UNIT 3 COMPUTABILITY AND COMPLEXITY

Structure

3.0 INTRODUCTION

Every system—natural or man-made, must be continuously, involved in some form of **computation** in its attempt at preserving its identity as a system.

In the previous unit, we discussed two major approaches to modeling of computation viz. the automata/machine approach and linguistic/grammatical approach. Under the automata approach, we discussed two models viz. Finite Automata and Nondeterministic Finite Automata. Under grammatical approach, we discussed only one model viz Regular Languages. We stated that the Finite Automata is a computational model which is equivalent to Regular Language Model and differentiated between a Deterministic Finite Automata and Nondeterministic Finite Automata.

In this unit we discuss a computational model called Turing Machine(TM) which is more powerful in terms of recognizing more languages and will help us **define and understand complexity classes** .

Turing Machine is named so, in Honor of its inventor Alan Mathison Turing (1921-1954). A.M. Turing, a British, was one of the greatest scholars of the twentieth century, and made profound contributions to the foundations of computer science.

We make an introduction to the concepts of a simple Turing Machine and Nondeterministic TM. Many notations and keywords are used to discuss the topics which are listed below:

Key words: Turing Machine (**TM**), Deterministic Turing Machine, Non-Deterministic Turing Machine, Turing Thesis, Computation, Configuration of TM, Turing-Acceptable Language, Turing DecidableLanguage,

Notations

TM	:	Turing Machine
Γ	:	Set of tape symbols, includes #, the blank symbol
Σ	:	Set of input/machine symbols, does not include #
Q	:	the finite set of states of TM
F	:	Set of final states
a,b,c…	:	Members of

σ	:	Variable for members of ∑
x or x	:	Any symbol of other than x
#	:	The blank symbol
α,β,γ	:	Variables for String over
L	:	Move the Head to the Left
R	:	Move the Head to the Right q : A state of TM, i.e, q ε Q
s or q_0	:	The start/initial state

Halt or h: The halt state. The same symbol h is used for the purpose of denoting halt state for all halt state versions of TM. And then h is not used for other purposes.

e or ϵ : The empty string

$C_1 \vdash_M C_2$: Configuration C_2 is obtained from configuration C_1in *one* moveOf the machine M

$C_1 \vdash^* C_2$: Configuration C_2 is obtained from configuration C_1in *finite* numberof moves.

$w_1 \underline{a} w_2$: The symbol a is the symbol currently being scanned by the Head

Or

$w_1 \underset{\uparrow}{a} w_2$: The symbol a is the symbol currently being scanned by the Head

3.1 OBJECTIVES

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

Define all the terms listed under the keywords

Illustrate a basic Turing machine.

Explain the symbols used in a Turing Machine

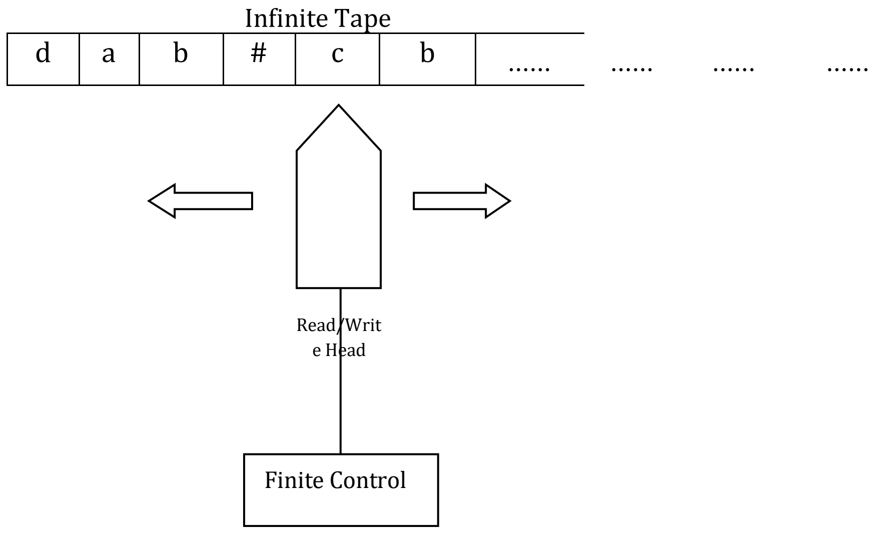
Explain the operations of a Turing Machine

Differentiate between TM and NDTM

Define Complexity Classes: P and NP

PRELUDE TO FORMAL DEFINITION

In the next section, we will notice through a formal definition of TM that a TM is an abstract entity constituted of mathematical objects like sets and a (partial) function. However, in order to help our understanding of the subject-matter of TMs, we can visualize a TM as a physical computing device that can be represented as a diagram as shown in below.



TURING MACHINE

Fig. 3.1

.

As shown in the above figure, TM consists of

(i) a **tape**, with an end on the left but infinite on the right side. The tape is divided into squares or cells, with each cell capable of holding one of the tape symbols including the blank symbol #. At any time, there can be only finitely many cells of the tape that can contain non-blank symbols. The set of **tape symbols** is denoted by

As the very first step in the sequence of operations of a TM, **the input, as a finite sequence of the input symbols is placed in the left-most cells of the tape.** The set of **input symbols** denoted by Σ , does not contain the blank symbol #. However, during operations of a TM, a cell may contain a **tape symbol** which is not necessarily an input symbol.
There are versions of TM, to be discussed later, in which the tape may be infinite in both left and right sides having neither left end nor right end.

(ii) **a finite control**, which can be in any one of the finite number of states. The states in TM can be divided in three categories viz.

(a) the Initial state, the state of the control just at the time when TM starts its operations. The initial state of a TM is generally denoted by q_0 or s.

(b) the Halt state, which is the state in which TM stops all further operations.

The halt state is generally denoted by h. The halt state is distinct from the initial state. Thus, a TM HAS AT LEAST TWO STATES.

(c) Other states

(iii) **a tape head** (*or simply Head*), is always stationed at one of the tape cells and provides communication for interaction between the tape and the finite control. The Head can read or scan the symbol in the cell under it. The symbol is communicated to the finite control. The control taking into consideration the symbol and its current state decides for further course of action including

the change of the symbol in the cell being scanned and/or

change of its state and/or

moving the head to the Left or to the Right. The control may decide not to move the head.

The course of action is called a move of the Turing Machine. In other words, the move is a function of current state of the control and the tape symbol being scanned.

In case the control decides for change of the symbol in the cell being scanned, then the change is carried out by the head. This change of symbol in the cell being scanned is called writing of the cell by the head.

Initially, the head scans the left-most cell of the tape.

Now, we are ready to consider a **formal definition** of a Turing Machine in the next section.

3.2 TURING MACHINE

3.2.1 Turing Machine: Formal Definition

There are a number of versions of a TM. We consider below Halt State version of formal definition of a TM.

Definition: Turing Machine (Halt State Version)
A Turing Machine is a sextuple of the form $(Q, \Sigma, \Gamma, \delta, q_o, h)$, where

(i) Q is the finite set of states,

(ii) Σ is the finite set of non-blank information symbols,

(iii) Γ is the set of tape symbols, including the blank symbol #

(iv) δ is the **next-move** partial function from Q x Γ to Q x Γ x {L, R, N},

where '**L**' denotes the tape Head moves to the left adjacent cell, '**R**' denotes tape Head moves to the Right adjacent cell and '**N**' denotes Head does not move, **i.e.**, continues scanning the same cell.

In other words, for $q_i \in Q$ and $a_k \in \Gamma$, there exists (not necessarily always, Because δ is a partial function) some $q_j \in Q$ and some $a_l \in \Gamma$ such that $\delta(q_i\, a_k) = (q_j, a_l, x)$, where x may assume any one of the values 'L', 'R' and 'N'.

The **meaning** of δ (q_i, a_k) = (q_j, a_l, x) is that if q_i is the current state of the TM, and a_k is cell currently under the Head, then TM writes a_l in the cell currently under the Head, enters the state q_j and the *Head moves to the right adjacent cell, if the value of x is R, Head moves to the left adjacent cell, if the value of x is L* and continues scanning the same cell, if the value of x is N.

(v) $q_0 \in Q$, is the initial/start state.

(vi) h $\in$ Q is the 'Halt State', in which the machine stops any further activity.

In order to illustrate the ideas involved, let us consider the following simple examples.

Example 1

Consider the Turing Machine (Q,Σ , ⎾ , δ , q_o, h) defined below that erases all the non-blank symbols on the tape, where the sequence of non-blank symbols does not contain any blank symbol # in-between:

Q= {q_o, h} Σ= {a, b}, ⎾ = {a, b, #}
and the next-move function is defined by the following table:

q: State	**σ: Input Symbol**	**δ (q, σ)** Type equation here. **)**
q_0	A	{q_0, #, R}
q_0	B	{q_0, #, R}
q_0	#	{h, #, N}
H	#	ACCEPT

Next, we consider how to design a Turing Machine to accomplish some computational task through the following example. For this purpose, we need the definition.

A string Accepted by a TM

A string over is said to be accepted by a TM M = (Q, Σ, ⎾, δ, q_0, h) **if** when the string is placed in the left-most cells on the tape of M and TM is started in the initial state q_0 then after a finite number of moves of the TM as determined by δ, TM is in state h (and hence stops any further operations. Further, a string is said to be rejected if under the conditions mentioned above, the TM enters a state $q \neq h$.

3.2.2 INSTANTANEOUS DESCRIPTION AND TRANSITION DIAGRAMS

Instantaneous Description

Some authors use the term *Instantaneous Description* instead of *Total Configuration.*

Initial Configuration: The total configuration at the start of the (Turing) Machine is called the initial configuration.

Halted Configuration: is a configuration whose state component is the Halt state

There are various notations used for denoting the total configuration of a Turing Machine.

Notation 1: *We use the notations, illustrated below through an example:*

Let the TM be in state q_3 scanning the symbol g with the symbols on the tape as follows:

#	#	b	D	a	f	#	G	h	K	#	#	#	#

Then one of the notations is

#	#	b	D	a	f	#	g	h	k	#	#	#	#
							$\uparrow$ q_3						

Notation 2: However, the above being a two-dimensional notation, is sometimes inconvenient. Therefore the following linear notations are frequently used: **(q_3,##bdaf#,g,hk), in which third component of the above 4-component vector, contains the symbol being scanned by the tape head.**

Alternatively, the configuration is also denoted by (q_3,## bdaf#$\underline{g}$hk), where the symbol under the tape head is underscored but two last commas are dropped.

It may be noted that the sequence of blanks after the last non-blank symbol, is not shown in the configuration. The notation may be alternatively written (q_3, w, g, u) where w is the string to the left and u the string to the right respectively of the symbol that is currently being scanned.

In case g is the left-most symbol then we use the empty string e instead of w. Similarly, if g is being currently scanned and there is no non-blank character to the right of g then we use e, the empty string instead of u.

Notation 3: The next notation neither uses parentheses nor commas. Here the state is written just to the left of the symbol currently being scanned by the tape Head. Thus the configuration (q_3, ##bdaf#, g, h, k) is denoted as # # bdaf#$q_3$ghk

Thus if the tape is like

g	w	#	....
$\uparrow$ q_5			

then we may denote the corresponding configuration as (q_5, e, g, u). And, if the tape is like

A	b	c	g	#	#	…
			$\uparrow$ q_6			

Then the configuration is (q_6, abc, g, e) or (q_6, abcg) or alternatively as abcq_6g by the following notation.

3.2.3 Transition Diagrams

In some situations, **graphical** representation of the next-move (partial) function δ of a Turing Machine may give better idea of the behavior of a TM in comparison to the **tabular** representation of δ .

A **Transition Diagram** of the next-move functions δ of a TM is a graphical representation consisting of a finite number of nodes and (directed) labelled arcs between the nodes. Each node represents a state of the TM and a label on an arc from one state (say p) to a state (say q) represents the information about the required **input symbol say x** for the transition from p to q to take place **and the action** on the part of the control of the TM. The action part consists of (i) the symbol say y to be written in the current cell and (ii) the movement of the tape Head.

Then the label of an arc is generally written as x/(y, M) where M is L, R or N.

Example 3.4.2.1

Let M ={Q, Σ, Γ , δ , q_0, h}
Where Q = { q_0, q_1, q_2, h}
Σ = { 0, 1}
Γ = {0, 1, #}
and δ be given by the following table.

	0	1	#
q_0	-	-	(q_2, #, R)
q_1	(q_2, 0, R)	(q_1, #, R)	(h, #, N)
q_2	(q_2, 0, L)	(q_1, 1, R)	(h, #, N)
H	-	-	-

Then, the above Turing Machine may be denoted by the Transition Diagram shown below, where we assume that q_0 is the initial state and h is a final state.

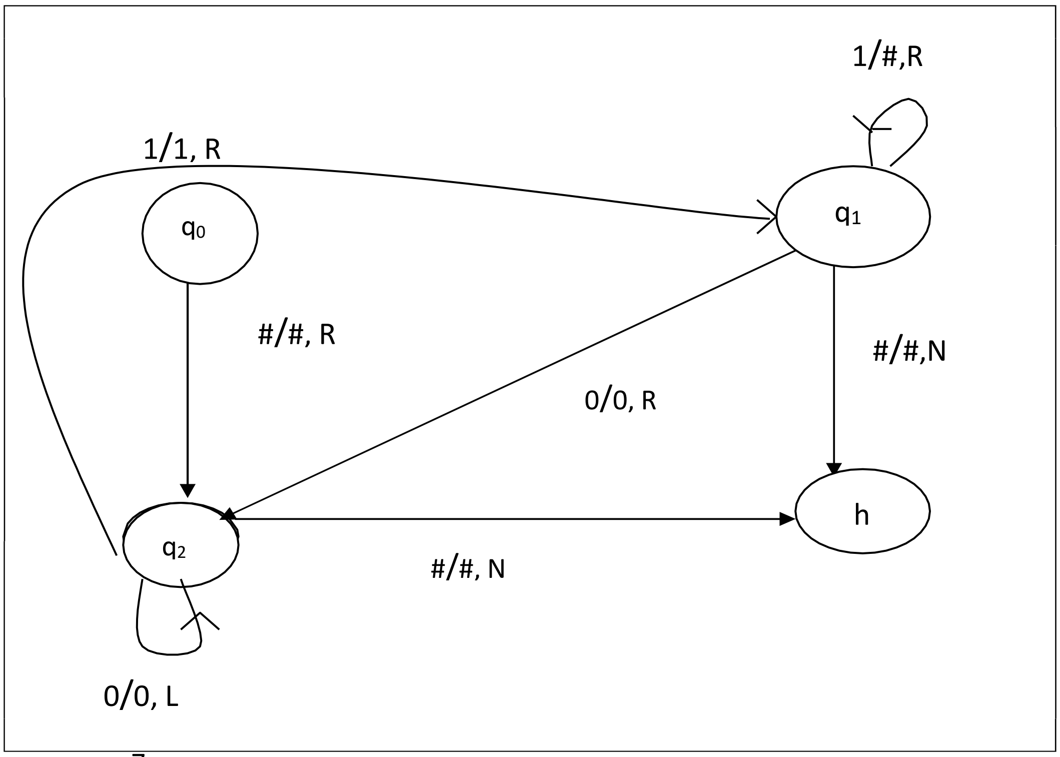


Fig. 3.2: Transition Diagram

3.2.4 SOME FORMAL DEFINITIONS

L(M), the language accepted by the TM M is the set of all finite strings over which are accepted by M.

Definition: Turing Acceptable Language

A language L over some alphabet is said to be Turing Acceptable Language, if there exists a Turing Machine M such that L = L (M)

Definition: Turing Decidable Language
A language over some alphabet is said to be Turing acceptable language if there exists a Turing Machine M such that L= L(M) and M halts on every input.
Remark 3.5.1

A very important fact in respect of Turing acceptability of a string (or a language) needs our attention. The fact has been discussed in very brief in a later section about undecidability. However, we briefly mention it below.

For a TM M and an input string $w \in \Sigma^*$, even after a large number of moves we may not reach the halt state. However, from this we can neither conclude that 'Halt state will be reached in a finite number of moves' nor can we conclude that Halt state will not be reached in a finite number moves.

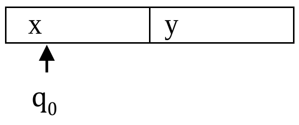
This raises the question of how to decide that an input string w is not accepted by a TM M.

An input string w is said to be *'not accepted'* by a TM M = ($Q, \Sigma, \Gamma, \delta, q_0, h$) if any of the following three cases arise:

(i) There is a configuration of M for which there is no next move i.e., there may be a state and a symbol under the tape head, for which δ does not have a value.

(ii) The tape Head is scanning the left-most cell containing the symbol x and the state of M is say q and δ (x, q) suggests a move to the 'left' of the current cell. However, there is no cell to the left of the left-most cell. Therefore, move is not possible. The potentially resulting situation (can't say exactly configuration) is called **Hanging configuration**.

(iii) The TM on the given input w enters an infinite loop. For example if configuration is as



and we are given
$\delta(q_0, x) = (q_1, x, R)$
and $\delta(q_1, y) = (q_0, y, L)$
Then we are in an **infinite loop**.

3.2.5 OBSERVATIONS

The concept of TM is one of the most important concepts in the theory of Computation. In view of its significance, we discuss a number of issues in respect of TMs through the following remark:.

Turing Machine is not just another computational model, which may be further extended by another still more powerful computational model. It is not only the most powerful computational model known so far but also is conjectured to be the ultimate computational model. In this regard, we state below the

Turing Thesis: ***The power of any computational process is captured within the class of Turing Machines.***

It may be noted that Turing thesis is just a ***conjecture and not a theorem, hence, Turing Thesis*** can not be logically deduced from more elementary facts. However, the conjecture can be shown to be false, if a more powerful computational model is proposed that can recognize all the languages which are recognized by the TM model and also recognizes at least one more language that is not recognized by any TM.
In view of the unsuccessful efforts made in this direction since 1936, when Turing suggested his model, at least at present, it seems to be unlikely to have a more powerful computational model than TM Model.

Check Your Progress-1

Q1 What is the meaning of the following symbols:

(i) Q

(ii) Σ

(iii) Γ

(iv) δ

Q2 Explain what is a Turing Machine?

Q3 How TM is different from Finite Automata?

3.3 NONDETERMINISTRIC TURING MACHINE

Like nondeterministic finite automata, we can also imagine nondeterministic TM (NDTM). ***An* NDTM *is like the standard TM with the difference as described below*.** In Standard TM, *to each* pair of the current state (except the halt state) and the symbol being scanned, *there is a* **unique** triplet comprising of *the next state*, *unique action* in terms of writing a symbol in the cell being scanned and *the motion*, if any, to the right or left. **However, in the case NDTM**, **to each pair** (q, s) with q as current state and s as symbol being scanned, *there may be a* **finite** *set of the triplets* { (q_i, s_i, m_i) : I =1,2,…….} of possible next moves. *This set of triplets may be empty*, i.e. for some particular (q,s) the TM *may not have any next move*. Or alternatively the set {(q_i, s_i, m_i)} may *have more than one* triplet, meaning thereby that the NDTM in the state q and scanning symbols s, has the *alternatives* for next move to choose from the set {(q_i, s_i, m_i)} of next moves.

It can be easily seen that standard TM is a special case of the NDTM in which for each (q,s) the set {(q_i, s_i, }of next moves is a singleton set or empty.

In order to define formally the concept of Non-Deterministic TM (NDTM), and a configuration in NDTM etc, we assume *that the tape is one-way infinite*.

For the extensions of the standard TM, discussed so far, we did not state the full formal definition of each of the extension. We only discussed the definition only relative to the standard TM. Mainly we discussed configurations and partial move function δ for each of the extensions. However, in view of the significant though small, difference in the behaviour of an NDTMs, we provide below full formal definition of NDTM.

Remark 1:

An important point about the definition of NDTM needs to the highlighted. By the definition of δ which maps an element of (q, x) of Q x Γ to a set {(q_i**,** x_i, M_i) **} means that each element (q, x) of Q x Γ has the potential of leading to more than one configurations. In other words, there are various possible routes to a final configuration from one configuration. However, during one computation only one of these possible values (**q_i**,** x_i, M_i) will be associated with (q, x) through δ**. But we can not tell in advance which one out of the ordered triples from the set** {(q_i**,** x_i, M_i)**}**

This is why the adjective Non-Deterministic **is used for this version of the T.M.**

Remark 2:

The set **{(q_i**, **x_i, M_i)**} associated with (q, x) under δ, may be empty. This means there is no possible next move for (q, x), a situation that occurred even in the case of standard TM and other versions discussed so far. This is why δ was called a **partial** function from Q x Γ to Q x Γx**{*L*,*R*,*N*).**

Remark 3:

In the standard TM and the versions discussed before NDTM, we allowed δ as a partial function to Q x Γ x {L, R, N}. In other words, if a value under δ exists
for (q, x) then the value has to be unique, i.e, can be determined. Therefore, the earlier versions are prefixed with the adjective *Deterministic*. The Non- Deterministic form of each of the earlier versions can be obtained by making suitable modifications in the corresponding definitions of δ etc on the lines of modifications suggested in the definition of NDTM from standard TM.

Remark 4:

Proper non-determinism means that at some stage, there are at least two next possible moves. Now, if we are engage two different persons or machines to work out further possible moves according to each of these two moves, the two can work **independent** of each other. **This means Non-Determination allows parallel computations**. This characteristic of Non-**Determinism,** also allows is further computations even if some of the sequences of moves may be locked as there may not be any next moves at some stages.

Definition: An Non-Deterministic Turing Machine
is a sextuple (Q, Σ, Γ, δ, q_o, h) where
Q: Set of States

Σ: Set of input symbols
Γ: Set of tape symbols
q_o: The initial state
h: The halt state and

δ: Q x Γ → **Power set of** (Q x Γ x {L, R, N})

The concept of a configuration *is same as in the case of standard TM. But the concept of 'yields in one step' denoted by* $\vdash_m$ *, has different meaning.* ***Here one configuration may yield more than one configurations.***

We explain these ideas through a suitable example, which also demonstrates the ***advantage of the Non- Deterministic Turing Machine*** *over the standard Turing Machine.* **The advantage is in respect of the relative ease of construction of NDTM**.

Remarks 5

Before coming to the example, showing advantage of an NDTM in solving some problems; we need to understand properly the concept of acceptance of a language by an NDTM. *First of all, let us recall below what is meant by acceptance of a language L by a standard TM M.*

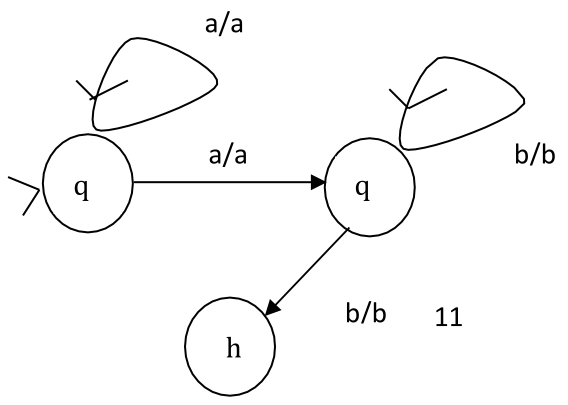
A language L is accepted by a **TM M** ***if each string*** α ∈ L, is acceptable by M. Further a string α is acceptable M, if staring in the initial state q_0 of M, with α as input on the tape of M, if we are able to reach halt state in a finite number of moves,
A characteristic feature of the **standard TM**, in this case, is that if there is to be a sequence of moves from (q_0, α) to a final state, than that sequence might the unique. However **in the case of Non-Deterministic machines**, the halt state may be reached through any one of various permissible sequences of moves. Therefore in this version a string α over the set of input symbols of an NDTM is acceptable by an NDTM M, if **by at least one but by any one** of the sequences of moves halt state is reached from (q_0, α). Now we discuss the example showing advantage of NDTM over standard TM.

Example 1:

Construct an NDTM which accepts the language { $a^n b^m$: n≥1, m ≥1}, i.e., the language of all strings over {a,b}, in which there is at least one a and one b and all a's precede all b's.

Solution: The diagrammatic representation of the required NDTM is as given below:

In the proposed NDTM, as the motion of the head is always to the Right except in the Halt state. Therefore, R is not mentioned in the labels in the diagram below:



where the label i/j on an arc denotes that if symbol in the current cell is i then contents of the cell are to be replaced by j.

Formally the proposed NDTM may be defined as

M={ {q_0, q_1, h}, {a, b}, { a, b, #}, δ, q_o, h }

Where δ is defined as follows:

δ (q_0, a)= {($q_{0,}$ a, R), (q_1, a, R)}

δ (q_0, b)= empty

δ (q_1, a)= empty

δ (q_1, b)= {(q_1, b, R), (h, b, N)}

If the machine has no next move, then it halts without accepting the string.

Remarks 6:

Though we have already mentioned earlier on a number occasions, yet, in view of the significance of non-determinism in designing TMs comparatively *more easily*, we again bring to notice that in the state q_0 on scanning symbol a, the TM may move in any one of the two next possible states viz to q_0 after moving the head to the right or to q_1 (after moving the head to the right). And, if the TM is implemented as a parallel computer then the computer can presume independently both branches initiated by (q_0,a,R) and (q_1,a,R)

3.4 UNDECIDABLE AND HALTING PROBLEMS

UNDECIDABLE PROBLEM

A function g with domain D is said to be computable if there exists some Turing machine whose whose initial configuration is q0 w, i.e., the left-most symbol of the string w being scanned in state q0 and whose final configuration is q f g(w) for some final state qf , i.e., when the machine halts the only non-blank cells on the tape represent the function value g(w).

A function is said to be un-computable if no such machine exists. There may be a Turing machine that can compute f on part of its domain, but we call the function computable only if there is a Turing machine that computes the function on the whole of its domain.

For some problems, we are interested in simpler solution in terms of "yes" or "no". We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

A class of problems with two outputs "yes" or "no" is said to be decidable (solvable) if there exists some definite algorithm which always terminates (halts) with one of two outputs "yes" or "no". Otherwise, the class of problems is said to be undecidable (unsolvable).

THE HALTING PROBLEM

There are many problems which are not computable. But, we start with a problem which is important and that at the same time gives us a platform for developing later results. One such problem is the halting problem. Algorithms may contain loops that may be infinite or finite in length. The amount of work done in an algorithm usually depends on the data input. Algorithms may consist of various numbers of loops, nested or in sequence. Informally, the **Halting problem** can be put as:

Given a Turing machine M and an input *w* to the machine M, determine if the machine M will eventually halt when it is given input *w*.

Trial solution: Just run the machine M with the given input *w*.

- If the machine M halts, we know the machine halts.
- But if the machine doesn't halt in a reasonable amount of time, we cannot conclude that it won't halt. Maybe we didn't wait long enough.

What we need is an algorithm that can determine the correct answer for any M and *w* by performing some analysis on the machine's description and the input. But, it is shown by Alan Turing that no such algorithm exists.

3.5 COMPLEXITY

In the previous unit, we introduced you to the fact that there are a large number of problems which cannot be solved by algorithmic means and discussed a number of issues about such problems.

The advantage of such a study is our becoming aware of the fact that in stead of attempting to write an algorithm for every problem that we are required to solve using a computer, we should first study the essential nature of the problem. In case the problem under consideration is not solvable by algorithmic means, we may adopt other computational techniques including use of heuristics, numerical and/or statistical techniques. Even out of problems, which though theoretically have algorithmic solutions, yet require such large amount of resources, that this type of problems are designated as *infeasible* for the purpose of computational solution. Out of the problems, which are *feasibly* solvable, there are problems each of which may have more than one algorithms to solve the problem. For us, it is desirable to know which one is better among the available ones. For example, we can use the algorithms viz, Bubble sort, Insertion sort, Heapsort and Quicksort, for sorting a list of numbers. Their designs are different but the outcome is the same for all, for a given list of numbers. As, there are more than one algorithms available to us to sort a list of numbers, it is natural for us to think of using the algorithm which solves a particular sorting problem, in some way better than the others. In context of practical disciplines like computer applications, an *efficient* solution is generally taken as a *better* solution. Efficiency of an algorithm can be considered in terms of the efficient use of computer resources, such as processor time and memory space used. In addition to the efficiency of execution of algorithms, other factors like time (taken by a team of software engineers and/or programmers) *required for developing algorithms* and *reliability* may also be taken into consideration as factors towards overall efficiency of an algorithm.

However, most of the time, in respect of efficiency of algorithms, we are only concerned with the time and space requirements of execution of algorithms.

In this unit, we will discuss the issue of efficiency of computation of an algorithm in terms of the *amount of time* used in its execution. On the basis of analysis of an algorithm, the amount of time that is estimated to be required in executing an algorithm, will be referred to as the **time complexity** of the algorithm. The time complexity of an algorithm is measured in terms of some (basic) **time unit** (not second or nano-second). Generally, time taken in executing one move of a TM, is taken as (basic) time unit for the purpose. or, alternatively, time taken in executing some elementary operation like addition, is taken as one unit. More complex operations like multiplication etc, are assumed to require an integral number of basic units. As mentioned earlier, given many algorithms (solutions) for solving a problem, we would like to choose the most efficient algorithm from amongst the available ones. For comparing efficiencies of algorithms, that solve a particular problem, time complexities of algorithms are considered as functions of the sizes of the problems (to be discussed). The *time complexity functions of the algorithms are compared in terms of their growth rates (to be defined)* as growth rates are considered important measures of *comparative efficiencies*.

The concept of the size of a problem, though a fundamental one, yet is difficult to define precisely. Generally, the size of a problem, is measured in terms of the size of the *input*. The concept of the size of an input of a problem may be explained informally through examples. In the case of multiplication of two nxn (squares) matrices, the size of the problem may be taken as n^2, i.e, the number of elements in each matrix to be multiplied. For problems involving polynomials, the degrees of the polynomials may be taken as measure of the sizes of the problems.

Also, we may have an intuitive idea about the term **growth rate** *and its significance* in the comparative study of algorithms that can be designed to solve problems. For the time being, in stead of attempting a formal definition, we illustrate the concept of *growth rate of time complexity function* of an algorithm and its significance through the following example.

Let us consider two algorithms to solve a problem P, having time-complexities respectively as $f_1(n) = 1000n^2$ and $f_2(n) = 5n^4$, where size of the problem is assumed to be n. Then

$f_1 (n) \geq f_2 (n)$ for $n \leq 14$ and

$f_1 (n) \leq f_2 (n)$ for $n \geq 15$.

Also, the increase in the ratio $(f_2 (n)/f_1 (n))$ is faster than increase in n. Thus, informally, growth rate of $f_2 (n)$ is more than the growth rate of $f_1 (n)$. In one sense, the algorithm having time complexity $f_2 (n)$ is *inferior* to the algorithm having time complexity $f_1(n)$ as growth rate of $f_2(n)$ is faster than that of $f_1 (n)$.

For a problem, a solution with time complexity which can be expressed as a polynomial of the size of the problem, is considered to have an **efficient solution**. Unfortunately, not many problems that arise in practice, admit any efficient algorithms, as these problems can be solved, if at all, by only non-polynomial time algorithms. A problem which does not have any (known) polynomial time algorithm is called an **intractable** problem.

At this stage, it is important to be aware of the following relevant facts

(i) **A non-polynomial function need not always be exponential:**. For example, the function f(n) = $n\log_2 n$ is neither polynomial function nor exponential function of n, but, somewhere between the two but n^log_2(n) is a polynomial function

(ii) **The term *solution* in its general form: need not be an algorithm.** If by tossing a coin, we get the correct answer to each instance of a problem, then the process of tossing the coin and getting answers constitutes a solution. But, the process is not an algorithm. Similarly, we solve problems based on **heuristics**, i.e, good guesses which, generally but not necessarily always, lead to solutions. All such cases of solutions are not algorithms, or algorithmic solutions. To be more explicit, by an algorithmic solution A of a problem L (*considered as a language*) from a problem domain Σ*, we mean that among other conditions, the following are satisfied:

(a) A is a step-by-step method in which for each instance of the problem, there is a definite sequence of execution steps (*not involving any guesswork*).

(b) A *terminates* for each $x \in \Sigma^*$, irrespective of whether $x \in L$ or $x \notin L$.

In this sense of algorithmic solution, only a **solution by a Deterministic TM** is called an **algorithm**. A solution by a **Non-Deterministic TM may not be an algorithm.**

(iii) However, for every NTM solution, there is a Deterministic TM (DTM) solution of a problem. Therefore, if there is an NTM solution of a problem, then there is an algorithmic solution of the problem. However, *the symmetry may end here*.

The *computational equivalence* of Deterministic and Non-Deterministic TMs does not state or guarantee any *equivalence in respect of requirement of resources* like time and space by the Deterministic and Non-Deterministic models of TM, for solving a (solvable) problem. To be more precise, if a problem is solvable in polynomial-time by a Non-Deterministic Turing Machine, then it is, of course, *guaranteed* that there is a deterministic TM that solves the problem, but *it is not guaranteed* that there exists a Deterministic TM that solves the problem *in polynomial time*. Rather, **this fact forms the basis for one of the deepest open questions of Mathematics, which is stated as 'whether P = NP?'**(P and NP to be defined soon).

The question put in simpler language means: Is it possible to design a Deterministic TM to solve a problem in polynomial time, for which, a Non-Deterministic TM that solves the problem in polynomial time, has already been designed?

We summarize the above discussion from the intractable problem's definition onward. Let us begin with definitions of the notions of P and NP.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.

NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e, to a polynomial time DTM.

We can define p and NP to be the classes of languages because problems from graph theory, combinatorics can often be formulated as language recognition problems. Consider a problem which requires an answer in form of " Yes" or "No" for each instance. Each instance of a problem can be encoded as a string and reformulate the problem as one of recognizing the language consisting of all the strings representing those instance of the problem whose answer is "Yes".

With this logic P and NP classes of complexities can be defined as classes of languages:

Definition

P = { L| L can be decided or accepted by a DTM(Deterministic TM) in polynomial time}

NP = { L| L can be decided or accepted by a Nondeterministic TM in polynomial time}

NP is a set of decision problems (with Yes or No answer) that can be solved by NDTM in polynomial time

Thus starting with two distinct classes of problems, viz, **tractable** problems and **intractable** problems, we introduced two classes of problems called **P and NP**. Some interesting relations known about these classes are:

(i) P = set of tractable problems

(ii) $P \subseteq NP$.

(The relation (ii) above simply follows from the fact that every Deterministic TM is a special case of a Non-Deterministic TM).

Check your Progress-2

Q1 What is Nondeterministic Turing Machine?

Q2 Define P and NP Complexity Classes.

3.6 SUMMARY

In this unit, after giving the informal idea of what a Turing machine is, the concept is formally defined and illustrated through an example. A Nondeterministic TM was introduced next and also explained how it is different from a standard TM. Besides TM and NDTM, P and NP classes of complexities were defined.

3.7 Solutions/Answers

Check Your Progress-1

Q1 What is the meaning of the following symbols:

(i) Q

(ii) Σ

(iii) Γ

(iv) δ

Ans1. (i) Q is the finite set of states,

(ii) Σ is the finite set of non-blank information symbols,

(iii) Γ is the set of tape symbols, including the blank symbol #

(iv) δ is the **next-move** partial function from Q x Γ to Q x Γ x {L, R, N},

where '**L**' denotes the tape Head moves to the left adjacent cell, '**R**' denotes tape Head moves to the Right adjacent cell and '**N**' denotes Head does not move, **i.e**., continues scanning the same cell.

Q2 Explain what is a Turing Machine?

Ans. 2: TM defines an abstract machine/mathematical model that consists of an infinite length tape divided into cells for storing input symbols and a head which scans and reads the input. If the TM reaches the final state, the input string is accepted, otherwise it is rejected. TMs are considered to be the most important formal models in the study of Computer Science.

Q3 How TM is different from Finite Automata?

Ans.3 The *Finite Automata* is used only as **accepting devices** for languages in the sense that the automata, when given an input string from a language, tells whether the string is acceptable or not. ***The Turing Machines are designed to play at least the following three different roles:***

(i) **As accepting devices for languages**, similar to the role played by FAs .

(ii) **As a computer of functions**. In this role, a TM represents a particular function (say the SQUARE function which gives as output the square of the integer given as input). Initial input is treated as representing an argument of the function. And the (final) string on the tape when the TM enters the Halt State is treated as representative of the value obtained by an application of the function to the argument represented by the initial string.

(iii) **As an enumerator of strings of a language** that outputs the strings of a language, one at a time, in some systematic order, i.e, as a list.

Check your Progress-2

Q1 What is Nondeterministic Turing Machine?

Ans.1: In Standard TM, *to each* pair of the current state (except the halt state) and the symbol being scanned, *there is a* **unique** triplet comprising of *the next state*, *unique action* in terms of writing a symbol in the cell being scanned and *the motion*, if any, to the right or left. **However, in the case NDTM**, **to each pair** (q, s) with q as current state and s as symbol being scanned, *there may be a* **finite** *set of the triplets* { (q_i, s_i, m_i) : I =1,2,…….} of possible next moves.

Q2 Define P and NP Complexity Classes.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.
NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e, to a polynomial time DTM.

We can define p and NP to be the classes of languages because problems from graph theory, combinatorics can often be formulated as language recognition problems. Consider a problem which requires an answer in form of " Yes" or "No" for each instance. Each instance of a problem can be encoded as a string and reformulate the problem as one of recognizing the language consisting of all the strings representing those instance of the problem whose answer is "Yes".

With this logic P and NP classes of complexities can be defined as classes of languages:

Definition

P = { L| L can be decided or accepted by a DTM(Deterministic TM) in polynomial time}

NP = { L| L can be decided or accepted by a Nondeterministic TM in polynomial time}

NP is a set of decision problems (with Yes or No answer) that can be solved by NDTM in polynomial time

3.7 SOLUTIONS AND ANSWERS