AUTOMATA THEORY AND COMPUTAILITY (18CS54)
Module-5 Notes

5.1 ALGORITHMS:

An algorithm is a finite sequence of instructions that terminates after a finite number of steps for any input. In 1900-Mathematician David Hilbert identified 23 mathematical problems as a challenge for future mathematicians; only ten of the problems have been solved so far.

Hilbert's tenth problem was to devise 'a process according to which

- Hilbert's tenth problem was to devise 'a process according to which it can be determined by a finite number of operations'- which means an algorithm

Algorithm Vs Procedure:

- A ***procedure*** for solving a problem is a finite sequence of instructions which can be mechanically carried out given any input.
- An ***algorithm*** is a procedure that **terminates** after a finite number of steps for any input.

Recursive set vs Recursively enumerable set (or function)

- A set X is **recursive** if we have an **algorithm** to determine whether a given element belongs to X or not.
- **A recursively enumerable** set is a set X for which we have **procedure** to determine whether a given element belongs to X or not.

5.2 DECIDABILITY:

The recursive function and recursively enumerable function terms can also be defined in terms of Turing machine and how it halts for a given function.

A Turing machines Halts in two cases :

1. When a Turing machine reaches a final state, it 'halts.'
2. We can also say that a Turing machine M halts when M reaches a state q and a current symbol a to be scanned so that δ(q, a) is undefined.

There are TMs that never halt on some inputs in any one of these ways. So, we make a distinction between the languages accepted by a TM that halts on all input strings and a TM that never halts on some input strings.

Recursive function and recursively enumerable functions defined using TM:

- Language L ⊆ Σ*is **recursively enumerable** if there exists a TM M. such that L =T(M)
- A language L ⊆ Σ* is **recursive** if there exists some TM M that satisfies the following two conditions.

  (i) If w ∈ L then M accepts w (that is, M reaches an accepting state on processing w) and halts.

(ii) If w ∉ L then M eventually halts without reaching an accepting state. i.e TM halts in either case.

Decidable: A problem with two answers (Yes/No) is decidable if the corresponding language is recursive. In this case, the language L is also called **decidable.**

Partially Decidable: A language is L is partially decidable if 'L' is recursively enumerable language.

Undecidable: A problem/ language is undecidable if it is not decidable. If is is not even partially decidable then there exist no Turing machine to accept that language.

Language	Turing Machine
Recursive Language / Decidable Language	**Turing Machine always halts (Accept/Reject)**
Recursively Enumerable Language / Partially Decidable Language	**Turing Machine Halts (Accept)/ May or may not halt (Reject)**
Undecidable Language	**No Turing machine exists**

5.3 DECIDABLE LANGUAGES:

5.3.1

Any language accepted by a DFA is decidable

Or

All regular languages are decidable.

Or

If A_{DFA} = *{(B,w)* I *B* accepts the input string *w} then*

Theorem: A_{DFA} is decidable.

All three definitions are same

Proof: To prove the theorem. we have to construct a TM that always halts and also accepts $A_{DFA.}$

- We define a TM *M* as follows:
    1. Let *B* be a DFA and w an input string. *(B,* w) is an input for the Turing machine *M.*
    2. Simulate *B* and input w in the TM *M.*
    3. If the simulation ends in an accepting state of *B.* then *M* accepts *w.* If it ends in a nonaccepting state of *B,* then *M* rejects w.

The input *(B,* w) for M is represented by representing the five components $(Q,\Sigma, \delta, q_0, F\}$by strings of Σ * and input string w ⊆ Σ*, *M* checks whether *(B.* w) is a valid input. If not. it rejects *(B, w)* and halts. If *(B,* w)is a valid input M writes the initial state *qo* and the leftmost input symbol of *w.* It updates the state using δ and then reads the next symbol in *w.*

5.3.2

Any languages given by a CFG is decidable

Or

All context free languages are decidable.

Or

A_{CFG} = {(G, w) | the context-free grammar G accepts the input string *w}*

All three definitions are same

Theorem: A_{CFG} is decidable.

Proof: We convert a CFG into Chomsky Normal form. Then any derivation of w of length k requires 2k - 1 steps if the grammar is in CNF. So, for checking whether the input string w of length k is in L(G), it is enough to check derivations in 2k - 1 steps. We know that there are only finitely many derivations in 2k - 1 steps. Now we design a TM M that halts as follows.

1. Let G be a CFG in Chomsky normal form and w an input string.(G, w) is an input for M.
2. If k = 0, list all the single-step derivations. If k ≠ 0, list all the derivations with 2k - 1 steps.
3. If any of the derivations in step 2 generates the given string w, accepts (G, w). Otherwise M rejects.

The next step of the derivation is obtained by the production to be applied.

5.4 UNDECIDABLE LANGUAGES

5.4.1 Existence of undecidable languages are proved by proving existence of languages that are not recursively enumerable.

Theorem: There exists a language over Σ that is not recursively enumerable.

Proof: A language L is recursively enumerable if there exists a TM M such that L =T(M). As L is finite, Σ* is countable, that is, there exists a one-to-one correspondence between Σ * and N. As a Turing machine M is a 7-tuple (Q. Σ, Γ, δ, q_0, b, F) and each member of the 7-tuple is a finite set. M can be encoded as a string. So the set I of all TMs is countable.

Let L be the set of all languages over Σ. Then a member of L is a subset of Σ. We show that L is uncountable (that is, an infinite set not in one-to correspondence with N).

We prove this by contradiction. If L were countable then L can be written as a sequence {L1, L2, L3, ... }. We write Σ * as a sequence {w1,w2,w3….}. So L1 can be represented as an infinite binary sequence $X_{i1}X_{i2}X_{i3}$ .. where

$$x_{ij} = \begin{cases} 1 & \text{if } w_j \in L_i \\ 0 & \text{otherwise} \end{cases}$$

Using this representation, we write L_i as an infinite binary sequence.

$$\begin{aligned} L_1 &: x_{11}x_{12}x_{13} \ldots x_{1j} \ldots \\ L_2 &: x_{21}x_{22}x_{23} \ldots x_{2j} \ldots \\ &\vdots \qquad\qquad \vdots \\ L_i &: x_{i1}x_{i2}x_{i3} \ldots x_{ij} \ldots \end{aligned}$$

We define a subset L of Σ * by the binary sequence Y_1,Y_2,Y_3 …where $Y_i = 1 - X_{ii}$. If X_{ii} =0, Y_i =1 and if X_{ii} =I, Y_i=0. Thus according to our assumption the subset L of Σ* represented by the infinite binary sequence Y_1,Y_2,Y_3 ... should be L_k for some natural number k. But $L \neq L_k$ since $w_k \in L$ if and only if $w_k \notin L$.

This contradicts our assumption that L is countable. This proves the existence of a language over Σ that is not recursively enumerable.

5.4.2 Universal Turing Machine

A Universal Turing machine is a Turing machine that simulates an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input to that machine from its own tape.

Language A_{TM} = {(M, w)|The TM M accepts w} is Turing recognizable

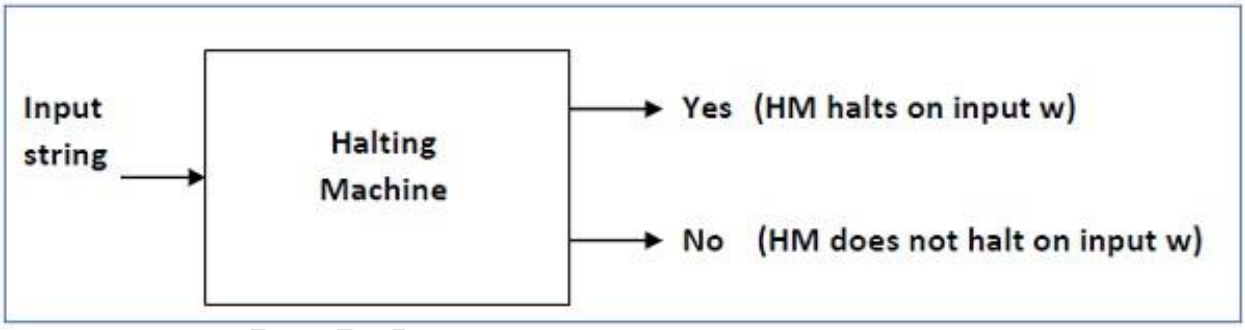
5.4.3 Undecidable problems:

1. Halting Problem
2. Post Correspondence problem

Halting Problem:

Halting problem statement: Is it possible to tell whether a given machine will halt for some given input? It is analogous to asking Given a program and input to the program, is it possible to tell if the program terminates?

Proving Halting problem is undecidable: At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a Halting machine that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine −

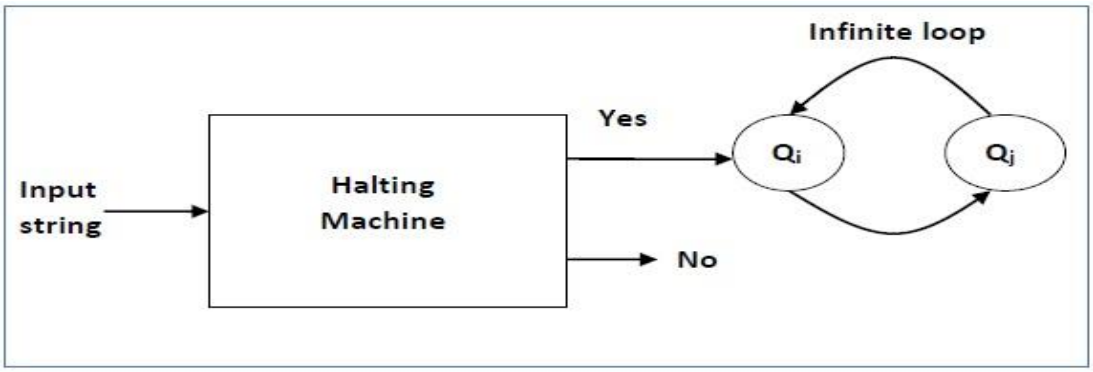


Now we will design an inverted halting machine (HM)' as –

If H returns YES, then loop forever.

If H returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine' –



A machine (HM)2 which input itself is constructed as follows −

If (HM)2 halts on input, loop forever.

Else, halt.

Here, we have got a contradiction. Hence, the halting problem is undecidable.

Post Correspondence Problem:

The Post Correspondence Problem (PCP) was first introduced by Emil Post in 1946. Later, the problem was found to have many applications in the theory of formal languages. The problem over an alphabet Σ: belongs to a class of yes/no problems and is stated as follows:

Consider the two lists x =(x_1 ….. x_n), y = (y_1 … y_n) of nonempty strings over an alphabet Σ= {0, 1}. The PCP is to determine whether or not there exist

i_I …… i_m, where 1<=i_j<= n, such that

$$xi_1 \ldots\ldots xi_m = yi_1 \ldots\ldots yi_m$$

Note: The indices i_j ’s need not be distinct and m may be greater than n Also, if there exists a solution to PCP, there exist infinitely many solutions.

Example problem: Does the PCP with two lists

x = (b, bab^3, ba) and

y = (b^3, ba, a)

have a solution?

bab^3	b	b	ba	=	ba	b^3	b^3	a
x_2	x_1	x_1	x_3		y_2	y_1	y_1	y_3

- The PCP may be thought of as a game of dominoes in the following way: Let each domino contain some Xi in the upper-half, and the corresponding substring of Y in the lower-half. A typical domino is shown below:

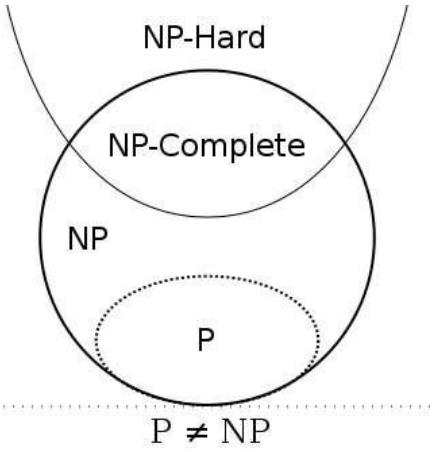
x_i	upper-half
y_i	lower-half

- The PCP is equivalent to placing the dominoes one after another as a sequence with repetitions allowed. To win the game, the same string should appear in the upper-half and in the lower-half. So winning the game is equivalent to a solution of the PCP.

5.5 COMPLEXITY

5.5.1 Class of problems:

- **P** stands for the class of problems that can be solved by a deterministic algorithm (i.e. by a Turing machine that halts) in polynomial time:
- **NP** stands for the class of problems that can be solved by a nondeterministic algorithm (that is, by a nondeterministic TM) in polynomial time;
- Another important class is the class of **NP-complete** problems which is a subclass of NP.



Growth Rate Of Functions

The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. The rate of growth of an algorithm helps to determine how much more time would be required to complete the task on increasing the input. Growth functions can be represented in term of

n, n being the input size and growth rates of two functions solving the same problem are compared using asymptotic notations such big oh(O), big theta(Θ) and big omega(Ω).

When n increases, each of n, n^n, 2^n increases. But a comparison of these functions for specific values of n will indicate the vast difference between the growth rate of these functions.

TABLE 12.1 Growth Rate of Polynomial and Exponential Functions

n	$f(n) = n^2$	$g(n) = n^2 + 3n + 9$	$q(n) = 2^n$
1	1	13	2
5	25	49	32
10	100	139	1024
50	2500	2659	$(1.13)10^{15}$
100	10000	10309	$(1.27)10^{30}$
1000	1000000	1003009	$(1.07)10^{301}$

Classes of P,NP and NP-Complete in terms TM

A Turing machine M is said to be of time complexity T(n) if the following holds: Given an input w of length n. M halts after making at most T(n) moves. M eventually halts.

CLASS P: A language L is in **class P** if there exists some polynomial T(n) such that L = T(M) for some deterministic TM M of time complexity T(n).

CLASS NP: A language L is in **class NP** if there is a nondeterministic TM M and a polynomial time complexity T(n) such that L = T(M) and M executes at most T(n) moves for every input w of length n.

CLASS NP-COMPLETE: Class NP complete can be defined only by knowing the process of **reduction**.

- Let P1 and P2 be two problems. A reduction from P1 to P2 is an algorithm which converts an instance of P1 to an instance of P2. If the time taken by the algorithm is a polynomial p(n), n being the length of the input of P1. then the reduction is called a polynomial reduction P1 to P2.
- Example: LCM to GCD

  LCM(24, 36) = 24*36 / GCD(24, 36) = 24*36 / 12 = 72

Let L be a language or problem in NP. Then L is **NP-complete** if

1. L is in NP

2. For every language L' in NP there exists a polynomial-time reduction of L' to L

5.6 QUANTUM COMPUTING:

A bit (a 0 or a 1) is the fundamental concept of classical computation and information. A classical computer is built from an electronic circuit containing wires and logical gates. Quantum bits and quantum circuits are analogous to bits and (classical) circuits.

A **quantum bit**, or simply **qubit** can be described mathematically as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

$|0\rangle$ and $|1\rangle$ they are pronounced "ket 0" and "ket 1", respectively. Unlike a classical bit, a qubit can be in infinite number of states other than $|0\rangle$ and $|1\rangle$. It can be in a state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. The 0 and 1are called the computational basis states and $|\psi\rangle$ is called a superposition. We can call $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ a quantum state.

Multiple qubits can be defined in a similar way. For example. a two-qubit system has four computational basis states, $|00\rangle$, $|01\rangle$,$|10\rangle$ and $|11\rangle$ and quantum states $|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{00}|01\rangle + \alpha_{00}|10\rangle + \alpha_{00}|11\rangle$ with $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$.

A quantum computer is a system built from quantum circuits, containing wires and elementary quantum gates, to carry out manipulation of quantum information.

Ex of quantum gate: NOT gate

The classical NOT gate interchanges 0 and 1. In the case of the qubit the NOT gate, $\alpha|0\rangle + \beta|1\rangle$ is changed to $\alpha|1\rangle + \beta|0\rangle$. The action of the qubit NOT gate is linear on two-dimensional complex vector spaces. So the qubit NOT gate can be described by

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

5.7 CHURCH TURING THESIS:

Church-Turing thesis: **"Any algorithm that can be performed on any computing machine can be performed on a Turing machine as well"**

Miniaturization of chips has increased the power of the computer. The growth of computer power is now described by Moore's law. which states that the computer power will double for constant cost once in every two years. Now it is felt that a limit to this doubling power will be reached in two or three decades. since the quantum effects will begin to interfere in the functioning of electronic devices as they are made smaller and smaller. So efforts are on to provide a theory of quantum computation which will compensate for the possible failure of the Moore's law.

As an algorithm requiring polynomial time was considered as an efficient algorithm. a strengthened version of the Church-Turing thesis was enunciated. Any algorithmic process can be simulated efficiently by a Turing machine. But a challenge to the strong Church-Turing thesis arose from analog computation. Certain types of analog computers solved some problems efficiently whereas these problems had no efficient solution on a Turing machine. But when the presence of noise was taken into account, the power of the analog computers disappeared. In mid-1970s. Robert Soiovay and Volker Strassen gave a randomized algorithm for testing the primality of a number. A deterministic polynomial algorithm was given by Manindra Agrawal. Neeraj Kayal and Nitein Saxena of IIT Kanpur in 2003.This led to the modification of the Church thesis.

Strong Church-Turing Thesis: "An algorithmic process can be simulated efficiently using a nondeterministic Turing machine."

5.8 DEFINING THE SYNTAX OF PROGRAMMING LANGUAGES

Context Free Grammars provide the basis for defining most of the syntax of most programming languages. It is not enough to design a programming language but it is also necessary to produce an unambiguous language specification.

5.8.1 BNF (Backus Naur Form/Backus Normal Form)

BNF has served as the basis for the description of early programming languages like ALGOL languages without any ambiguity, as well as others. The BNF language that Backus and Naur used exploited these special symbols:

- ::= corresponds to ->
- I means or
- < > surround the names of the nonterminal symbols.

A grammar for arithmetic expression will be written ads below in BNF

<E> ::= <E> + <T> I <T>
<T> ::- <T> * <F> I <F>
<F> ::= id I (<E>)

Since its introduction in 1960, BNF has become the standard tool for describing the context-free part of the syntax of programming languages, as well as a variety of other formal languages: query languages, markup languages, and so forth. In later years. it has been extended both to make better use of the larger character codes that are now in widespread use and to make specifications more concise and easier to read. For example, modern versions of BNF

- often use ->instead of::=
- provide a convenient notation for indicating optional constituents. One approach is to use the subscript 'opt'·
- Another is to declare square brackets to be metacharacters that surround optional constituents. The following rules illustrate three wavs to say the same thing:

S-T|ε
S->T_{opt}
S->[T]

These various dialects are called Extended BNF or EBNF.

5.8.2 Railroad Diagram

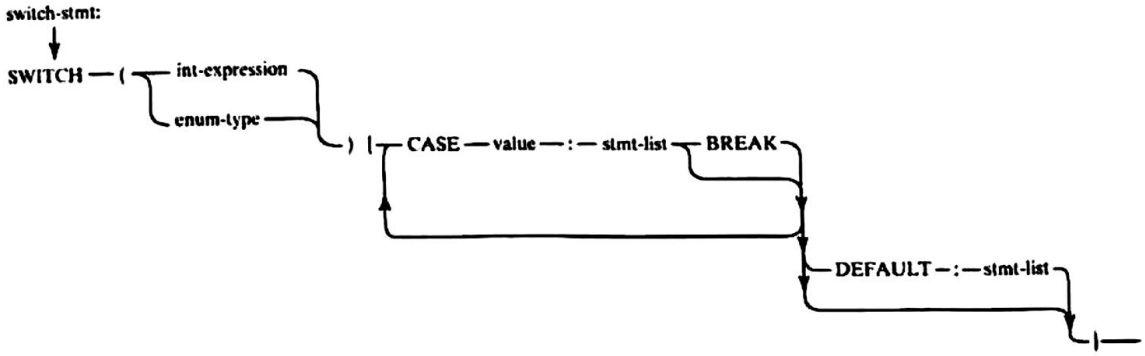
Sometimes more modem definitions look superficially different from BNF. And hence other notations have been developed over the years like Railroad Diagrams (also called syntax diagrams or railway tracks) are graphical renditions of the rules of a context-free grammar. Railroad diagrams have the same expressive power as BNF. but they are sometimes easier to read.

Ex: A Railroad Diagram for a Switch Statement

BNF for switch:

<switch-statement> ::=SWITCH (<int- expression> | <enum-type>}) {<case-list>}
<case-list> ::- <case-body> <default-clause>$_{opt}$
<case-body> ::= <case-item> l <case-item> <case-body>
<case-item> : :=CASE <value> : <Stmt- list> BREAK$_{opt}$
<default-clause>::= DEFAULT: <Stmt-list>

Railroad diagram for above switch statement



Terminal strings are shown in upper case. Non terminals are shown in lower case. To generate a switch statement, we follow the lines and arrows, starting from switch-stmt. The word SWITCH appears first, followed by '('Then one of the two alternative paths is chosen. They converge and then the symbols ')' and '{'appear. There must be at least one case alternative, but, when it is complete, the path may return for more. Both BREAK and DEFAULT clause are optional since in both cases there are detours around them.

5.9 APPLICATION OF ATC IN SECURITY

5.9.1 Physical Security Systems as FSMs

Some intrusion-detection systems arc complex: They may, for example, divide the region that is being protected into multiple zones. Then the state of each zone may be partially or completely independent of the states or the other zones. But we can easily see the essential structure of such systems by considering the simple DFSM shown in Figure

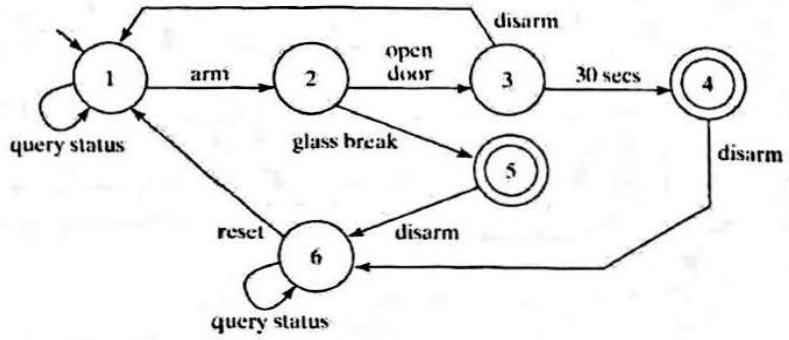
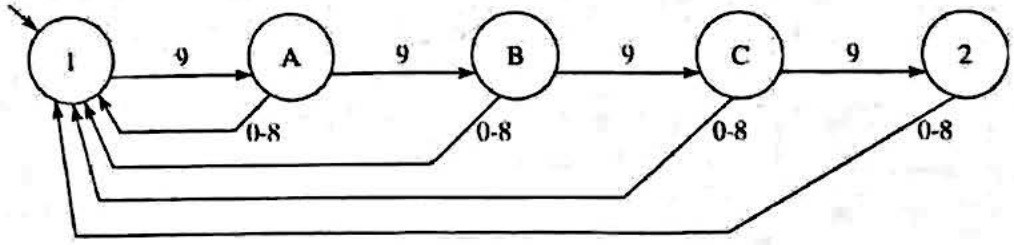


Fig. A simple physical security system as FSM

A realistic system has many more states. For example, suppose that alarm codes consist of four digits. Then the single transition from state 1 to state 2 is actually a sequence of four transitions, one for each digit that must be typed in order to arm the system. Suppose. for example, that the alarm code is 9999. Then we can describe the code-entering fragment of the system as the DFSM shown in Figure



QUESTION BANK FOR MODULE-5

1. Discuss the following
    a. Recursively enumerable language
    b. Post Correspondence problem — 10M (SEP- 2020)

2. Write short Note on the following
    a. Quantum Computer
    b. Class NP
    c. Church Turing Thesis
    d. Model of Linear Bounded Automata (mod4)
    e. Halting Problem of Turing machine — 20M (SEP-2020)

3. Write short notes on
    a. Undecidable languages
    b. Halting problem of TM'
    c. Post correspondence problem
    d. Church-Turing Thesis — 16M (DEC- 2019)

4. Defined the following terms:
    a. Recursively enumerable language
    b. Decidable Language — 06M (DEC -2018)

5. Explain with example
    a. Decidability
    b. Decidable languages
    c. Undecidable languages — 10M (JAN-2020)
6. Prove that there exists a language over Σ that is not recursively enumerable.
7. Explain the application of CFG in defining the
    a. syntax of programming languages.
    b. Physical security system
8. Define the following terms
    a. BNF
    b. Railroad Diagram