- **Turing Machines:**
  - Problems that computer cannot solve
  - The Turing Machine
    - Notation for the Turing Machine
    - Instantaneous Descriptions for Turing Machines
    - Transition diagrams for Turing Machines
    - The language of a Turing Machine
    - Turing Machines and Halting
  - Programming Techniques for Turing Machine
    - Storage in the State
    - Multiple tracks
    - Subroutines
  - Extensions to the Basic Turing Machine
    - Multi tape Turing machines
    - Non-Deterministic Turing Machines
- **Un-decidability:**
  - Definition of Recursively Enumerable Language
  - Universal Turing Machines (UTM)
  - Halting Problem
  - Post Correspondence Problem
  - Church Hypothesis

1. Problems that computer can not solve

- Programs that print “Hello world”
- The hypothetical “Hello World” tester
- Reducing one problem to another

Programs that print “Hello world”

- It is easy to see that the following C program

```
main()
{
printf(‘‘hello, world\n’’);
}
```

- prints hello, world and terminates.

Other programs that also prints Hello World!

- Given an input n, it prints hello, world only if the equation $x^n + y^n = z^n$ has a solution
- where x, y, and z are integers.
- We know now a days that it will print hello, world for n = 2, and loop forever for n > 2.

– The problem described alternatively using symbols:

Is there a program H that could examine any program P and input I for P, and tell whether P, run with I as its input, would print hello, world? (A program *H* means *an algorithm* in concept here.)

– The answer is: undecidable! That is, there exists no such program *H*. We will prove this by contradiction.

The hypothetical “Hello World” tester

- **8.1.2 Hypothetical “Hello, World” Tester**
  - 1st step: assume H exists in the following form:

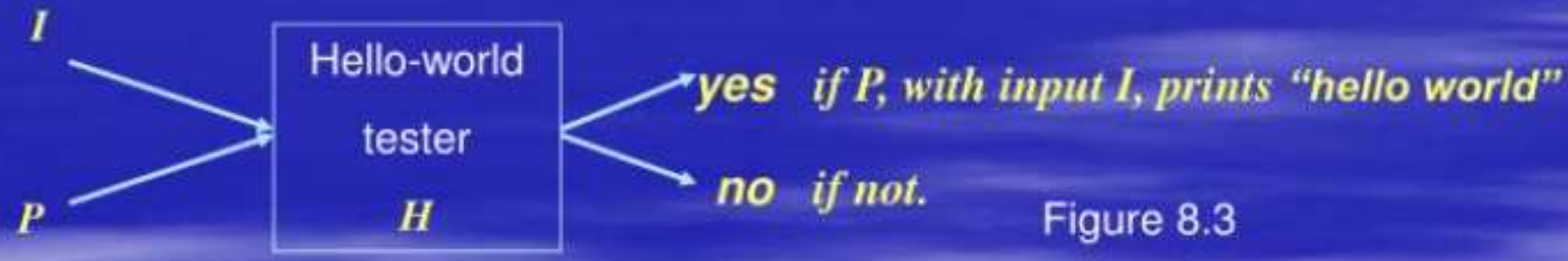


Figure 8.3

  - 2nd step: transform H to another form H_2 in simple ways which can be done by C programs
  - 3rd step: prove H_2 not existing. So H also not existing.

– 2nd step:

- (1) transform H to H_1 in the following way -



Figure 8.4

- (2) transform H_1 to H_2 in the following way –



Figure 8.5

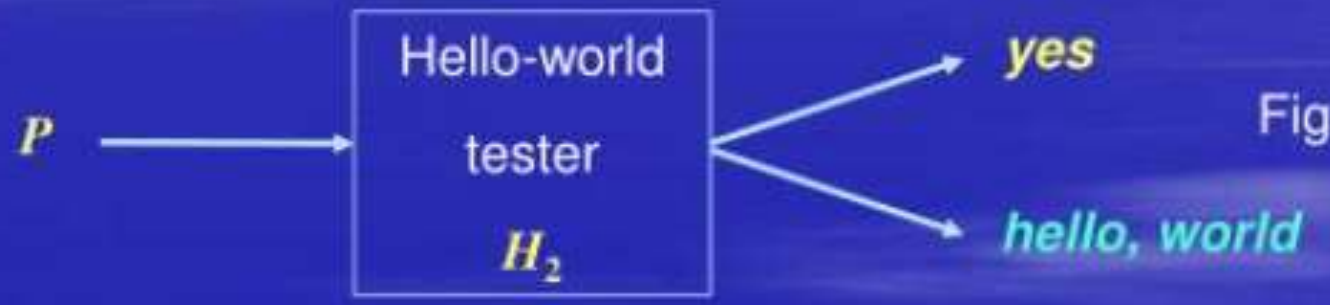


Figure 8.5

- The function of H_2 is:

  given any program P as input,

  if P prints hello world as first output, then H_2 makes output yes;

  if P does not prints hello world as first output, then H_2 prints hello world.

– 3rd step:

- Prove H_2 does not exist as follows –
  - Let P for H_2 in Fig. 8.5 (last figure) be H_2 itself, as follows:

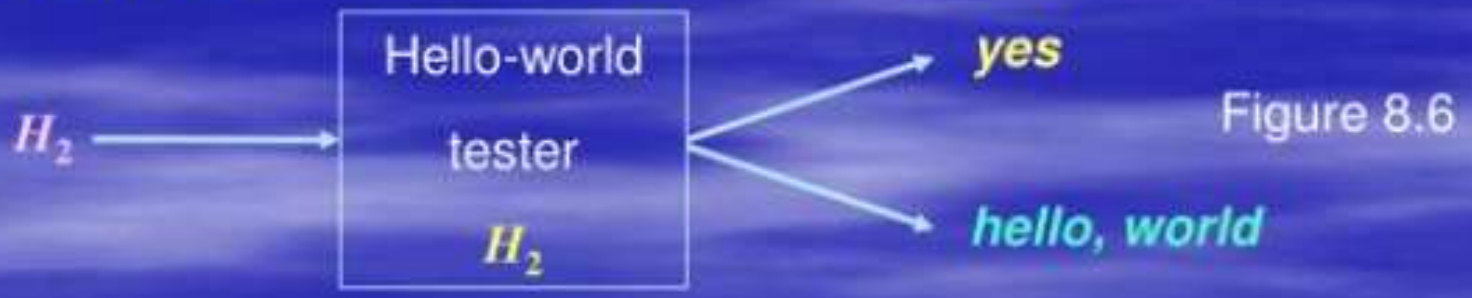


Figure 8.6

(H_2 Takes H_2 as input to itself)

– 3rd step (cont'd):

- Prove H_2 does not exist by contradiction as follows (cont'd) –



Figure 8.6

- Now,

(1) if the box H_2, given itself as input, makes output *yes*, then it means that

the input H_2, given itself as input, prints hello world as first output.

But this is contradictory because we just suppose that H_2, given itself as input, makes output *yes*.

– 3rd step (cont'd):

- Prove H_2 does not exist as follows (cont'd) –



Figure 8.6

- The above contradiction means the other alternative must be true (since it must be one or the other), that is ---

(2) the box H_2, given itself as input, prints ***hello, world***. This then means that

such H_2, when taken as input H_2 to the box H_2(itself), will make the box H_2 to make output ***yes***.

Contradiction again because we just say that the box H_2, given itself as input, prints ***hello, world***.

– 3rd step (cont'd):

- Prove H_2 does not exist as follows (cont'd) –

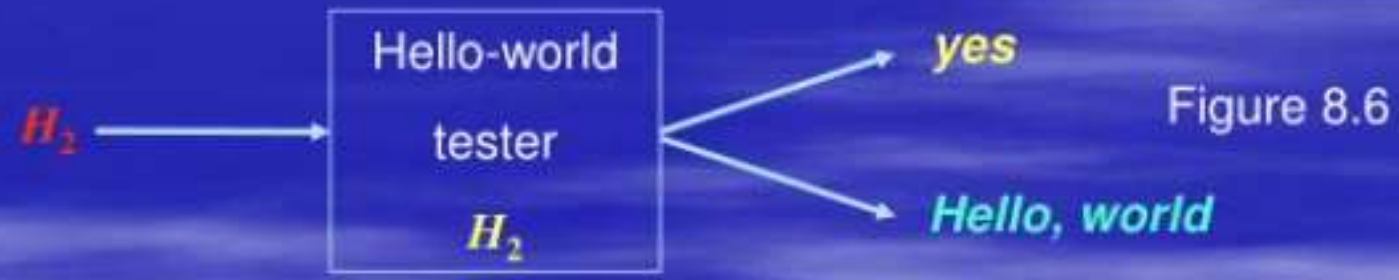


Figure 8.6

- Since **both** cases lead to contradiction, we conclude that the assumption that H_2 exists is wrong by the principle of contradiction for proof.

– 3rd step (cont'd):

- H_2 does not exist

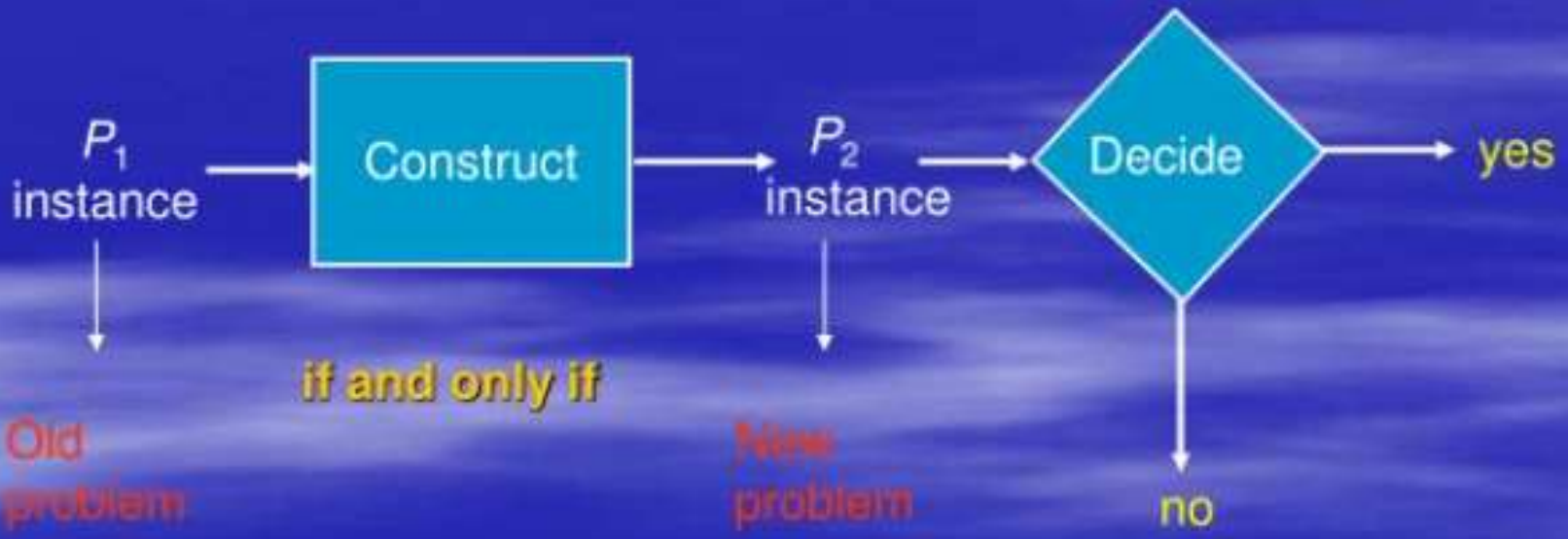
  $\Rightarrow H_1$ does not exist (otherwise, H_2 must exist)

  $\Rightarrow H$ does not exist (otherwise, H_1 must exist)

– The above self-contradiction technique, similar to the diagonalization technique (to be introduced later), was used by Alan Turing for proving undecidable problems.

- **8.1.3 Reducing One Problem to Another**
  - Now we have an undecidable problem, which can be used to prove other undecidable problems by a technique of "**problem reduction**."
  - That is, if we know P_1 is undecidable, then we may *reduce P_1 to a new problem P_2*, so that we may prove P_2 undecidable by contradiction in the following way:
    - *If P_2 is decidable, then P_1 is decidable.*
    - *But P_1 is known undecidable. So, contradiction!*
    - *Consequently, P_2 is undecidable.*

– An illustration of this idea is:



– **Example 8.1**

We want to prove a new problem P_2 (called *calls-foo problem*):

"does program Q, given input y, ever call function foo?"

to be undecidable.

Solution: *regard Q as* P_1;

reduce P_1: the hello-world problem to P_2: the calls-foo problem in the following way:

Solution: reduce P_1 to P_2 in the following way:

- If Q has a function called foo, *rename it* and all calls to that function $\Rightarrow$ a new program Q_1 doing the same as Q. (去除碰巧同名)
- Add to Q_1 a function foo doing nothing & *not* called $\Rightarrow$ a new Q_2
- Modify Q_2 to remember the first 12 characters that it prints, storing them in a global array A $\Rightarrow Q_3$
- Modify Q_3 so that whenever it executes any output statement, it checks A to see if it has written 12 characters or more, and if so, whether *hello, world* are the first characters. In that case, call the new function foo $\Rightarrow R$ with input $z = y$.

Solution (cont'd): Now,

(1) if Q with input y prints hello, world as its first output, then R will call foo;

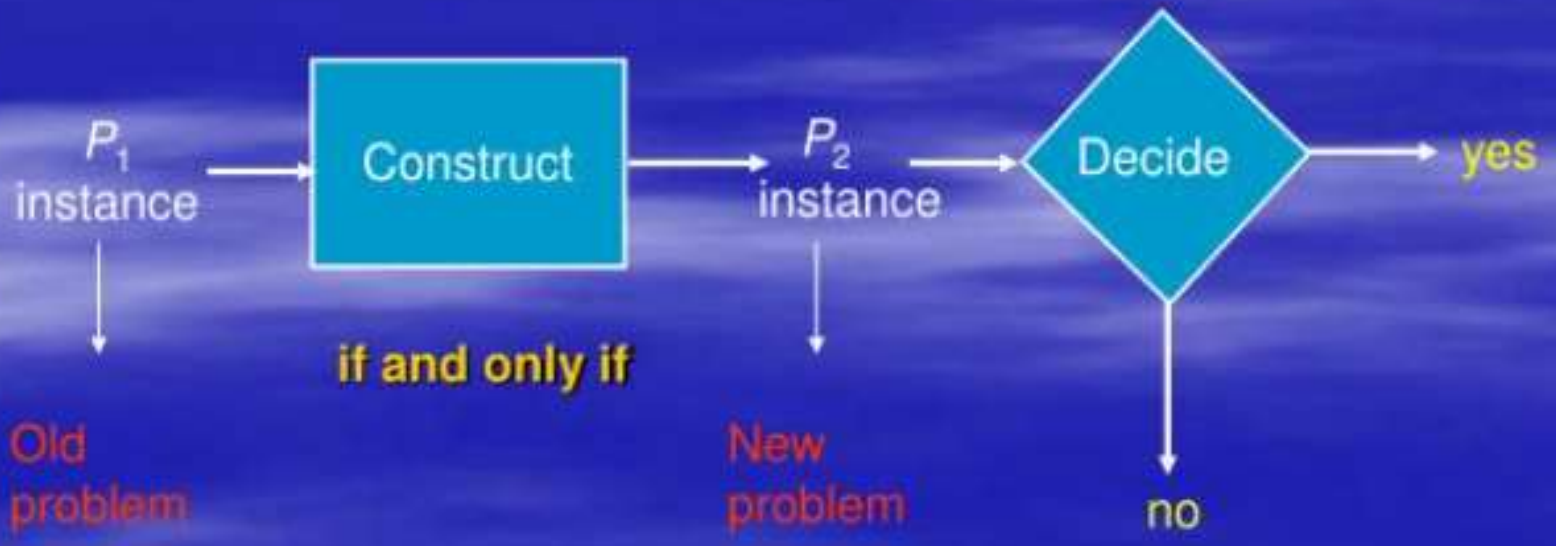
(2) if Q with input y does not print hello, world, then R will never call foo.

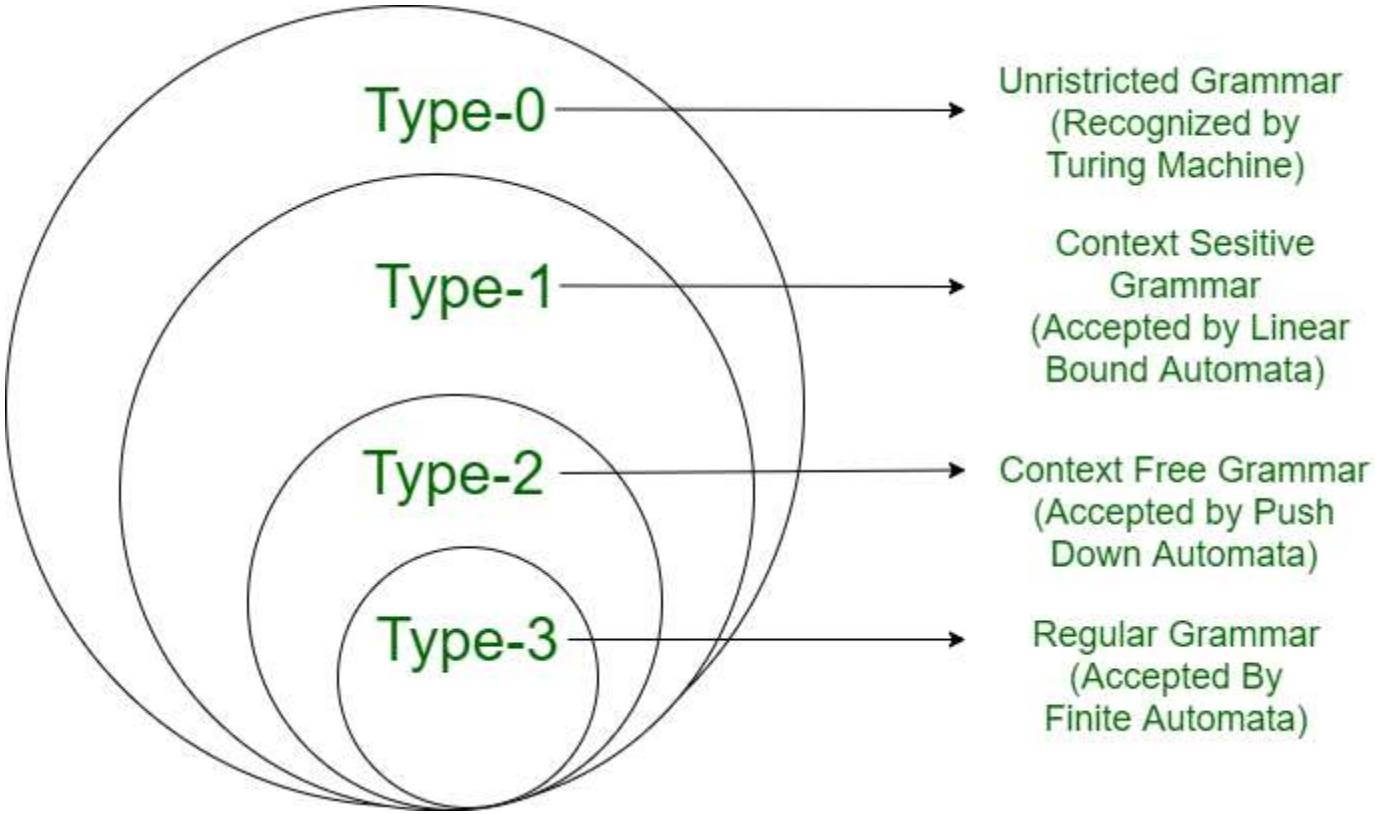
(3) That is, R, with input $z = y$, calls foo **if and only if** Q, with input y, prints hello, world.

(4) So, if we can decide whether R, with input z, calls foo, then we can decide whether Q, with input y, prints hello, world.

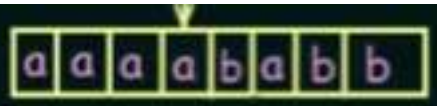
(5) But the latter is impossible, so the former is impossible.

– The above example illustrates how to reduce a problem to another.
– An illustration of this idea is:



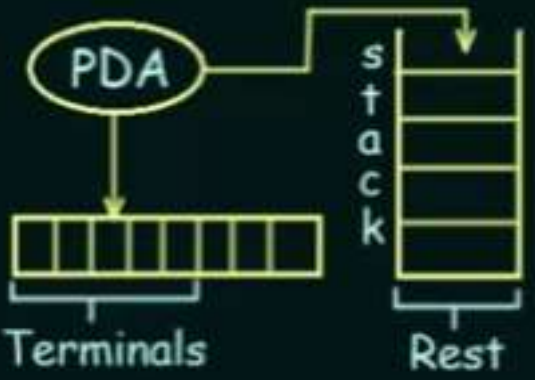

Type-0
Type-1
Type-2
Type-3
Unristricted Grammar
(Recognized by
Turing Machine)
Context Sesitive
Grammar
(Accepted by Linear
Bound Automata)
Context Free Grammar
(Accepted by Push
Down Automata)
Regular Grammar
(Accepted By
Finite Automata)

FSM: The Input String



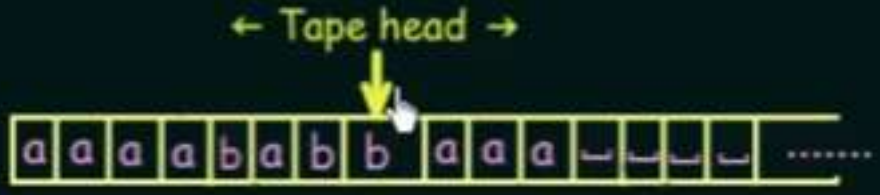
PDA: -> The Input String

-> A Stack



TURING MACHINE:

-> A Tape

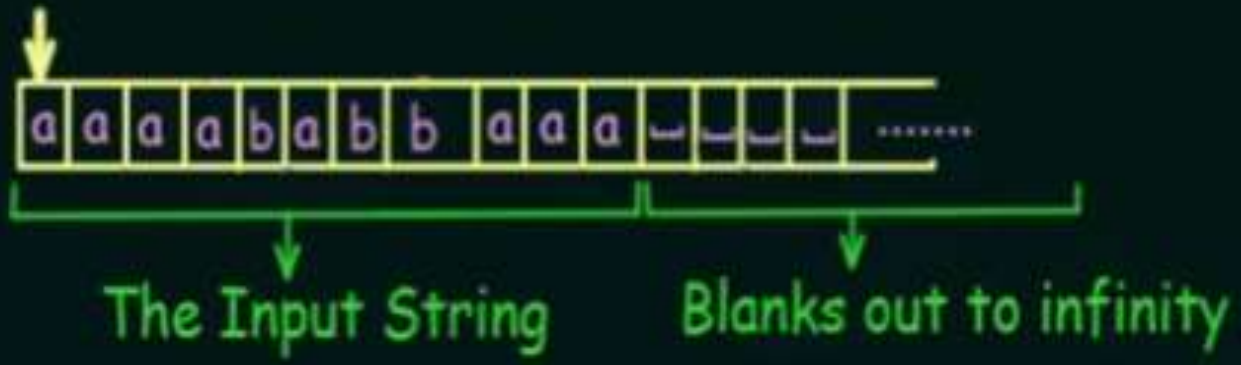


Tape Alphabets: $\Sigma = \{ 0,1,a,b,x,Z_0 \}$

The Blank ⌴ is a special symbol. $\quad ⌴ \notin \Sigma$

The blank is a special symbol used the fill the infinite tape

Initial Configuration:



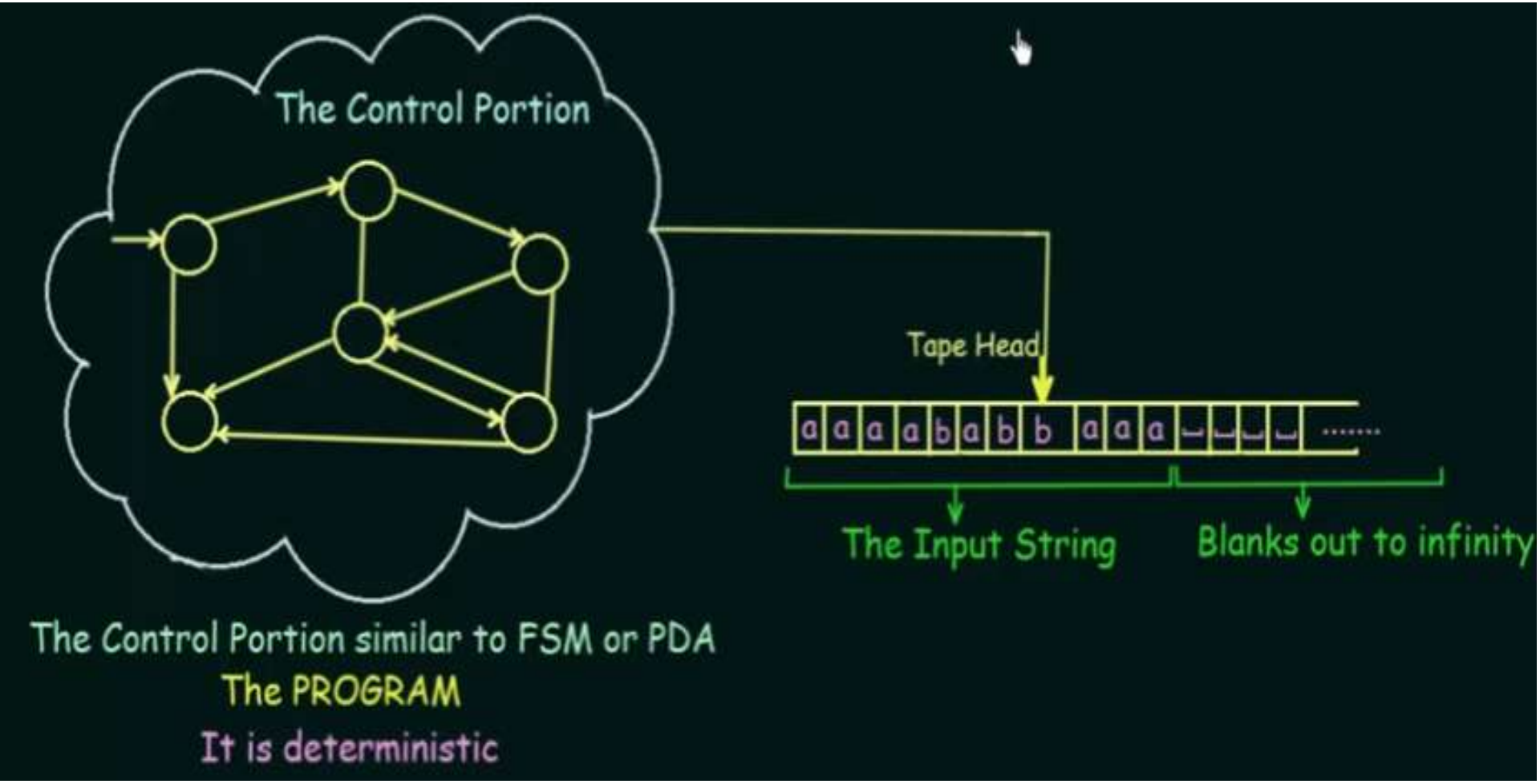
Operations on the Tape:

-> Read /Scan symbol below the Tape Head

-> Update / Write a symbol below the Tape Head

-> Move the Tape Head one step LEFT

-> Move the Tape Head one step RIGHT


The Control Portion
Tape Head
a a a a b a b b a a a
The Input String
Blanks out to infinity
The Control Portion similar to FSM or PDA
The PROGRAM
It is deterministic

Rules of Operation - 1

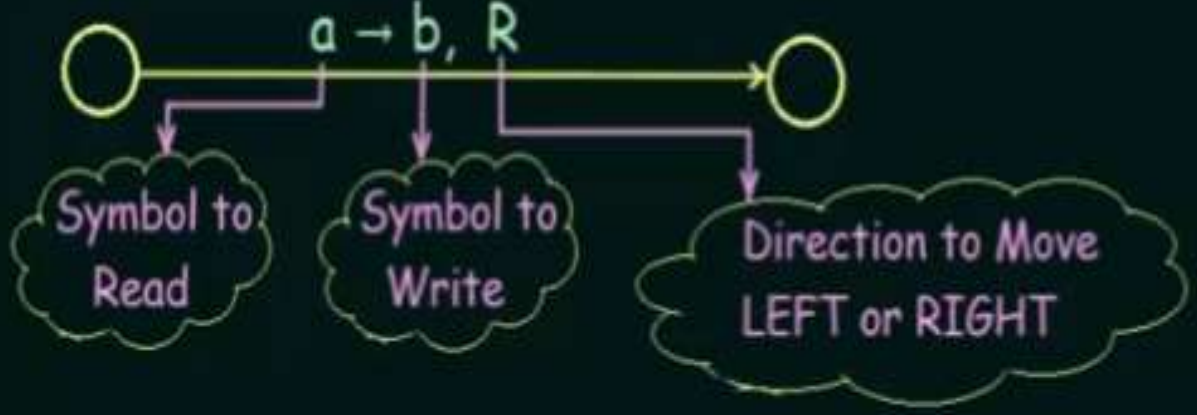
At each step of the computation:

-> Read the currect symbol

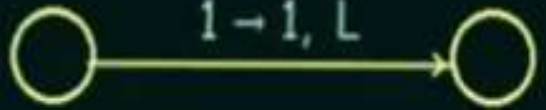
-> Update (i.e. write) the same cell

-> Move exactly one cell either LEFT or RIGHT

If we are at the left end of the tape, and trying to move LEFT, then do not move. Stay at the left end



If you dont want to update the cell, JUST WRITE THE SAME SYMBOL



Rules of Operation - 2

-> Control is with a sort of FSM

-> Initial State

-> Final States: (there are two final states)

1) The ACCEPT STATE

2) The REJECT STATE

-> Computation can either

1) HALT and ACCEPT

2) HALT and REJECT

3) LOOP (the machine fails to HALT)

Turing Machine (Formal Definition)

A Turing Machine can be defined as a set of 7 tuples

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

$Q \rightarrow$ Non empty set of States

$\Sigma \rightarrow$ Non empty set of Symbols

$\Gamma \rightarrow$ Non empty set of Tape Symbols

$\delta \rightarrow$ Transition function defined as

$$Q \times \Sigma \rightarrow \Gamma \times (R/L) \times Q$$

$q_0 \rightarrow$ Initial State

$b \rightarrow$ Blank Symbol

$F \rightarrow$ Set of Final states (Accept state & Reject State)

Thus, the Production rule of Turing Machine will be written as

$$\delta (q_0, a) \rightarrow (q_1, y, R)$$

Recursively Enumerable Language:

A Language L and Σ is said to be Recursively Enumerable if there exists a Turing Machine that accepts it.

Instantaneous descriptions

- Instantaneous description (ID): depicts the characteristics of the machine as transitions are carried out
  - For finite automata, the state of the machine and the remaining input is sufficient
- For TM's, the following are needed for an ID:
  - State
  - Symbols on the tape
  - Position of the tape head
- Can be expressed as $X_1X_2...X_{i-1}qX_iX_{i+1}...X_n$ which means the TM is in state q, the tape contains $X_1X_2...X_n$ and the tape head is at X_i

- The *instantaneous description* (ID) of a TM is represented by
- $X_1 X_2 \ldots X_{i-1} q X_i X_{i+1} \ldots X_n$ in which
- q is the current state;
- The tape head is scanning the ith symbol X_i from the left;
- $X_1 X_2 \ldots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank symbols.

– Moves of a TM M are denoted by $\vdash_M$ or $\vdash$ as follows:

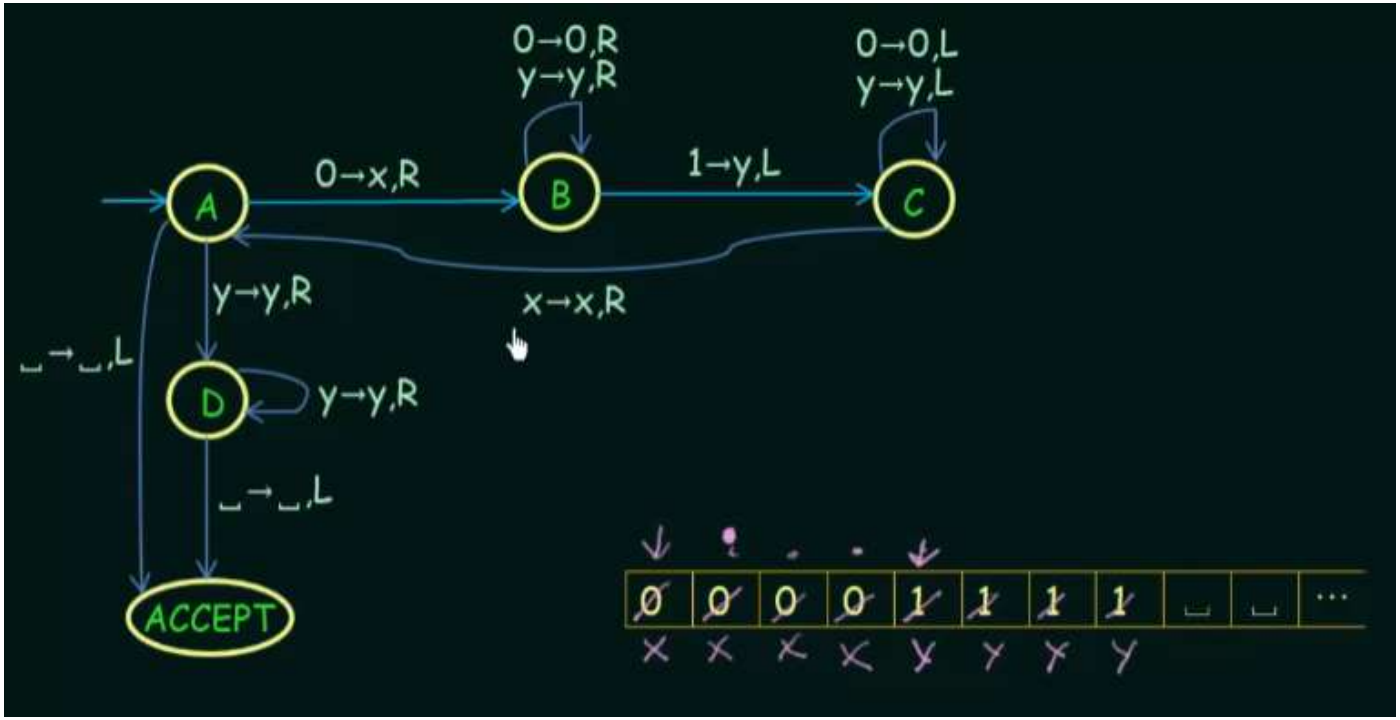
If $\delta(q, X_i) = (p, Y, L)$ (a leftward move), then we write the following to describe the left move:

$$X_1 X_2 \ldots X_{i-1} q X_i X_{i+1} \ldots X_n \vdash_M X_1 X_2 \ldots X_{i-2} p X_{i-1} Y X_{i+1} \ldots X_n$$

– Right moves are defined similarly.

– **Example 8.2** --- Design a TM to accept the language $L = \{0^n1^n \mid n \geq 1\}$ as follows. (continued)





- To accept 0011 ---

(use $\Rightarrow$ instead of $\vdash$)

$$q_0 0011 \Rightarrow X q_1 011 \Rightarrow X0 q_1 11 \Rightarrow X q_2 0 Y1 \Rightarrow q_2 X0 Y1$$

$$\Rightarrow X q_0 0 Y1 \Rightarrow XX q_1 Y1 \Rightarrow XXY q_1 1 \Rightarrow XX q_2 YY \Rightarrow X q_2 XYY$$

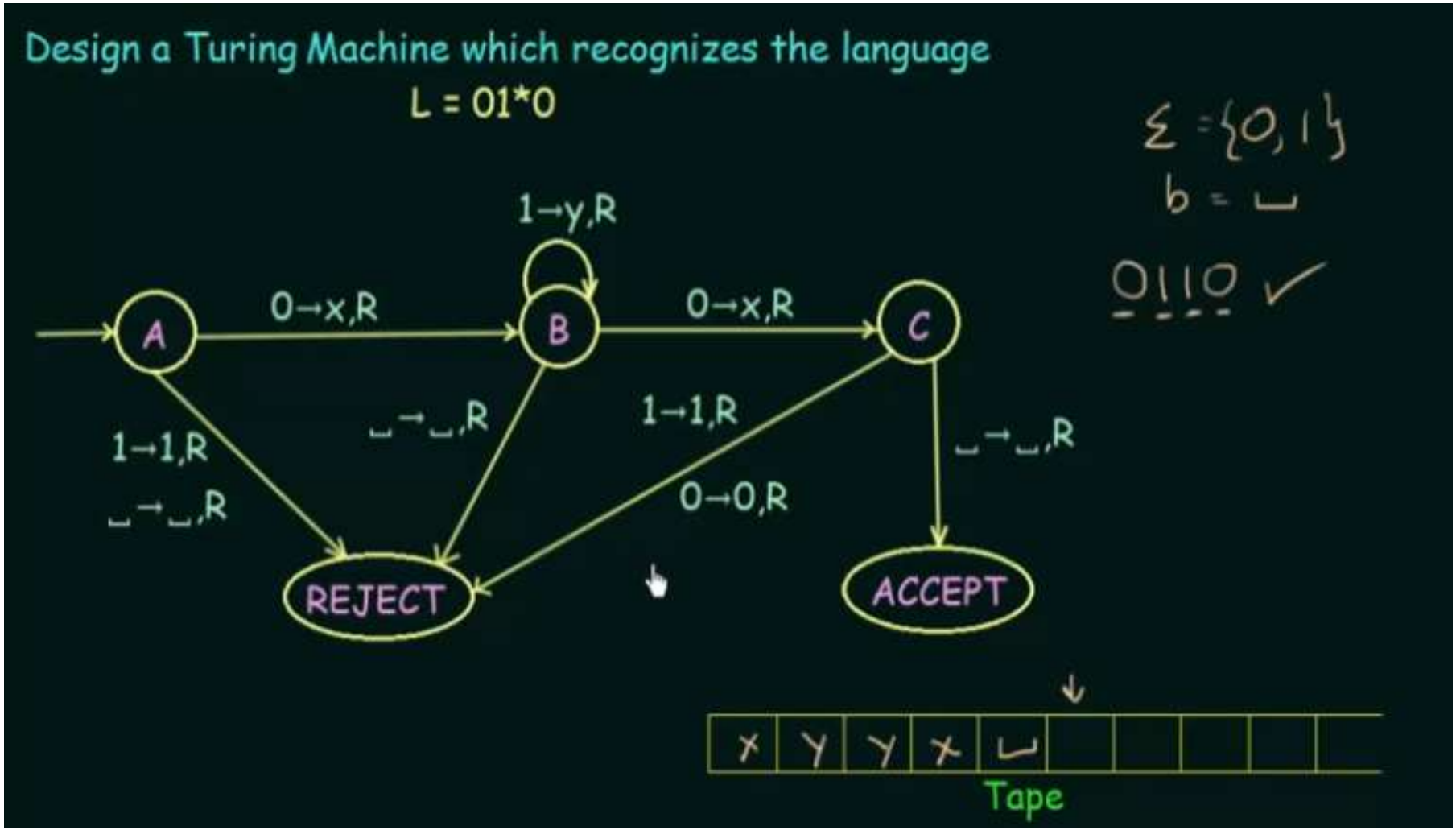
$$\Rightarrow XX q_0 YY \Rightarrow XXY q_3 Y \Rightarrow XXYY q_3 B \Rightarrow XXYY B q_4 B$$

8.2.6 TM's and Halting

- Another notion for accepting strings by TM's --- acceptance by halting.
- We say a TM halts if it enters a state q scanning a tape symbol X, and there is no move in this situation, i.e., $\delta(q, X)$ is *undefined*.

- Languages with TM's that do halt eventually, regardless whether or not they accept, are called *recursive languages* (considered in Sec. 9.2.1)
- TM's that always halt, regardless of whether or not they accept, are a good model of an "algorithm."
- So TM's that always halt can be used for studies of decidability (see Chapter 9).

Example 1

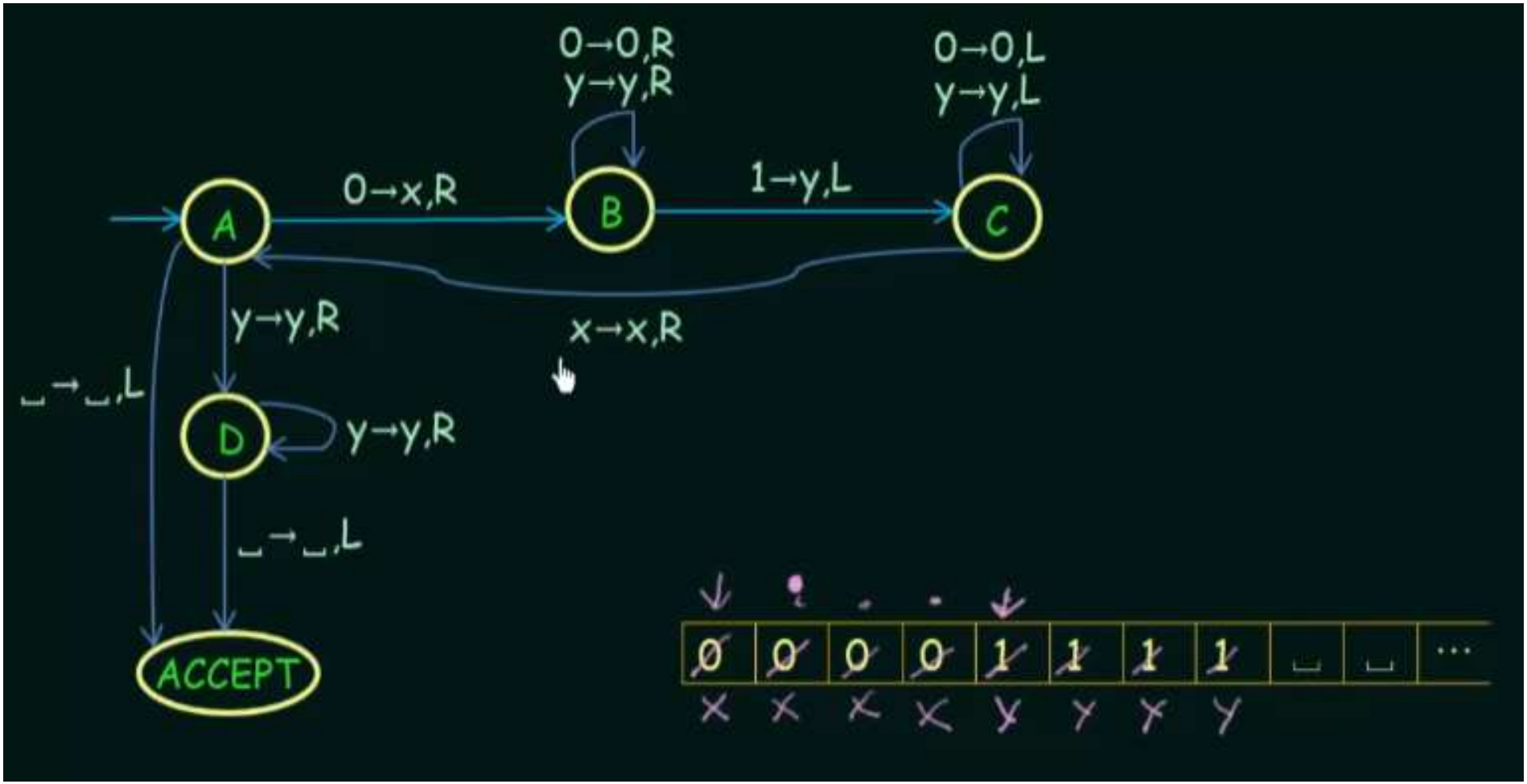

Design a Turing Machine which recognizes the language
L = 01*0
Σ = {0, 1}
b = ␣
0110 ✓
A
B
C
REJECT
ACCEPT
0→x,R
1→y,R
0→x,R
1→1,R
␣→␣,R
␣→␣,R
1→1,R
␣→␣,R
0→0,R
x
y
y
x
␣
Tape

Design a Turing Machine which recognizes the language $L = 0^N 1^N$

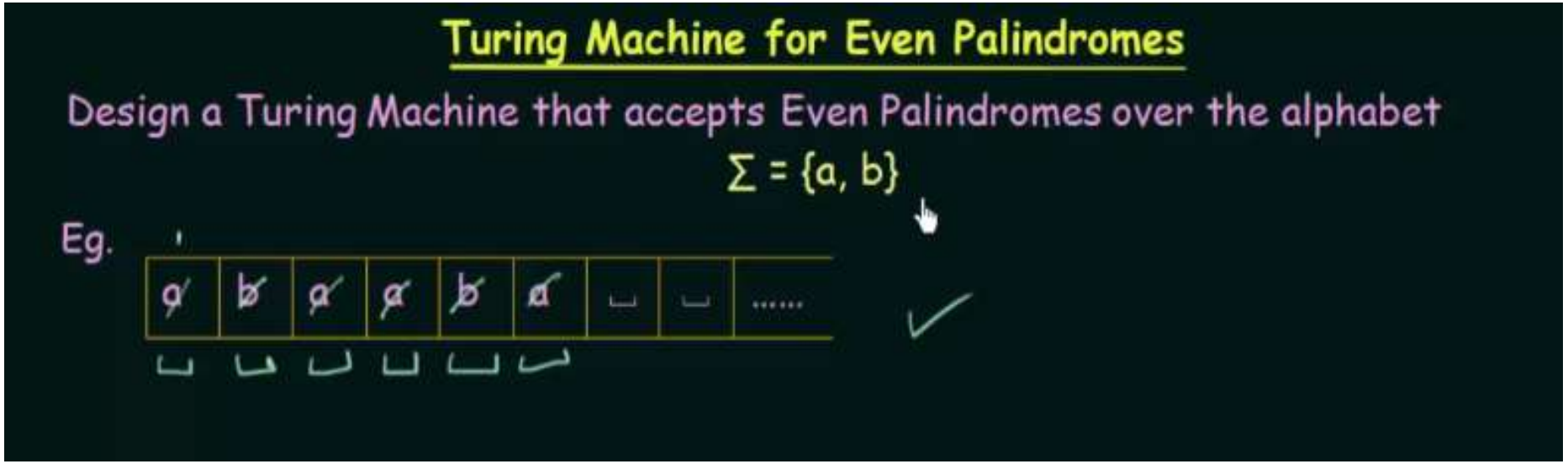


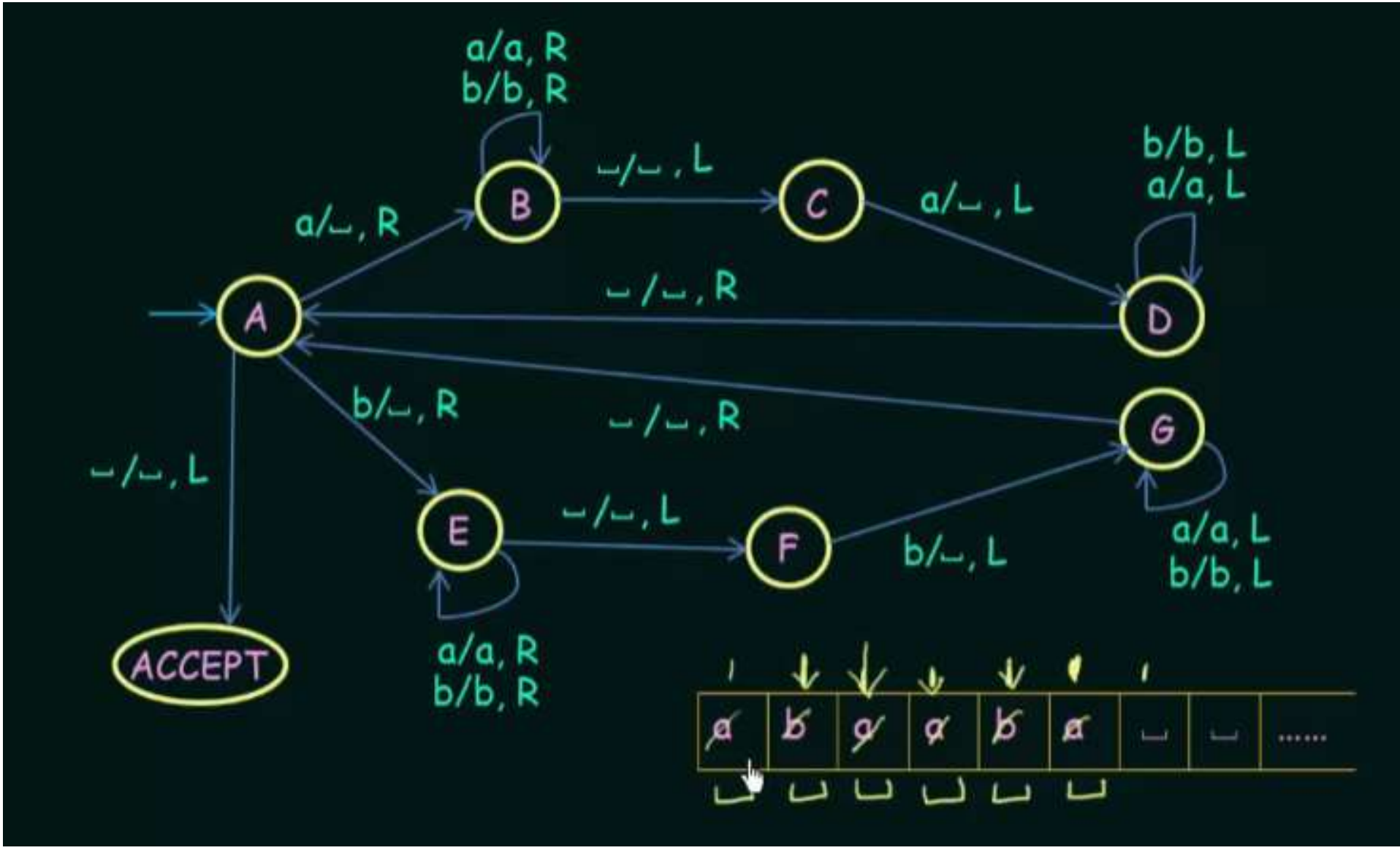
Algorithm:

- Change "0" to "x"
- Move RIGHT to First "1"
  If None: REJECT
- Change "1" to "y"
- Move LEFT to Leftmost "0"
- Repeat the above steps until no more "0"s
- Make sure no more "1"s remain


0→0,R
y→y,R
0→0,L
y→y,L
A
0→x,R
B
1→y,L
C
y→y,R
x→x,R
␣→␣,L
D
y→y,R
␣→␣,L
ACCEPT
0 0 0 0 1 1 1 1 ␣ ␣ ⋯
x x x x y y x y

ww^r


Turing Machine for Even Palindromes
Design a Turing Machine that accepts Even Palindromes over the alphabet
Σ = {a, b}
Eg.
a b a a b a ⊔ ⊔ .....

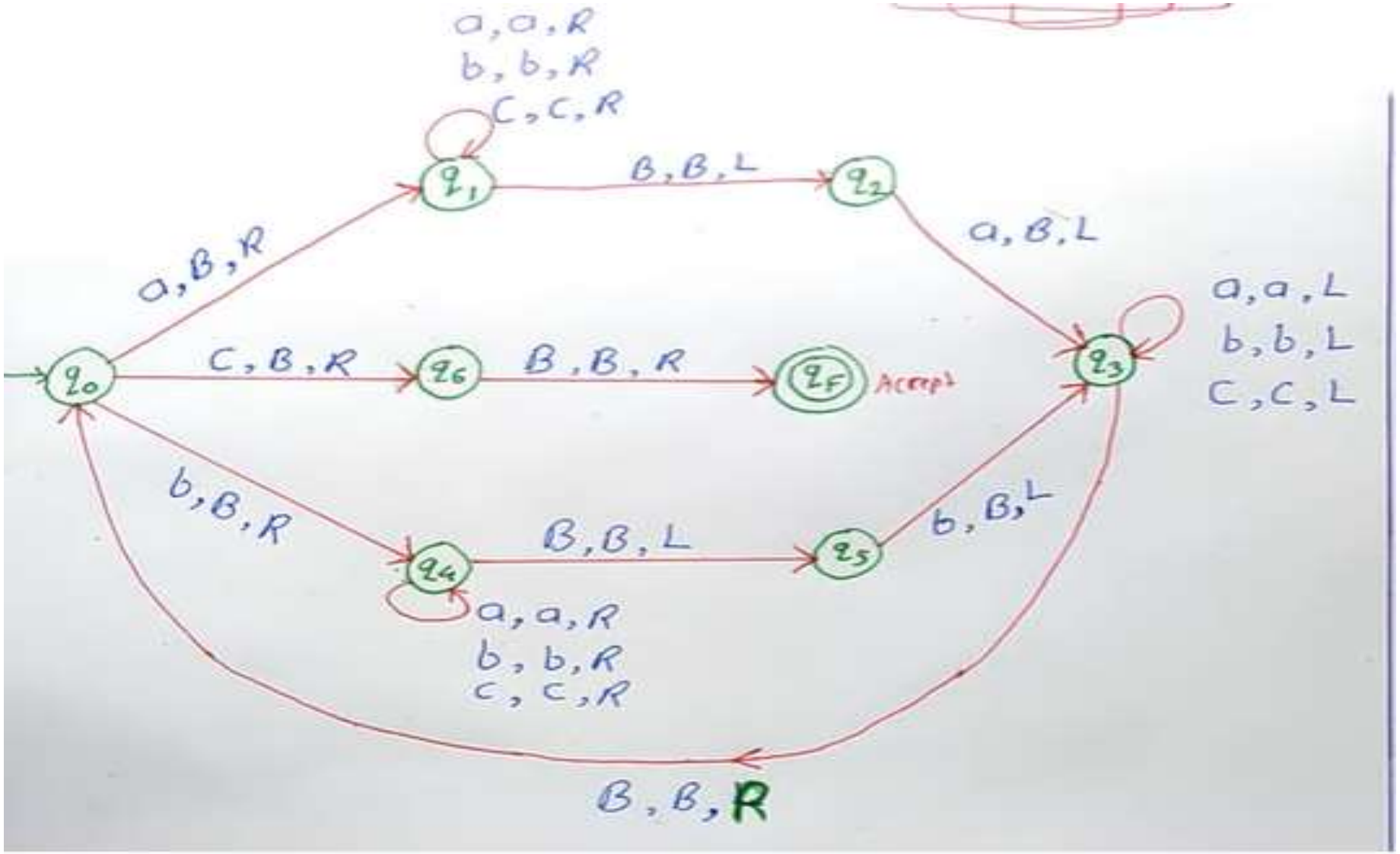

a/a, R
b/b, R
␣/␣, L
b/b, L
a/a, L
a/␣, R
a/␣, L
B
C
A
D
␣/␣, R
b/␣, R
␣/␣, R
G
␣/␣, L
E
␣/␣, L
F
b/␣, L
a/a, L
b/b, L
ACCEPT
a/a, R
b/b, R
a
b
a
a
b
a
␣
␣
.....

TM for addition

- **Examples –** (2 + 3) will be given as 0 0 c 0 0 0:

- Input : 0 0 c 0 0 0 // 2 + 3
- Output : 0 0 0 0 0 // 5
- Input : 0 0 0 0 c 0 0 0 // 4 + 3
- Output : 0 0 0 0 0 0 0 // 7

WCW^r -Odd length palindromes



Transition Table

δ	Tape Symbols(Γ)				
states	a	b	X	Y	B
q0	(q1,X,R)	-	-	(q3,Y,R)	-
q1	(q1,a,R)	(q2,Y,R)	-	(q1,Y,R)	-
q2	(q2,a,L)	-	(q0,X,R)	(q2,Y,L)	-
q3	-	-	-	(q3,Y,R)	(q4,B,R)
q4	-	-	-	-	-

Programming Techniques for Turing Machine

- TM is exactly as powerful as a conventional computer.
- We present a number of examples of how we might think of the tape and finite control of the Turing machine.
- None of these tricks extend the basic model of the TM they are only notational conveniences
  - Storage in the State
  - Multiple tracks
  - Subroutines

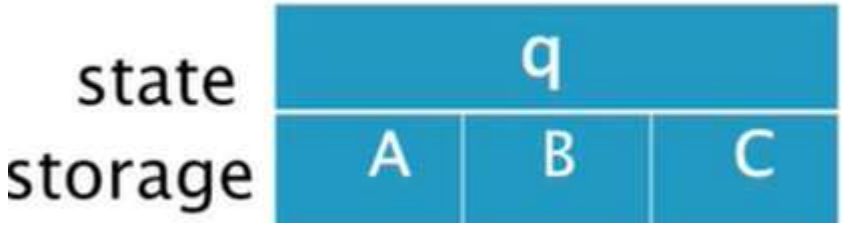
- **Programming techniques for Turing Machines (TMs)**, which aim to make the process of designing and understanding TMs more efficient or easier, but without changing the fundamental computational power of the TM. The key techniques mentioned **storage in the state, multiple tracks, and subroutines** help to improve the organization and functionality of Turing machine programs. Let's break down the need for each technique:

1. Storage in the State

- **Need**: Turing machines traditionally store information on their tape. However, sometimes it's more efficient or necessary to store temporary data in the **state** of the machine itself. By encoding data in the machine's states, we reduce the need to manage complex tape modifications or keep unnecessary symbols.
- **Example**: If you're running a Turing machine that counts or performs a sequence of actions based on prior steps, storing intermediate results in states can simplify the process, avoiding excess tape rewriting.

Programming techniques

- We can use the finite control not only to represent a position in the program of the Turing machine but to hold a finite amount of data



q– control state

A,B,C– data elements

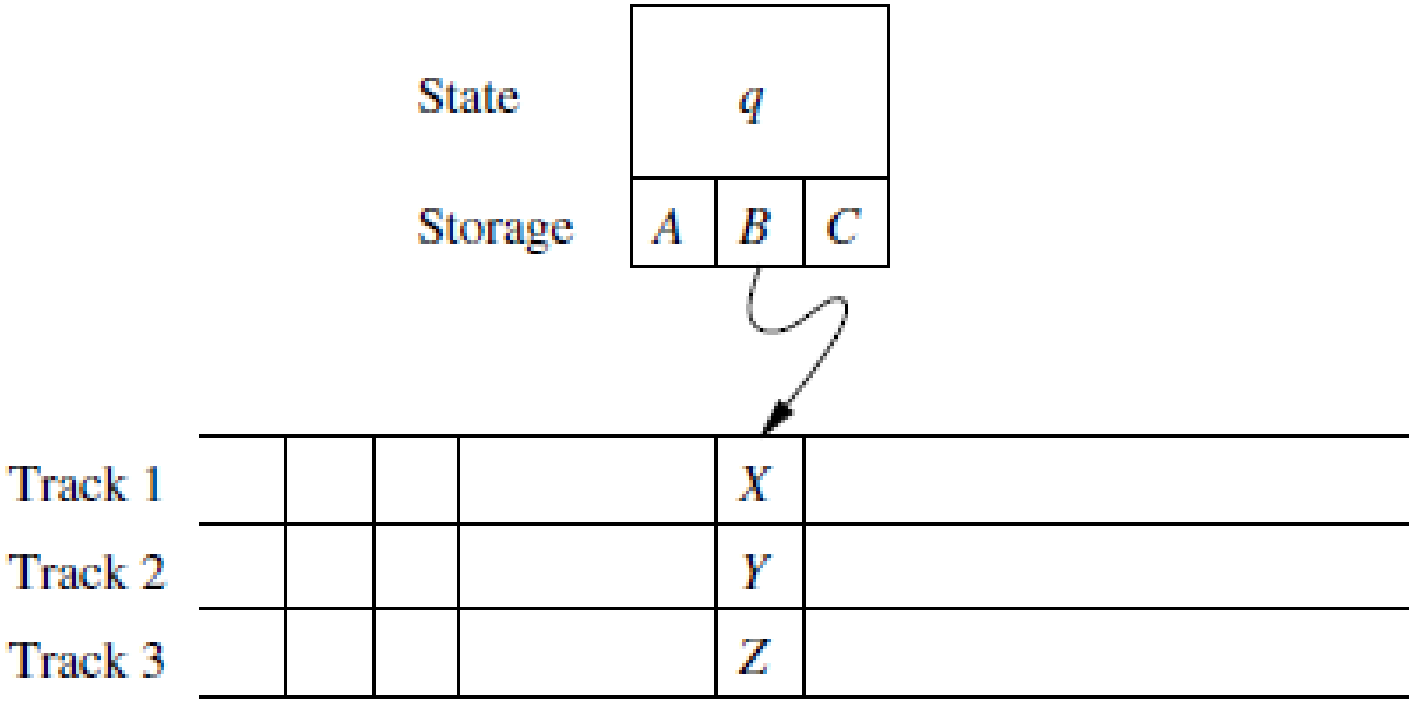
  - The state of the machine is thought of as a tuple [q,A,B,C]

2. Multiple Tracks

- **Need**: Turing machines typically have a single tape and a single head for reading and writing. By using **multiple tracks**, you can simulate multiple tapes (effectively using more than one data store). This technique can greatly reduce the complexity of certain computations by enabling parallel storage of information, which can be read or written independently.
- **Example**: If a Turing machine needs to compare two binary strings, rather than overwriting symbols on one tape, you can use one track for each string, simplifying the comparison process.

Multiple Tracks

- Another useful "trick" is to think of the tape of a TM as composed of several tracks.
- Each track can hold one symbol and the tape alphabet of the TM consists of tuples, with one component for each "track".
- A multitrack Turing Machine is a specific type of Multi-tape Turing Machine.


State
q
Storage
A
B
C
Track 1
Track 2
Track 3
X
Y
Z

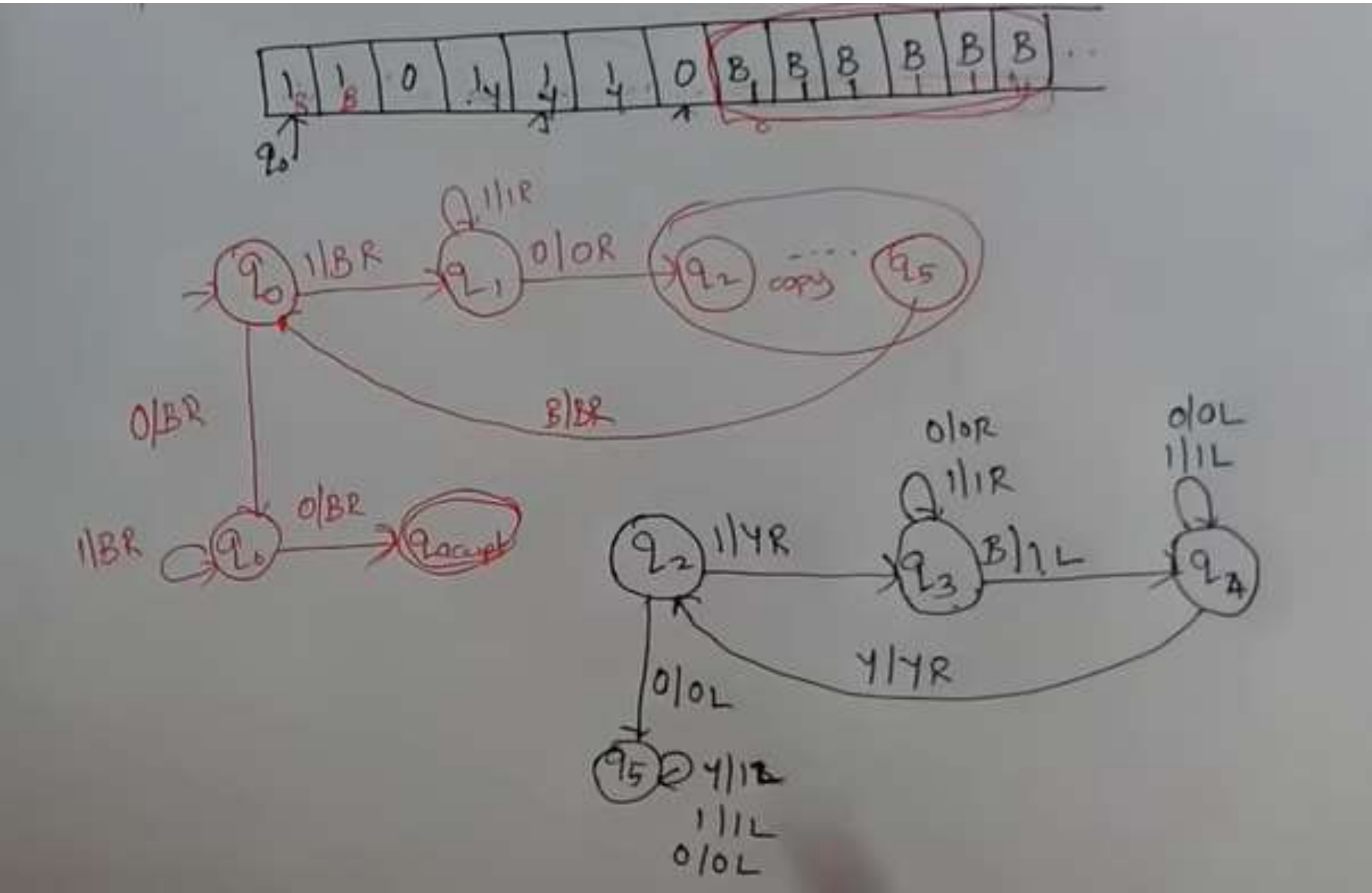
3. Subroutines

- **Need**: Just like in conventional programming, **subroutines** (or "functions") allow you to modularize and reuse code. This technique helps avoid the redundancy of writing the same sequence of operations multiple times. It can also make your Turing machine program easier to understand and debug.
- **Example**: If the machine has to perform a sequence of actions (like finding the first blank on the tape or copying a segment of the tape), instead of repeating these actions, a subroutine can be called whenever that sequence is needed.

Subroutines

- It helps to think of TM as built from a collection of interacting components or "subroutines".
- This set of states includes start state & another temporary state.
- The "call" of a subroutine occurs when there is a transition to its initial state.

Example: Multiplication 2 *3



- A Turing machine subroutine is a set of states that perform some useful process
- This set of states includes a start state and another state that temporarily has no moves and that serves as the return state to pass control to whatever other set of states called the subroutine
- The call of a subroutine occurs whenever there is a transition to its initial state
- Since the TM has no mechanism for remembering a return address that is a state to go to after it finishes should our design of a TM call for one subroutine to be called from several states
- We can make copies of the subroutine using a new set of states for each copy
- The calls are made to the start states of different copies of the subroutine and each copy returns to a different state

Extensions to the Basic Turing Machine

- Multi tape Turing machines
- Non-Deterministic Turing Machines

Multi tape Turing machines

- The device has a finite control (state) and some finite number of tapes.
- Each tape is divided into cells and each cell can hold any symbol of the finite tape alphabet
- As in the single tape TM the set of tape symbols includes a blank and has a subset called the input symbols of which the blank is not a member
- The set of states includes an initial state and some accepting states

- Initially

1. The input, a finite sequence of input symbols, is placed on the first tape.
2. All other cells of all the tapes hold the blank.
3. The finite control is in the initial state.
4. The head of the first tape is at the left end of the input.
5. All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially; all cells of these tapes "look" the same.

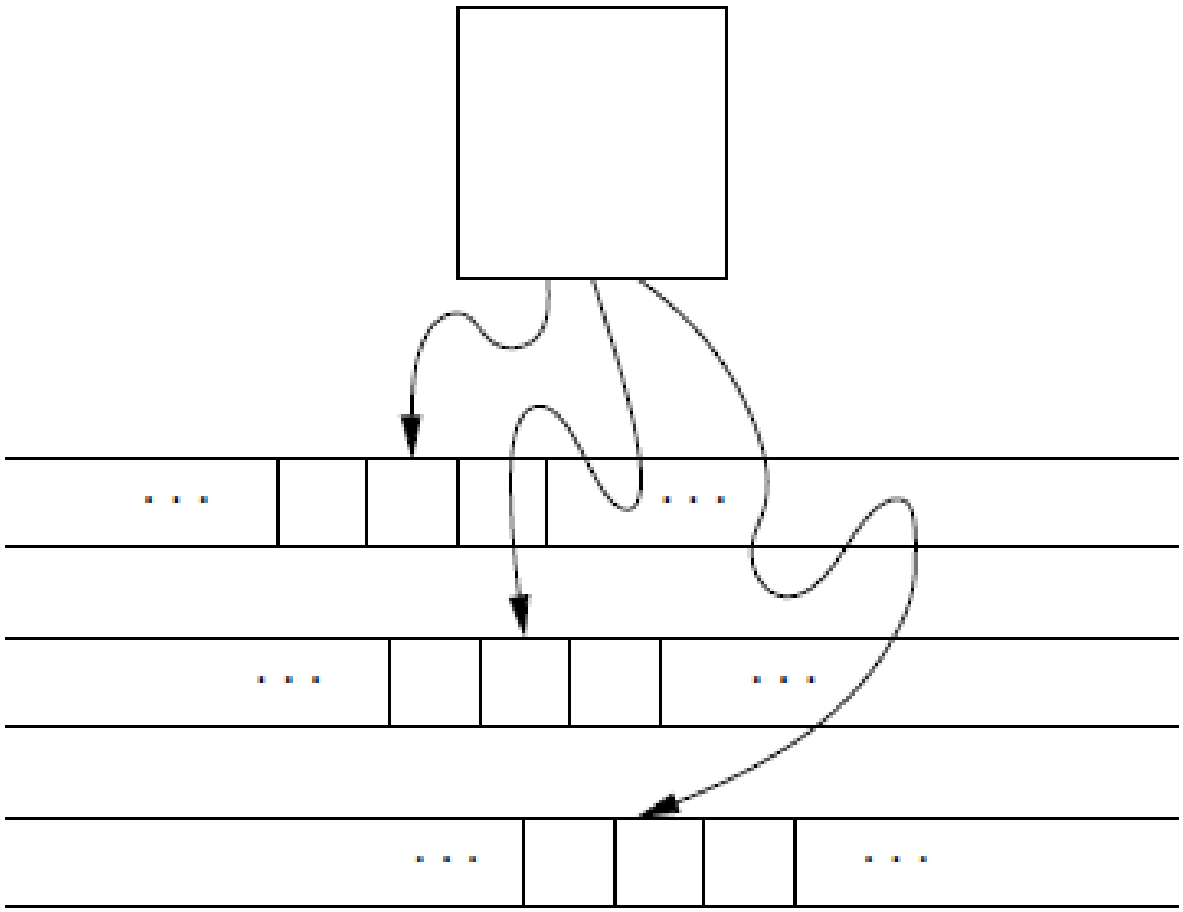


Figure 8.16: A multitape Turing machine

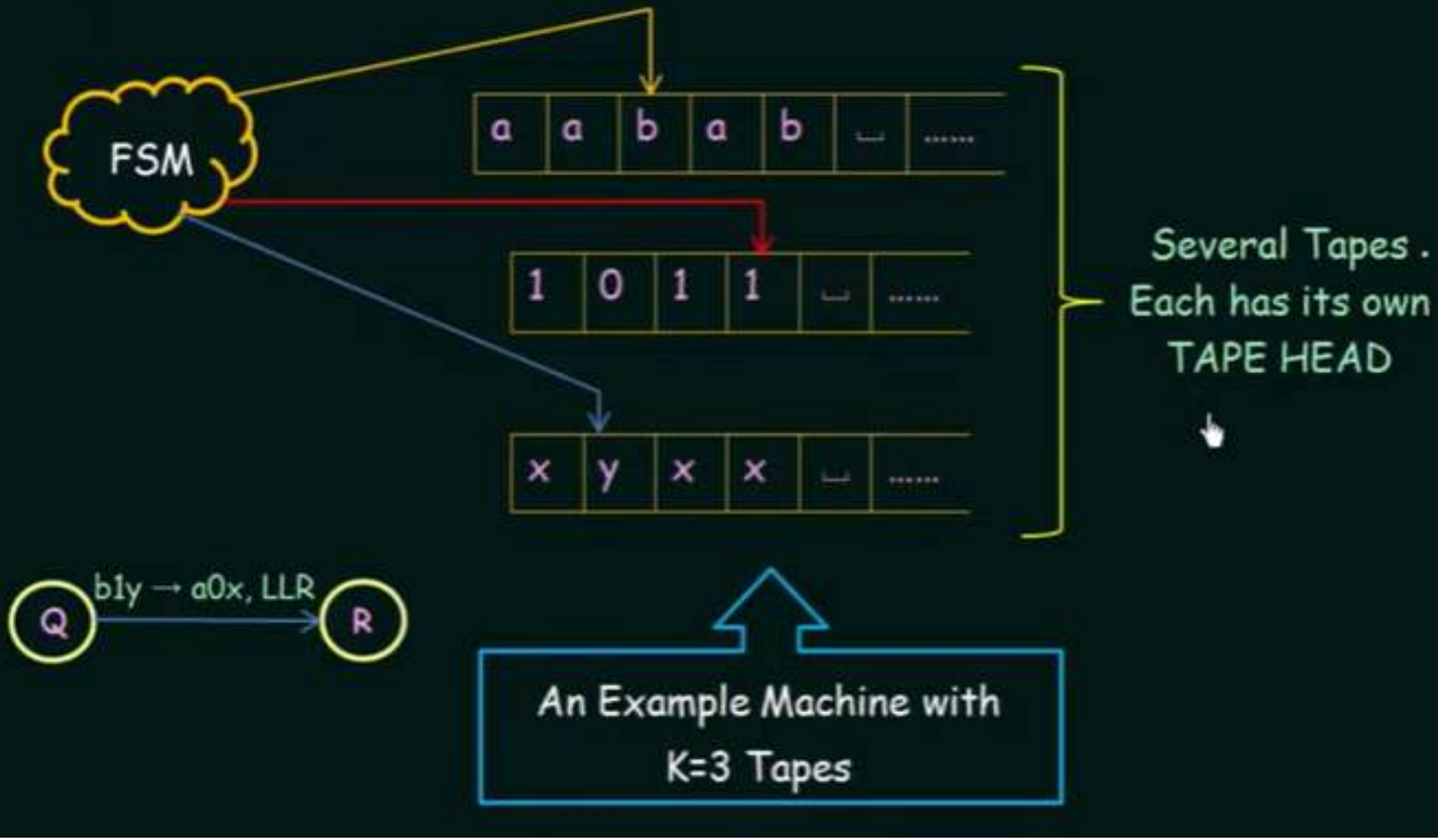
A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

1. The control enters a new state, which could be the same as the previous state.

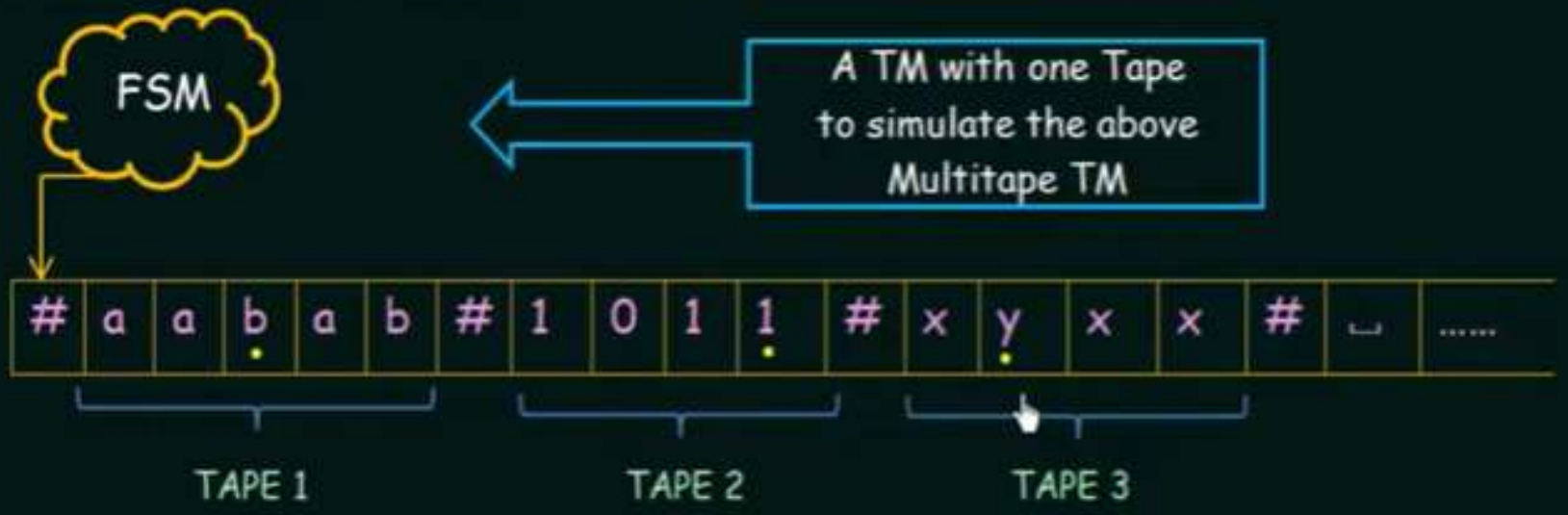
2. On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.

3. Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

Multitape Turing Machine



Single Tape Turing Machine



- Add "dots" to show where Head "K" is
- To simulate a transition from state Q, we must scan our Tape to see which symbols are under the K Tape Heads
- Once we determine this and are ready to MAKE the transition, we must scan across the tape again to update the cells and move the dots

- **1. Multi-Tape Turing Machines**
- **Definition**: A multi-tape Turing machine is a variant of the standard Turing machine that has more than one tape and multiple read/write heads. Each tape operates independently but can be accessed simultaneously. This allows the machine to perform more complex operations in parallel.

- **Key Characteristics**:
  - **Multiple Tapes**: Instead of having a single tape, a multi-tape Turing machine has multiple tapes, each with its own read/write head.
  - **Parallel Processing**: The machine can perform operations on multiple tapes at once, effectively allowing it to "store" and process more information simultaneously.
  - **Enhanced Efficiency**: By using multiple tapes, the machine can carry out tasks that would otherwise require much more time or complexity on a single tape. For example, comparing two strings on separate tapes instead of overwriting one of the strings on the same tape.

- **Computational Power**: Although multi-tape Turing machines can be more efficient in practice, they do not increase the computational power of Turing machines. In terms of **computational complexity**, multi-tape Turing machines are equivalent to single-tape Turing machines. Any problem solvable by a multi-tape machine can also be solved by a standard Turing machine, but the multi-tape version might do it more efficiently.

Non-Deterministic Turing Machines

- A nondeterministic Turing machine NTM differs from the deterministic variety.

state q and tape symbol X, $\delta(q, X)$ is a set of triples

$$\{(q_1, Y_1, D_1),\ (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

- The NTM can choose at each step any of the triples to be the next move It cannot however pick a state from one a tape symbol from another and the direction from yet another
- The NTM's accept no languages not accepted by a deterministic TM

- That is M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input to an ID with an accepting state

If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$.

Nondeterministic Turing Machines:

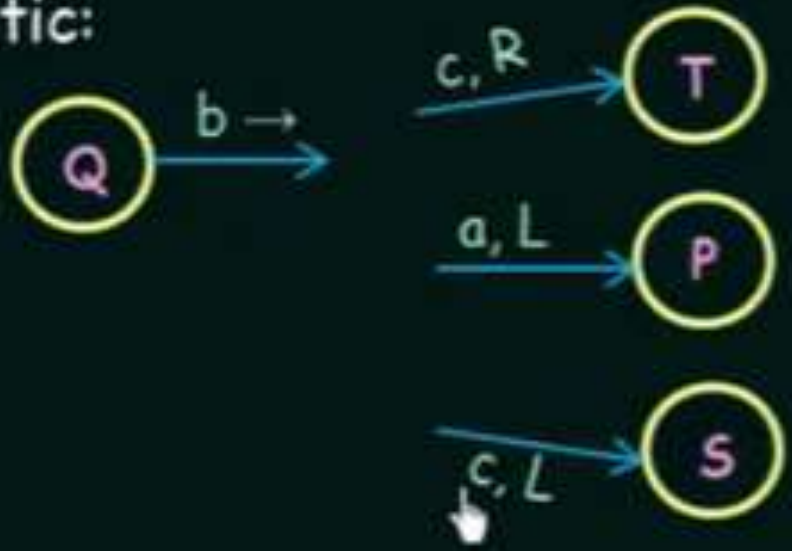
Transition Function:

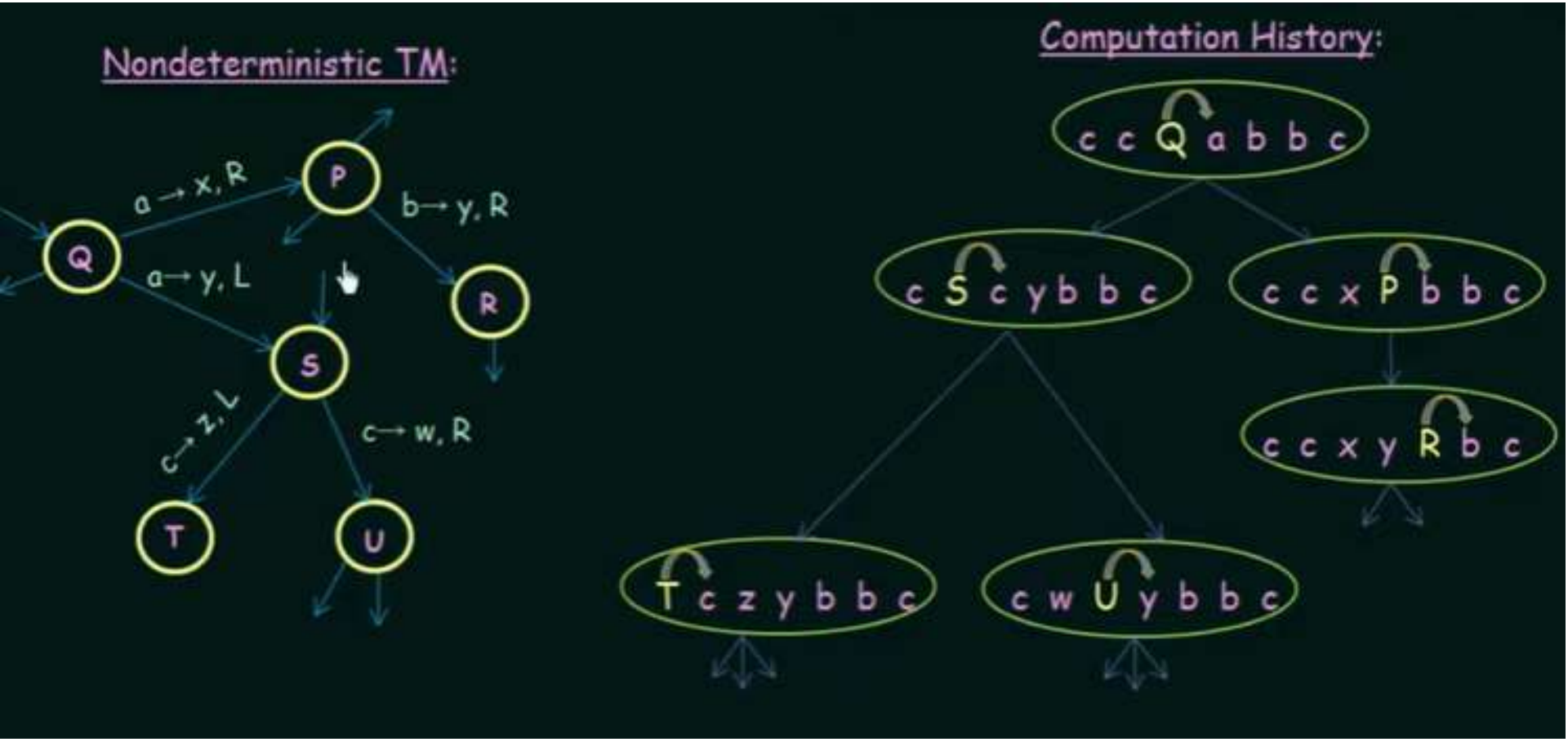
$\delta : \quad Q \times \Sigma \rightarrow P\{\Gamma \times (R/L) \times Q\}$

Deterministic:



Nondeterministic:




Nondeterministic TM:
P
Q
R
S
T
U
a → x, R
b → y, R
a → y, L
c → z, L
c → w, R
Computation History:
c c Q a b b c
c S c y b b c
c c x P b b c
c c x y R b c
T c z y b b c
c w U y b b c

Outcomes of a Nondeterministic Computation:

ACCEPT If any branch of the computation accepts, then the nondeterministic TM will Accept.

REJECT If all branches of the computation HALT and REJECT (i.e. no branches accept, but all computations HALT) then the Nondeterministic TM Rejects.

LOOP Computation continues but ACCEPT is never encountered. Some branches in the computation history are infinite.

Non-Deterministic Turing Machines (NDTMs)

- A non-deterministic Turing machine can choose from multiple possible transitions at each step, unlike a deterministic machine that has only one action per input. This means an NDTM can explore many computation paths simultaneously.
- **Acceptance**: The input is accepted if *any* path leads to an accept state.
- **Power**: NDTMs recognize the same class of languages as deterministic Turing machines but can solve some problems more efficiently in theory.
- **Note**: NDTMs are theoretical models; physical machines can’t truly implement non-determinism. They are important in complexity theory, especially for problems like NP-complete problems that are solvable in polynomial time by NDTMs but not known to be efficiently solvable by deterministic machines.

Decidability and Undecidability

Recursive Language:

- A language 'L' is said to be recursive if there exists a Turing machine which will accept all the strings in 'L' and reject all the strings not in 'L'.
- The Turing machine will halt every time and give an answer (accepted or rejected) for each and every string input.

Recursively Enumerable Language:

- A language 'L' is said to be a recursively enumerable language if there exists a Turing machine which will accept (and therefore halt) for all the input strings which are in 'L'.
- But may or may not halt for all input strings which are not in 'L'.

Decidable Language:

A language 'L' is decidable if it is a recursive language. All decidable languages are recursive languages and vice-versa.

Partially Decidable Language:

A language 'L' is partially decidable if 'L' is a recursively enumerable language.

Undecidable Language:

- A language is undecidable if it is not decidable.
- An undecidable language may sometimes be partially decidable but not decidable.
- If a language is not even partially decidable, then there exists no Turing machine for that language

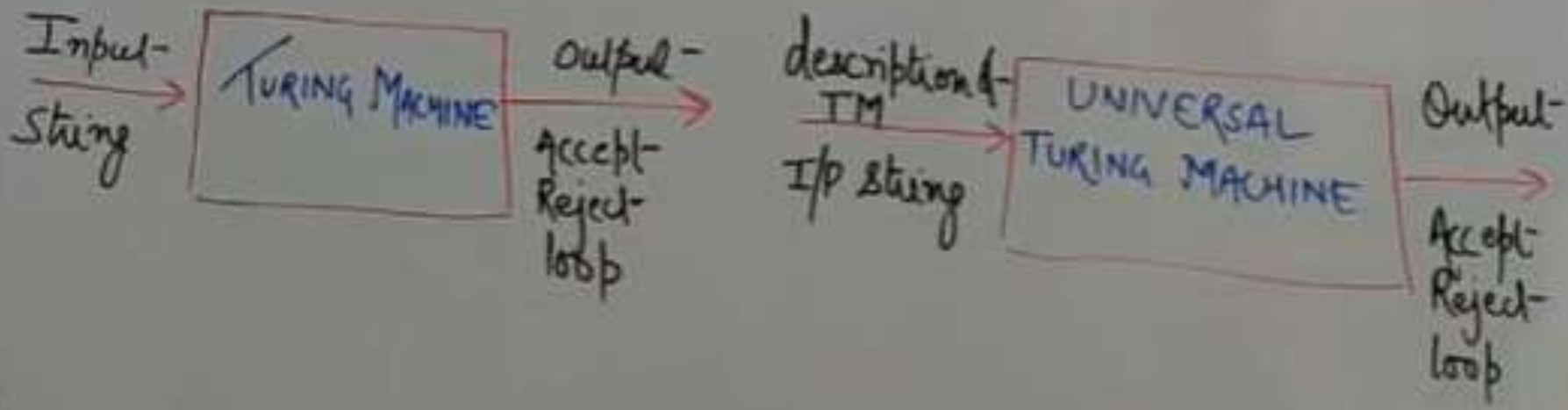
Recursive Language	TM will always Halt
Recursively Enumerable Language	TM will halt sometimes & may not halt sometimes
Decidable Language	Recursive Language
Partially Decidable Language	Recursively Enumerable Language
UNDECIDABLE	No TM for that language

UNIVERSAL TURING MACHINE

Till now for 01^*1, 01^+1, a^nb^n, $a^nb^nc^n$, WcW...
we have created different T.M for all diff lang

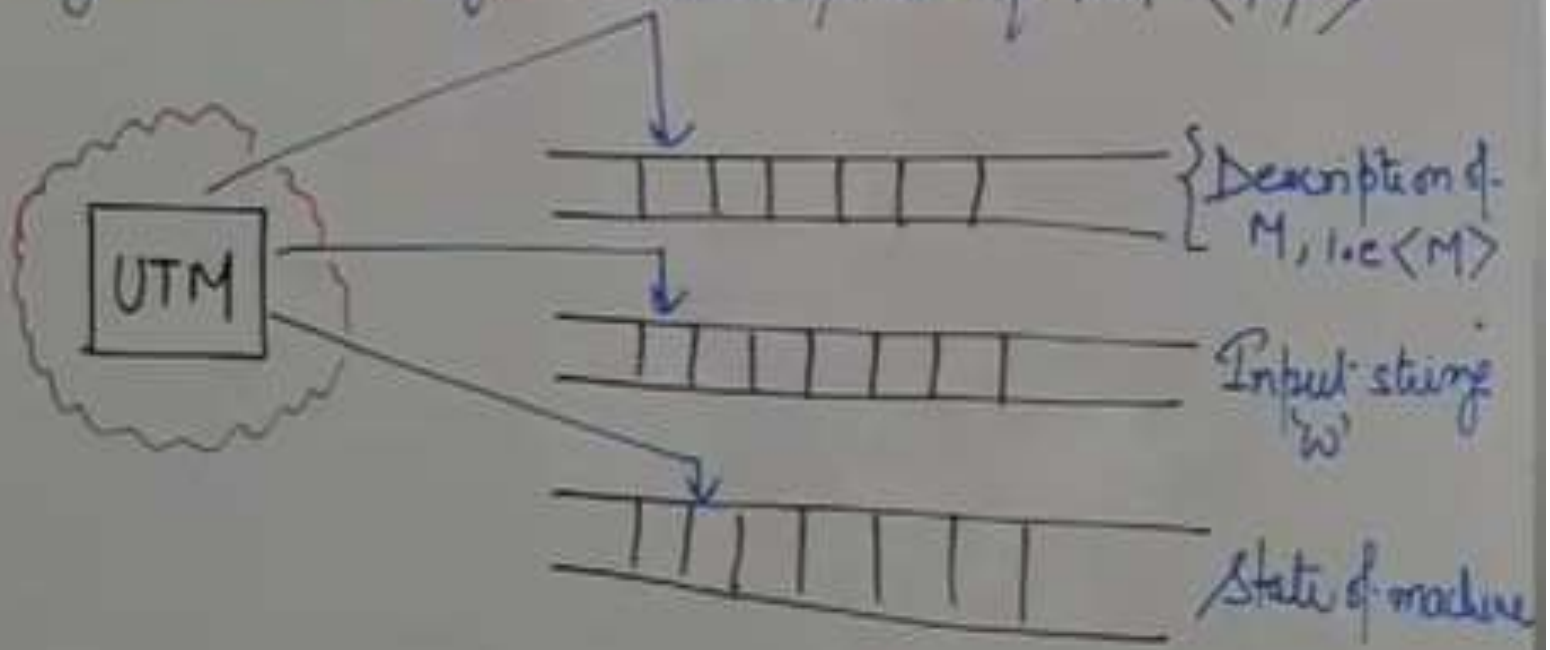
So we need a single machine which can solve any type of problem. Such machine is known as UTM

hence Universal turing machine is a specified TM that can simulate the behaviour of any T.M



UNIVERSAL TURING MACHINE

⇒ To design UTM we use MTTM because along with input string 'w' we also give description of TM <M>



UNIVERSAL TURING MACHINE

UTM can act like general purpose computer

i.e TM is a program & UTM is a computer

There is good diff b/w UTM and General purpose computer

- TM has Infinite length tape(memory) but computer doesn't

The Universal Turing Machine

The Language

$A_{TM} = \{ \langle M,w \rangle \mid M \text{ is a Turing Machine and } M \text{ accepts } w \}$

is Turing Recognizable

Given the description of a TM and some input, can we determine whether the machine accepts it?

- Just Simulate/ Run the TM on the input

M Accepts w: Our Algorithm will Halt & Accept
M Rejects w: Our Algorithm will Halt & Reject.
M Loops on w: Our Algorithm will not Halt.

The Universal Turing Machine

Input: M = the description of some TM

w = an input string for M

Action: - Simulate M

- Behave just like M would (may accept, reject or loop)

The UTM is a recognizer (but not a decider) for

$A_{TM} = \{ \langle M,w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$

THE POST CORRESPONDENCE PROBLEM (PCP)

- The Post Correspondence Problem (PCP) was first introduced by Emil Post in 1946.
- The problem over an alphabet Σ belongs to a class of yes/no problems and is stated as follows:
- Consider the two lists $X = (x_1, x_2 \ldots x_n)$, $Y = (y_1, y_2 \ldots y_n)$ of nonempty strings over an alphabet $\Sigma = \{0, 1\}$. The PCP is to determine whether or not there exist $i_1\, i_2 \ldots i_m$, where $1 <= i_i <= n$, such that

$$x_{i1}\, x_{i2} \ldots x_{im} = y_{i1}\, y_{i2} \ldots y_{im}$$

THE POST CORRESPONDENCE PROBLEM (PCP)

Example 1: Does the PCP with two lists x = (b, bab^3, ba) and v = (b^3, ba, a) have a solution?



Solution : < 2, 1, 1, 3 >

Ex-2

- Let’s try understanding this by taking two **lists A** and **B**
- A=[aa, bb, abb] and B=[aab, ba, b]
- 1, 2, 1, 3

Ex-3

	A	B
1	1	111
2	10111	10
3	10	0

2 1 1 3

Ex-4

	A	B
1	100	1
2	0	100
3	1	00

Find whether the lists —

$M = (abb, aa, aaa)$ & $N = (bba, aaa, aa)$

have a post correspondence solution?

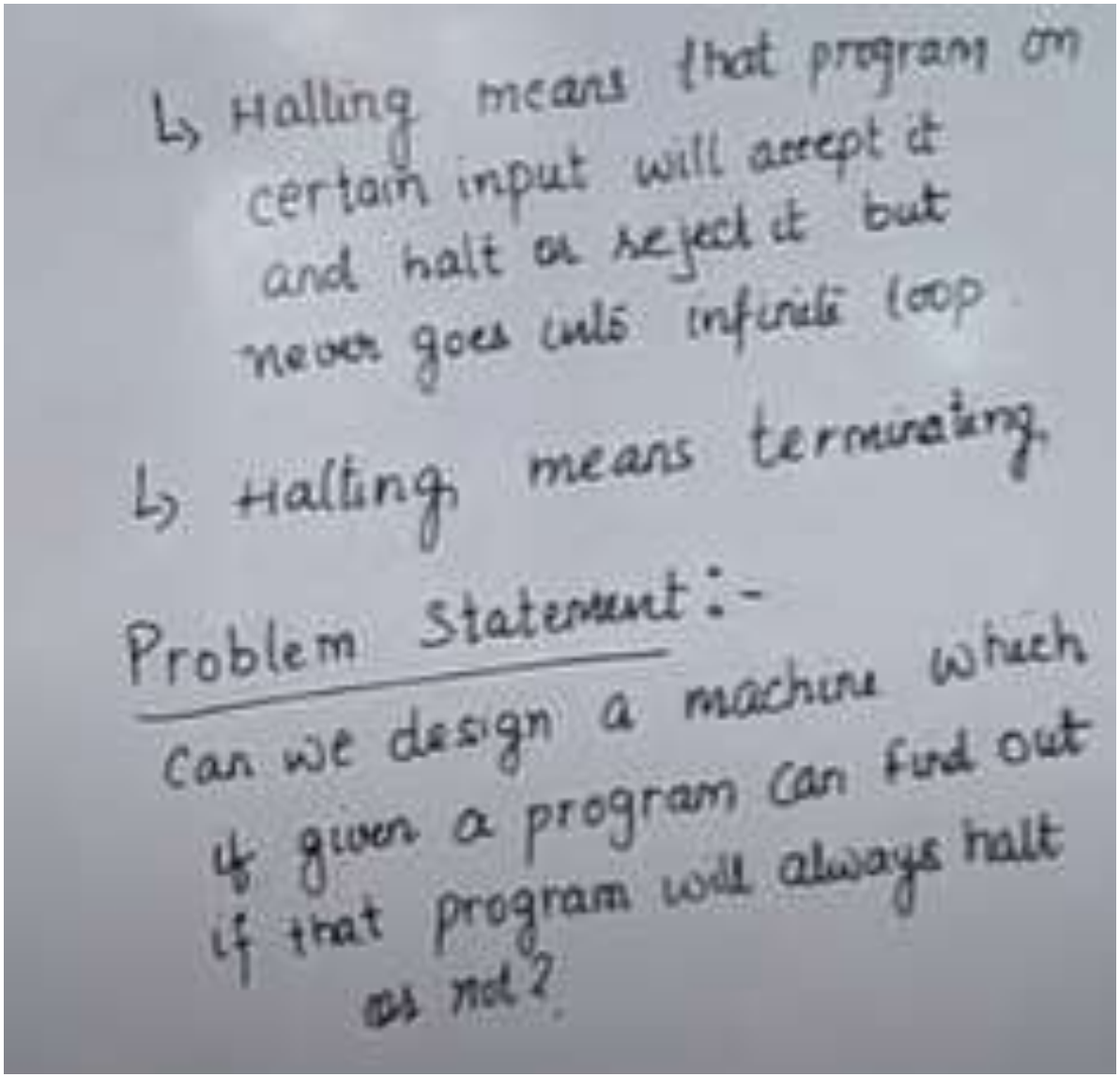
Find whether the list $M = (ab, bab, bbaa)$ &

$N = (a, ba, bab)$ have a PCP solution?

$$\left[\frac{ab}{a}\right]\left[\frac{bab}{ba}\right]\left[\frac{bbaa}{bab}\right]$$

- 2 1 3
- Un decidable

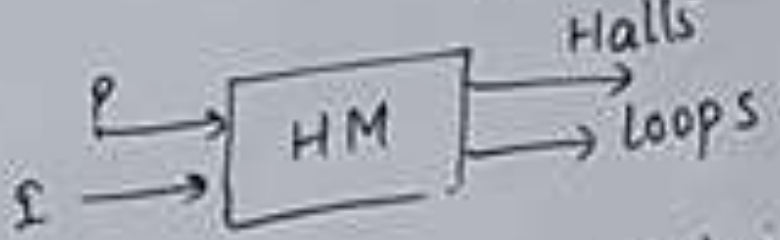
Halting problem



Solution:-

Let assume that we can design a machine HM(P, I)

Program Input



If HM is possible then we design another program CM(X)

```
HM(P,I)
{
  Halt
  or
  Not Halt
}
```

```
CM(X)
{
  if(HM(X,X)==Halt)
    loop forever;
  else
    return;
}
```

Church Turing Thesis

- What does computable mean?
- Meaning of term Computable was given by:
  1. Alonzo church (Lambda calculus)
  2. Allen Turing (Turing Machine)

- Any *algorithmic procedure* that can be carried out at all, by a human computer or a team of humans or an electronic computer, can be carried out by a TM. This statement usually referred to as Church's thesis, or the Church-Turing thesis.

Church Turing Thesis

- Here is an informal summary of some of the evidence.
  1. Humans normally work with a two-dimensional sheet of paper. A TM tape could be organized so as to simulate two dimensions; one likely consequence would be that the TM would require more moves to do what a human could do in one.
  2. Various enhancements of the TM model have been suggested to make the operation more like that of a human computer, or more convenient & efficient. The multitape TM is an example.
  3. Other theoretical models includes abstract machines with two stacks or with a queue.
  4. Since the introduction of the Turing machine, no one has suggested any type of computation that ought to be included in the category of "*algorithmic procedure*" and cannot be implemented on a TM.