1

Computational complexity

Computational complexity theory is a brach of theory of computation in computer Science & mathematics that focuses on classifying computational problems according to their inherent difficulty (or complexity). It includes:

1) The efficiency of algorithms
2) The inherent difficulty of problems of practical and/or theoritical importance.

In other words,

Computational complexity theory is a part of theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are:

(1) time (how many steps it takes to solve a problem).

(2) space (how much memory it takes)

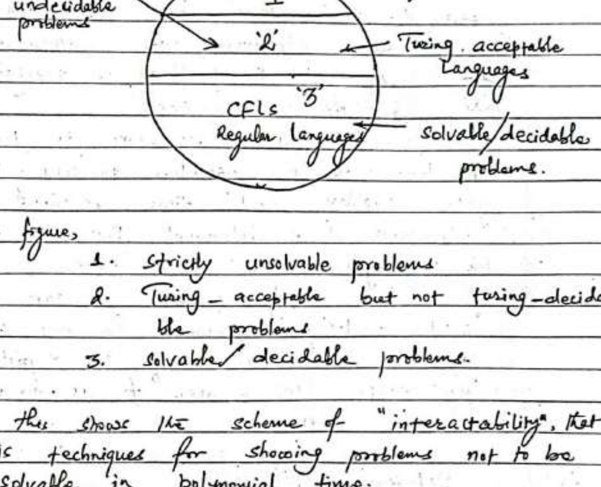
Other resources may be, how many parallel processors are needed to solve a problem in parallel.

Computability theory deals only with whether a problem can be solved at all, regardless of the resources required. Much of complexity theory deals with decision problems. A decision problem is a problem where the answer is always yes/No.

Introduction:- (Interactibility)

Now we focus on the problems that are decidable, and which of them can be computed by turing machine that run in an amount of time that is polynomial in the size of the input.

Below given figure shows, the view about the complexity of the problems.



In figure,

1. Strictly unsolvable problems
2. Turing-acceptable but not turing-decidable problems
3. Solvable/decidable problems.

So, this shows the scheme of "interactability", that is techniques for showing problems not to be solvable in polynomial time.

G1 The Classes P:-

We have studied that a solvable problem is one that be solved by a particular algorithm. In this case problem is solvable "in principle" that there is a certain algorithm to solve this problem. But in practice algorithms may require a lot of space and time. When the space and time required for implementing the steps of the particular algorithm are reasonable, we can say that the problem is tractable, that is solvable in practice.

"A decision problem is tractable if there is an algorithm to solve the given problem and time required is expressed as a polynomial $P(n)$, n being the length of the input string"

Usually problems are intractable if the time required for any of the algorithm (which can solve the problem) is at least $f(n)$, where f is an exponential function of n.

Definition of class P:

A language 'L' is said to be in 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 'L' is also called polynomial decidable.

Theorem: 'P is closed under complement'

proof:-

If a language 'L' is decidable by a polynomial bounded turing machine. TM then complement is decided by the version of TM that inverts yes and no. Obviously, the polynomial bound is unaffected.

Note:-

Computing practice reveals that many problems which are solvable in principle, can not be solved in practice due to excessive time requirements.

Ex- Travelling sales man problem for n-cities, need $(n-1)!$ itineraries. i.e. If there is 10 cities, then $(10-1)!$ $= 362,880$ itineraries.

So, theoritically computable may not be practically computable.

Polynomially bounded Turing Machine:-

A turing machine $M = (K, \Sigma, \delta, s, H)$ is said to be polynomially bounded Turing Machine if there is a polynomi p(n) such machine always halts after p(n) steps, where 'n' is the length of the input.

polynomially decidable languages:-

A language is called polynomially decidable if there is a polynomially bounded Turing machines that decides it. The class of all polynomially decidable languages are denoted by P.

Examples of problems in class-P:-

These are several examples of problems in class-P such as.

1. Eulerian and Hamilton graph problem
2. Integer partition problem
3. Equivalence of finite Automata
4. Kruskal's algorithm for minimum weight spanning tree.

1. Eulerian & Hamilton Graph:-

These two graphs can be defined as class-P problems. Let us define Euler cycle first.

Euler cycle: Given a graph G, is there a closed path in G that uses each edge exactly

once?

A graph G is eulerian iff:

(a) for any pair of nodes u & v ∈ V, neither of which is isolated, there is a path from u to v.

(b) All nodes have equal numbers of incoming and outgoing edges.

The conditions (a) & (b) can be tested in polynomial time by computing reflexive-transitive closure of the graph & testing the connections in all possible ways. We know that reflexive-transitive closure can be computed in polynomial number of steps.

Again, number of incoming & outgoing nodes can be obviously done in polynomial time.

Similarly, for Hamilton cycle:

Given a graph G, is there a cycle that passes through each node of G exactly once?

Here, the nodes, not the edges that must be traversed exactly once.

It can never be computed in polynomial time as. Examine all possible permutations the nodes, for each test whether it is a Hamilton cycle.

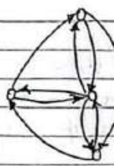


Fig. (a) Eulerian + Hamilton graph

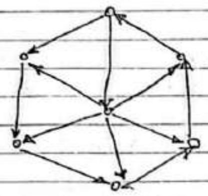


Fig (b) Hamilton graph.

(4.) Kruskal's algorithm:- Minimum weighted spanning tree:-

A spanning tree is a subset of all edges such that all nodes are connected through these edges, yet there are no cycles. A minimum weighted spanning tree has the least possible weight of the edges.

[obtained through Kruskal's algorithm]

In figure, the bold lines shows the edges for minimum-weight spanning tree.

This problem is class P problem because it is possible to implement Kruskal's algorithm (with a computer) on a graph with m nodes and x edges in time $O(m + x\log x)$

6.2 The class - NP

A non-deterministic Turing machine $M = (K, \Sigma, \Delta, s, H)$ is said to be polynomially bounded if there is a polynomial $p(n)$ such that no computation of this machine continues for more than polynomially many steps.

Now, we can define NP as

We can define NP (for nondeterministic polynomial) to be the class of all languages that are decided by a polynomially bounded Turing machine (TM).

The class-P versus class-NP:-

Although the definitions of 'P' and 'NP' are seems to be similar, but there is a vast difference between them.

When a Language 'L' is in 'P', the number of moves to test whether any string of length n is less than or equal to $p(n)$.

When 'L' is in NP, the number of moves for testing is less than $p(n)$ only f strings accepted by Turing machine. Thus, in class of NP, the bound p(n) is useful only when we are able to find string 'w' in 'L' but is very difficult.

Summary of class-P & class-NP:-

The class-P:-

① ⇒ P is a class of problems that can be solved deterministically in polynomial time.

The class P is important because:

(a) It is invariant over all models of computation.

(b) practical problems in P class have efficient (low-degree polynomial) algorithms.

The class-NP:-

NP is the class of problems that can be solved non-deterministically in polynomial time.

The class NP is also invarient over all resonal models of computation.

Clearly, $P \subseteq NP$, but it is not known whether or not $P = NP$.

Examples of class-NP:-

① Travelling salesman problems:-

The travelling salesman problem with four-nodes is simple, since there can never be more than two different Hamilton cycles. But in m-node graphs, the number of distinct cycles grows as $O(m!)$, which is more than 2^{cm} for any constant C.

6.8. NP- Completeness:- CC — 7

let 'L' be a problem (Language) in NP. we say 'L' is NP-Complete if the following statements are true about 'L'.

1. L is in NP
2. for every Language L' in NP, there is a polynomial-time reduction of L' to L.

An example of NP-complete problem is the travelling sales man problem.

once, we have some NP-Complete problems, we can prove a new problem to be NP-complete by reducing some known NP-Complete problem to it by using the polynomial-time reduction.

polynomial-time Reduction:-

A problem P_1 is polynomially reducible to problem P_2 if there exists a polynomial time algorithm that transforms every instance I_1 to P_1 to an instance I_2 of P_2 such that the answer to I_1 is "yes" ($I_1 \in P$ if and only if answer to I_2 is "yes" ($I_2 \in P$

CC-12

Np-Hard problems:-

Some problems are so hard that we can prove condition (2) of the definition of Np-completeness (every languages in Np-reduces to L in polynomial time). we can not prove condition 1, that 'L' is in Np. If so, we call L as Np-hard problem. Ex- [Tower of Hanoi]
[Halting problem]

Exponentially Bounded Turing Machine:-

A Turing machine $M = (K, \Sigma, \delta, s, H)$ is said to be exponentially bounded if there is a polynomial $p(n)$ such that the machine always halts after at most exponentially many steps.

Theorem.

$\boxed{\text{If } L \in Np, \text{ then } L \in exp.}$

Od

CHAPTER - 5.

Ud-1

*5 Undecidability:-

As we know that, recursive languages are those language which are accepted by at least one Turing machine and these sets of recursive languages are subclass of the recursively enumerable languages.

A problem whose language is recursive is said to be decidable. Otherwise problem is undecidable i.e., a problem is undecidable if there exist no algorithm that takes on input an instances of the problem and determine whether the answer to that instance is 'yes' (y) or 'no' (n).

Turing acceptable:-

We know that a Turing machine TM accepts $\omega \in \Sigma^*$ if TM halts on input ω, language 'L' is turing acceptible if there is some turing machine that accepts it. Then is:

1) If 'L' is Turing - decidable then 'L' is Turing acceptable

2) If 'L' is Turing - decidable then so is '$\bar{L}$'

*5.1 Church-Turing Thesis (or Church's Thesis):-

Turing machines that decides Language and Compute functions and therefore halts on every input are useful computational devices. Turing machine corresponding to formal notion of algorithm must halt on all inputs and therefore such machines are called algorithms. This principle is called "Church – Turing thesis". This is not a theorem and so, can not be proved mathematically.

It also says that problems unsolvable by turing Machine are impossible problems. It is believed that the Turing machine is the ultimate calculating mechanism.

In general the thesis may be written in short as, " No computational procedure will be considered as an algorithm unless it can be represented as a Turing machine."

*5.2 Universal Turing Machine:-

The turing machine that takes other turing machine (TM) and their inputs (ω), in encoded form and is capable to run 'TM' on 'ω' is called universal Turing machine (TM_U).

Generally, we consider a Turing machine as an "unprogrammable" piece of hardware, specialized at solving one particular problem, with instructions that are "hard-wired" at the factory.

But, we can define Turing

machine not only as hardware but as soft[ware] also. Such interpretation of Turing machine [leads] to the formulation of the above defined [Univer]sal Turing machine. That is, we shall [show] that there is a certain "generic" Tu[ring] machine that can be programmed, about [the] same way that a general purpose com[puter] can, to solve any problem that ca[n be] solved by Turing machine.

So, considering a Turing m[achine] as a program written in any program[ming] language;

programs written in this program[ming] language can be interpreted by another p[rogram] written in same language called Uni[versal] Turing machine.

Let $TM = (K, \Sigma, \delta, s, H)$ be a [Turing] machine, and let 'i' and 'j' be the smalles[t inte]gers such that $2^i \geq |K|$, and $2^j \geq |\Sigma| + 2$. Th[en] each state in K will be represented as a 'q' followed by a binary string of length 'i'[, and] each tape symbol is of letter 'a' followed b[y a string] of 'j' bits. Some representations are:-

blank symbol $\sqcup$ — $a0^j$

left end symbol $\triangleright$ — $a0^{j-1}1$

left move $\leftarrow$ — $a0^{j-2}10$

Right move $\rightarrow$ — $a0^{j-2}11$

s — $q0^i$

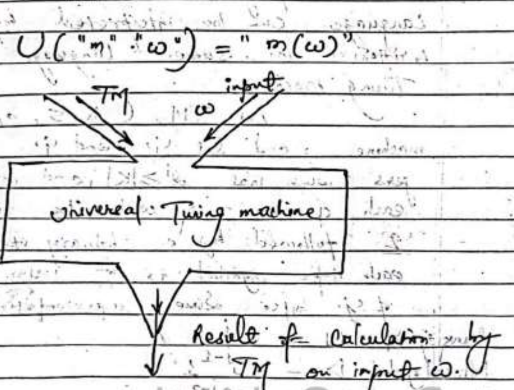
$a - a0^{j-3}100$
$b - a0^{j-3}101$
$c - a0^{j-3}110$
$d - a0^{j-3}111$

only upto d i.e. 8 symbols can be encoded.

and the transition δ with (q, a, p, b), where q and p are states and a & b are tape symbols.

Then, Universal Turing machine 'U' takes two arguments, one for description of Turing machine TM and next input string ω.

$$U(\text{"m"}\ \text{"}\omega\text{"}) = \text{"}m(\omega)\text{"}$$



Follow book page.

Example of Universal Turing machine (TM_U):-

Consider a Turing machine $M = (K, \Sigma, \delta, s, \{h\})$, where, $K = \{s, q, h\}$, $\Sigma = \{\sqcup, \triangleright, a\}$ and δ is defined as follows:

state	symbol	δ
s	a	$(q, \sqcup)$
s	$\sqcup$	$(h, \sqcup)$
s	$\triangleright$	$(s, \rightarrow)$
q	a	(s, a)
q	$\triangleright$	$(q, \rightarrow)$
q	$\sqcup$	$(s, \rightarrow)$

There are three states and three symbols in Σ. So, we have $i = 2$ and $j = 3$.

$[\because 2^i \geq |K| \ \& \ 2^j \geq |\Sigma| + 2]$

The states & symbols can be represented as follows:

state/symbol	representation
s	q00
q	q01
h	q11
$\sqcup$	a000
$\triangleright$	a001
$\leftarrow$	a010
$\rightarrow$	a011
a	a100

Thus, the representation of the string,

Ud-6

$\triangleright aa \sqcup a$ is

" $\triangleright aa\sqcup a$" = a001a100a100a000a100.

The representation "M" of the Turing Machine TM_m is the following string.

"M" = (q00, a100, q01, a000), (q00, a000, q11, a000) ------ (q01, a001, q01, a011).

This is the universal Turing machine version of the above transition table of the Turing machine.

*5.3 The Halting problem:- *

"For an arbitrary given Turing Machine 'TM' and input, ω, there is no algorithm that decides whether or not 'TM' accepts 'ω'"

Such problems for which no algorithm exists are known as undecidable or unsolvable problems.

Telling whether a given Turing Machine halts on given input is also an undecidable problem and is called "Halting problem" for Turing Machine.

"Prove that Halting problem is Unsolv

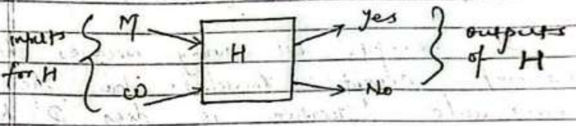
proof:

Let us suppose, there exist a turing 'H' that takes other turing machine input 'ω' as input and decides or not 'M' Halts on 'ω'.

say.

'H' says "yes" if 'M' ha

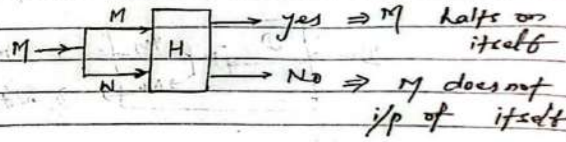
and 'H' says 'no' if 'M' does not



Using this algorithm, we can know if machine halts on input of itself.

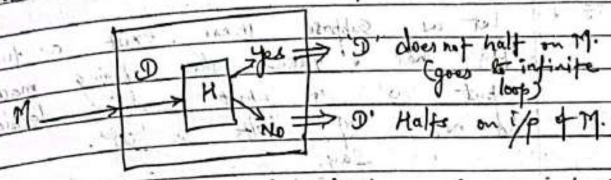
[By taking 'M' itself as input &c 'M' (ω = M)]

So, 'H' can be modified as



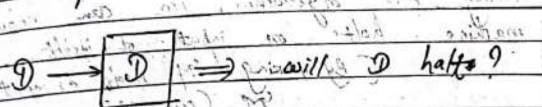
This valuable machine 'H', that solves

problems can be used to construct another machine 'D', that halts only if 'H' says 'no'.

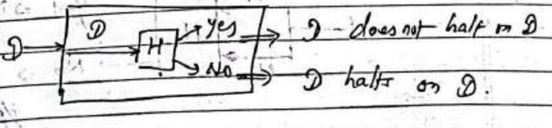


That means, if Machine 'M' with input 'M' is given to 'D' then 'D' halts if and only if (iff) M does not halt on 'M'.

i.e. 'D' accepts all turing machines that does not accept themselves. Now, the unanswer-able question is 'does 'D' halt on input of 'D'?



The answer would be
"D halts on 'D' iff D does not halt on D"



The statement is self-contradictory. Which means, our assumption that 'H' exists is wrong.

Hence, 'H' doesn't exist i.e "Halting problem is unsolvable".

* 5.4 Undecidable Problems about Turing Machines:-

1) Given a turing machine (TM) 'T' and 'w', does 'T' halts on 'w'? (Halting problems)

2) Given a TM, T, does 'T' halt on empty tape?

3) Given a TM, T, is there any string on which 'T' halts?

4) Given a TM, T, does 'T' halt on every input string?

5) Does two TMs T_1 and T_2 halts on same input strings?

6) Given a TM, 'T' is the language that 'T' semi-decides regular? Context free? Recursive?

The unsolvability of halting problem (that we proved earlier) implies the unsolvability of many other problems in mathematics and computer science. Such problems are proved unsolvable by reducing unsolvability of halting problem to those problems. PCP (post correspondence problem). Tiling problems etc are some examples.

Undecidable problems about Grammars (unsolvable) (unrestricted Grammars):-

The unsolvable problems related to grammars are as follows:

1. For a given Grammar 'G' and string 'w', to determine whether $w \in L(G)$?
2. For a given grammar G, does $\epsilon \in L(G)$?
3. For given two grammars G_1 & G_2, is it is not possible to determine is $L(G_1) = L(G_2)$?
4. For an arbitrary grammar G, to determine whether $L(G) = \phi$?
5. There is a certain fixed grammar G_0, such that it is undecidable to determine whether any given string w is in $L(G_0)$.

etc are undecidable problems on grammar and yet unsolvable also.

Rice's Theorem:

Every Non-trivial property of a Recursively enumerable Language are enumer...

Properties of Recursive & Recursively Enumerable Languages:-

Recursive language:-

A Language is recursive if there exist a turing machine that accepts every string of the language and rejects every string that are not in the language.

→ So, we are sure about the rejection of strings which are not in the given lang...

→ Recursive language always halts the Turing machine.

→ Recursive language is subset of recursively enumerable Language.

Recursively Enumerable (RE) Language:-

The set of RE languages are precisely those language that can be accepted by the

Turing machine.

→ strings that are not in the language may be rejected or may cause the Turing machine to go into an infinite loop.

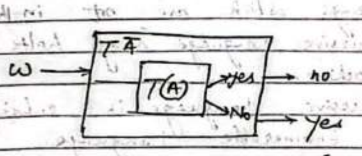
→ Set of RE language is larger than that of recursive.

* **properties:-**

Theorem. "Statement"

" If A is recursive, its complement is also recursive".

proof:- For every recursive language, TM [T(A)] always halts and say "yes" if $\omega \in A$ & 'No' if $\omega \notin A$. We can easily construct TM for $\bar{A}$, using T(A) as

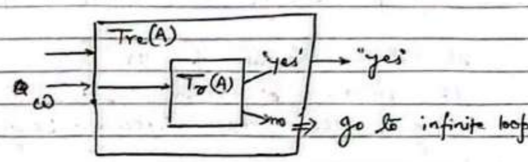


Then This T.M, $T(\bar{A})$ say yes if $\omega \notin A$ i.e. $\omega \in \bar{A}$ and no if $\omega \in A$ i.e. $\omega \notin \bar{A}$

Hence $\bar{A}$ is also recursive.

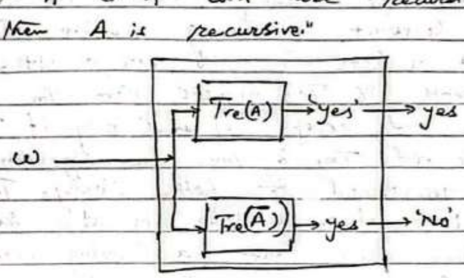
* "If A is recursive, it is also recursively enumerable"

proof:



Here, Tr(A), Turing machine that decides A can be used to construct Tre(A) that is turing machine that semi-decides A. So, A is also recursively enumerable.

* "If A & $\bar{A}$ both are recursively enumerable then A is recursive."



As A and $\bar{A}$ are recursively enumerable, let, turing machine TM's Tre(A) & Tre-($\overline{A}$), semi-decides Language A & $\overline{A}$ respectively. for any input ω, either Tre(A) accepts ω if $\omega \in A$ or Tre($\overline{A}$) accepts.

We can make use of these two machines to design ~~a~~ machine Tr(A) as shown in figure above that decides Language A.

$\therefore$ A is recursive.

** "The union of two recursive Language is recursive."

proof:-

Let L_1 & L_2 be two recursive languages accepted by Turing machines Tm_1 & Tm_2. We construct a Turing machine Tm, which first simulate Tm_1. if Tm_1 accepts, then Tm accepts. If Tm_1 rejects then Tm simulates Tm_2 and accepts if and only if Tm_2 accepts. Since both Tm_1 & Tm_2 are algorithm, so Tm is guaranteed to halt. Clearly, Tm accepts $L_1 \cup L_2$. Hence, $L_1 \cup L_2$ is also recursive. Since there exist a Turing machine Tm for it.

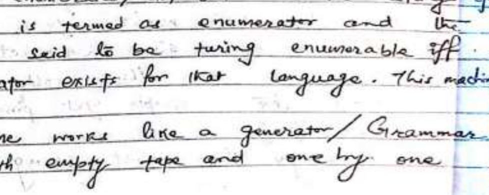
Statement

"The union of two recursively enumerable languages is recursively enumerable."

proof:-

Let L_1 & L_2 are recursively enumerable languages and their enumerative Turing machines are Tm_1 & Tm_2 respectively.

let us design a Turing machine Tm which can simulate Tm_1 & Tm_2 simultaneously on separate tape. If either accepts then Tm accepts as follows.



* Turing Machine as an Enumerator:-

Machine that enumerates / Lists out all the strings of a language is termed as enumerator and the Language is said to be turing enumerable iff such enumerator exists for that language. This machine behaves as:

1) This machine works like a generator / Grammar
2) Start with empty tape and one by one

list out all the strings belonging to Language into the tape.

Theorem:- "A Language is recursively enumerable if and only if (iff) it is turing enumerable."

proof:- **Case I.**

"If Language is recursively enumerable, then it is turing enumerable"

proof:-

let 'T' be turing machine that semi-decides the recursively enumerable, we need to show 'L' is turing enumerable, we need to construct an enumerator, 'E' for L.

Simple approach for E would be to feed every possible input string, 'w' into T one-by-one and when-ever T accepts w, just print 'w' in the tape. But the big problem in this approach is that the machine 'T' is not guaranteed to halt for $w \notin L$. 'T' might go into infinite loop and so will our enumerator, E.

This problem can be solved by using the simple, yet powerful technique of dovetailing.

→ Arrange the input string in lexicographic

order (increasing number of strings alphabets symbols eg- 0, 1, 00, 01, 10, 11, 000, 001, ---)

In 1st phase, Carry out 1st step of computation of T on 1st string (input)

In 2nd phase, carry out 2nd step of computation 1st input string & 1st step on 2nd input string.

Continue in similar fashion and so on, so that,

in n^{th} phase, n^{th} step of computation of on 1st i/p string is carried out $(n-1)^{step}$ on 2nd and so on.

In the process, if for some string 'w' T accepts and halts, just write 'w', on tape and continue processing for other. In this way, sooner or later, all input strings will be processed and $w \in L$ will be printed in the tape of our Enumerator E.//

A point to be noted here the order in which enumerator, E prints the strings 'w' is not necessarily lexicographic. Longer strings might get printed earlier than the shorter ones.

Case II:- "If Language is turing enumerator recursively Enumerable."

proof:-

Let E be enumerator for turing enumerable Language 'L'. T be a turing machine that takes input 'ω' and compares 'ω' with each strings generated/printed by E. If a match is found, T accepts 'ω' and halts as $\omega \in L$, otherwise just keep on comparing 'ω' with enumerator output. So, T semi-decides L, hence L is recursively enumerable (RE).

Hence proved

Theorem:-

"A language is recursive if and only if (iff) it is Lexicographically Turing enumerable."

proof:-

Case I:-

"Language is Recursive implies 'L' is Lexicographically Turing Enumerable"

proof:-

Let T be turing machine that Decides the recursive language 'L'. Construct an enumerator, E that feeds all possible

input strings 'ω' to 'T' in lexicographic order (increasing # alphabet symbols eg. 0, 1, 00, 01, 10, 11, 000, - - -)

As, T is guaranteed to halt on every input and decide whether or not $\omega \in L$, print $\omega \in L$ into tape and skip all $\omega \notin L$. This way, E will print ~~out~~ on its tape all $\omega \in L$ in lexicographic order and hence 'L' is lexicographically turing Enumerable.

Case- II:-

"L is lexicographically turing Enumerable implies 'L' is recursive."

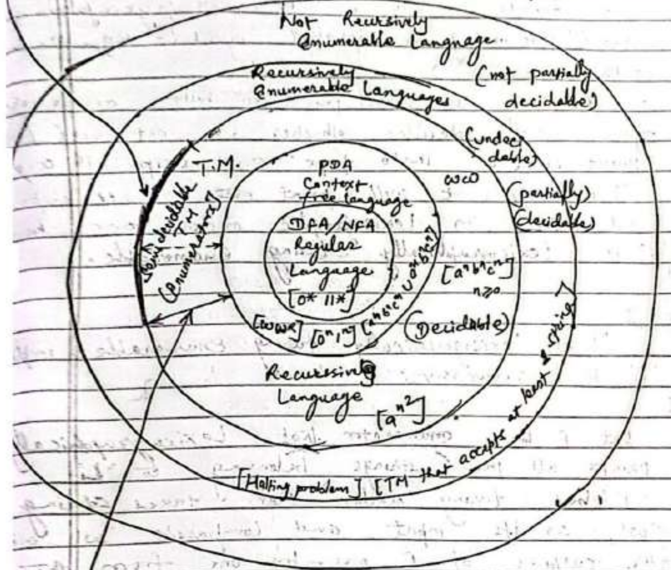
proof:-

Let E be enumerator that lexicographically prints all the strings belonging to 'L'. 'T' be turing machine that takes string 'ω' as its input and compare's 'ω' with the outputs of E, one-by-one from the begining. When a match is found, T simply accepts 'ω'. As the order in which strings of L are enumerated by E is lexicographic, T rejects 'ω' if it reads the string that should appear later than 'ω' in lexicographic order. Hence, as 'T' decides the language L, L is Recursive.

proved

(Boarder bet decidibility & undecidibility)

Different levels of Languages:-



[Computational Complexity theory lies here.]

fig:- Different levels of Languages, corresponding machines accepting them with examples.

* Algorithms for CFG:-

→ Given a CFG, Construct a PDA ⇒ polynomial time

→ Given, a PDA, Construct a CFG ⇒ polynomial time

→ Membership problem for CFL is polynomially decidable. (i.e there is polynomial time algorithm that can decide if input $x \in L(G)$ or not).

End of Chapter- 5.

CHAPTER- 5 & 6 Completed