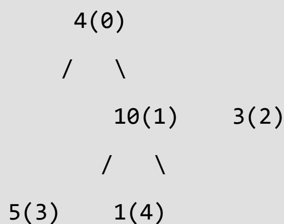
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

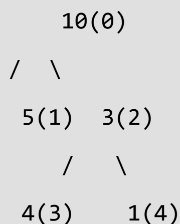
```
Input data: 4, 10, 3, 5, 1
```



```
                 4(0)
        /   \
         10(1)   3(2)
            /   \
     5(3)    1(4)
The numbers in bracket represent the indices in the array
representation of data.
Applying heapify procedure to index 1:
```



```
         4(0)
        /   \
            10(1)    3(2)
           /   \
    5(3)    1(4)
Applying heapify procedure to index 0:
```



```
            10(0)
        /  \
         5(1)  3(2)
            /   \
         4(3)    1(4)
The heapify procedure calls itself recursively to build heapin top down manner.
```

Program

```
// Java program for implementation of Heap Sort
publicclassHeapSort
{
    publicvoidsort(intarr[])
    {
        intn = arr.length;

        // Build heap (rearrange array)
        for(inti = n / 2- 1; i>= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for(inti=n-1; i>=0; i--)
        {
            // Move current root to end
            inttemp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    voidheapify(intarr[], intn, inti)
    {
        intlargest = i;  // Initialize largest as root
        intl = 2*i + 1;  // left = 2*i + 1
        intr = 2*i + 2;  // right = 2*i + 2

        // If left child is larger than root
        if(l < n &&arr[l] >arr[largest])
            largest = l;

        // If right child is larger than largest so far
        if(r < n &&arr[r] >arr[largest])
            largest = r;

        // If largest is not root
        if(largest != i)
        {
            intswap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    /* A utility function to print array of size n */
    staticvoidprintArray(intarr[])
    {
        intn = arr.length;
```

```
for(inti=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    // Driver program
    publicstaticvoidmain(String args[])
    {
        intarr[] = {12, 11, 13, 5, 6, 7};
        intn = arr.length;

        HeapSortob = newHeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

Run on IDE

Output:

```
Sorted array is

5 6 7 11 12 13
```

- Sorting with tapes:

1. Given n programs to be stored on tape, the lengths of these programs are i1, i2….in respectively. Suppose the programs are stored in the order of i1, i2…in. 2 We have a tape of length L i.e. the storage capacity of the tape is L. We are also given n programs where length of each program is i is Li.
2. Let Tj be the time to retrieve program ij.
3. It is now required to store these programs on the tape in such a way so that the mean retrieval time is minimum. MRT is the average tome required to retrieve any program stored on this tape.
4. Assume that the tape is initially positioned at the beginning.
5. Tj is proportional to the sum of all lengths of programs stored in front of the program ij.
6. The goal is to minimize MRT (Mean Retrieval Time),(1/n) ∑ni=0Tj∑i=0nTj

I.e., want to minimize ∑nj=1∑jk=1Ti∑j=1n∑k=1jTi

Problem:

1. Here, n=3 and (L1, L2, L3) = (5, 10, 3). We can store these 3 programs on the tape in any order but we want that order which will minimize the MRT.
2. Suppose we store the programs in order (L1, L2, L3).
3. Then MRT is given as (5+(5+10)+(5+10+3))/3=38/3
4. To retrieve L1 we need 5 units of time. Because a tape is a sequential device we will have to first pass through entire L1 even if we want to retrieve L2.
5. Hence, retrieval time (RT) is 5 for program 1 and (5+10) for program 2.
6. Similarly, if program 3 is also considered then the total RT becomes 5+ (5+10) + (5+10+3) where (5+10+3) is the RT for program 3.
7. Since we want to find the mean retrieval time we add all the RT and then divide the sum by n.
8. The aim over here is to find the minimum MRT. To do this we consider all the possible orderings of these 3 programs. Since there 3 programs we can have at the most 6(3!) combinations.
9. Consider the below table:

Ordering	MRT(MEAN RETREIVALTIME)
L1,L2,L3	5+(5+10)+(5+10+3)/3=38/3
L1,L3,L2	5+(5+3)+(5+10+3)/3=31/3
L2,L1,L3	10+(5+10)+(5+10+3)/3=43/3
L2,L3,L1	10+(3+10)+(5+10+3)/3=41/3
L3,L1,L2	3+(5+3)+(5+10+3)/3=29/3
L3,L2,L1	3+(3+10)+(5+10+3)/3=34/3

1. It should be seen that the minimum MRT of (29/3) is obtained in case of (L1, L2, L3). Hence the optimal solution is achieved if the programs are stored in increasing order of their lengths.
2. Hence, a greedy approach to solving the problem is continuously select programs in increasing order of their lengths.
3. If L is an array having program length in ascending order then the following would be an algorithm to this problem:

```
4. Sum=0;
5. For (i=1; i<=n; i ++)
6.    For (j=1; j<=I; j++)
7.       Sum = sum+ L[j]
```

```
MRT = sum/n
```

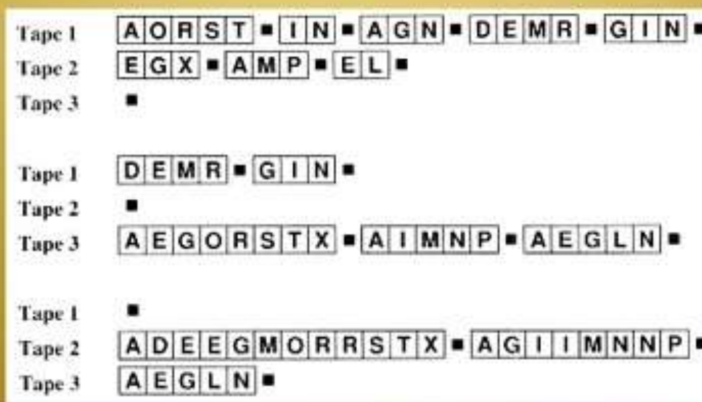
```
8. The time complexity of this algorithm including the timetodo sorting is
9. O(n2)
```

- Polyphase merge

A **polyphase merge sort** is an algorithm which decreases the number of *runs* at every iteration of the main loop by merging runs into larger runs. It is used for external sorting.

In this type of sort, the tapes being merged, and the tape to which the merged sub files are written, vary continuously throughout the sort. In this technique, the concept of a pass through records is not as clear-cut as in the straight or the natural merge.

EXAMPLE


Polyphase merging
For example, suppose that we have just three tapes, and we start out with the initial configuration of sorted blocks on the tapes shown at the top of figure:
Tape 1 A O R S T ■ I N ■ A G N ■ D E M R ■ G I N ■
Tape 2 E G X ■ A M P ■ E L ■
Tape 3 ■
Tape 1 D E M R ■ G I N ■
Tape 2 ■
Tape 3 A E G O R S T X ■ A I M N P ■ A E G L N ■
Tape 1 ■
Tape 2 A D E E G M O R R S T X ■ A G I I M N N P ■
Tape 3 A E G L N ■

4.2.1.2 Analysis

In above program the **basic operation** is comparision -

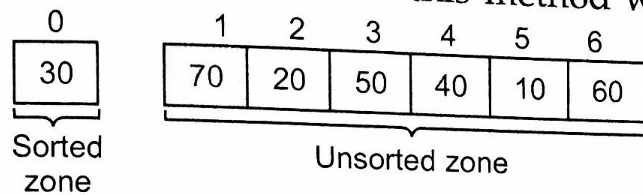
```
if (A[j] > A[j+1])
```

which is executed within nested for loops. Hence the time complexity of bubble sort is $\theta(n^2)$.

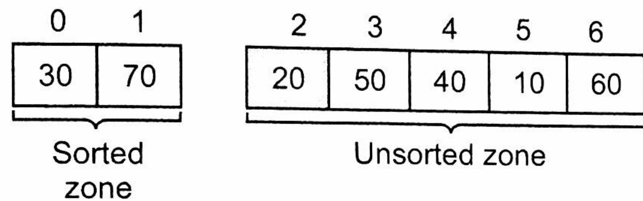
[Note : In all the sorting program I have given **getChar( )** function, so that sorting on character type elements can be done with appropriate changes.

4.2.2 Insertion Sort

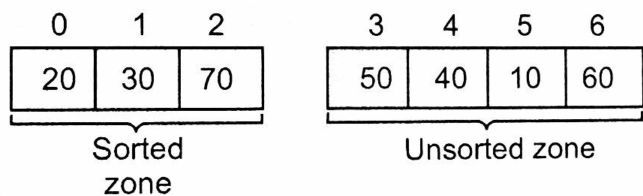
In this method the elements are inserted at their appropriate place. Hence is the name **insertion sort**. Let us understand this method with the help of some example -

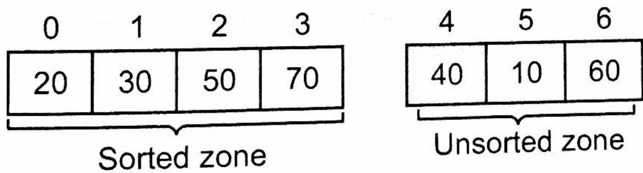


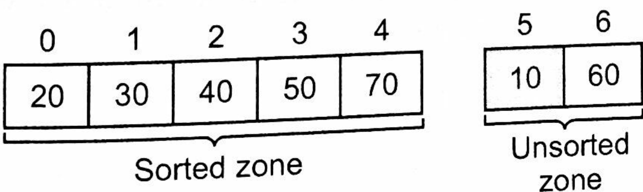
Compare 70 with 30 and insert it at its position

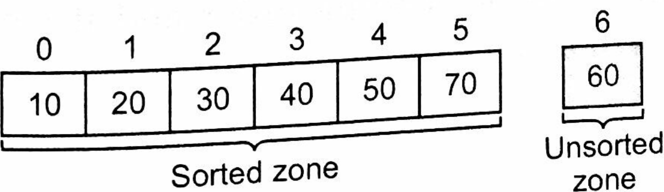


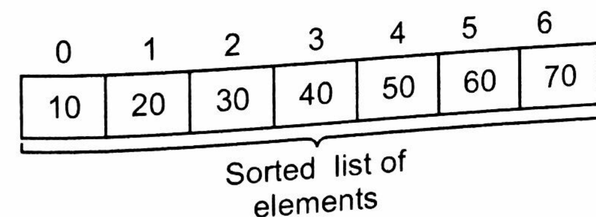
Compare 20 with the elements in sorted zone and insert it in that zone at appropriate position











For Example

Consider a list of elements as,

0	1	2	3	4	5	6
30	70	20	50	40	10	60

The process starts with first element

Algorithm

Although it is very natural to implement insertion using recursive(top down) algorithm but it is very efficient to implement it using bottom up(iterative) approach.

```
Algorithm Insert_sort(A[0...n-1])
//Problem Description: This algorithm is for sorting the
//elements using insertion sort
//Input: An array of n elements
//Output: Sorted array A[0...n-1] in ascending order
 for i ← 1 to n-1 do
 {
        temp ← A[i]//mark A[i]th element
        j ← i-1//set j at previous element of A[i]
        while(j>=0)AND(A[j]>temp)do
        {
    //comparing all the previous elements of A[i] with
    //A[i].If any greater element is found then insert
    //it at proper position
        A[j+1] ← A[j]
        j ← j-1
        }
        A[j+1] ← temp //copy A[i] element at A[j+1]
}
```

Analysis

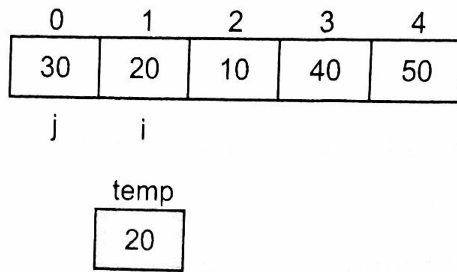
When an array of elements is almost sorted then it is **best case** complexity. The best case time complexity of insertion sort is **O(n).**

If an array is randomly distributed then it results in **average case** time complexity which is $O(n^2)$.

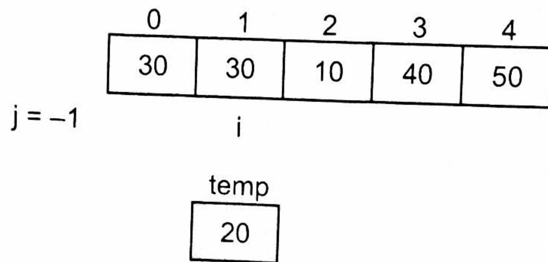
If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $O(n^2)$.

Logic Explanation

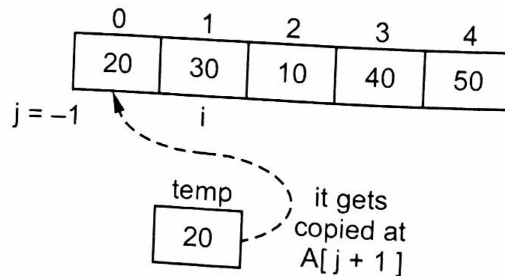
For understanding the logic of above Java program consider a list of unsorted elements as,



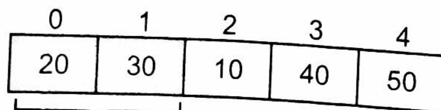
Then the control moves to **while loop**. As j >= 0 and A[j] > temp is **True**, the while loop will be executed.

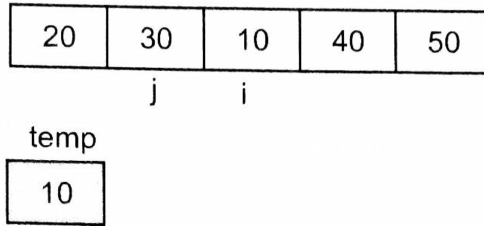


Now since j >= 0 is false, control comes out of while loop.

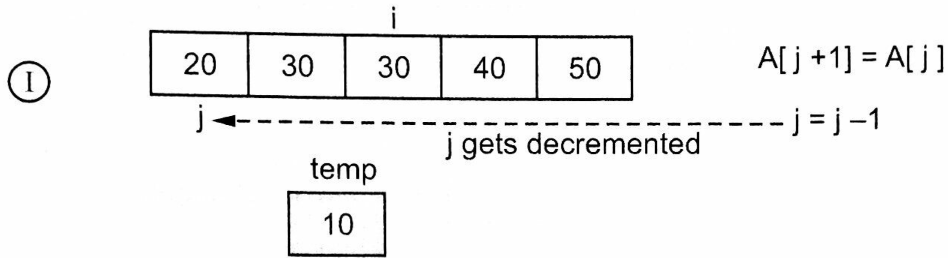


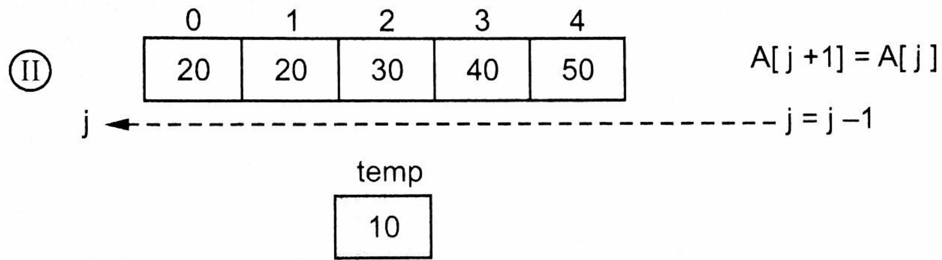
Then list becomes,

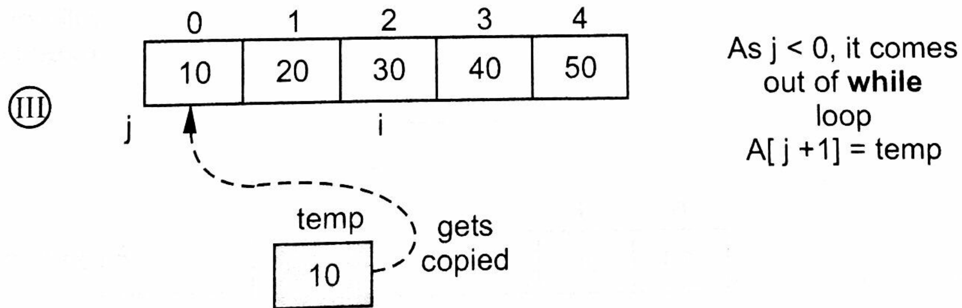




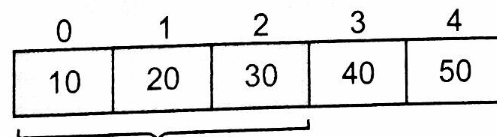
It moves to **while loop**
As j > = 0 and
(A [j] > temp) is **true**,
the while loop gets
executed.







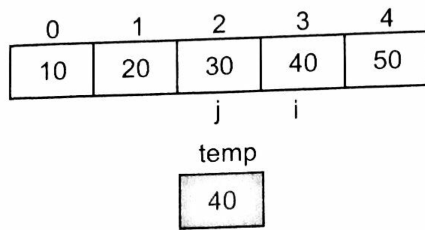
Thus,



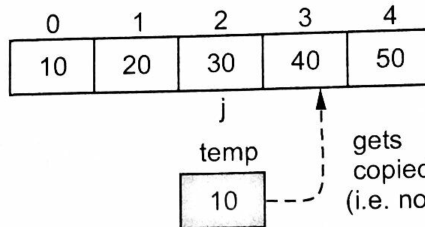
This much
list gets
sorted

Again **for loop** gets
executed.
set i = 3,
temp = A [i]
j = i –1

Then,



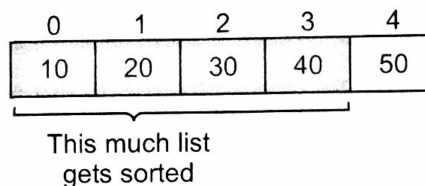
It moves to **while loop**
As A [j] > temp
is false , while loop
will not get executed.



`A[j+1]=temp`

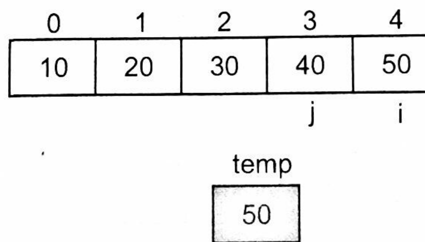
gets
copied
(i.e. no change!)

Then,

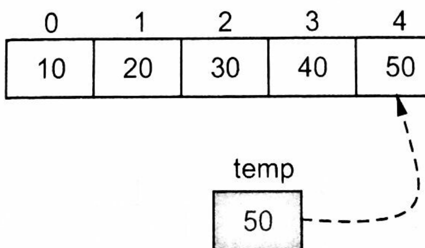


Again for loop gets
executed
set i = 4
temp = A [i]
j = i –1

Then,

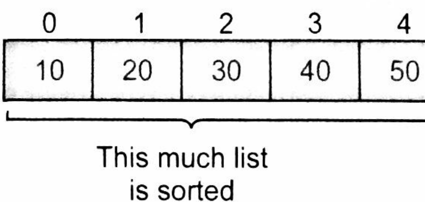


It moves to **while loop**
As A[j] > temp is
false, while loop
will not get
executed.



`A[j+1]=temp`

gets
copied



Thus we have scanned the entire list and inserted the elements at corresponding locations. Thus we get the sorted list by insertion sort.

12
14
18

4.2.6 Merge Sort

The merge sort is a sorting algorithm that uses the divide and conquer strat this method division is dynamically carried out.

Merge sort on an input array with n elements consists of three steps:

Divide : partition array into two sub lists s1 and s2 with n/2 elements each.

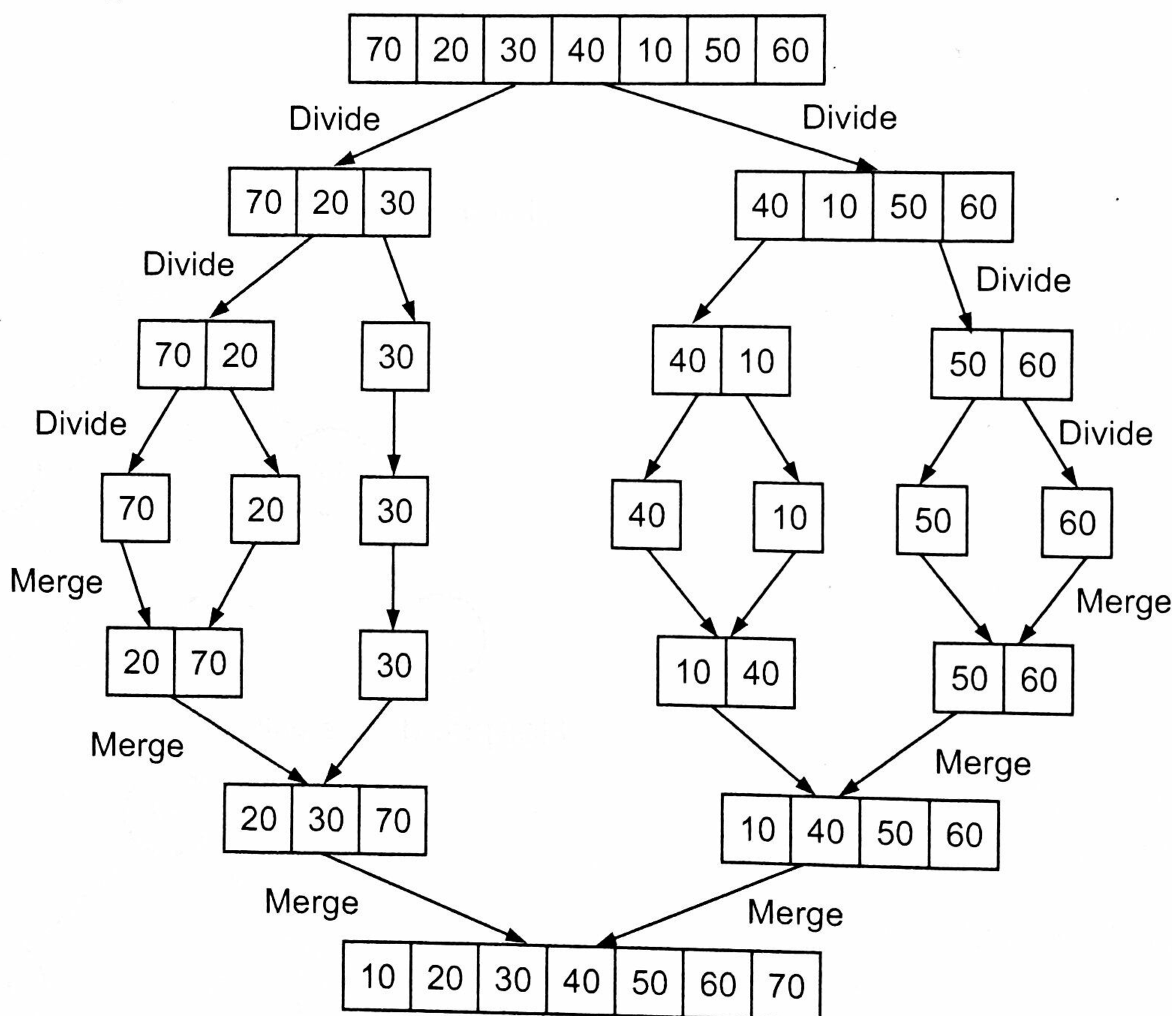
Conquer : Then sort sub list s1 and sub list s2.

Combine : merge s1 and s2 into a unique sorted group.

Example 1 : Consider the elements as

70, 20, 30, 40, 10, 50, 60

Now we will split this list into two sublists.

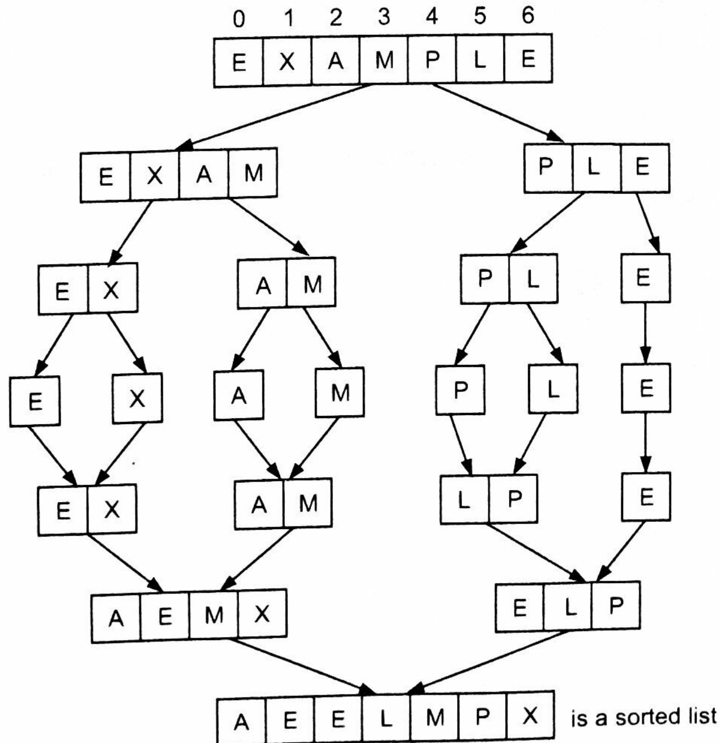


Example 2 : Sort the following elements using merge sort.

10, 5, 7, 6, 1, 4, 8, 3, 2, 9.

Now

Example 3 : Apply merge sort to sort the list E, X, A, M, P, L, E in alphabetical order.



```
Algorithm MergeSort(int A[0…n-1],low,high)
//Problem Description: This algorithm is for sorting the
//elements using merge sort
//Input: Array A of unsorted elements, low as beginning
//pointer of array A and high as end pointer of array A
//Output: Sorted array A[0…n-1]
    if(low < high)then
    {
        mid ← low+high)/2      //split the list at mid
        MergeSort(A,low,mid)         //first sublist
        MergeSort(A,mid+1,high)  //second sublist
        Combine(A,low,mid,high)  //merging of two sublists
     }
Algorithm Combine(A[0…n-1],low, mid, high)
{
    k  ← low;    //k as index for array temp
    i ← low;     //i as index for left sublist of array A
    j ← mid+1    //j as index for right sublist of array A
    while(i <= mid and j <= high)do
    {
    if(A[i]<=A[j])then
     //if smaller element is present in left sublist
        {
            //copy that smaller element to temp array
            temp[k] ← A[i]
            i ← i+1
            k ← k+1
        }
        else //smaller element is present in right sublist
         {
            //copy that smaller element to temp array
                temp[k] ← A[j]
                j ← j+1
                k ← k+1
            }
        }
        //copy remaining elements of left sublist to temp
    while(i<=mid)do
        {
                temp[k] ← A[i]
                i ← i+1
                k ← k+1
        }
```

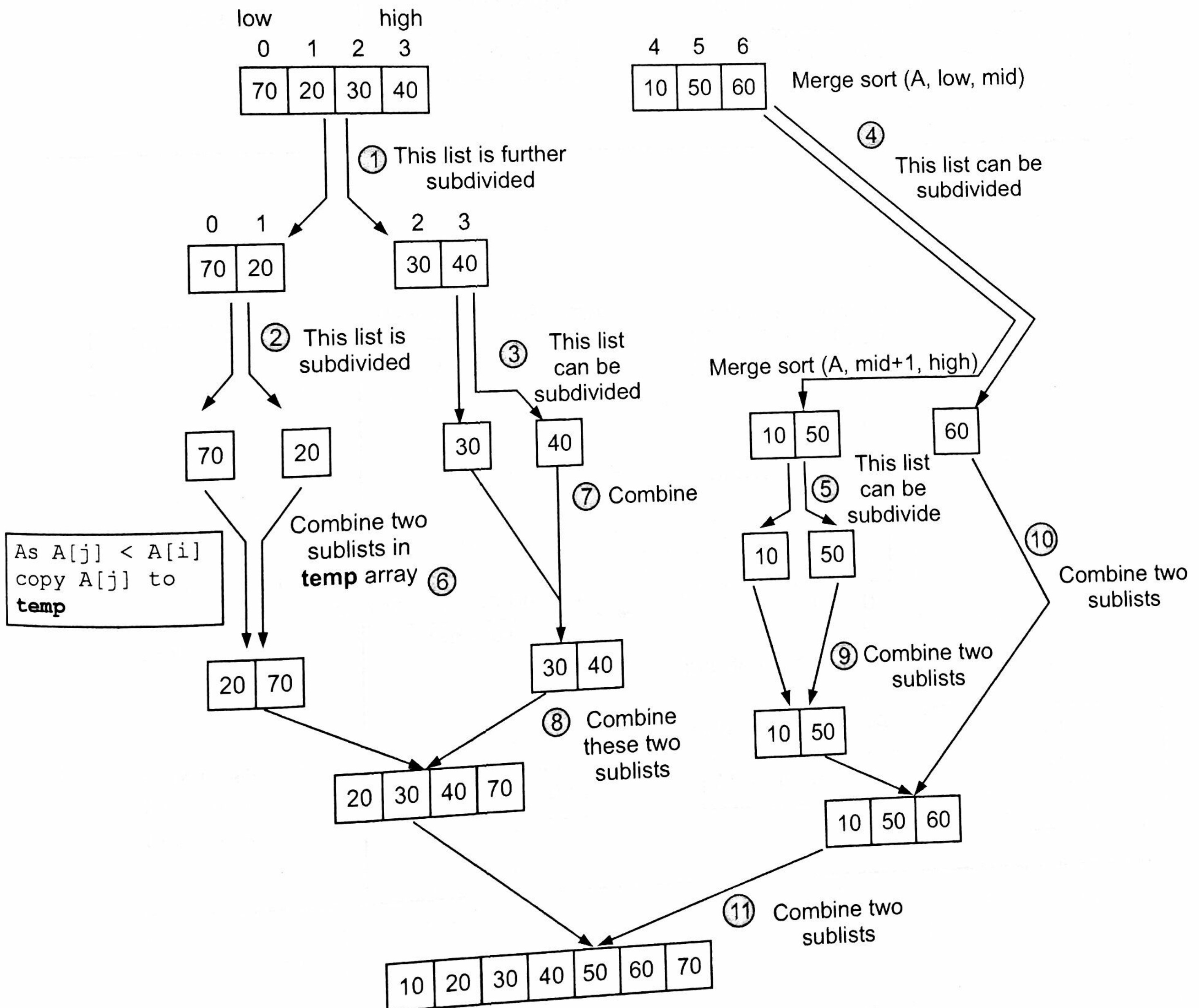
```
//copy remaining elements of right sublist to temp
while(j<=high) do
{
     temp[k] ← A[j]
     j ← j+1
     k ← k+1
}
```

Logic Explanation

To understand above algorithm consider a list of elements as

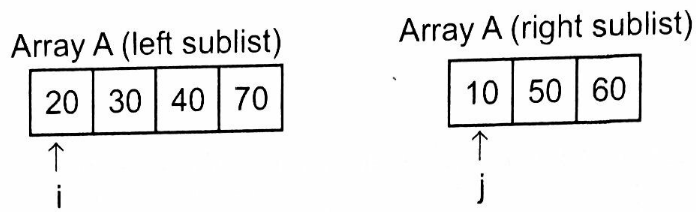
70	20	30	40	10	50	60
0	1	2	3	4	5	6
↑			↑			↑
low			mid			high

Then we will first make two sublists as



Let us see the **combine** operation more closely with the help of some example.

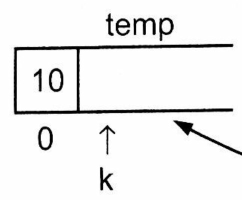
Consider that at some instance we have got two sublits 20, 30, 40, 70 and 10, 50, 60, then



Applicable part of Algorithm

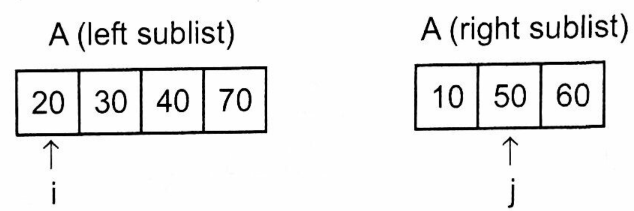
```
if (A[i]<= A[j])
{
  temp[k]← A[i]
   i←i+1
   k←k+1
}
else
{
  temp[k]← A[j]
   j←j+1
   k←k+1
}
```

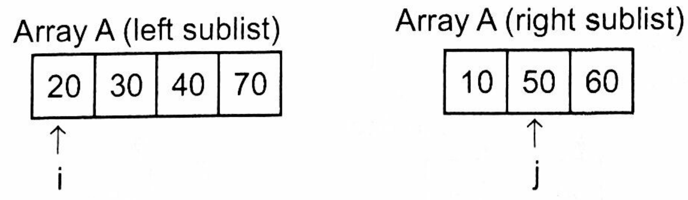
Initially k = 0. Then k will be incremented



else part of algorithm gets executed

Note that i remains there and j is incremented

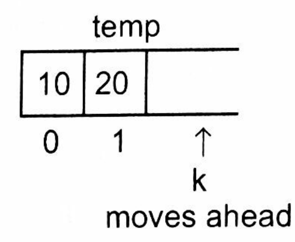




Applicable part of Algorithm

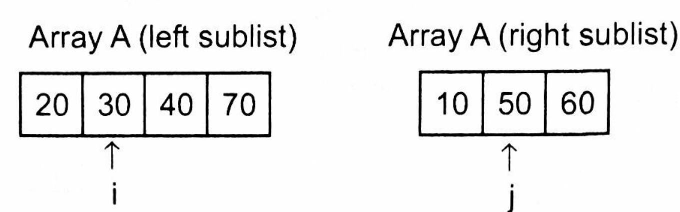
k = 1. It is advanced later on

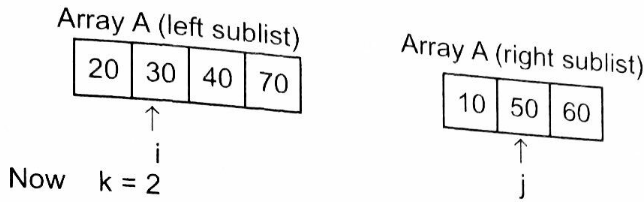
```
if (A[i]<= A[j])
{
  temp [k] ← A[i]
   i ← i+1
   k ← k+1
}
else
{
  temp [k] ← A[j]
   j ← j+1
   k ← k+1
}
```



if part of algorithm gets executed

Note that j remains there and only i is incremented





Now k = 2

temp

10	20	30	
0	1	2	↑ k

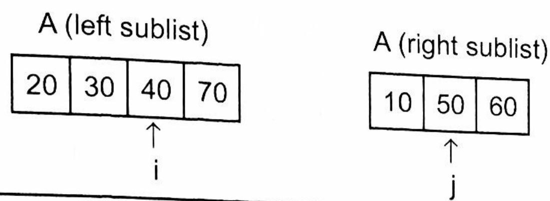
k moves ahead

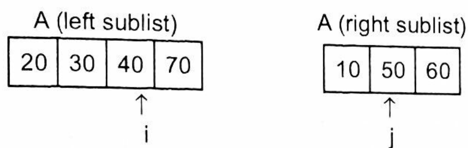
if part gets executed

Applicable part of Algorithm

```
if (A[i]<= A [j])
{
  temp [k] ← A [i]
   i ← i+1
   k ← k+1
}
else
{
  temp [k] ← A [j]
   j ← j+1
   k ← k+1
}
```

Note that j is as it is and only i is incremented





temp

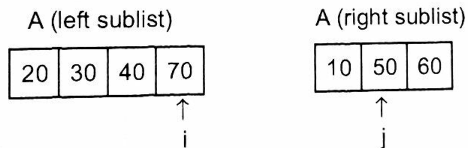
10	20	30	40	
				↑ k

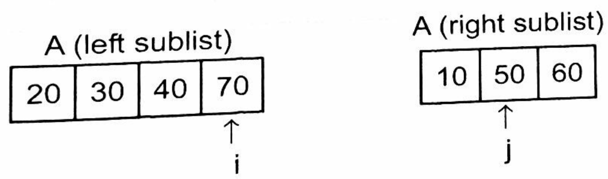
if part gets executed

Applicable part of Algorithm

```
if (A[i]<= A [j])
{
  temp [k] ← A [i]
   i ← i+1
   k ← k+1
}
else
{
  temp [k] ← A [j]
   j ← j+1
   k ← k+1
}
```

Note that j remains there and only i is incremented





temp

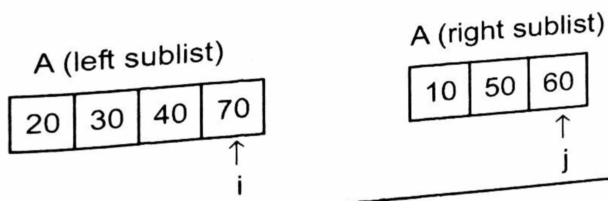
10	20	30	40	50	
					↑ k

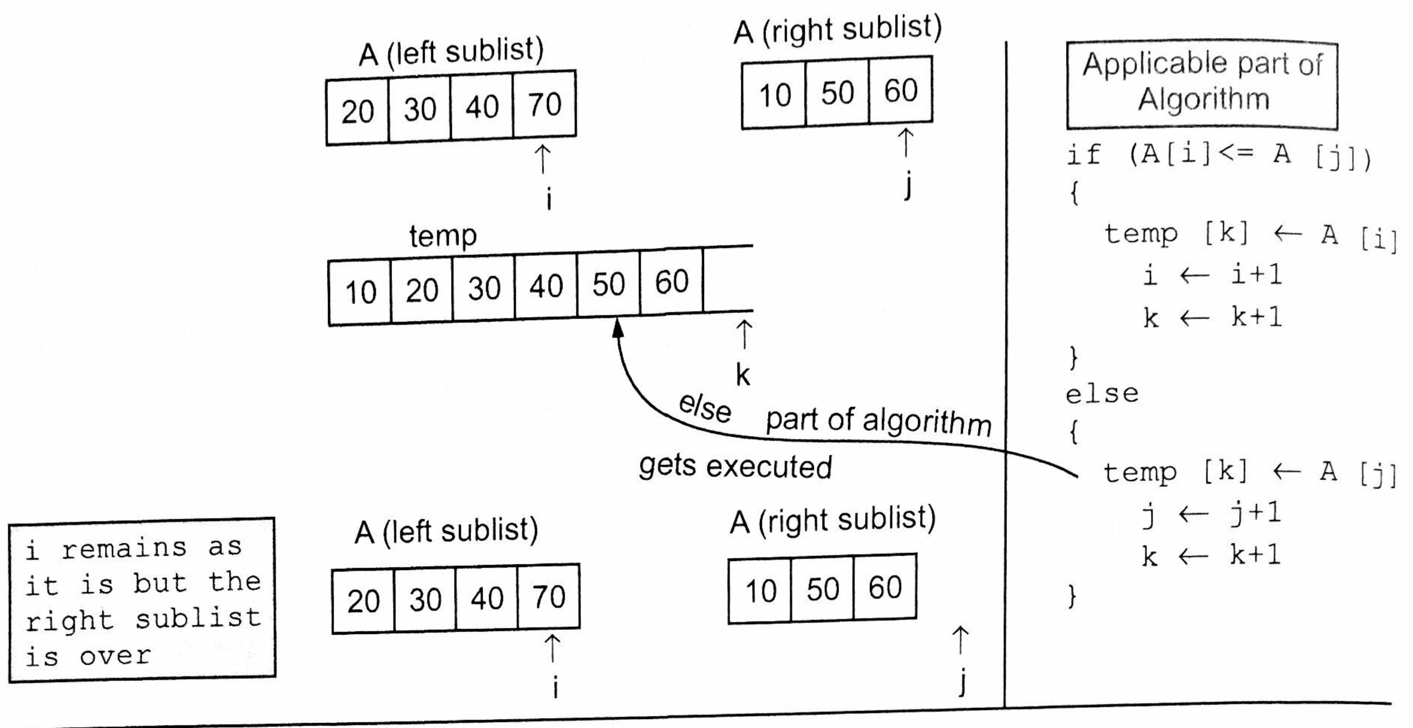
else part gets executed

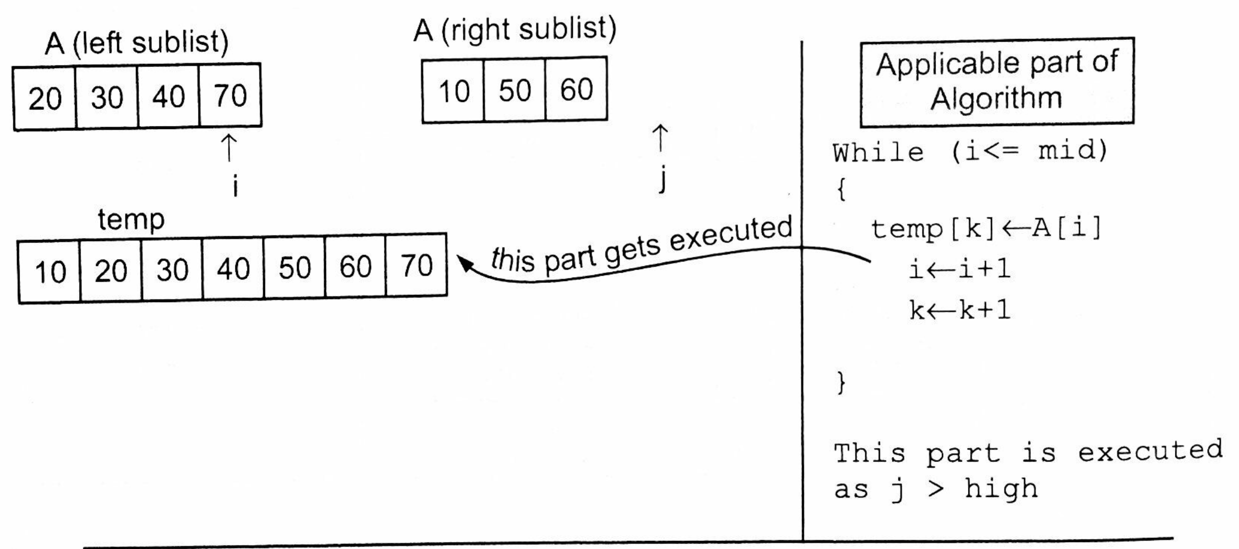
Applicable part of Algorithm

```
if (A[i]<= A [j])
{
  temp [k] ← A [i]
   i ← i+1
   k ← k+1
}
else
{
  temp [k] ← A [j]
   j ← j+1
   k ← k+1
}
```

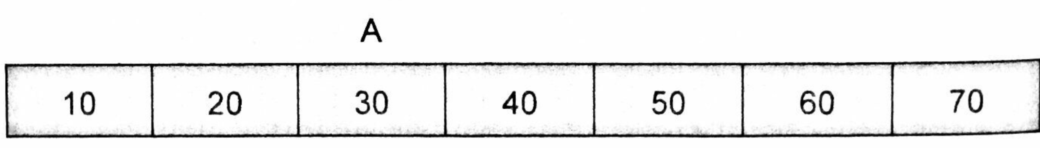
Only j and k will be incremented







Finally we will copy all the elements of array **temp** to array **A.** Thus array A contains sorted list.



```
    while(j<=high)
      {
        temp[k]=A[j];
        j++;
       k++;
     }
   //copy the elements from temp array to A
    for(k=low;k<=high;k++)
      A[k]=temp[k];
   }
```

Analysis

In merge sort algorithm two recursive calls are made. Each recursive call focuses on n/2 elements of the list. After two recursive calls one call is made to combine two sublists i.e. to merge all the elements. We can write it as -

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

$T(n/2)$	$T(n/2)$	Cn
↓	↓	↓
Time taken by left sublist to get sorted	Time taken by right sublist to get sorted	Time for combining two soblists

$$T(n) = O(n\log_2 n)$$

The average and worst case time complexity of merge sort is $\mathbf{O(n\log_2 n)}$.

```
  {
      String str = getString();
      return Integer.parseInt(str);//converting console string to
                                   //numeric value
   }
}//end of class
```

Output

```
D:\dsf_java>javac MergeSrt.java
D:\dsf_java>java MergeSrtDemo

        Program for Merge Sort

 How many elements are there?
7

 Enter the elements
30
20
10
40
60
70
50

 The Elements are...
30
20
10
40
60
70
50

 The Sorted Elements are...
10
20
30
40
50
60
70
```

4.2.7 Radix Sort

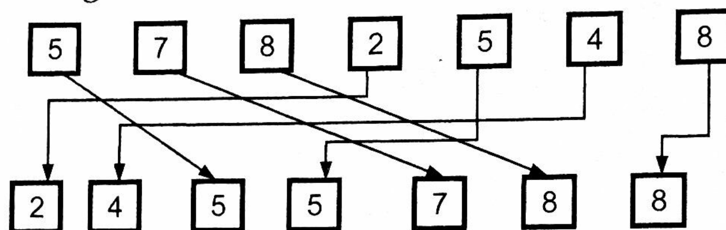
This is a sorting method in which data is sorted by scanning each digit of every element. We sort the list based on least significant bit (LSB) in first pass and then move towards most significant bit (MSB) which results in sorted data.

Algorithm :

1. Read the total number of elements to be sorted.
2. Store the unsorted elements in the array.
3. Start sorting the elements digit by digit, starting from LSB to MSB.
4. Store sorted elements in the array and print them.

For example :

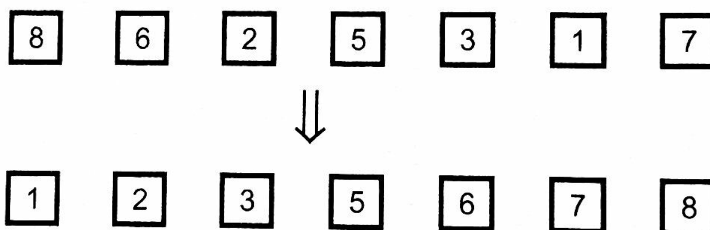
Consider following list of elements 25, 37, 18, 82, 55, 64, 78 we will now consider LSB of elements and arrange the data accordingly.



Then corresponding numbers indicated by unit's places are -

82	64	25	55	37	18	78

Now let us consider ten's place of each element and sort the data accordingly.



The corresponding numbers indicated by tens places are -

18	25	37	55	64	78	82

Thus we get a sorted list.

4.2.8 External Sorting

When we want to handle a large amount of data then it is not possible to sort such a huge data using internal sort. And most important is that such a sorting is not possible using main memory simply. What we need is a disk memory to perform such data.

The data stored on secondary memory is part by part loaded into main memory, sorting can be done over there. The sorted data can be then stored in the intermediate files. Finally these intermediate files can be merged repeatedly to get sorted data. Thus **huge amount** of data can be sorted using this technique.

For example :

Consider that there are 10,000 records that has to be sorted. Clearly we need to apply external sorting method. Suppose main memory has a capacity to store 500 records in blocks, with each block size of 100 records.

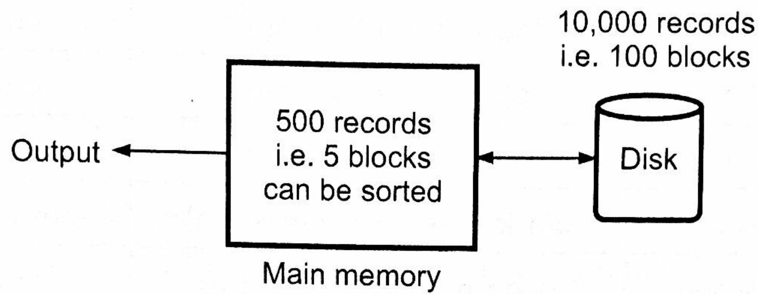


Fig. 4.3

The sorted 5 blocks (i.e. 500 records) are stored in intermediate file. This process will be repeated 20 times to get all the records sorted in chunks.

In the second step, we start merging a pair of intermediate files in the main memory to get output file.

F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13 F14 F15 F16 F17 F18 F19 F20

G1 G2 G3 G4 G5 G6 G7 G8 G9 G10

H1 H2 H3 H4 H5

I1 I2 I3

J1 J2

K1 Sorted record

Fig. 4.4

4.2.9 Analysis of all the Sorting Method

Sorting	Best case	Average case	Worst case
Bubble sort	$o(n^2)$	$o(n^2)$	$o(n^2)$
Insertion sort	o(n)	$o(n^2)$	$o(n^2)$
Shell sort	o(n)	$o(n^2)$	$o(n^2)$
Heap sort	o(n log n)	o(n log n)	o(n log n)
Merge sort	o(n log n)	o(n log n)	o(n log n)
Quick sort	0(log n)	o(log n)	$o(n^2)$

Sorting is a technique for arranging the data in some order. This arrangement is useful for searching any data from large amount of data storage. This data storage is usually refered as databases. In our real life, dictionary or directiories are the typical examples of sorted data. Usually sorting can be done using internal sorting or external sorting. Real applications of sorting technique is for handling databases.

4.3 Searching Techniques

In this section we will learn some searching techniques. The basic characteristics of any searching algorithm is – the searching method should be efficient, it should have less number of computations involved into it as well as the space occupied by such a technique should be less. Of course, the most perfect searching technique, having all the above mentioned properties may not be existing! There must be some compromise! Some good and bad points of every method are there. And so, we will discuss the advantages and disadvantages of each method.

The searching can be done by various methods. In this section we will discuss various techniques such as-

1. Sequential Search.
2. Index sequential Search.
3. Binary Search.

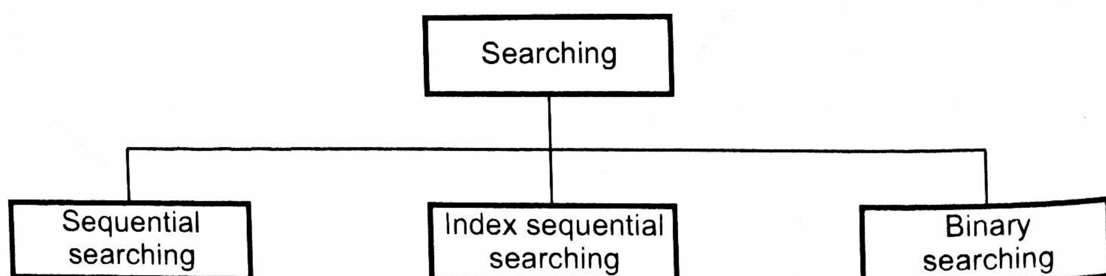


Fig. 4.5

4.4 Sequential Search

A record is defined as a group of one or more fields which collectively describe some entity or an object. All these records representing distinct entities or objects are placed in one single file. When some operations are to be performed on these records these records are brought into main memory and are normally stored into an array of records. Now the main issue is when these records are brought into an array how we can distinguish one record from the other. The very simple answer to this is that from the record itself some field will be taken to differentiate all the records. Such a field is called as key. For example in the students database all the records of students are stored. Now these records can be distinguished from each other by some key let's say roll number. Every roll number is different and every students record can be identified from his roll number. Here the key field should be roll number only!. Imagine if set the key field as the name of the student, which is not appropriate to distinguish the record from each other. Because- two names can be the same.

Linear search does not expect specific ordering of the data. Infact the data is arranged in the list. At every iteration the record will be compared with the help of the key. In this case sometimes all the elements get compared with the key value. All though this is an simple searching technique. Some unnecessary comparisons has to be performed. This method does not give the satisfactory solution for the system for large number of elements.

The time complexity of this algorithm is O(n). The time complexity will increase linearly with the value of n. For higher value of n the linear search is not the satisfactory solution.

Array

	Roll no	Name	Marks
0	15	Parth	96
1	2	Anand	40
2	13	Lalita	81
3	1	Madhav	50
4	12	Arun	78
5	3	Jaya	94

Fig. 4.6 Represents students Database for sequential search

From the above Fig. 4.6 the array is maintained to store the students record. The record is not sorted at all. If we want to search the student's record whose roll number is 12 then with the key-roll number we will see the every record whether it is of roll number = 12. We can obtain such a record at Array [4] location.

```
numeric value
   }
}//end of class
```

Output(Run 1)

```
D:\dsf_java>javac SeqSearch.java
D:\dsf_java>java SeqSearchDemo

 Inserting  items in the array

 The elements are
  20
  10
  30
  50
  40
  60

 Enter the elements to be searched
40

 The element is present in the list
```

Output(Run 2)

```
D:\dsf_java>java SeqSearchDemo

 Inserting  items in the array

 The elements are
  20
  10
  30
  50
  40
  60

 Enter the elements to be searched
98

 The element is not present in the list
```

4.4.1 Indexed Sequential Search

Since we are clear with sequential serching lets now got to other type which is 'Indexed Sequential' searching. There are many advantages of using an 'indexed sequential' over a normal sequential. Firstly, in an indexed sequential format, we maintain two files – 1. Normal sequential file

2. A sorted index file.

Whatever is our data we store it in the sequential file and in the index file we have the id or say the primary key of the sequential file along with the offset of that particular record in the sequential file.

To be more precise let's take the following example:

Example

We have to maintain grossery details, where in the sequential file we have

itemno. name price type etc.

To refer to any item if we were using normal sequential file, we would have to scan the entire file to find that particular item details. But with the indexed sequential we have another index file which will in this case contain

itemno. offset

This will be in sorted format for the index file. Hence whenever we are referring to any record, first the index file will be searched. From that search the offset is retrieved and the required record can be seek from the sequential file.

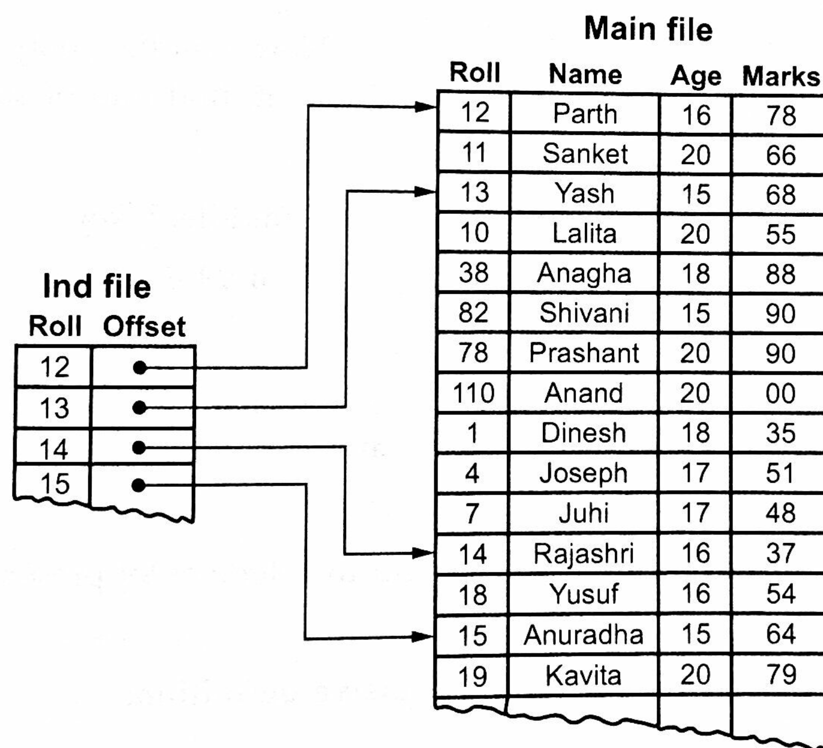


Fig. 4.7 Index sequential file

Advantages of Indexed Sequential Search :

1. More efficient than sequential file.
2. Time required is relatively less than sequential files.

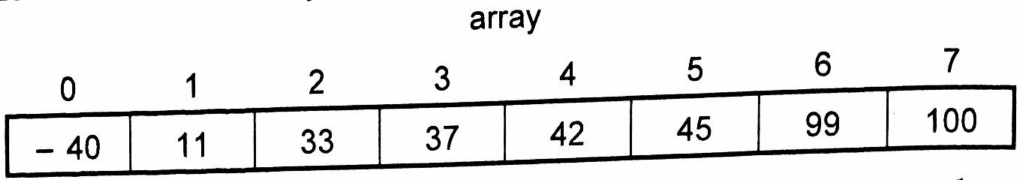
Disadvantages :

Maintaining two files relatively increases the overhead.

4.5 Binary Search

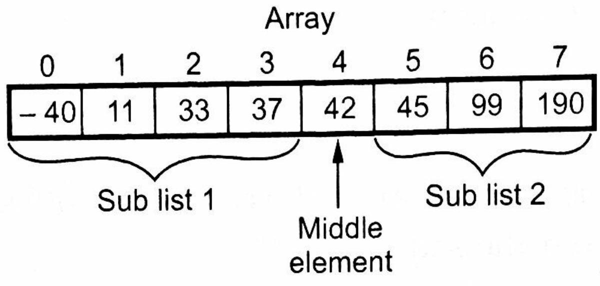
Example :

As mentioned earlier the necessity of this method is that all the elements should be sorted. So let us take an array of sorted elements.



Step 1 : Now the key element which is to be searched is = 99 ∴ key = 99.

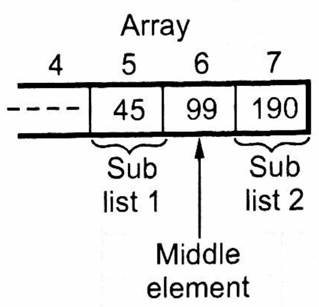
Step 2 : Find the middle element of the array. Compare it with the key



if middle $\underset{=}{?}$ key

i.e. if 42 $\underset{=}{?}$ 99

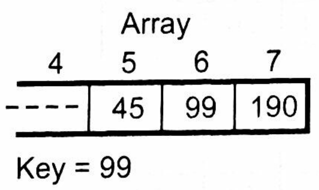
if 42 < 99 search the sublist 2



Now handle only sublist 2. Again divide it, find mid of sublist 2

if middle $\underset{=}{?}$ key

i.e. if 99 $\underset{=}{?}$ 99



So match is found at 7th position of array i.e. at array [6]

Thus by binary search method we can find the element 99 present in the given list at array $[6]^{th}$ location.

Algorithm For Binary Search Using Recursive definition:

1. if(low>high)
2. return;
3. mid =(low+high)/2;
4. if(x==a[mid])
5. return (mid);
6. if(x<a[mid])
7. search for x in a[low] to a[mid-1];

8. else
9. search for x in a[mid+1] to a[high];

Java Program

```
/*********************************************************************
Program for searching the number by Binary search. The list
of numbers should be in ascending order. The program also
shows the location at which the number is present(if at all)
*********************************************************************/
import java.io.*;
import java.util.*;
class BinSearch
  {
  private int size;
  private int[] a;
  private int index;
  public int n=0;

//--------------------------------------------------------------------
//                    constructor defined
//--------------------------------------------------------------------
  public BinSearch(int Max)
      {
      size = Max;
      a = new int[size];
      index=0;
       }
//--------------------------------------------------------------------
//                inserting an element in the array
//--------------------------------------------------------------------
  public void insert(int val)
      {
       try
       {
        a[index++] = val;
        n=index;
        }catch(Exception e)
        {
         System.out.println("e.getMessage");
       }
    }
//--------------------------------------------------------------------
//  Searching the element by Binary search
//  method
//--------------------------------------------------------------------
 public void Search(int low,int high,int key)
 {
  int mid;
  if(low>high)
  {
```

4.6 Analysis of Searching Algorithms

As we have discussed basic searching algorithms, its a time to analyze which one is a faster and slower algorithm. Normally, sequential search has a poor time complexity than index sequential search. And Binary search is supposed to be the efficient algorithm as compare to other. Here is a time efficiency of these algorithms :

Searching Algorithm	Time Complexity
Sequential search	0 (n)
Binary search	0 (logn)

4.7 Hashing

Hashing is an effective way to reduce the number of comparisons. Actually hashing deals with the idea of proving the direct address of the record where the record is likely to store. To understand the idea clearly let us take an example-

Suppose the manufacturing company has an inventory file that consists of less than 1000 parts.Each part is having unique 7 digit number.The number is called 'key' and the particular keyed record consists of that part name.If there are less than 1000 parts then a 1000 element array can be used to store the complete file.Such an array will be indexed from 0 to 999.Since the key number is 7 digit it is converted to 3 digits by taking only last three digits of a key.This is shown in the Fig. 4.8.

(See Fig. 4.8 on next page.)

Observe in Fig. 4.8 that the first key 496700 and it is stored at 0th position. The second key is 8421002. The last three digits indicate the position 2nd in the array. Let us search the element 4957397. Naturally it will be obtained at position 397. This method of searching is called hashing. The function that converts the key (7 digit) into array position is called hash function.

Here hash function is

h(key) = key % 1000

Where key %1000 will be the hash function and the key obtained by hash function is called hash key.

Collision : From above discussion it is now clear that the hash function is that function which simply takes some value as key value, performs some computations on it and gives (returns) us the key which is actually the address where the desired record can be placed. Thus this function helps us in placing the record in the hash table at appropriate position and due to which we can retrieve the record directly from that location. Now this function can be anything. The programmer has to design it on his own. Such functions are to be designed very carefully and main precaution

Position	Key	Record
0	4967000	
1		
2	8421002	
3		
⋮		
395		
396	4618396	
397	4957397	
398		
399	1286399	
400		
401		
402		
403		
404		
405		
406		
407		
408		
⋮		
990	0000990	
991	0000991	
992	1200992	
993	0047993	
994		
995	9846995	
996	4618996	
997	4967997	
998		
999	0001999	

Fig. 4.8 Hashing

for such functions is that it should not return the same key address (same hash keys) for two different records. The situation in which the hash function returns the same addresses (same hash keys) for more than one records is called collision. Note one thing that occurrences of collision mean poor design for the hash functions.

Rules for Choosing Good Hash Function

1. The hash function should be simple to compute.
2. Number of collisions should be less while placing the record in the hash table. Ideally no collision should occur. Such a function is called perfect hash function.
3. Hash function should produce such keys which will get distributed uniformly over an array.
4. The hash function should depend on every bit of the key. Thus the hash function that simply extracts the portion of a key is not suitable.

If collisions occur then it should be handled by applying some techniques, such a technique is called collision handling techniques. Let us discuss these in detail.

4.8 Collision Handling Techniques

1. Linear Probing

When collision occurs i.e. when two records demand for the same location in the hash table, then the collision can be solved by placing second record linearly down whereever the empty location is found.

For example : Consider the elements 131, 21, 31, 4, 5, 7, 8, 61.

Index	data
0	
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	

Fig. 4.9 Linear Probing

In the hash table given in Fig. 4.9 the hash function used is number % 10. If the first number which is to be placed is 131 then 131%10 = 1. i.e. remainder is 1 so hash key = 1. That means we are supposed to place the record at index 1. Next number is 21 which gives hash key = 1 as 21 % 10 = 1. But already 131 is placed at index 1. That means collision is occurred. We will now apply linear probing. In this method, we will search the place for number 21 from location of 131. In this case we can place 21 at index 2. Then 31 at index 3. Similarly 61 can be stored at 6 because number 4 and 5 are stored before 61. Because of this technique, the searching becomes efficient, as we have to search only limited list to obtain the desired number.

2. Chaining without replacement

In collision handling method **chaining** is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs we store the second colliding data by linear probing method.The address of this colliding data can be stored with the first colliding element in the chain table, without replacement.

For example consider elements

131, 3, 4, 21, 61, 6, 71, 8, 9

Index	Data	Chain
0	–1	–1
1	131	2
2	21	5
3	3	–1
4	4	–1
5	61	7
6	6	–1
7	71	–1
8	8	–1
9	9	–1

Fig. 4.10 Chaining without replacement

From the example, you can see that the chain is maintained the number who demands for location 1. First number 131 comes we will place at index 1. Next comes 21 but collision occurs so by linear probing we will place 21 at index 2, and chain is maintained by writing 2 in chain table at index 1 similarly next comes 61 by linear

probing we can place 61 at index 5 and chain will be maintained at index 2. Thus any element which gives hash key as 1 will be stored by linear probing at empty location but a chain is maintained so that traversing the hash table will be efficient.

The drawback of this method is in finding the next empty location. We are least bothered about the fact that when the element which actually belonging to that empty location can not obtain its location. This means logic of hash function gets disturbed.

3. Chaining with replacement

As previous method has a drawback of loosing the meaning of the hash function, to over come this drawback the method known as changing with replacement is introduced. Let us discuss the example to understand the method. Suppose we have to store following elements :

131, 21, 31, 4, 5

0	–1	–1
1	131	2
2	21	3
3	31	–1
4	4	–1
5	5	–1
6		
7		
8		
9		

Now next element is 2. As hash function will indicate hash key as 2 but already at index 2. We have stored element 21. But we also know that 21 is not of that position at which currently it is placed.

Hence we will replace 21 by 2 and accordingly chain table will be updated. See the figure :

Index	data	chain
0	–1	–1
1	131	6
2	2	– 1

3	31	–1
4	4	–1
5	5	–1
6	21	3
7	–1	–1
8	–1	–1
9	–1	–1

The value –1 in the hash table and chain table indicate the empty location.

The advantage of this method is that the meaning of hash function is preserved. But each time some logic is needed to test the element, whether it is at its proper position.