

UNIT-VIII

COMPUTABILITY THEORY

CONTEXT SENSITIVE LANGUAGE

- ✓ A Context Sensitive Grammar is a 4-tuple, $G = (N, \Sigma, P, S)$ where:

$N \rightarrow$ Set of non terminal symbols

$\Sigma \rightarrow$ Set of terminal symbols

$S \rightarrow$ Start symbol of the production

$P \rightarrow$ Finite set of Productions

Where all the rules in P are of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ where $\alpha, \beta, \gamma \in (N \cup \Sigma)^+$ i.e. γ should be non empty.

In addition a rule of the form $S \rightarrow \epsilon$ provided S does not appear on the right hand side of any rule.

- ✓ The language generated by the context sensitive grammar is called as context sensitive language.
- ✓ In theoretical computer science, a context-sensitive language is a formal language that can be defined by a context-sensitive grammar.

COMPUTATIONAL PROPERTIES

- ❖ Computationally, a context-sensitive language is equivalent with a linear bounded nondeterministic Turing machine, also called a linear bounded automaton.
- ❖ That is a non-deterministic Turing machine with a tape of only kn cells, where n is the size of the input and k is a constant associated with the machine.
- ❖ This means that every formal language that can be decided by such a machine is a context-sensitive language, and every context-sensitive language can be decided by such a machine.

PROPERTIES OF CONTEXT-SENSITIVE LANGUAGES

- ❖ The union, intersection, concatenation and Kleene star of two context-sensitive languages is context-sensitive.
- ❖ The complement of a context-sensitive language is itself context-sensitive.
- ❖ Every context-free language is context-sensitive.

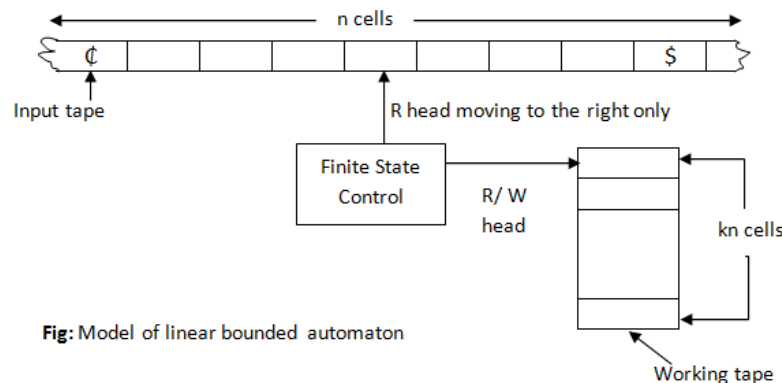
LINEAR BOUNDED AUTOMATA

MODEL OF LBA

- ✓ The model of LBA is important because of (a) the set of context sensitive languages is accepted by the model and (b) the infinite storage is restricted in size but not in accessibility to the storage in comparison with Turing machine model.
- ✓ It is called as linear bounded automaton (LBA) because a linear function is used in restricts (to bound) the length of the tape.
- ✓ It should be noted that the study of the context sensitive languages is important from practical point of view because many compiler languages lie between context sensitive and context free languages.
- ✓ A linear bounded automaton is a nondeterministic Turing machine which has a single tape whose length is not infinite but bounded by a linear function of the length of the input string.
- ✓ The models can be described formally by the following set format:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, b, \$, F)$$

- ✓ All the symbols have the same meaning as in the basic model of Turing machines with difference that the input alphabet Σ contains two special symbols $\$$ and $\%$.
- ✓ $\%$ is called the left end marker which is entered in the left most cell of the input tape and prevents the R/W head from getting off the left end of the tape.
- ✓ $\%$ is called the right end marker which is entered in the right most cell of the input tape and prevents the R/W head from getting off the right end of the tape.
- ✓ Both the end markers should not appear on any other cell with in the input tape and the R/W head should not print any other symbol over both the end markers.
- ✓ Let us consider the input string w with $|w| = n-2$. The input string w can be recognized by an LBA if it can also be recognized by a Turing machine using no more than kn cells of input tape, where k is a constant specified in the description of LBA.
- ✓ The value of k does not depend on the input string but is precisely a property of the machine. Whenever we process any string in LBA, we shall assume that the input string is enclosed within the end markers $\%$ and $\%$.
- ✓ The above model of LBA can be represented by the block diagram given as follows:



- ✓ There are two tapes one is called the input tape and the other is called the working tape. On the input tape the head never prints and never moves left.
- ✓ On the working tape the head can modify the contents in any way, without any restriction.
- ✓ In the case of LBA an ID is denoted by (q, w, k) where $q \in Q$, $w \in \Gamma$ and k is some integer between 1 and n .
- ✓ The transition of ID's is similar except that k changes to $k-1$ if the R/W head moves to the left and to $k+1$ if the head moves to the right.
- ✓ The language accepted by the LBA is defined as the set

$$\{w \in (\Sigma - \{\$, \#\})^* \mid (q_0, \#w\$, 1) \vdash^* (q, \alpha, i) \text{ for some } q \in F \text{ and for some integer } i \text{ between } 1 \text{ \& } n\}$$

Note: As a null string can be represented either by the absence of the input string or by a completely blank tape, an LBA may accept the null string.

RELATION BETWEEN LBA AND CONTEXT SENSITIVE LANGUAGE

- ✓ The set of strings accepted by nondeterministic LBA is the set of strings accepted by the context sensitive grammars, excluding null strings.
- ✓ Now we give an important result:
 - If L is a context sensitive language then L is accepted by LBA. The converse is also true.

TURING MACHINES AND TYPE 0 GRAMMARS

- ✓ In this we construct a type 0 grammar generating the set accepted by a given Turing machine M . the productions are constructed in two steps:
 - ❖ Step-1: we construct productions which transform the string $[q_1\#w\#]$ into the string $[q_2b]$, where q_1 is the initial state, q_2 is an accepting state, $\#$ is the left end marker and $\$$ is the right end marker. The grammar obtained by step -1 is called *transformational grammar*.
 - ❖ Step-2: we obtain inverse production rules by reversing the productions of the transformational grammar to get the required type 0 grammar G .

CONSTRUCTION OF GRAMMAR CORRESPONDING TO TM

- ✓ For understanding the construction, we have to note that a transition of ID corresponds to a production. We enclose the ID's within brackets.
- ✓ So acceptance of w by M corresponds to the transformation of initial ID $[q_1\#w\#]$ into $[q_2b]$. Also the length of ID may change if R/W head reaches the left end or right end, i.e. when the left hand side or right hand side bracket is reached.
- ✓ So we get productions corresponding to transition of ID's with (a) no change in length and (b) change in length.

- ✓ We now describe the construction which involves two steps:

Step-1:

(1) *No change in length of ID's:* (a) right move. $q_i a_j \rightarrow a_k q_i$ (b) left move. $a_m q_i a_j \rightarrow q_i a_m a_k$ for all $a_m \in \Gamma$

(2) *Change in length of ID's:* (a) left end. $[q_i a_j \rightarrow [q_i b a_k$

When b occurs next to the left bracket, it can be deleted. This is achieved by the production $[b \rightarrow [$

(b) Right end. When b occurs to the left of], it can be deleted. This is achieved by the production $a_j b] \rightarrow a_j]$ for all $a_j \in \Gamma$

When R/W head moves to the right of], the length increases. Corresponding to this we have a production $q_i] \rightarrow q_i b]$ for all $q_i \in Q$

(3) *Introduction of end markers:* For introducing end markers for the input string, the following productions are included:

$a_i \rightarrow [q_1 \{ a_i$ for all $a_i \in \Gamma, a_i \neq b$

$a_i \rightarrow a_i \$$] for all $a_i \in \Gamma, a_i \neq b$

For removing the brackets from $[q_2 b]$, we include the production $[q_2 b] \rightarrow S$

Recall q_1 and q_2 are initial and final states, respectively.

Step-2:

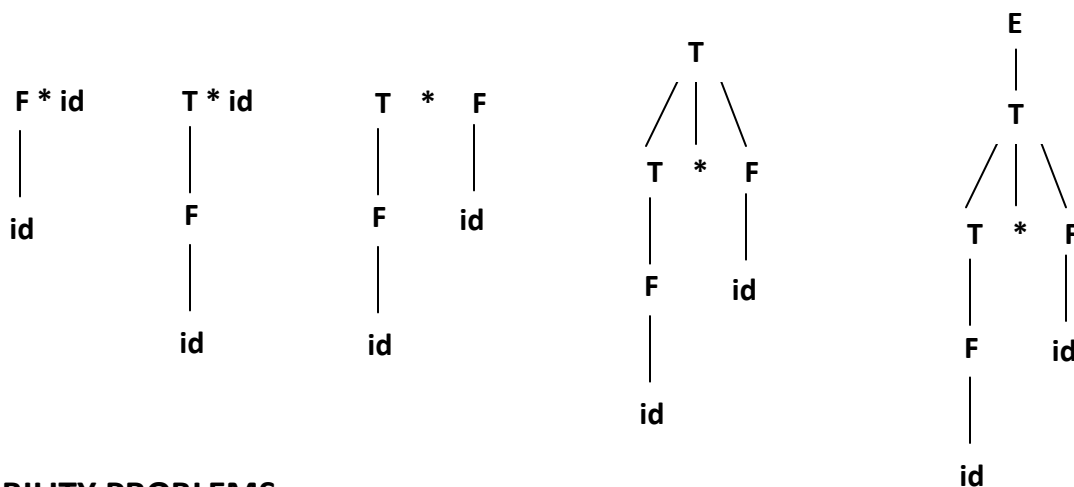
- ✓ To get the required grammar, reverse the arrows of the production obtained in step 1. The productions we get can be called as *inverse productions*. The new grammar is called *generative grammar*.
- ✓ **Example:** Ref Pg. No. 198-99, "Theory of computer science (Automata, computations and languages)" Author: K.L.P. Mishra, N. Chandrasekaran.

LINEAR BOUNDED AUTOMATA AND LANGUAGES

- ✓ A linear bounded automaton M accepts a string w if, after starting at the initial state with R/W head reading the left end marker, M halts over the right end marker in a final state. Otherwise w is rejected.

LR(0) GRAMMAR

- ✓ A grammar is said to be LR grammar, if it is sufficient that a left-to-right shift-reduce parser be able to recognize the handles of right-sentential forms when they appear on top of stack.
- ✓ LR Grammar is normally termed as Left-to-right scan of the input producing a rightmost derivation. Simply L stands for Left-to-right and R stands for Right most derivation
- ✓ To understand shift reduce parsing let us consider the construction of the string id * id for the grammar "E → E + T | T, T → T * F | F, F → (E) | id" as shown in the figure as follows:



DECIDABILITY PROBLEMS

- ✓ We begin with certain computational problems concerning finite automata; we give algorithms for testing whether a finite automaton accepts a string, whether the language of a finite automaton is empty, and whether two finite automata are equivalent.
- ✓ Note that we choose to represent various computational problems by languages. Doing so is convenient because we have already set up terminology for dealing with languages.
- ✓ For example the acceptance problem for DFA's of testing whether a particular DFA accepts a given string can be expressed as a language, A_{DFA} . This language contains the encodings of all DFA's together with strings that the DFA's accept.
Let $A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA that accepts input string } w \}$
- ✓ The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether $\langle B, w \rangle$ is a member of the language A_{DFA} .
- ✓ Similarly we can formulate other computational problems in terms of testing membership in a language. Showing that the language is decidable is the same as showing that the computational problem is decidable.
- ✓ In the following theorem we show that A_{DFA} is decidable. Hence this theorem shows that the problem of testing whether a given finite automaton accepts a given string is decidable.

DECIDABLE PROBLEMS CONCERNING REGULAR LANGUAGES:

Theorem 6.1: A_{DFA} is a decidable language.

Proof Idea: we simply need to present a TM M that decides A_{DFA} .

- ✓ $M =$ "On input $\langle B, w \rangle$, where B is a DFA and w is a string:
 - ❖ Simulate B on input w .
 - ❖ If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

Proof:

- ✓ *First let's examine the input $\langle B, w \rangle$.* It is a representation of a DFA B together with a string w .

- ✓ One reasonable representation of B is simply a list of its five components, i.e. Q, Σ, δ, q_0 and F . when M receives its input, M first determines whether it properly represents a DFA B and a string w . If not M rejects.
- ✓ Then M carries out the simulation directly. It keeps track of B 's current state and B 's current position in the input w by writing this information down on its tape.
- ✓ Initially B 's current state is q_0 and B 's current input position is the left most symbol of w .
- ✓ The states and position are updated according to the specified transition function δ . When M finishes processing the last symbol of w , M accepts the input if B is in an accepting state; M rejects the input if B is in a nonaccepting state.
- ✓ We can prove a similar theorem for nondeterministic finite automata. Let $A_{NFA} = \{ \langle B, w \rangle \mid w \text{ is an NFA that accepts input string } w \}$.

Theorem 6.2: A_{NFA} is a decidable language.

Proof:

- ✓ We present a TM N that decides A_{NFA} . We could design N to operate like M , simulating an NFA instead of a DFA. Instead, we will do it differently to illustrate a new idea: have N use M as a subroutine.
- ✓ Because M is designed to work with DFA's, N first converts the NFA it receives as input to a DFA before passing it to M .
- ✓ $N =$ "On input $\langle B, w \rangle$ where B is an NFA, and w is a string:
 - ❖ Convert NFA B to an equivalent DFA C .
 - ❖ Run TM M from theorem 6.1 on input $\langle C, w \rangle$.
 - ❖ If M accepts, *accept*; otherwise, *reject*."
- ✓ Running TM M in **stage 2** means incorporating M into the design of N as a sub procedure.
- ✓ Similarly we can determine whether a regular expression generates a given string. Let $A_{REG} = \{ \langle R, w \rangle \mid R \text{ is a regular expression that generates string } w \}$.

Theorem 6.3:

A_{REG} is a decidable language.

Proof:

- ✓ The following TM P decides A_{REG} .
- ✓ $P =$ "On input $\langle R, w \rangle$ where R is a regular expression and w is a string:
 - ❖ Convert regular expression R to an equivalent NFA A .
 - ❖ Run TM N on input $\langle A, w \rangle$.
 - ❖ If N accepts, *accept*; if N rejects, *reject*."

DECIDABILITY PROBLEMS CONCERNING CONTEXT-FREE LANGUAGES

Theorem: A_{CFG} is a decidable language.

Proof:

- ✓ The TM S for A_{CFG} follows.

$S =$ “On input $\langle G, w \rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
 2. List all derivations with $2n-1$ steps, where n is the length of w , except if $n = 0$, then instead list all derivations with 1 step.
 3. If any of these derivations generate w , *accept*; if not *reject*”
- ✓ The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.
 - ✓ The algorithm in TM S is very inefficient and would never be used in practice.

UNIVERSAL TURING MACHINE

Theorem: A_{TM} is undecidable.

Proof:

- ✓ Before we get to the proof, let’s first observe that A_{TM} is Turing-recognizable. Thus this theorem shows that recognizers are more powerful than deciders.
- ✓ Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize. The following Turing machine U recognizes A_{TM} .
- ✓ $U =$ “On input $\langle M, w \rangle$, where M is a TM and w is a string:
 - ❖ Simulate M on input w .
 - ❖ If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*.”
- ✓ Note that a machine loops on input $\langle M, w \rangle$ if M loops on w , which is why this machine does not decide A_{TM} .
- ✓ If the algorithm had some way to determine that M was not halting on w , it could *reject*. Hence A_{TM} is sometimes called the **halting problem**. As we demonstrate, an algorithm has no way to make this determination.
- ✓ The Turing machine U is interesting in its own right. It is an example of the Universal Turing machine first proposed by Turing.
- ✓ This machine is called universal because it is capable of simulating any other Turing machine from the description of that machine.
- ✓ The universal Turing machine played an important early role in stimulating the development of stored-program computers.
- ✓ Designing a general purpose Turing machine usually called the Universal Turing machine is a more complex task. We must design a machine that accepts two inputs i.e. (1) the input data and (2) a description of computation (algorithm).

A SIMPLE UNDECIDABLE PROBLEM

POST CORRESPONDENCE PROBLEM

- ✓ In this section we show that the phenomenon of undecidability is not confined to problems concerning automata. We give an example of an undecidable problem concerning simple manipulations of strings. It is called the **Post Correspondence Problem** or **PCP**.
- ✓ We can now describe this problem easily as a type of puzzle. We begin with a collection of dominos, each containing two strings, one on each side. An individual domino looks like

$$\begin{bmatrix} a \\ ab \end{bmatrix}$$

- ✓ And a collection of dominos looks like

$$\left\{ \begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix} \right\}$$

- ✓ The task is to make a list of these dominos (repetitions permitted) so that the string we get by reading off the symbols on the top is the same as the string of symbols on the bottom. This list is called a **match**.
- ✓ For example the following list is the match for this puzzle.

$$\begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix}$$

- ✓ Reading of the top string we get **abcaabc**, which is the same as reading off the bottom. For some collections of dominos finding a match may not be possible. For example the collection below cannot contain a match because every top string is longer than the corresponding bottom string.

$$\left\{ \begin{bmatrix} abc \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} acc \\ ba \end{bmatrix} \right\}$$

- ✓ The post correspondence problem is to determine whether a collection of dominos has a match. This problem is unsolvable by algorithms.

Theorem:

PCP is undecidable.

Proof:

- ✓ Refer: Pg. No. 204-209 in "Introduction to the Theory of Computation" - second edition – Author: Michael Sipser.

TURING REDUCIBILITY

- ✓ A reduction is a way of converting one problem to another problem such a way that a solution to the second problem can be used to solve the first problem.
- ✓ For example suppose that we want to find a way around a new city. We know that doing so would be easy if we had a map. Thus we can reduce the problem of finding the way around the city to the problem of obtaining a map of the city.

- ✓ Reducibility always involves two problems, which we call A and B. If A reduces to B, we can use a solution to B to solve A. so in our example, A is the problem of finding the way around the city and B is the problem of obtaining a map.
- ✓ Note that reducibility says nothing about solving A or B alone, but only about the solvability of A in the presence of a solution to B.
- ✓ Reducibility plays an important role in classifying problems by decidability and later in complexity theory as well.
- ✓ When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives the solution to A.
- ✓ In terms of computability theory if A is reducible to B and B is decidable, then A is also decidable. Equivalently, if A is undecidable and reducible to B, then B is also undecidable.

Definition:

- ✓ An oracle for a language B is an external device that is capable of reporting whether any string w is a member of B. An oracle Turing machine is a modified Turing machine that has the additional capability of querying an oracle.
- ✓ We write M^B to describe an oracle Turing machine that has an oracle for language B.

Example:

- ✓ Consider an oracle for A_{TM} . An oracle Turing machine with an oracle for A_{TM} can decide more languages than an ordinary Turing machine can. Such a machine can (obviously) decide A_{TM} itself, by querying the oracle about the input.
- ✓ It can also decide E_{TM} , the emptiness testing problem for TM's with the following procedure: called T_{TM}^A .
- ✓ $T_{TM}^A =$ " On Input $\langle M \rangle$, where M is a TM:
 - ❖ Construct the following TM N.
 - N = "On any input:
 - ◆ Run M in parallel on all strings in Σ^* .
 - ◆ If M accepts any of these strings, accept."
 - ❖ Query the oracle to determine whether $\langle N, 0 \rangle \in A_{TM}$.
 - ❖ If the oracle answers No, accept; if YES, reject."
- ✓ If M's language isn't empty, N will accept every input and in particular input 0. Hence the oracle will answer YES, and T_{TM}^A will reject.
- ✓ Conversely if M's language is empty, T_{TM}^A will accept.
- ✓ Thus T_{TM}^A decides E_{TM} . So we say that E_{TM} is decidable relative to A_{TM} . That brings us the definition of Turing reducibility.

Definition

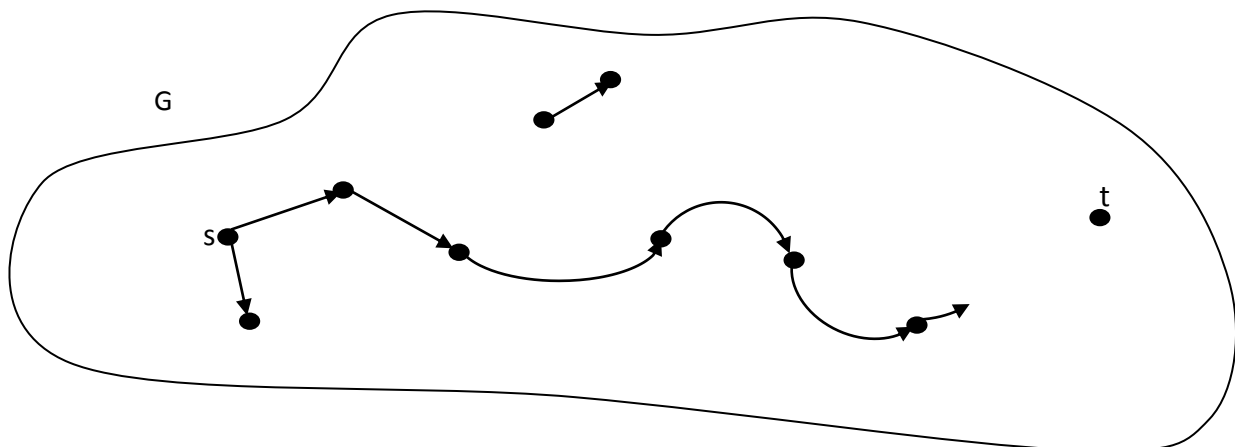
- ✓ Language A is Turing reducible to language B, written $A \leq_T B$, if A is decidable relative to B.

DEFINITION OF P AND NP PROBLEMS**Definition of P**

- ✓ P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words, P is

$$\bigcup_k \text{TIME}(n^k)$$

- ✓ The class P plays a central role in our theory and is important because
 1. P is invariant for all models of computation that are Polynomially equivalent to the deterministic single-tape Turing machine, and
 2. P roughly corresponds to the class of problems that are realistically solvable on a computer.
- ✓ Item 1 indicates that P is mathematically robust in class. It isn't affected by the particulars of the model of computation that we are using.
- ✓ Item 2 indicates that P is relevant from a practical standpoint. When a problem is in P, we have a method of solving it that runs in time n^k and on the constant k.
- ✓ Whether this running time is practical depends on k and on the application. Of course a running time of n^{100} is unlikely to be of any practical use. Nevertheless, calling polynomial time the threshold of practical solvability has proven to be useful.
- ✓ For example consider a directed graph G contains nodes s and t as shown in the figure below:



- ✓ The PATH problem is to determine whether a directed path exists from s to t. let $\text{PATH} \in \text{P}$

Proof: A polynomial time algorithm M for PATH operates as follows.

M = "On input $\langle G, s, t \rangle$ where G is directed graph with nodes s and t.

1. Place a mark on nodes s.
2. Repeat the following until no additional nodes are marked:

3. Scan all the edges of G . If an edge (a, b) is found going from a marked node a to an unmarked node b , mark node b .
4. If t is marked, *accept*. Otherwise, *reject*."

Definition of NP:

- ✓ A verifier for a language A is an algorithm V , where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$
- ✓ We measure the time of a verifier only in terms of the length of w , so a polynomial time verifier runs in polynomial time in the length of w . Language A is polynomially verifiable if it has a polynomial time verifier.
- ✓ NP is the class of languages that have polynomial time verifiers.

P versus NP:

- ✓ P versus NP problem is a major unsolved problem in computer science. Informally, it asks whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer.
- ✓ The informal term quickly used above means the existence of an algorithm for the task that runs in polynomial time. The general class of questions for which some algorithm can provide an answer in polynomial time is called "class P" or just "P".
- ✓ For some questions, there is no known way to find an answer quickly, but if one is provided with information showing what the answer is, it may be possible to verify the answer quickly. The class of questions for which an answer can be verified in polynomial time is called NP.

P = the class of languages for which membership can be *decided* quickly.

NP = the class of languages for which membership can be *verified* quickly.

NP-COMPLETENESS

- ✓ One important advance on the P versus NP question came in the early 1970's with the work of Stephen Cook and Leonid Levin. They discovered certain problems in NP whose individual complexity is related to that of the entire class.
- ✓ If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**.
- ✓ The phenomenon of NP-completeness is important for both theoretical and practical reasons.
- ✓ On the theoretical side, a researcher trying to show that P is unequal to NP may focus on an NP-complete problem. Furthermore a researcher attempting to prove that P equals NP only needs to find a polynomial time algorithm for an NP-complete problem to achieve this goal.
- ✓ On the practical side, the phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm to solve a particular problem.

- ✓ The first NP complete problem that we present is called the satisfiability problem. For example consider a Boolean formula i.e. an expression involving Boolean variables and operations, as given below:

$$\Phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$$

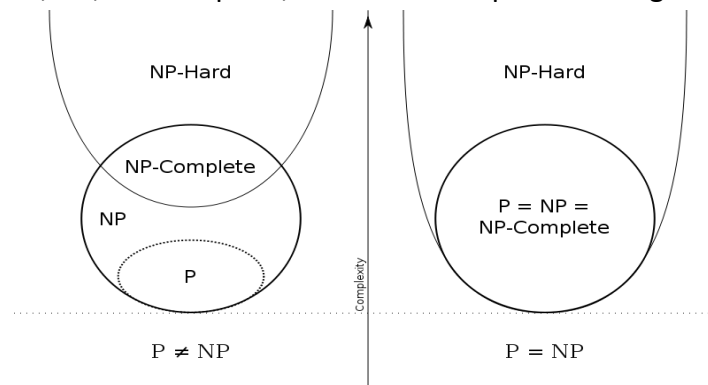
- ✓ A Boolean formula is satisfiable if some assignment of 0's and 1's to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $x = 0$ and $y = 1$ and $z = 0$ makes Φ evaluate to 1. We say the assignment satisfies Φ .

NP – HARD

- ✓ NP-hard is called as non-deterministic polynomial-time hard.
- ✓ In computational complexity theory, it is a class of problems that are, informally, "at least as hard as the hardest problems in NP".
- ✓ A problem H is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H (i.e., $L \leq_T H$). In other words, L can be solved in polynomial time by an oracle machine with an oracle for H.
- ✓ Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving H, and solves L in polynomial time, if the subroutine call takes only one step to compute. NP-hard problems may be of any type: decision problems, search problems, or optimization problems.
- ✓ As consequences of definition, we have (note that these are claims, not definitions):
 1. Problem H is at least as hard as L, because H can be used to solve L;
 2. Since L is NP-complete, and hence the hardest in class NP, also problem H is at least as hard as NP, but H does not have to be in NP and hence does not have to be a decision problem (even if it is a decision problem, it need not be in NP);
 3. Since NP-complete problems transform to each other by polynomial-time many-one reduction (also called polynomial transformation), all NP-complete problems can be solved in polynomial time by a reduction to H, thus all problems in NP reduce to H; note, however, that this involves combining two different transformations: from NP-complete decision problems to NP-complete problem L by polynomial transformation, and from L to H by polynomial Turing reduction;
 4. If there is a polynomial algorithm for any NP-hard problem, then there are polynomial algorithms for all problems in NP, and hence $P = NP$;
 5. If $P \neq NP$, then NP-hard problems have no solutions in polynomial time, while $P = NP$ does not resolve whether the NP-hard problems can be solved in polynomial time;
 6. If an optimization problem H has an NP-complete decision version L, then H is NP-hard.
- ✓ A common mistake is to think that the NP in NP-hard stands for non-polynomial. Although it is widely suspected that there are no polynomial-time algorithms for NP-hard problems, this has

never been proven. Moreover, the class NP also contains all problems which can be solved in polynomial time.

- ✓ The Euler diagram for P, NP, NP-Complete, NP-hard set of problems is given as follows:



NP-NAMING CONVENTION

The NP-family naming system is confusing: NP-hard problems are not all NP, despite having NP as the prefix of their class name. However, the names are now entrenched and unlikely to change. On the other hand, the NP-naming system has some deeper sense, because the NP family is defined in relation to the class NP:

NP-Hard

- ✓ At least as hard as the hardest problems in NP. Such problems need not be in NP; indeed, they may not even be decision problems.

NP-complete

- ✓ These are the hardest problems in NP. Such a problem is in NP.

APPLICATION AREAS

NP-hard problems are often tackled with rules-based languages in areas such as:

1. Configuration
2. Data mining
3. Selection
4. Diagnosis
5. Process monitoring and control
6. Scheduling
7. Planning
8. Rosters or schedules
9. Tutoring systems
10. Decision support