Unit-VI

Computability and Complexity Theory

Class-TE Comp

- **Decidable Problems:**
- A problem is said to be Decidable if we can always construct a corresponding algorithm that can answer the problem correctly.
- We can possibly understand Decidable problems by considering a simple example.
- Suppose we are asked to compute all the prime numbers in the range of 1000 to 2000.
- To find the solution of this problem, we can easily devise an algorithm that can enumerate all the prime numbers in this range.
- Decidability in terms of a Turing machine, a problem is said to be a Decidable problem if there exists a corresponding Turing machine which halts on every input with an answer- yes or no.

- **Undecidable Problems:**
- The problems for which we can’t construct an algorithm that can answer the problem correctly in finite time are termed as Undecidable Problems.
- These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.
- popular Undecidable Problem which states that no three positive integers a, b and c for any n>2 can ever satisfy the equation: a^n + b^n = c^n.
- If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable values of n, a, b and c.

- ## Undecidable Problems:

➢ Whether a CFG generates all the strings or not?

- As a CFG generates infinite strings, we can’t ever reach up to the last string and hence it is Undecidable.

➢ Whether two CFG L and M equal?

- Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.

➢ Ambiguity of CFG?

- There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.

Theorem 1 : Prove that following decision problems are recursive
(i) Two DFA's are equivalent or Not (ii) NFA Accepts a word or not

SPPU : May-15 End Sem, Marks 10, May-16, End Sem, Marks 5

Proof : (i)

- To prove that the given decision problem is recursive. We require a TM T(M) which simulates the DFA.
- Let, M_1 and M_2 are two DFAs. Consider the Turing machines $T(M_1)$ and $T(M_2)$ Obtain set of strings accepted by $T(M_1)$ and rejected by $T(M_2)$. That is

$$S_1 = T(M_1) \cap \overline{T(M_2)}.$$

- Similarly obtain the set of strings which $T(M_2)$ accepts and $T(M_1)$ rejects. That is

$$S_2 = T(M_2) \cap \overline{T(M_1)}.$$

- If these two sets are empty then it shows that $L(M_1)$ and $L(M_2)$ are equivalent and can be simulated by TM for the set of languages as.

$$L(M) = L(M_1) \cap \overline{L(M_2)} \cup \overline{L(M_1)} \cap L(M_2). \text{ The } L(M) = \phi.$$

- This shows that two DFA's are equivalent or not is a recursive problem.

(ii) Let, M be the NFA and input word w which decides if M accepts w. Convert this NFA to DFA. Then run the algorithm for DFA on Turing Machine T(M). The TM T(M) accepts w if and only if M reaches a final state on w. This shows that NFA accepts a word or not is recursive.

Class-TE Comp

Theorem 6 : Prove that i) AREX = $\{<R,W> | R$ is a regular expression that generates string w} is a decidable language.
ii) ECFG = $\{<G> | G$ is a CFG and $L(G) = \phi\}$ is a decidable language.

SPPU : May-19, End Sem, Marks 8

Proof :

i)

- Let M be the DFA and input word w ∈ R which is accepted by M.
- Simulate an algorithm for DFA to run on Turing Machine TM
- The TM accepts w if and only if M reaches a final state for w otherwise rejects.
- This shows that the regular expression generates string that denotes decidable language.

ii)

- We construct TM M to decide a problem.
- Scan input string <G>, determining whether the input constitutes a valid CFG. If not, then reject.
- Mark every terminal in every rule.
- For each rule $A \rightarrow w$, if every symbol of w is marked, mark every occurence of A in rules.
- Continue until no new variables are marked.
- If S (Start symbol) is not marked then accept <G> otherwise reject.
- This shows that if G is a CFG then then $L(G) = \phi$ is decidable language.

Class-TE Comp

(vii) Prove that A_{TM} = {m, w > | M is a TM and accepts w } is undecidable.

1. Assume A_{TM} is decidable → there's a decider H, L(H) = ATM

2. H on = Accept if M accepts w

   Reject if M rejects w (halts in qREJ or loops on w)

3. Construct new TM D: On input :

   Simulate H on <M,<M>> (here, w =<M>)

   If H accepts, then Reject input <M>

   If H rejects, then Accept input <M>

4. What happens when D gets <D>as input ?

   D rejects <D> if H accepts <D,<D>> if D accepts<D>

   D accepts <D> if H rejects <D,<D>> if D rejects <D>

   Either way: Contradiction! D cannot exist → H cannot exist

   Therefore, A_{TM} is not a decidable language.

Theorem 4 : Prove that CFG G generates the string w or not.

SPPU : May-15, End Sem, Marks 5, May-16, End Sem, Marks 5

Proof :

Convert CFG G to G″ in chomsky normal form. Now the string $w \in L(G'')$ iff w can be derived in $2|w|-1$ steps where none of the intermediate string is of length more than $|w|$. If these derived step will derive w, the T(M) accepting L(G″) will accept otherwise rejects. This shows CFG G generates string w or not is recursive.

Class-TE Comp

The Church-Turing thesis:

- The Church-Turing thesis says that every solvable decision problem can be transformed into an equivalent Turing machine problem.
- It can be explained in two ways, as given below −
  - ✓The Church-Turing thesis for decision problems.
  - ✓The extended Church-Turing thesis for decision problems.
- Let us understand these two ways.

✓**The Church-Turing thesis for decision problems:**

There is some effective procedure to solve any decision problem if and only if there is a Turing machine which halts for all input strings and solves the problem.

✓**The extended Church-Turing thesis for decision problems:**

A decision problem Q is said to be partially solvable if and only if there is a Turing machine which accepts precisely the elements of Q whose answer is yes.

The Church-Turing thesis:

Proof

- A proof by the Church-Turing thesis is a shortcut often taken in establishing the existence of a decision algorithm.
- For any decision problem, rather than constructing a Turing machine solution, let us describe an effective procedure which solves the problem.
- The Church-Turing thesis explains that a decision problem Q has a solution if and only if there is a Turing machine that determines the answer for every $q \in Q$. If no such Turing machine exists, the problem is said to be undecidable.

- **Growth Rates:** Algorithms analysis is all about understanding growth rates. That is as the amount of data gets bigger, how much more resource will my algorithm require? Typically, we describe the resource growth rate of a piece of code in terms of a function.

description	order of growth	example	framework
constant	1	`count++;`	*statement (increment an integer)*
logarithmic	$\log n$	`for (int i = n; i > 0; i /= 2)` `count++;`	*divide in half (bits in binary representation)*

linear	n	`for (int i = 0; i < n; i++)`
		`if (a[i] == 0)`
		`count++;`
			single loop (check each element)

linearithmic	$n \log n$	[*see mergesort* (PROGRAM 4.2.6)]	*divide-and-conquer (mergesort)*

quadratic	n^2	`for (int i = 0; i < n; i++)`
		`for (int j = i+1; j < n; j++)`
		`if (a[i] + a[j] == 0)`
		`count++;`
			double nested loop (check all pairs)

cubic	n^3	`for (int i = 0; i < n; i++)`
		`for (int j = i+1; j < n; j++)`
		`for (int k = j+1; k < n; k++)`
		`if (a[i] + a[j] + a[k] == 0)`
		`count++;`
			triple nested loop (check all triples)

exponential	2^n	[*see Gray code* (PROGRAM 2.3.3)]	*exhaustive search (check all subsets)*

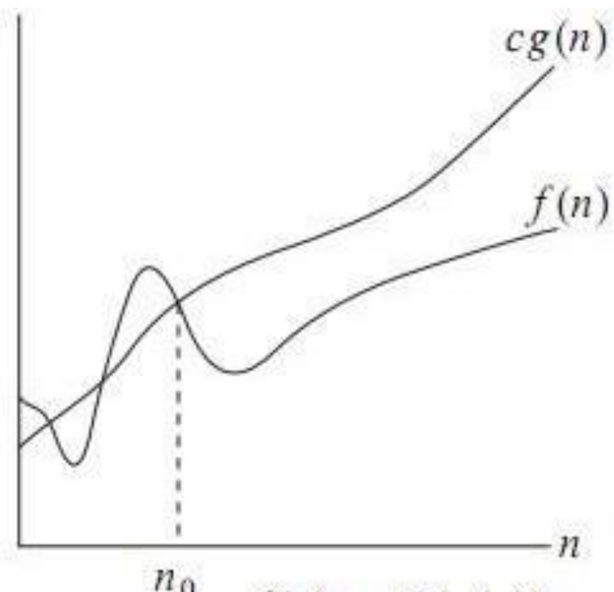
Asymptotic notation

- Big Theta (T)
- Big Oh(O)
- Big Omega ()

Class-TE Comp

Big Oh (*O*):-

- f(n)= O(g(n)) *if* there exist positive constants c and n0 such that f(n) ≤ cg(n) for all n ≥ n0
- O-notation to give an upper bound on a function.
- For example, consider the case of Insertion Sort.
- g(n) is upper bound of the f(n) if there is exists some positive constants c and n0.It is denoted as f(n)=O(g(n)).



$f(n) = O(g(n))$

Class-TE Comp

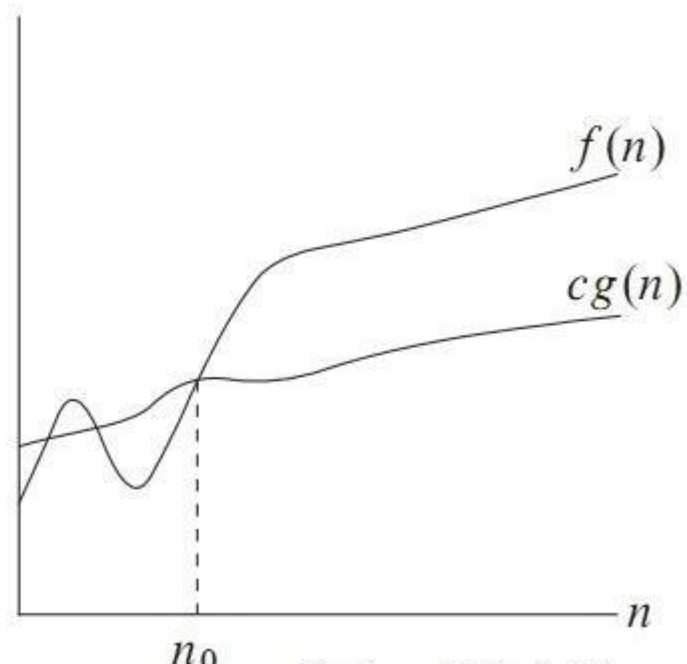
- **Omega Notation:-**

  Big oh provides an asymptotic upper bound on a function.

  Omega provides an asymptotic lower bound on a function.

  Running time of the algorithm cannot be less than asymptotic lower bound for sequence of the data.

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \}.$$

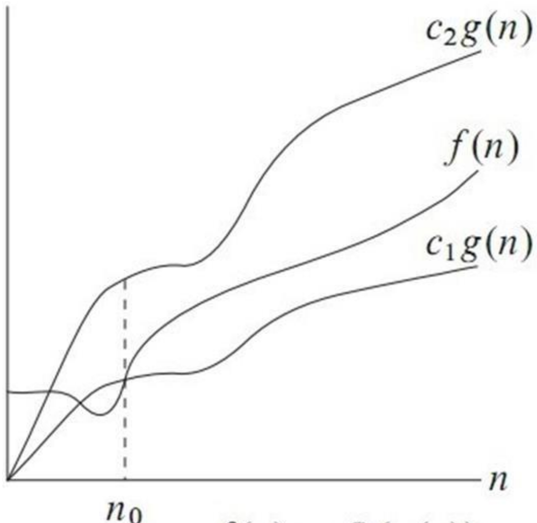


$f(n) = \Omega(g(n))$

Theta Notation:-

- Theta notation is used when function f can be bounded both from above and below by the same function g.
- Running time of the algorithm cannot be less than or greater than its asymptotic tight bound for random sequence of the data.

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \}.$$[1]



$f(n) = \Theta(g(n))$

Class-TE Comp

DETERMINISTIC ALGORITHM & NON-DETERMINISTIC ALGORITHM

DETERMINISTIC ALGORITHM

- In Deterministic algorithm For a particular input the computer will give same output every time.
- Examples are finding odd or even,sorting,finding max etc.
- Most of the algorithm are deterministic in nature.
- It is solve in polynomial time.

- **NON-DETERMINISTIC ALGORITHM:-**
- In non deterministic algorithm for a same input the computer will give different output on different execution.
- This algorithm operates in two phases Guessing and Verification.
- Randomly picking some elements from the list and check if it is maximum is non-deterministic.
- It is not solve in polynomial time.
- To specify such non deterministic algorithm there are 3 function used in algorithm.

The following Function

1) **Choice()-** Select one of the element of set 's'.
2) **Failure()-** Check for unsuccessful completion.
3) **Success()-** check for successful completion.

Ex) Write algorithm for searching for an element 'x' from the given set of element n.

Solution:-

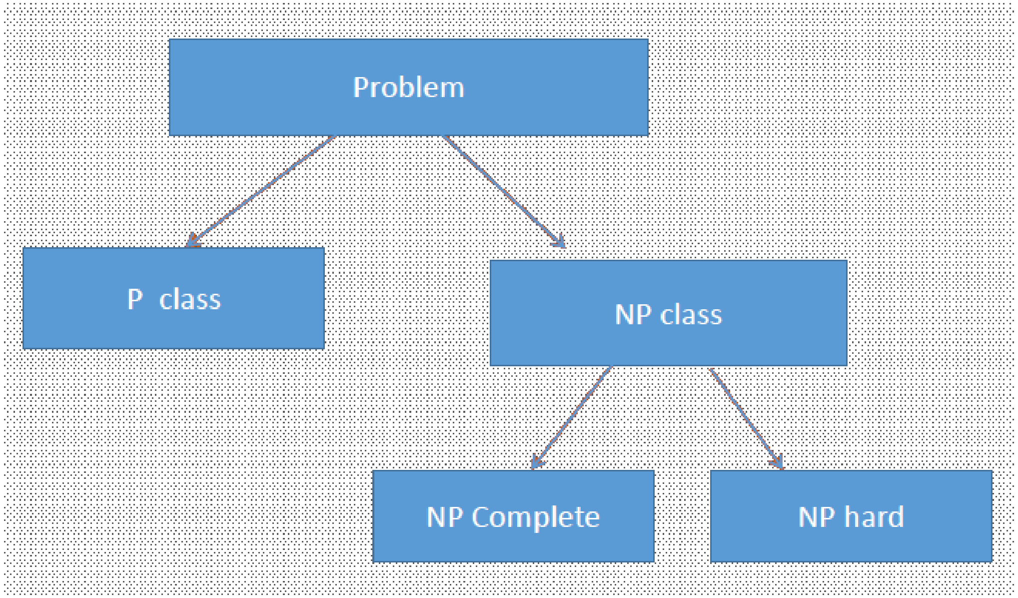
```
// A is an array of size n
1) Mid = choice(1,n);--------------------Guessing Stage
2) If A[mid]==x then
     {
        write(mid);----------------------------Verification Stage
        success();
      }
Else
{
Write(0);
Failure();
}
```

POLYNOMIAL AND NON-POLYNOMIAL PROBLEMS

The computing time or any algorithm is divided into two groups

1) ***POLYNOMIAL Problem:-***It is Problem whose solution times are bounded.

2) ***Non POLYNOMIAL Problem:-***It is Problem whose solution not times are bounded. That is output is not predictable.



P-Class Problem:-

- The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time O(nk) in worst-case, where k is constant.
- These problems are called tractable, while others are called intractable or super polynomial.
- The advantages in considering the class of polynomial-time algorithms is that all reasonable deterministic single processor model of computation can be simulated on each other with at most a polynomial.

NP-Class Problem:-

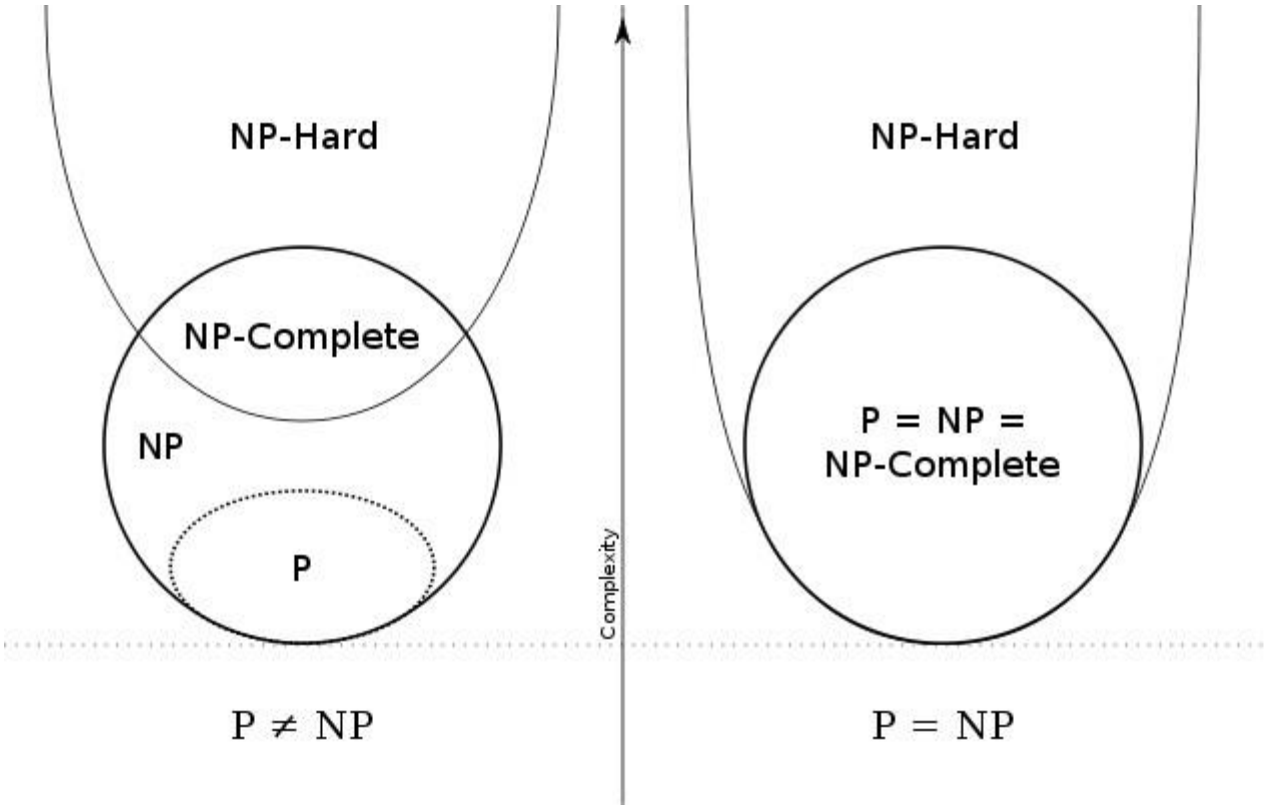
- The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information.
- Every problem in this class can be solved in exponential time using exhaustive search.

DIFFERENCE BETWEEN P PROBLEMS AND NP PROBLEMS

P PROBLEMS	NP PROBLEMS
P problems are set of problems which can be solved in polynomial time by deterministic algorithms.	NP problems are the problems which can be solved in non-deterministic polynomial time.
The problem belongs to class P if it's easy to find a solution for the problem.	The problem belongs to NP, if it's easy to check a solution that may have been very tedious to find.
P Problems can be solved and verified in polynomial time.	Solution to NP problems cannot be obtained in polynomial time, but if the solution is given, it can be verified in polynomial time.
P problems are subset of NP problems.	NP problems are superset of P problems.
It is not known whether P=NP. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P=NP.	If P≠NP, there are problems in NP that are neither in P nor in NP-Complete.
All P problems are deterministic in nature.	All the NP problems are non-deterministic in nature.
Selection sort, linear search	TSP, Knapsack problem.

Class-TE Comp

Relation between P,NP,NP hard and NP Complete



Class-TE Comp

DIFFERENCE BETWEEN NP HARD AND NP COMPLETE PROBLEM

BASIS OF COMPARISON	NP HARD PROBLEM	NP COMPLETE PROBLEM
Description	NP-Hard problems (say X) can be solved if and only if there is a NP-Complete problem (say Y) can be reducible into X in polynomial time.	NP-Complete problems can be solved by deterministic algorithm in polynomial time.
Solution	To solve this problem, it must be a NP problem.	To solve this problem, it must be both NP and NP-hard problem.
Nature Of Problem	It is not a decision problem.	It is exclusively a decision problem.
Examples	-Halting problem -Vertex cover problem -Circuit-satisfiability problem etc.	-Minesweeper problem -Determining whether a graph has a Hamiltonian cycle. -Determining whether Boolean formula is satisfiable or not

- Examples:
- Knapsack problem
- Hamiltonian path problem
- vertex cover problem
- Boolen satisfiabiltiy problem
- clique problem

Prove that: Vertex Cover is NP complete

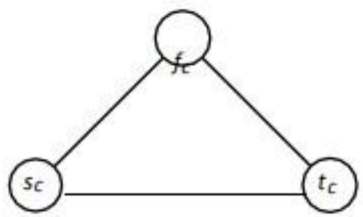
Given a graph $G = (N, E)$ and an integer k, does there exist a subset S of at most k vertices in N such that each edge in E is touched by at least one vertex in S?

- No polynomial-time algorithm is known Is
- in NP (short and verifiable solution):
  - If a graph is "k-coverable", there exists k-subset $S \subseteq N$ such that each edge is touched by at least one of its vertices
  - Length of S encoding is polynomial in length of G encoding
  - There exists a polynomial-time algorithm that verifies whether S is a valid k-cover
    - Verify that $|S| \leq k$
    - Verify that, for any $(u, v) \in E$, either $u \in S$ or $v \in S$

Class-TE Comp

- Reduction of 3-Sat to Vertex Cover:
- Technique: component design
  - For each variable a gadget (that is, a sub-graph) representing its truth value
  - For each clause a gadget representing the fact that one of its literals is true
  - Edges connecting the two kinds of gadget
- Gadget for variable u:

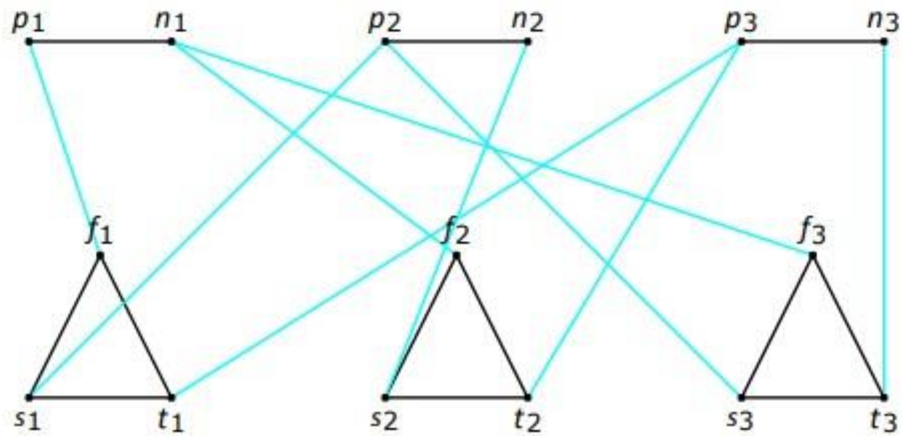
  - One vertex is sufficient and necessary to cover the edge
- Gadget for clause c:



  - Two vertices are sufficient and necessary to cover the three edges
- $k = n + 2m$, where n is number of variables and m is number of clauses

Class-TE Comp

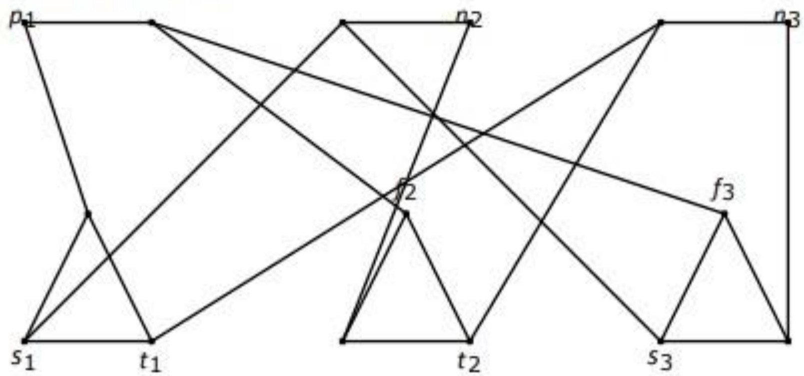
- Connections between variable and clause gadgets
  - First (second, third) vertex of clause gadget connected to vertex corresponding to first (second, third) literal of clause
  - Example: $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$



  - Idea: if first (second, third) literal of clause is true (taken), then first (second, third) vertex of clause gadget has not to be taken in order to cover the edges between the gadgets

Class-TE Comp

- Show that Formula satisfiable $\Rightarrow$ Vertex cover exists:
  - Include in S all vertices corresponding to true literals
  - For each clause, include in S all vertices of its gadget but the one corresponding to its first true literal
  - Example
    - $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3)$
    - x_1 true, x_2 and x_3 false



- Show that Vertex cover exists $\Rightarrow$ Formula satisfiable:
  - Assign value true to variables whose p-vertices are in S
  - Since $k = n + 2m$, for each clause at least one edge connecting its gadget to the variable gadgets is covered by a variable vertex
    - Clause is satisfied

Class-TE Comp

Reducibility

Definition Let L_1 and L_2 be problems. L_1 *reduces to* L_2 (also written $L_1 \propto L_2$) if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.

This definition implies that if we have a polynomial time algorithm for L_2 then we can solve L_1 in polynomial time. One may readily verify that $\propto$ is a transitive relation (i.e. if $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$).

SAT Problem :

- SAT means satisfiability problem.

- Boolean formula $f(a_1, a_2, \cdots, a_n)$ is satisfiable if there is a way to assign values to variable of formula such that it produce result 1.

- Example : $f(x, y, z) = (x \vee (y \wedge z)) \wedge (x \wedge z)$

sol^n :→

x	y	z	$y \wedge z$	$x \vee (y \wedge z)$	$x \wedge z$	$(x \vee (y \wedge z)) \wedge (x \wedge z)$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	1	0	0
1	0	0	0	1	0	0
1	0	1	0	1	1	1
1	1	0	0	1	[illegible]	[illegible]

- Function is true for the combinations ($x=1, y=0, z=1$) and ($x=1, y=1, z=1$) hence it is satisfiable.

- For given input sequence, it can be verified in linear time but with 'n' input 'n' variables there exists 2^n instances. Testing each of them takes $O(n2^n)$ time which says SAT $\in$ NP class problem.

3-Satisfiability:-

•Satisfiability's role as the first NP-complete problem implies that the problem is hard to solve in the worst case, but certain instances of the problem are not necessarily so tough. .

•**Input:** A collection of clauses C where each clause contains exactly 3 literals, over a set of Boolean variables V.

•**Output:** Is there a truth assignment to V such that each clause is satisfied? Since this is a more restricted problem than satisfiablity, the hardness of 3-SAT implies that satisfiability is hard. The converse isn't true, as the hardness of general satisfiability might depend upon having long clauses. We can show the hardness of 3-SAT using a reduction that translates every instance of satisfiability into an instance of 3-S.

3-Satisfiability:-

- **Theorem:** CNF-SAT is in NP complete.
- **Proof:** Let S be the Boolean formula for which we can construct a simple non-deterministic algorithm which can guess the values of variables in Boolean formula and then evaluates each clause of S.
- If all the clauses evaluate S to 1 then S is satisfied.
- Thus CNF-SAT is in NP-complete.

3-Satisfiability:-

•**Theorem:** 3SAT is in NP complete.

•**Proof:** Let S be the Boolean formula having 3 literals in each clause for which we can construct a simple non-deterministic algorithm which can guess an assignment of Boolean values to S.

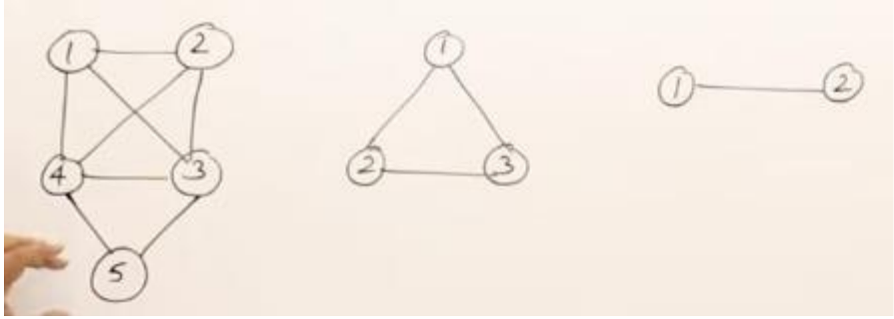
•If the S is evaluated as 1 then S is satisfied.

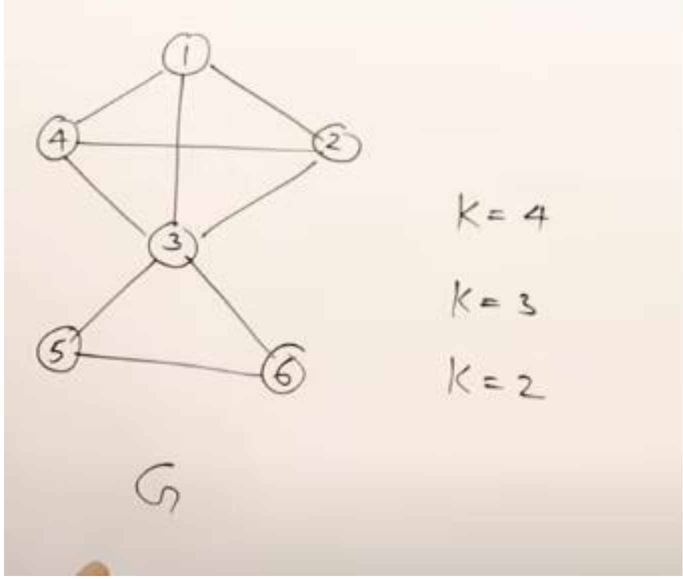
•Thus we can prove that 3SAT is in NP-complete.

Clique Decision Problem:-

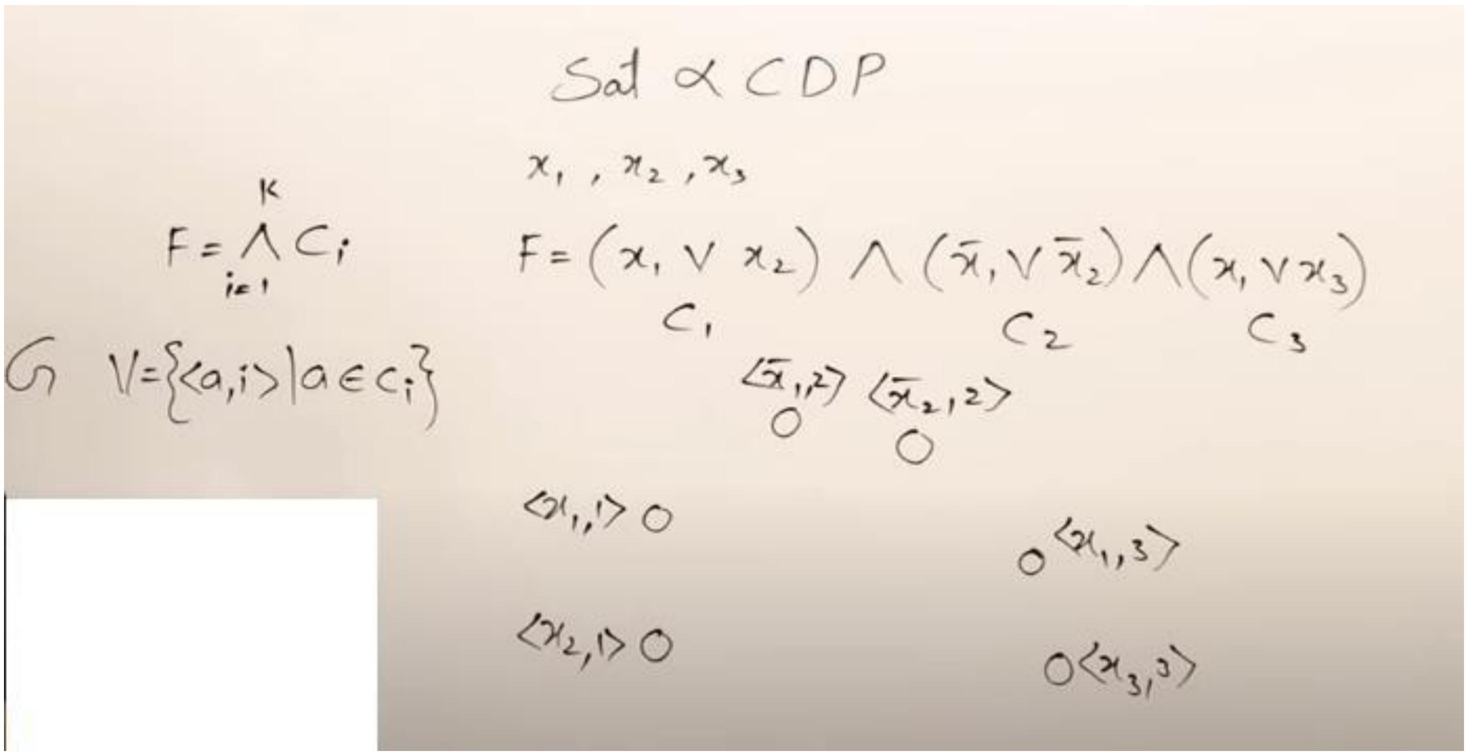
What is Clique?

It is sub graph of graph which is complete.





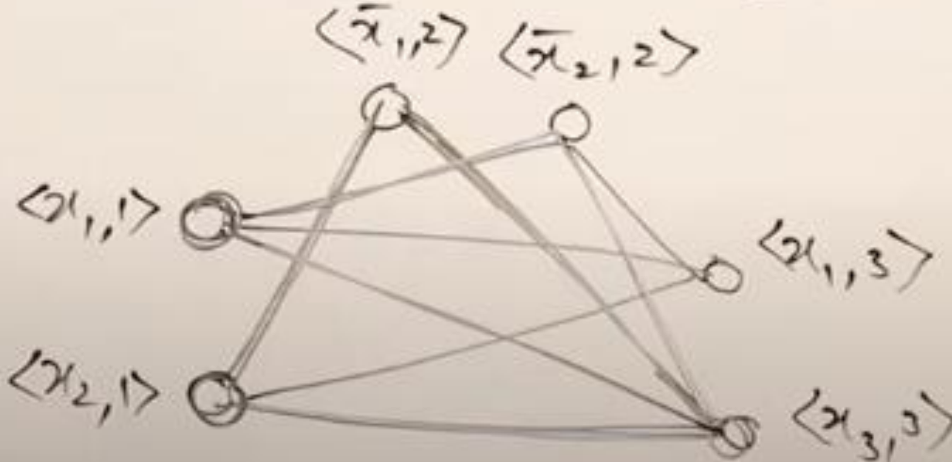
Prove that Clique Decision problem is NP Hard



$$F = (\overset{0}{x_1} \vee \overset{1}{x_2}) \wedge (\overset{1}{\bar{x}_1} \vee \overset{0}{\bar{x}_2}) \wedge (\overset{0}{x_1} \vee \overset{1}{x_3}) = 1$$

$C_1 = 1$ $\quad C_2 = 1$ $\quad C_3 = 1$

x_1	x_2	x_3
0	1	1



$k = 3$

Class-TE Comp

Hamiltonian Cycle:-

- Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once.
- A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path.

Hamiltonian Cycle:-

- **HAMILTONIAN CYCLE is in NP Complete.**
- **Proof:** Let A be some non-deterministic algorithm to which graph G is given as input.
- The vertices of graph are numbered from 1 to N. We have to call the algorithm recursively in order to get the sequence S.
- This sequence will have all the vertices without getting repeated.
- The vertex from which the sequence starts must be ended at the end.
- This check on the sequence S must be made in polynomial time n.

Hamiltonian Cycle:-

- **HAMILTONIAN CYCLE is in NP Complete.**
- Now if there is a Hamiltonian cycle in the graph then algorithm will output "yes".
- Similarly if we get output of algorithm as "yes" then we could guess the cycle in G with every vertex appearing exactly once and the first visited vertex getting visited at the last.
- That means A non-deterministically accepts the language HAMILTONIAN CYCLE.
- It is therefore proved that HAMILTONIAN CYCLE is in NP Complete.

Post Correspondence Problem (PCP)

•The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem.

•The PCP problem over an alphabet ∑ is stated as follows −

•Given the following two lists, **M** and **N** of non-empty strings over ∑ −

✓$M = (x_1, x_2, x_3, \ldots\ldots\ldots, x_n)$

✓$N = (y_1, y_2, y_3, \ldots\ldots\ldots, y_n)$

•We can say that there is a Post Correspondence Solution, if for some $i_1, i_2, \ldots\ldots\ldots\ldots\ i_k$, where $1 \leq i_j \leq n$, the condition $x_{i1} \ldots\ldots x_{ik} = y_{i1} \ldots\ldots y_{ik}$ satisfies.

Class-TE Comp

Post Correspondence Problem (PCP)

Example 1

Find whether the lists

M = (abb, aa, aaa) and N = (bba, aaa, aa)

have a Post Correspondence Solution?

Solution:-

	x_1	x_2	x_3
M	Abb	aa	aaa
N	Bba	aaa	aa

$\mathbf{x_2x_1x_3}$ **= 'aaabbaaa'**

and $\mathbf{y_2y_1y_3}$ **= 'aaabbaaa'**

We can see that

$\mathbf{x_2x_1x_3 = y_2y_1y_3}$

Hence, the solution is **i = 2, j = 1, and k = 3.**

Post Correspondence Problem (PCP)

Example 2

Find whether the lists **M = (ab, bab, bbaaa)** and **N = (a, ba, bab)** have a Post Correspondence Solution?

Solution:-

	x_1	x_2	x_3
M	ab	bab	bbaaa
N	a	ba	bab

In this case, there is no solution because −

$| \mathbf{x_2 x_1 x_3} | \neq | \mathbf{y_2 y_1 y_3} |$ (Lengths are not same)

Hence, it can be said that this Post Correspondence Problem is **undecidable**.

TRAVELLING SALESPERSON PROBLEM

Let G = (V, E) be a directed graph with edge costs C_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \alpha$ if $< i, j> \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex 1.

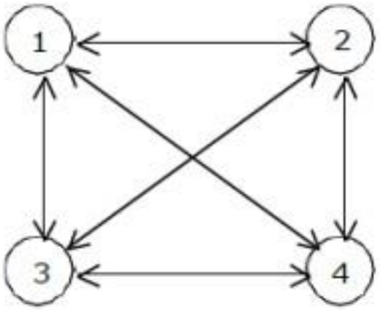
Let g (i, S) be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function g (1, V – {1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that:

- **Formula:-**
- $g(i, s) = \min\{cij + g(J, s - \{J\})\}$
- **Time Complexity:-**
- **O (n 2n).**

TRAVELLING SALESPERSON PROBLEM

Example 1:

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix = $\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

Solution:-

TRAVELLING SALESPERSON PROBLEM

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\}$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \leq i \leq n$. So,

$$g(2, \Phi) = C_{21} = 5$$

$$g(3, \Phi) = C_{31} = 6$$

$$g(4, \Phi) = C_{41} = 8$$

Using equation – (2) we obtain:

$$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}, c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$$

$$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$$
$$= \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$$

$$g(3, \{4\}) = \min \{c_{34} + g(4, \Phi)\} = 12 + 8 = 20$$

$$g(4, \{3\}) = \min \{c_{43} + g(3, \Phi)\} = 9 + 6 = 15$$

TRAVELLING SALESPERSON PROBLEM

Therefore, $g(2, \{3, 4\}) = \min\{9 + 20, 10 + 15\} = \min\{29, 25\} = 25$

$g(3, \{2, 4\}) = \min\{(c_{32} + g(2, \{4\}), (c_{34} + g(4, \{2\})\}$

$g(2, \{4\}) = \min\{c_{24} + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min\{c_{42} + g(2, \Phi)\} = 8 + 5 = 13$

Therefore, $g(3, \{2, 4\}) = \min\{13 + 18, 12 + 13\} = \min\{41, 25\} = 25$

$g(4, \{2, 3\}) = \min\{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$

$g(2, \{3\}) = \min\{c_{23} + g(3, \Phi\} = 9 + 6 = 15$

$g(3, \{2\}) = \min\{c_{32} + g(2, \Phi\} = 13 + 5 = 18$

Therefore, $g(4, \{2, 3\}) = \min\{8 + 15, 9 + 18\} = \min\{23, 27\} = 23$

$$g(1, \{2, 3, 4\}) = \min\{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$$
$$= \min\{10 + 25, 15 + 25, 20 + 23\} = \min\{35, 40, 43\} = 35$$

The optimal tour for the graph has length = 35

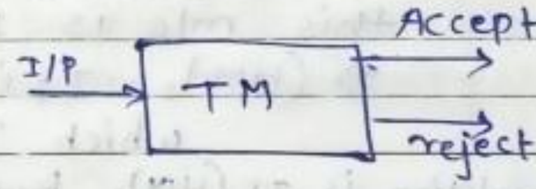
The optimal tour is: 1, 2, 4, 3, 1.

A turing - unrecognizable Language :-

1> A TM can accept Type 0. Grammer.

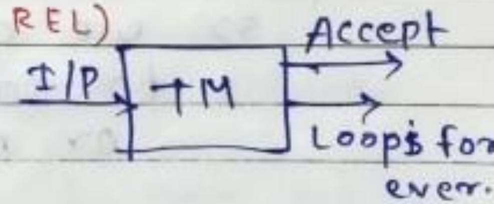
2> Type 0 Grammer are called recursively enumerable Lang.

3> Recursive Lang :- (RE)



It is said to be recursive if there exists a TM that accepts every string of the Lang. & every string is rejected if it is a not belonging to that Lang.

4> Recursively Enumerable Lang :- (REL)



A Lang is said to be recursive if there exists a turing M/c that accepts every string belonging to that lang. & if the String does not belong to that lang then It can cause a TM to enter in an infinite loop.

* Show that for two recursive languages L_1 and L_2 each of the following is recursive.

i) $L_1 \cup L_2$ ii) $L_1 \cap L_2$ iii) L_1'

→

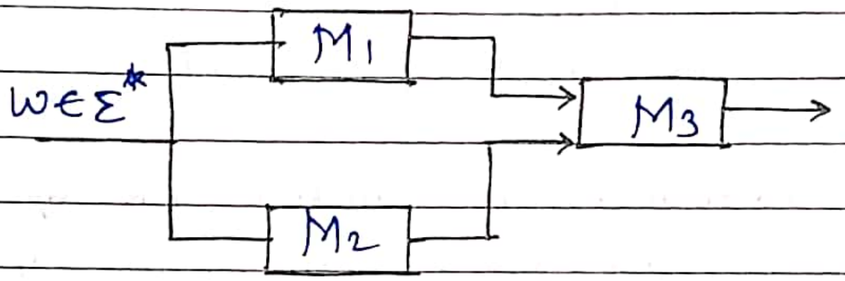
1) $L_1 \cup L_2$ is recursive :-

→ Let the TM M_1 decides L_1 and M_2 decides L_2.

- If a word $w \in L_1$ then M_1 returns 'Yes' else it returns 'No'

Similarly if word $w \in L_2$ then M_2 return 'Yes' else 'No'

- Let us construct TM M_3 as shown in figure



A Turing Machine for $L_1 \cup L_2$

- output of machine M_1 is written on the tape of M_3
- output of m/c M_2 is written on the tape of M_3
- The m/c M_3 returns 'Yes' as o/p, if atleast one of the o/p of M_1 or M_2 is 'Yes'.

- It should be clear that M_3 decides $L_1 \cup L_2$.
As both L_1 & L_2 are turing decidable. after finite time both M_1 & M_2 will halt with answer 'Yes' or 'No'
- The machine M_3 gets activated after M_1 or M_2 is halted. (stopped)
- The machine M_3 halts with answer 'Yes' if $w \in L_1$ or $w \in L_2$ else M_3 halts with o/p 'No'

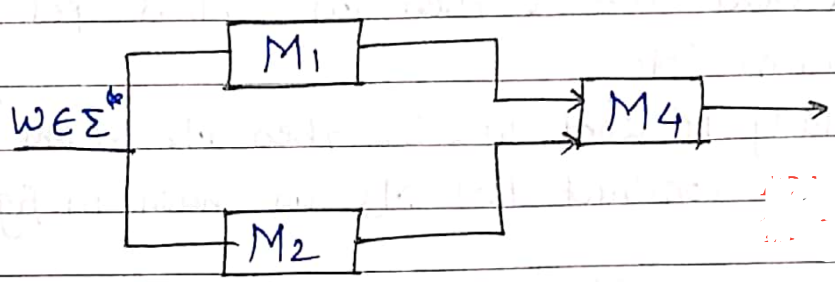
Thus, $L_1 \cup L_2$ is turing decidable or Recursive.

ii) $L_1 \cap L_2$ is recursive :-

→ - Let the turing machine M_1 decides L_1 and M_2 decides L_2. If word $w \in L$ then M_1 return 'Yes' else it returns 'No'

Similarly if $w \in L_2$ then M_2 returns 'Yes' else 'No'

- Let's consider TM M_4 as shown in fig.



A Turing Machine for $L_1 \cap L_2$

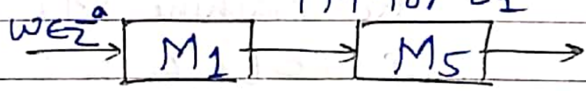
- o/p of machine M_1 is written on the tape of M_4.
- o/p of m/c M_2 is written on the tape of M_4.
- The m/c M_4 returns 'Yes' as o/p, If both M_1 and M_2 are "Yes" otherwise M_4 returns 'No'
- It should be clear that M_4 decides $L_1 \cap L_2$. As both L_1 & L_2 are turing decidable, after finite time both M_1 & M_2 ~~are~~ will halt with answer 'Yes' / 'No'.
- The m/c M_4 is activated after M_1 & M_2 are halted. The m/c M_4 halts with answer 'Yes' if $w \in L_1$ & $w \in L_2$ else M_4 halts with answer 'No'.

iii) L_1' is recursive :-

→ Let the TM ~~decides~~ M_1 decides L_1

- Let's construct TM M_5

TM for L_1'



- o/p of machine m_1 is written on the tape of M_5.
- The m/c M_5 returns "Yes" as o/p if the o/p of M_1 is 'No' otherwise M_5 returns 'No'.
- It should be clear that M_5 decides L_1'. As L_1 is turing decidable after finite time M_1 will halt with answer Yes / No.
- The M/c M_5 is activated after M_1 halts.