Variations of Turing Machine:-

⇒ There are a number of variations in which slightly different conventions are followed with regard to starting configuration, permissible moves, protocals followed to accept strings and so forth.

1. Multitape Turing Machine:-

⇒ A multitape turing machine consists of a finite control with k tape heads and k tapes.

⇒ Each tape is infinite in both directions.

⇒ Each tape is divided into cells and each cell can hold any symbol of the finite tape alphabet.

⇒ Thus, in a multi-tape Turing machine each tape is controlled by its independent Read/write head.

⇒ On a single move, depending upon the state of finite control and the symbol scanned by each of the tape heads, the multiple turing machine does the following.

1. change state i.e enters a new state.
2. Print a new symbol on each of cells scanned by its tape heads.
3. Each of the tape head makes a move, which can be either left right or stationary and the head moves independently.

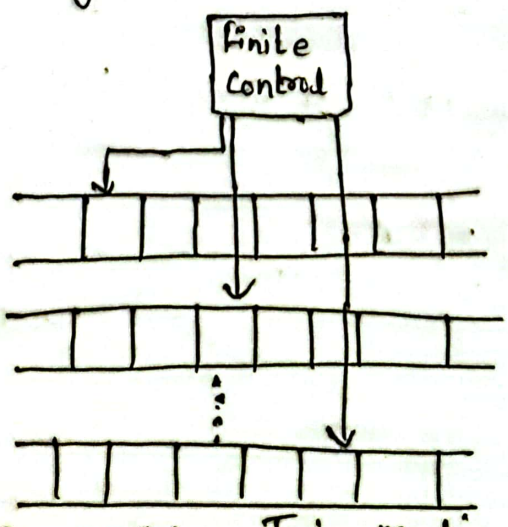


Fig. Multi tape Turing Machine.

2. Non-deterministic Turing Machine :-

⇒ A non-deterministic Turing Machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where $Q, \Sigma, \Gamma, q_0, F$ have same meaning as in basic Turing machine but transition function is defined as

$$\delta : Q \times \Gamma \rightarrow \text{Power set of } Q \times \Gamma \times \{L, R\}$$

$$\text{or } Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R\}}$$

⇒ It can be seen as a machine that has the ability to replicate itself whenever necessary.

⇒ When more than one move is possible, the machine produces as many replicas as needed and gives each replica the task of carrying out one of the alternatives.

3. Multi-dimensional Turing Machine :-

⇒ A Turing machine is said to be multi-dimensional turing machine if its tapes can be viewed as extending infinitely in more than one dimension.

⇒ The device has the usual finite control but the tape consists of a k-dimensional array of cells infinite in all 2k directions for some fixed k.

⇒ Depending upon the state and the scanned symbol, the device changes states, prints a new symbol and move its tape head in one of the 2k directions either positively or negatively along one of k-axis.

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\}$$

where U and D specify movement of R/W head in up and down directions.

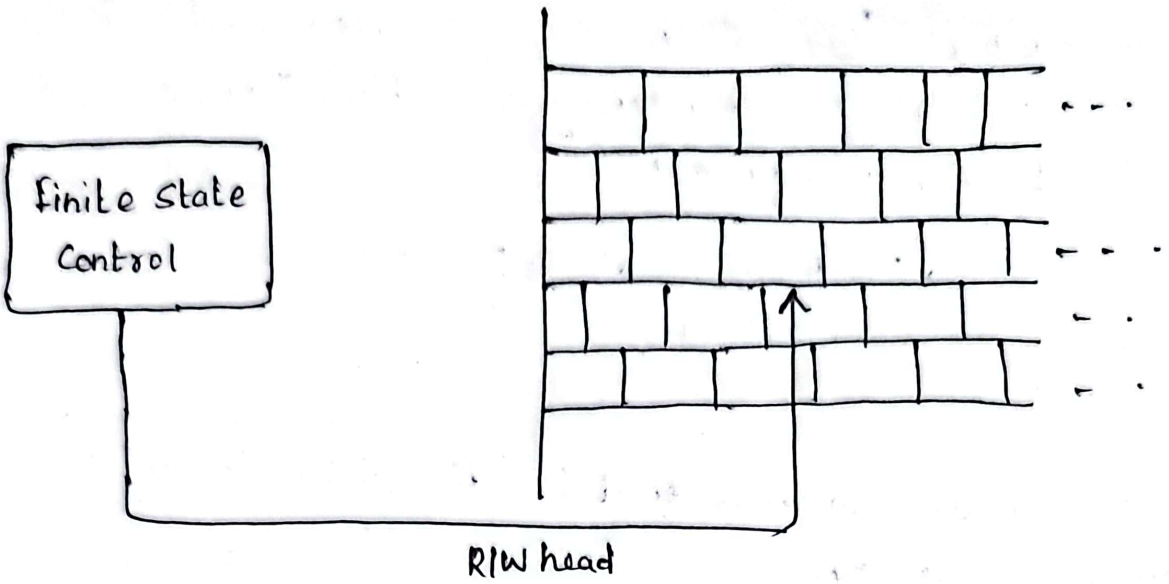


fig: Two dimensional head

4 Multi-head Turing Machine

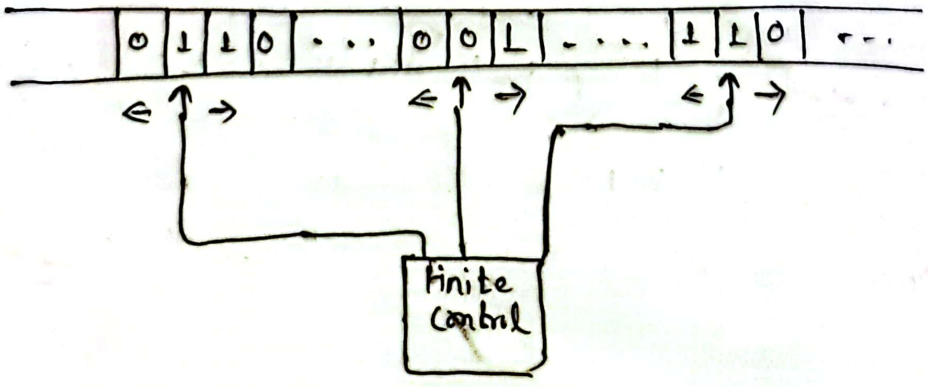
A multi-head Turing Machine can be defined as a Turing machine with a single tape and a single finite state control but with multiple independent R/W heads.

Thus the transition function of multihead Turing machine can be defined as

$$\delta : Q \times \Gamma_1 \rightarrow Q \times \Gamma_T \times \{L, R\}^n$$

where, $\Gamma_T = \Gamma \times \Gamma \times \Gamma$ and n is the number of R/W heads

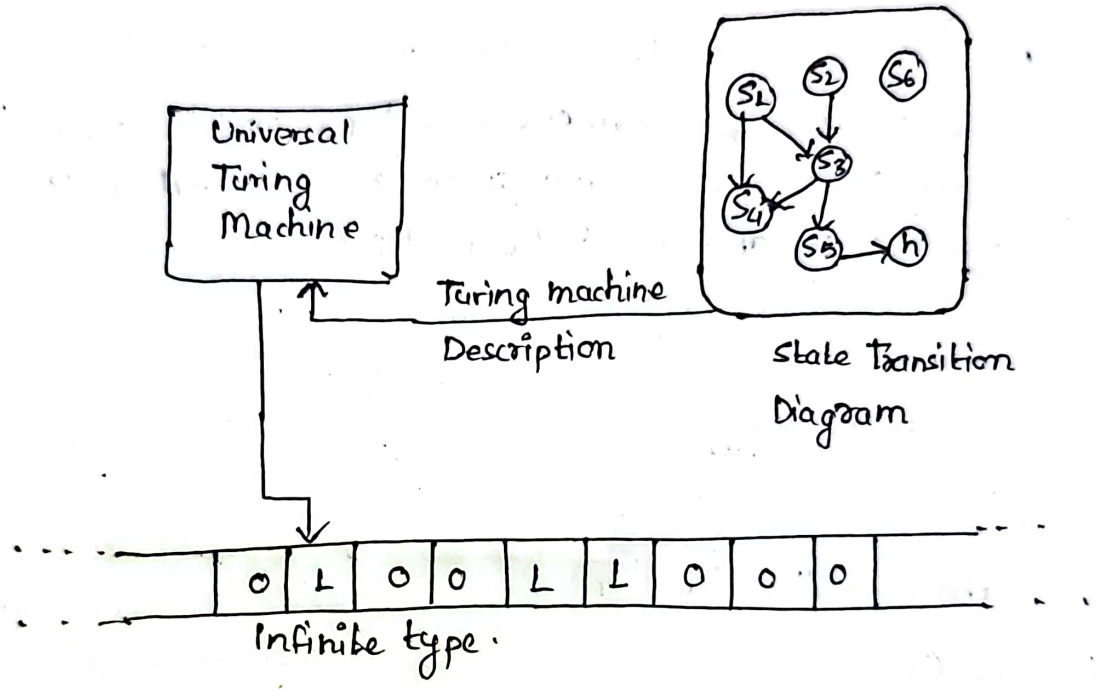
A three head Turing machine i.e a multi head Turing machine with 3 head is shown in figure.



* Universal Turing Machine :-

⇒ A general purpose Turing machine is said to be a universal Turing machine, if it is powerful enough to simulate the behaviour of any digital computer. A universal machine can accept two inputs :

1. The input data
2. An algorithm for computation.



⇒ A universal turing machine is an idealized computing device consisting of a read/write head and a infinite tape which goes through the head.

⇒ The tape is divided into squares, each square bearing a single symbol '0' or '1' for example.

⇒ This tape is the machine's general purpose storage medium: the machine is set in motion with its input inscribed on the tape, output is written onto the tape by the head and the tape serves as a short-term working memory for the results of intermediate step of the computation.

⇒ The program governing the particular computation that the machine is to perform is also stored on the tape.

* Working of Universal Turing Machine:-

1. Scan the cell on the state area of the tape and read the symbol that turing machine reads and the initial state of Turing machine.
2. Move the tape to the program, which contains the finite automation table. Find the row which is headed by the symbol scanned in the previous state.
3. Find the required column.
4. Move the tape to the appropriate square on the data area of the tape, replace the symbol, move the tape in the given direction read the next symbol, reach the state area and replace the state by the current state and repeat from step 1.

* Recursive function theory:-

Initial functions

There are three initial functions:

① Zero function (z)

It is defined as $z(x) = 0$.

eg. $z(5) = 0$, $z(70) = 0$

② Successor function(s)

It is defined as $s(x) = x+1$

eg. $s(6) = 6+1 = 7$

$s(70) = 70+1 = 71$

③ Projection functions(P)

It is defined as $P_i^n(x_1, x_2, x_3 \ldots x_n) = x_i$

eg. $P_1^4(3, 7, 8, 9) = 3$

$P_2^3(7, 9, 12) = 9$

* Composition of function :-

A composite function is defined when one function is substituted into another function.

eg. let $f(x) = 3x+2$

$g(x) = x+5$

then, $f(g(x)) = f(x+5)$

$= 3(x+5)+2$

$= 3x+17$

* Recursive function

→ A function that represents itself again and again is called Recursive function.

* Primitive Recursive function

↳ A function is called primitive recursive function if it can be obtained from initial functions through finite number of composition and recursive steps.

* Partial Recursive function

↳ A function is called partial recursive function, if it is defined for some of its arguments.

eg. substraction of two positive numbers m and n

$$f(m,n) = m-n \quad m \geq n$$

$$m < n$$

Chomsky Hierarchy :-

The chomsky Hierarchy as originally defined by Noam chomsky, comprises four types of languages and their associated grammars and machines.

Language	Grammar	Machine	Example
Regular Language	Regular Grammar	Finite Automata	a^*
Context-Free Language	Context free Grammar	Push down Automata	$a^n b^n$
Context Sensitive Language	Context Sensitive Grammar	Linearly bounded Automata	$a^n b^n c^n$
Recursively Enumerable Language	Unrestricted Grammar	Turing Machine	Any Computable function

*Unrestricted Grammars :-

⇒ An unrestricted grammar (also called phrase-structure grammar) is a 4-tuple $G = (V, T, P, S)$ where V and T are disjoint set of variables and terminals respectively. $S \in V$ is a start symbol and P is the set of productions of the form $\alpha \to \beta$ where $\alpha, \beta \in (V \cup T)^*$ and α contains at least one variable

The productions are of the form

$$\alpha A \beta \to \alpha \gamma \beta \text{ or } \alpha A \beta \to \gamma$$

⇒ An unrestricted grammar are used to describe the language that are not context-free. In fact the language described by unrestricted grammar can be accepted by a TM. Hence they are recursively enumerable.

An unrestricted grammar generating $\{a^n b^n c^n \mid n \geq 1\}$

$S \rightarrow FS_L$

$S_L \rightarrow ABCS_L$

$S \rightarrow ABC$

$BA \rightarrow AB$

$CA \rightarrow AC$

$CB \rightarrow BC$

$FA \rightarrow a$

$aA \rightarrow aa$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

The string aabbcc can be derived as,

$S \Rightarrow FS_L$

$\Rightarrow FABCS_L$

$\Rightarrow FABCABC$

$\Rightarrow FABACBC$

$\Rightarrow FAABCBC$

$\Rightarrow FAABBCC$

$\Rightarrow aABBCC$

$\Rightarrow aaBBCC$

$\Rightarrow aabBCC$

$\Rightarrow aabbCC$

$\Rightarrow aabbcc$

Chapter - 5

Undecidability

Chapter Outline

1. The church - Turing thesis Machine
2. Halting problem, Universal Turing Machine
3. Undecidable problems about Turing Machine, grammars
4. Properties of Recursive, Recursively enumerable languages.

1. Church - Turing Thesis :-

⇒ In 1930, Alan Turing proposed a thesis

"If there is a procedure, algorithm for solving a problem that can be implemented on Turing Machine.

⇒ Many mathematicians were working independently on same problem (on which turing worked).

Alanzo church - lambda calculus

Noam chomsky - Unrestricted Grammar

Emil post - production system

⇒ All of above formalism were proved equivalent to one another. This led to

@ Turing thesis (weak form) :-

A Turing Machine can compute anything that can be computed by digital computer.

(b) Turing Thesis (strong form) :-

A Turing machine can compute anything that can be computed.

2. Halting Problem, Universal Turing Machine:-

* Halting Problem:-

⇒ As we know, the output of TM can be

(i) Halt. The machine starting at this configuration will halt after a finite number of states.

(ii) No Halt. The machine starting at this configuration never reaches a halt state, no matter how long it runs.

Now, the question arises based on these two observations: Given any functional matrix, input data tape and initial configuration then, is it possible to determine whether the process ever halt? This is called halting problem.

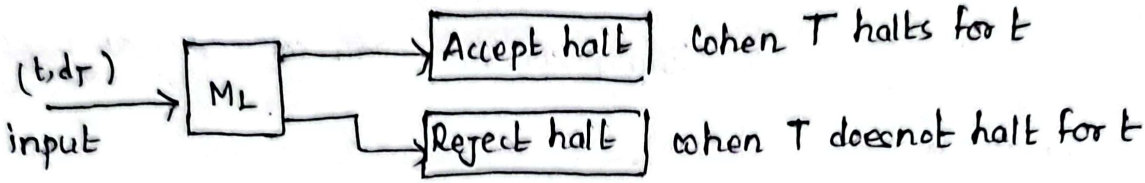
* Halting Problem is Unsolvable ??

Proof:

Let there exists a TM, M_L which decides whether or not any computation by a TM T will ever halt when a description d_T of T and tape t of T is given [That means the input to machine M_L will be (machine, tape) pair].

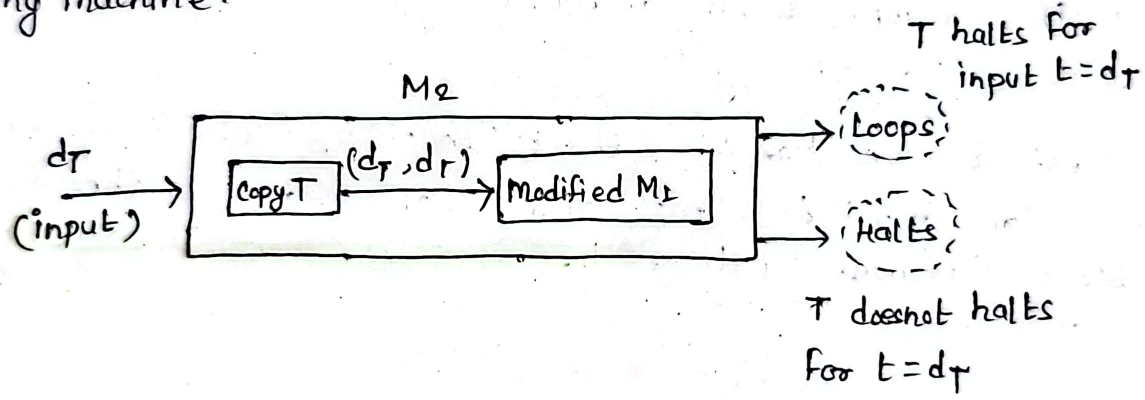
Then for every input (t, d_t) to M_L if T halt for input t, M_L also halts which is called accept halt.

Similarly, if T doesnot halt for input t then the M_L will halt which is called reject halt. This is shown by given figure.

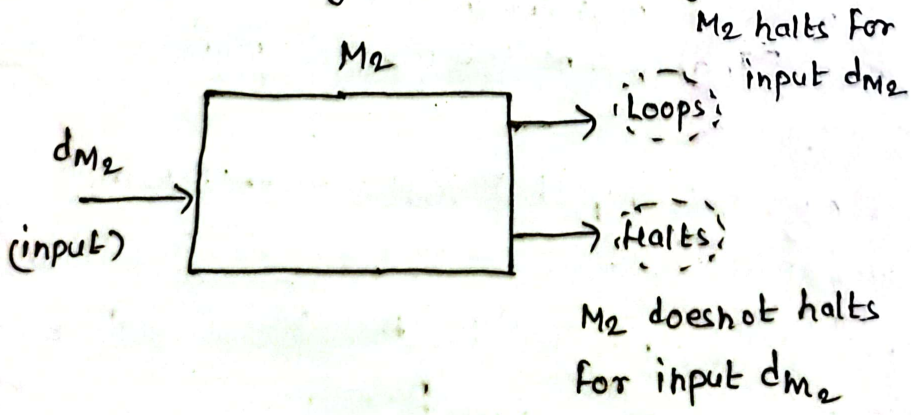


⇒ Now we will consider another Turing machine M_2 which takes an input d_T. If first copies d_T and duplicates d_T on it's tape and then this duplicated tape information is given as input to machine M_1.

⇒ But machine M_1 is a modified machine with the modification that whenever M_1 is supposed to reach an accept halt, M_2 loops forever. Hence behavior of M_2 is as given. It loops if T halts for input $t = d_T$ and halts if T doesnot halt for $T = d_T$. The T is any arbitary turing machine.



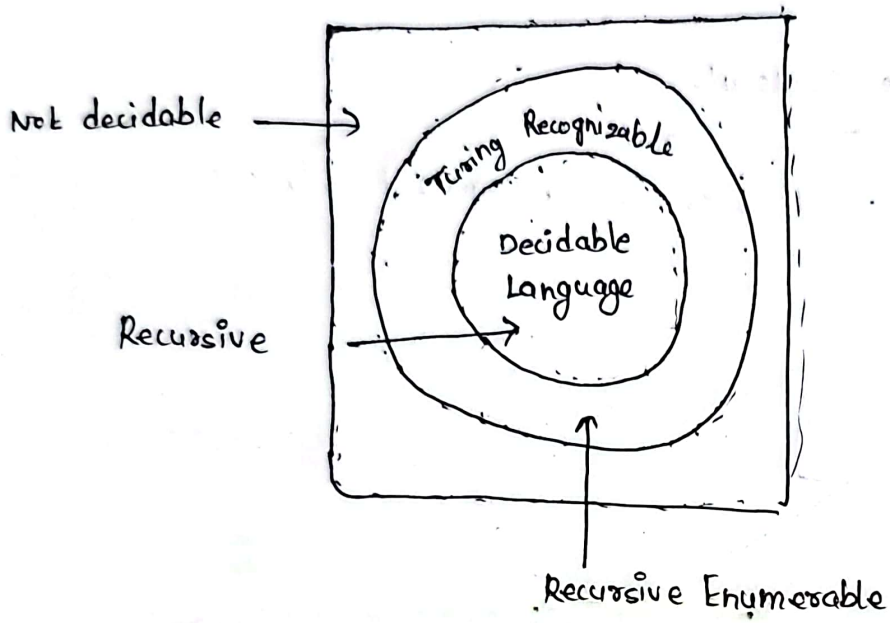
⇒ As M_2 itself is one Turing machine we will take $M_2 = T$. That means we will replace T by M_2 from above given machine.



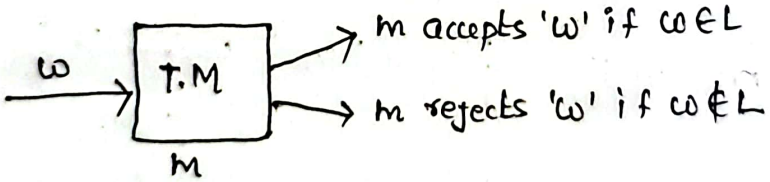
⇒ Thus, machine M_2 halts for input d_{M2} if M_2 doesnot halt for input d_{M2}. This is a contradiction. That means a machine M_1 which can tell whether any other TM will halt on particular input doesnot exist. Hence, halting problem is unsolvable.

#Undecidable Problem about Turing Machine

* Decidable Language / Recursive Language:-

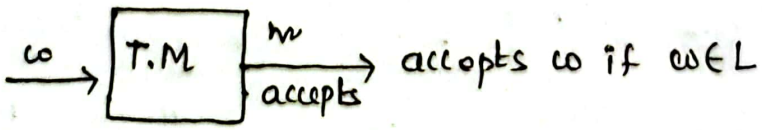


⇒ A Language 'L' is decidable if some Turing machine decides it



* Recognizable:-

A Language is recognizable if some turing machine recognizes it.



if $w \notin L$ → reject
or doesnot terminate loop.

⇒ Decidable language is also known as decidable language or Computable language or solvable language.

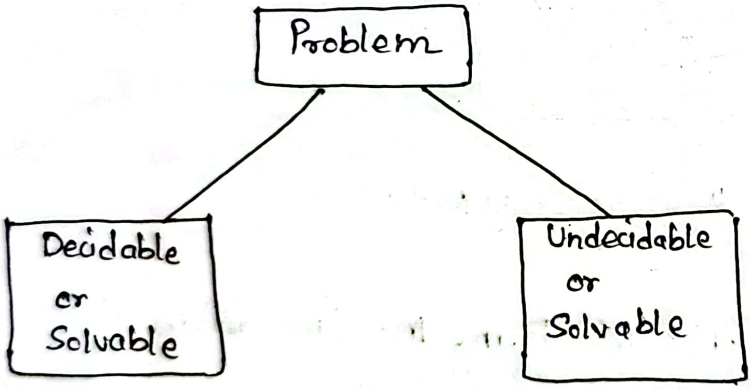
⇒ On given input ω T.M always halt → accept $\omega \in L$
→ reject $\omega \notin L$

Example: Regular language and Context free language.

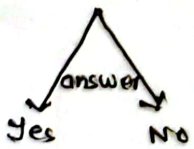
⇒ Recursive Enumerable language also known as Turing Recognizable or Partially decidable or semi-decidable.

Decidable and Undecidable Problem:-

⇒ Decidable and Undecidable problem is very analogous to solvable and unsolvable Problems.



⇒ Solution is definite in this case.

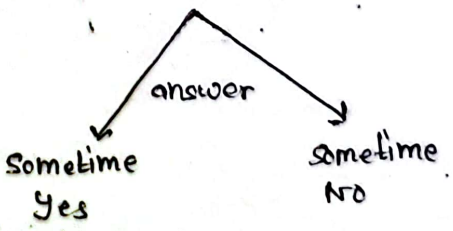


⇒ Examples of decidable problems are:

① Does the sun rises in the east?

② Does the earth moves around sun.

⇒ Solution is undefinite in this case



⇒ Example of Undecidable problems

① Will be raing day tomorrow?

② One day I will be software engineer?

Examples of Decidable problem

① Does the finite automaton accept regular language?

② L_1 and L_2 are two regular language are closed under following
- (A) Union
- (B) Concatenation
- (C) Kleene star

③ If $L_1 \cup L_2$ are two CFL then $L_1 \cup L_2$ is CFL?

Example of Undecidable problem

① For given context free grammar G_1, is $L(G_1)$ is ambiguous

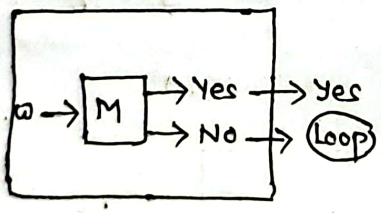
② For two given CFL, L_1 and L_2 whether $L_1 \cap L_2$ is CFL or not?

③ A computational problems is said to be unsolvable if no algorithm solve it

Properties of Recursive, Recursively Enumerable Language:-

1. Every recursive language (Turing decidable language) is recursively enumerable (Turing acceptable).

Proof:



Let M be a Turing Machine that decides L. let L' be another Turing Machine which simulates M. M' would accept strings for which M halts on yes and go to infinite loop if M halts and write 'No' on the tape. Hence proof.

2. If L is recursive language the L' is also a recursive Language.

Proof :-

let there will be some L that can be accepted by Turing machine M. Hence we can denote language L by L(M). On acceptance of L(M) the machine M always halts.

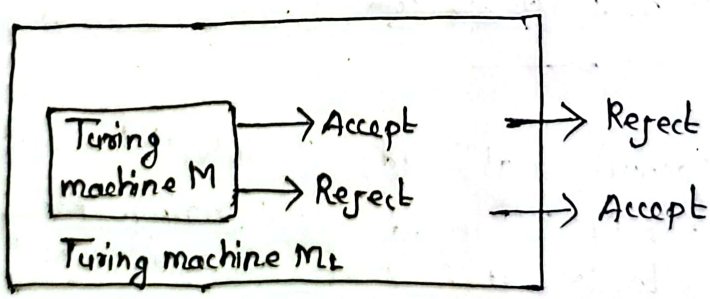
Now we construct a Turing machine M_L such that $L' = M_L$

M_L is obtained by modifying M as follows.

(i) The accepting states of M are made non-accepting states of M_L. That means we have created a states such that M_L will halts without accepting.

(ii) let M_L have a new state q_f. After reaching q_f, M_L doesnot move in further transitions.

(iii) If q is a non-accepting state of M and $\delta(q, x)$ is not defined add a transition from q to q_f for M_L. That is, in machine M for each of the transition with combination of non-accepting state and input tape symbol, make the same transition having the combination of accepting state and input tape symbol for machine M_L.



As M halts, M_L also halts (if M reaches an accepting state on w, then M_L doesnot accept w and halts and conversely.).

3. The union and intersection of two recursive languages are recursive.

Proof

Let L_1 and L_2 be two recursive languages and M_1 and M_2 be the corresponding TMs that halt.

We design a TM M as a two-tape TM as follows:

(i) w is an input string to M.

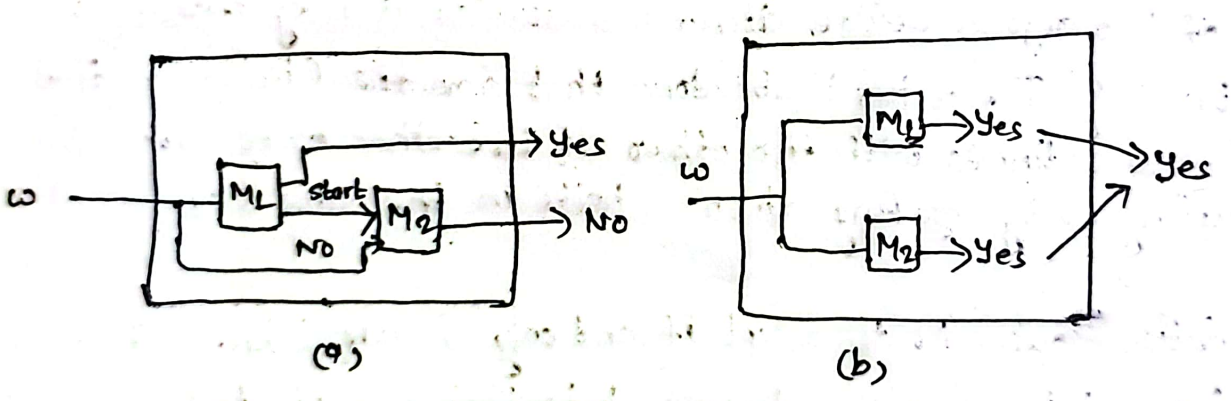
(ii) M copies w to its second tape.

(iii) M simulates M_1 on the first tape. If w is accepted by M_1 then M accepts w.

(iv) M simulates M_2 on the second tape. If w is accepted by M_2 then M accepts w.

M always halts for any input w.

Thus, $L_1 \cup L_2 = T(M)$ and hence $L_1 \cup L_2$ is recursive. Proof $L_1 \cap L_2$ is similar except that M accepts iff both M_1 and M_2 accepts.



4. If L_1 and L_2 are recursively enumerable languages over Σ, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are also recursively enumerable.

Proof:- Suppose $M_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, q_1, B_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \Gamma_2, \delta_2, q_2, B_2, F_2)$ are TMs accepting L_1 and L_2 respectively. We now construct $L_1 \cup L_2$ and $L_1 \cap L_2$. In both cases, it is useful to use a two-tape machine.

The two tape machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ begins by placing a copy of the input string, which is already on tape 1, onto tape 2. It inserts the marker # at the beginning of both tapes in order to detect a crash resulting from T_1 or T_2 trying to move its tape head off the tape. From this point on, the simultaneous of T_1 on tape 1 and T_2 on tape 2 is accomplished by allowing every possible move

$$\delta(p_1, p_2), (a_1, a_2) = [(q_1, q_2), (b_1, b_2), (D_1, D_2)]$$

where for both values of i,

$$\delta_i(p_i, a_i) = (q_i, b_i, D_i)$$

The possible outcomes of the simulation are:

1. Neither T_1 nor T_2 ever stops, in which case T never stops.
2. At least one of the two accepts, in which case T accepts.
3. One of two rejects before either has accepted. If they both rejects simultaneously, T rejects; T abandons that simulation (i.e ignores that tape) and continues with the other. If the other halts, either by accepting or by rejecting, then T halts in the same way.

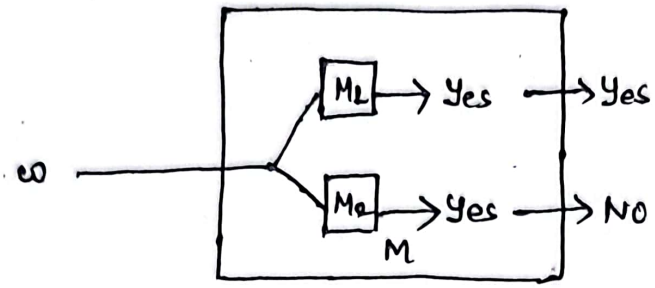
The construction causes M to accept if and only if atleast one of two machines M_1 and M_2 accepts and we conclude that T accepts the language $L_1 \cup L_2$.

We can prove for $L_1 \cap L_2$ in the same way, except that this time T can reject if either T_1 or T_2 rejects and it can accept only when T_1 and T_2 both have accepted.

5. If both a language L and it's complement L' are recursively enumerable then L is recursive.

Proof:

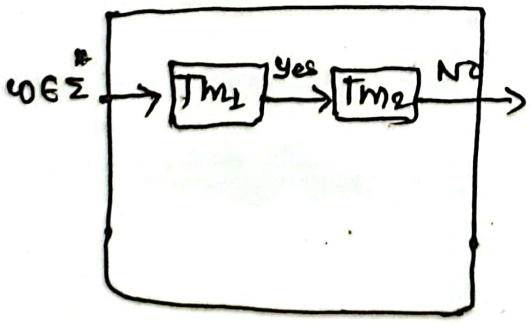
Let M_1 and M_2 accepts L and L'. Now construct M to simulate M_1 and M_2 as shown in figure.



M accepts w if M_1 accepts w and rejects w if M_2 accepts w. Since w is in either L or L', we see that exactly one of M_1 or M_2 will accept. Thus M will always say either "yes" or "no" but will never say both. Note that there is no apriori limit on how long it may take before M_1 or M_2 accepts, but it is certain that one or other will do. Since M is an algorithm that accept L, it follows that L is recursive.

6. If L is a recursive language than Σ^*-L is recursive.

Proof:- The required Turing machine Tm-complement can be represented by following diagram.



The machine Tm-complement functions as follows: when a string $W \in \Sigma^*$ is given an input to Tm-complement, its control passes the string to Tm_1 as input to Tm_2. As Tm_1 decides the language L, therefore

for $\omega \in L$ after a finite number of moves, Tm_1 output yes which is given as input to Tm_2, which in turn returns No.

Similarly, for $\omega \notin L$, Tm_2 returns "yes". Hence, there exist a turing machine Tm-complement for $\Sigma^* - L$. So, it is turing decidable that is recursive.

Chapter - 6

Computational Complexity

Time complexity	Space Complexity
1. Time complexity means how long computation takes to execute.	1. Space complexity means how much storage is required for computation.
2. In T.M, this could be measured as the number of moves which are required to perform computation.	2. In T.M, number of tape squares(cell) used.
3. In computer, number of machines cycles in the RAM is known as Time complexity.	3. In computer, number of bytes used in the RAM to store data or functional known as space complexity.

* Tractability and Intractability :-

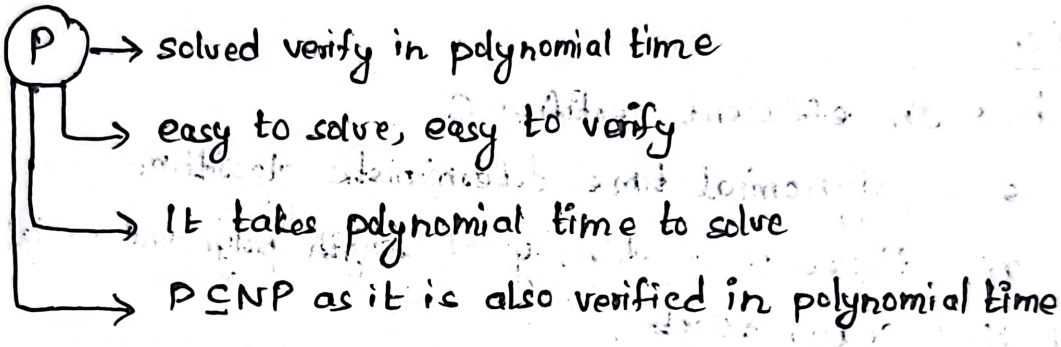
Tractable Problem	Intractable Problem
⇒ A problem is tractable if and only if there is an efficient algorithm that solves it in polynomial time.	⇒ A problem is called Intractable if there is no efficient algorithm to solve it.
{i.e n^k where k is constant	or
i.e $O(n), O(n^2) \ldots O(n \cdot \log n)$ }	A problem that cannot be solved by polynomial time algorithm.
⇒ Solve in Polynomial time.	{i.e $2^n \ldots n!$}
⇒ Sum of n integers	⇒ Save in exponential time.
$S = 1+2+\ldots+n$	⇒ Tower of hanoi $= 2^n - 1 \rightarrow O(2^n)$
$O(n)$	

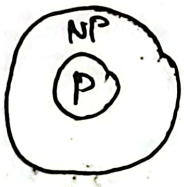
* Class P Problem :-

→ P is a set of problem that can be solved in polynomial time using deterministic algorithm.

→ If a problem can be solved in polynomial time on a regular computer, that problem is known as class P problem.

→ As P class problem verified in time also, hence it is also a subset of NP class problem.





→ In complexity theorem P is also known as PTime or DTime is one of the most fundamental classes.

→ It contains all the decision problems that are solved by a deterministic turing machine using polynomial amount of computation time. i.e for a problem with size N, there must some way to solve the problem in f(n) steps.

Defination :-

The Language L is said to be class P, if there exist a Polynomial bounded Turing Machine (Deterministic) such that TM is of time complexity P(n) for some polynomial P and TM accepts L. This language is also called polynomial decidable.

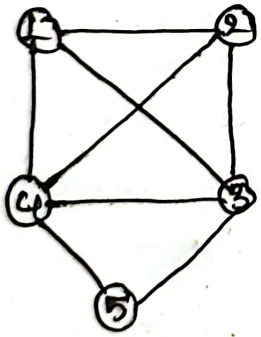
Examples:

- Eulerian and Hamilton graph problem
- Optimization problem
- Integer partition problem

Explanation: Hamilton Cycle

⇒ Hamilton cycle is a path in a given graph that visit each vertex exactly once and there is a edge/path between first and last vertex.

⇒ There may be any number of Hamiltonian cycle exist in a graph



1-2-3-5-4-1
1-3-5-4-2-1

⇒ Here we have to determine whether a given graph contains hamiltonian cycle or not. If it contains then print the path.

⇒ The cyclic condition ensures that the circuit is closed and the requirement that all the nodes are included (with no repeats) ensures that circuit doesnot cross over itself and passes through every node.

* Class NP Problems:-

→ NP is a set of problems that can be solved in polynomial time using non-deterministic algorithms.

→ NP problems are problems which can be solved in polynomial time on non-deterministic turing machine.

→ It can be verified in polynomial time.

→ NP problems can be solved by a polynomial time using "non-deterministic algorithm" a magical algorithm that always makes a right guess among the given set of choices.

→ Example:

Graph Coloring:-

Graph coloring in such a way that each adjacent nodes has different color.

→ This NP class problem is solved in exponential time, but solution verification is easy because once a solution is provided it is much easier to check for it's correctness.

→ Defination:- A language L is in class NP if there is polynomially bounded non-deterministic turing machines such that TM is of time complexity P(n) for some polynomial P and TM accept L.

→ Example Explanation: Travelling Salesman Problem: NP complete example

⇒ Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that a salesman has to visit every city at exactly once and return to the starting point.

For a given graph in which nodes (cities) are connected by the directed edges (routes) where the weight of an edge is the distance between two cities.

⇒ Assume a salesman has travels to n cities.

⇒ He starts from particular city, visiting each city once and then returns to his starting point.

⇒ The objectives is to select the sequence in which the cities are visited (with shortest possible route) in such a way that his total travelling cost is minimized.

⇒ The only know solution that grows exponentially with the problem size. (i.e number of cities).

⇒ There is an example of NP complete problem for which no known efficient (i.e polynomial time) algorithm exists

* Polynomial Time Reducibility:-

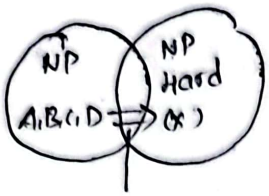
Let E and D be two decision problems. we say that D is polynomial time reducible to E if there exists an algorithm A such that

- A takes intances of D as inputs and always outputs the correct answer "Yes" or "No" for each instances of D.
- A uses a subroutine a hypothetical algorithm B for solving E.
- There exists a polynomial p such that for every instance of D of size n the algorithm A terminate in at most p(n) steps if each cell call subroutine B is counted as only m steps. where m is the size of the actual input of B.

Np Hard and NP Complete:-

* NP-Hard :-

⇒ A problem is NP-Hard if every problem in NP can be polynomial reduced to it.



Reducible
Polynomial

X is NP Hard

⇒ A problem is NP-hard of an algorithm for solving it can be translated into one for solving any other NP problem. NP hard therefore means "at least as hard as any NP-problem.

⇒ Although it might in fact be harder. The NP-hard class is the superset containing NP complete class this is potentially harder to solve than NP complete problem.

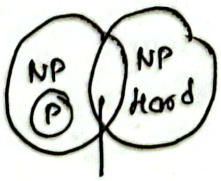
* NP-Complete :-

⇒ A decision Yes/No problem is NP-complete if:

(a) L is in NP

(b) L is in NP-hard

(Every problem in NP is reduced to L in polynomial time)



NP Complete

⇒ The most important property of NP complete problem is polynomial time reducibility. Any NP complete polynomial can be transformed into any other NP complete problem in polynomial time.

⇒ Thus if it could be proved that, any NP complete problem is formally intraceable or such algorithm exist.

⇒ Once we have some NP-complete problem we can prove a new problem to be NP-complete by reducing some known NP-complete problem to it by using polynomial time reduction.

⇒ Examples: Hamilton Cycle problem
Traveling Salesman problem