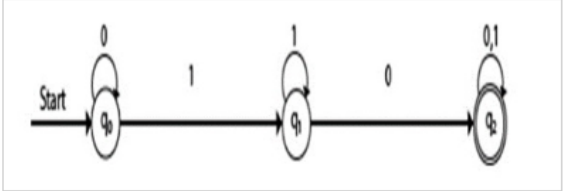
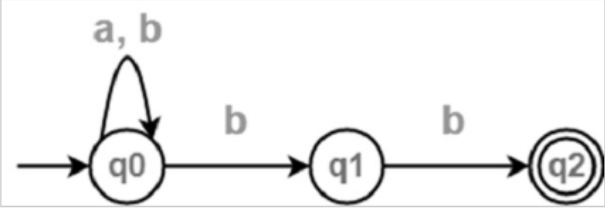


Theory Questions:

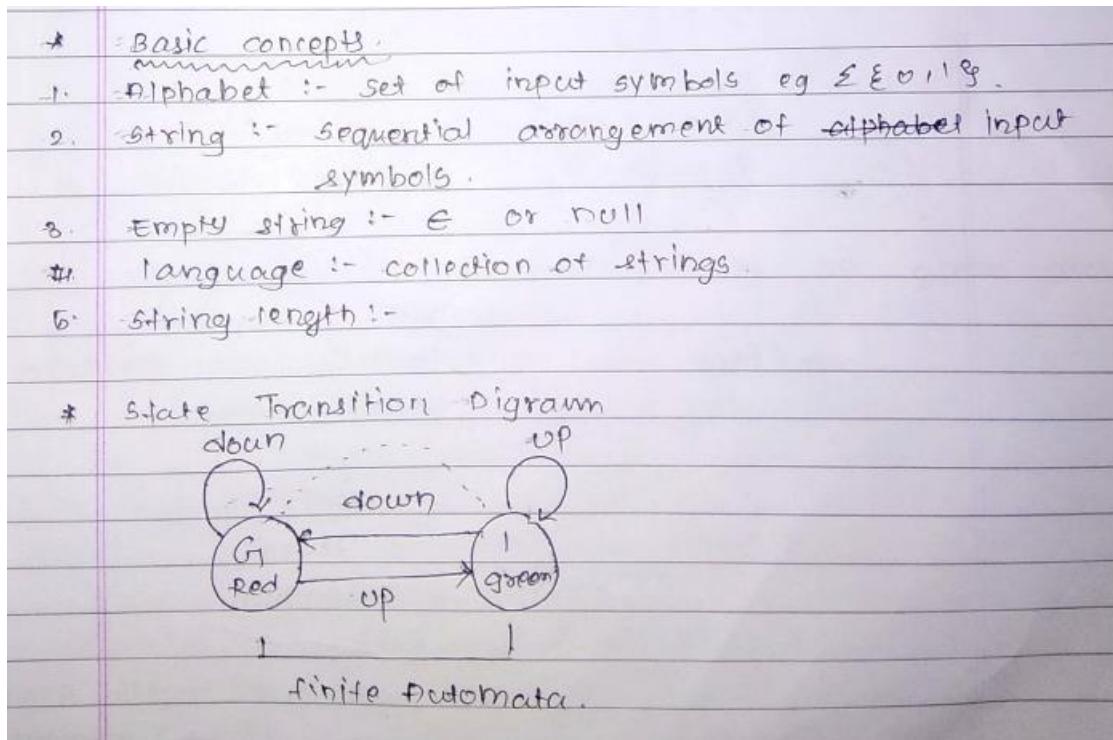
Mod-1

DFA vs NFa

Deterministic Finite Automata	Non-Deterministic Finite Automata
Each transition leads to exactly one state called as deterministic	A transition leads to a subset of states i.e. some transitions can be non-deterministic.
Accepts input if the last state is in Final	Accepts input if one of the last states is in Final.
Backtracking is allowed in DFA.	Backtracking is not always possible.
Requires more space.	Requires less space.
Empty string transitions are not seen in DFA.	Permits empty string transition.
For a given state, on a given input we reach a deterministic and unique state.	For a given state, on a given input we reach more than one state.
DFA is a subset of NFA.	Need to convert NFA to DFA in the design of a compiler.
$\delta : Q \times \Sigma \rightarrow Q$ For example – $\delta(q_0, a) = \{q_1\}$	$\delta : Q \times \Sigma \rightarrow 2^Q$ For example – $\delta(q_0, a) = \{q_1, q_2\}$
DFA is more difficult to construct.	NFA is easier to construct.
DFA is understood as one machine.	NFA is understood as multiple small machines computing at the same time.
	

Concept of Automata

Automata theory is a branch of theoretical computer science that deals with the logic of computation and the construction of abstract machines. Automata are mathematical models of computation that help in understanding the capabilities and limitations of different computing devices. Here's an overview of the concept of automata and its applications:



Epsilon NFA is a part of Finite Automata. ϵ is a symbol that represents empty inputs. ϵ -NFA is the representation that allows an automaton to change its state without input. ϵ -Non-Deterministic Finite Automata has a different transition function than regular NFA. Here is the formal definition of ϵ -NFA.

ϵ -NFA is defined in 5 tuple representation $\{Q, q_0, \Sigma, \delta, F\}$ where

- Q is the set of all states,
- q_0 is the initial state,
- Σ is the set of input symbols,
- δ is the transition function which is $\delta: Q \times (\Sigma \cup \epsilon) \rightarrow 2^Q$ and
- F is the set of final states.

Mod-2

Regular Language Properties:

Properties of Regular Expressions

R is regular expression whose language is L

1. Kleene closure : $R^* = (a)^* \Rightarrow L^* = \{\epsilon, a, aa, aaa, \dots\}$
2. Positive closure : $R^+ = (a)^+ \Rightarrow L^+ = \{a, aa, aaa, aaaa, \dots\}$
3. Complement : $L \rightarrow (\Sigma^* - L)$; complement of regular is regular
4. Union : $L_3 = (L_1 \cup L_2)$; L_3 is also regular
5. Concatenation : $L_3 = (L_1 . L_2)$; L_3 is also regular
6. Intersection : $L_3 = (L_1 \cap L_2)$; L_3 is also regular
7. Difference : $L_3 = (L_1 - L_2)$; L_3 is also regular
8. Reverse : $L = \{0, 01, 100\} \Rightarrow L^R = \{0, 10, 001\}$
9. Quotient : divider completely cut off from left or right
 - i) if $L_1 = \{11\}$ & $L_2 = \{10\}$ then $L_1/L_2 = \epsilon$
 - ii) if $L_1 = \{10\}$ & $L_2 = \{10\}$ then $L_1/L_2 = \epsilon$

5. Practical Applications of Regular Languages

There are many ways regular languages are used in computer science and related fields. A few examples include:

- **Pattern matching:** They are often used in text editors, word processors, and programming languages for searching and manipulating strings that match a given pattern
- **Lexical analysis:** Regular languages are used in the lexical analysis phase of compiler design to identify and tokenize keywords, identifiers, and other elements of a programming language
- **Input validation:** Regular languages are used in programming to validate user input by checking if it matches a given pattern
- **Network protocols:** Regular languages are used to define the syntax of messages in network protocols such as HTTP, FTP, and SMTP
- **DNA sequence analysis:** Regular languages are used to analyze DNA sequences in bioinformatics

Arden's Theorem:

In order to find out a regular expression of a Finite Automaton, we use Arden's Theorem along with the properties of regular expressions.

Statement –

Let **P** and **Q** be two regular expressions.

If **P** does not contain null string, then **R = Q + RP** has a unique solution that is **R = QP***

Proof –

$$\begin{aligned} R &= Q + (Q + RP)P \quad [\text{After putting the value } R = Q + RP] \\ &= Q + QP + RPP \end{aligned}$$

When we put the value of **R** recursively again and again, we get the following equation –

$$R = Q + QP + QP^2 + QP^3 \dots$$

$$R = Q (\epsilon + P + P^2 + P^3 + \dots)$$

$$R = QP^* \quad [\text{As } P^* \text{ represents } (\epsilon + P + P^2 + P^3 + \dots)]$$

Hence, proved.

Assumptions for Applying Arden's Theorem

- The transition diagram must not have NULL transitions
- It must have only one initial state

Pumping lemma for regular languages:

Pumping Lemma

conditions :-

- i) $xy^iz \in A$ for every $i \geq 0$
- ii) $|xy| \leq P$ for original s of every case
- iii) $|y| > 0$

To prove language is NOT REGULAR

$$A = \{a^n b^n \mid n \geq 0\}$$

Assume A is regular and pumping length P .

assume value of P ; say $P = 7$

let s be $a^P b^P \Rightarrow s = aaaaaaaabbbbbbb$

Divide s in x, y and z

case 1 : y is in 'a' part

$$s = \underbrace{aaaaaa}_x \underbrace{a}_y \underbrace{bbbbbb}_z$$

case 2 : y is in 'b' part

$$s = \underbrace{aaaaaa}_x \underbrace{abbb}_y \underbrace{bbb}_z$$

case 3 : y is in 'ab' part

n 1 2

case 3 : y is in 'ab' part

$$s = \underbrace{aaaaaaaa}_{n} \underbrace{abbb}_{y} \underbrace{bbbbb}_{z}$$

For all cases show $xy^iz \notin A$ for some i ; say $i=2$
 i.e no. of a's \neq no. of b's and it follows pattern $a^n b^n$
 \therefore does not lie in original language.

xy^2z : no. of a's \neq no. of b's

$$s = \underbrace{aa}_{x} \underbrace{aaaa}_{y} \underbrace{aaaa}_{y} \underbrace{abbbbbbb}_{z}$$

Chomsky Hierarchy:

The Chomsky Hierarchy is a classification of formal grammars, named after linguist and cognitive scientist Noam Chomsky. It categorizes formal languages into four types based on the generative power of the corresponding grammars. The hierarchy is often represented as follows, from least to most powerful:

1. Type 3 - Regular Languages (Regular Grammars):

- Description: These languages can be recognized by finite automata, including deterministic and nondeterministic finite automata, as well as regular expressions.
- Key Feature: Regular languages are the simplest and have limited expressive power compared to higher types.

2. Type 2 - Context-Free Languages (Context-Free Grammars):

- Description: These languages can be recognized by pushdown automata, and they are generated by context-free grammars.
- Key Feature: Context-free languages are more expressive than regular languages and are commonly used in the syntax of programming languages.

3. Type 1 - Context-Sensitive Languages (Context-Sensitive Grammars):

- Description: These languages can be recognized by linear-bounded automata, and they are generated by context-sensitive grammars.
- Key Feature: Context-sensitive languages have greater generative power than context-free languages but are less common in practical applications.

4. Type 0 - Recursively Enumerable Languages (Unrestricted Grammars):

- Description: These languages can be recognized by a Turing machine, and they are generated by unrestricted grammars.
- Key Feature: Recursively enumerable languages have the highest generative power and can describe any language that can be recognized by a computing device, although at the expense of decidability.

As you move up the Chomsky Hierarchy, the grammars become more expressive, allowing for the definition of more complex languages. However, this increase in expressive power comes at the cost of increased complexity in the corresponding automata or computational devices required for recognition. The hierarchy provides a theoretical framework for understanding the relationships between different classes of languages and their associated grammars.

Mod-4

Q.3: Can a Turing machine simulate any modern computer?

Answer:

Yes, in theory, a Turing machine can simulate any modern computer, as long as there is enough tape and time available. However, practical considerations, such as the enormous amount of tape needed for even simple computations, make this infeasible for complex computations.

Universal Turing machine:

A Universal Turing Machine (UTM) is a theoretical construct in computer science and mathematics that plays a fundamental role in the theory of computation. It was introduced by Alan Turing in the 1930s as part of his work on the foundations of computation.

Here are key points about the Universal Turing Machine:

1. Simulation of Turing Machines:

- A UTM is designed to simulate the behavior of any other Turing Machine (TM).
- Instead of having a fixed, pre-defined set of rules like a regular Turing Machine, the UTM takes as input the description of another Turing Machine along with its input.

2. Description of Turing Machines:

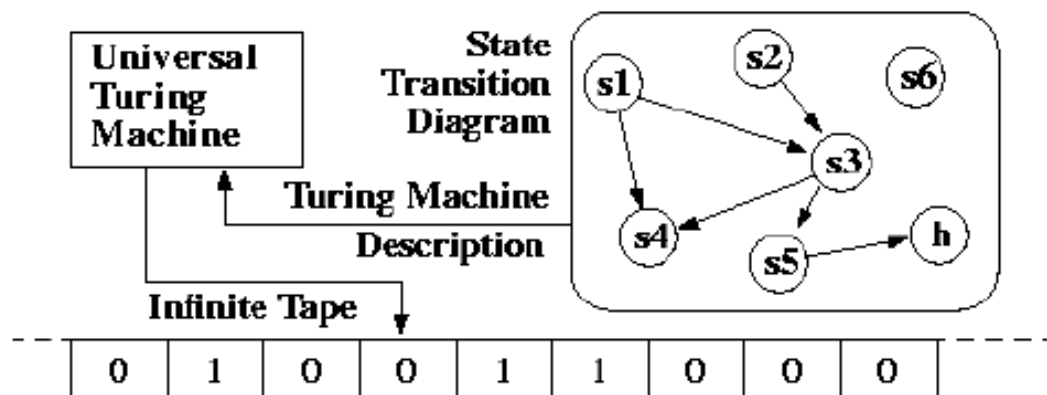
- The description includes information about the states, transitions, and symbols of the other Turing Machine.
- This description serves as the "program" for the UTM.

3. Universal Capability:

- The UTM is said to be "universal" because it can emulate the computations of any Turing Machine by reading the description and input.
- This concept is foundational to the idea that all computers, despite their specific architectures or programming languages, are equivalent in terms of computational power.

4. Proof of Undecidability:

- The existence of a UTM is crucial in proving undecidability results, such as the famous Halting Problem.



Turing Machine:

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple $(Q, X, \Sigma, \delta, q_0, B, F)$ where –

- Q is a finite set of states
- X is the tape alphabet
- Σ is the input alphabet
- δ is a transition function; $\delta : Q \times X \rightarrow Q \times X \times \{\text{Left_shift}, \text{Right_shift}\}$.
- q_0 is the initial state
- B is the blank symbol
- F is the set of final states

Variations of turing machine:

Turing Machines (TMs) come in various forms, each with its own characteristics and modifications to the basic model introduced by Alan Turing. Some notable variations include:

1. **Non-deterministic Turing Machine (NDTM):**

- Allows multiple possible transitions from a given state on a particular symbol.
- Non-deterministic choices are resolved during computation.

2. **Multi-Tape Turing Machine:**

- Has multiple tapes (each with its own tape head) working in parallel.
- Allows for more efficient computation for certain problems.

3. **Multi-Head Turing Machine:**

- Similar to the multi-tape TM but with multiple tape heads sharing a single tape.

4. **Quantum Turing Machine (QTM):**

- Utilizes principles of quantum mechanics, allowing for superposition of states.
- Can perform quantum parallelism, potentially solving certain problems faster than classical TMs.

5. **Probabilistic Turing Machine:**

- Introduces randomness into the computation.
- Transition probabilities determine the next state.

Church Turing-thesis:

TURING THESIS

Any computation that can be carried out by mechanical means can be performed by some Turing machine.

⇒ Anything that can be done by existing digital computers can also be done by TM.

⇒ No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which TM program cannot be written.

⇒ Alternative models have been proposed for mechanical computation, but none of them are more powerful than the Turing machine model.

Church-Turing Thesis

- Any mechanical computation can be performed by a Turing Machine
- There is a TM- n corresponding to every computable problem
- We can model any mechanical computer with a TM
- The set of languages that can be decided by a TM is identical to the set of languages that can be decided by any mechanical computing machine
- If there is no TM that decides problem P, there is no algorithm that solves problem P.

All of these statements are implied by the Church-Turing thesis

Halting Problem:

In other words, the Halting Problem is concerned with determining whether a particular Turing Machine, when given a specific input, will eventually reach a halting state (accept or reject the input) or whether it will continue running indefinitely.

Here's why the Halting Problem is particularly interesting:

1. Undecidability:

- Turing proved that there is no algorithm (Turing Machine) that can decide the Halting Problem for all possible inputs.
- This means there is no general procedure that, given the description of any Turing Machine and input, can determine whether that machine will eventually halt on that input or run forever.

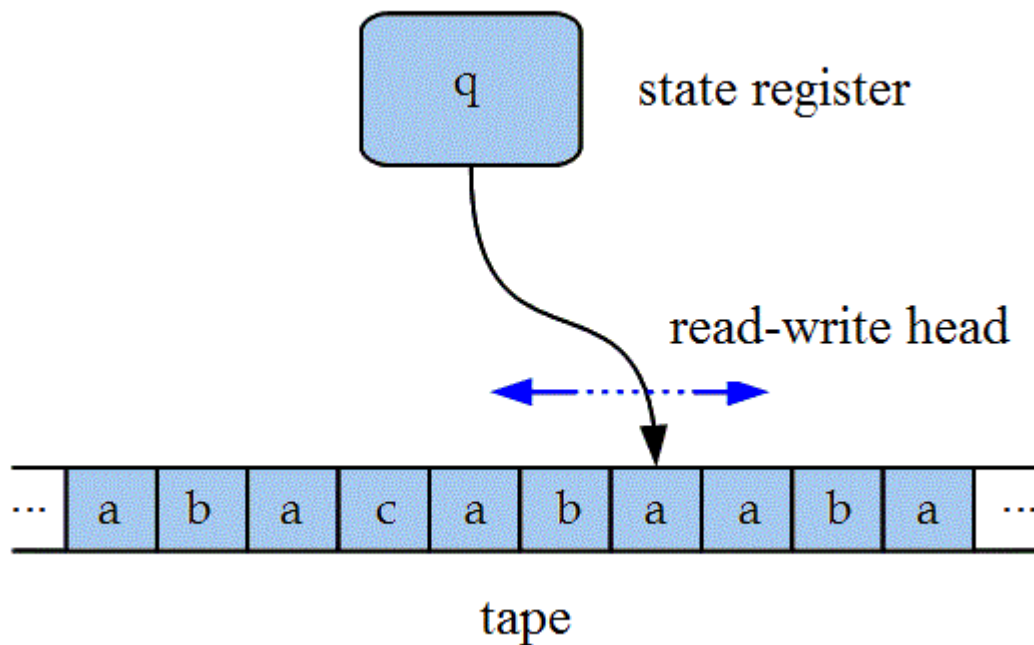
2. Proof by Contradiction:

- Turing's proof involves assuming the existence of a "universal" Turing Machine that could solve the Halting Problem.
- Using this assumption, he constructs a new Turing Machine that leads to a logical contradiction, demonstrating the impossibility of a general solution to the Halting Problem.

3. Implications for Computation:

- The undecidability of the Halting Problem has far-reaching consequences.
- It implies that there are limits to what can be algorithmically computed or decided.
- There are problems for which we cannot construct a general algorithm to determine the halting behavior of arbitrary Turing Machines.





When describing Turing machines (TMs) and their components, several key terms and concepts are commonly used in the field of theoretical computer science. Here's a list of important terminology related to Turing machines:

1. **Tape:**

- The infinite tape on which symbols can be written. It extends indefinitely in both directions.

2. **Alphabet:**

- The set of symbols that can appear on the tape.

3. **Head:**

- The read/write head that scans a single tape cell at a time and can move left or right.

4. **State:**

- The internal state of the Turing machine. The machine's behavior is determined by its current state.

5. **Transition Function (δ):**

- A function that defines how the Turing machine transitions from one state to another based on the current symbol being read.

6. **Configuration:**

- A snapshot of the Turing machine's current state, the symbols on the tape, and the position of the tape head.

7. **Accepting State:**

- A state in which the Turing machine enters when it successfully recognizes the input, indicating acceptance of the input.
8. **Rejecting State:**
- A state in which the Turing machine enters when it determines that the input is not accepted.

Hilberts problems

In simpler terms, Hilbert's Tenth Problem is about finding a way to tell whether a specific type of math problem has a solution. The problem deals with equations that involve whole numbers (integers).

Imagine you have an equation with variables (like x and y) and coefficients (numbers in front of the variables), and all these must be whole numbers. Hilbert wanted to know if there's a step-by-step method (algorithm) that can determine whether there are whole number values for the variables that make the equation true.

Mod-5

Certainly, here's a concise table summarizing the differences between recursively enumerable languages (RE) and recursive languages (R):

Feature	Recursively Enumerable Language (RE)	Recursive Language (R)
Acceptance Mechanism	Turing machine may not halt on non-members	Turing machine halts on all inputs
Decidability	Semi-decidable (may not halt on all inputs)	Decidable (halts on all inputs)
Membership Test	May not terminate on some inputs	Always terminates on any input
Language Description	Described by a Turing machine or equivalent	Described by a Turing machine
Closure Properties	Closed under union, concatenation, and Kleene star	Closed under union, concatenation, and Kleene star
Halting Guarantee	No guarantee of halting on rejection	Always halts on acceptance or rejection
Examples	The language of all Turing machine encodings	The language of even-length palindromes

Explanation:

Post Correspondance Problem:

Post Correspondence Problem

← Prev

Next →

In this section, we will discuss the undecidability of string and not of Turing machines. The undecidability of the string is determined with the help of Post's Correspondence Problem (PCP). Let us define the PCP.

"The Post's correspondence problem consists of two lists of string that are of equal length over the input. The two lists are $A = w_1, w_2, w_3, \dots, w_n$ and $B = x_1, x_2, x_3, \dots, x_n$ then there exists a non empty set of integers i_1, i_2, i_3, \dots , in such that, $w_{i_1}, w_{i_2}, w_{i_3}, \dots, w_{i_n} = x_{i_1}, x_{i_2}, x_{i_3}, \dots, x_{i_n}$ "

To solve the post correspondence problem we try all the combinations of i_1, i_2, i_3, \dots , in to find the $w_{i_1} = x_{i_1}$ then we say that PCP has a solution.

Example 1:

Undecidable problems about Turing Machine



Undecidable Problems About Turing Machine



ChatGPT

Undecidable problems about Turing machines are fundamental results in the field of theoretical computer science, demonstrating the existence of problems for which no algorithmic solution exists. Here are some notable undecidable problems related to Turing machines:

1. Halting Problem:

- **Problem:** Given a description of a Turing machine M and an input w , determine whether M halts on w or runs forever.
- **Significance:** Alan Turing first proved the undecidability of this problem, showing that there is no general algorithm to decide whether an arbitrary Turing machine halts on a given input.

2. Post Correspondence Problem (PCP):

- **Problem:** Given a set of pairs of strings, is there a way to concatenate the strings from the pairs in a sequence such that the resulting two strings are equal?
- **Significance:** Emil Post introduced the PCP to prove the undecidability of this seemingly innocent problem. The proof involves a reduction from the Halting Problem.

3. Universal Turing Machine (UTM) Halting Problem:

3. Universal Turing Machine (UTM) Halting Problem:

- **Problem:** Given a description of a Universal Turing Machine U and an input w , determine whether U halts on w .
- **Significance:** This problem is often used in reductions to prove the undecidability of other problems, including the undecidability of the Halting Problem itself.

4. Emptiness Problem for Turing Machines:

- **Problem:** Given a description of a Turing machine M , is the language recognized by M empty?
- **Significance:** The undecidability of this problem is proven by a reduction from the Halting Problem.

5. Turing Machine Equivalence Problem:

- **Problem:** Given descriptions of two Turing machines M_1 and M_2 , do they recognize the same language?
- **Significance:** This problem is undecidable, and the proof typically involves reductions from the Halting Problem.