12.4.6 Polynomial Time Complexity

In mathematics, a polynomial is an expression of finite length constructed by the operations of addition, subtraction, multiplication, and non-negative integer exponents from variables and constants. As an example, $f(x) = x^4 + 2x^3 - 3x + 4$ is a polynomial of degree 4.

An algorithm is said to be in polynomial time complexity if its execution time is upper bounded by a polynomial expression in the size of the input for the algorithm. The complexity is denoted as O(nk), where k is a constant.

Let us discuss some algorithms to make the idea clear.

Example 12.10 Find the time complexity of bubble sort.

Solution: Consider the example of bubble sort to sort a list of n items.
Here, an array called list is taken as input. The array is sorted by bubble sort() using the following algorithm.

```
Procedure:  BubbleSort(List[])
  Inputs:  List[] – A list of numbers
  Locals:  i, j– integers
  Begin:
     For i = 0 to List.Size–1
        For j = i + 1 to List.Size–1
           If List[i] > List[j], Then
              Swap List[i] and List[j]
           End If
        Next j
     Next i
  End
```

Start calculating the complexity of the algorithm from the 'swap' operation of two elements. Regardless of size of the list, 'swap' always takes the same amount of time, i.e., $O(1)$.

The 'swap' operation depends on the 'if statement' condition. If the condition is false, it does not execute swap.In the worst case (if condition holds true), swap executes. Again, the 'if' statement is of O(1). Thus, if the condition is true, it takes maximum time which is still $O(1)$.

The 'if' statement is surrounded by two 'for' loops. The 'if' statement is executed as many times as the inner 'for' loop iterates, and the inner 'for' loop executes as many times as the outer 'for' loop iterates. For each iteration of the outer loop, the inner loop executes $N-1$ times, where N is the size of list[]. The outer loop iterates $N-1$ times. So, the complexity for this algorithm is:

$$\left(\sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1}(1)\right)=\sum_{i=0}^{N-1}(N-1-(i+1)+1)$$

$$\rightarrow \sum_{i=0}^{N-1}(N-i-1) \rightarrow N\sum_{i=0}^{N-1}1-\sum_{i=0}^{N-1}(i+1) \rightarrow N^2-(1,2,3\ldots.N) \rightarrow N^2-\frac{N(N+1)}{2} \rightarrow \frac{N^2-N}{2}$$

So, the complexity is $O(N^2)$. Here, $\frac{N^2-N}{2}$ is a polynomial of N of degree 2.

Example 12.10 Find the time complexity of matrix multiplication.

Solution : *Matrixmultiplication* is a two-dimensional array. Matrix multiplication can be done if the number of rows of the first matrix is equal to the number of columns of the second matrix. Matrix multiplication is done by the following algorithm.

```
Procedure:  MatrixMul(MAT1[][], MAT2[][])
Inputs:  MAT1[][], MAT2[][]
  Locals:  i, j, k - integers
  Mult[][] = 0 -integers
  If No.  of row of MAT1 = No.  of column of MAT2
```

```
For i = 0 to No.  of row of MAT1
  For j = 0 to No.  of column of MAT2
     Mult [i][j] = 0
     For k = 0 to No.  of row of MAT1
        Mult[i][j] = Mult[i][j] + MAT1[i][k]* MAT2[k][j]
     Next k
        Return Mult [i][j]
  Next j
Next i
End If
```

Start calculating the complexity of the algorithm from 'mult'. 'Mult' consists of addition and multiplication. Let the time taken for addition and multiplication be t_1 and t_2, respectively. The total time for 'mult' operation is $t_1 + t_2$, and thus constant, i.e., $O(1)$. This operation is surrounded by three 'for' loops of i, j, and k. So, the complexity for this algorithm is:

$$\left(\sum_{i=0}^{N-1}\left(\sum_{j=0}^{N-1}\left(\sum_{k=0}^{N-1}(1)\right)\right)\right) = \sum_{i=0}^{N-1}\left(\sum_{j=0}^{N-1}N\right) = \sum_{i=0}^{N-1}N\sum_{j=0}^{N-1}1 = \sum_{i=0}^{N-1}N^2 = N^2\sum_{i=0}^{N-1}1 = N^3$$

So, the complexity is $O(N^3).N^3$ is a polynomial of N of degree 3.

12.4.7 Super Polynomial Time Complexity

This is sometimes called exponential time complexity. An algorithm is called super polynomial time complexity if it runs in 2^n steps for an input of size n. The execution time of such algorithms increases rapidly as n grows.

A funny story on exponential growth is the suitable place to mention in this context. An Indian Brahmin Sissa wanted to teach his king a good lesson. The king told this Brahmin proudly that he can pay any amount of paddy grain as he has a large grain container. The Brahmin challenged the king to play chess. The king was defeated. In return, the Brahmin requested the king to pay him such amount of grains so that he can place 1 packet of grain (about 20 kg) on the fi rst box, 2 on the second, 4 on the third, 8 on the fourth, and so on. So, the last box would contain 2^{63} packets of grains. (A chess board contains 64 boxes.) The total amount of grains payable to the Brahmin is $2^{64} - 1 = 1,84,467,44,07,370,95,51,616$ packets. India requires several years to produce this amount of paddy grains!

This is called exponential grows. The problems with exponential grows are hard to compute by a computer.

Consider the following problems.

There are 15 different-sized tennis balls. The job of the computer is to arrange the balls such that no two sets of arrangement are equal.

The numbers of such different arrangements are $15! = 1307674368000$.

Let there be a computer which can print 10^6 such arrangements in 1 second. This computer will take about 2.5 years to print all such arrangements!

Example 12.12 Find the time complexity of the tower of the Hanoi problem.

Solution: Tower of Hanoi is a mathematical puzzle invented by E. Lucas in 1883. This consists of three rods and a number of different-sized disks with a hole in the middle so that the disks can be arranged by sliding them on the rods. At the initial stage, the disks are placed in ascending order with the smallest radius on the top on the leftmost rod.

The objective of the puzzle is to move the entire stack of disks from the leftmost to the rightmost rod, obeying the following conditions:

- At a time, only one disk can be moved.
- In each move, the upper disk from a rod is slided ,out and placed on the top of the other disks that may already be present on the another rod.
- A larger radius disk is not allowed to be placed on the top of a smaller disk.

To solve this problem, three rods are named as 'source', 'auxiliary', and 'destination'. There are n disks to be shifted from one rod and placed on another rod, which abide by the rules given previously. If the number of disks is 1, it can be easily moved from one rod to another. It can be considered as the base case. For n disks, this can be implemented by the following process.

i) Move the top $n-1$ disks from source to auxiliary.

ii) Move the bottom disk from source to destination.

iii) Move $n-1$ disks from auxiliary to destination.

The algorithm to implement this is given as follows.

```
Tower of Hanoi (n, source, auxiliary, destination)
  {
     If n = 1 move disk from source to destination; (base
case)
     Else,
        {
           Tower of Hanoi(top n − 1, source, destination,
auxiliary);
           Move the nth disk from source to destination;
           Tower of Hanoi(n − 1, auxiliary, source,
destination);
        }          }
```

Moving operation takes a unit time. And the recursive functions take $T(n-1)$ time each. Thus, the time required for running the else part of the algorithm one time is $T(n) = 2T(n-1)+1$. The subsequent operation will continue up to $T(2)$. The total time for the algorithm is

$$T(n) = 2T(n-1)+1 \quad (1)$$

$$T(n-1) = 2T(n-2)+1 \quad (2)$$

$$T(n-2) = 2T(n-3)+1 \quad (3)$$

.

.

.

.

.

$$T(2) = 2T(1) + 1 \qquad (n-1)th$$

Multiplying equation $(2), (3), (4), ...(n-1)$ th by $2^1, 2^2, ...2^{n-1}$ on both sides, we get

$$T(n) = 2T(n-1) + 1 \quad (1)$$

$$2[T(n-1) = 2T(n-2) + 1] \quad (2)$$

$$2^2[T(n-2) = 2T(n-3) + 1] \quad (3)$$

.

.

.

.

$$2n - 2[T(2) = 2T(1) + 1] \qquad (n-1)^{th}$$

Thus, the complexity of the algorithm is $O(2^n)$

12.5 The Classes P

An algorithm is known as polynomial time complexity problem if the number of steps (or time) required to complete the algorithm is a polynomial function of n (where n is some non-negative integer) for a problem of size n. In other words, for a problem of size n, the time complexity of the algorithm to solve the problem is a function of the polynomial of n.

In general, polynomial class contains all of the problems which are solved using computers easily. The problems those we solve using computers do not require too many computations, but hardly $O(n^3)$ or $O(n^4)$ time. Truly speaking, most of the important algorithms we have learnt are somewhere in the range of $O(\log n)$ to $O(n^3)$. Thus, we shall state that the time complexity of practical computation resides within polynomial time bounds. These classes of problems are written as the P class problem.

*Definition*1: The class of polynomial solvable problems, P, contains all sets in which the membership may be decided by an algorithm whose running time is bounded by a polynomial.

*Definition*2: P is defined as the set of all decision problems for which an algorithm exists which can be carried out by a deterministic Turing machine in polynomial time.