Module 5: Turing Machine

Definition: A Turing machine M is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where,

$Q \rightarrow$ is a finite nonempty set of states

$\Gamma \rightarrow$ is a finite nonempty set of tape symbols

$b \rightarrow$ is the blank

$\Sigma \rightarrow$ is a nonempty set of input symbol and is a subset of Γ and $b \notin \Sigma$

$\delta \rightarrow$ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R/W

$q_0 \in Q$ initial state

$F \subseteq Q$ is the set of final states.

Representation of Turing Machines

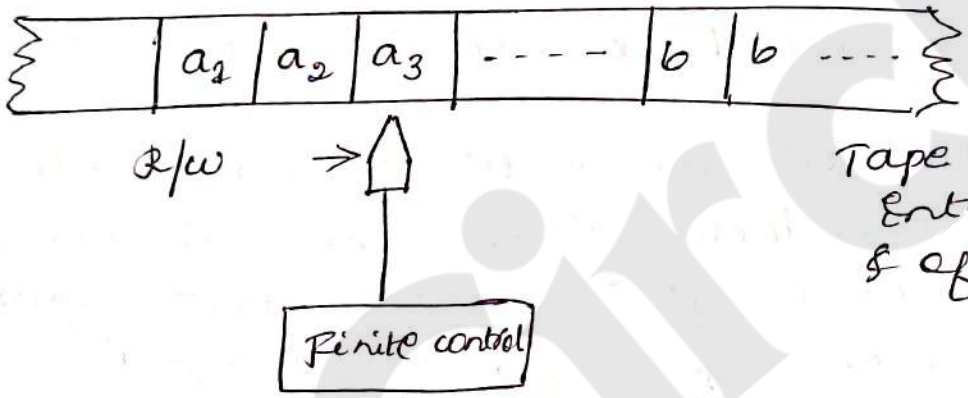
1. Transition diagram
2. Instantaneous descriptions using move relations.
3. Transition table

Imp 6 marks

Turing Machine Model

The turing machine can be thought of as finite control connected to a R/w (read/write) head.

- It has one tape which is divided into a number of cells. The block diagram of the basic model for the turing machine is given below.



→ Each cell can store only one symbol

→ The input to and the output from the finite state automation are affected by the R/w head which can examine one cell at a time.

In one move, the machine examines the present symbol under the R/w head on the tape and the present state of an automation to determine,

(i) A new symbol to be written on the tape in the cell under the R/w head

(ii) A motion of the R/w head along the tape either the head moves one cell left (L) or one cell (R).

(iii) the next state of the automation &
(iv) whether to halt or not

Component of Turing Machine

→ Input tape
→ R/w head
→ Finite control unit

Techniques for Turing Machine construction

1. Turing Machine with Stationary Head

suppose, we want to include the option that the head can continue to be in the same cell for some input symbol.
then we define (q, a) as (q', Y, S)
this means that the TM, on reading the input symbol a, changes the state to q and writes Y in the current cell in place of a and continues to remain in the same cell.

2. Storage in the State:

we can use a state to store a symbol as well so the state becomes a pair (q, a) where q is the state and a is the tape symbol stored in (q', a).

3. Multiple track Turing machine

In a multiple track TM, a single tape is assumed to be divided into several tracks. Now the tape alphabet is required to consists of k-tuples of tape symbols, k being number of tracks.

4. Subroutines

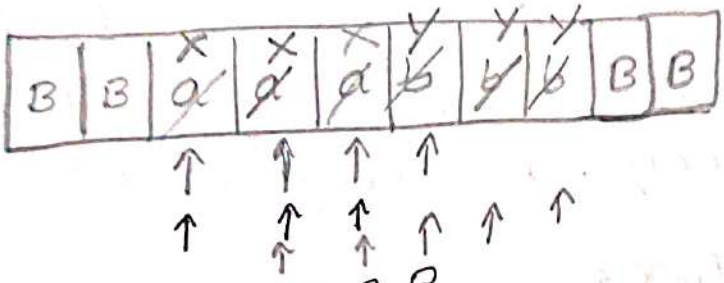
first a TM a program for the subroutine is written. This will have an initial state and a return state., After reaching the return state, there is a temporary halt for using a subroutine, new states are introduced, when there is a need for calling the subroutines moves are affected to enter the initial state for the subroutine. when the return state of the subroutine is reached. return to the main program of TM.

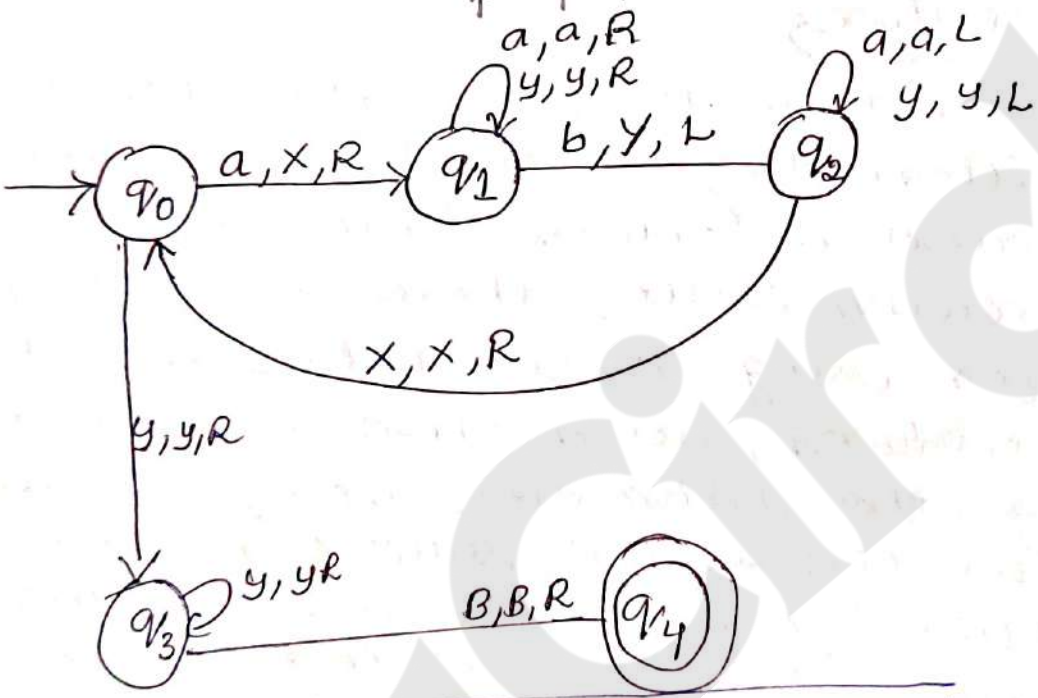
(1) Design a turing machine for

$L = \{ a^n b^n \mid n \geq 1 \}$

$n=2$

a a b b
X X Y Y





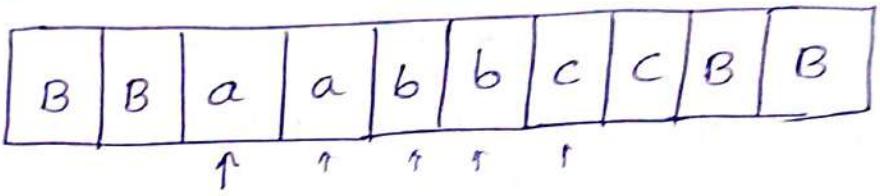
	a	b	X	Y	B
q_0	q_1, X, R			q_3, Y, R	
q_1	q_1, a, R	q_2, Y, L		q_1, Y, R	
q_2	q_2, a, L		q_0, X, R	q_2, Y, L	
q_3				q_3, Y, R	Halt
q_4					

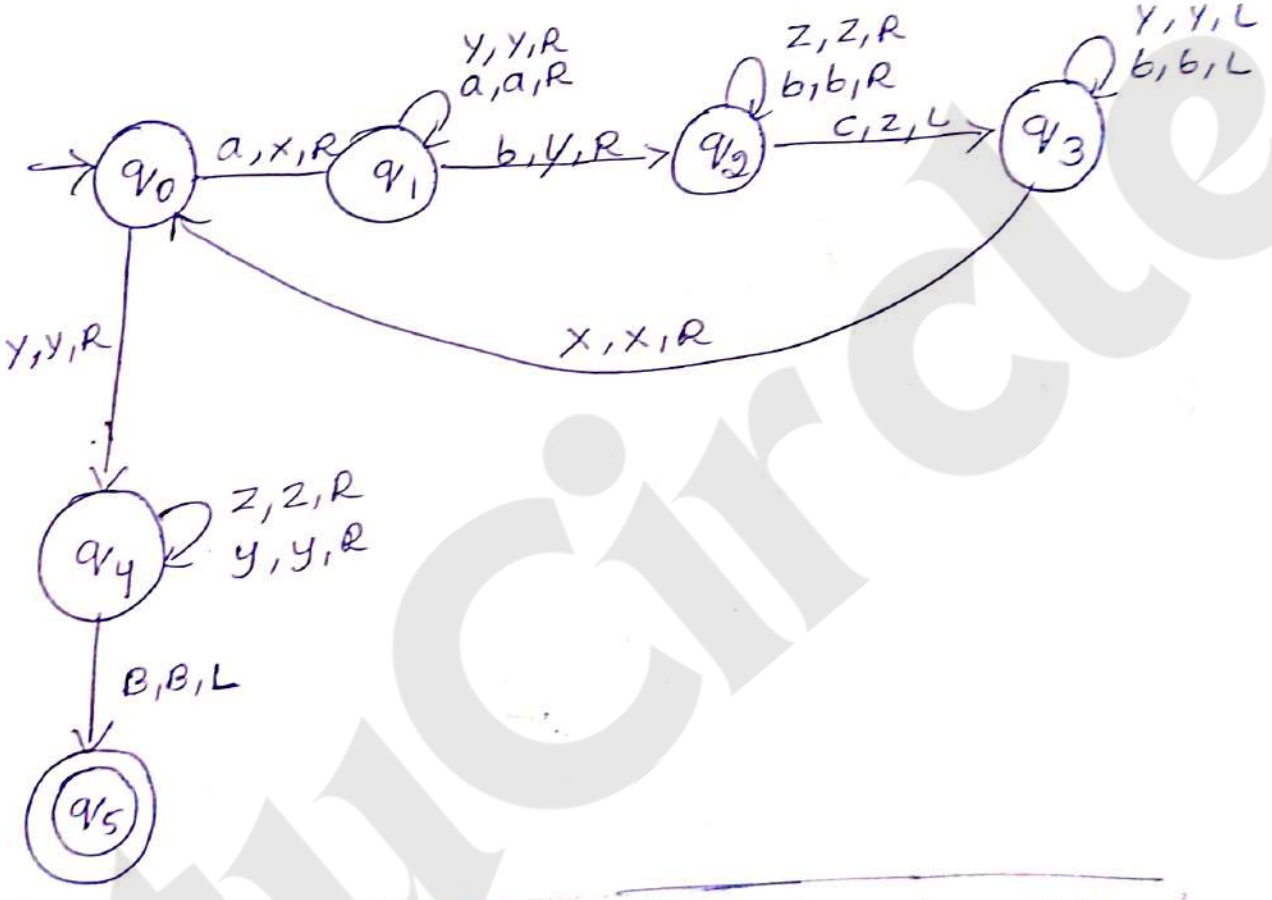
TM, $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

$Q = \{ q_0, q_1, q_2, q_3, q_4 \}$
$\Sigma = \{ a, b \}$
$\Gamma = \{ a, b, X, Y, B \}$
q_0 – initial

q_4 final state
B Blank symbol

② Design Turing Machine for $a^n b^n c^n$ $n \geq 1$





	a	b	c	X	Y	Z	B
q_0	q_1, X, R				q_4, Y, R		
q_1	q_1, a, R	q_2, Y, R			q_1, Y, R		
q_2		q_2, b, R	q_3, Z, L			q_2, Z, R	
q_3	q_3, a, L	q_3, b, L		q_0, X, R	q_3, Y, L	q_3, Z, L	
q_4					q_4, Y, R	q_4, Z, R	Halt
q_5							

(3) Design TM for $L = \{a^{2n} b^n \mid n \geq 1\}$

(4) Design TM for $L = \{a^n b^{2n} \mid n \geq 1\}$

(5) Design TM for even palindrome ww^R

(6) Design TM for odd palindrome wcw^R

(3) $L = \{a^{2n} b^n \mid n \geq$

Show the sequence ID for string aabbcc

$q_0aabbcc \vdash X q_1 abbcc$

$\vdash Xa q_1 bbcc$

$\vdash Xay q_2 bcc$

$\vdash Xayb q_2 cc$

$Xay q_3 b\cancel{q_3}c$

$\vdash Xay q_3 yb zc$

$\vdash X q_3 ayb zc$

$\vdash q_3 Xayb zc$

$\vdash X q_0 ayb zc$

$\vdash XX q_1 yb zc$

$\vdash XXY q_1 b zc$

$\vdash XXYY q_2 zc$

$\vdash XXYYz q_2 c$

$\vdash XXYY q_3 zz$

$\vdash XXY q_3 yzz$

$\vdash XX q_3 Yyzz$

$\vdash XX q_3 \cancel{X} Yyzz$

$\vdash XX q_0 Yyzz$

$\vdash XXY q_1 yzz$

$\vdash XXYY q_1 zz$

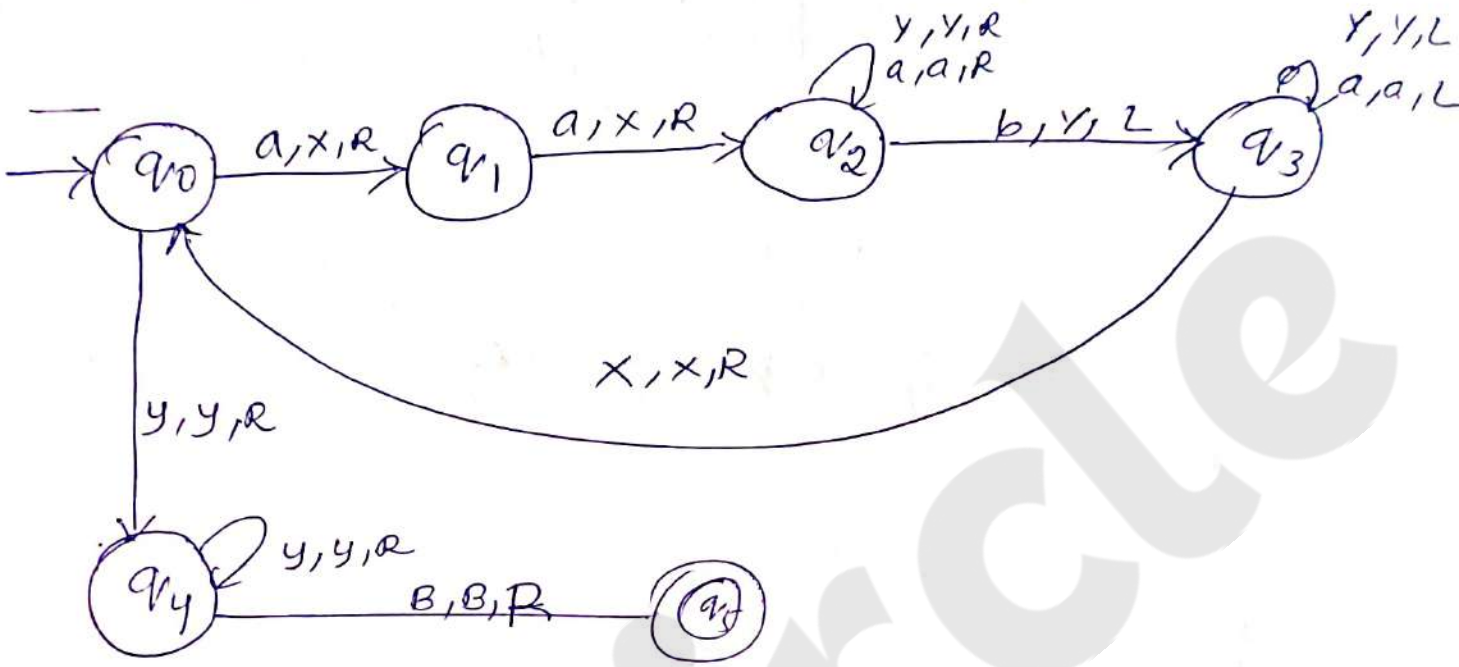
$\vdash XXYYz q_4 z$

$\vdash XXYYzz q_4 B$

$\vdash XXYYz \cancel{q_4} q_5 z$

③ $L = \{a^{2n} b^n \mid n \geq 1\}$ [For every 2a → 1b]

$L = \{aab, aaaabb, aaaaaabbb \ldots\}$



	a	b	X	Y	B
q_0	q_1, X, R			q_4, Y, R	-
q_1	q_2, X, R				
q_2	$q_2 a, R$	q_3, Y, L		q_2, Y, R	
q_3	q_3, a, L		q_0, X, R	q_3, Y, L	
q_4				q_4, Y, R	q_5, B, R
q_5					

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

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$\Sigma = \{a, b\}$

$\Gamma = \{a, b, X, Y, B\}$

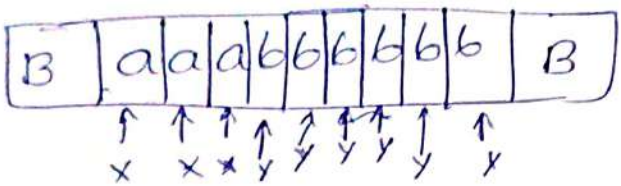
$q_0 \rightarrow$ initial state

q_5 – final state

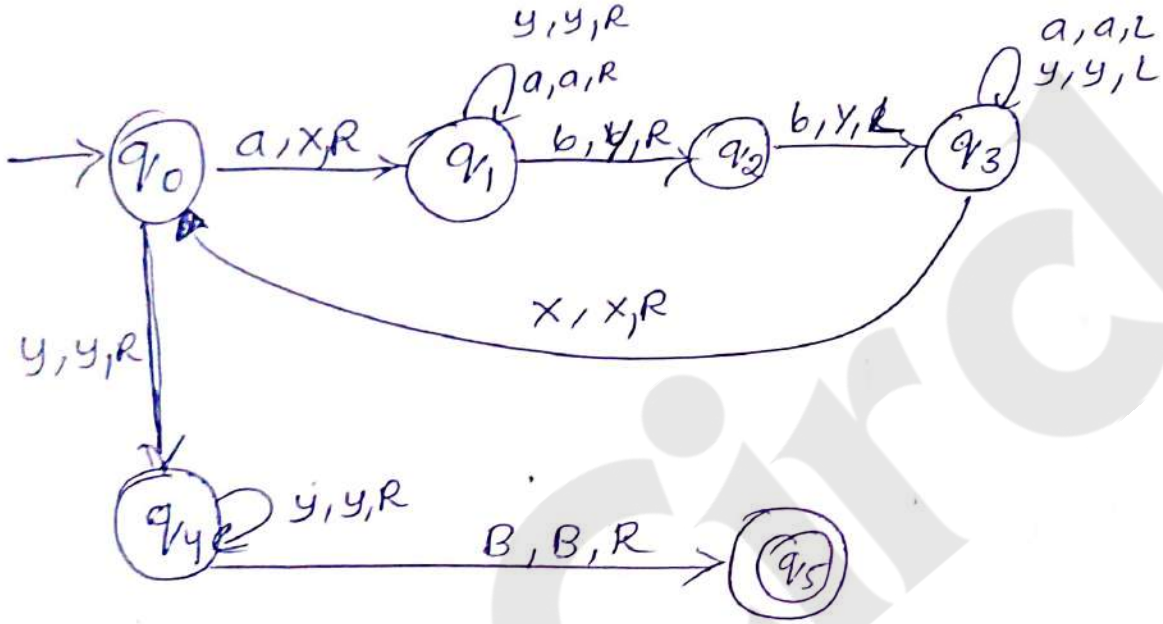
B – Blank symbol

(4) Design TM for $L = \{ a^n b^{2n} \mid n \geq 1 \}$

$L = \{ abb, aabbbb, aaabbbbbb \cdots \}$



[for every a → 2 b's]



	a	b	X	Y	B
q_0	q_1, X, R			q_4, Y, R	
q_1	q_1, a, R			q_1, Y, R	
q_2		q_3, Y, L			
q_3	q_3, a, L		q_0, X, R	q_3, Y, L	
q_4				q_4, Y, R	q_5, B, R (Halt)
final state q_5					

Transition function δ

$\delta(q_0, a) = (q_1, X, R)$

$\delta(q_0, Y) = (q_4, Y, R)$

$\delta(q_1, a) = (q_1, a, R)$

$\delta(q_1, Y) = (q_1, Y, R)$

$\delta(q_2, b) = (q_3, Y, L)$

$\delta(q_3, a) = (q_3, a, L)$

$\delta(q_3, X) = (q_0, X, R)$

$\delta(q_3, Y) = (q_3, Y, L)$

$\delta(q_4, Y) = (q_4, Y, R)$

$\delta(q_4, B) = (q_5, B, R)$ //

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$\Sigma = \{a, b\}$

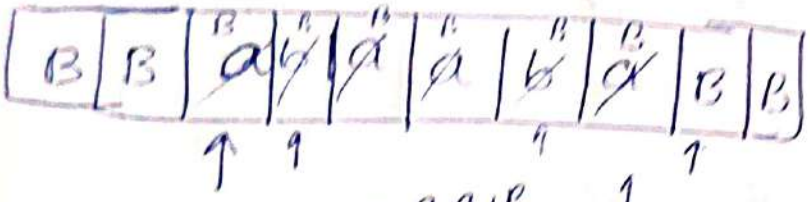
$T = \{a, b, X, Y, B\}$

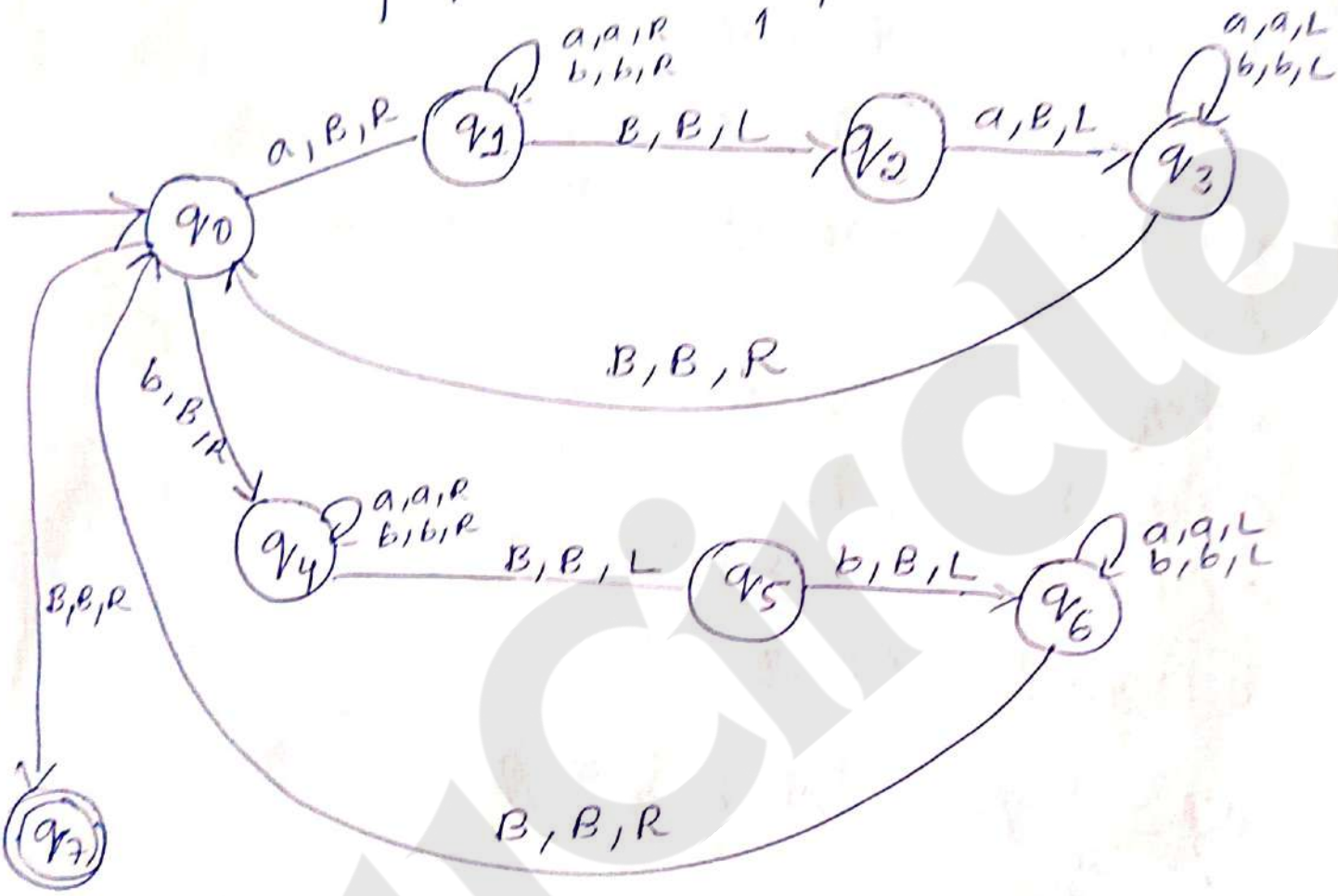
q_0 - initial state

q_5 - final state

B - Blank symbol //

(5) Design TM for even palindrome ww^R





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

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$

q_0 initial state

q_7 final state

$\Gamma = \{a, b, B\}$

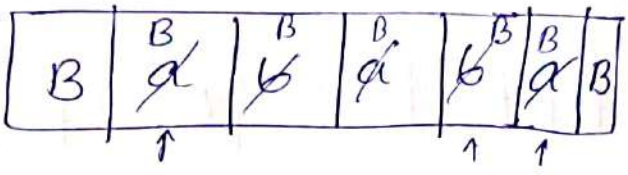
$\Sigma = \{a, b\}$

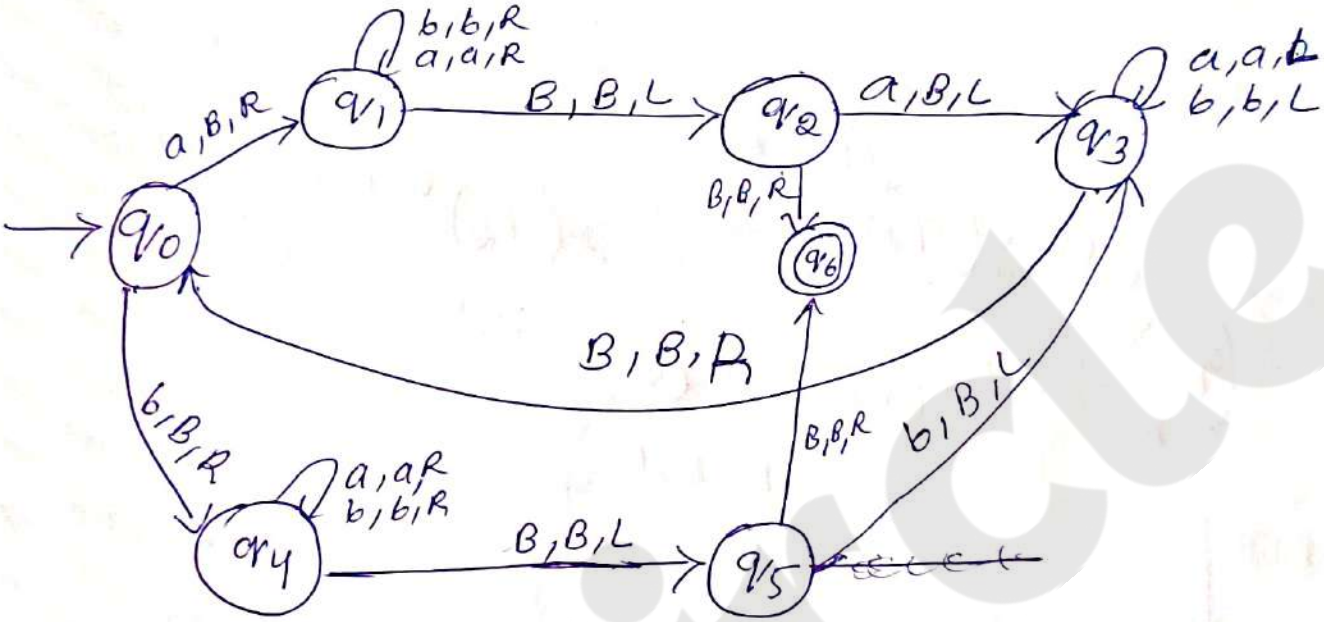
B - Blank Symbol

write transition table refer transition diagram

(6) Design TM for odd palindrome

↳ $wawᴿ$





$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$

q_6 final state

q_0 initial state

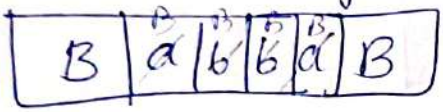
$\Sigma = \{a, b, B\}$

$T = \{a, b, B\}$

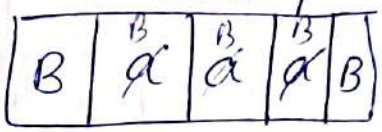
B = Blank symbol,

(7) Design TM for all palindrome (even & odd)

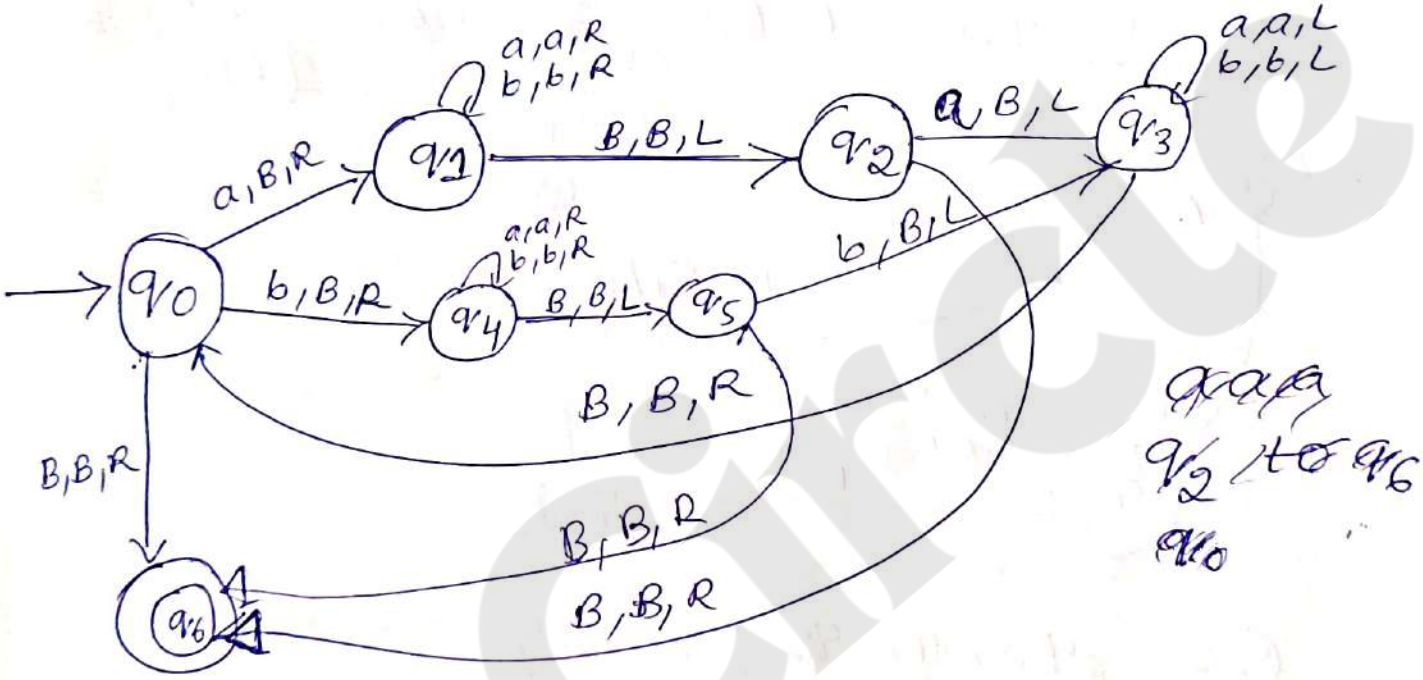
even palindrome



odd palindrome



a a a
a b a



q q q q
q_2 to q_6
q_0

M $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$

q_6 → final state

q_0 → initial state

$\Gamma = \{a, b, B\}$

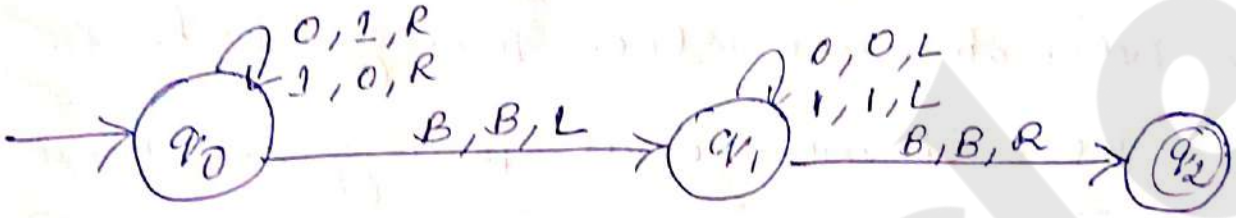
$\Sigma = \{a, b\}$

B Blank symbol

⑧ Design TM for 1' complement

1's complement means 1 to 0 & 0 to 1

B ↓ ↓ ↓ ↓
B 1 0 1 0 B
↳ ↳ ↳ ↳ ↓
B 0 1 0 1 B
↑ ↑ ↑ ↑



$Q = \{q_0\ q_1\ q_2\}$

$\Sigma = \{0, 1\}$

$\Gamma = \{0, 1, B\}$

q_0 initial state

q_2 final state //

Different types of Turing Machine
OR
Variant of Turing Machine.

1) The multi-dimensional Turing Machine.
2) Multi-tape turing Machine
3) Non deterministic Turing Machine.
4) Multidemensional turing machine

In this case, Turing machine has multidimensional tape with a read write head. It uses basic concept of Turing Machine with modification of transition function as shown in figure

$$\delta : (Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\}$$

where U & D specifies movement of the read-write head up & down.

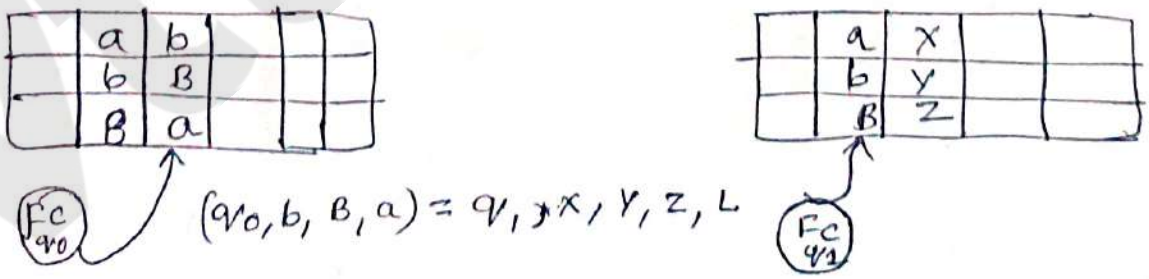


Fig: Multi-demensional Turing Machine

(ii) Multitape Turing Machine:

Turing Machine has multi tape, means standard TM with more number of tapes.

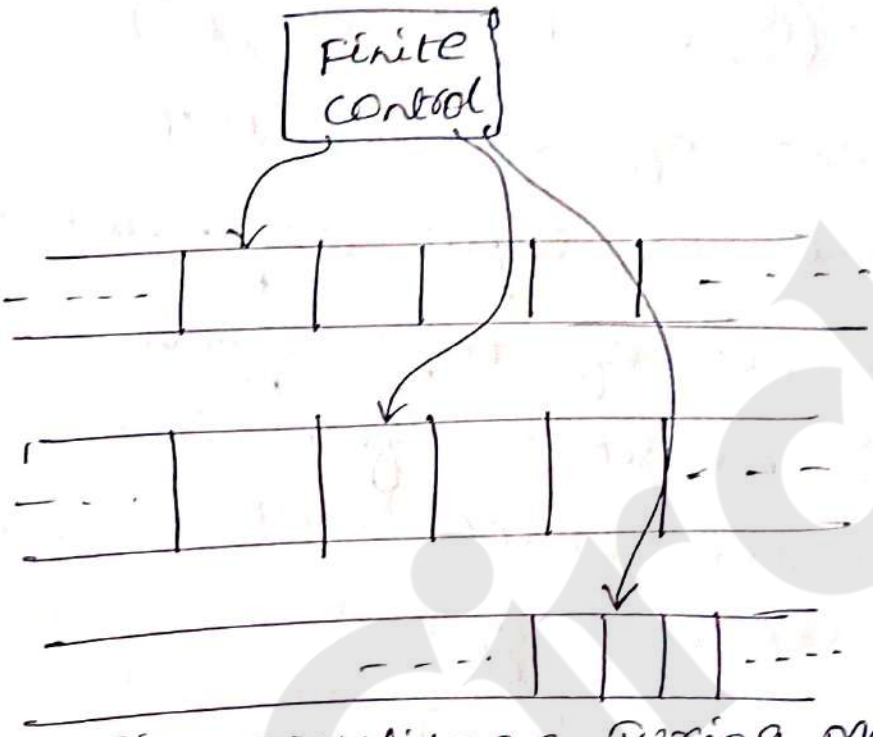


Fig: Multitape Turing Machine.

The component of multitape TM are * Finite control * multiple R/W heads.

⇒ Each tape divided into cells which can hold any symbol from the given alphabet. to start with TM should be in start state q0.

⇒ If the r/w head is pointing to tape1 moves towards right, the R/W head pointing to tape2 & tape3 many move towards right or left depending on the transition

⇒ The move of the multitape TM depends on the current state & scanned symbol by each of the tape heads.

(iii) Nondeterministic Turing Machine

Definition: The non deterministic turing machine is a 7 tuple

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

Q is set of finite state
Σ is set of input alphabet
Γ is set of tape symbols
δ is transition function
q_0 is the start state
B is Blank symbol
$F \subseteq Q$ is set of final state

It is clear from the definition of δ that for each state q and tape symbol X, $\delta(q_0, X)$ is a set of triples

$$\{(q_1, X_1, D), (q_2, X_2, D), (q_3, X_3, D) \ldots (q_i, X_i, D)\}$$

where
* i → finite integer
* D → direction with Left or right
* the machine can choose any of the triples in the next move.

Application of Turing Machine

1. Theory of computation:
   * Turing machines help define what it means for a problem to computable
   * They are used to explore the limitation of computation
2. Decidability and undecidability:
   * Turing machines are instrumental in proving whether certain problems are decidable
3. complexity theory:
   * They are used to analyze the computational complexity of algorithms by defining complexity classes such as P, NP, and PSPACE.
4. Algorithm design and Analysis:
   * Turing machine provide a blueprint for designing algorithms for complex problems.
5. cryptography:
   * Turing machine are used to study encryption and decryption process
6. formal verification & model checking:
   * Turing machine model systems for formal verification to ensure correctness of s/w and h/w systems.

7. Quantum computing:

+ The classical Turing machine concept serves as a foundation for the development of quantum computation models

8. Bioinformatics and DNA computing:

The Turing machine model has been adapted to study biological computing and DNA based computation

9. Artificial Intelligence:

Turing machine contribute to understanding learning algorithm and the theoretical capabilities of AI.

10. programming language design:

Turing machine influence the development of programming languages by formalizing concepts such as loops, conditional statements and recursion.

MODULE 5

Introduction to Turing Machines: Table of Contents

Introduction to Quantum Computers and Turing Machines

Turing machines are theoretical models of computation that provide a fundamental framework for understanding what problems computers can and cannot solve. They consist of a tape, a head, a set of states, and a transition function. Despite their simplicity, they are powerful enough to simulate any algorithm that can be performed by a modern computer. The study of Turing machines led to the discovery of **undecidable problems**, like the **Halting Problem**, which demonstrates the inherent limits of computation.

Quantum computers, on the other hand, are a new type of computing device that harnesses the principles of quantum mechanics to perform computations. Unlike classical computers, which store information as bits (0 or 1), quantum computers use **qubits**, which can exist in a superposition of states, representing both 0 and 1 simultaneously. This allows them to perform certain computations exponentially faster than classical computers.

Quantum computers can solve problems more efficiently, but they cannot solve problems that are undecidable for Turing machines.

Problems That Computers Cannot Solve:

Despite the incredible advancements in computing, there exist certain problems that computers fundamentally cannot solve. These problems are termed **undecidable**

problems, signifying the **absence of any algorithm capable of providing a correct solution for all possible inputs**.

- One of the most renowned undecidable problems is the **Halting Problem**. This problem poses the question of whether a given computer program, when executed, will eventually halt (terminate) or continue running indefinitely. A proof by contradiction establishes the undecidability of the Halting Problem. This proof demonstrates that **no Turing Machine, a theoretical model encompassing all possible computations, can solve the Halting Problem for every program and input combination.**
- Another prominent example of an undecidable problem is the **Post Correspondence Problem (PCP).** PCP involves two sequences of strings, and the task is to determine if a specific sequence of indices exists, such that concatenating the corresponding strings from both sequences results in identical strings. The **undecidability of PCP is established through a reduction from the Halting Problem**. This reduction implies that if PCP were solvable, it could be leveraged to solve the Halting Problem, creating a logical contradiction, as the Halting Problem is already proven to be undecidable.

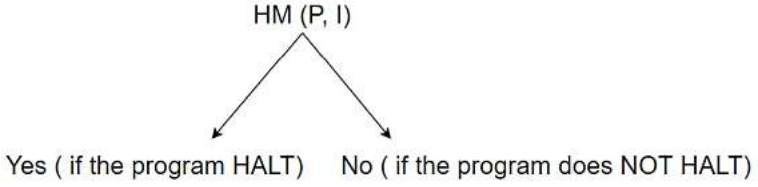
The existence of undecidable problems underscores the inherent limitations of computation, demonstrating that certain problems remain beyond the reach of any algorithm or computer, regardless of advancements in technology.

The Halting Problem

The **Halting Problem** is a fundamental problem in computer science that explores the limits of what computers can decide. It asks whether, given a computer program and an input, it is possible to determine in advance whether the program will eventually halt (stop running) or run forever on that input.

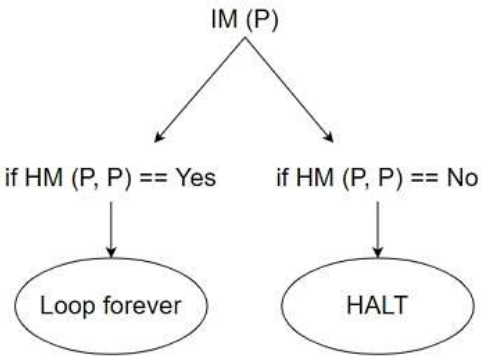
The Halting Problem is undecidable, meaning **no algorithm exists that can solve it for all possible programs and inputs**. This undecidability is proven using a proof by contradiction:

1. **Assume a Halting Machine (HM) exists that can determine whether any program P halts on input I.** HM(P, I) would output YES if P halts on I and NO if it runs forever.

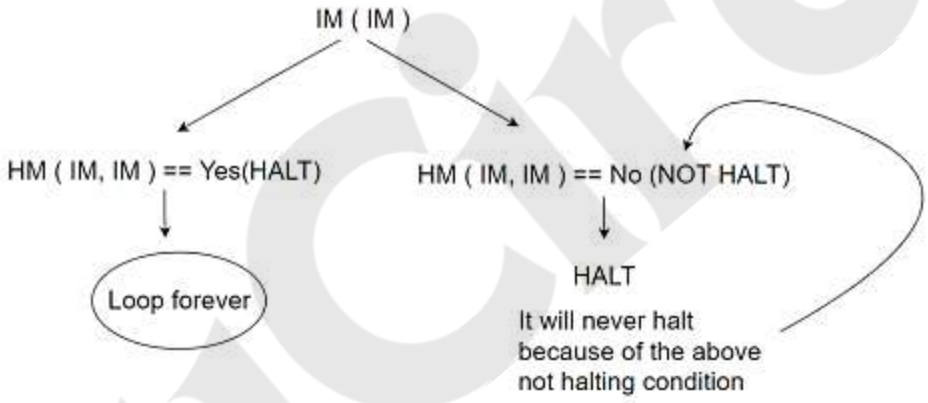


2. **Construct an Inverted Halting Machine (IM) that takes a program P as input.** IM behaves as follows:
   - If HM(P, P) returns YES (meaning P halts on input P), then IM loops forever.

- If HM(P, P) returns NO (meaning P runs forever on input P), then IM halts.



3. **Consider what happens when IM is given itself as input (IM(IM)).**
   - If IM(IM) halts, then according to IM's definition, HM(IM, IM) must have returned NO (meaning IM runs forever on input IM). This is a contradiction.
   - If IM(IM) runs forever, then HM(IM, IM) must have returned YES (meaning IM halts on input IM). This is also a contradiction.



This contradiction arises because we assumed the existence of a Halting Machine (HM). Therefore, **such a machine cannot exist, and the Halting Problem is undecidable**.

The implications of the Halting Problem's undecidability are significant. It means that **no universal algorithm can be created to determine whether any given program will halt or run forever**. This has profound implications for software verification and automated program analysis.

The Halting Problem also serves as a cornerstone for proving the undecidability of other problems. For example, the undecidability of the Post Correspondence Problem is proven by showing that if we could solve PCP, we could also solve the Halting Problem.

The Halting Problem is an example of a **recursively enumerable language that is not recursive**. A recursively enumerable language is one for which there exists a Turing Machine that can list out all the strings in the language. However, it may not be possible to determine whether a given string is *not* in the language. In the case of the

Halting Problem, we can list out all the programs that halt on given inputs, but we cannot always definitively determine if a program will run forever.

Post Correspondence Problem (PCP)

The **Post Correspondence Problem (PCP)** is a decision problem in theoretical computer science. It is **undecidable**, meaning there is no general algorithm that can solve the problem for all possible inputs.

Problem Definition:

The PCP involves two sequences of strings, denoted as A and B:

- $A = (A_1, A_2, \ldots, A_n)$
- $B = (B_1, B_2, \ldots, B_n)$

The goal is to determine if there exists a sequence of indices $(i_1, i_2, \ldots, i_m)$ such that the concatenation of the corresponding strings from both sequences results in identical strings:

- $A[i_1] + A[i_2] + \ldots + A[i_m] = B[i_1] + B[i_2] + \ldots + B[i_m]$

Example:

Consider the following sequences:

- $A = (ab, bc)$
- $B = (a, b)$

We need to find a sequence of indices such that:

- $A + A = 'ab' + 'bc' = 'abbc'$
- $B + B = 'a' + 'b' = 'ab'$

In this case, no solution exists because the concatenation of strings from A ($'abbc'$) does not match the concatenation from B ($'ab'$).

Undecidability of PCP:

The undecidability of PCP can be proven through a reduction from the Halting Problem. This means that if we could solve PCP, we could also solve the Halting Problem, which is known to be undecidable.

- The Halting Problem asks whether a given Turing machine halts on a given input.
- If we could solve PCP, we could construct an instance of PCP that encodes the behaviour of a Turing machine on a specific input.

- A solution to this PCP instance would correspond to the Turing machine halting on that input.

Since the Halting Problem is undecidable, solving PCP would imply solving an undecidable problem, which is impossible. Therefore, PCP is also undecidable.

Applications and Importance:

Despite its undecidability, PCP has significant theoretical implications:

- **Undecidability Proofs:** PCP is often used as a tool to prove the undecidability of other problems.
- **Understanding Limits of Computation:** PCP helps us understand the boundaries of what can be computed and the complexity of certain problems.

PCP is a classic example of an undecidable problem in the theory of computation, highlighting the inherent limitations of algorithms and the existence of problems that cannot be solved by any computer program.

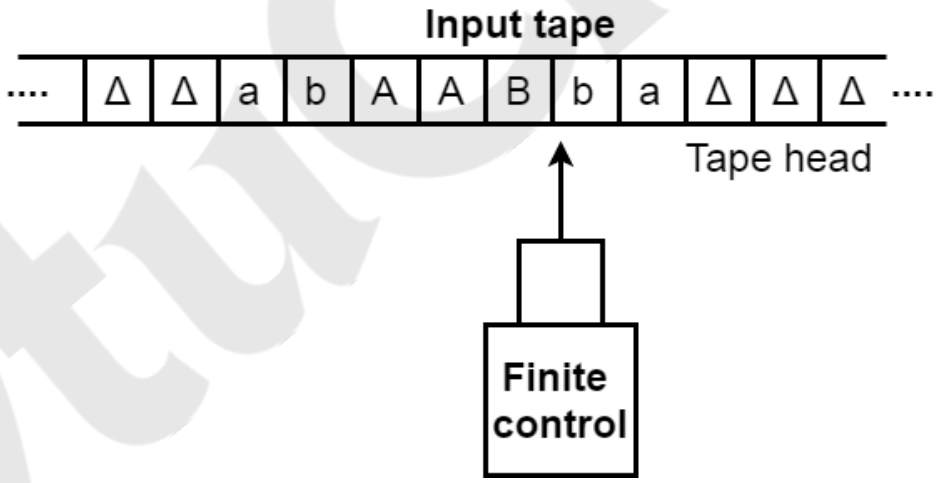
Turing Machine

A **Turing machine** is a theoretical model of computation that serves as a foundation for understanding the capabilities and limitations of computers. It was introduced by Alan Turing in 1936 and consists of the following components:

- **Tape:** An infinite strip of cells, each capable of holding a symbol from a finite alphabet.
- **Head:** A read/write head that can move left or right along the tape, reading and writing symbols.
- **State:** A finite set of states that the machine can be in, representing its current computational stage.
- **Transition Function:** A set of rules that dictate how the machine behaves based on its current state and the symbol read by the head.

The **transition function** defines the actions of the Turing machine. It specifies, for each combination of current state and symbol read, the following:

- The new symbol to be written on the tape.
- The direction the head should move (left or right).
- The new state the machine should enter.



The formal notation we shall use for a Turing machine (TM) is similar to that used for finite automata or PDA's. We describe a TM by the 7-tuple

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

whose components have the following meanings:

Q: The finite set of states of the finite control.

Σ: The finite set of input symbols.

F: The complete set of tape symbols; Σ is always a subset of F.

δ: The transition function. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

1. p is the next state, in Q.
2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
3. D is a direction, either L or R, standing for "left" or "right," respectively, and telling us the direction in which the head moves.

q_0: The start state, a member of Q, in which the finite control is found initially.

B: The blank symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

F: The set of final or accepting states, a subset of Q.

Techniques for construction of Turing machine:

1. **Turing Machine with Stationary Head**
   - **Description:**
     In this type of Turing machine, the head does not move after a transition. Instead, the tape symbol is updated, and the machine transitions to a new state.
   - **Technique:**
     - Typically achieved by designing a special state transition that keeps the head stationary, often interpreted as a "read-and-write-only" step.
     - Instead of moving left or right, the head remains on the current tape cell while performing state transitions.
   - **Applications:**
     - Useful in tasks where frequent overwriting of the same cell is needed before moving to the next one.
2. **Storage in the State**
   - **Description:**
     A Turing machine's state can be designed to encode additional information about the computation, effectively allowing the state to act as temporary storage.

- **Technique:**
  - Augment the state set to include compound states that encode specific information.
  - For example, instead of having a simple state q1, you might use q1X to encode that the machine is in state q1 and has seen a specific symbol X.
- **Applications:**
  - Simplifies tape usage for some computations by encoding more information in the states.
  - Commonly used in TMs for pattern matching or for storing small amounts of context.

3. **Multiple Track Turing Machine**
   - **Description:**
     A multi-track TM uses a single tape divided into multiple parallel tracks. Each track can hold a separate sequence of symbols, but the head reads/writes on all tracks simultaneously.
   - **Technique:**
     - Define the tape alphabet as a Cartesian product of symbols from individual track alphabets.
     - Transitions operate on tuples representing the current symbols on each track.
   - **Advantages:**
     - Efficient for simulating multiple tapes or keeping auxiliary information (e.g., counters, markers) in parallel.
   - **Applications:**
     - Useful for managing multiple data streams or when auxiliary calculations need to be performed alongside the primary computation.
4. **Subroutines**
   - **Description:**
     Subroutines allow modular design by breaking down the overall task into smaller, reusable parts. Each subroutine is essentially a small TM designed to perform a specific task.
   - **Technique:**
     - Define separate state sets for each subroutine, ensuring no overlap between states of different subroutines.
     - Use "call" and "return" states to transition between subroutines.
     - Subroutines can handle tasks like shifting, copying, or checking for specific patterns.
   - **Applications:**
     - Simplifies complex machine designs by promoting reusability.
     - Common in TMs that implement complex algorithms or simulate higher-level computations.

Working Principle:

A Turing machine starts in an initial state with an input string written on its tape. It then repeatedly applies the transition function, reading symbols, writing symbols, moving the head, and transitioning between states. This process continues until the machine reaches a halting state, signifying the end of computation.

Applications of Turing Machines:

Turing machines (TMs) are a foundational concept in computer science and have applications in various domains, even though they are primarily theoretical. Their significance lies in their ability to model computation and provide insights into the limits and capabilities of algorithms.

1. Formalizing Computability

- **Defining Computable Functions:** Turing machines formalize the concept of algorithms and computation by defining what functions can be computed using finite steps.
- **Church-Turing Thesis:** The Turing machine is used to state that any function computable by an algorithm can also be computed by a Turing machine, making it a universal model of computation.

2. Algorithm Analysis

- **Complexity Theory:** Turing machines help define complexity classes like **P**, **NP**, **EXPSPACE**, and others, forming the basis of computational complexity.
- **Efficiency and Optimization:** They are used to study the time and space required for computations, offering theoretical insights into algorithm performance.

3. Language Recognition and Automata Theory

- **Context-Free and Regular Languages:** TMs extend the capabilities of simpler automata like finite automata and pushdown automata, enabling the recognition of more complex languages.
- **Decidability of Problems:** They are used to determine whether a language is decidable (can be recognized and verified by a machine) or undecidable.

4. Proofs of Decidability and Undecidability

- **Undecidable Problems:** TMs are used to prove that certain problems (e.g., the **Halting Problem**) cannot be solved algorithmically. Example: Determining whether a Turing machine halts for a given input.
- **Reduction Techniques:** Problems are reduced to Turing machine computations to prove their decidability or undecidability.

5. Simulation of Real-World Computation

- **Simulating Modern Computers:** A Turing machine can simulate the operations of any modern computer (given sufficient time and tape space), showing the equivalence of computational power.
- **Emulation of Algorithms:** Algorithms in programming languages can be converted into Turing machine representations for theoretical analysis.

6. Artificial Intelligence and Machine Learning

- **Theoretical Foundations:** Turing machines provide a baseline for designing learning algorithms and reasoning about their computational limits.
- **Universal Turing Machine (UTM):** The concept of the UTM, which can simulate any other TM, is akin to modern AI systems being capable of general-purpose problem-solving.

7. Cryptography

- **Complexity-Based Security:** Many cryptographic protocols rely on problems that are computationally hard for TMs (e.g., factoring large numbers).
- **Proofs of Security:** TMs are used to prove the infeasibility of breaking certain cryptographic systems under reasonable assumptions.

8. Compiler Design

- **Parsing and Language Translation:** TMs are used to model parsers that analyze and translate programming languages, forming a basis for understanding compiler construction.
- **Semantic Analysis:** Some compiler-related problems (e.g., type checking) are modelled using variants of TMs.

9. Cognitive Science and Philosophy

- **Nature of Thought and Computation:** Turing machines are used in discussions about whether the human brain functions like a computational machine.
- **Turing Test and AI:** The idea of machine intelligence and the limits of computational reasoning draw heavily on the concept of TMs.

10. Real-World Applications

While Turing machines are not directly used in practical systems, their theoretical insights influence the design and analysis of:

- **Operating Systems:** Scheduling and resource management algorithms are analyzed using TM concepts.

- **Database Systems:** Query processing and optimization are based on decidable languages modelled by TMs.
- **Search Engines:** Indexing and pattern-matching tasks can be studied through language recognition and TM simulations.

Variants of Turing Machines and Applications

1. **Multi-Tape Turing Machines**: Used for analyzing algorithms that require multiple streams of data processing.
2. **Non-Deterministic Turing Machines (NDTMs)**: Central to complexity theory, especially in defining **NP** problems.
3. **Universal Turing Machines (UTMs)**: Theoretical basis for the concept of modern general-purpose computers.

Limitations of Turing Machines:

While Turing machines are powerful theoretical tools, they have limitations that distinguish them from practical computers:

- **Infinite Tape:** Turing machines assume an infinite tape, which is unrealistic in physical computers.
- **Discrete Time Steps:** Turing machines operate in discrete time steps, whereas physical computers operate continuously.
- **Simple Operations:** Turing machines only perform basic operations like reading, writing, and moving the head. Modern computers have much richer instruction sets.

Despite these limitations, Turing machines remain a crucial concept in computer science, providing a framework for understanding the fundamental principles of computation and the limits of what computers can achieve.

Language accepted by Turing machine

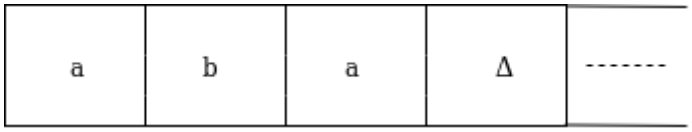
The Turing machine accepts all the language even though they are recursively enumerable. Recursive means repeating the same set of rules for any number of times and enumerable means a list of elements. The TM also accepts the computable functions, such as addition, multiplication, subtraction, division, power function, and many more.

Example:

Construct a Turing machine which accepts the language of aba over $\sum = \{a, b\}$.

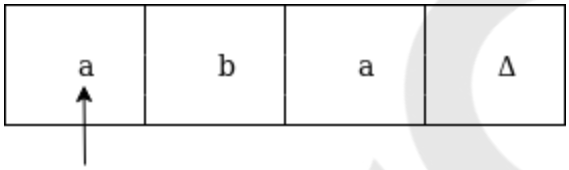
Solution:

We will assume that on input tape the string 'aba' is placed like this:

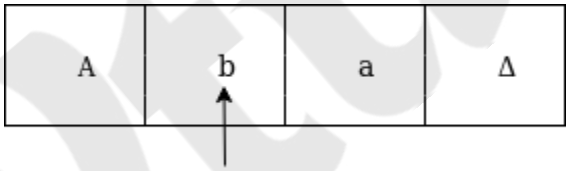


The tape head will read out the sequence up to the Δ characters (where Δ $is\ blank\ state$. If the tape head is readout 'aba' string then TM will halt after reading Δ.

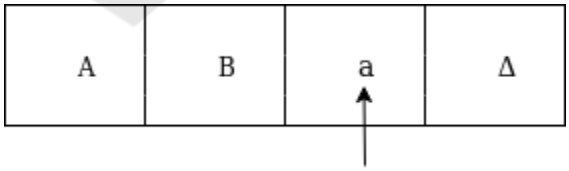
Now, we will see how this Turing machine will work for aba. Initially, state is q0 and head points to a as:



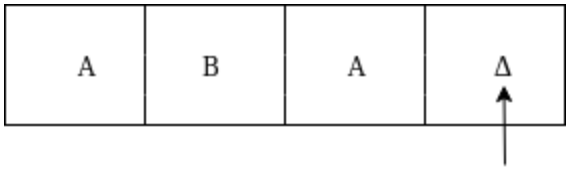
The move will be $\delta(q0, a) = \delta(q1, A, R)$ which means it will go to state $q1$, replaced a by A and head will move to right as:



The move will be $\delta(q1, b) = \delta(q2, B, R)$ which means it will go to state $q2$, replaced b by B and head will move to right as:



The move will be $\delta(q2, a) = \delta(q3, A, R)$ which means it will go to state q3, replaced a by A and head will move to right as:

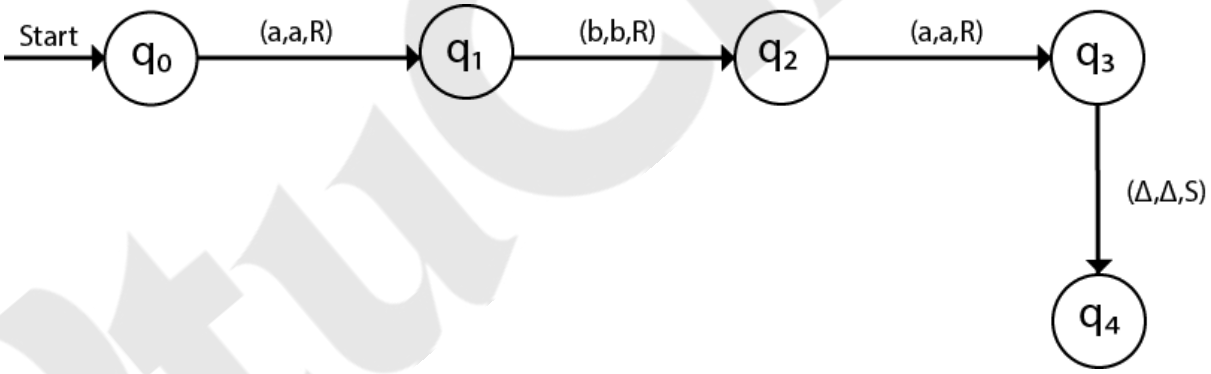


The move $\delta(q3, \Delta) = (q4, \Delta, S)$ which means it will go to state q4 which is the HALT state and HALT state is always an accept state for any TM.

The same TM can be represented by Transition Table:

States	***a***	***b***	Δ
q0	$(q1, A, R)$	–	–
q1	–	$(q2, B, R)$	–
q2	$(q3, A, R)$	–	–
q3	–	–	$(q4, \Delta, S)$
q4	–	–	–

The same TM can be represented by Transition Diagram:



Programming Techniques for Turing Machines

While Turing machines are primarily theoretical constructs, understanding how to "program" them provides valuable insights into the fundamentals of computation. Programming a Turing machine involves designing its transition function to solve a specific problem.

Designing the Transition Function

The transition function is at the heart of a Turing Machine. It dictates how the machine behaves based on its current state and the symbol it reads from the tape. It can be thought of as a set of rules of the form:

(Current State, Read Symbol) → (New State, Write Symbol, Head Movement).

- **Current State and Read Symbol**: These represent the input to the transition function. The machine examines its current state and the symbol under its head on the tape.
- **New State, Write Symbol, Head Movement**: This is the output of the transition function. It specifies how the machine should change its state, what symbol to write on the tape, and whether to move the head left or right (or stay in the same position).

To design a transition function for a specific task, you need to carefully consider how each combination of state and read symbol should be handled to ultimately achieve the desired computation.

Manipulating the Tape and Head

The tape is the Turing Machine's working memory, and the head is its tool for interacting with that memory.

Here are some common techniques for manipulating the tape and head:

- **Marking Cells:** You can use special symbols from your alphabet to mark specific cells on the tape. This helps the machine remember locations it needs to revisit, store intermediate results, or mark the beginning and end of data sections.
- **Shifting Data:** Implementing algorithms often requires moving data around on the tape. You might need to shift an entire block of symbols to the left or right to make space for new data or to align data for comparison.
- **Simulating Multiple Tapes:** While a basic Turing Machine has one tape, you can simulate multiple tapes using a single tape. You can partition the tape into sections and use special symbols to delimit these sections, effectively treating each section as a separate tape.

State Transitions

The finite set of states in a Turing machine represents its memory and control flow mechanisms.

- **States as Memory:** Each state can represent a piece of information or a stage in the computation. The transition from one state to another is triggered by the read symbol and encodes logic.
- **States for Control Flow:** States can be used to control the sequence of actions performed by the machine. You can design states to represent different phases of an algorithm (initialization, processing, output, etc.) and use transitions to move between these phases based on the symbols read and the intended logic of the algorithm.

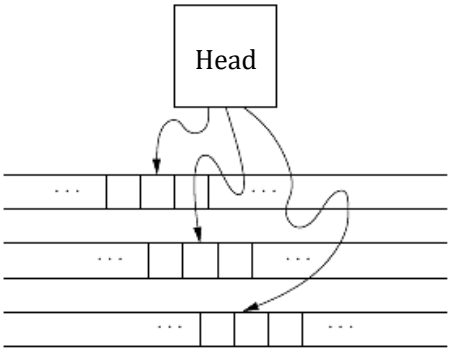
Example: Adding Two Unary Numbers

Imagine you want to design a Turing machine to add two unary numbers. The numbers are represented as strings of 1s separated by a 0. For instance, 11011 represents 2 + 2.

1. **States:** You might have states like "Start," "Find First Number," "Find Second Number," "Add," and "Halt."
2. **Tape Manipulation:** You'd need to move the head to locate the two numbers, replace the separating 0 with a 1, and possibly clean up extra symbols.
3. **Transitions:** Transitions would dictate how the machine changes states based on the symbol read. For example, if in the "Find First Number" state and the head reads a 1, it would stay in the same state and move right. If it reads a 0, it would transition to the "Find Second Number" state.

Multi-Tape Turing Machines

A multi-tape Turing machine is a variant of the basic Turing machine that possesses **multiple tapes**, each equipped with its own independent read/write head. This enhancement significantly expands the machine's capabilities while maintaining its theoretical equivalence to the single-tape model.



Key features and advantages:

- **Structure:** Instead of a single tape, a multi-tape Turing machine has *k* tapes, where *k* is a positive integer. Each tape is infinite in length, just like in the single-tape model. The machine has a separate head for each tape, allowing it to read and write on multiple tapes simultaneously.
- **Transition Function:** The transition function is modified to accommodate the multiple tapes. It now takes as input the current state and the symbols read by *all* the heads. The output of the function specifies:
    - The new state
    - The symbols to be written on each tape
    - The movement direction of each head (left, right, or stationary).
- **Enhanced Efficiency:** Multi-tape Turing machines can perform certain tasks much more efficiently than their single-tape counterparts. For example, copying a string of symbols can be done in linear time on a multi-tape machine, while it requires quadratic time on a single-tape machine.
- **Simulating Multiple Data Structures:** The multiple tapes can be used to represent different data structures or hold different parts of a computation. For instance, one tape could store the input data, another could hold intermediate results, and another could be used for output.
- **Equivalence to Single-Tape Machines:** Despite their increased power, multi-tape Turing machines are computationally equivalent to single-tape Turing machines. Any computation that can be performed on a multi-tape machine can also be performed on a single-tape machine, though it might require more steps and a more complex program. This equivalence reinforces the fundamental nature of the Turing machine model, showing that adding more tapes doesn't change the class of problems that can be solved.

Example: Palindrome Recognition

Consider the task of determining whether a given string is a palindrome (reads the same backward as forward). A two-tape Turing machine can solve this efficiently:

1. **Initialization:** The input string is written on the first tape.
2. **Copying:** Using both heads, the machine copies the input string from the first tape to the second tape in reverse order.
3. **Comparison:** The heads are moved to the beginning of each tape. The machine compares the symbols under each head, moving both heads to the right simultaneously. If all symbols match until the end of the tapes is reached, the string is a palindrome.

Non-Deterministic Turing Machine (NDTM)

A **Non-Deterministic Turing Machine (NDTM)** is a theoretical model of computation that extends the concept of a **Deterministic Turing Machine (DTM)** by allowing multiple possible actions at each step. Unlike DTMs, where the behavior is entirely predictable based on the current state and input, NDTMs can "choose" between different transitions.

Definition: An **NDTM** is formally described as a 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q0, q_{accept}, q_{reject})$$

Where:

- Q: Finite set of states.
- Σ: Input alphabet (does not include the blank symbol).
- Γ: Tape alphabet ($\Sigma \subseteq \Gamma$ and includes a blank symbol).
- δ: Transition function, which for an NDTM is a relation:

  $$\delta: Q \times \Gamma \to P(Q \times \Gamma \times \{L, R, S\})$$

  This means that for a given state and tape symbol, there may be multiple possible next states, tape symbols, and head movements.
- q_0: Start state.
- q_{accept}: Accept state.
- q_{reject}: Reject state ($q_{reject} \neq q_{accept}$).

Key Features

1. **Non-Determinism**:
   - At any step, the machine can choose among several possible transitions. It can "guess" the correct sequence of choices to solve a problem.
   - This is often visualized as a tree of computations where each branch represents a possible sequence of transitions.
2. **Acceptance Condition**:
   - An input is **accepted** if at least one computation branch leads to the accept state (q_{accept}).
   - If all branches lead to the reject state (q_{reject}), the input is rejected.
3. **Parallelism (Theoretical)**:
   - Conceptually, the NDTM can explore all computation branches simultaneously. This does not correspond to any physical machine but is useful for theoretical analysis.

Comparison with Deterministic Turing Machine (DTM)

Feature	*Deterministic TM*	*Non-Deterministic TM*
Transition Function	Single next state for each input	Multiple possible next states
Nature of Execution	Sequential and predictable	Parallel exploration of branches
Power of Computation	Equivalent (both are Turing complete)	Equivalent (NDTMs do not solve problems that DTMs cannot)
Time Complexity	Solves problems in O(f(n)) time	Can "guess" solutions in O(1) time on some branches, leading to NP problems

Undecidability: A Language That Is Not Recursively Enumerable:

Undecidability refers to the inherent impossibility of solving certain problems algorithmically for all possible inputs. In the context of Turing machines and formal languages, a problem is undecidable if there is no Turing machine that can always halt and correctly determine whether a given input belongs to a specific language.

Decidable Problems:

A problem is **decidable** if an algorithm exists that can always solve it correctly in a finite amount of time. This means a corresponding Turing Machine can be constructed that will halt on every input with a definite "yes" or "no" answer. These problems are also referred to as **Turing Decidable** problems.

- **Recursive languages** fall under the category of decidable problems. A Turing machine for a recursive language will always halt, accepting strings belonging to the language and rejecting those that do not. This characteristic equates recursive languages with decidable languages.

Semi-Decidable Problems:

Semi-decidable problems are characterized by a Turing machine that halts and accepts strings belonging to the corresponding language. However, for strings not in the language, the Turing machine might halt and reject or enter an infinite loop. This ambiguity in halting behavior distinguishes semi-decidable problems from decidable ones.

- Semi-decidable problems are also known as **Turing Recognizable** problems because the Turing machine can recognize and accept strings within the language, but it might not definitively reject all strings outside the language.

- **Recursively Enumerable languages** are synonymous with semi-decidable problems. They share the same halting behavior: the Turing machine might not halt for all inputs.

Undecidable Problems:

Undecidable problems are those for which no algorithm can be constructed that can answer the problem correctly in finite time. These problems may be partially decidable, but they will never be decidable. There will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer.

- A prominent example of an undecidable problem is the **Halting Problem**. The Halting Problem seeks to determine whether a given program, represented by a Turing machine, will eventually halt or run forever for a given input. While it's possible to create a Turing machine that halts for some program-input combinations, there is no universal Turing machine that can solve the Halting Problem for all possible programs and inputs.

Recursive Language (Decidable Language)	TM will always halt
Recursive Enumerable Language (Partially Decidable Language)	TM will halt sometimes and may not halt sometimes
Undecidable	NO TM for that language

Relationship and Examples:

- Every decidable problem is inherently semi-decidable, but the converse is not necessarily true. In essence, all recursive languages are recursively enumerable, but not all recursively enumerable languages are recursive.
- The **Halting Problem** exemplifies a problem that is semi-decidable but not decidable. While a Turing machine can be designed to halt and accept Turing Machine and input combinations that do halt, it cannot definitively decide for all combinations whether they will halt or run infinitely. This inherent undecidability of the Halting Problem is a fundamental concept in computer science.

Recursive and Recursively Enumerable Languages

To understand undecidable languages, it's crucial to distinguish between recursive and recursively enumerable languages:

- **Recursive Languages:** A language is **recursive** if there exists a Turing machine that:
    - Halts and accepts any string that belongs to the language.

    - Halts and rejects any string that does not belong to the language. These languages are also called **decidable** languages. The Turing machine acts as a decider, always providing a definitive answer (accept or reject) for any given input.

- **Recursively Enumerable (RE) Languages:** A language is **recursively enumerable** if there exists a Turing machine that:
    - Halts and accepts any string that belongs to the language.
    - May halt and reject or run forever for strings not in the language.
    - These languages are also known as **partially decidable** or **Turing recognizable** languages. The Turing machine can recognize strings belonging to the language, but it might not halt for strings not in the language.

A Language That Is Not Recursively Enumerable

If a language is not recursively enumerable, it means there's no Turing machine that can even list out all the strings in the language, let alone decide membership for all inputs. This signifies a higher level of undecidability compared to RE languages, where at least enumeration is possible.

Key Points to Remember

- Every recursive language is recursively enumerable, but not vice versa.
- **The Halting Problem is a classic example of a language that is recursively enumerable but not recursive.** It's possible to list out (enumerate) Turing machine and input pairs that halt, but it's impossible to design a Turing machine that always halts and correctly decides whether any given Turing machine will halt on a specific input.

The concept of undecidability is fundamental in computer science, as it sets limits on what can be computed algorithmically. Understanding the distinction between recursive, recursively enumerable, and non-RE languages helps us grasp the complexities and boundaries of computation.

Languages Beyond Turing Machine Capabilities

Turing Machines, as powerful as they are, have limitations. They cannot solve **undecidable problems** or recognize **languages that are not recursively enumerable**.

- **Undecidable problems** are those for which no algorithm can guarantee a solution for every input in a finite time. The **Halting Problem**, which tries to

predict whether any program will halt or run forever, is a classic example of an undecidable problem.

- **Recursively Enumerable languages**, on the other hand, can be recognized by a Turing Machine. This means the machine can accept strings that belong to the language. However, for strings that *don't* belong, the machine might either halt and reject them or run forever. The inability to guarantee a halting decision for all inputs is the defining characteristic of recursively enumerable languages.

Languages beyond Turing Machine capabilities would be those that are **not recursively enumerable**. This means there is **no Turing Machine** that can be constructed to recognize strings belonging to these languages. These languages are inherently more complex and lie outside the realm of problems solvable by computational models like Turing Machines.

It's important to remember:

- While Turing Machines are powerful models of computation, they have limitations.
- The existence of undecidable problems and non-recursively enumerable languages highlights the boundaries of what can be computed.

Therefore, understanding these limitations provides a deeper understanding of the theory of computation and the inherent complexities within the field of computer science.