UNIT 3 BASICS OF ANALYSIS

Structure **Page Nos.**

3.0 INTRODUCTION

Analysis of algorithms is an essential tool for making well-informed decision in order to choose the most suitable algorithm, out of the available ones, if any, for the problem or application under consideration. For such a choice of an algorithm, which is based on some efficiency measures relating to computing resources required by the algorithm, there is no systematic method. To a large extent, it is a matter of judgment and experience. However, there are ***some basic techniques and principles*** that help and guide us in analysing algorithms. These techniques are mainly for

(i) analysing control structures and

(ii) for solving recurrence relations, which arise if the algorithm involves recursive structures.

In this unit, we mainly discuss models, techniques and principles for analyzing algorithms.

Also, sorting algorithms, which form good sources for learning how to design and analyze algorithms, are discussed in detail in this unit.

It appears that the whole of the conditions which enable a **finite** machine to make calculations of **unlimited** extent are fulfilled in the Analytical Engine…I have converted the infinity of space, which was required by the conditions of the problem, into the infinity of time.

Charles Babbage
About
Provision of iteration & conditional branching in the design of his analytical engine (year 1834), the first general purpose digital computer

3.1 OBJECTIVES

After going through this Unit, you should be able to:

- explain and use various types of analyses of algorithms;
- tell how we compute complexity of an algorithm from the complexities of the basic instructions using the structuring rules;
- solve recurrence equations that arise in recursive algorithms, and
- explain and use any one of the several well-known algorithms discussed in the text, for sorting a given array of numbers.

3.2 ANALYSIS OF ALGORITHMS – SIMPLE EXAMPLES

In order to discuss some simple examples of analysis of algorithms, we write two algorithms for solving the problem of computing prefix averages (*to be defined*). And then find the complexities of the two algorithms. In this process, we also find that how minor change in an algorithm may lead to substantial gain in the efficiency of an algorithm. To begin with we define the *problem of computing prefix average*.

Computing Prefix Averages: For a given array A[1..n] of numbers, the problem is concerned with finding an array B[1..n] such that

B[1] = A[1]
B[2] = average of first two entries = (A[1] + A [2])/2
B[3] = average of first 3 entries = (A[1] + A[2] + A[3])/3

and in general for $1 \leq i \leq n$,

B[i] = average of first i entries in the array A[1..n]
= (A[1] + A[2] + …+A[i])/i

Next we discuss two algorithms that solve the problem, in which second algorithm is obtained by minor modifications in the first algorithm, but with major gains in algorithmic complexity – the first being a *quadratic* algorithm, whereas the second algorithm is *linear*. Each of the algorithms takes array A[1..n] of numbers as input and returns the array B[1..n] as discussed above.

Algorithm First-Prefix-Average (A[1..n])

```
begin {of algorithm}
  for i ← 1 to n do
     begin {first for-loop}
       Sum ← 0;
        {Sum stores the sum of first i terms, obtained in different
          iterations of for-loop}

     for j ← 1 to i do
      begin {of second for-loop}
        Sum ← Sum + A[j];
      end {of second for- loop}
          B[i] ← Sum/i
      end {of the first for-loop}
end {of algorithm}
```

Analysis of First-Prefix-Averages:

***Step 1*:** Intitalization step for setting up of the array A[1..n] takes constant time say C_1, in view of the fact that for the purpose, *only address of A (or of A[1]) is to be passed*. Also after all the values of B[1..n] are computed, then returning the array B[1..n] also takes constant time say C_2, again for the same reason.

***Step 2*:** The body of the algorithm has two nested for-loops, the outer one, called the *first for-loop* is controlled by i and is executed n times. Hence the *second for-loop alongwith its body,* which form a part of the *first for-loop,* is executed n times. Further each construct within *second for-loop*, controlled by j, is executed i times just because of the iteration of the *second for-loop*. However, the *second for-loop* itself is being executed n times because of the first for-loop. Hence each instruction within the *second for-loop* is executed (n.i) times for each value of i = 1, 2, …n.

***Step 3*:** In addition, each controlling variable i and j is incremented by 1 after each iteration of i or j as the case may be. Also, after each increment in the control variable, it is compared with the (*upper limit + 1*) of the loop to stop the further execution of the for-loop.

Thus, the *first for-loop* makes n additions (*to reach* (*n+1*)) and n comparisons with (n+1).

The second for-loop makes, for each value of i=1,2,…,n, one addition and one comparison. Thus total number of each of additions and comparisons done just for controlling variable j

$= (1+2+\ldots+n) = \frac{n(n+1)}{2}$.

***Step 4*:** Using the explanation of Step 2, we count below the number of times the various operations are executed.

(i) Sum $\leftarrow$ 0 is executed n times, once for each value of i from 1 to n

(ii) On the similar lines of how we counted the number of additions and comparisons for the control variable j, it can be seen that the number of each of additions ($Sum \leftarrow Sum + A[j]$) and divisions ($B[i] \leftarrow Sum/i$) is $\frac{n(n+1)}{2}$.

Summerizing, the total number of operations performed in executing the First-Prefix-Averages are

(i) (*From Step* 1) Constant C_1 for initialization of A[1..n] and Constant C_2 for returning B[1..n]

(ii) (*From Step 3*)

Number of additions for control variable i = n.
Number of Comparisons with (n+1) of variable i = n
Number of additions for control variable j = $\frac{n(n+1)}{2}$
Number of comparisons with (i+1) of control variable j = $\frac{n(n+1)}{2}$
Number of initializations ($Sum \leftarrow 0$) = n.
Number of additions ($Sum \leftarrow Sum + A[j]$) = $\frac{n(n+1)}{2}$
Number of divisions (*in Sum/i*) = $\frac{n(n+1)}{2}$

Number of assignments in ($Sum \leftarrow Sum + A[j]$) = $\frac{n(n+1)}{2}$

Number of assignments in ($B[i] \leftarrow Sum/i$) = $\frac{n(n+1)}{2}$

Assuming each of the operations counted above takes some constant number of unit operations, then total number of all operations is a **quadratic function of n**, the size of the array A[1..n].

Next, we show that a minor modification in the First-Prefix-Algorithm, may lead to an algorithm, to be called Second-Prefix-Algorithm and defined below of *linear complexity only.*

The main change in the algorithm is that the partial sums $\sum_{k+1}^{i} A[k]$ are not computed afresh, repeatedly (*as was done in First-Prefix-Averages*) but $\sum_{k+1}^{i} A[k]$ is computed only once in the variable *Sum* in Second_Prefix_Average. The new algorithm is given below:

Algorithm Second-Prefix-Averages (A[1..n]);

```
begin {of algorithm}
        Sum ← 0
  for i ← 1 to n do
    begin {of for loop}
          Sum ← Sum + A[i];
          B[i] ← Sum/i.
    end; {of for loop}
  return (B[1..n]);
end {of algorithm}
```

Analysis of Second-Prefix-Averages

***Step1*:** As in First-Prefix-Averages, the intialization step of setting up the values of A[1..n] and of returning the values of B[1..n] each takes a constant time say C_1 and C_2 (*because in each case only the address of the first element of the array viz A[1] or B[1] is to be passed*)

The assignment through Sum ← 0 is executed once

The *for loop* is executed exactly n times, and hence

***Step 2*:** There are *n additions* for incrementing the values of the loop variable and *n comparisons* with (n+1) in order to check whether *for loop* is to be terminated or not.

***Step 3*:** There *n additions*, one for each i (*viz Sum + A* [*i*]) and n *assignments*, again one for each i (*Sum* ← *Sum* + *A*[*i*]). Also there are n divisions, one for each i (*viz Sum/i*) and n (more) assignments, one for each i (*viz B*[*i*] ← *Sum/i*).

Thus we see that overall there are

(i) 2 n additions
(ii) n comparisons
(iii) (2n+1) assignments
(iv) n divisions
(v) C_1 and C_2, constants for initialization and return.

As each of the operations, viz addition, comparison, assignment and division takes a constant number of units of time; therefore, the total time taken is C.n for some constant C.

Thus Second-Prefix-Averages is a *linear algorithm* (or has linear time complexity)

3.3 WELL KNOWN SORTING ALGORITHMS

In this section, we discuss the following well-known algorithms for sorting a given list of numbers:

1. Insertion Sort
2. Bubble Sort
3. Selection Sort
4. Shell Sort
5 Heap Sort
6. Merge Sort
7. Quick Sort

For the discussion on Sorting Algorithms, let us recall the concept of *Ordered Set*.

We know given two integers, say n_1 and n_2, we can always say whether $n_1 \leq n_2$ or $n_2 \leq n_1$. Similarly, if we are given two rational numbers or real numbers, say n_1 and n_2, then it is always possible to tell whether $n_1 \leq n_2$ or $n_2 \leq n_1$.

Ordered Set: Any set S with a relation, say, ≤, is said to be ordered if for any two elements x and y of S, either $x \leq y$ or $y \leq x$ is true. Then, we may also say that (S, ≤) is an ordered set.

Thus, if I, Q and R respectively denote set of integers, set of rational numbers and set of real numbers, and if '≤' *denotes* '*the less than or equal to*' relation then, each of (I, ≤), (Q, ≤) and (R, ≤) is an ordered set. However, it can be seen that the set $C = \{x + iy : x, y \in R \text{ and } i^2 = -1\}$ of complex numbers *is not ordered* w.r.t '≤'. For example, it is not possible to tell for at least one pair of complex numbers, say 3 + 4i and 4+3i, whether 3 + 4i ≤ 4 +3i, or 4 + 3i ≤ 3 + 4i.

Just to facilitate understanding, we use the list to be sorted as that of numbers. However, the following discussion about sorting is equally valid for a list of elements from an **arbitrary ordered set**. In that case, we use the word *key* in stead of *number* in our discussion.

The general treatment of each sorting algorithm is as follows:

1. First, the method is briefly described in English.
2. Next, an example explaining the method is taken up.
3. Then, the algorithm is expressed in a pseudo- programming language.
4. The analysis of these algorithm is taken up later at appropriate places.

All the sorting algorithms discussed in this section, are for sorting numbers in *increasing order*.

Next, we discuss sorting algorithms, which form a rich source for algorithms. Later, we will have occasions to discuss general polynomial time algorithms, which of course include linear and quadratic algorithms.

One of the important applications for studying Sorting Algorithms is the area of designing efficient algorithms for searching an item in a given list. If a set or a list is ***already sorted***, then we can have more efficient searching algorithms, which include

binary search and B-Tree based search algorithms, each taking ($c.\ log\ (n)$) time, where n is the number of elements in the list/set to be searched.

3.3.1 Insertion Sort

The insertion sort, algorithm for sorting a list L of n numbers represented by an array A[1..n] proceeds by picking up the numbers in the array from left one by one and each newly picked up number is placed at its relative position, w.r.t the sorting order, among the earlier ordered ones. The process is repeated till the each element of the list is placed at its correct relative position, i.e., when the list is sorted.

Example 3.3.1.1

A[1]	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**A[6]**	
80	**32**	**31**	**110**	**50**	**40**	{← *given list, initially*}

{*Pick up the left-most number 80 from the list, we get the sorted sublist*}

80

{*Pick up the next number 32 from the list and place it at correct position relative to 80, so that the sublist considered so far is sorted*}.

32 80

{*We may note in respect of the above sorted sublist, that in order to insert 32 before 80, we have to shift 80 from first position to second and then insert 32 in first position.*

The task can be accomplished as follows:

1. *First 32 is copied in a location say m*
2. *80 is copied in the location A[2] = 32 so that we have*

A[1]	*A[2]*	*m*
80	*80*	*32*

3. *32 is copied in A[1] from m so that we have*

A[1]	*A[2]*	*m*
32	*80*	*32*

thus we get the sorted sublist given above}

32 80

{*Next number 31 is picked up, compared first with 80 and then* (*if required*) *with 32. in order to insert 31 before 32 and 80, we have to shift 80 to third position and then 32 to second position and then 31 is placed in the first position*}.

The task can be accomplished as follows:

1. *First 31 is copied in a location say m*
2. *80 is copied in the location A[3] = 31 so that we have*

A[1]	*A[2]*	*A[3]*	*m*
32	*80*	*80*	*31*

3. *32 is copied in A[2] from A[1] so that we have*

A[1]	*A[2]*	*A[3]*	*m*
32	*32*	*80*	*31*

4. *the value 31 from m is copied in A[1] so that we get*

A[1]	*A[2]*	*A[3]*	*m*
31	*32*	*80*	*31*

thus we get the sorted sublist}

31 32 80

{*Next 110 is picked up, compared with 80. As 110>80, therefore, no shifting and no more comparisons. 110 is placed in the first position after 80}.*

31 32 80 110

{*Next, number 50 is picked up. First compared with 110, found less; next compared with 80, again found less; again compared with 32. The correct position for 50 is between 32 and 80 in the sublist given above. Thus, each of 110 and 80 is shifted one place to the right to make space for 50 and then 50 is placed over there*

The task can be accomplished as follows:

1. *First 50 is copied in a location say m*
2. *110 is copied in the location A[5] = 50 so that we have*
   A[1] A[2] A[3] A[4] A[5] m
   31 32 80 110 110 50
3. *80 is copied in A[4] from A[3] so that we have*
   A[1] A[2] A[3] A[4] A[5] m
   31 32 80 80 110 50
4. *the value 50 from m is copied in A[3] so that we get*
   A[1] A[2] A[3] A[4] A[5] m
   31 32 50 80 110 50

thus we get the following sorted sublist}

31 32 50 80 110

{*Next in order to place 40 after 32 and before 50, each of the values 50, 80 and 110 need to be shifted one place to the right as explained above. However, values 31 and 32 are not to be shifted. The process of inserting 40 at correct place is similar to the ones explained earlier*}.

31 32 40 50 80 110

Algorithm: The Insertion Sort

The idea of Insertion Sort as explained above, may be implemented through procedure *Insertion-Sort* given below. It is assumed that the numbers to be sorted are stored in an array A[1..n].

Procedure Insertion-Sort (A[1..n] : real)
```
 begin {of procedure}
  if n = 1
  then
   write ('A list of one element is already sorted')
   else
    begin {of the case when n ≥ 2}
     for  j ← 2 to n do

        {to find out the correct relative position for A[ j] and insert it there  among
         the already sorted elements A[1] to A [j − 1]}

   begin {of for loop}
     If A[j] < A[j − 1]  then begin
      {We shift entries only if A [j] < A [j − 1]}
      i ← j − 1;   m ← A[j]
```

{*In order to find correct relative position we store A[j] in m and start with the last element A[j−1] of the already sorted part. If m is less than A[j−1], then we move towards left, compare again with the new element of the array. The process is repeated until either m ≥ some element of the array or we reach the left-most element A[1]*}.

```
while (i > 0 and m < A[i]) do
begin {of while loop}
  A[i+1] ← A[i]
    i ← i − 1
end
```

{*After finding the correct relative position, we move all the elements of the array found to be greater than m = A[j], one place to the right so as to make a vacancy at correct relative position for A[j]*}

```
end; {of while loop}

A[i +1] ← m
  {i.e., m = A[j] is stored at the correct relative position}
 end {if}
 end; {of for loop}
 end; {of else part}
end; {of procedure}
```

Ex. 1) Sort the following sequence of number, using Insertion Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

3.3.2 Bubble Sort

The Bubble Sort algorithm for sorting of n numbers, represented by an array A[1..n], proceeds by scanning the array from left to right. At each stage, compares *adjacent pairs* of numbers at positions A[i] and A[i+1] and whenever a pair of adjacent numbers is found to be out of order, then the positions of the numbers are swapped. The algorithm repeats the process for numbers at positions A[i+1] and A[i+2].

Thus in the first pass after scanning once all the numbers in the given list, the largest number will reach its destination, but other numbers in the array, may not be in order. In each subsequent pass, one more number reaches its destination.

3.3.2.1 Example

In the following, in each line, pairs of adjacent numbers, **shown in bold**, are compared. And if the pair of numbers are not found in proper order, then the positions of these numbers are exchanged.

The list to be sorted has n = 6 as shown in the first row below:

iteration number i = 1

80	**32**	31	110	50	40	(j = 1)
32	**80**	**31**	110	50	40	(j = 2)
32	31	**80**	**110**	50	40	
32	31	80	**110**	**50**	40	
32	31	80	50	**110**	**40**	(j = 5)
32	**31**	81	50	40	*110*	(j = 1)

↑

remove from further consideration

In the first pass traced above, the maximum number 110 of the list reaches the rightmost position (i.e 6 th position). In the next pass, only the list of remaining $(n - 1) = 5$ elements, as shown in the first row below, is taken into consideration. Again pairs of numbers in bold, in a row, are compared and exchanged, if required.

iteration number i = 2

31	**32**	**81**	50	40	(j = 2
31	32	**81**	**50**	40	(j = 3)
31	32	50	**81**	**40**	(j = 4)
31	**32**	50	40	*81*	(j = 1)

↑

remove from further consideration

In the second pass, the next to maximum element of the list viz, 81, reaches the 5th position from left. In the next pass, the list of remaining $(n - 2) = 4$ elements are taken into consideration.

iteration number i = 3

31	**32**	**50**	40	(j = 2)
31	32	**50**	**40**	(j = 3)
31	**32**	40	*50*	(j = 1)

↑

remove from further consideration

In the next iteration, only (n – 3) = 3 elements are taken into consideration.

31	**32**	**40**
31	**32**	40

In the next iteration, only n – 4 = 2 elements are considered

31 32

These elements are compared and found in proper order. The process terminates.

Procedure ***bubblesort*** **(A[1..n])**
begin
for **i ← 1** to **n – 1 do**
for j ← 1 to **(n – i).**
{*in each new iteration, as explained above, one less number of elements is taken into consideration. This is why j varies upto only* $(n - i)$}

if **A[j] > A[j+1]** then **interchange A[j]and A[j+1].**
end
{A[1..n] is in increasing order}

Note: As there is only one statement in the scope of each of the two for-loops, therefore, no 'begin' and 'end' pair is used.

Ex. 2) Sort the following sequence of numbers using Bubble Sort:
15, 10, 13, 9, 12, 17.
Further, find the number of comparisons and assignments required by the algorithm in sorting the list.

3.3.3 Selection Sort

Selection Sort for sorting a list L of n numbers, represented by an array A[1..n], proceeds by finding the maximum element of the array and placing it in the last position of the array representing the list. Then repeat the process on the subarray representing the sublist obtained from the list by excluding the current maximum element.

*The difference between Bubble Sort and Selection Sort, is that in **Selection Sort** to find the maximum number* in the array, a new variable MAX is used to keep maximum of all the values scanned upto a particular stage. On the other hand, in **Bubble Sort**, the maximum number in the array under consideration is found by comparing adjacent pairs of numbers and by keeping larger of the two in the position at the right. Thus after scanning the whole array once, the maximum number reaches the right-most position of the array under consideration.

The following steps constitute the Selection Sort algorithm:

Step 1: Create a variable MAX to store the maximum of the values scanned upto a particular stage. Also create another variable say MAX-POS which keeps track of the position of such maximum values.

Step 2: In each iteration, the whole list/array under consideration is scanned once to find out the current maximum value through the variable MAX and to find out the position of the current maximum through MAX-POS.

Step 3: At the end of an iteration, the value in last position in the current array and the (maximum) value in the position MAX-POS are exchanged.

Step 4: For further consideration, replace the list L by L ~ {MAX} {*and the array A by the corresponding subarray*} and go to Step 1.

Example 3.3.3.1:

80 32 31 **110** 50 **40** {← *given initially*}

Initially, MAX ← 80 MAX-POS ←1
After one iteration, finally; MAX ← 110, MAX-POS = 4

Numbers 110 and 40 are exchanged to get

80 32 31 **40** 50 **110**

New List, after one iteration, to be sorted is given by:

80 32 31 40 50

Initially MAX ← 80, MAX-POS ←1
Finally also MAX ← 80, MAX-POS ←1
∴ entries 80 and 50 are exchanged to get

50 32 31 40 80

New List, after second iteration, to be sorted is given by:

50 32 31 40

Initially Max ← 50, MAX-POS ←1
Finally, also Max ← 50, MAX-POS ←1

∴ entries 50 and 40 are exchanged to get

40 32 31 50

New List , after third iteration, to be sorted is given by:

40 32 31

Initially & finally

Max ← 40 ; MAX-POS ← 1

Therefore, entries 40 and 31 are exchanged to get

31 32 40

New List, after fourth iteration, to be sorted is given by:

31 32

Max ← 32 and MAX-POS ← 2

∴ No exchange, as 32 is the last entry

New List, after fifth iteration, to be sorted is given by:

31

This is a single-element list. Hence, no more iterations. The algorithm terminates This completes the sorting of the given list.

Next, we formalize the method used in sorting the list.

```
Procedure Selection Sort (A[1..n])
begin {of procedure}
   for i ← 1 to (n – 1) do
        begin {of i for loop}
           MAX ← A[i];
           MAX-POS ← i
            for j ← i+1 to n do
             begin {j for-loop}*
             If MAX < A[j] then
                  begin
                      MAX-POS ←j
                      MAX ← A[j]
                   end {of if }
             end {of j for-loop}
           A [MAX-POS] ← A[n–i+1]
           A[n–i+1] ← MAX
           {the ith maximum value is stored in the ith position from right or
             equivalently in (n – i + 1) th position of the array}
           end {of i loop}
       end {of procedure}.
```

Ex. 3) Sort the following sequence of number, using Selection Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

* as there is only one statement in j-loop, we can Omit 'begin' and 'end'.

3.3.4 Shell Sort

The sorting algorithm is named so in honour of D.L. Short (1959), who suggested the algorithm. Shell Sort is also called *diminishing –increment* sort. The essential idea behind Shell-Sort is to apply any of the other sorting algorithm (generally Insertion Sort) to each of the several interleaving sublists of the given list of numbers to be sorted. In successive iterations, the sublists are formed by stepping through the file with an increment INC_i taken from some pre-defined decreasing sequence of step-sizes $INC_1 > INC_2 > \ldots > INC_i \ldots > 1$, which must terminate in 1.

Example 3.3.4.2: Let the list of numbers to be sorted, be represented by the next row.

13 3 4 12 14 **10** 5 1 8 2 **7** 9 11 6 (*n = 14*)

Initially, we take INC as 5, i.e.,

Taking sublist of elements at 1st , 6th and 11th positions, viz sublist of values 13, 10 and 7. After sorting these values we get the sorted sublist

7 10 13

Taking sublist of elements at 2nd, 7th and 12th positions, viz sublist of values 3, 5 and 9. After sorting these values we get the sorted sublist.

3 5 9

Taking sublist of elements at 3rd, 8th and 13th positions, viz sublist of values 4, 1 and 11. After sorting these values we get the sorted sublist.

1 4 11

Similarly, we get sorted sublist

6 8 12

Similarly, we get sorted sublist

2 14

{*Note that, in this case, the sublist has only two elements* ∵ *it is 5th sublist and n = 14 is less than*

$\left(\left\lfloor \frac{14}{\text{INC}} \right\rfloor \bullet \text{INC} + 5\right)$ where INC = 5}

After merging or interleaving the entries from the sublists, while maintaining the initial relative positions, we get the **New List:**

7 3 1 **6** 2 10 **5** 4 8 **14** 13 9 **11** 12

Next, take INC = 3 and repeat the process, we get sorted sublists:

5 6 7 11 14,
2 3 4 12 13 and
1 8 9 10

After merging the entries from the sublists, while maintaining the initial relative positions, we get the **New List:**

New List

5 2 1 6 3 8 7 4 9 11 12 10 14 13

Taking INC = 1 and repeat the process we get the sorted list

1 2 3 4 5 6 7 8 9 10 11 12 13 14

Note: *Sublists should not be chosen at distances which are mutliples of each other e.g. 8, 4, 2 etc. otherwise, same elements may be technique compared again and again.*

```
Procedure Shell-Sort (A[1..n]: real)
        K: integer;
{to store value of the number of increments}
        INC [1..k]: integer
{to store the values of various increments}
        begin {of procedure}
        read (k)
        for i ← 1 to (k − 1) do
         read (INC [i])
          INC [k] ← 1
           {last increment must be 1}
            for i ← 1 to k do
             {this i has no relation with previous occurrence of i}
                begin
                  j ← INC [i]
                  r ← [n/j];
                   for t ← 1 to k do
                    {for selection of sublist starting at position t}
                     begin
                       if n < r * j + t
                       then  s ← r − 1

                        else s ← r
                      Insertion-Sort (A [t… (t + s * j)])
                    end {of t-for-loop}
                 end {i for-loop}
                end {of procedure}.
```

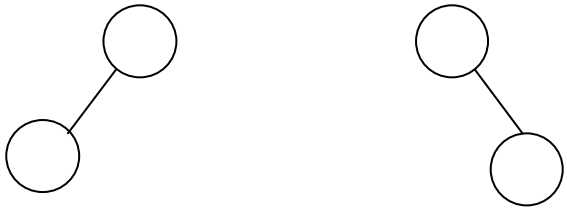
Ex. 4) Sort the following sequence of number, using Shell Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.
Also, initial INC = 3 and final INC = 1 for selecting interleaved sublists. For sorting sublists use Insertion Sort.

3.3.5 Heap Sort

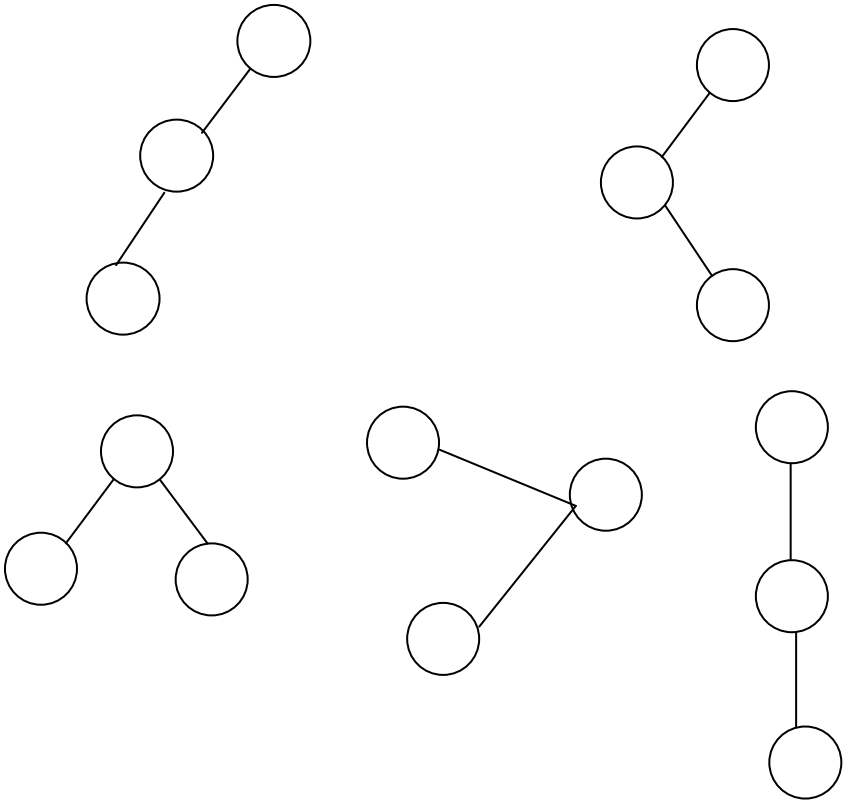
In order to discuss Heap-Sort algorithm, we recall the following definitions; where we assume that the concept of a tree is already known:

Binary Tree: A tree is called a binary tree, if it is either empty, or it consists of a node called the root together with two *binary trees* called the **left subtree** and a **right subtree**. In respect of the above definition, we make the following observations:

1. It may be noted that the above definition is a *recursive* definition, in the sense that definition of binary tree is given in its own terms (*i.e.*, *binary tree*). In Unit 1, we discussed other examples of recursive definitions.
2. The following are all distinct and the only binary trees having two nodes.



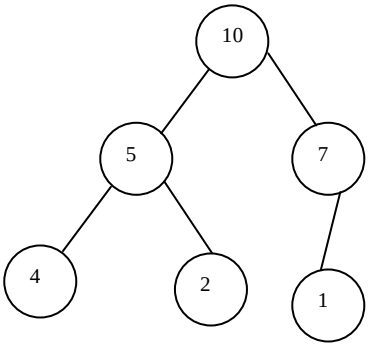
The following are all distinct and only binary trees having three nodes



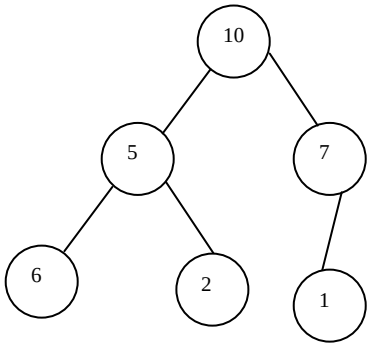
Heap: is defined as a binary tree with keys assigned to its nodes (one key per node) such that the following conditions are satisfied:

(i) The binary tree is essentially complete (or simply complete), i.e, all its levels are full except possibly the last level where only some rightmost leaves may be missing.
(ii) The key at each node is greater than or equal to the keys at its children.

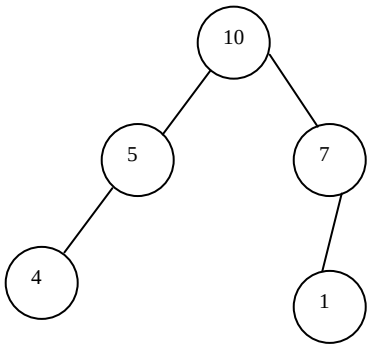
The following binary tree is a Heap



However, the following is not a heap because the value 6 in a child node is more than the value 5 in the parent node.



Also, the following is not a heap, because, some leaves (e.g., right child of 5), in between two other leaves (viz 4 and 1), are missing.



Alternative Definition of Heap:

Heap is an array H[1..n] in which every element in position i (*the parent) in the first half* of the array is greater than or equal to elements in positions 2i and and 2i+1 (*the children*):

HEAP SORT is a three-step algorithm as discussed below:

(i) *Heap Construction* for the a given array

(ii) (*Maximum deletion*) Copy the root value (which is maximum of all values in the Heap) to right-most yet-to-be occupied location of the array used to store the sorted values and copy the value in the last node of the tree (or of the corresponding array) to the root.

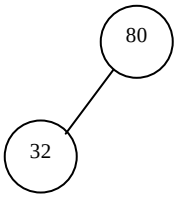
(iii) Consider the binary tree (which is not necessarily a Heap now) obtained from the Heap through the modifications through Step (ii) above and by removing currently the last node from further consideration. Convert the binary tree into a Heap by suitable modifications.

Example 3.3.5.1:

Let us consider applying Heap Sort for the sorting of the **list 80 32 31 110 50 40 120** represented by an array A[1…7]

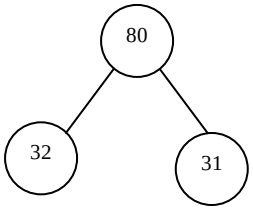
Step 1: **Construction of a Heap for the given list**
First, we create the tree having the root as the only node with node-value 80 as shown below:

Next value 32 is attached as left child of the root, as shown below



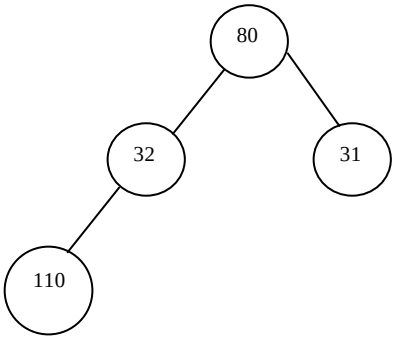
As 32 < 80, therefore, heap property is satisfied. Hence, no modification of the tree.

Next, value 31 is attached as right child of the node 80, as shown below

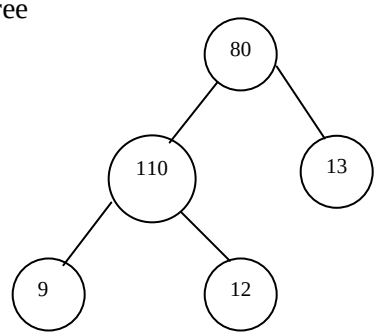


Again as 31 < 80, heap property is not disturbed. Therefore, no modification of the tree.

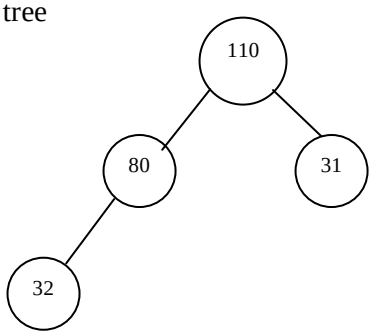
Next, value 110 is attached as left child of 32; as shown below.



However, *110 > 32*, the value in child node is more than the value in the parent node. Hence the *tree is modified* by exchanging the values in the two nodes so that, we get the following tree

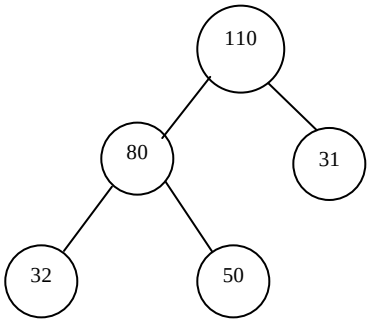


Again as *110 > 80*, the value in child node is more than the value in the parent node. Hence the *tree is modified* by exchanging the values in the two nodes so that, we get the following tree

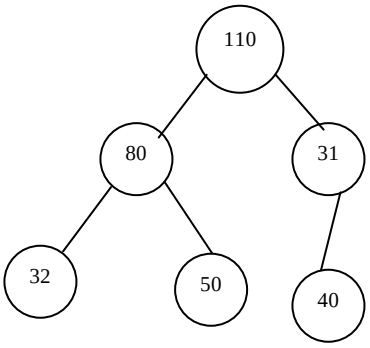


This is a Heap.

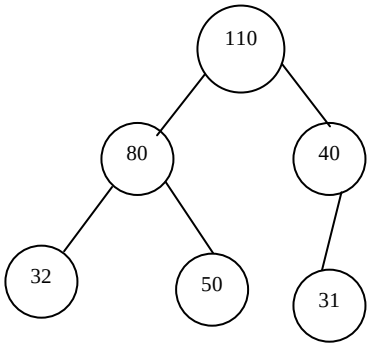
Next, number 50 is attached as right child of 80 so that the new tree is as given below



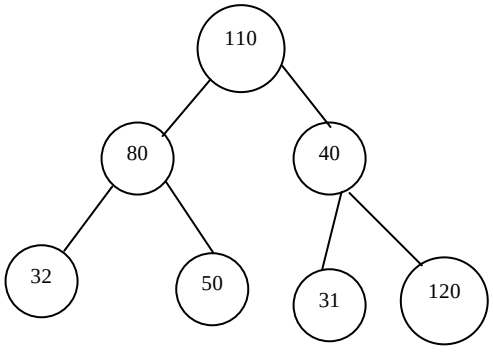
As the tree satisfies all the conditions of a Heap, we insert the next number 40 as left child of 31 to get the tree



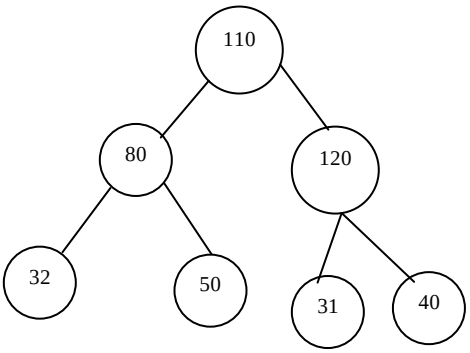
As the new insertion violates the condition of Heap, the values 40 and 31 are exchanged to get the tree which is a heap



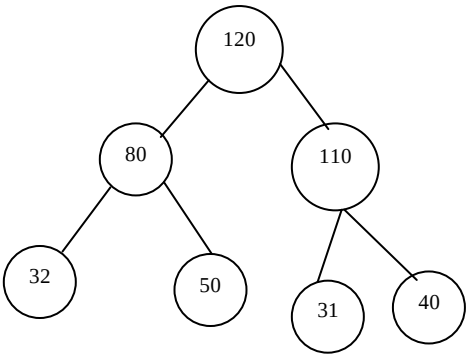
Next, we insert the last value 120 as right child of 40 to get the tree



The last insertion *violates* the conditions for a Heap. Hence 40 and 120 are exchanged to get the tree



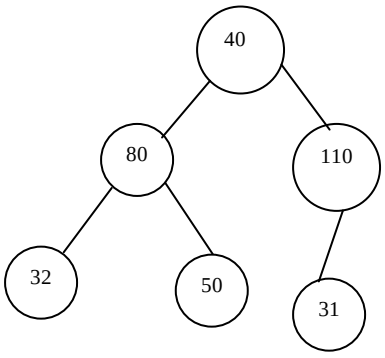
Again, due to movement of 120 upwards, *the Heap property is disturbed at nodes `110 and 120. Again 120 is moved up* to get the following tree which is a heap.



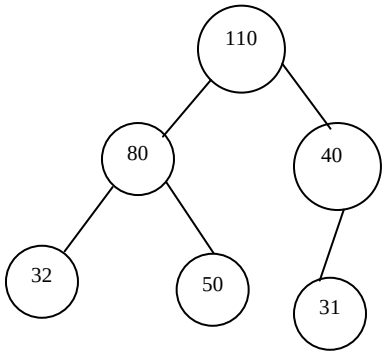
After having constructed the Heap through Step 1 above we consider

***Step 2&3*:** Consists of a sequence of actions viz (i) deleting the value of the root, (ii) **moving the last entry to the root and (iii) then** readjusting the Heap
The root of a Heap is always the maximum of all the values in the nodes of the tree. The value 120 currently *in the root is saved in the last location B[n] in our case B*[*7*] of the sorted array say B[1..n] in which the values are to be stored after sorting in increasing order.

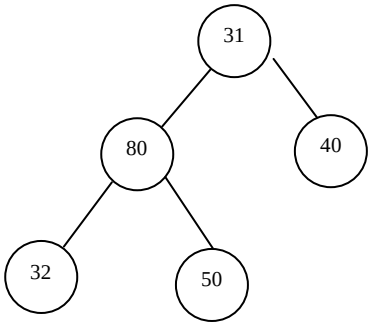
Next, value 40 is moved to the root and the node containing 40 is removed from further consideration, to get the following binary tree, which is *not* a Heap.



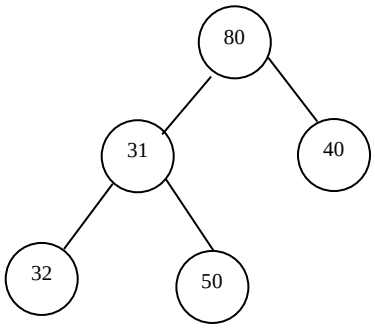
In order to restore the above tree as a Heap, the value 40 is exchanged with the maximum of the values of its two children. Thus 40 and 110 are exchanged to get the tree which is a Heap.



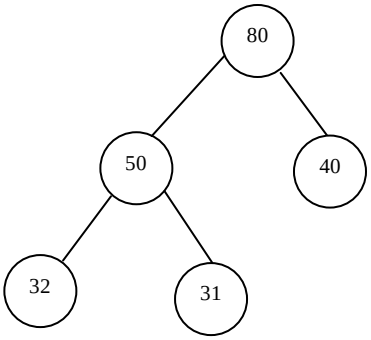
Again 110 is copied to B[6] and 31, the last value of the tree is shifted to the root and last node is removed from further consideration to get the following tree, which is not a Heap



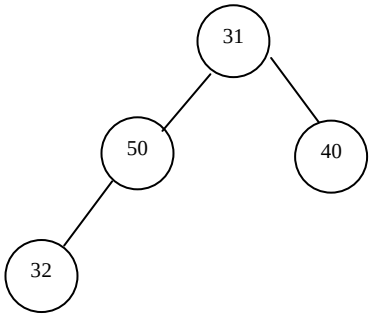
Again the root value is exchanged with the value which is maximum of its children's value i.e exchanged with value 80 to get the following tree, which again is *not* a Heap.



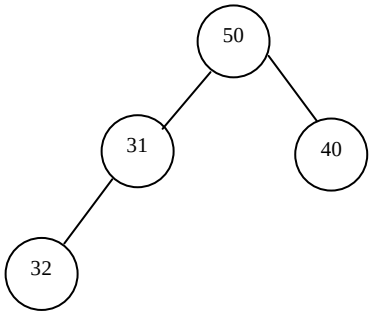
Again the value 31 is exchanged with the maximum of the values of its children, i.e., with 50, to get the tree which is a heap.



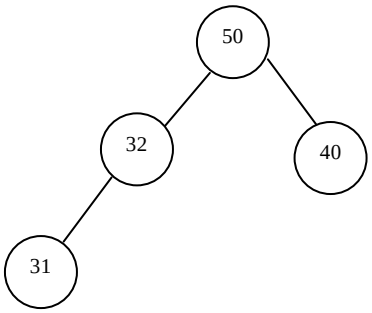
Again 80 is copied in B[5] and 31, the value of the last node replaces 80 in the root and the last node is removed from further consideration to get the tree which is not a heap.



Again, 50 the maximum of the two children's values is exchanged with the value of the root 31 to get the tree, which is *not* a heap.

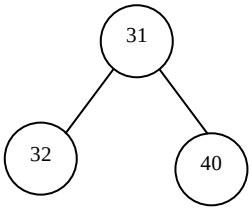


Again 31 and 32 are exchanged to get the Heap

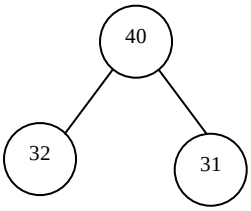


Next, 50 is copied in B[4].

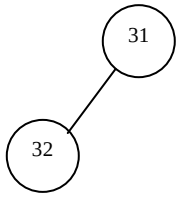
The entry 31 in the last node replaces the value in the root and the last node is deleted, to get the following tree which is *not* a Heap



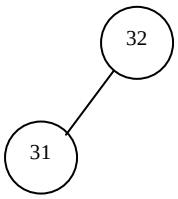
Again 40, the maximum of the values of children is exchanged with 31, the value in the root. We get the Heap



Again 40 is copied in B[3]. The value in the last node of the tree viz 31, replaces the value in the root and the last node is removed from further consideration to get the tree, which is not a Heap.



Again 32, the value of its only child is exchanged with the value of the root to get the Heap



Next, 32 is copied in B[2] and 31, the value in the last node is copied in the root and the last node is deleted, to get the tree which is a Heap.

This value is copied in B[1] and the Heap Sort algorithm terminates.

Next, we consider the two procedure

1. Build-Heap and
2. Delete-Root-n-Rebuild.

which constitute the Heap-Sort algorithm. The list L of n numbers to be sorted is represented by an array A[1..n]

The following procedure reads one by one the values from the given to-be-sorted array A[1..n] and gradually builds a Heap. For this purpose, it calls the procedure *Build-Heap*. For building the Heap, an array H[1..n] is used for storing the elements of the Heap. Once the Heap is built, the elements of A[1..n] are already sorted in H[1..n] and hence A may be used for sorting the elements of finally sorted list for which we used the array B. Then the following three steps are repeated n times, (n, the number of elements in the array); in the ith iteration.

(i) The root element H[1] is copied in A[n − i+1] location of the given array A. The first time, root element is stored in A[n]. The next time, root element is stored in A[n −1] and so on.

(ii) The last element of the array H[n − i + 1] is copied in the root of the Heap, i.e., in H[1] and H[n − i + 1] is removed from further consideration. In other words, in the next iteration, only the array H[1..(n − i)] (*which may not be a Heap*) is taken into consideration.

(iii) The procedure *Build-Heap* is called to build the array H[1..(n − i)] into a Heap.

Procedure Heap-Sort (A [1 .. n]: real)
begin {*of the procedure*}
H[1..n] ← Build-Heap (A[1..n])
For i ← 1 to n do

```
      begin {for loop}
       A [n – i + 1] ← H[1]
           H[1] ← H[n – i + 1]
       Build-Heap (H [1.. (n – i)])
      end {for-loop}
end {procedure}
```

Procedure Build-Heap

The following procedure takes an array B[1..m] of size m, which is to be sorted, and builds it into a Heap

```
Procedure Build-Heap (B[1..m]: real)
 begin {of procedure}
  for j = 2 to m do
    begin {of for-loop}
      location ← j
    while (location > 1)  do
     begin
         Parent ← [location/2]
          If A[location] ≤ A [parent] then return {i.e., quit while loop}
          else
        {i.e., of A[location] > A [parent] then}
      begin {to exchange A [parent] and A [location]}
       temp ← A [parent]
        A[parent] ← A [location]
        A [location] ← temp
      end {of if .. then .. else}
          location ← parent
   end {while loop}
    end {for loop}
 end {of procedure}
```

Ex. 5) Sort the following sequence of number, using Heap Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

3.3.6 Divide and Conquer Technique

The 'Divide and Conquer' is a technique of solving problems from various domains and will be discussed in details later on. Here, we briefly discuss how to use the technique in solving sorting problems.

A sorting algorithm based on '*Divide and Conquer*' technique has the following outline:

```
    Procedure  Sort (list)
        If the list has length 1 then return the list
        Else {i.e., when length of the list is greather than 1}
     begin
                Partition the list into two sublists say L and H,
                Sort (L)
                Sort (H)
                Combine (Sort (L)), Sort (H))
{during the combine operation, the sublists are merged in sorted order}
    end.
```

There are two well-known 'Divide and conquer' methods for sorting viz:

(i) Merge Sort
(ii) Quick Sort

3.3.7 Merge Sort

In this method, we recursively chop the list into two sublists of **almost equal** sizes and when we get lists of sizes one, then start sorted merging of lists in the reverse order in which these lists were obtained through chopping. The following example clarifies the method.

Example 3.3.7.1 of Merge Sort:

Given List: 4 6 7 5 2 1 3
Chop the list to get two sublists viz.
((4, 6, 7, 5), (2 1 3))
where the symbol '/ ' separates the two sublists

Again chop each of the sublists to get two sublists for each viz
(((4, 6), (7, 5))), ((2), (1, 3)))
Again repeating the chopping operation on each of the lists of size two or more obtained in the previous round of chopping, we get lists of size 1 each viz 4 and 6, 7 and 5, 2, 1 and 3. In terms of our notations, we get
((((4), (6)), ((7), (5))), ((2), ((1), (3))))
At this stage, we start merging the sublists in the reverse order in which chopping was applied. *However, during merging the lists are sorted.*

Start merging after sorting, we get sorted lists of at most two elements viz
(((4, 6), (5, 7)), ((2), (1, 3)))
Merge two consecutive lists, each of at most two elements we get sorted lists
((4, 5, 6, 7), (1, 2, 3))
Finally merge two consecutive lists of at most 4 elements, we get the sorted list: (1, 2, 3, 4, 5, 6, 7)

Procedure ***mergesort*** **(A [1..n])**
if **n > 1** then
m ← [n/2]
L_1← A[1..m]
L_2← A[m+1..n]
L← merge (mergesort (L_1), mergesort (L_2))
end

begin {*of the procedure*}
{L is now sorted with elements in nondecreasing order}
Next, we discuss merging of already sorted sublists
Procedure ***merge*** **(L_1, L_2: lists)**
L ← empty list
While **L_1 and L_2 are both nonempty do**
begin
Remove smaller of the first elements of L_1 and L_2 from the list it is in and place it in L, immediately next to the right of the earlier elements in L. If **removal of this element makes one list empty** then **remove all elements from the other list and append them to L keeping the relative order of the elements intact.**
else **repeat the process with the new lists L_1 and L_2**
end
{L is the merged list with elements sorted in increasing order}

Ex. 6) Sort the following sequence of number, using Merge Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

3.3.8 Quick Sort

Quick Sort is also a '*divide and conquer*' method of sorting. It was designed by C.A.R. Hoare, one of the pioneers of Computer Science and also Turing Award Winner for the year 1980. This method does **more work** in the **first step of partitioning the list** into two sublists. Then combining the two lists becomes trivial.

To partition the list, we first choose **some** value from the list for which, we hope, about half the values will be less than the chosen value and the remaining values will be more than the chosen value.

Division into sublists is done through the choice and use of a **pivot value,** which is a value in the given list so that all values in the list less than the pivot are put in one list and rest of the values in the other list. The process is applied recursively to the sublists till we get sublists of lengths one.

Remark 3.3.8.1:

The choice of pivot has significant bearing on the efficiency of Quick-Sort algorithm. Sometime, the very first value is taken as a pivot.
However, the **first values** of given lists may be poor choice, specially when the given list is already ordered or nearly ordered. Because, then one of the sublists may be empty. For example, for the list
7 6 4 3 2 1
the choice of first value as pivots, yields the list of values greater than 7, the pivot, as empty.

Generally, some **middle value** is chosen as a pivot.

Even, choice of middle value as pivot may turn out to be very poor choice, e.g, for the list
2 4 6 **7** 3 1 5
the choice of middle value, viz 7, is not good, because, 7 is the maximum value of the list. Hence, with this choice, one of the sublists will be empty.

A *better method for the choice of pivot position* is to use a random number generator to generate a number j between 1 and n, for the position of the pivot, where n is the size of the list to be sorted. Some simpler methods take median value of the of a sample of values between 1 to n, as pivot position. For example, the median of the values at first, last and middle (or one of the middle values, if n is even) may be taken as pivot.

Example 3.3.8.1 of Quick Sort

We use two indices (*viz i & j, this example*) one moving from left to right and other moving from right to left , **using the first element as pivot**
In each iteration, the index i while moving from left to the right, notes the position of the value first from the left that is greater than pivot and stops movement for the iteration. Similarly, in each iteration, the index j while moving from right to left notes the position of the value first from the right that is less than the pivot and stops movement for the iteration. The values at positions i and j are exchanged. Then, the next iteration starts with current values of i and j onwards.

5 3 1 **9** 8 2 **4** 7 {← *Given list*}
i → ← j

5 3 1 **9** 8 2 **4** 7
i j

{*the value 9 is the first value from left that is greater than the pivot viz 5 and the value 4 is the first value from right that is less than the pivot. These values are exchanged to get the following list*}

5 3 1 4 **8** **2** 9 7
i j

{*Moving i toward right and j toward left and i stops, if it reaches a value greater than the pivot and j stops if j meets a value less than the pivot. Also both stop if j ≤ i. We get the list*}

5 3 1 4 8 2 9 7
i ↔ j

{*After exchanging the values 2 and 8 we get the list*}

5 3 1 4 **2** 8 9 7
i j

{*The next moves of i to the right and j to the left make j < i and this indicates the completion of one iteration of movements of i and j to get the list with positions of i and j as follows*}

5 3 1 4 2 8 9 7
j i

{*At this stage, we exchange the pivot with value at position j, i.e., 2 and 5 are exchanged so that pivot occupies almost the middle position as shown below.*}

2 3 1 4 **5** 8 9 7

{It may be noted that all values to the left of 5 are less than 5 and all values to the right of 5 are greater than 5. T*hen the two sublists viz 2,3,1,4 and 8,9,7 are sorted independently*}

2 **3** **1** 4 and 8 7 9
i → ← j i → ← j

2 3 1 4 and 8 7 9
i ↔ j i j

2 1 3 4 and 8 7 9
i j j i

{as j < i, e*xchanging the pivot values with 7, value at jth position, we get*}

{*moving i and j further we get*}

2 **1** 3 4 and 7 8 9
j i i j

{*as j < i, exchange pivot 2 with value 1 at j, we get neglecting the pivot the two sublists to be sorted as 1 and 3, 4*}

1 **2** **3** 4
and i j,
3 4
j i

{*Pivot position = 1 = j. And pivot position is neglected therefore, we need to sort the remaining lists which are respectively empty and {4}, which are already sorted. This completes the sorting. Merging does not require any extra time as already the entries are in sorted order*}

On basis of the discussion above, we now formalize the Quick Sort algorithm/procedure

```
Procedure      Quick-Sort (A[min… max])
  {min is the lower index and max the upper index of the array to be sorted using Quick Sort}
      begin
        if  min < max then
              p ← partition (A[min..max]);
                 {p is the position s.t for  min ≤ i ≤ p − 1,      A [i] ≤ A [p] and  for all j ≥ p+1,
                  A[j] ≥ A [p]}
               Quick-Sort (A [min .. p − 1]);
               Quick-Sort (A[p+1 .. max]);
       end;
```

In the above procedure, the procedure *partition* is called. Next, we define the procedure partition.

```
Procedure      partition (A [min .. max])
i, j: integer;
s: real;
   {In this procedure the first element is taken as pivot; the index i, used below moves from left to right
    to find first value i = 1i  from left such that A[v1 ] > A[1].  Similarly j moves from right to left  and
    finds first value j = v2  such that A[v2] < A[1]. If j >i, then A[i] and A[j] are exchanged.  If j ≤ I
    then A[1] and A[j] are exchanged}.
    s ← A [min]; i ← min + 1; j ← max
while (i < j)   do

      begin
        While (A[i] < p)       do
              i ← i +1
         While (A[j] > p)      do
              j ← j − 1
         exchange (A[i], A[j])
           {the exchange operation involves three assignments, viz, temp ←A [i ];  A[i] ← A [j] and
           A[j] ← temp, where
             temp is a new variable }
      end;   {of while loop}
      exchange  (A[1], A[j]);
      return j
          {the index j is s.t in the next iteration sorting is to be done for  the two subarray
           A[ min .. j–1] and  A[j+1 .. max]
```

Ex. 7) Sort the following sequence of number, using Quick Sort:
15, 10, 13, 9, 12 7
Further, find the number of comparisons and copy/assignment operations required by the algorithm in sorting the list.

3.3.9 Comparison of Sorting Algorithms

We give below *just the summary* of the time complexities of the various algorithms discussed above. Some of these time complexities will be derived at different places, after sufficient tools are developed for the purpose. Here these are included so that we may have an idea of the comparative behaviors of the algorithms discussed above. Each algorithm requires time for execution of two different types of actions, viz

(i) Comparison of keys
(ii) assignment of values

The following table gives the time complexities requirement in terms of size n of the list to be sorted, for the two types of actions for executing the algorithms. Unless mentioned otherwise, the complexities are for *average case behaviors* of the algorithms:

Name of the algorithm	Comparison of Keys	Assignments
Selection Sort	$0.5n^2 + O(n)$	$3.0\ n + O(1)$
Insertion Sort	$0.25n^2 + O(n)$ (*average)*	$0.25n^2 + O(n)$
Shell Sort	$n^{1.25}$ to $1.6n^{1.25}$ for large n (*empirically*)	
Heap Sort	$2\ n \log n + O(n)$ (*worst case*)	$n \log n + O(n)$
Bubble Sort	$\frac{1}{2}(n^2 - n \log n)$ (*average*)	$\frac{1}{4}(n^2 - n)$ (*average*)
	$\frac{1}{2}(n^2 - n)$ (*worst*)	$\frac{1}{2}(n^2 - n)$ (*worst*)
Quick Sort	$1.39\ n \log n$	$0.69\ n \log n$
Merge Sort	$n \log n$ to $(n \log n - 1.583n + 1)$ (*for linked lists*)	$n \log n$ (*for contiguous lists*)

Merge Sort is good for linked lists but poor for contiguous lists. On the other hand, Quick Sort is good for contiguous lists but is poor for linked lists.
In context of complexities of a sorting algorithm, we state below an important theorem without proof.

Theorem: Any algorithm, *based on comparison of keys*, that sorts a list of n elements, in its average case must perform at least $\log (n!) \approx (n \log n + O(n))$ number of comparisons of keys.

3.4 BEST-CASE AND WORST CASE ANALYSES

For certain algorithms, the times taken by *an algorithm on two different instances*, may differ considerably. For this purpose, consider the algorithm *Insertion-Sort, already discussed, for sorting n given numbers, in* ***increasing order***.

Again, for this purpose, one of the two lists is TS[1..n] in which the *elements are already sorted* in *increasing* order, and the other is **TR[1..n]** in which *elements are in reverse-sorted order*, i.e., elements in the list are in *decreasing order*.

The algorithm *Insertion-Sort* for the array TS[1..n] already sorted in increasing order, does not make any displacement of entries (i.e., values) for making room for out-of-order entries, because, there are no out-of-order entries in the array.

In order to find the complexity of Insertion Sort for the array TS[1..n], let us consider a specific case of sorting.

The already sorted list {1,3,5,7,9}. Initially A[1] = 1 remains at its place. For the next element A[2] = 3, at its place. For the next element A[2] = 3, the following operations are performed by Insertion Sort:

(i) $j \leftarrow 2$
(ii) $i \leftarrow j - 1$
(iii) $m \leftarrow A[j]$
(iv) $A[i + 1] \leftarrow m$
(v) $m < A[i]$ in the while loop.

We can see that these are the minimum operation performed by Insertion Sort irrective of the value of A[2]. Further, had A[2] been less than A[1], then more operations would have been performed, as we shall see in the next example.

To conclude, as if A[2] ≥ A[1] (as is the case, because A[2] = 3 and A[1] = 1) then Insertion Sort performs 4 assignments and 1 comparison. We can see that in general, if A[l+1] ≥ A[l] then we require (exactly) 4 additional assignments and 1 comparison to place the value A[l+1] in its correct position, viz (l+1)th.

Thus in order to use Insertion Sort to attempt to sort an already, in proper order, sorted list of n elements, we need 4(n─1) assignments and (n ─ 1) comparison.

Further, we notice, that these are the minimum operations required to sort a list of n elements by Insertion Sort.

Hence in the case of *array TS*[*1..n*], the Insertion-Sort algorithm *takes linear time* for an already sorted array of n elements. **In other words, Insertion-Sort has linear *Best-Case* complexity for TS[1..n].**

Next, we discuss the number of operations required by Insertion Sort for sorting TR[1..n] which is sorted in *reverse order*. For this purpose, let us consider the sorting of the list {9, 7, 5, 3, 1} stored as A[1] = 9, A[2] = 7, A[3] = 5 A[4] = 3 and A[5] = 1. Let m denote the variable in which the value to be compared with other values of the array, is stored. As discussed above in the case of already properly sorted list, 4 assignments and one comparison, of comparing A[k +1] with A[k], is *essentially* required to start the process of putting each new entry A[k+1] after having already sorted the list A[1] to A[k]. However, as 7 = A[2] = m < A[1] = 9. A[1] is copied to A[2] and m is copied to A[1]. Thus Insertion Sort requires one more assignment (viz A[2] ← A[1]).
At this stage A[1] = 7 and A[2] = 9. *It is easily seen that for a list of two elements, to be sorted in proper order using Insertion-Sort, at most one comparison and 5 (= 4 +1) assignments are required.*

Next, m ← A[3] = 5, and m is first compared with A[2] = 9 and as m < A[2], A[2] is copied to A[3]. So, at this stage both A[2] and A[3] equal 9.

Next, m = 5 is compared with A[1] = 7 and as m < A[1] therefore A[1] is copied to A[2], so that, at this stage, both A[1] and A[2] contain 7. Next 5 in m is copied to A[1]. Thus at this stage A[1] = 5, A[2] = 7 and A[3] = 9. And, during the last round, we made two additional comparisons and two addition assignments (viz A[3] ← A[2] ← A[1]), and hence total (4 + 2) assignments, were made.

Thus, so far we have made 1+ 2 comparisons and 5 + 6 assignments.
Continuing like this, for placing A[4] = 3 at right place in the sorted list {5,7,9}, we make 3 comparisons 7 assignments. And for placing A[5] = 1 at the right place in the sorted list {3,5,7,9}, we make 4 comparisons and 8 assignments. Thus, in the case of a list of 5 elements in reverse order, the algorithm takes 1+2+3+4 = $\frac{4 \times 5}{2}$ comparisons and 5+6+7+8+9 = 4 × 5 + (1+2+3+4+5) assignments.

In general, for a list of n elements, *sorted in reverse order,* the Insertion Sort algorithm makes $\frac{(n-1)n}{2} = \frac{1}{2}$ (n^2 ─ n) comparisons and 4n + $\frac{1}{2}$ (n^2 + n) = $\frac{1}{2}$ (n^2 + 9n) assignments.

Again, it can be easily seen that for a list of n elements, Insertion-Sort algorithm should make *at most* $\frac{1}{2}$ ($n^2 - n$) comparisons and $\frac{1}{2}$ ($n^2 + 9n$) assignments.

Thus, the total number of operations

$$=\left(\frac{1}{2}\left(n^2-n\right)\right)+\left(\frac{1}{2}\left(n^2+n\right)+4n\right)=n^2+4n$$

If we assume that time required for a comparison takes a constant multiple of the time than that taken by an assignment, then time-complexity of Insertion-Sort *in the case of* ***reverse-sorted*** *list* is a **quadratic in n.**

When actually implemented for n = 5000, the Insertion-Sort algorithm took 1000 times more time for TR[1..n], the array which is sorted in reverse order than for TS[1..n], the array which is already sorted in the required order, to sort both the arrays in increasing order.

3.4.1 Various Analyses of Algorithms

Earlier, we talked of time complexity of an algorithm as a function of the size n of instances of the problem intended to be solved by the algorithm. However, from the discussion above, we have seen ***that for a given algorithm and instance size n,*** the execution times for two different instances of the same size n, may differ by a factor of 1000 or even more. This necessitates for us to consider *further differentiation or classification of complexity analyses* of an algorithm, which take into consideration not only the size n but also types of instances. Some of the well-known ones are:

(i) worst case analysis
(ii) best case analysis
(iii) average case analysis
(iv) amortized analysis

We discuss worst-case analysis and best-case analysis in this section and the other two analyses will be discussed in Section 2.9.

3.4.2 Worst-Case Analysis

Worst-Case analysis of an algorithm for a given problem, involves finding the **longest** of all the times that can (*theoretically*) be taken by various instances of a given size, say n, of the problem. The worst-case analysis may be carried out by first (i) finding the instance types for which algorithm runs slowest and then (ii) finding running time of the algorithm for such instances. If **c-worst (n)** denotes the worst-case complexity for instances of size n, then by the definition, it is guaranteed that no instance of size n of the problem, shall take more time then c-worst (n).
Worst-Case analysis of algorithms is appropriate for problems in which response-time is critical. For example, in the case of problems in respect of controlling of nuclear power plants, it is important to know an upper limit on the system's response time than to know the time of execution of particular instances.

In the case of the Insertion-Sort algorithm discussed earlier, we gave an outline of argument in support of the fact that Insertion-Sort algorithm takes *longest time*

(i) for lists sorted in reverse order and

(ii) if the size of such a list is n, then the longest time should correspond to $\frac{1}{2}\left(n^2+n\right)$ comparisons and $\frac{1}{2}\left(n^2+3n\right)$ assignments.

In other words, the worst time complexity of Insertion-Sort algorithm is a quadratic polynomial in the size of the problem instance.

3.4.3 Best-Case Analysis

Best- Case Analysis of an algorithm for a given problem, involves finding the **shortest** of all the times that can (*theoretically*) be taken by the various instances of a given size, say n, of the problem. In other words, the best-case analysis is concerned with

(i) finding the instances (types) for which algorithm runs **fastest** and then
(ii) with finding running time of the algorithm for such instances (types).

If ***C-best* (*n*) denotes** the best-case complexity for instances of size n, then by definition, it is guaranteed that *no instance* of size n, of the problem, shall take *less time than*
C-best (*n*).

The best-case analysis is not as important as that of the worst-case analysis. However, the best-case analysis may be useful guide for application to situations, which need not necessarily correspond to the instance types taking shortest time, yet which are close to such instance types. For example, a telephone directory of a metropolitan city like Delhi, contains millions of entries already properly sorted. Each month, if a few thousand of new entries are to made, then if these entries are put in the beginning of the directory, then in this form, without further processing, the directory is a *nearly sorted* list. And, the Insertion-Sort algorithm which makes only n comparisons and no shifts for an already properly sorted list of n elements sorted in the required order, may be useful for application to this *slightly out-of-order* new list obtained after addition of a few thousand entries in the beginning of the earlier sorted list of millions of entries.

In general, it can be shown that

(i) for an already sorted list of w elements, sorted in the required order, the

Insertion-Sort algorithm will make $(n - 1)$ comparisons and $4 \times (n - 1)$ assignments and

(ii) $(n - 1)$ comparisons and $4 (n - 1)$ assignments are the minimum that are required of sorting any list of n elements, that is already sorted in the required order.

Thus the best time complexity of Insertion Sort algorithm is a linear polynomial in the size of the problem instance.

3.5 ANALYSIS OF NON-RECURSIVE CONTROL STRUCTURES

Analysis of algorithms is generally a *bottom-up process* in which, first we consider the time required for *executing individual instructions*. Next, we determine recursively time required by more complex structures where times required for less complex structures are already calculated, already known, given or assumed. We discuss below how the time required for structures obtained by applying *basic structuring rules* for sequencing, repetition and recursion, are obtained, provided times required for individual instructions **or component program fragments**, are already known or given. First we consider the structuring rule: sequencing.

3.5.1 Sequencing

Let F_1 and F_2 be two program fragments, with t_1 and t_2 respectively the time required for executing F_1 and F_2. Let the program fragment $F_1 ; F_2$ be obtained by sequencing the two given program fragments, i.e, *by writing* F_1 *followed by* F_2.

Then **sequencing rule** states that the time required for the program fragment $F_1 ; F_2$ *is* $t_1 + t_2$.

Word of Caution: *The sequencing rule, mentioned above, is valid only under the assumption that no instruction in fragment F_2 depends on any instruction in Fragment F_1. Otherwise, instead of $t_1 + t_2$, the time required for executing the fragment $F_1 ; F_2$ may be some more complex function of t_1 and t_2, depending upon the type of dependency of instruction(s) of F_2 on instructions of F_1. Next, we consider the various iterative or looping structures, starting with "For" loops.*

3.5.2 *For* Construct

In order to understand better the ideas involved, let us first consider the following two simple examples involving *for* construct.

Example 3.5.2.1: The following program fragment may be used for computing sum of
first n natural numbers:

```
for i = 1 to n do
    sum = sum + i.
```

The example above shows that the instruction *sum = sum + i* depends upon the loop variable 'i'. Thus, if we write

P(i) : sum = sum + i

then the above-mentioned 'for' loop may be rewritten as

```
for i= 1 to n do
  P (i),
end {for}
```

where i in P(i) indicates that the program fragment P depends on the loop variable i.

Example 3.5.2.2: The following program fragment may be used to find the sum of n numbers, each of which is to be supplied by the user:

```
for i = 1 to n do
    read (x);
   sum = sum + x;
end {for}.
```

In the latter example, the program fragment P, consisting of two instructions viz., *read* (*x*) and *sum = sum + x*, do not involve the loop variable i. But still, there is nothing wrong if we write P as P(i). This is in view of the fact that a function f of a variable x, given by

$f(x) = x^2$

may also be considered as a function of the two variables x and y, because

$f(x,y) = x^2 + 0.y$

Remark 3.5.2.3:

The *for* loop
for i = 1 to n do
P(i) ;
end for,
is actually a shorthand for the following program fragment

i = 1
while i ≤ n do
P(i) ;
i =i + 1
end while ;

Therefore, we need to take into consideration the above-mentioned fact while calculating the time required by the *for loop* fragment.

Remark 3.5.24:

The case when n = 0 in the loop *for i = 1 to n do P* (*i*) would not be treated as an error. The case *n = 0* shall be interpreted that P(i) *is not executed even once*.

Let us now calculate the time required for executing the loop

for i = 1 to n do
P(i).
end for

For this purpose, we use the expanded definition considered under Remark and, in addition, we use the following notations:

fl : the time required by the 'for' loop

a : the time required by each of the assignments
i= 1 and i = i+ 1;

c : for each of the test i ≤ n and
s: for sequencing the two instructions P(i) and i = i+1 in the While loop.

t : the time required for executing P(i) once.

Then, it can be easily seen that

fl	=	a	*for i = 1*
		+ (n+1) c	*for (n+1) times testing i ≤ n*
		+ n t	*for n times execution of P(i)*
		+ n a	*for n times execution of i = i+1*
		+ n s	*for n times sequencing P(i) and i = i + 1.*

i.e.

fl = (n+1)a + (n+1)c + n s + n t

In respect of the last equality, we make the following observations

(i) the quantity on R.H.S is bounded below by nt, n times the time of execution of P(i)

(ii) if t, the time of execution of P(i) is much larger than each of

(a) a, the time for making an assignment
(b) c, the time for making a comparison and
(c) s, the time for sequencing two instructions, one after the other,
then fl, the time to be taken by the 'for' loop is approximately nt, i.e., fl ≈ nt

(iii) If n = 0 or n is negative, the approximation fl ≈ nt is completely wrong. Because , w.r.t Remark 2.6.2.1 above, at least the assignment i = 1 is executed and also the test i ≤ n is executed at least once and hence fl ≤ 0 or fl ≈ 0 can not be true.

3.5.3 *While* and *Repeat* Constructs

From the point of view of time complexity, the analysis of *while* or *repeat* loops is more difficult as compared to that of *for* loops. This is in view of the fact that in the case of *while* or *repeat* loops, we do not know, in advance, how many times the loop is going to be executed. But, in the case of *for* loops, we can know easily, in advance the number of times, the loop is going to be executed.

One of the frequently used techniques for analysing while/repeat loops, is *to define a function* say f of the involved loop variables, in such a way that

(i) the value of the *function f* is an integer that decreases in successive iterations of the loop.

(ii) the *value of f remains non-negative* throughout the successive executions of the loop, and as a consequence.

(iii) the value of f reaches some minimum non-negative value, when the loop is to terminate, of course, only if the loop under consideration is a terminating loop.

Once, such a *function* f, if it exists, is found, the analysis of the *while/repeat* loop gets simplified and can be accomplished just by close examination of the sequence of successive values of f.

We illustrate the techniques for computing the time complexity of a ***while loop*** through an example given below. The ***repeat loop*** analysis can be handled on the similar lines.

Example 3.5.3.1:

Let us analyze the following *Bin-Search algorithm* that finds the location of a *value v* in an already sorted *array A[1..n*], where it is given that v occurs in A[1.. n]
*The Bin-Search algorithm, as defined below, is, in its rough version, intuitively applied by us in finding the meaning of a given word in a dictionary or in finding out the telephone number of a person from the telephone directory, where the name of the person is given. In the case of dictionary search, if the word to be searched is say **CARTOON**, then in view of the fact that the word starts with letter **C** which is near the beginning of the sequence of the letters in the alphabet set, we open the pages in the dictionary near the beginning. However, if we are looking for the meaning of the word REDUNDANT, than as R is 18th letter out of 26 letters of English alphabet, we generally open to pages after the middle half of the dictionary.*

However, in the case of search of a known value v in a given sorted array, the values of array are not know to us. Hence we do not know the relative position of v. This is why, we find the value at the middle position $\left\lfloor \frac{1+n}{2} \right\rfloor$ in the given sorted array A[1..n] and then compare v with the value A $\left\lfloor \frac{1+n}{2} \right\rfloor$. These cases arise:

(i) If the value v = A $\left\lfloor \frac{n+1}{2} \right\rfloor$, then search is successful and stop. Else,

(ii) if $v < A\left[\frac{1+n}{2}\right]$, then we search only the part $A\left[1..\left(\left[\frac{1+n}{2}\right]-1\right)\right]$ of the given array. Similarly

(iii) if $v > A\left[\frac{1+n}{2}\right]$, than we search only the part $A\left[\left(\frac{1+n}{2}\right)+1..n\right]$ of the array.

And repeat the process. The explanation for searching v in a sorted array, is formalized below as function Bin-Search.

Function **Bin-Search (A[1..n], v)**
begin
i = 1 ; j = n
while **i < j** ***do***
{*i.e.,* $A[i] \le v \le A[j]$ }

k = [(i+ j) ÷ 2]

Case v < A[k]: j = k–1
v = A[k]: { return k }
v > A[k]: i = k + 1
end case
end while { return i }
end function;

We explain, through an example, how the Bin-Search defined above works.

Let the given sorted array A[1..12] be

1	4	7	9	11	15	18	21	23	24	27	30

and let v = 11. Then i = 1, j = 12 and

$$k = \left[\frac{1=12}{2}\right] = \left[\frac{13}{2}\right] = 6$$

and A[6] = 15
As v = 11 < 15 = A[6]

Therefore, for the next iteration

j = 6 – 1 = 5
i = 1 (unchanged)
hence $k = \left[\frac{1+5}{2}\right] = 3$
A [3] = 7
As v = 11 > 7 = A[3]

For next iteration
∴ i becomes (k+1) = 4, j remains unchanged at 5.

Therefore new value of $k = \left[\frac{4+5}{2}\right] = 4$

As v = 11 > 9 = A[k]

Therefore, in new iteration i becomes k + 1 = 5, j remains unchanged at 5.
Therefore new

$$k = \left\lfloor \frac{5+5}{2} \right\rfloor = 5$$

And A[k] = A[5] = 11 = v

Hence, the search is complete.

In order to analyse the above algorithm Bin-Search, let us define

Before analyzing the algorithm formally, let us consider tentative complexity of algorithm informally. From the algorithm it is clear that if in one iteration we are considering the array A [i...j] having j – i +1 elements then next time we consider either A [i .. k – 1] or A[(k + 1)..j] each of which is of length less than half of the length of the earlier list. Thus, at each stage length is getting at least halved. Thus, we expect the whole process to be completed in $\log 2^n >$ iterations.

We have seen through the above illustration and also from the definition that the array which need to be searched in any iteration is A[i..j] which has (j – i + 1) number of elements

Let us take f = j – i + 1 = length of the sequence currently being searched.
Initially f = n – 1+1 = n. Then, it can be easily seen that f satisfies the conditions (i), (ii) and (iii) mentioned above.

Also if f_{old}, j_{old} and i_{old} are the values of respectively f, j and i before an iteration of the loop and f_{new} , j_{new} and i_{new} the new values immediately after the iteration, then for each of the three cases v < A [k], v = A [k] and v > A [k] we can show that

$$f_{new} \leq f_{old}/2$$

Just for explanations, let us consider the case v < A [k] as follows:
for v < A [k], the instruction j = k – 1 is executed and hence

$i_{new} = i_{old}$ and $j_{new} = [(i_{old} + j_{old}) \div 2] - 1$ and hence

$$f_{new} = j_{new} - i_{new} + 1 = [(i_{old} + j_{old}) \div 2 - 1] - i_{old} + 1$$

$$\leq (j_{old} + i_{old})/2 - i_{old}$$
$$< (j_{old} - i_{old} + 1)/2 = f_{old}/2$$

$\therefore f_{new} < fold/2$
In other words, after each execution of the *while* loop, the length of the sub array of A [1.. n], that needs to be searched, if required, is less than half of the previous subarray to be searched. As v is assumed to be one the values of A [1 .. n], therefore, in the worst case, the search comes to end when j = i, i.e., when length of the subarray to be searched is 1. Combining with $f_{new} < f_{old}/2$ after each interaction, in the worst case, there will be t iterations satisfying

$$1 = (n/2^t) \quad \text{or} \quad n = 2^t$$

i.e., $t = \lfloor \log_2 n \rfloor$

Analysis of "repeat" construct can be carried out on the similar lines.

3.6 RECURSIVE CONSTRUCTS

We explain the process of analyzing time requirement by a recursive construct/ algorithm through the following example.

Example 3.6.1:

```
Function factorial (n)
        {computes n!, the factorial of n recursively
         where n is a non-negative integer}
begin
    if  n = 0 return 1
    else return (n * factorial (n– 1))
end factorial
```

Analysis of the above recursive algorithm

We take n, the input, as the *size of an instance* of the problem of computing factorial of n. From the above (recursive) algorithm, it is clear that multiplication is its basic operation. **Let M(n) denote the number of multiplications** required by the algorithm in computing factorial (n). The algorithm uses the formula

Factorial (n) = n* factorial (n–1) for n > 0.

Therefore, the algorithm uses one extra multiplication for computing factorial (n) then for computing factorial (n–1). Therefore,

$$M(n) = M(n-1) + 1. \quad \text{for } n > 0 \qquad (3.6.1)$$

Also, for computing factorial (0), no multiplication is required, as, we are given factorial (0) = 1. Hence

$$M(0) = 0 \qquad (3.6.2)$$

The Equation (3.6.1) does *not* define M(n) *explicitly* but defines *implicitly* through M(n–1). Such an equation is called a **recurrence relation**/equation. The equation (3.6.2) is called an **initial condition**. The equations (3.6.1) and (3.6.2) together form a **system of recurrences**.

By solution of a system of recurrences, we mean an explicit formula, say for M(n) in this case, free from recurrences in terms of n only, and not involving the function to be defined, i.e., M in this case, directly or indirectly.

We shall discuss in the next section, in some detail, how to solve system of recurrences. Briefly, we illustrate a method of solving such systems, called **Method of Backward Substitution**, through solving the above system of recurrences viz.

$$M(n) = M(n-1) + 1 \qquad (3.6.1)$$

and

$$M(0) = 0 \qquad (3.6.2)$$

Replacing n by (n – 1) in (2.7.1), we get $M(n-1) = M(n-2) + 1$ (3.6.3)
Using (2.7.3) in (2.7.1) we get

$$M(n) = M(n-1) + 1$$
$$= [M(n-2) + 1] + 1,$$

Repeating the above process by replacing n in (2.7.1) successively by (n – 2), (n – 3),……, 1, we get

$M(n) = M(n-1) + 1$	$= M(n-1) + 1$
$= [M(n-2) + 1] + 1$	$= M(n-2) + 2$
$= [M(n-3) + 1] + 2$	$= M(n-3) + 3$
$= M(1) + (n-1)$	$= M(n-I) + i$

= M (0) + n , Using (2.7.2) we get

M(n) = 0 + n = n

3.7 SOLVING RECURRENCES

In the previous section, we illustrated, with an example, the method of backward substitution for solving system of recurrences. We will discuss, in the next section, some more examples of the method.
In this section, we discuss some methods of solving systems of recurrences. However, the use of the methods discussed here, will be explained in the next section though appropriate examples.

3.7.1 Method of Forward Substitution

We explain the method through the following example.

Example 3.7.1.1:

Let us consider the system of recurrences

$$F(n) = 2F(n-1) + 1 \quad \text{for} \quad n > 1 \qquad (3.7.1)$$
$$F(1) = 1 \qquad (3.7.2)$$

First few terms of the sequence < F (n) > are, as given below.

$$f(1) = 1$$
$$f(2) = 2 * f(1) + 1 = 2 \times 1 + 1 = 3$$
$$f(3) = 2 * f(2) + 1 = 2 \times 3 + 1 = 7$$
$$f(4) = f(3) + 1 = 2 \times 7 + 1 = 15$$

Then, it can be easily seen that

$$F(n) = 2^n - 1 \quad \text{for } n = 1, 2, 3, 4$$

We (intuitively) feel that

$$F(n) = 2^n - 1 \quad \text{for all } n \geq 1$$

We attempt to establish the correctness of this intuition /feeling through Principle of Mathematical Induction as discussed.
As we mentioned there, it is a Three-Step method as discussed below:

***Step* (*i*):** We show for n = 1

$$F(1) = 2^1 - 1 = 1,$$

But F(1) = 1 is given to be true by definition of F given above.

***Step* (*ii*):** Assume for any k > 1

$$F(k) = 2^k - 1$$

***Step* (*iii*):** Show

$$F(k+1) = 2^{k+1} - 1.$$

For showing

$$F(k+1) = 2^{k+1} - 1,$$

Consider, by definition,

$$\begin{aligned} F(k+1) &= 2\,F\,(K + 1 - 1) + 1 \\ &= 2\,F\,(k) + 1 \\ &= 2\,(2^k - 1) + 1 \text{ (by Step (ii))} \\ &= 2^{k+1} - 2 + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

Therefore, by Principle of Mathematical Induction, our feeling that $F(n) = 2^n - 1$ for all $n \geq 1$ is mathematically correct.

Next, we discuss methods which can be useful in many practical problems.

3.7.2 Solving Linear Second-Order Recurrences with Constant Coefficients

Definitions: Recurrences of the form

$$a\,F(n) + b\,F(n-1) + cF(n-2) = g(n) \qquad (3.7.2.1)$$

where a, b, c are real numbers, $a \neq 0$, are called linear second-order recurrences with constant coefficients. Further, if $g(n) = 0$ then the recurrence is called **Homogeneous**. Otherwise, it is called **inhomogeneous**. Such systems of recurrences can be solved by neither **backward** substitution method nor by **forward** substitution method.

In order to solve recurrences of the form (2.8.2.1).

First we consider only the homogeneous case, i.e., when $g(n) = 0$. The recurrence becomes

$$a\,F(n) + b\,F(n-1) + c\,F(n-2) = 0 \qquad (3.7.2.2)$$

The above equation has infinitely many solutions except in the case when both $b = 0$ and $c = 0$.

With equation (2.8.2.2) we associate an auxiliary quadratic equation, called **characteristic equation**, given by

$$ax^2 + bx + c = 0 \qquad (3.7.2.3)$$

Then the solutions of the recurrences (2.8.2.1) are given by the following theorem, which we state without proof.

Theorem 3.7.2.1:

Let x_1 and x_2 be the solutions of the auxiliary quadratic equation

$a\,x^2 + bx + c = 0$. Then

Case 1: If x_1 and x_2 are real and distinct then solution of (2.8.2.2) is given by

$$F(n) = \alpha\,x_1^n + \beta\,x_2^n \qquad (3.7.2.4)$$

Where α and β are two arbitrary real constants.

Case II: If the roots x_1 and x_2 are real but $x_1 = x_2$ then solutions of (3.7.2.2) are given by

$$F(n) = \alpha x_1^n + \beta n x_1^n, \qquad (3.7.2.5)$$

Where, again, α and β are arbitrary real constants.

Case III: If x_1 and x_2 are complex conjugates given by u + iv, where u and v are real numbers. Then solutions of (2.8.2.2) are given by

$$F(n) = r^n [\alpha \cos n\theta + \beta \sin n\theta] \qquad (3.7.2.6)$$

where $r = \sqrt{u^2 + v^2}$ and $\theta = \tan^{-1}(v/u)$ and α and β are two arbitrary real constants.

Example 3.7.2.2:

Let the given recurrence be

$$F(n) - 4F(n-1) + 4F(n-2) = 0. \qquad (3.7.2.7)$$

Then, its characteristic equation is given by

$$x^2 - 4x + 4 = 0,$$

The two solutions equal, given by x = 2.

Hence, by (3.7.2.5) the solutions of (3.7.2.7) are given by

$$F(n) = \alpha 2^n + \beta n 2^n.$$

Next, **we discuss the solutions of the general (including inhomogeneous) recurrences**. The solution of the general second-order non-homogeneous recurrences with constant coefficients are given by the next theorem, which we state without proof.

Theorem 3.7.2.3:

The **general** solution to inhomogeneous equation

$$a F(n) + b F(n-1) + c F(n-2) = g(n) \quad \text{for } n > 1 \qquad (3.7.2.8)$$

can be obtained **as the sum** of the general solution of the homogeneous equation

$$a F(n) + b F(n-1) + c F(n-2) = 0$$

and a **particular** solution of (3.8.2.8).

The method of finding a particular solution of (3.8.2.8) and then a general solution of (3.8.2.8) is explained through the following examples.

Example 3.7.2.4

Let us consider the inhomogeneous recurrence

$$F(n) - 4F(n-1) + 4F(n-2) = 3$$

If F(n) = c is a particular solution of the recurrence, then replacing F(n), F(n−1) and F(n−2) by c in the recurrence given above, we get

$$c - 4c + 4c = 3$$

i.e., $c = 3$

Also, the general solution of the characteric equation (*of the inhomogeneous recurrence given above*) viz

$$F(n) - 4F(n-1) + 4F(n-2) = 0$$

Are (*obtained from Example 2.8.2.2, as*) $\alpha 2^n \beta n 2^n$

Hence general solution of the given recurrence is given by

$$F(n) = \alpha 2^n + \beta n 2^n + 3$$

Where α and β are arbitrary real constants.

3.8 AVERAGE-CASE AND AMORTIZED ANALYSIS

In view of the inadequacy of the best-case analysis and worst-case analysis for all types of problems, let us study two more analyses of algorithms, viz, average-case analysis and amortized analysis.

3.8.1 Average-Case Analysis

In Section 3.4, we mentioned that efficiency of an algorithm may not be the same for all inputs, even for the same problem-size n. In this context, we discussed *Best-Case analysis* and *Worst-Case analysis* of an algorithm.

However, these two analyses may not give any idea about the behaviour of an algorithm on a *typical or random* input. In this respect, *average-case analysis* is more informative than its two just-mentioned counter-parts, particularly, when the algorithm is to be used frequently and on varied types of inputs. In order to get really useful information from the average-case analysis of an algorithm, we must explicitly mention the properties of the set of inputs, obtained either through empirical evidence or on the basis of theoretical analysis. We explain the ideas through the analysis of the following algorithm, that, for given an element K and an array A[1..n], returns the index i in the array A[1..n], if A[i] = K otherwise returns 0.

```
Algorithm Sequential_ Search ( A [ 1.. n ], K)
  begin
       i ← 1
       while (i < n    and  A [ i ] ≠ K)    do
                i ←i + 1
       If   i < n        return i
            else    return    0
  end;
```

Some of the assumptions, that may be made, in the case of the above algorithm, for the purpose of average-case analysis, are

(i) for some number p, $0 \le p \le 1$, p is the probability of successful search and

(ii) in the case of successful search, the probability that first time K occurs in ith position in A[1..n], is the same for all i= 1, 2, …n.

With these two assumptions, we make *average-case analysis* of the Algorithm Sequential Search given above as follows:

From (i) above, Probability of unsuccessful search = (1− p).
In view of assumption (ii) above, in the case of successful search,
Probability of K occurring for the first time in the ith position in A[1..n] = p/n
for i= 1,2,..,n.

Therefore, if $C_{avg}(n)$, the average complexity for an input array of n elements, is given by

$$\mathbf{C_{avg}(n) = [\,1.(p/n) + 2.(p/n) + \ldots + i.(p/n) + \ldots + n.(p/n)] + n(1-p),}$$

where the term i . (p/n) is the contribution of i comparisons that have been made when executing while-loop i times such that i is the least index with A[i] = K after which while-loop terminates.

Also, the last term (n . ($1 - p$)) is the contribution in which while-loop is executed n times and after which it is found that A [i] ≠ K for i = 1, 2, …., n.

Simplifying R.H.S of the above equation, we get

$$C_{avg}(n) = \left(\frac{p}{n}\right) [1 + 2 + \ldots + i + .. + n] + n(1 - p)$$

$$= \frac{p}{n} \cdot \frac{n(n+1)}{2} + n. \quad (1-p)$$

$$= \frac{p(n+1)}{2} + n \ (1-p)$$

As can be seen from the above discussion, the average-case analysis is more difficult than the best-case and worst-case analyses.
Through the above *example* we have obtained an idea of how we may proceed to find average-case complexity of an algorithm. *Next we outline the process of finding average-case complexity of any algorithm, as follows*:

(i) First categories all possible input instances into classes in such a way that inputs in the same class require or are expected to require the execution of the same number of the basic operation(s) of the algorithm.

(ii) Next, the probability distribution of the inputs for different class as, is obtained empirically or assumed on some theoretical grounds.

(iii) Using the process as discussed in the case of Sequential-Search above, we compute the average-case complexity of the algorithm.

It is worth mentioning explicitly that average-case complexity need not be the average of the worst-case complexity and best-case complexity of an algorithm, though in some cases, the two may coincide.
Further, the effort required for computing the average-case, is worth in view of its contribution in the sense, that in some cases, the average-case complexity is much better than the Worst-Case Complexity. For example, in the case of Quicksort algorithm, which we study later, for sorting an array of elements, the *Worst-Case complexity* is a quadratic function whereas the *average-case complexity* is bounded by some constant multiple of n log n. For large values of n, *a constant multiple of (n log n)* is much smaller than *a quadratic function of n*. Thus, without average-case analysis, we may miss many on-the-average good algorithms.

3.8.2 Amortized Analysis

In the previous sections, we observed that

(i) *worst-case* and *best-case* analyses of an algorithm may not give a good idea about the behaviour of the algorithm on a *typical or random* input.

(ii) validity of the conclusions derived from *average-case* analysis depends on the quality of the assumptions about probability distribution of the inputs of a given size.

Another important fact that needs our attention is the fact that most of the operations, including the most time-consuming operations, on a data structure (used for solving a problem) do not occur in isolation, but different operations, with different time complexities, occur as a part of a *sequence* of operations. *Occurrences of a particular operation in a sequence of operations are dependent on the occurrences of*

other operations in the sequence. As a consequence, it may happen that the most time consuming operation can occur but only rarely or the operation only rarely consumes its theoretically determined maximum time. *We will support our claim later through an example*. But, we continue with our argument in support of the need for another type of analysis, viz., ***amortized analysis,*** for better evaluation of the behaviour of an algorithm. However, this fact of *dependence* of both the occurrences and complexity of an *operation* on the occurrences of other operations, is not taken into consideration in the earlier mentioned analyses. As a consequence, the complexity of an algorithm is generally over-evaluated.

Next, we give an example in support of our claim that the frequencies of occurrences of operations and their complexities, are generally interdependent and hence the impact of the most time-consuming operation may not be as bad as it is assumed or appears to be.

Example 3.8.2.1:

We define a new data structure say *MSTACK*, which like the data structure STACK has the usual operations of PUSH and POP. In addition there is an operation MPOP(S, k), where S is a given stack and k is a non-negative integer. Then MPOP(S, k) removes top k elements of the stack S, if S has at least k elements in the stack. Otherwise it removes all the elements of S. *MPOP (S, k) may be formally defined as*

Procedure MPOP (S, k);

```
begin
While (not Empty (S) and k ≠ 0) do
   POP (S)
   k ← k−1
Endwhile;
EndMPOP.
```

For example, If, at some stage the Stack S has the elements

```
35    40    27    18    6    11
↑                            ↑
TOP                        BOTTOM
```

Then after MPOP (S, 4) we have

```
6     11
↑     ↑
TOP   BOTTOM
```

Further another application of MPOP (S, 3) gives empty stack.

Continuing with our example of MSTACK, next, we make the following assumptions and observations:

(i) Cost of each PUSH and POP is assumed to be 1 and if $m \geq 0$ is the number of elements in the Stack S when an MPOP (S, k) is issued, then

$$\text{Cost (MPOP(S, k))} = \begin{cases} k & \text{if } k \leq m \\ m & \text{otherwise} \end{cases}$$

(ii) If we start with an empty stack S, then at any stage, the number of elements that can be POPed off the stack either through a POP or MPOP, can not exceed the total number of preceding PUSHes.

The above statement can be further strengthened as follows:

(ii a) If we start with an empty stack S, then, **at any stage**, the number of elements that can be popped off the stack **through all** the POPs and MPOPs can not exceed the number of all the earlier PUSHes.

For Example, if S_i denotes *ith* PUSH and M_j denote *jth* POP/MPOP and if we have a sequence of PUSH/POP/MPOP as (say)

$$S_1\ S_2\ S_3\ M_1\ S_4\ S_5\ M_2\ S_6\ S_7\ M_3\ S_8\ S_9\ S_{10}\ S_{11}\ M_4$$

Then in view of (i) above

$$\text{Cost}(M_1) \leq \text{Cost}(S_1\ S_2\ S_3) = 3$$
$$\text{Cost}(M_1) + \text{Cost}\ M_2 \leq \text{Cost}(S_1\ S_2\ S_3) + \text{Cost}(S_4\ S_5) = 5$$
$$\text{Cost}(M_1) + \text{Cost}(M_2) + \text{Cost}(M_3) \leq \text{Cost}(S_1\ S_2\ S_3) + \text{Cost}(S_4\ S_5) + \text{Cost}(S_6\ S_7) = 7$$

In general if we have a sequence of PUSH/POP/MPOP, **total n in number**, then for a sequence.

$$S_{11}\ S_{12} \ldots S_{1i_1}\ M_1\ S_{21}\ S_{22} \ldots S_{2i_2}\ M_2 \ldots S_{t1}\ S_{t2} \ldots St i_t\ M_t$$

Where Mj is either a POP or MPOP and (3.8.2.1)

$$(i_1 + 1) + (i_2 + 1) + \ldots\ldots + (i_t + 1) = n. \quad (3.8.2.2)$$
$$\Rightarrow i_1 + i_2 + \ldots\ldots + i_t \leq n \quad (3.8.2.3)$$
i.e, cost of all PUSHes $\leq n$. (3.8.2.4)
$$\text{Cost}(M_1) \leq \text{Cost}(S_{11}\ S_{12} \ldots S_{1i_1}) = i_1$$

$$\text{Cost}(M_2) \leq \text{Cost}(S_{21}\ S_{22} \ldots S_{2i_2}) = i_2$$
$$\text{Cost}(M_t) \leq \text{Cost}(S_{t1}\ S_{t2} \ldots S_{ti_t}) = i_t$$

$\therefore \text{Cost}(M_1) + \text{Cost}(M_2) + \ldots\ldots + \text{Cost}(M_t) \leq$ sum of costs of all Pushes
$= i_1 + i_2 + \ldots + i_t \leq n$ (from (3.8.2.3))

Therefore total cost sequence of n PUSHES/POPs/MPOPs in (3.8.2.1) is $\leq n + n = 2n$

Thus, we conclude that

Total cost a sequence of n operations in MSTACK $\leq 2n$, (3.8.2.5)
whatever may be the frequency of MPOPs in the sequence.

However, if we go by the worst case analysis, then in a sequence of n operations

(i) all the operations may be assumed to be MPOPs

(ii) the worst-case cost of each MPOP may be assumed as (n−1), because, theoretically, it can be assumed for worst cas analysis purpose, that before each MPOP all the operations, which can be at most (n−1), were pushes.

Thus, the worst-case cost of a sequence of n operations of PUSHes, and MPOPs = n . $(n-1) = n^2 - n$, which is quadratic in n. (3.8.2. 6)

Thus, further, we conclude that **though**

(i) the *worst-case* cost in this case as given by (3.8.2.6) is **quadratic**, **yet**

(ii) because of interdependence of MPOPs on preceding PUSHes, the *actual cost* in this case, as given by (3.8.2.5) which is only **linear**.

Thus, we observe that operations are not considered in isolation but as a part of a sequence of operations, and, because of interactions among the operations, highly-costly operations may either not be attained or may be distributed over the less costly operations.

The above discussion motivates the concept and study of AMORTIZED ANALYSIS.

3.9 SUMMARY

In this unit, the emphasis is on the *analysis* of algorithms, though in the process, we have also *defined* a number of algorithms. Analysis of an algorithm generally leads to computational complexity/efficiency of the algorithm.

It is shown that general analysis of algorithms may not be satisfactory for all types of situations and problems, specially, in view of the fact that the same algorithm may have vastly differing complexities for different instances, though, of the same size. This leads to the discussion of worst-case and best-case analyses in Section 3.4 and of average-case analysis and amortized analysis in Section 3.8.

In, Section 3.2, we discuss two simple examples of algorithms for solving the same problem, to illustrate some simple aspects of design and analysis of algorithms. Specially, it is shown here, how a minor modification in an algorithm may lead to major efficiency gain.

In Section 3.3, the following sorting algorithms are defined and illustrated with suitable examples:

(i) Insertion Sort

(ii) Bubble Sort

(iii) Selection Sort

(iv) Shell Sort

(v) Heap Sort

(vi) Merge Sort

(vii) Quick Sort

Though these algorithms are not analyzed here, yet a summary of the complexities of these algorithms is also included in this section.

Next, the process of analyzing an algorithm and computing complexity of an algorithm in terms of basic instructions and basic constructs, is discussed in Section 3.5 and 3.6.

The Section 3.5 deals with the analysis in terms of non-recursive control structures in and the Section 3.6 deals with the analysis in terms of recursive control structures.

3.10 SOLUTIONS/ANSWERS

Ex. 1) List to be sorted: 15, 10, 13, 9, 12, 17 by Insertion Sort.

Let the given sequence of numbers be stored in A[1..6] and let m be a variable used as temporary storage for exchange purposes.

Iteration (i): For placing A[2] at its correct relative position w.r.t A[1] in the finally sorted array, we need the following operations:

(i) As A[2] = 10 < 15 = A [1], therefore, we need following additional operations
(ii) 10= A[2] is copied in m, s.t A[1] = 15, A[2] = 10, m = 10
(iii) 15 = A [1] is copied in A [2] s.t A[1] = 15, A[2] = 15, m = 10
(iv) 10 = m is copied in A[1], so that A[1] = 10, A[2] = 15.

Thus, we made one comparison and 3 assignments in this iterations
Iteration (ii): *At this stage A [1] = 15, A [2] = 10 and A [3] = 13*

Also, for correct place for A [3] = 13 w.r.t A[1] and A[2], the following operations are performed

(i) 13 = A[3] *is compared* with A[2] = 15

As A[3] < A[2], therefore, the algorithm further performs the following operations

(ii) 13 = A[3] copied to m so that m = 13

(iii) A [2] is copied in A [3] s.t. A[3] = 15 = A[2]

(iv) Then 13 = m is *compared with*

A [1] = 10 which is less than m.

Therefore A[2] is the correct location for 13

(v) 13 = m is copied in A[2] *s.t., at this stage*

A[1] = 10, A[2] = 13, A[3] = 15

And m = 13

In this iteration, the algorithm made 2 comparisons and 3 assignments

Iteration III: For correct place for A[4] = 9 w.r.t A[1], A[2] and A[3], the following operations are performed:

(i) A[4] = 9 *is compared with* A[3] = 15.
As A[4] = 9 < 15 = A[3], hence the algorithm.
(ii) 9 = A[4] is copied to m so that m = 9
(iii) 15 = A [3] is copied to A [4] s.t A[4] = 15 = A[3]
(iv) m is *compared with* A[2] = 13, as m < A [2], therefore, further the following operations are performed.
(v) 13 = A[2] is copied to A[3] s.t A[3] = 13
(vi) m = 9 is *compared with A[1]* = 10, as performs the following additional operations.
(vii) 10 = A[1] is copied to A[2]
(viii) finally 9 = m is copied to A[1]

In this iteration 3 comparisons and 5 assignments were performed

So that at this stage we have

A[1] = 9, A[2] = 10, A[3] = 13, A[4] = 15

Iteration IV: For correct place for A[5] = 12, the following operations are performed.

(i) 12 = A [5] is compared with A[4] = 15

In view of the earlier discussion, and in view of the fact that the number 12 (contents of A[5]) occurs between A[2] = 10 and A[3] = 13, the algorithm need to perform, in all, the following operations.

(a) Copy 12 = A[5] to m so that m becomes 12 **(one assignment)**
(b) *Three Comparisons* of A[5] = 12 are with A[4] = 15, A[3] = 13 and A[2] = 10. The algorithm stops comparisons on reaching a value less than value 12 of current cell
(c) The following THREE assignments: A[4] = 15 to A[5], A[3] = 13 to A[4] and m = 12 to A[3]

Thus in this iteration 4 assignments and 3 comparisons were made. And also, at this stage A[1] = 9, A[2] = 10, A[3] = 12, A[4] = 13, A[5] = 15 and m = 12.

Iteration V: The correct place for A[6] = 17, after sorting, w.r.t the elements of A[1..5] is A[6] itself. In order to determine that A[6] is the correct final position for 17, we perform the following operations.

(i) 17 = A[6] is copied to m **(one assignment)**
(ii) m is compared with A[5] = 15 and as A[5] = 15 < 17 = m, therefore, no more comparisons and no copying of elements A[1] to A[5] to the locations respectively on their right.
(iii) (*though this step appears to be redundant yet*) *the algorithm executes it*
17 = m is copied to A[6]

In this iteration 2 assignments and one comparison were performed.

To summerize:

In	I iteration,	1 comparison and	3 assignment were performed
In	II iteration,	2 comparison and	3 assignment were performed
In	III iteration,	3 comparison and	5 assignment were performed
In	IV iteration,	3 comparison and	4 assignment were performed
In	V iteration,	1 comparison and	3 assignment were performed

Thus, in all, 10 comparisons and 18 assignments were performed to sort the given array of 15,10,13,9,12 and 17.

Ex. 2) **List to be sorted: 15, 10, 13, 9, 12, 17 by Bubble Sort.**

A temporary variable m is used for exchanging values**. An exchange A [i]↔ A [j] takes 3 assignments viz**

m ← A [i]
A [i] ← A [j]
A [j] ← m

There are (6 – 1) = 5 iterations. In each iteration, whole list is scanned once, comparing pairs of neighbouring elements and exchanging the elements, if out of order. Next, we consider the different iterations.

In the following, the numbers in the bold are compared, and if required, exchanged.

Iteration I:

15	**10**	13	9	12	17
10	**15**	**13**	9	12	17
10	13	**15**	**9**	12	17
10	13	9	**15**	**12**	17
10	13	9	12	**15**	**17**
10	13	9	12	15	17

In this iteration, 5 comparisons and 5 exchanges i.e., 15 assignments, were performed

Iteration II: The last element 17 is dropped from further consideration

10	**13**	9	12	15
10	**13**	**9**	12	15
10	9	**13**	**12**	15
10	9	12	**13**	**15**
10	9	12	13	15

In this iteration 4 comparisons, 2 exchanges i.e., 6 assignments, were performed.

Iteration III: The last element 15 is dropped from further consideration

10	**9**	12	13
9	**10**	**12**	13
9	10	**12**	**13**
9	10	12	13

In this iteration 3 comparisons and 1 exchange i.e., 3 assignments were performed.

Iteration IV: The last element 13 is dropped from further consideration

9	**10**	12
9	**10**	**12**
9	10	12

In this iteration, 2 comparisons and 0 exchanges, i.e., 0 assignments were performed

Iteration V: The last element 12 is dropped from further consideration

9	**10**
9	**10**

In this iteration, 1 comparison and 0 exchange and hence 0 assignments were performed.

Thus, the Bubble sort algorithm performed (5+4+3+2+1) = 15 comparisons and 24 assignments in all.

Ex. 3) List to be sorted: 15, 10, 13, 9, 12, 17 by Selection Sort.

There will be five iterations in all. *In each of the five iterations at least, the following operations are performed:*

- 2 initial assignments to MAX and MAX-POS and

- 2 final assignments for exchanging values of A [MAX_POS] and A [n – i+1], the last element of the sublist under consideration.

Next we explain the various iterations and for each iteration, count the operations in addition to these 4 assignments.

Iteration 1: MAX ← 15
MAX_ POS ← 1
MAX is compared with successively 10, 13, 9, 12, and 17, one at a time i.e, 5 comparisons are performed.

Also MAX = 15 < 17 ∴ MAX ← 17 and MAX_ POS ← 6

Thus in addition to the 4 assignments mentioned above, 2 more assignments and 5 comparisons are performed:

Iteration 2: Now the list under consideration is

15, 10, 13, 9, 12

Again MAX ← 15, MAX_POS ← 1
Further MAX is compared successively with 10, 13, 9, 12 one at a time and as none is more than 15, therefore, no change in the value of MAX and MAX_POS. Ultimately 12 and 15 are exchanged to get the list 12, the last element of the sublist under consideration 10, 13, 9, 15.

Thus, this iteration performs 4 comparisons and no additional assignments besides the 4 mentioned earlier.

Iteration 3: The last element 15, is dropped from further consideration. The list to be consideration in this iteration is 12, 10, 13, 9

Initially MAX ← 12 and MAX_POS ← 1. MAX is compared with 10 and then with 13. As 13 > 12 = current MAX. Therefore, MAX ← 13, and MAX_POS ← 3 *(2 additional assignments).* Then MAX is compared with 9 and then 13 and 9 are exchanged we get the list: 12, 10, 9, 13 .

Thus in this iteration, 3 comparisons and six assignments are performed, in addition to the usual 4.

Iteration 4: The last element 13, is dropped from further consideration. The list to be sorted is 12, 10, 9
Again MAX ← 12, MAX_POS ← 1

The list after the iteration is 10, 9, 12. Again 2 comparisons of 12 with 10 and 9 are performed. No additional assignments are made, in addition to normal 4.

Iteration 5: The last element 12, is dropped from further consideration. The list to be sorted is 10, 9. MAX ← 10 and MAX_POS ← 1. One comparison of MAX = 10 with 9. No additional assignment over and above the normal 4.

Finally, the list to be sorted is: 9 and the process of sorting terminates.

The number of operations performed iteration-wise, is given below:

In Iteration I	:	5 comparisons and 6 assignments were performed
In Iteration II	:	4 comparisons and 4 assignments were performed
In Iteration III	:	3 comparisons and 6 assignments were performed
In Iteration IV	:	2 comparisons and 4 assignments were performed
In Iteration V	:	1 comparison and 4 assignments were performed

Hence total 15 comparisons and 24 assignments were performed to sort the list.

Ex.4) The array to be sorted is A [1 .. 6] = {15, 10, 13, 9, 12, 17}

(i) To begin with , increments in indexes to make sublists to be sorted, with value 3 from INC [1] and value 1 from INC [2], are read.

Thus two READ statements are executed for the purpose.

(ii) For selection of sublists A [1] = 15, A [2] = 9 **and** A [2] = 10, A [5] = 12 **and** A[3] = 13 A [6] = 17, the following operations are performed:

for INC [1] = 3
j ← INC [1] = 3
r ← [n/j] = [6/3] = 2

Two assignments are and one division performed. (A)

Further for each individual sublist, the following *type* of operations are performed.

t ← 1 (*one assignment*)

The comparison 6 = n < 2+3+1 is performed which returns true, hence

s ← r – 1 = 2 – 1 = 1 is performed (*one subtraction and one assignment*)

Next, to calculate the position of A [t+3 × s], one multiplication and one addition is performed. Thus, for selection of a particular sublist, 2 assignments, one comparison, one subtraction, one addition and one multiplication is performed

Thus, just for the selection of all the three sublists, the following operations are performed:

6 Assignments
3 Comparisons
3 Subtractions
3 additions
3 multiplications

(iii) (a) For sorting, the sublist

A [1] = 15, A [2] = 9 using Insertion sort,

first, the comparison 9 = A [2] < A [1] = 15 is performed which is true, hence

$i \leftarrow 1$
$m \leftarrow A[2] = 9$ **(two assignments performed)**

Next, the comparisons m = 9 < A [1] =15 and 1 = i < 0 are performed, both of which are true. **(two comparisons)**

15 = A [1] is copied to A [2]
and $i \leftarrow 1 - 1 = 0$ **(assignment)**

Next again the one comparisons is performed viz i > 0 which is false and hence 9 = m < A [0] is not performed

Then 9 = m is copied to A [1] **(one assignment)**

(iii) (b) For sorting the next sublist 10, 12

As A [2] = 12 < 10 = A [1] is not true hence no other operation is performed for this sublist

(iii) (c) For sorting the next sublist 13, 17

only one comparison of A[2] = 17< 13 = A [1], which is not true, is performed.

We can count all the operations mentioned above for the final answer.

Ex.5) To sort the list 15, 10, 13, 9, 12, 17 stored in A[1..6], **using Heap Sort** first build a heap for the list and then recursively delete the root and restore the heap.

Step I

(i) the five operations of assignments of values from 2 to 6 to j the outermost loop of variable, are made. Further, five assignments of the form: location $\leftarrow$ j are made one for each value of j.

Thus 10 assignments are made so for.

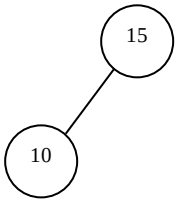
(ii) (a) Next, enter the while loop

For j = 2
2 = location > 1 is tested, which is true. **(one comparison)**

Hence
parent $\leftarrow \left(\frac{\text{location}}{2}\right) = 1,$ is performed. **(one comparison)**

A [location] = A [2] = 10 < 15 = A [1] = A [parent]
is tested which is true. Hence no exchanges of values. **(one comparison)**

The heap at this stage is



(ii) (b) For j = 3

3 = location >1 is tested **(one comparison)**

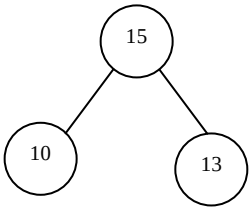
which is true. Therefore,

Parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = \left\lfloor \frac{3}{2} \right\rfloor = 1$ is performed **(assignments)**

A [location] = A [3] = 13 < 15 = A [1] = A [parent] is tested **(one comparison)**

As the last inequality is true, Hence no more operation in this case.

The heap at this stage is



(ii) (c) For j $\leftarrow$ 4

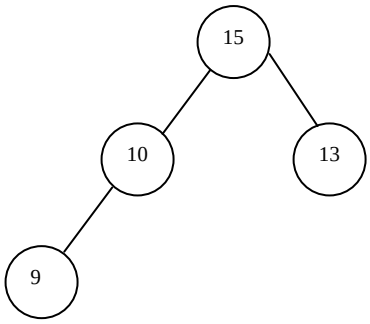
Location = 4>1 is tested **(one comparison)**

which is true. Therefore

Parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = \left\lfloor \frac{4}{2} \right\rfloor = 2$ is performed **(one assignment)**

A [location] = A [4] = 9 < 10 = A [2] is performed. **(one comparison)**

As the above inequality is true, no more operations in this case. The heap at this stage is



(ii) (d) For j $\leftarrow$ 5 is tested which

The Comparison

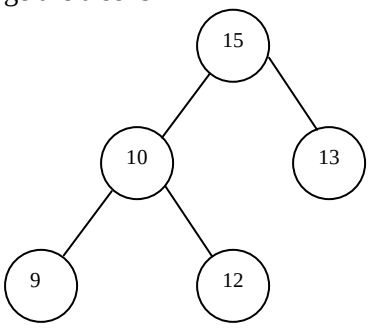
Location = 5 > 1 is performed, **(one comparison)**

which is true. Therefore,

Parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = \left\lfloor \frac{5}{2} \right\rfloor = 2$

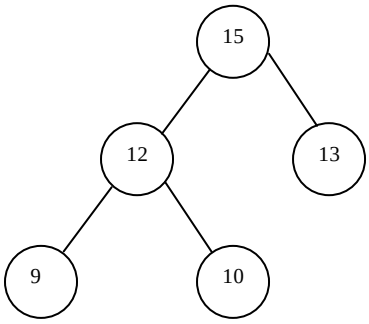
is performed **(one comparison)**

At this stage the tree is



A [location] = A [5] = 12 < A [Parent] = A [2] = 10 is
Performed. **(one comparison)**

which is not true. Hence the following additional operations are performed:
A [2] and A [5] are exchanged **(3 assignments)** so that the tree becomes



Also, the operations
location $\leftarrow$ parent = 2 and parent $\leftarrow$ (location/2) = 1 are performed
(two assignment)

Further 2 = location >1 is tested, **(one comparison)**

which is true. Therefore,
A[location] = A [2] $\leq$ A [1] = A [parent] is tested **(one comparison)**

which is true. Hence no more operations.

(ii) (e) j $\leftarrow$ 6

The comparison
Location = 6>1 is performed **(one comparison)**
Therefore, parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = 3$ is performed **(one assignment)**
A [location] = A [6] = 17 < 9 = A[3] is performed **(one comparison)**

which is not true. Hence, the following additional operations are performed

A [3] and A[6] are exchanged **(3 assignment)**

Next
location $\leftarrow$ 3 **(one assignment)**
(location>1) is performed **(one comparison)**

and

parent $\leftarrow \left\lfloor \frac{location}{2} \right\rfloor = 1$ **(one assignment)**

is performed
Further

A [location] = 17 < 15 = A [parent] is performed, **(one comparison)**

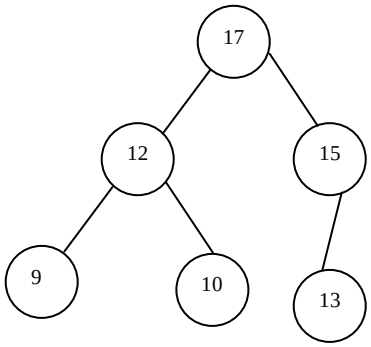
which is false.

Hence, the following operations are further performed

A [1] and A [3] are exchanged **(3 assignments)**

And A [location] ← A [parent] = 1 is performed **(one assignments)**
Also
(1 = location >1) is performed **(one comparison)**

which is not true. Hence the process is completed and we get the heap

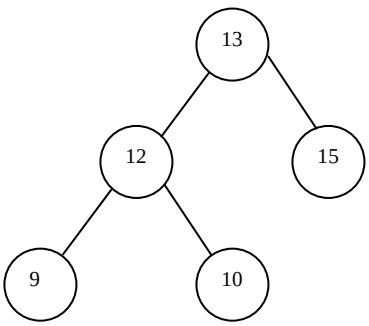


Step II: The following three substeps are iterated 1 repeated 5 times:

a. To delete the root
b. to move the value of the last node into the root and the last node is removed from further consideration
(iii) Convert the tree into a Heap.

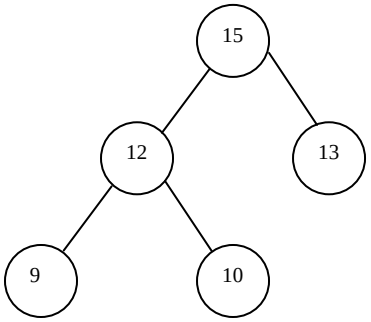
The sub steps (i) and (ii) are performed 5 times each, which contribute to 10 assignments

Iteration (i): after first two sub steps the *heap* becomes the *tree*



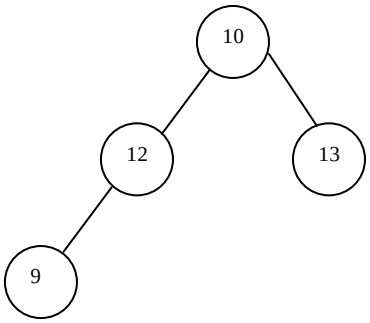
Root node is compared, one by one, with the values of its children **(2 comparisons)**
The variable MAX stores 15 and MAX_POS stores the index of the right child **(Two assignment)**

Then 15 is exchanged with 13 **(3 assignments)**
is performed, to get the heap



Thus in this iteration, 2 comparisons and 5 assignments were performed

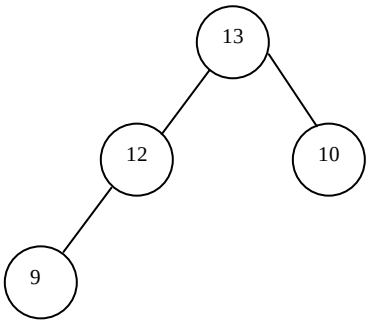
Iteration (ii): 15 of the root is removed and 10, the value of last node, is moved to the root to get the tree



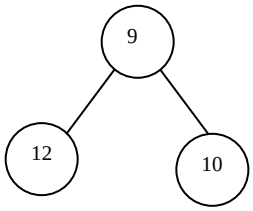
Again, the value 10 of the root is compared with the children
(2 comparison)

MAX first stores value 12 and then 13, and MAX_POS stores first the index of the left child and then index of the right child **(4 assignments)**

Then 13 and 10 are exchanged so that we get the Heap

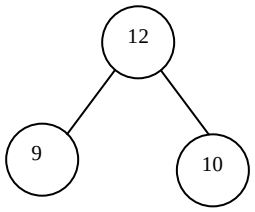


Iteration (iii): Again 13 of the root mode is removed and 9 of the last node is copied in the root to get the tree

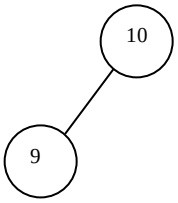


2 comparisons are made of value 9 with 12 and 10. Also **two assignments** for MAX ← 12 and MAX_POS ← index of the Left –child, are made. Also 12 and 9 are exchanged requiring **3 additional assignments.**

Thus in this iteration, 2 comparisons and 5 assignments are performed to get the heap



Iteration (iv): 12 is removed and the value 10 in the last node is copied in the root to get the tree



10 is compared with 9 **(only one comparison)**
and no assignments

In this iteration only one comparison is made

Iteration (v) (i): 10 is deleted from the root and 9 is copied in the root. As the root is the only node in the tree, the sorting process terminates.

Finally, by adding the various operations of all the iterations, we get the required numbers of operations.

Ex. 6) **List to be sorted: 15, 10, 13, 9, 12, 17 by Merge Sort.**

Chop the given list into two sublists

((15, 10, 13) (9, 12, 17))
Further chopping the sublists we get

((15) (10, 13)), ((9) (12, 17))
Further chopping the sublists we get sublists each of one element

(((15), ((10), (13))), ((9), ((12), (17)))

merging after sorting, in reverse order of chopping, we get

((15), (10, 13)) ((9), (12, 17))

Again merging, we get

((10, 13, 15) (9,12, 17))

Again merging, we get

(9,10, 12, 13, 15, 17)
This completes the sorting of the given sequence.

Ex. 7)

The sequence
15, 10, 13, 9, 12, 17,

to be sorted is stored in A [1.. 6]
We take A [1] = 15 as pivot
i is assigned values 2, 3, 4 etc. to get the first value from the left such that A[i] > pivot.

Similarly, j is moved backward from last index to get first j so that A[j] < pivot.

The index i = 6, is the first index s.t 17 = A[i] > pivot = 15.
Also j = 5 i.e. A [5] = 12, is the first value from the right such that A [j] < pivot.

As j < i, therefore
A[j] = A [5] = 12 is exchanged with pivot = 15 so that we get the array

12, 10, 13, 9, 15, 17

Next the two sublists viz 12, 10, 13, 9 and 17 separated by the pivot value 15, are sorted separately. However the relative positions of the sublists w.r.t 15 are maintained so we write the lists as

(12, 10, 13, 9), 15, (17).

The right hand sublist having only one element viz 17 is already sorted. So we sort only left-hand sublist but continue writing the whole list. Pivot for the left sublist is 12 and i = 3 and j = 4 are such that A[i] = 13 is the left most entry more than the pivot and A[j] =9 is the rightmost value, which is less than the pivot = 12. After exchange of A[i] and A[j], we get the list (12, 10, 9, 13), 15, (17). Again moving i to the right and j to the left, we get, i = 4 and j = 3. As j < i, therefore, the iteration is complete and A[j] = 9 and pivot = 12 are exchanged so that we get the list **((9, 10) 12 (13)) 15 (17)**Only remaining sublist to be sorted is (9, 10). Again pivot is 9, I = 2 and j = 1, so that A [i] is the left most value greater than the pivot and A[j] is the right most value less than or equal to pivot. As j < i, we should exchange A [j] = A[1] with pivot. But pivot also equals A [1]. Hence no exchange. Next, sublist left to sorted is {10} which being a single element is already sorted. The sublists were formed such that any element in a sublist on the left is less than any element of the sublist on the right, merging does not require.

3.11 FURTHER READINGS

1. *Foundations of Algorithms,* R. Neapolitan & K. Naimipour: (D.C. Health & Company, 1996).

2. *ALGORITMICS: The Spirit of Computing,* D. Harel*:* (Addison-Wesley Publishing Company, 1987).

3. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth: (Narosa Publishing House).

4. *Fundamentals of Algorithmics,* G. Brassard & P. Brately: (Prentice-Hall International, 1996).

5. *Fundamentals of Computer Algorithms,* E. Horowitz & S. Sahni: (Galgotia Publications)

6. *The Design and Analysis of Algorithms,* Anany Levitin: (Pearson Education, 2003).

7. *Discrete Mathematics and Its Applications,* K.N. Rosen: (*Fifth Edition*) Tata McGraw-Hill (2003).

8. *Introduction to Alogrithms* (*Second Edition*), T.H. Coremen, C.E. Leiserson & C. Stein: Prentice – Hall of India (2002).