2020-21 Onwards(MR-20)	**MALLA REDDY ENGINEERING COLLEGE (Autonomous)**	**B.Tech. V Semester**		
Code: A0523	**FORMAL LANGUAGES AND AUTOMATATHEORY**	**L**	**T**	**P**
Credits: 4		**3**	**1**	**-**

Prerequisites: NIL

Course Objectives:

This course enables the students to define basic properties of formal languages, explain the Regular languages and grammars, inter conversion, Normalizing CFG, describe the context free grammars, minimization of CNF, GNF and PDA, designing Turing Machines and types of Turing Machines, church's hypothesis counter machines, LBA, P & NP problems and LR grammar.

MODULE I: **Introduction** [13 Periods]

Basics of Formal Languages - Alphabet, Strings, Language, Operations, Chomsky hierarchy of languages, Finite state machine Definitions, finite automation model, acceptance languages.

Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions.

Deterministic Finite Automata: Definition of DFA, How A DFA Process Strings, The language of DFA, Conversion of NFA with €-transitions to NFA without €-transitions. Conversion of NFA to DFA, Moore and Melay machines

MODULE II: **Regular Languages** [13 Periods]

Representation of Regular Expressions - Regular Sets, Regular Expressions, identity Rules, Constructing Finite automata for the given regular expressions, Conversion of Finite automata to regular expressions.

Pumping Lemma - **Pumping lemma of regular sets, closure properties of regular sets (proofs not required).** Regular Grammars – right linear and left linear grammars, equivalence between regular grammar and FA.

MODULE III: **CFG and PDA** [14 Periods]

A: **Context Free Grammar** - Derivation trees, sentential forms, right most and left most derivations of strings. Ambiguity in Context frees Grammars. Minimization of Context free grammars, CNF, GNF, Pumping Lemma for Context Free Languages.

B: **Push Down Automata** - Definition, model, acceptance of CFL, Acceptance by final state, acceptance by empty state and its equivalence, Equivalence of CFL and PDA (proofs not required), Introduction to DCFL and DPDA.

MODULE IV: **Computable Functions** [12 Periods]

Turing Machine - Definition, model, Design of TM, computable functions.

Recursive Enumerable Languages and Theorems - Recursively enumerable languages, Church's hypothesis, counter machine, types of Turing Machines (proofs not required)

MODULE V: **Computability Theory** [12 Periods]

Linear Bounded Automata - Linear Bounded Automata and context sensitive languages, LR (0) grammar, decidability of problems, Universal TM.

P and NP Problems - Undecidable problems about Turing Machine – Post's Correspondence Problem, The classes P and NP.

TEXT BOOKS:

1. H.E. Hopcroft, R. Motwani and J.D Ullman, "Introduction to Automata Theory, Languages and Computations", Second Edition, Pearson Education, 2007.
2. KVN SUNITHA N Kalyani, "Formal languages and Automata Theory", Pearson Education

REFERENCES:

1. H.R.Lewis and C.H.Papadimitriou, "Elements of The theory of Computation", Second Edition, Pearson Education/PHI, 2003
2. J.Martin, "Introduction to Languages and the Theory of Computation", Third Edition, TMH, 2003.
3. Micheal Sipser, "Introduction of the Theory and Computation", Thomson Brokecole, 1997.

E-RESOURCES:

1. https://www.iitg.ernet.in/dgoswami/Flat-Notes.pdf
2. https://www.pdfdrive.com/introduction-to-automata-theory-formal-language-and-computability-theory-e37220113.html
3. https://freevideolectures.com/course/3379/formal-languages-and-automata-theory
4. http://nptel.ac.in/courses/111103016/

Course Outcomes:

At the end of the course, students will be able to

1. **Define** the theory of automata types of automata and FA with outputs.
2. **Differentiate** regular languages and applying pumping lemma.
3. **Classify** grammars checking ambiguity able to apply pumping lemma for CFLvarious types of PDA.
4. **Illustrate** Turing machine concept and in turn the technique applied in computers.
5. **Analyze** P vs NP- Class problems and NP-Hard vs NP-complete problems, LBA, LR Grammar, Counter machines, Decidability of Problems.

CO- PO, PSO Mapping
(3/2/1 indicates strength of correlation) 3-Strong, 2-Medium, 1-Weak

COs	**Programme Outcomes (POs)**												**PSOs**		
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**
CO1	3	2	2									2	2	2	
CO2		2	2	2	2							2	2	2	
CO3		2	2	2	2							2	2	2	
CO4		2	2	2	2							2	2	2	
CO5		2	2	2	2							2	2	2	

MODULE V: Computability Theory

Linear Bounded Automata - Linear Bounded Automata and context sensitive languages, LR (0) grammar, decidability of problems, Universal TM.

P and NP Problems - Undecidable problems about Turing Machine – Post's Correspondence Problem, The classes P and NP.

Module – V Subjective Question Bank with Notes

S No	Question	Marks	BT Level	CO
1	i) Explain in detail about Linear bounded automata model. ii) Identify the equivalence of LBA's and CSG's	2 3	L2 L3	5
	OR			
2	Explain about the decidability and undecidability problems with examples	5	L2	5
3	i. Define and explain Post's Correspondence Problem in detail ii. Obtain the solution for the following post's correspondence problem {{100, 1}, {0, 100}, {1, 00}}	3 2	L2	5
	OR			
4	Outline about Universal Turing Machine? Explain what are the actions take place in TM?	5	L2	5
5	i) How do you say the given grammar is LR(0)? ii) State whether the following grammar is an LR(0) grammar. $A \to aAa \mid B$, $B \to b$	3 2	L2 L3	5
	OR			
6	i) Describe recursive languages and recursively enumerable languages ii) Explain the halting problem of TMs in detail.	2 3	L2 L2	5
7	Design a TM for accepting strings of the language $L = \{a^n b^n c^n \mid n \geq 0\}$.	5	L4	5
	OR			
8	State and Explain P and NP class problems with example. Differentiate between P class problems with NP Class problems.	3 2	L4	5

Linear bounded automata

A non-deterministic TM is called linear bound automata (LBA) if

- Its input alphabet **includes two special symbols ∅ and $ as left and right end markers.**
- It has no moves beyond these end markers, i.e, no left move from ∅ and no right move from $.
- It never changes the symbols ∅ and $.

A linear bound automaton is defined using 8-tuple form by M = (Q, Σ, Γ, δ, q_0, ∅, $, F), where Q, Σ, Γ, δ, q_0, F are same as for non-deterministic TM, and ∅ and $ are left and right end markers.
The language accepted by M is defined as L(M) and is given by

$$\{w \mid w \in (\Sigma - \{\emptyset, \$\})^* \text{ and } q_0\emptyset w\$ \overset{*}{\underset{M}{\longmapsto}} aqb \text{ for some } q \text{ in } F\}$$

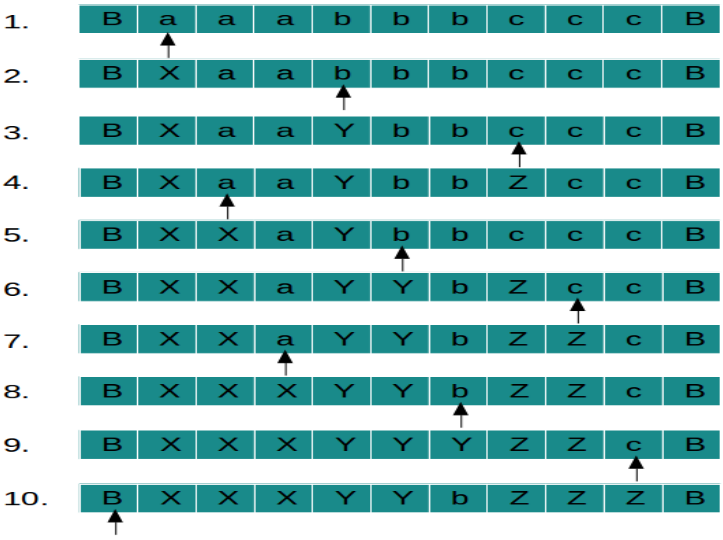
LBA is powerful than PDA for example: **$a^n b^n c^n \mid n \geq 1$** cannot be accepted by PDA whereas it can be accepted by LBA **without using any extra space or BLANK symbol**.

Approach

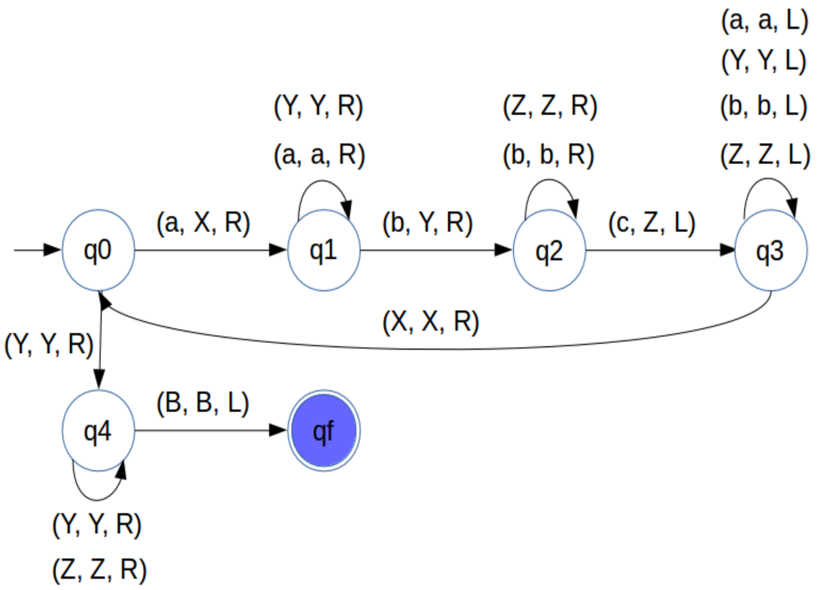
Suppose input is : "aaabbbccc“

- Mark 'a' as 'X' and move right, mark 'b' as ‘Y' and move right, mark 'c' as ‘Z' and move left.
- And repeat this process till all the symbols a,b,c are marked equally
- At last if everything is marked that means string is accepted.

TAPE movement for string "aaabbbccc":



1.	B	a	a	a	b	b	b	c	c	c	B
2.	B	X	a	a	b	b	b	c	c	c	B
3.	B	X	a	a	Y	b	b	c	c	c	B
4.	B	X	a	a	Y	b	b	Z	c	c	B
5.	B	X	X	a	Y	b	b	c	c	c	B
6.	B	X	X	a	Y	Y	b	Z	c	c	B
7.	B	X	X	a	Y	Y	b	Z	Z	c	B
8.	B	X	X	X	Y	Y	b	Z	Z	c	B
9.	B	X	X	X	Y	Y	Y	Z	Z	c	B
10.	B	X	X	X	Y	Y	b	Z	Z	Z	B



Standard Examples of LBA

1. Following are standard example of LBA to remember
   L = {$a^n b^n c^n$ | n ≥1}
2. L = {$a^{n!}$ | n ≥ 0}
3. L = {a^n | n = m^2, m ≥ 1}, means n is perfect square
4. L = {a^n | n is prime}
5. L = {a^n | n is not a prime}
6. L = {ww | w ε {a, b}$^+$}
7. L = {w^n | w ε {a, b}$^+$, n ≥ 1}
8. L = {$ww w^R$ | w ε {a, b}$^+$}

context-sensitive grammars:

- These grammars define the **context-sensitive languages**.
- In context-sensitive grammar (CSG), all the productions of the form **$\alpha \rightarrow \beta$** where **length of α is less than or equal to length of β** i.e.

  $|\alpha| \leq |\beta|$, α and β may have any number of terminals and non-terminals.
- These grammars can have rules of the form **$\alpha A \beta \rightarrow \alpha \gamma \beta$** with A as non-terminal and α, β and γ are strings of terminals and non-terminals.
- We **can replace A by γ** where **A lies between α and β**.
- Hence the name CSG.
- The **strings α and β may be empty**, but **γ must be nonempty**.
- It **cannot include the rule $S \rightarrow \varepsilon$.**
- These languages are exactly all languages that **can be recognized by a linear bound automata.**

Example:

- **aAbcD → abcDbcD**

- **Context-sensitive grammars are more powerful than context-free grammars** because there are some languages that can be described by CSG but not by context-free grammars and **CSL are less powerful than Unrestricted grammar.**
- That's why context-sensitive grammars are positioned between context-free and unrestricted grammars in the Chomsky hierarchy.

- Context-sensitive grammar has 4-tuples. **G = {N, T, P, S},** Where
  N = Set of non-terminal symbols
  T = Set of terminal symbols
  S = Start symbol of the production
  P = Finite set of productions
  All rules in P are of the form **$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$**

We represent the non-terminal symbols with capital letters and terminal symbols with lower-case letters.

- **Context-sensitive Language:** **The language that can be defined by context-sensitive grammar is called CSL**.

Properties of CSL are :

- **Union, intersection and concatenation of two context-sensitive languages is context-sensitive.**
- **Complement** of a **context-sensitive language is context-sensitive**.

Example –

- Consider the following CSG.
  S → abc/aAbc
  Ab → bA
  Ac → Bbcc
  bB → Bb
  aB → aa/aaA

What is the language generated by this grammar?

- **Solution**:
  S → aAbc
  → a**bA**c
  → ab**Bbcc**

→ a**B**bbcc
→ **aaA**bbcc
→ aa**bA**bcc
→ aab**bA**cc
→ aabb**Bbcc**c
→ aab**Bb**bccc
→ aa**Bb**bbccc
→ a**aa**bbbccc

The language generated by this grammar is $\{\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n\geq 1\}$.

Equivalence of LBA's and CSG's

- We can show that if L is a context sensitive language (CSL), then there exists a linear bound automaton M such **that L(M) = L – ε** .

Ex: If L is a CSL, then L is accepted by some LBA.

Proof: Let us construct a linear bound automaton M with two-track tape, to recognize L.

- The **first track holds** the input **string w as Øw$.**
- The **second track** is **used while the input is processed**.

- **LBA initializes the second track with S just below the leftmost symbol of w.**
- **If w = ε**, then the **system halts without accepting**.
- **Otherwise, it repeatedly guesses a production and a position in the sentential form which is on the second track**.
- **If the sentential form expands**, then **it shifts the portion of string from the current position to right**.
- **If the new sentential form is longer than w**, **then the system halts without accepting.**

Since the right side of all the productions are at least as long as left side, there would not be any derivation as $S \overset{*}{\Rightarrow} \alpha \overset{*}{\Rightarrow} w$, where α is longer than w. Hence, the LBA accepts a string if and only if $S \overset{*}{\Rightarrow} w$, where **w is a word generated by CSG.**

LR (0) grammar

A grammar is said to be LR(0) grammar if

- Its start symbol does not appear **on the right-hand side of any other production.**
- If the closure set of a item has a production of the form **A → α•,** then there is no production of the form B → β• or B → β•γ

Items

- The **construction of the parsing tables** is **based on the notion of LR(0) *items*** (**simply called *items* here**) which are grammar rules with a special dot added somewhere in the right-hand side.
- For example, the rule E → E + B has the following four corresponding items:
  - E → •E + B
  - E → E• + B
  - E → E + •B
  - E → E + B•
- Rules of the form A → ε have only a single item A → •.
- These **rules will be used to denote the state of the parser**.
- The item E → E• + B, for example, indicates that the **parser has recognized a string corresponding with E on the input stream** and **now expects to read a + followed by another string corresponding with B.**

Item sets

- It is usually **not possible to characterize the state of the parser with a single item** because **it may not know in advance which rule it is going to use for reduction.**
- For example, if there is also a rule E → E * B,

then the items E → E • + B and

E → E• * B will **both apply after a string corresponding with E has been read.**

- Therefore, we will characterize the state of the parser by a set of items, in this case, the set

{

E → E• + B,

E → E• * B

}.

Closure of item sets

- An item with **a dot in front of a non-terminal**, such as E → E + •B, **indicates that the parser expects to parse the non-terminal B next**.
- **To ensure the item set contains all possible rules the parser may need** in the middle of parsing, **it must include all items describing how B itself will be parsed.**
- This means that **if there are rules such as B → 1 and B → 0, then the item set must also include the items B → •1 and B → •0.**

In general, this can be formulated as follows:

- If there is an item of the form **A → v • Bw** in an item set and in the grammar there is a rule of the form **B → w′, then the item B → • w′ should also be in the item set.**
- Any set of items can be extended such that it satisfies this rule.
- This is because **we can simply continue to add the appropriate items until all non-terminals preceded by dots are accounted for**.
- The minimal extension is called the ***closure of an item set*** *and written as* **closure(I) where I is an item set.**
- **It is these closed item sets that we will take as the states of the parser,** although **only the ones that are actually reachable from the begin state will be included in the tables.**

Augmented grammar

- Before we start determining the transitions between the different states, the grammar is always augmented with an extra rule:

(0) S → E

- where **S is a new start symbol** and **E is the old start symbol**.
- The parser will use this rule for reduction exactly when it has accepted the input string.

Is the grammar S → C | D, C → aC | b, D → aD | C is LR(0)?

Consider the string aaaab belonging to the language. To derive the string,

a a a b	No look ahead is needed to select C.
a a a C	No look ahead is needed to select C
a a C	No look ahead is needed to select C
a C	No look ahead is needed to select C
S	No look ahead is needed to select S

(Or)

a a a b	No look ahead is needed to select C.
a a a C	No look ahead is needed to select C
a a a D	No look ahead is needed to select D
a a D	No look ahead is needed to select D
a D	No look ahead is needed to select D
D	No look ahead is needed to select D
S	No look ahead is needed to select S

Since we do not need any look ahead of symbol for proper substitution, it is LR(0) grammar.

Halting problem of TMs.

Halting problem of the TM is formulated as follows: Given an arbitrary TM and an arbitrary input for the machine, will the given machine halt on the given input?

In other words, Turing wondered whether it would be possible to write a TM program that would take two inputs <program P, input i> and would answer 'Yes' if the TM halts when executing program P on input i, and would answer 'No' if it wouldn't halt, that is, if it would loop for ever.

Solution can be found by giving a description number to every possible TM program, so that no possible programs is left out. One way to figure out if there is a program that can solve the halting problem would be to look through all the whole numbers, interpreting each as the description number of a TM program, and checking to see if it is the program that solves the halting problem.

Of course, this is completely impractical. But Turing realized that if we could prove that no whole number was the right one, then we would know that no program to solve the halting problem exists.

Decision Problems

- Decision problems are closely related to function problems, which can have answers that are more complex than a simple 'yes' or 'no'.
- A corresponding function problem is 'given two numbers x and y, how to program for x divided by y?'
- This is related to optimization problems, which is concerned with finding the best answer to a particular problem.
- A method for solving a decision problem given in the form of an algorithm is called a decision procedure for that problem.
- A decision procedure for the decision problem 'given two numbers x and y, does x evenly divide y?' would give the steps for determining whether x evenly divides y.
- One such algorithm is by long division, taught to many school children.
- If the remainder is zero, the answer produced is 'yes'; otherwise, it is 'no'.
- A decision problem that can be solved by an algorithm, such as the example of divisibility discussed above, is called decidable.
- The field of Computational Complexity categorizes the decidable decision problems depending on difficulty with which they are solved.
- 'Difficulty', in this sense, is described in terms of the computational resources needed by the most efficient algorithm for a certain problem.
- The field of Recursion Theory, categorizes undecidable decision problems by Turing degree, which is a measure of the non-computability inherent in any solution.
- A problem whose language is recursive is said to be decidable.
- Otherwise, the problem is said to be undecidable.
- Decidable problems have an algorithm that takes as input an instance of the problem and determines whether the answer to that instance is 'yes' or 'no'.

- An example of an undecidable problem is the Halting problem of the TM.

Undecidable Problems

- For some problems, we can prove that there is no algorithm that always solves them, no matter how much time or space is allowed.
- One very uninformative proof of this is based on the fact that there are as many problems as there real numbers, and only as many programs as there are integers.
- So, there are not enough programs to solve all the problems.
- But we can also define explicit and useful problems which cannot be solved.

Undecidable Problems

- M = {<M, w>| M is a TM, w is a string, M accepts w} Assume a TM is decidable, which halts and says accepted or rejected.
- Let H be a machine for a TM <M, w>. Then, H halts and accepts, if M accepts w; or H rejects, if M fails to accept w.

To put more formally,

- H(<H, w>) = {accept if M accept w.

  = {reject if M does not accept w.

- Construct a new truing machine D with H as subroutine.
- D calls H to find what M does when input to M is its own description <M>. that is, D is running a machine as its own description.
- It is just like a compiler written and compiled in the same language.
- D gets information and complements the action.

D is defined as <M> where M is a TM.

1. Runs H on input <M, <M>>.
2. If H accepts, it rejects, if H rejects, it accepts.

In summary,

$$D(<M>) = \begin{cases} \text{accept if M does not accept} <M> \\ \text{reject if M accepts} <M> \end{cases}$$

- When we run D with its own description <D> as input? In that case, we get

$$D(<D>) = \begin{cases} \text{accept if D does not accept} <D> \\ \text{reject if D accepts} <D> \end{cases}$$

- It is forced to do opposite to what D does. Thus neither TM D nor TM H exists.

When do we say a problem is decidable? Give an example of an undecidable problem?

A problem whose language is recursive is said to be decidable. Otherwise, the problem is said to be undecidable. Decidable problems have an algorithm that takes as input an instance of the problem and determines whether the answer to that instance is 'yes' or 'no'. An example of an undecidable problem is the Halting problem of the TM.

Language of context free grammar G is a decidable

- We need to find whether the given string can be generated by the grammar G.

- One approach that can be used is to generate all the strings and check whether the given string w is generated.
- But this approach does not work, sometimes the turing machine may not halt.
- It may enter into an infinite loop trying to recognize rather than decide and report whether it is valid or not.

- Second procedure is to convert the given grammar to CNF, so that for a string w of length n, there would be at most 2^{n-1} steps to generate the string.
- Hence, it ensures that the turing machine would halt after generating strings using at most 2^{n-1} steps.
- It can also decide and say whether the string is valid or not.

TM for this can be defined as S = <G, w> where G is the given grammar and w is the string given:

➢ Convert the G to CNF.

➢ List all derivations of the grammar starting from 1 to a maximum of 2^{n-1} steps, where n is the length of w.

➢ If any of these derivations correspond to w, then accept; otherwise, reject.

Universal Turing Machine?

A universal TM M_u is an automaton that, given as input the description of any TM M and a string w, can simulate the computation of M for input w. To construct such a Mu, we first choose a standard way of describing TMs. We may, without loss of generality, assume that M = (Q, {0, 1}, {0, 1, B}, d, q1, B, q2) where Q = {q_1, q_2, … q_n}, q_1 the initial state, and q_2 the single final state. The alphabet {0, 1, B} ∈ Γ are represented as a_1, a_2 and a_3. The directions left and right are represented as D_1 and D_2, respectively. The transitions of TM are encoded in a special binary representation where each symbol is separated by 1. For example, if there is a transition

$$d\ (q_i, a_j) = (q_k, a_l, D_m)$$

the binary representation for the transition is given as

$$0^i10^j10^k10^l10^m$$

The binary code for the Turing machine M that has transitions t1, t2, t3…tn is represented as

$$111\ \boldsymbol{t_1}\ 11\ \boldsymbol{t_2}\ 11\ \boldsymbol{t_3}\ 11\ \ldots\ 11\ \boldsymbol{t_n}\ 111$$

Note: The transitions need not be in any particular order.

If a string has to be verified, then the problem is represented as a tuple <M, w> where M is the definition of TM, and w is the input string.

Ex: Let M = ({q_1, q_2, q_3}, {0, 1}, {0, 1, B}, d, q1, B, {q_2}) have moves defined as

$$d\ (q_1, 1) = (q_3, 0, R)$$
$$d\ (q_3, 0) = (q_1, 1, R)$$
$$d\ (q_3, 1) = (q_2, 0, R)$$
$$d\ (q_3, B) = (q_3, 1, L)$$

Give the problem representation for the string w = 1011.

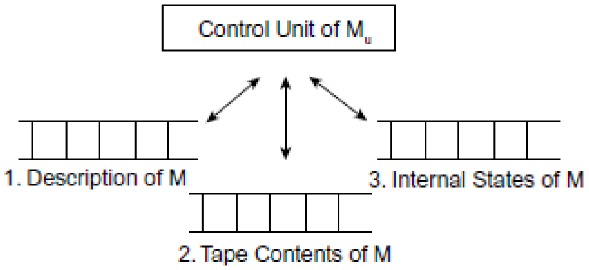
Solution: Let binary representation for states { q_1, q_2, q_3} be {0, 00, 000}, for alphabet {0, 1, B} be {0, 00, 000} and for directions {R, L} be {0, 00}. The transitions are represented as follows:

Transition	Binary representation
δ (q_1, 1) = (q_3, 0, R)	010010001010
δ (q_3, 0) = (q_1, 1, R)	000101010010
δ (q_3, 1) = (q_2, 0, R)	0001001001010
δ (q_3, B) = (q_3, 1, L)	00010001000100100

The problem instance <M, 1011> is represented as

111 010010001010 **11** 000101010010 **11** 0001001001010 **11** 00010001000100100 **111** 1011

The following figure shows the organization of a universal TM that has a control unit and three tapes:



For any input M and w, Tape 1 will keep an encoded definition of M, Tape 2 will contain the tape contents of M and Tape 3, the internal state of M. Mu looks first at the contents of Tapes 2 and 3 to determine the configuration of M. The behaviour of the M is as follows.

1. Check the format of Tape 1 for the validations of the TM model.

   a. No two transitions should begin with 0^i10^j1 for the same i and j.

   b. Check that if $0^i10^j10^k10^l10^m$ represents a transition, then $1 \leq j \leq 3$, $1 \leq l \leq 3$, and $1 \leq m \leq 3$.

2. Initialize Tape 2 to contain w. Initialize Tape 3 to hold a single 0 representing initial state q_1. For all the tapes, the tape heads are positioned at the left end and these symbols are marked to identify the starting position.

3. When Tape 3 holds 00, it is said to reach the final state, and the machine can halt.

4. Let, at any given time, a_j be the symbol currently scanned by tape head 2 and let 0^i, the contents of Tape 3 (which indicates state). Scan Tape 1 from the left end to the second 111 looking for a substring beginning with 110^i10^j1.

   a. if no such string is found, then halt and reject.

   b. if found, then let the suffix be $0^k10^l10^m11$. Put 0^k on Tape 3, print al on the tape cell scanned by head 2 and move the head in direction D_m.

It is clear that M_u accepts <M, w> if and only if M accepts w. It is also true that if M runs forever on w, Mu runs forever on <M, w> and if M halts on w without accepting, M_u also halts on w without accepting.

P and NP class problems

P Problems

In computational complexity theory, P, also known as PTIME or DTIME, is one of the most fundamental complexity classes. It contains all decision problems that can be solved by a deterministic TM using a polynomial amount of computation time or, simply put, polynomial time. P is known to contain many natural problems, including the decision versions of linear programming calculating the greatest common divisor and finding a maximum matching.

NP Problems

In computational complexity theory, NP is one of the most fundamental complexity classes. Intuitively, NP is the set of all decision problems for which the 'yes' answers have simple proofs because of the fact that the answer is indeed 'yes'. More precisely, these proofs have to be verifiable in polynomial time by a deterministic TM. In an equivalent formal definition, NP is the set of decision problems solvable in polynomial time by a non-deterministic TM. The Figure indicates the relation between P and NP problems.

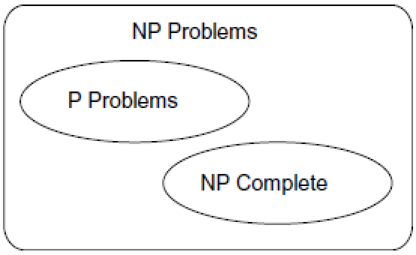


Fig. 8.1 *Class of P and NP Problems*

The complexity class P is contained in NP, but NP contains many important problems, called NPC problems, for which no polynomial-time algorithms are known. The most important open question in complexity theory, the P = NP problem, asks whether such algorithms actually exist for NPC problems. It is widely believed that this is not the case.

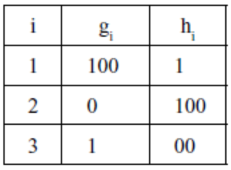
Post’s Correspondence Problem

Def: Given an alphabet S, one instance of Post’s Correspondence Problem (PCP) of size s is a finite set of pairs of strings (g_i, h_i) (i = 1 ... s s ≥ 1) over the alphabet S. A solution of length n ≥ 1 to this instance is a sequence i1 i2 ... in of selections such that the strings g_{i1} g_{i2} ... g_{in} and h_{i1} h_{i2} ... h_{in} formed by concatenation are identical.

Width of a PCP instance is the length of the longest string in gi and h_i (i = 1, 2, ..., s). We use *Pair* i as a short name for pair (gi, hi), where gi and hi are the top string and bottom string of the pair, respectively. Mostly, people are interested in *optimal solution*, which has the shortest length over all possible solutions to an instance. The corresponding length is called *optimal length*. We use the word *hard* or *difficult* to describe instances whose optimal lengths are very large. For simplicity, we restrict the alphabet S to {0, 1}, and it is easy to transform other alphabets to their equivalent binary format.

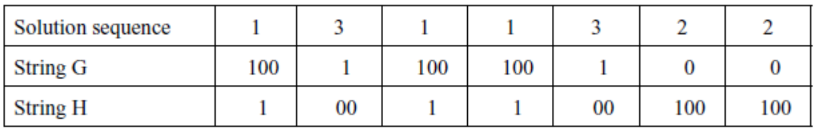
To describe subclasses of Post’s Correspondence Problem, we use PCP[s] to represent the set of all PCP instances of size s, and PCP[s, w] to represent the set of all PCP instances of size s and width w.

For convenience, we use a matrix of 2 rows and s columns to represent instances of PCP[s], where string gi is located at (i, 1) and hi at (i, 2). The following is the matrix representation of the instance {{100, 1}, {0, 100}, {1, 00}} in PCP [3, 3].



i	g_i	h_i
1	100	1
2	0	100
3	1	00

Let us consider the results of selections of sequence {1, 3, 1, 1, 3, 2, 2} accordingly. They are shown in the following table and each selection is shown in the table.



Solution sequence	1	3	1	1	3	2	2
String G	100	1	100	100	1	0	0
String H	1	00	1	1	00	100	100

After the elimination of blanks and concatenation of strings in the top and bottom rows separately, we get

1001100100100

1001100100100

Now, the string in the top is identical to the one in the bottom. Therefore, these selections form a solution to the PCP problem.

NP-hard and NP-complete

- The subject of *computational complexity theory* is focused on classifying problems by how hard they are.
- There are many different classifications depending on the time taken by the problem.
- The following are the types of classification.

- P problems are those that can be solved by a Turing machine (TM) (deterministic) in polynomial time. (‘P’ stands for polynomial).
- P problems form a class of problems that can be solved efficiently.
- NP problems are those that can be solved by non-deterministic TM in polynomial time.
- A problem is in NP if you can quickly (in polynomial time) test whether a solution is correct (without worrying about how hard it might be to find the solution).
- NP problems are a class of problems that cannot be solved efficiently.
- NP does not stand for ‘non-polynomial’.
- There are many complexity classes that are much harder than NP.

1. Differentiate recursive and recursively enumerable languages

Recursive languages	Recursively enumerable languages
1. A language is said to be recursive iff there exists a membership algorithm for it.	1. A language is said to be r.e. if there exists a TM that accepts it.
2. A language L is recursive if there is a TM that decide languages have algorithms.	2. L is recursively enumerable if there is a TM that partially decidable L. (Turing acceptable languages). TMs that partially decides languages are not algorithms.