Theory of Computations

Reference Books:

1. Formal languages and Automata 5th Edition, By Peterlinz
2. Introduction to the theory of computation 2nd Edition, by M. Sipser

This PDF contains the notes from the standards books and are only meant for GATE CSE aspirants.

Notes Compiled By-
Manu Thakur
Mtech CSE, IIT Delhi
worstguymanu@gmail.com
https://www.facebook.com/Worstguymanu

Theory of Computations

Alphabet: finite, nonempty set Σ of symbols, called the alphabet.

Strings which are finite sequences of symbols from the alphabet

Concatenation of two strings w and υ is the string obtained by appending the symbols of υ to the right end of w, that is, if

$w = a_1 a_2 \cdots a_n$ And $v = b_1 b_2 \cdots b_m$, then the concatenation of w and υ, denoted by **wυ**, is

$$wv = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$$

Reverse of a string is obtained by writing the symbols in reverse order.

If $w = a_1 a_2 \cdots a_n$ then $w^R = a_n \cdots a_2 a_1$

tring.

$$|\lambda| = 0,$$
$$\lambda w = w\lambda = w$$ **Empty string**, which is a string with **no symbols at**

Length of a string w, denoted by |w|, is the number of symbols in the s **all**. It will be denoted by **λ**.

$|u^n| = n\,|u|$ for all strings u and all n.

Substring string of consecutive symbols in some w. If w = uv

Prefix if w = abbab then prefix = {λ, a, ab, abb, abba, abbab}

Suffix: if w = abbab then suffix = { λ, b, ab, bab, bbab, abbab}

If **w is a string**, then w^n stands for the string obtained by **repeating w, n times.** As a special case defined as $w^0 = \lambda$,

If Σ is an alphabet, then we use Σ* to denote the set of strings obtained **by concatenating zero or more symbols** from Σ. The set Σ* always contains λ. $\Sigma^+ = \Sigma^* - \{\lambda\}$

While Σ is **finite** by assumption, **Σ* and Σ^+ are always infinite** since there is no limit on the length of the strings in these sets.

A language is defined very generally as a subset of Σ*. A string in a language L will be called a **sentence** of L.

The set $\{a, aa, aab\}$ is a language on Σ. Because it has a finite number of sentences, we call it a **finite language**.

The **complement of a language** is defined $\overline{L} = \Sigma^* - L$

The **reverse of a language** is the set of all string reversals, that is, $L^R = \{w^R : w \in L\}$

$$a^R = a,$$
$$(wa)^R = aw^R$$

For all a ∈ Σ, w ∈ Σ*.

1. $(uv)^R = v^R u^R$ For all u, υ ∈ Σ+.
2. $(w^R)^R = w \text{ for all } w \in \sum^*$

The **concatenation of two languages** L1 and L2 is the set of all strings obtained by concatenating **any element of L1 with any element of L2**; specifically $L_1L_2 = \{xy : x \in L_1, y \in L_2\}$

For every language L., We define **L^n** as **L concatenated with itself n times**, with the special cases

$$L^0 = \{\lambda\}$$

And

$$L^1 = L$$

We define the **star-closure** of a language as

$$L^* = L^0 \cup L^1 \cup L^2 \cdots$$

And the **positive closure** as

$$L^+ = L^1 \cup L^2 \cdots .$$

Example: If

$$L = \{a^n b^n : n \geq 0\}$$

Then

$$L^2 = \{a^n b^n a^m b^m : n \geq 0, m \geq 0\}$$ n and m are unrelated.

The of L is

$$L^R = \{b^n a^n : n \geq 0\}$$

Grammars

A grammar G is defined as a **quadruple** **G = (V, T, S, P)**

Where V is a **finite** set of objects called **variables**,
T is a **finite** set of objects called **terminal symbols**,
S ∈ V is a special symbol called the **start variable**,
P is a **finite** set of productions.

$w \Rightarrow z$ We say that w derives z or that z is derived from w.

$w_1 \overset{*}{\Rightarrow} w_n$ The * indicates that an **unspecified number of steps (including zero)** can be taken to derive wn from w1.
The set of all **such terminal strings is the language defined or generated by the grammar**.

Let G = (V, T, S, P) be a grammar. Then the set

$$L(G) = \left\{ w \in T^* : S \overset{*}{\Rightarrow} w \right\}$$

is the language generated by G.

Sentential forms

If w ∈ L (G), then the sequence. $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$
is a **derivation** of the sentence w.
The strings **S, w1, w2… wn**, which contain variables as well as terminals, are called **sentential forms** of the derivation.

Two grammars ***G*1 and *G*2 are equivalent** if they generate the same language

$$L(G_1) = L(G_2)$$

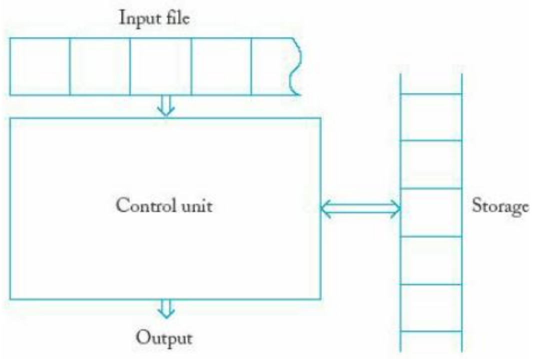
Note:- A given language has many grammars that generate it.

Automata

An automaton is an abstract model of a digital computer. It will be assumed that the input is a string over a given alphabet, written on an **input file**, **which the automaton can read but not change**.
The input file is divided into **cells**, each of which can hold one symbol. The input mechanism can read the input file from left to right, one symbol at a time.
The input mechanism can also detect the **end of the input** string (by sensing an end-of-file condition)

The automaton has a **control unit**, **which can be "in" any one of a finite number of internal states**, and which can change state in some defined manner.

Transition function it gives the next state in terms of the **current state**, the **current input symbol**, and the information currently in the **temporary storage**.



Deterministic Automata
A deterministic automaton is one in which each move is uniquely determined by the current configuration. If we know the internal state, the input, and the contents of the temporary storage, we can predict the future behavior of the automaton exactly.

Nondeterministic automata

Nondeterministic automaton may have several possible moves, so we can only predict a set of possible actions.

Note

a. An automaton whose output response is limited to a simple "yes" or "no" is called an **accepter**.
b. Automaton, capable of producing strings of symbols as output, is called a **transducer**.

Properties:-

1. (L*)' != (L')* as epsilon doesn't belong to LHS language.
2. $(L_1L_2)^R = L_2^R L_1^R$ for all languages L1 and L2.
3. (L*)* = L*
4. $(L_1 \cup L_2)^R = L_1^R \cup L_2^R$ for all language L1 and L2.
5. $(L^R)^* = (L^*)^R$ for all languages L.
6. Let L be any language on non-empty alphabet, both L and L' can't be finite as **L U L' = Σ***
7. Take Σ = { a, b}, and let na(w) and nb(w) denote the number of a's and b's in the string w, respectively. Then the grammar G with productions:

$$\begin{aligned} S &\rightarrow SS, \\ S &\rightarrow \lambda, \\ S &\rightarrow aSb, \\ S &\rightarrow bSa \end{aligned}$$

$$L = \{w : n_a(w) = n_b(w)\}$$

(b) $L = \{w : n_a(w) > n_b(w)\}$.

*(c) $L = \{w : n_a(w) = 2n_b(w)\}$.

Solution

(b) $S \rightarrow aS \,|S_1S|\, aS_1$.

where S_1 derives the language in Example 1.13.

(c) $S \rightarrow aSbSa \,|aaSb|\, bSaa \,|SS|\, \lambda$.

Finite Automata

A **deterministic finite accepter** or dfa is defined by the quintuple **M = (Q, Σ, δ, q0, F)**,

Where Q is a **finite** set of **internal states**,
Σ is a **finite** set of symbols called the input alphabet,
δ :Q × Σ → Q is a **total function** called the **transition function**,
q0 ∈ Q is the **initial** state,
F ⊆Q is a set of **final** states.

Extended transition function

$$\delta^* : Q \times \Sigma^* \to Q.$$

The second argument of δ* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string.

The language accepted by a DFA M = (Q, Σ, δ, q0, F) is the set of all strings on Σ accepted by M.
In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$$

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}$$

Nondeterministic Finite Accepters

Nondeterminism means **a choice of moves** for an automaton.
Rather than prescribing a unique move in each situation, we allow a set of possible moves.

A **nondeterministic finite accepter** or nfa is defined by the **quintuple**:
M=(Q, Σ, δ, q0, F)

Where Q, Σ, q0, F are defined as for deterministic finite accepters, but

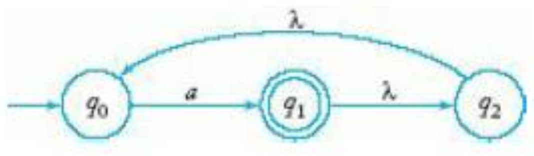
$$\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$$

In a nondeterministic accepter, **the range of δ is in the powerset 2^Q, so that its value is not a single element of Q but a subset of it**. This subset defines the set of all possible states that can be reached by the transition.

$$\delta(q_1,a) = \{q_0,q_2\}$$

1. Either q0 or q2 could be the next state of the nfa.
2. We allow λ as the second argument of δ. This means that the nfa can make a transition without consuming an input symbol.
3. Although we still assume that the input mechanism can only travel to the right, **it is possible that it is stationary on some moves**.
4. Finally, in an nfa, **the set δ (qi,a) may be empty**, meaning that **there is no transition defined** for this specific situation.

A string is accepted by an nfa **if there is some sequence of possible moves that will put the machine in a final state at the end of the string**. A string is **rejected** (that is, not accepted) only i**f there is no possible sequence of moves** by which a final state can be reached.



$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "a"] & q_1 \arrow[r, "\lambda"] & q_2 \arrow[ll, bend right, "\lambda"'] \end{tikzcd}$$

We see that δ*(q2, λ) contains q0. Also, **<u>since any state can be reached from itself by making no move</u>**, thus

$$\delta^*(q_2, \lambda) = \{q_0, q_2\}$$
$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

The language L accepted by an nfa **M = (Q, Σ, δ, q0, F)** is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}$$

The language consists of all strings w for which <u>there is a walk labeled w from the initial vertex of the transition graph to some final vertex</u>.

<u>Reduction of the Number of States in Finite Automata</u>

For a given language, there are many dfa's that accept it.

<u>For a given language, there are many DFA's that accept it.</u>

1. Two states p and q of a dfa are called **<u>indistinguishable</u>** or **<u>equivalent</u>** if

$$\delta^*(p, w) \in F \text{ implies } \delta^*(q, w) \in F,$$

OR

$$\delta^*(p, w) \notin F \text{ implies } \delta^*(q, w) \notin F,$$

On same input w, states p and both are either going to final state or non-final state will be called **<u>equivalent states.</u>**

2. For all w ∈ Σ*. If, on the other hand, there exists some string w ∈ Σ* such that

$$\delta^*(p, w) \in F \text{ and } \delta^*(q, w) \notin F,$$

OR vice versa then the states p and q are said to be **<u>distinguishable</u>** by a string w.

1. First remove **<u>unreachable</u>** states from initial state
2. Then, merge **<u>equivalent</u>** states.
3. If there are more than one final states then check for final states also if they are equivalent or not

<u>Regular Languages and Regular Grammars</u>

Let Σ be a given alphabet. Then

1. Ø, λ and a ∈ Σ are all regular expressions. These are called **primitive regular expressions**.
2. If r1 and r2 are regular expressions, so are r1+ r2, r1.r2, r1*, and (r1).

<u>Languages Associated with Regular Expressions</u>

1. Ø is a regular expression denoting the empty set { },
2. λ is a regular expression denoting {λ}.

3. For every a ∈ Σ, a is a regular expression denoting {a}.

Types of Grammars:

1. **Regular Grammar (Type – 3 grammar) (Regular Language)**

   a. **Right Linear Grammar** a grammar G =(V, T, S, P) is said to be **right-linear** if all productions are of the form.

   A → xB,

   A → x,

   where $A, B \in V$, and $x \in T^*$

   b. **Left Linear Grammar** a grammar is said to be **left-linear** if all productions are of the form.

   A → Bx,

   A → x

   where $A, B \in V$, and $x \in T^*$

   Conversion from Left Linear grammar to Right Linear grammar

   Given any left-linear grammar G with productions of the form

   A → Bx,

   A → x

   We construct from it a right-linear grammar G^ by replacing every such production of G with

   A->x^RB

   A-> x^R respectively. It will give the reverse of the language.

   L(G) = (L(G^))^R

2. **Context Free Grammar (Type – 2 grammar) (Context Free Language)**

A grammar G = (V, T, S, P) is said to be **context-free** if all productions in P have the form

A → x where $A \in V$ and $x \in (V \cup T)^*$

3. **Context Sensitive Grammar (Type – 1 grammar) (Context Sensitive Language)**

   A grammar G = (V, T, S, P) is said to be **context-sensitive** if all productions are of the form

   X → y and $|x| \le |y|$

   where $x, y \in (V \cup T)^+$ and

4. **Unrestricted Grammar (Type – 0 grammar) (RE Language)**

   A grammar G =(V, T, S, P) is called **unrestricted** if all the productions are of the form

   u → v

   where u is in $(V \cup T)^+$ and v is in $(V \cup T)^*$

   In an unrestricted grammar, essentially no conditions are imposed on the productions.

Context-Free Languages

Simple grammar or S-grammar

A context-free grammar G = (V, T, S, P) is said to be a **simple grammar** or **s-grammar** if all its productions are of the form.

A → ax,

Where **A ∈ V, a ∈ T, x ∈ V***, and **any pair (A, a) occurs at most once** in P.

a. The grammar S → aS | bSS | c
   Is an s-grammar.
b. The grammar
   S-> aS | bSS | aSS | c
   Is not an s-grammar because the pair (S,a) occurs in the two productions S→aS and S→aSS.

Note: - If G is an **s-grammar**, **any string w in L(G) can be parsed with efforts proportional to |w|**. Each step produces one terminal symbol and hence the whole process must be completed in no more than |w| steps.

Ambiguity in Grammars and Languages

1. A context-free grammar G is said to be **ambiguous** if there exists some w ∈ L(G) that has **at least two distinct derivation trees**.
2. Alternatively, ambiguity implies the existence of **two or more leftmost or rightmost derivations**.

Inherently ambiguous

If L is a context-free language for which there exists an **unambiguous grammar**, then L is said to be **unambiguous**. If every grammar that generates L is ambiguous, then the language is called **inherently ambiguous**.

Simplification of Context-Free Grammars and Normal Forms

1. Removing Useless Productions

Remove productions from a grammar that can never take part in any derivation.
Let G = (V, T, S, P) be a **context-free grammar**. A **variable A ∈ V** is said to be useful if and only if there is **at least one w** ∈ L (G) such that

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w,$$

With x, y in (V U T)*. A variable is useful if and only if it occurs in at least one derivation.

A variable that is not useful is called **useless**. **A production is useless if it involves any useless variable**.

There are two reasons why a variable is useless
a. Either it can't be reached from start variable S.
b. It cannot derive a terminal string.

1. First remove those variables which can't generate at least a terminal string.
2. Second. Remove those variables those can't be reached from start variable. Draw a dependency graph and remove them.
3. Remove those productions also where useless appear.

Let $G = (V, T, S, P)$ be a context-free grammar. Then there exists an equivalent grammar $\hat{G} = \left(\hat{V}, \hat{T}, S, \hat{P}\right)$ that does not contain any useless variables or productions.

<u>Removing λ-Productions</u>

Any production of a context-free grammar of the form

$$A \rightarrow \lambda$$

Is called a **λ-production**.

Any variable A for which the derivation

$$A \stackrel{*}{\Rightarrow} \lambda$$

Is possible is called **<u>nullable</u>**.

Note: A grammar <u>may generate a language not containing λ</u>, yet have some λ-productions or nullable variables. In such cases, **<u>the λ-productions can be removed</u>**.

1. Let G be any context-free grammar with λ not in L (G). Then there exists an equivalent grammar having no λ-productions.
2. If a language contains empty string then we can't remove **<u>λ-productions</u>** from the grammar.

<u>Chomsky Normal Form</u>

DEFINITION 2.8

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \rightarrow BC$$
$$A \rightarrow a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \rightarrow \varepsilon$, where S is the start variable.

<u>Note:- we can convert any grammar into CNF</u>

1. If the start symbol S occurs on some right side, we create a new start variable S' and add a new productions S' → S.
2. Then, we eliminate all **<u>λ-productions</u>** of the form **A → λ**
3. We also eliminate all **unit rules** of the form A → B.
4. If there is production S->S remove it as it's trivial production.
5. We need (2n-1) productions to generate n length string.

Any context-free grammar $G = (V, T, S, P)$ with $\lambda \notin L(G)$ has an equivalent grammar $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ in Chomsky normal form.

<u>Greibach Normal Form</u>

We put restrictions not on the length of the right sides of a production, but <u>on the positions in which terminals and variables can appear</u>.

A context-free grammar is said to be in **Greibach normal form** if all productions have the form

A → ax,

Where **a ∈ T** and **x ∈ V***

Example-

The grammar

$$
\begin{aligned}
S &\rightarrow AB, \\
A &\rightarrow aA\,|bB|\,b, \\
B &\rightarrow b
\end{aligned}
$$

From CFG to GNF

$$
\begin{aligned}
S &\rightarrow aAB\,|bBB|\,bB, \\
A &\rightarrow aA\,|bB|\,b, \\
B &\rightarrow b,
\end{aligned}
$$

Note:-

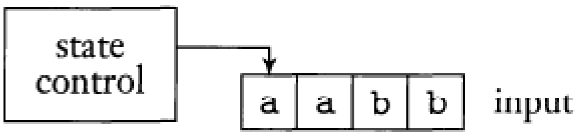
For every context-free grammar G with $\lambda \notin L(G)$, there exists an equivalent grammar $\widehat{G}$ in Greibach normal form.

<u>A Membership Algorithm for Context-Free Grammars</u>

CYK Membership and parsing algorithms for context-free grammars exist that require approximately **|w|^3** steps to parse a string w.
<u>CYK algorithm works only if the grammar is in CNF</u> and succeeds by breaking one problem into a sequence of smaller ones.

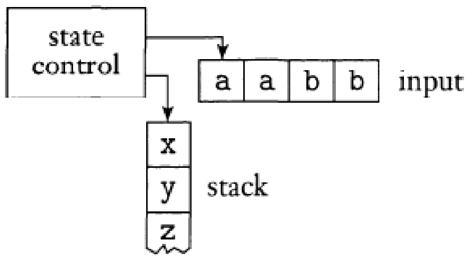
Push Down Automata

Pushdown automata, are like non-deterministic finite automata but have an extra component called a **stack.**
Pushdown automata are equivalent in power to context-free grammars.



Schematic of finite automata

With the addition of a stack component we obtain a schematic representation of a PDA



Schematic of PDA

A pushdown automaton (PDA) can write symbols on the stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack. At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up. Writing a symbol on the stack is often referred to as ***pushing*** the symbol, and removing a symbol is referred to as ***popping*** it. Note that all access to the stack, for both reading and writing, may be done only at the top. In other words a stack is a "last in, first out" storage device. If certain information is written on the stack and additional information is written afterward, the earlier information becomes inaccessible until the later information is removed.

A nondeterministic pushdown accepter (npda) is defined by the septuple

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

Where
Q is a finite set of internal states of the **control unit**,
Σ is the **input alphabet**,
Γ is a finite set of symbols called the **stack alphabet**
δ: Q × (Σ ∪ {λ}) × Γ → set of finite subsets of Q × Γ* is the transition function.
q0 ∈ Q is the initial state of the control unit,
z ∈ Γ is the **stack start symbol**,
F ⊆ Q is the set of final states.

Transition function: δ: Q × (Σ ∪ {λ}) × Γ → set of finite subsets of Q × Γ*

1. The arguments of δ are the **current state of the control unit**, the **current input symbol**, and the **current symbol on top of the stack**.
2. The **result is a set of pairs (q, x)**, where **q is the next state of the control unit** and **x is a string** that is put on top of the stack in place of the single symbol there before.

3. Note that the second argument of δ **<u>may be λ</u>**, indicating that a move that does not consume an input symbol is possible. **<u>We will call such a move a λ-transition</u>**.
4. Note also that <u>δ is defined so that it needs a stack symbol</u>; **<u>no move is possible if the stack is empty</u>**.
5. Finally, the requirement that the elements of the range of δ be a **<u>finite subset is necessary because Q × Γ* is an infinite set and therefore has infinite subsets</u>**.
6. While an <u>npda may have several choices for its moves</u>, this **<u>choice must be restricted to a finite set of possibilities</u>**.

Suppose the set of transition rules of an npda contains

$$\delta(q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}.$$

If at any time the control unit is in state q1, the input symbol read is a, and the symbol on top of the stack is b, **<u>then one of two things can happen</u>**: (1) the control unit goes into state q2 and the string cd replaces b on top of the stack, or (2) the control unit goes into state q3 with the symbol b removed from the top of the stack. In our notation **<u>we assume that the insertion of a string into a stack is done symbol by symbol, starting at the right end of the string</u>**.

<u>The Language Accepted by a Pushdown Automaton</u>

$$L(M) = \left\{ w \in \Sigma^* : (q_0, w, z) \overset{*}{\vdash}_M (p, \lambda, u), p \in F, u \in \Gamma^* \right\}$$

The language accepted by M is the **<u>set of all strings that can put M into a final state at the end of the string</u>**.
<u>The final stack content u is irrelevant to this definition of acceptance</u>.

<u>Deterministic Pushdown Automata and Deterministic Context-Free Languages</u>

A **<u>deterministic pushdown accepter</u>** (**<u>dpda</u>**) is a pushdown automaton that never has a choice in its Move.

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

Where
Q is a finite set of internal states of the **control unit**,
Σ is the **input alphabet**,
Γ is a finite set of symbols called the **stack alphabet**
It is subject to the restrictions that, for every q ∈ Q, a ∈ Σ ∪ {λ} and b ∈ Γ,
1. δ(q, a, b) **<u>contains at most one element</u>**.
2. If δ (q, λ, b**<u>) is not empty</u>**, then δ **<u>(q, c, b) must be empty for every c ∈ Σ</u>**.

- The first of these conditions simply requires that for any given input symbol and any stack top, **<u>at most one move can be made</u>**.
- The second condition is that when **<u>a λ-move is possible for some configuration, no input-consuming alternative is available</u>**.

Note:-
1. We retain λ-transitions in DPDA also.
2. λ-transitions **<u>does not automatically imply nondeterminism</u>**. Also, some transitions of a dpda

may be to the empty set, that is, undefined, so there may be **dead configurations**.

3. Only criterion for determinism is that at all times at most one possible move exists.

A language L is said to be a deterministic context-free language if and only if there exists a dpda M such that L = L (M).

A Pumping Lemma for Context-Free Languages

If A is a context-free language, then there is a **number p** (the pumping length) where, **if s is any string in A of the length at least p**, then **s may be divided into five pieces s = uvxyz** the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$
2. $|vy| > 0$, and
3. $|vxy| \leq p$

1. Case 2: When S is being divided into uvxyz, condition 2 says that either v or y is not empty string**.**
2. Case 3: the pieces v, x and y together have length at most p.

Note: -

1. A DPDA with acceptance by **EMPTY STACK** is proper subset of the languages accepted by a DPDA with **final state.**
2. For each DCFL which satisfied prefix property, can be accepted by a DPDA with empty stack.

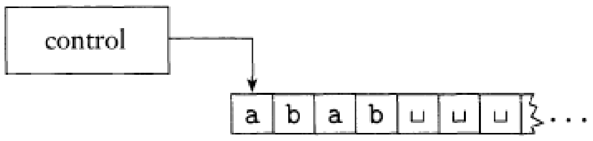
A Pumping Lemma for Regular Languages

Given an infinite regular language L, there exists an integer p (pumping leema length) for any string w, such that |w|>=p, we can divide string w into xyz.
|xy|<=p (at most of length p) and |y|>=1, is not null.
Such that for i>=0 xy^iz belongs to L.

Turing Machine

A Turing Machine can do everything that a real computer do. Nonetheless, even a Turing machine cannot solve certain problems. **Those problems are beyond the theoretical limits of computation**.

1. The Turing machine model uses an **infinite tape as unlimited memory**.
2. It has a **tape head** that can **read and write** symbols and move around on the tape.
3. Initially **the tape contains only the input string** and is blank everywhere else.
4. If the machine needs to store information, it may **write the information on the tape**.
5. To read the information that it has written, the machine can move its **head back** over it.
6. The machine continues computing until it decides to produce an output.
7. The outputs **accept** and **reject** are obtained by **entering designated accepting and rejecting states**.
8. If it **doesn't enter an accepting or a rejecting state, it will go on forever**, **never halting**.



Schematic of a Turing Machine

The following list summarizes the differences between finite automata and Turing machines
1. A Turing machine can both write on the tape and read from it.
2. The read-write head can move both to the left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting **take effect immediately**

Definition of a Turing Machine

A Turing machine M is defined by

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

Where
Q is the set of **internal states**,
Σ is the **input alphabet** not containing blank symbol.
Γ is the finite set of symbols called the **tape alphabet**,
δ is the **transition function**,
$\square \in \Gamma$ is a special symbol called the **blank**,
q0 ∈ Q is the **initial state**,
F ⊆ Q is the **set of final states**.

Note: - In the definition of a Turing machine, we assume that $\Sigma \subseteq \Gamma - \{\square\}$

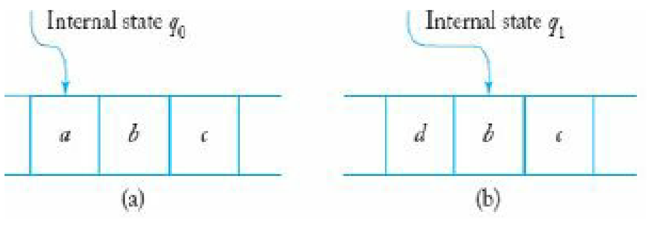
Transition Function: In general, **δ is a partial function on Q × Γ**; for a Turing machine, δ takes the form: $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$

When machine is in a certain state q and the head is over a tape cell containing a symbol a, and If δ(q, a) = (r, b, L), the **machine writes the symbol b replacing a** and machine goes to state r.

The third component is either L or R and indicates whether the head moves to the left or right after writing. In this case the L indicate a move to the left.

Example: shows the situation before and after the move

δ (q0, a) = (q1, d, R).



A Turing machine is said to halt whenever it reaches a configuration for which δ is not defined; **this is possible because δ is a partial function**. We will assume that no transitions are defined for any final state, so the Turing machine will halt whenever it enters a final state.

Turing Machines as Language Accepters

1. A string w is written on the tape, with blanks filling out the unused portions.
2. The machine is started in the i**nitial state q0 with the read write head positioned on the leftmost symbol of w**.
3. If, after a sequence of moves, the Turing machine enters a final state and halts, then w is considered to be accepted.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a Turing Machine. Then the language accepted by M is

$$L(M) = \left\{ w \in \Sigma^+ : q_0 w \overset{*}{\vdash} x_1 q_f x_2 \text{ for some } q_f \in F, x_1, x_2 \in \Gamma^* \right\}$$

1. This definition indicates that the input w is written on the tape with **blanks on either side**.
2. Exclusion of blanks from the input assures us that all the input is re**stricted to a well-defined region of the tape**, **bracketed by blanks on the right and left**.
3. Without this convention, the machine could not limit the region in which it must look for the input; no matter how many blanks it saw, it could never be sure that there was not some nonblank input somewhere else on the tape.

Note that the Turing machine also halts in a final state if started in state q0 on a blank. We could interpret this as acceptance of λ, but for technical reasons the empty string is not included.

Note:-
The collection of strings that M accepts **is the language of M**, or the **language recognized by M**, denoted L(M)

Call a language ***Turing-recognizable*** if some Turing machine recognizes it.[1]

Call a language ***Turing-decidable*** or simply ***decidable*** if some Turing machine decides it.[2]

Variants of Turing Machine

1. **Multiple Turing Machine** It's like an ordinary T.M with several tapes. **Each tape has its own head for reading and writing**. Initially the input appears on tape 1, and the others start out blank. The transition function is:

$$\delta: Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R, S\}^k$$

Where k is the number of tapes.

Every multiple Turing machine has an equivalent single-tape Turing machine

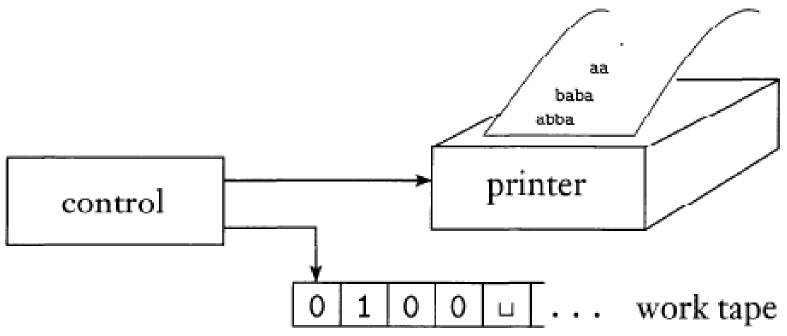
2. **Nondeterministic Turing Machine** transition function for a NTM has the form

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

I think, P is powerset.

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

3. **Enumerators** Some people use the term recursively enumerable language for Turing-recognizable language. Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings.



Schematic of an enumerator

An enumerator E starts with a blank input tape. If the enumerator doesn't halt it may print an infinite list of strings. The language enumerated by E is the collection of all strings that eventually prints out.

A language is Turing recognizable if and only if some enumerator enumerates it.

1. If we have an enumerator E that enumerates a language A, **A TM M recognizes A**. The TM M works in the following way:

   M – "On input w"

   a. Run E. Every time that E outputs a string, compare it with w
   b. If w ever appears in the output of E, **accept.**

   Clearly, M accepts those strings that appear on E's list.

2.

Now we do the other direction. If TM M recognizes a language A, we can construct the following enumerator E for A. Say that $s_1, s_2, s_3, \dots$ is a list of all possible strings in Σ^*.

```
E = "Ignore the input.
    1. Repeat the following for $i = 1, 2, 3, \dots$
    2.     Run $M$ for $i$ steps on each input, $s_1, s_2, \dots, s_i$.
    3.     If any computations accept, print out the corresponding $s_j$."
```

Decidability

Closure Properties

Operations	REG	DCFL	CFL	CSL	REC	RE
Union	✓	X	✓	✓	✓	✓
Intersection	✓	X	X	✓	✓	✓
Complement	✓	✓	X	✓	✓	X
Concatenation	✓	X	✓	✓	✓	✓
Kleene star	✓	X	✓	✓	✓	✓
Homomorphism	✓	X	✓	X	X	✓
Inverse Homomorphism	✓	✓	✓	✓	✓	✓
Reverse	✓	X	✓	✓	✓	✓
Substitution (∈ - free)	✓	X	✓	✓	X	✓

1. All types of languages are closed under all the operations with regular languages such as LUR, L∩R, L-R.
2. CFLs are not closed under difference operation as L1-L2 = L1∩L2^c, and CFLs are not closed under complement operation.
3. No languages are closed under subset ⊆ and Infinite union.
4. Regular languages are not closed under **infinite UNION** and **infinite INTERSECTION**
5. If L is DCFL then so are **MIN(L)** and **MAX(L)**.
6. Complement of **non-regular** is always non-regular
7. If L be a DCFL and R is a regular language then **L/R** is DCFL.
8. DCFL U CFL = CFL
9. If something is closed under UNION and COMPLEMENT then it will be surely closed under INTERSECTION.

Operations	REG	CFL
INIT	✓	✓
L/a	✓	✓
CYCLE	✓	✓
MIN	✓	X
MAX	✓	X
HALF	✓	X
ALT	✓	X

Let L be a language

1. **HALF(L)** = {x | for some y such that |x| = |y| and xy ∈ L }
2. **MIN(L)** = { w | w is in L and no proper prefix of w is in L}
3. **MAX(L)** = {w | w is in L and for no x other than epsilon wx is in L}
4. **INIT(L)** = {w | for some x, wx is in L}
5. **CYCLE(L)** = { w | we can write w as w=xy such that yx is in L}
6. **ALT(L, M)** is regular provided that L and M are regular languages.
7. **SHUFFLE(L, L')** is a CFL if L is CFL and L' is regular.
8. **SUFFIX(L) =** { y | xy ∈ L for some string x }, **CFL is closed under SUFFIX operation.**
9. **NOPREFIX(L)** = {w ∈ A | and no prefix of w is member of A }
10. **NOEXTEND(L) =** { w ∈ A | w is not proper prefix of any string in A }
11. **DROP-OUT(L)** let A be any language, define DROP-OUT(L) to be the language containing all strings that can be obtained by **removing one symbol from a string in L.**
12. Regular languages are closed under NOPREFIX, NOEXTEND, and DROP-OUT operations.

Operations	**REG**	**DCFL**	**CFL**	**CSL**	**REC**	**RE**	**Comments**
w ∈ L(G)	✓	✓	✓	✓	✓	X	Membership property.
L(G)=Φ	✓	✓	✓	X	X	X	Emptiness Property
L(G)=Σ∗	✓	✓	X	X	X	X	Language accepts everything?
L(G1)⊆L(G2)	✓	X	X	X	X	X	Is L(G1) subset of L(G2)?
L(G1)=L(G2)	✓	✓	X	X	X	X	Are both languages equal?
L(G1)∩L(G2)=Φ	✓	X	X	X	X	X	Disjointness Property.
L(G) is regular?	✓	X	X	X	X	X	G generates Regular language.
L(G) is finite?	✓	✓	✓	X	X	X	G generates finite language?
Ambiguity	✓	X	X	X	X	X	Is the given grammar ambiguous?

<u>**Decidability Table**</u>

1. **w ∈ L(G)** upto REC every language satisfies membership property.

Problem 1: <u>For a given CFG grammar G and string w, does G generate w?</u>

$$A_{CFG} = \{\langle G, w\rangle | G \text{ is a CFG that generates string } w\}$$

ACFG is a **decidable language.**

Method 1:

1. Go through <u>**all derivations generated by G**</u> checking whether any is a derivation of w. Since there are infinitely many derivations this idea does not work. <u>If does not generate the algorithm doesn't halt</u>. I.e, this idea provides a <u>**recognizer but not a decider**</u>.

Method 2:

2. Make the recognizer a decider. For that we need to ensure that the al<u>**gorithm tries only finitely many derivations**</u>.
   If is a CFG in <u>Chomsky normal form</u> then for any **w ∈ L(G)** where |w| = n **exactly 2n-1** steps are required for any derivation of w.
   a. Convert CFG into CNF
   b. For a string w of length |w|=n, n>1, <u>**list all derivations with 2n-1 steps**</u>.
   c. If n <= 1 list all derivations with 1 step
   d. If any of the derivations listed above generates w, <u>**accept**</u>, if not <u>**reject**</u>**.**

Problem 2: For a given CFG G is L(G) = Φ?
Language:

ECFG ={ G | G is a CFG and L(G) = Φ } is **decidable language.**

To test whether L(G) is empty we need to test whether the productions of G can generate a string of terminals.

a. Determine for each variable whether that variable can generate a string of terminals.
b. When the algorithm determines that a variable can generate a string of terminals the algorithm mark that variable.
c. The <u>algorithm start by marking first all terminals</u>. Then <u>**it marks variables**</u> that have on their rhs in some rules only terminals.
d. If the start symbol **S is not marked, accept** otherwise **reject.**

Problem 3: for a CFL L and a string w does w belong to L? i.e. is there a CFG G such that w ∈ L(G)?

Language: ACFL = {< L > | L is a CFL and w a string}
Use the <u>TM S that decides string generation problem by converting into Chomsky normal form</u>.

Let G be a CFG for L, i.e L(G) = L. Design a TM M that decides L by building a copy of G into M"
M = "on input w"

1. Run TM S on input <G, w>
2. If this machine accepts, <u>**accept**</u>; if it rejects, <u>**rejects**</u>.

Problem 4: For two CFL languages generated by two CFGs A and B **is L(A) = L(B) true?**

Language: EQCFG = {<A, B> | A, B CFGs and L(A) = L(B)}

Since the class of CFL is not closed under **INTERSECTION** and **COMPLEMENT** we cannot use the symmetric difference for EQCFG.

The symmetric difference of A and B, $L(C) = L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B)$
And test if L(C) is empty. But CFLs are not closed under complement and intersection, and CSL is not closed under emptiness property. <u>**Hence EQCFG is UNDECIDABLE.**</u>

<u>**Problem 5:**</u> **Is it decidable whether a given context free grammar generates a finite language?**

GATECSE: checking if <u>**L(CFG) is finite is decidable**</u> because we just need to see if L(CFG) contains any string with length **between n and 2n−1**, where <u>n is the pumping lemma constant</u>. If so, <u>**L(CFG) is infinite otherwise it is finite**</u>**.**

Stackoverflow:
Let G be a context free grammar, and let us assume that it is in Chomsky normal form. If it's not, we'll convert it first. An important property of this normal form is that the only way to derive the empty word is with the single rule S0→ϵ (where S0 is the initial variable, which cannot be derived from other variables).
Thus, any other derivation adds some non-trivial part to a word. Now, let <u>**n be the number of variables in the grammar, and let k be the maximal length on the right-side of a derivation rule**</u>. That is, A→B1⋯BkA→B1⋯Bk is the maximal length.

<u>**G generates an infinite word iff it generates a word of length at least k^(n+2)**</u>

Decidable Problems concerning regular languages

Problem1: $A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$

The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether <B, w> is a member of the language ADFA. **ADFA is decidable.**

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:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

Problem 2: $A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$

ANFA is decidable. We could design a TM N that decides ANFA.

N = "On input <B, w> where B is an NFA and w is a string"

1. Convert NFA B to an equivalent DFA C.
2. Run another TM M on input <C, w>
3. If M accepts, **accept**; otherwise, **reject**.

Problem3: $A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$

AREX is decidable.
The following TM P decides AREX.
P = "On input <R, w> where R is regular expression and w is a string"

1. Convert regular expression R to an equivalent NFA A.
2. Run TM N on <A, w>
3. If N accepts, **accept**; otherwise, **reject**.

Problem 4: $E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$

EDFA is decidable.
A DFA accepts some string iff reaching an accept state from the start by travelling along the arrows of the DFA is possible. To test this condition we can design a TM T.
T = "On input <A> where A is a DFA

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into from any state that is already marked.
4. If no accept state is marked, **accept**; otherwise, **reject**.

Problem 5: $EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$

EQDFA is decidable, calculate **symmetric difference**.
We construct a new DFA C from A and B, where C accepts only those strings that are accepted by **either A or B but not by both.** Thus, if A and B recognize the same language, C will accept nothing.

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

Problem 6: $A = \{\langle R, S\rangle \mid R \text{ and } S \text{ are regular expressions and } L(R) \subseteq L(S)\}$.

Solution: $\boldsymbol{L(R) \subseteq L(S)}$ **if and only if** $\boldsymbol{\overline{L(S)} \cap L(R) = \emptyset}$.

Que 7: $ALL_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \Sigma^*\}$ **is decidable?**

The following TM L decides ALLDFA.
L = "On input <A> where A is DFA

1. Construct DFA B that recognizes $\overline{L(A)}$
2. Run Turing machine M on input B, If DFA B accepts empty language.
3. If T accepts, **accept**. If T rejects, **reject**.

Que 8: A = {<G> | G is a CFG that generates ϵ } **is Decidable language**
We can apply membership algorithm for this problem.

Que 9: $\boldsymbol{INFINITE_{\text{DFA}} = \{\langle A\rangle \mid L(A) \text{ is an infinite language}\}}$ the following TM I decides INFINITEDFA.

I = "On input <A> where A is DFA:

1. Let k be the number of states of A.
2. Construct a DFA D that accepts strings of length >=k
3. Construct a DFA M such that **L(M) = L(A)∩L(D)**.
4. Run TM M on input <M> to check emptiness property.
5. If **T accepts then reject. If T rejects, accept**.

If A accepts infinitely many strings, it must accept strings of at least k length.

4.23 Let $BAL_{\text{DFA}} = \{\langle M\rangle \mid M$ is a DFA that accepts some string containing an equal number of 0s and 1s$\}$. Show that BAL_{DFA} is decidable. (Hint: Theorems about CFLs are helpful here.)

4.24 Let $PAL_{\text{DFA}} = \{\langle M\rangle \mid M$ is a DFA that accepts some palindrome$\}$. Show that PAL_{DFA} is decidable. (Hint: Theorems about CFLs are helpful here.)

4.25 Let $E = \{\langle M\rangle \mid M$ is a DFA that accepts some string with more 1s than 0s$\}$. Show that E is decidable. (Hint: Theorems about CFLs are helpful here.)

4.26 Let $C = \{\langle G, x\rangle \mid G$ is a CFG that generates some string w, where x is a substring of $w\}$. Show that C is decidable. (Suggestion: An elegant solution to this problem uses the decider for E_{CFG}.)

1. Let A = { <M> | M is a DFA which doesn't accept any string containing an odd number of 1's} is **Decidable language**.
2. Let A = {<G> | G is a CFG over {0, 1} and 1*∩L(G) != Φ} is **Decidable.**
3. LCFG = {<G, k> | G is a CFG, L(G) contains exactly k strings where k>=0 or k=infinity} is **decidable.**

___________________ TURING MACHINE ____

From my notes:-

1. HALT-TM = {<M, w> | M is a TM and M halts on input w} – **Undecidable, RE**
2. E-TM = {<M> | M is a TM and L(M) = Φ} – **Undecidable, NOT RE**
3. EN-TM = {<M> | M is a TM and L(M) != Φ} – **Undecidable, RE**
4. REGULAR-TM = {<M> | M is a TM and L(M) is a regular language} – **Undecidable, NOT RE**
5. REGULAR-TM = {<M> | M is a TM and L(M) is a REC}, **Undecidable, NOT RE**
6. REGULAR-TM = {<M> | M is a TM and L(M) is a NOT REC}, **Undecidable, NOT RE**
7. EQ-TM = {<M1, M2> | M1 and M2 are TMs and L(M1) = L(M2)} – **Undecidable, NOT RE**
8. A-LBA = {<M, w> | M is an LBA that accepts string w} – **Decidable**
9. E-LBA = {<M> | M is an LBA where L(M) = Φ} – **Undecidable, NOT RE**
10. All-CFG = {<G>|G is a context free grammar and L(G) = Σ∗} **Undecidable, NOT RE**
11. T = {<M > | M is a TM that accepts w^r whenever it accepts w} **undecidable, RE?**
12. A TM ever writes a blank symbol over a non-blank symbol during the course of its computation. **Undecidable, NOT RE**
13. L3 = {< G> | G is ambiguous} **RE**, while L3' is **NOT RE**.

Note:-

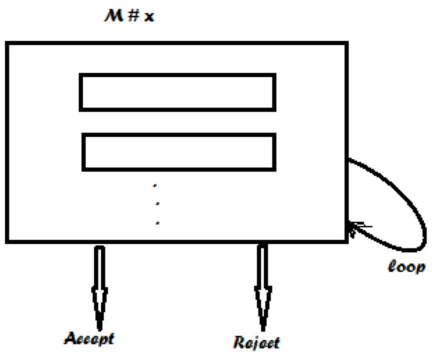
1. TM >> LBA(FA + 2 counter) > NPDA(NFA + 1 counter) > DPDA (DFA + 1 counter) > NFA = DFA
2. Any TM with m symbols and n states can be simulated using **4mn + n** states by other TM.
3. If A <=p B (A is polynomial reducible to B, if A is NOT RE then B is NOT RE too.
4. A <=p A', If A is Turing recognizable, then A is decidable.

<u>**Some Decidable/ Undecidable Problems**</u>

a. L(M) has **at least** 10 strings – **RE**
b. L(M) has **at most** 10 strings – **NOT RE**
c. L = {M | M is a TM that accepts a string of length 2014} – **RE**
There are finite number of strings of length 2014, if we can execute multiple instances of TM in parallel, if any string is accepted we can stop.
d. L(M) is recognized by a TM having even number of states. **Decidable (trivial property)**
e. If $A \leq_m B$ and B is a regular language, does that imply that A is a regular language? **(NO)**
f. if a language A is in RE and $A \leq_m \overline{A}$, then A is recursive. **(TRUE)**
g. L={⟨M,w⟩|M does not modify the tape on input w} - **decidable**
h. an arbitrary TM ever prints a specific letter – **Undecidable**

GATEBOOK

Universal or Total Turing machine.



a. m#x is an input to UTM where n is binary coding of a valid TM.
b. # is a separator.
c. X is an input string to Turing machine M, x is binary representation.
d. We will simulate TM M on x by UTM, if M accepts x, then UTM will accept **m#x**.
e. If m is not valid representation of a TM, reject it without doing any simulation. UTM will reject it.

Decidable – If the language/ property has a total T.M
Semi-decidable – If the language has just a TM
Undecidable – No TM exists.

Give a Turing machine A
(a) A has at least 481 states **(decidable)**
(b) A takes more than 481 steps on epsilon **(decidable)**
(c) A takes more than 481 steps on some input **(decidable)**
(d) A takes more than 481 steps on all inputs **(decidable)**

Solutions:

(a) Property of a TM to be part of our language: TM M has at least 481 states.
By looking at the description of TM M, we can tell how many states a TM has.

(b) L = collection of TM, takes more than 481 steps on epsilon.
Run a TM on eps, and count the steps. This process will be completed in finite time for sure. If in 481 steps if M on x has not reached accept or reject state, it will take more than 481 states.
Note – in UTM, take one more tape that counts number of steps.

(C) A takes more than 481 steps on some input
Infinite strings are possibile eps, 0, 1, 00, 01, 10, 11, 000, 0001, …….

Method 1:

Suppose on eps TM m takes only 60 steps and halts on eps. But any other input may take more than 481 steps. In worst case all the tried inputs took less than 481 steps but we cannot stop because any i/p among we didn't try may take more than 481 steps. **In worst case TM will always keep running and keep checking.**
Method 2: take all the permutations of inputs of length 481 and run inputted TM M only on these inputs.

Suppose Sigma = {a, b}
1 length input = a, b
2 length input = aa, ab, ba, bb

On 2 input alphabets 2^481 strings are possible, write all of them on a tape separated by #.

S1 # S2 # S3 # S4 ……
If TM M takes less than 481 steps on S1, run M on S2 and so on **but input of length atmost 481.**

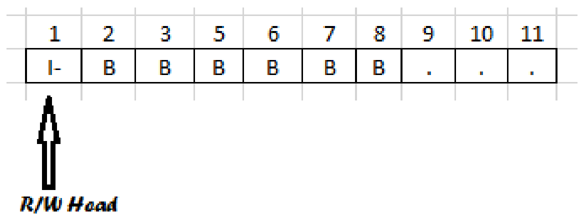
Case1: If some string takes more than 481 steps then we don't require to check bigger set.

Case2: all strings in small set not taking more than 481 steps (halting on less no. of steps)
Now we should check for bigger set strings, but that is not required because
Take 500 length strings, divide strings into two parts 481 symbols + 19 symbols.
In case of the strings of 500 length also, TM will start from the beginning of the string, as we already checked for 481 symbols part, if this smaller part has been accepted than TM will remain in the same state for next 19 symbols also.

Hence, by looking at small part only, we can tell that whether a TM will take more than 481 steps on some input or not. It's not required to check for bigger set.

(d) Takes more than 481 steps on all inputs.
Run M on all the strings of length up to 481 symbols, if M on any string halts in less than 481 steps, reject the TM.
If all strings upto the 481 length run at least for 481 steps accept the language (**no infinite loop)**

<u>Problem</u>: given a TM, which ever moves its head more than 481 tape cells away from the left end marker on input epsilon?



Suppose we run TM for 5 steps and Header remain in the same position in first cell. We check in which cell currently header is there. But we can't conclude anything because header may move to right after 5 steps.
<u>It's a decidable language</u>, because we can check for all the possible combinations of no. of heads can be moved*no. of states*no. of ways each cell can be modified.

No. of symbols = 2 = {a, b}
No. of states = 2 = {q1, q2}

No. of ways we can move our head pointer = 3 ways
No. of ways we can change the tape pattern = 3^2 ways, 2 tape symbol + blank
Total combinations = 3*2*3^2 = 54
Now, if we run our TM for 55 times (54+1) TM will be either in any of these 54 configurations or Somewhere else in new configuration. Systematic approach hence **decidable.**

Note 1:

1. **Element distinctness problem** A TM M is given a list of strings over {0, 1} separated by #, as follows s1 # s2 # s3 # S4 # . . . .
   Its job is to accept if all the strings are different **is decidable problem.**
2. D = {P | P is a polynomial with multi variable with an integral root} is **RE** but **NOT REC.**
3. D = {P | P is a polynomial with single variable with an integral root} is **REC.**
4. A = {<G> | G is a connected undirected graph } – **decidable**

https://www.cs.rice.edu/~nakhleh/COMP481/final_review_sp06_sol.pdf

Prob1: L1=$\{\langle M\rangle | M \text{ is a TM and there exists an input on which } M \text{ halts in less than } |\langle M\rangle| \text{ steps}\}$
Is a **Decidable** language.
It first finds the length of <M>, and stores it. Then, it runs M on **all inputs of length at most |<M>|, for at most |<M>|steps**, and accepts if M accepts at least one of the strings within the specified number of steps.

Prob2: $L_2 = \{\langle M\rangle | M \text{ is a TM and } |L(M)| \leq 3\}$ **(NOT RE)**
Language of M contains at most 3 strings. It's **NOT RE.** we have check that this TM M won't accept more than 3 strings that is not possible. After checking 1 crore strings may be it will accept 4th string. **Hence, it's NOT RE language.**

Prob3: $L_3 = \{\langle M\rangle | M \text{ is a TM and } |L(M)| \geq 3\}$**(RE)**
M* that **semidecides** the language, runs M on all inputs in **an interleaved mode**, and halts whenever 3 inputs have been accepted. Notice that M¤ generates the input strings for M one by one as they are needed (It is not allowed that M* first generates all strings, and then starts running M on them, since generating the inputs takes infinite time!).

Prob4: $L_4 = \{\langle M\rangle | M \text{ is a TM that accepts all even numbers}\}$ **(NOT RE)**
It keeps running to check if TM rejects at least one even number.

Prob5: $L_5 = \{\langle M\rangle | M \text{ is a TM and } L(M) \text{ is finite}\}$ **(NOT RE)**

Prob6: $L_6 = \{\langle M\rangle | M \text{ is a TM and } L(M) \text{ is infinite}\}$ **(NOT RE)**

Prob7: $L_7 = \{\langle M\rangle | M \text{ is a TM and } L(M) \text{ is countable}\}$ **(Decidable)**
This is the language of all TM's, since there are no uncountable languages. (Over finite alphabets and finite-length strings).

Prob8: $L_8 = \{\langle M \rangle | M \text{ is a TM and } L(M) \text{ is uncountable}\}$ **(Decidable)**
This is the empty set; there are no uncountable languages (over finite alphabets and finite-length strings).
Prob9: $L_9 = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are two TMs, and } \varepsilon \in L(M_1) \cup L(M_2)\}$ **(RE)**
M* that semidecides the language run the two machines on epsilon " (it interleaves the run between the machines), and accepts if at least one of them accepts. But **not recursive** in worst case both machines can keep running.

Prob10: $L_{10} = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are two TMs, and } \varepsilon \in L(M_1) \cap L(M_2)\}$**(RE)**

Prob11: $L_{11} = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are two TMs, and } \varepsilon \in L(M_1) \setminus L(M_2)\}$. **(NOT RE)**

Prob12: $L_{12} = \{\langle M \rangle | M \text{ is a TM, } M_0 \text{ is a TM that halts on all inputs, and } M_0 \in L(M)\}$ **(RE)**
M0 is a halting TM which halts on all inputs. If TM M0 is a member of L(M) then M will halt on M0 but if M0 is not member of L(M) then M will keep running. Hence it's **RE language.**

Prob13: $L_{13} = \{\langle M \rangle | M \text{ is a TM, } M_0 \text{ is a TM that halts on all inputs, and } M \in L(M_0)\}$ **(REC)**
M0 is HTM, which halts on every input. When M is inputted to M0, M0 will always halt, either it accepts of rejects M.

Prob14:
$L_{14} = \{\langle M, x \rangle | M \text{ is a TM, } x \text{ is a string, and there exists a TM, } M', \text{ such that } x \notin L(M) \cap L(M')\}$.
(Recursive) For any TM, M, there is always a TM, M', such that $x \notin L(M) \cap L(M')\}$
In particular, take M' to be machine that rejects all inputs. Intersection of M' and M will accept nothing.

Prob15:
$L_{16} = \{\langle M \rangle | M \text{ is a TM, and there exists an input whose length is less than 100, on which } M \text{ halts}\}$.
(RE) it semidecides the language runs M on all strings of length at most 100 in an interleaved mode, and halts if M accepts at least one.

Prob16: $L_{17} = \{\langle M \rangle | M \text{ is a TM, and } M \text{ is the only TM that accepts } L(M)\}$ **(REC)**
This is the empty set, since every language has an infinite number of TMs that accept it.

Prob17: $L_{19} = \{\langle M \rangle | M \text{ is a TM, and } |M| < 1000\}$. **(REC)**
We are talking about all the descriptions of Turing machines using a fixed alphabet (of finite size, of course), i.e., TM's that are encoded as input to the universal TM. So, **L19 is finite**, and hence **recursive**.

Prob18: $L_{21} = \{\langle M \rangle | M \text{ is a TM, and } M \text{ halts on all palindromes}\}$ **(NOT RE)**

Prob19: $L_{22} = \{\langle M \rangle | M \text{ is a TM, and } L(M) \cap \{a^{2^n} | n \geq 0\} \text{ is empty}\}$. **(NOT RE)**

This language contains all TM's that do not accept any string of the form a^2^n. TM keeps on checking infinitely before it reaches this conclusion, hence NOT RE.

Prob20:
$L_{23} = \{\langle M, k\rangle | M$ is a TM, and $|\{w \in L(M) : w \in a^*b^*\}| \geq k\}$.

– **RE**. Easy; prove it yourself.

Prob21:
$L_{24} = \{\langle M\rangle | M$ is a TM that halts on all inputs and $L(M) = L'$ for some undecidable language $L'\}$.
(Recursive)
R. Since M halts on all inputs, then $L(M)$ is decidable, and hence it can't be that $L(M)$ equals some undecidable language L'. Therefore, $L_{24} = \emptyset$ which is recursive.

Prob22: $L_{25} = \{\langle M\rangle | M$ is a TM, and M accepts (at least) two strings of different lengths$\}$ **(RE)**
It semidecides the language runs M on all inputs in an interleaved mode, halts and accepts once M accepts two strings of different lengths.

Prob23: $L_{26} = \{\langle M\rangle | M$ is a TM such that both $L(M)$ and $\overline{L(M)}$ are infinite$\}$ **(NOT RE)**

- $L_{27} = \{\langle M, x, k\rangle | M$ is a TM, and M does not halt on x within k steps$\}$.
  - **R**. Easy.
- $L_{28} = \{\langle M\rangle | M$ is a TM, and $|L(M)|$ is prime$\}$.
  - **Not RE**. Reduce $\overline{HP}$ as follows. $\tau(\langle M, w\rangle) = \langle M^*\rangle$. M^* on input x:

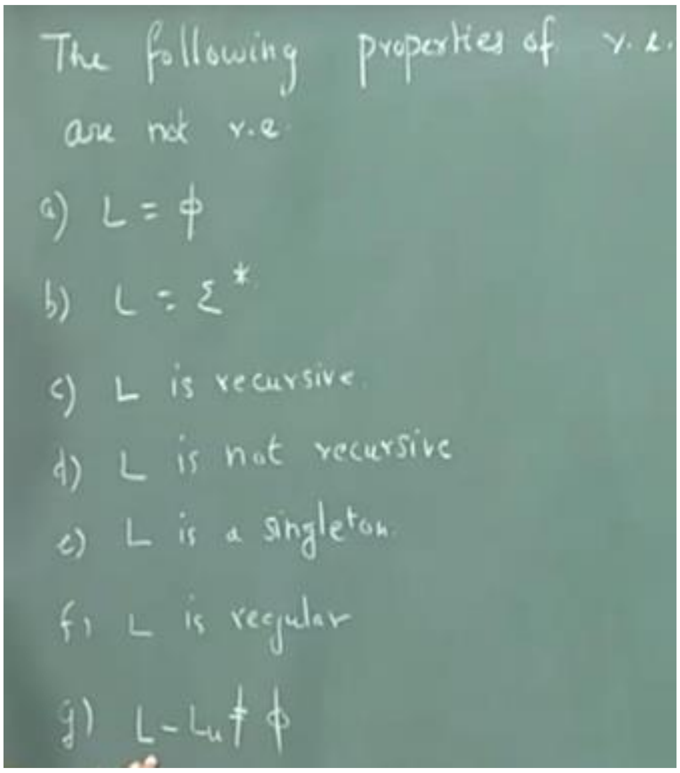
$L_{30} = \{\langle M\rangle |$ there exist $x, y \in \Sigma^*$ such that either $x \in L(M)$ or $y \notin L(M)\}$.

– **Recursive**. This is the language of all Turing machines!

- $L_{33} = \{\langle M\rangle | M$ does not accept any string w such that 001 is a prefix of $w\}$. **(NOT RE)**

- $L_{34} = \{\langle M, x\rangle | M$ does not accept any string w such that x is a prefix of $w\}$.
  - **Not RE**. Proof similar to L_{33}.
- $L_{35} = \{\langle M, x\rangle | x$ is prefix of $\langle M\rangle\}$.
  - **Recursive**. The machine M^* that decides the language simply checks whether the string x is a prefix of the string $\langle M\rangle$.
- $L_{36} = \{\langle M_1, M_2, M_3\rangle | L(M_1) = L(M_2) \cup L(M_3)\}$.
  - **Not RE**. Reduction from $\overline{HP}$. $\tau(\langle M, w\rangle) = \langle M_1, M_2, M_3\rangle$, where

- $L_{37} = \{\langle M_1, M_2, M_3\rangle | L(M_1) \subseteq L(M_2) \cup L(M_3)\}$.
  - **Not RE**. Reduction from $\overline{HP}$. $\tau(\langle M, w\rangle) = \langle M_1, M_2, M_3\rangle$, where
- $L_{38} = \{\langle M_1\rangle |$ there exist two TM's M_2 and M_3 such that $L(M_1) \subseteq L(M_2) \cup L(M_3)\}$.
  - **Recursive**. This is the language of all Turing machines (simply take M_2 to be the machine that accepts everything, for example).
- $L_{39} = \{\langle M, w\rangle | M$ is a TM that accepts w using at most $2^{|w|}$ squares of its tape$\}$. **(REC)**



The following properties of r.e.

are not r.e

a) $L = \phi$

b) $L = \Sigma^*$

c) L is recursive

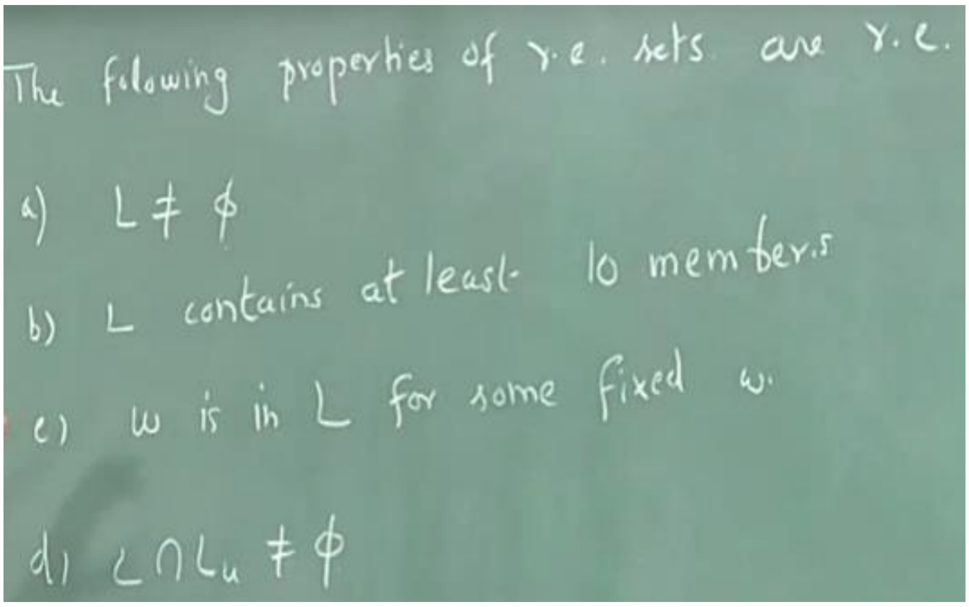
d) L is not recursive

e) L is a singleton

f) L is regular

g) $L - L_u \neq \phi$

Singleton: if TM only accepts one string



The folowing properties of r.e. sets are r.e.

a) $L \neq \phi$

b) L contains at least 10 members

c) w is in L for some fixed w.

d) $L \cap L_u \neq \phi$

http://gatecse.in/rices-theorem/

Complement a DFA:

In a given DFA,

1. Convert final states into non-final states, and
2. Convert non-final states into final states.
3. Don't change initial state

This DFA will accept Complement of the language accepted by the original DFA.

Reversal of DFA

L = Language start with a.
L^r = Language ends with a.

In the given DFA,

1. Make the final states as initial state.
2. Make the initial state as final state.
3. Reverse all the transition from q0 → q1 to q1→q0 for any two states in DFA.
4. Self-loops are unchanged.
5. Reversal of a DFA may result into a DFA or an NFA.
6. If there are multiple initial state into resulted DFA (or NFA) take an initial state and add all initial state with epsilon transitions.