Turing Machines

By:
Dr. Sandeep Rathor

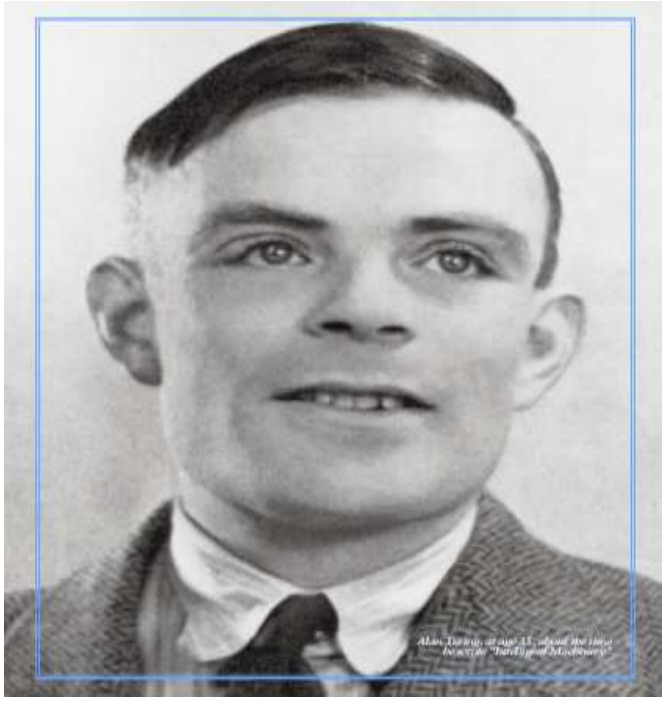
Syllabus or Contents

Turing machines (TM):

- Basic Model, Definition and Representation,
- TM for Computing Integer Functions,
- Variants of Turing Machine and their equivalence,
- Universal TM,
- Church's Thesis,
- Recursive and Recursively Enumerable Languages,
- Halting Problem,
- Introduction to Computational Complexity

Foundation

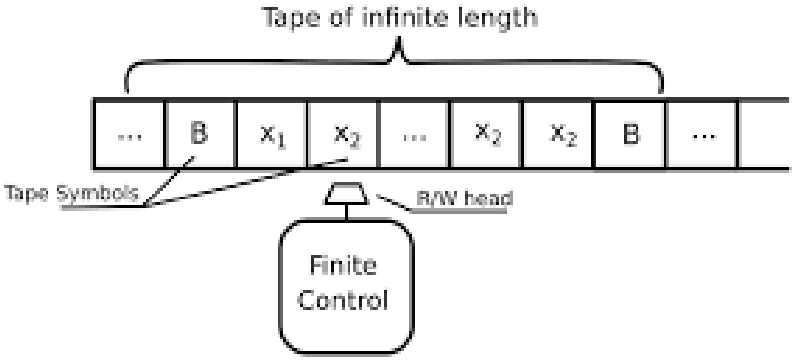
The theory of computation and the practical application it made possible — the computer — was developed by an Englishman called Alan Turing.



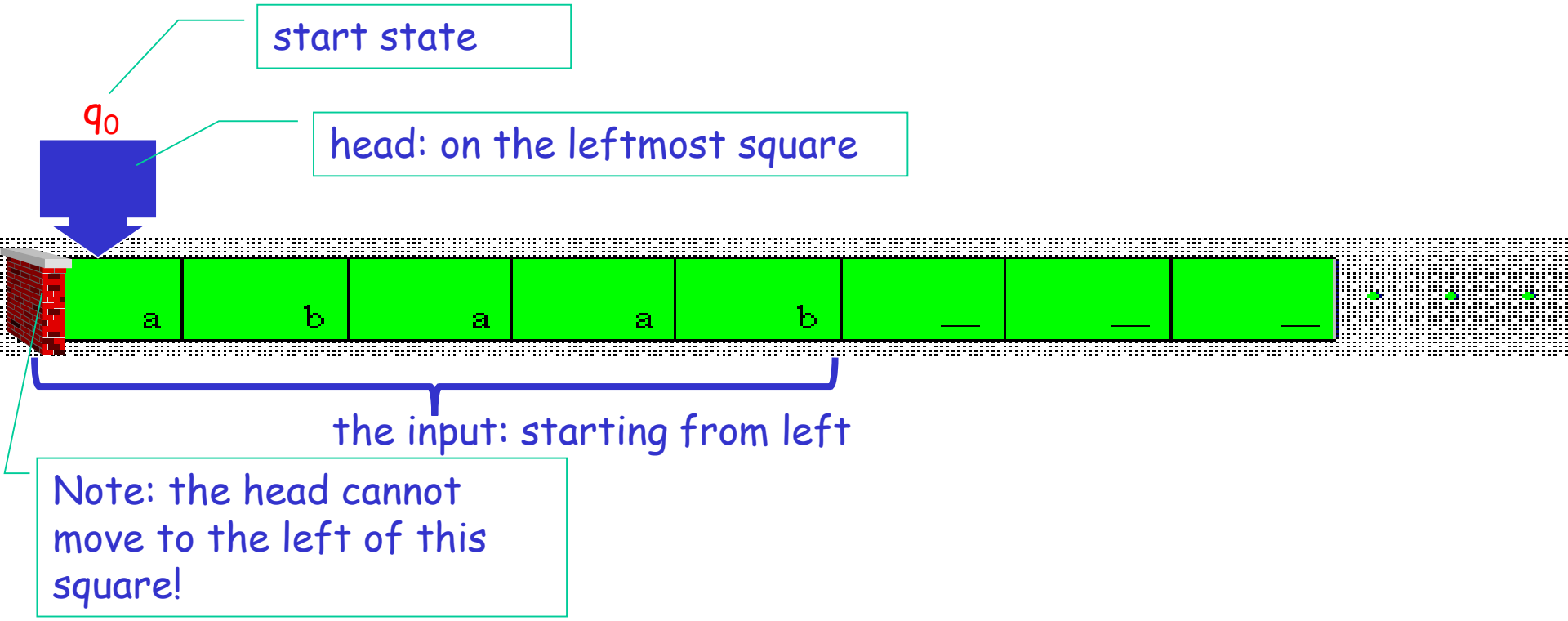
Turing Machine

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

- Q - The finite set of states of the finite control
- Σ - The finite set of input symbols
- Γ- The set of tape symbols; $\Sigma \subset \Gamma$
- δ - $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
- q_0 – The initial state
- B or # or ◊ - Blank Symbol; $B \in \Gamma$ but $B \notin \Sigma$
- F – Accept & Halt State; $F \in Q$

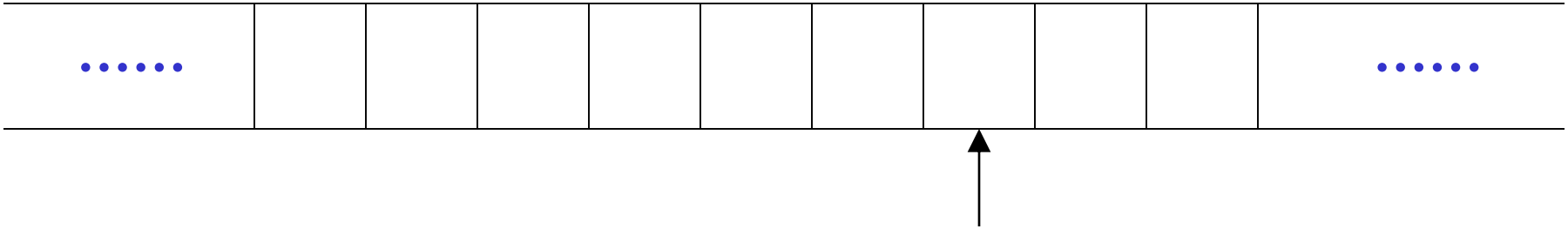


Computations: The Start Configuration



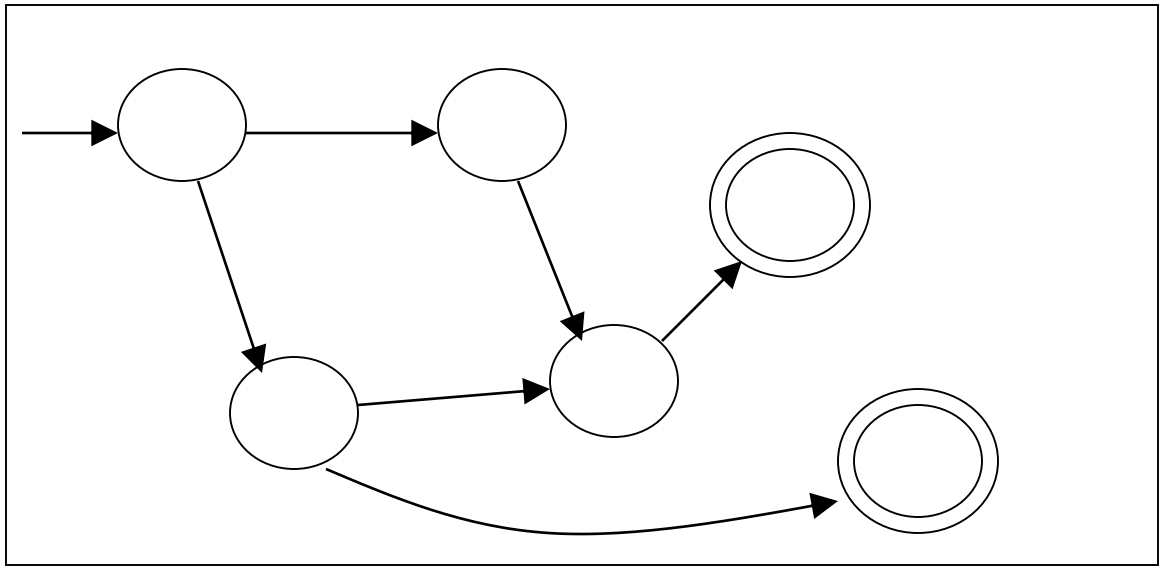
A Turing Machine

Tape



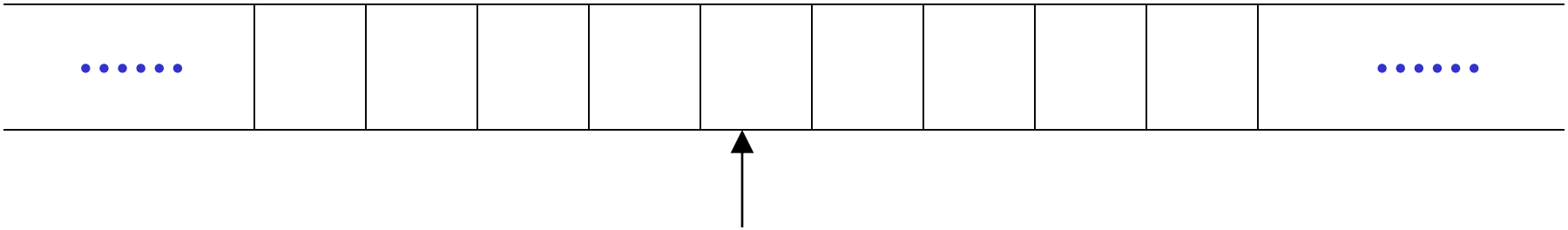
Read-Write head

Control Unit



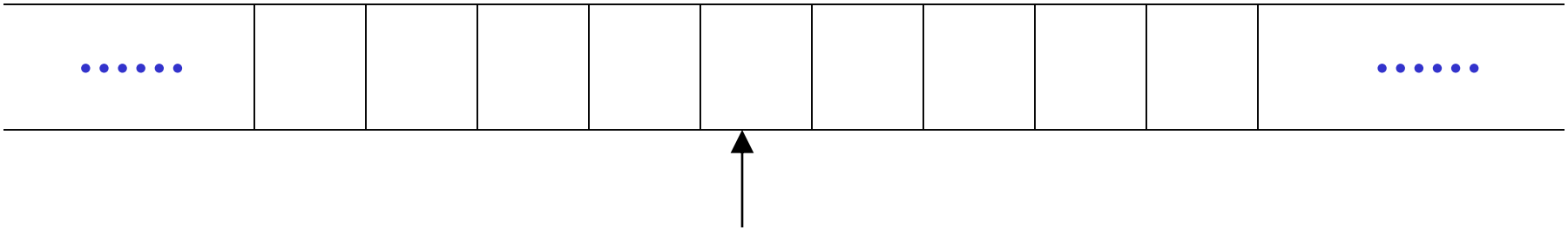
The Tape

No boundaries -- infinite length



Read-Write head

The head moves Left or Right



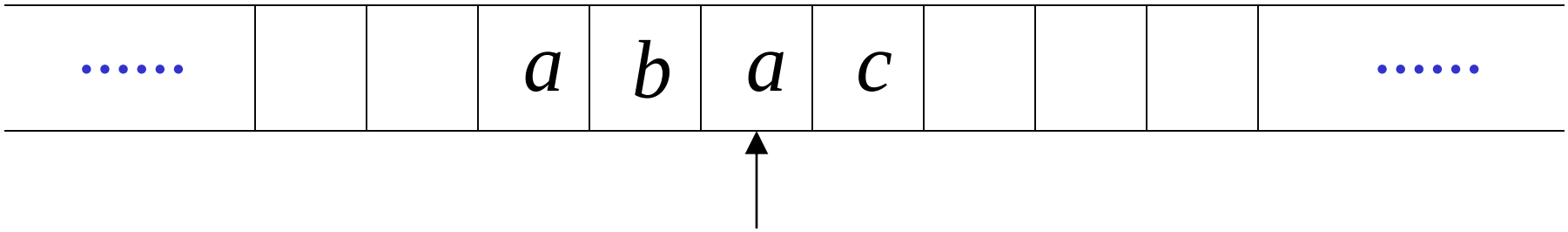
Read-Write head

The head at each time step:

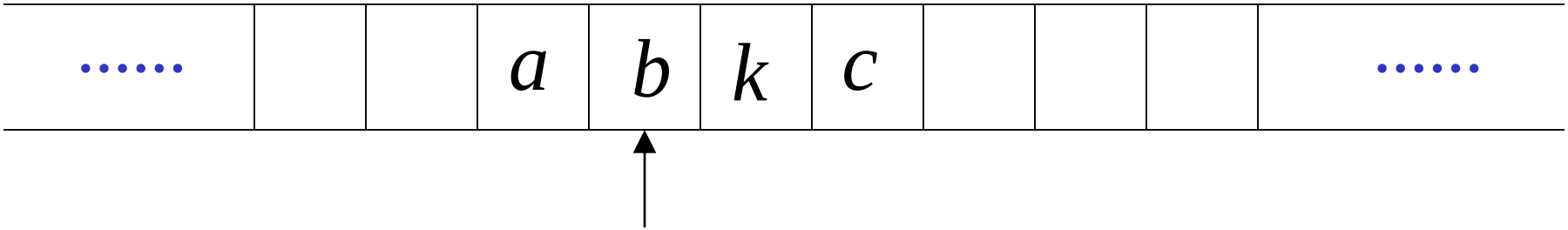
1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

Example:

Time 0

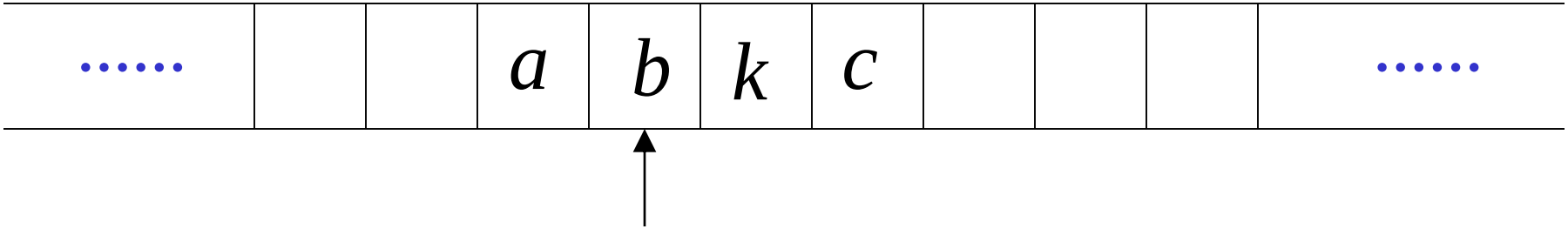


Time 1

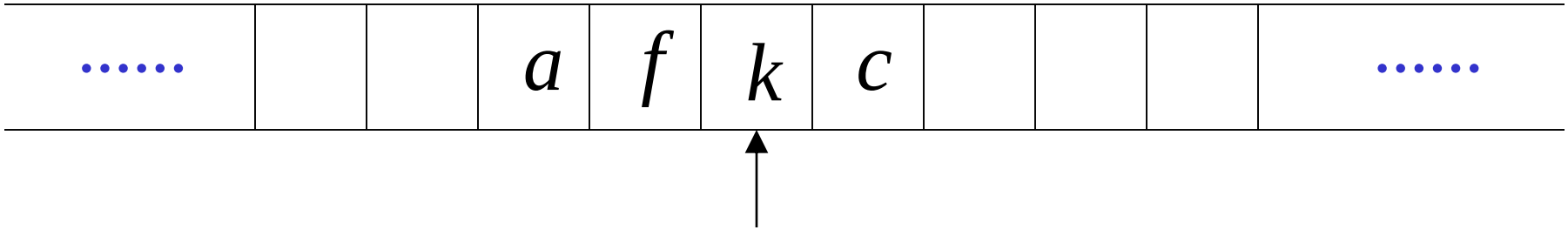


1. Reads a
2. Writes k
3. Moves Left

Time 1

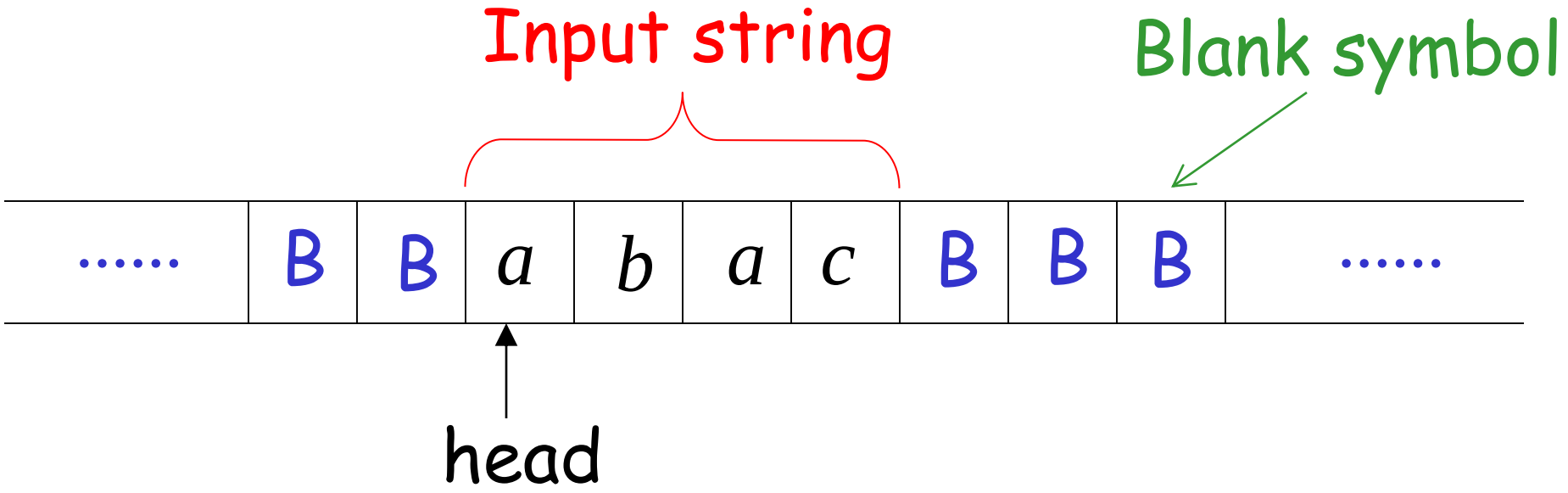


Time 2

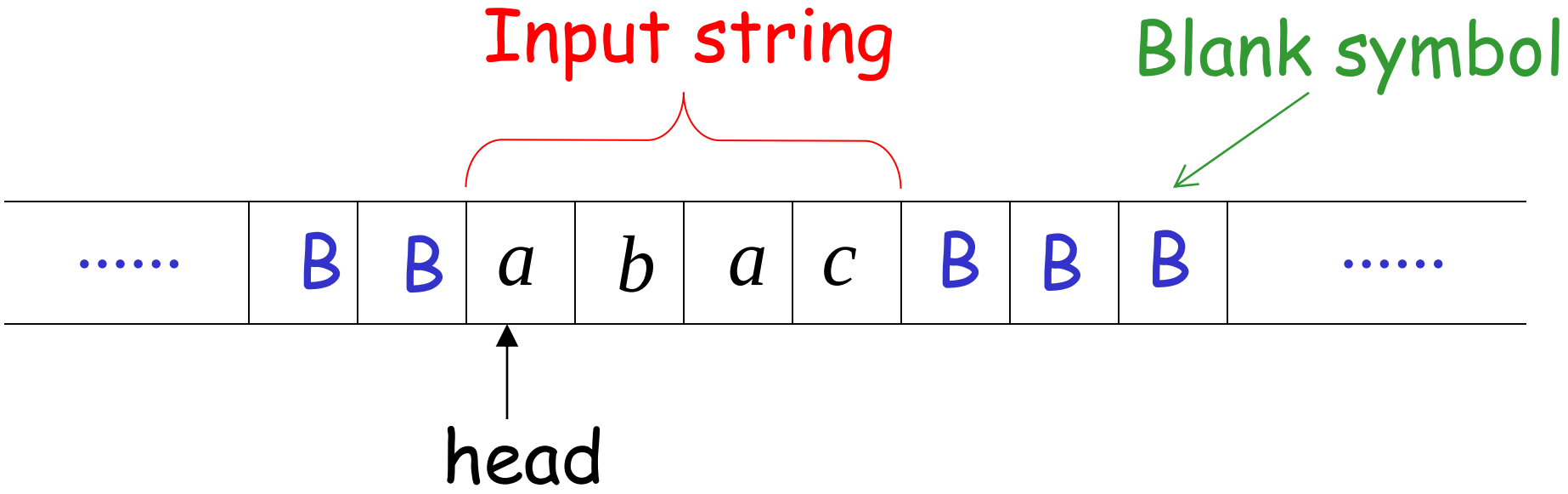


1. Reads b
2. Writes f
3. Moves Right

The Input String

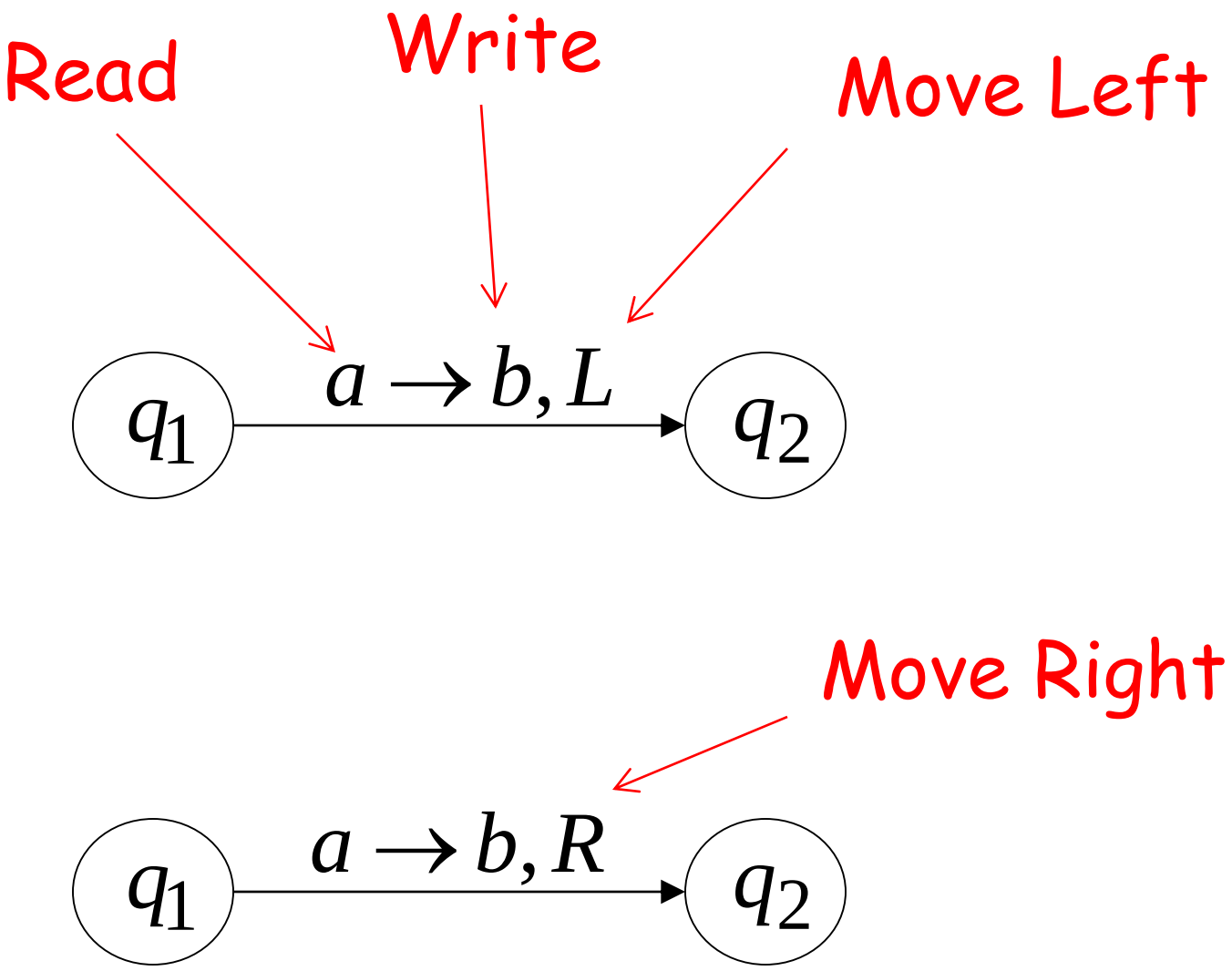


Head starts at the leftmost position of the input string



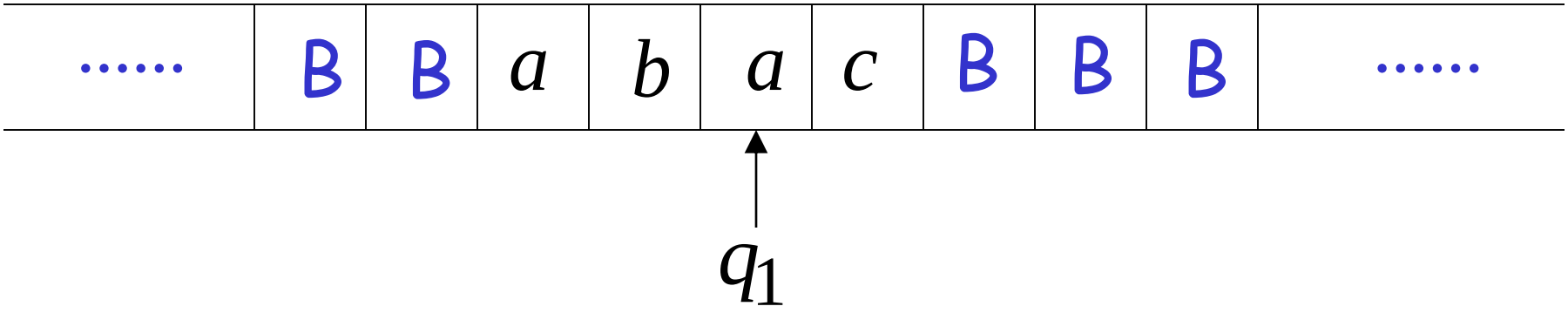
Remark: the input string is never empty

States & Transitions

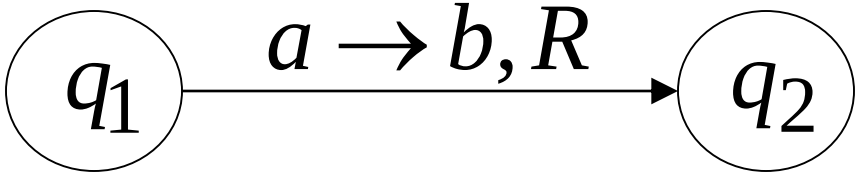


Example:

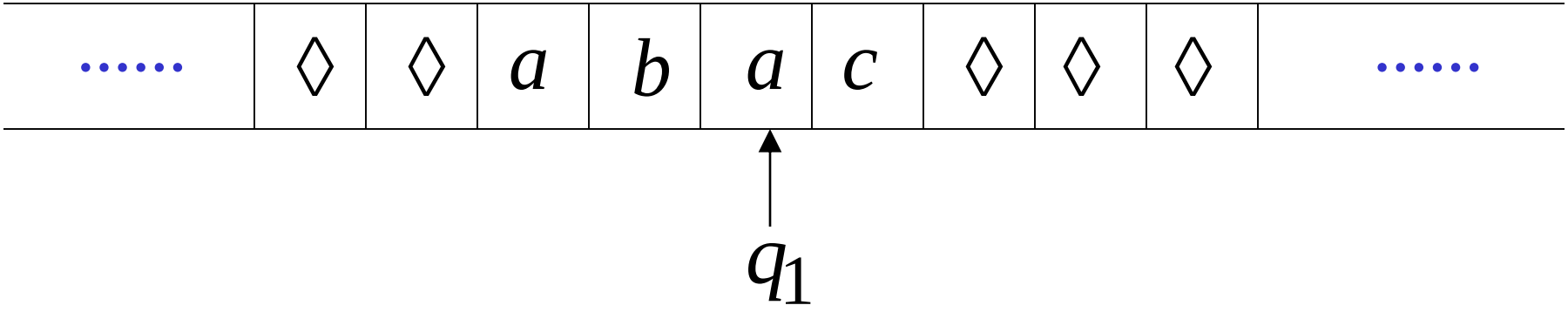
Time 1



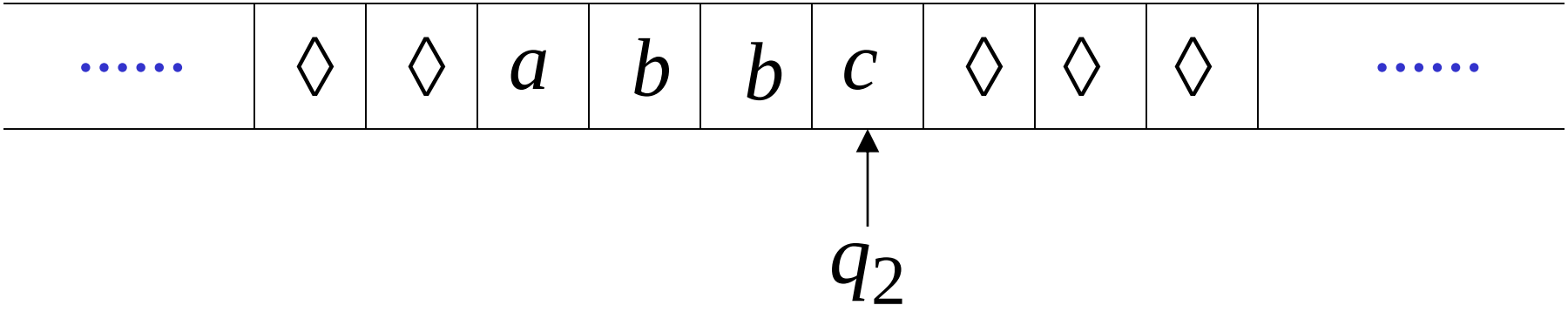
current state

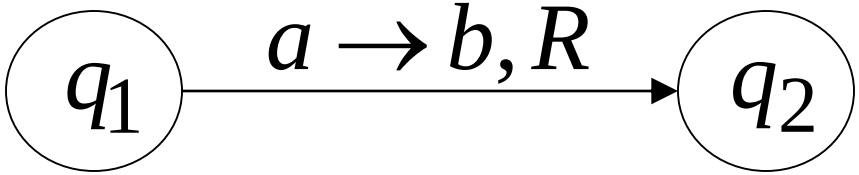


Time 1

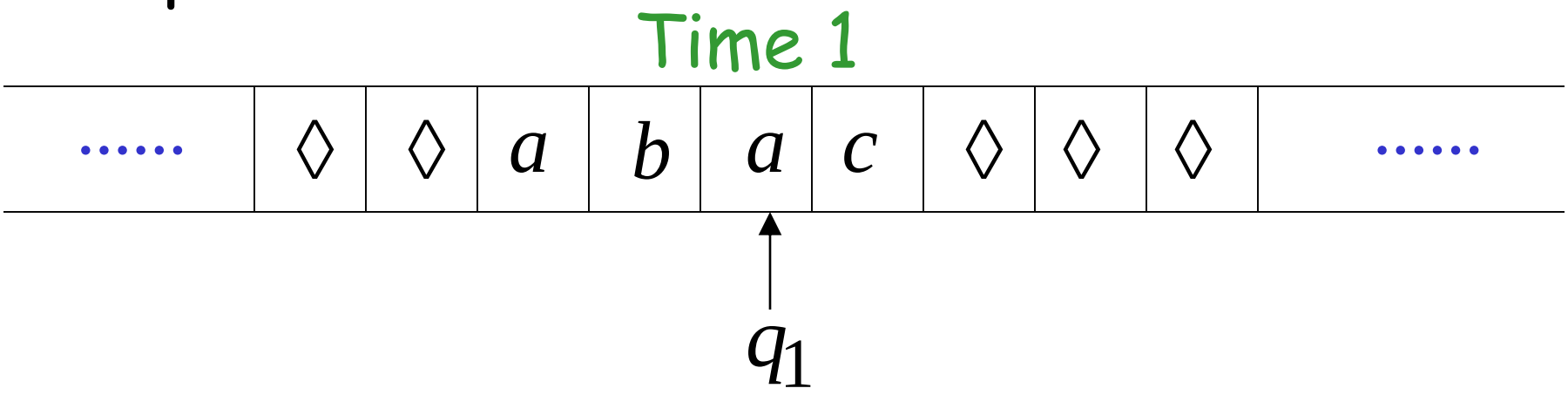


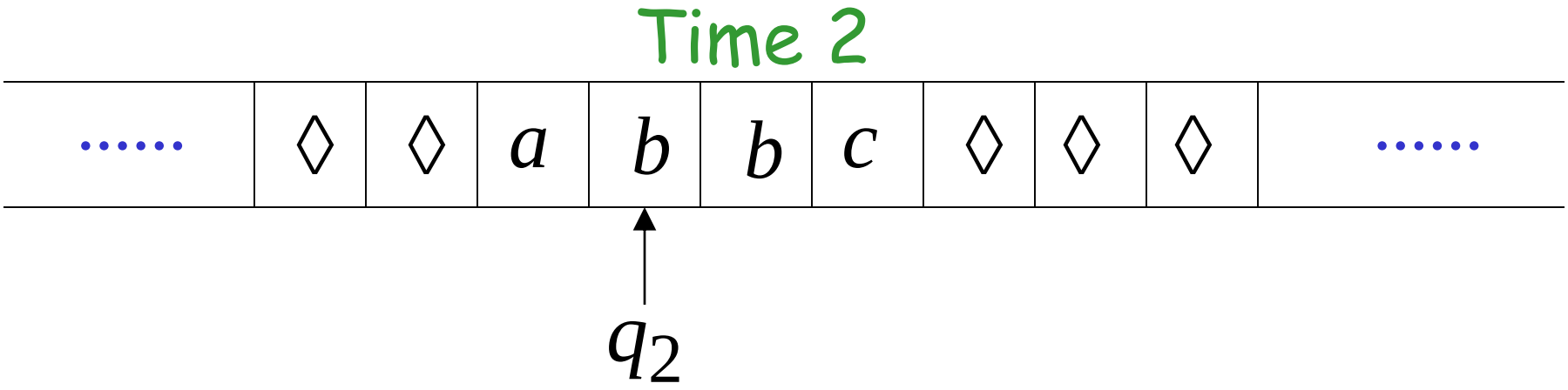
Time 2

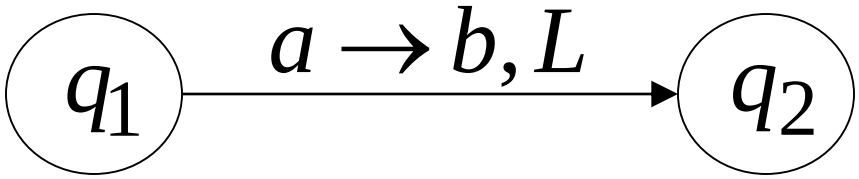




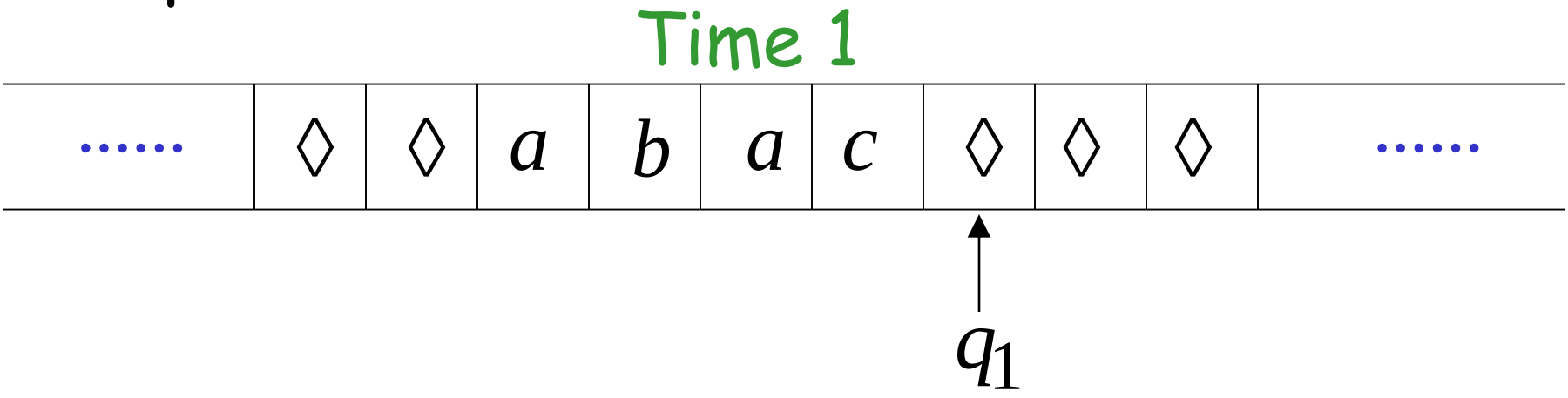
Example:

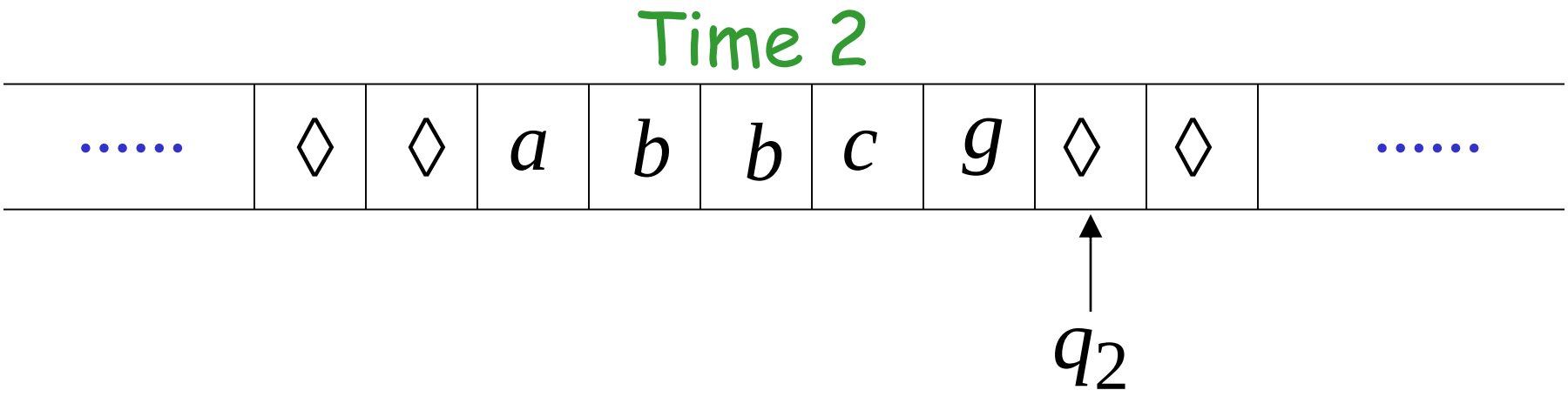


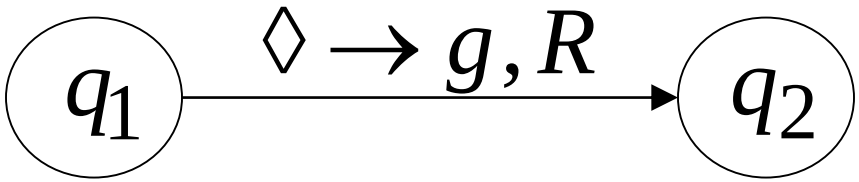




Example:



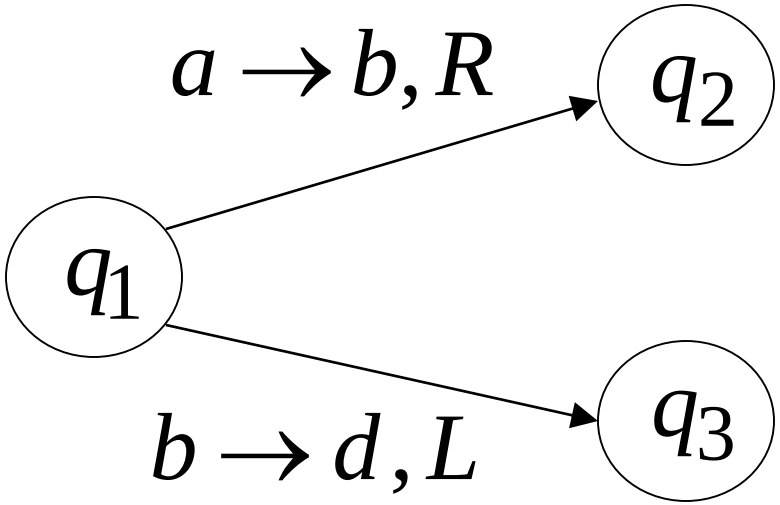




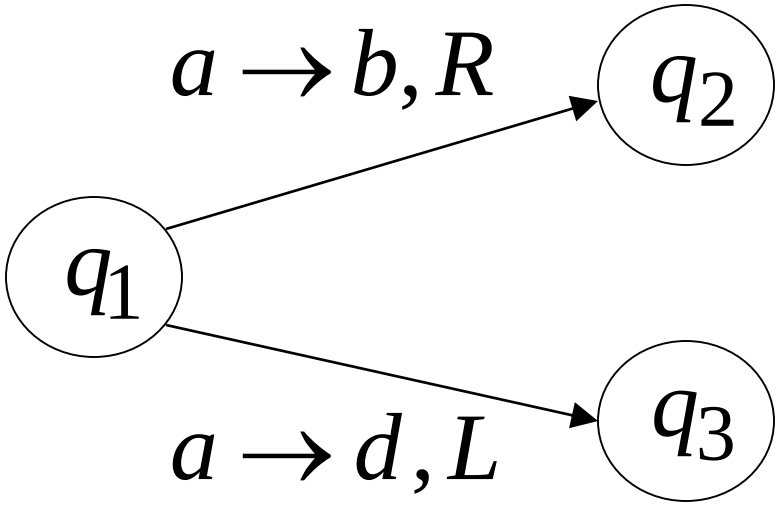
Determinism

Turing Machines are deterministic

Allowed



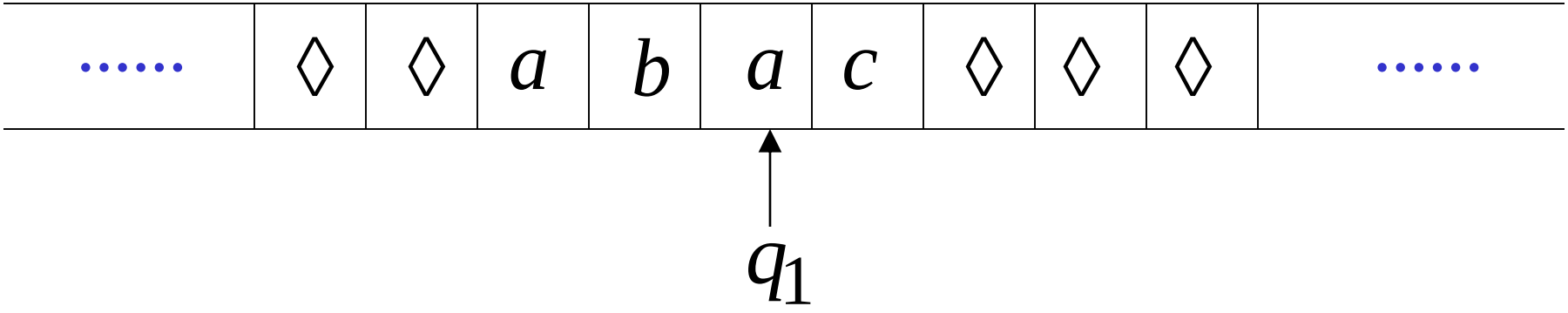
Not Allowed

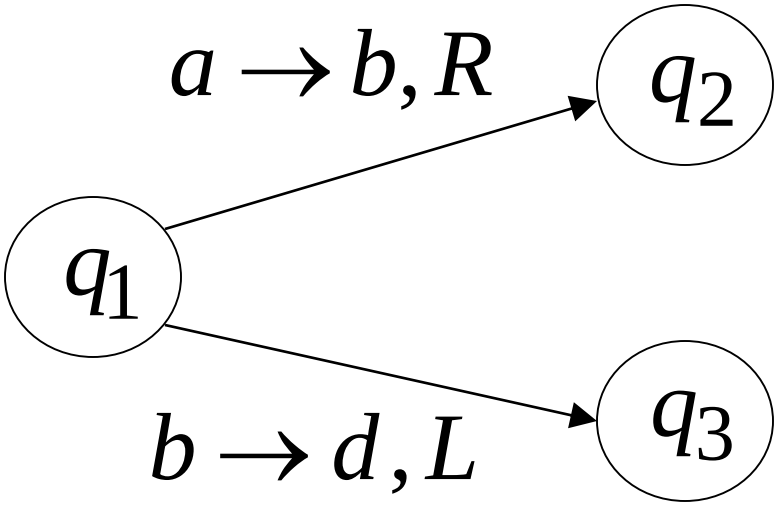


No lambda transitions allowed

Partial Transition Function

Example:





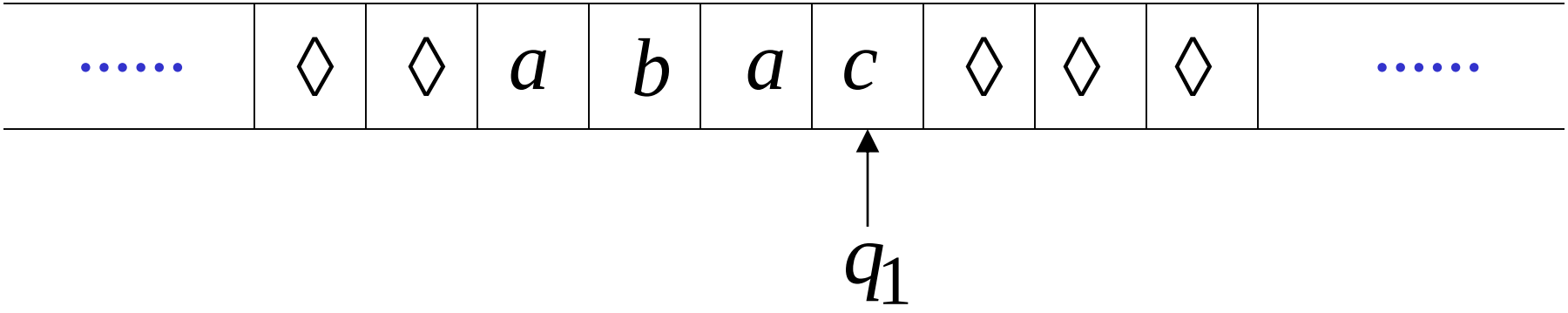
Allowed:

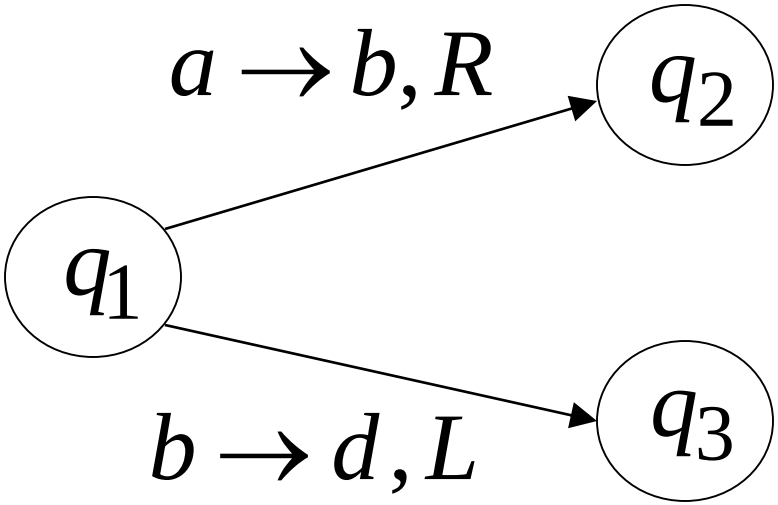
No transition
for input symbol c

Halting

The machine ***halts*** if there are no possible transitions to follow

Example:

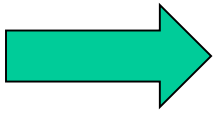


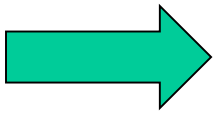


No possible transition

HALT!!!

Acceptance

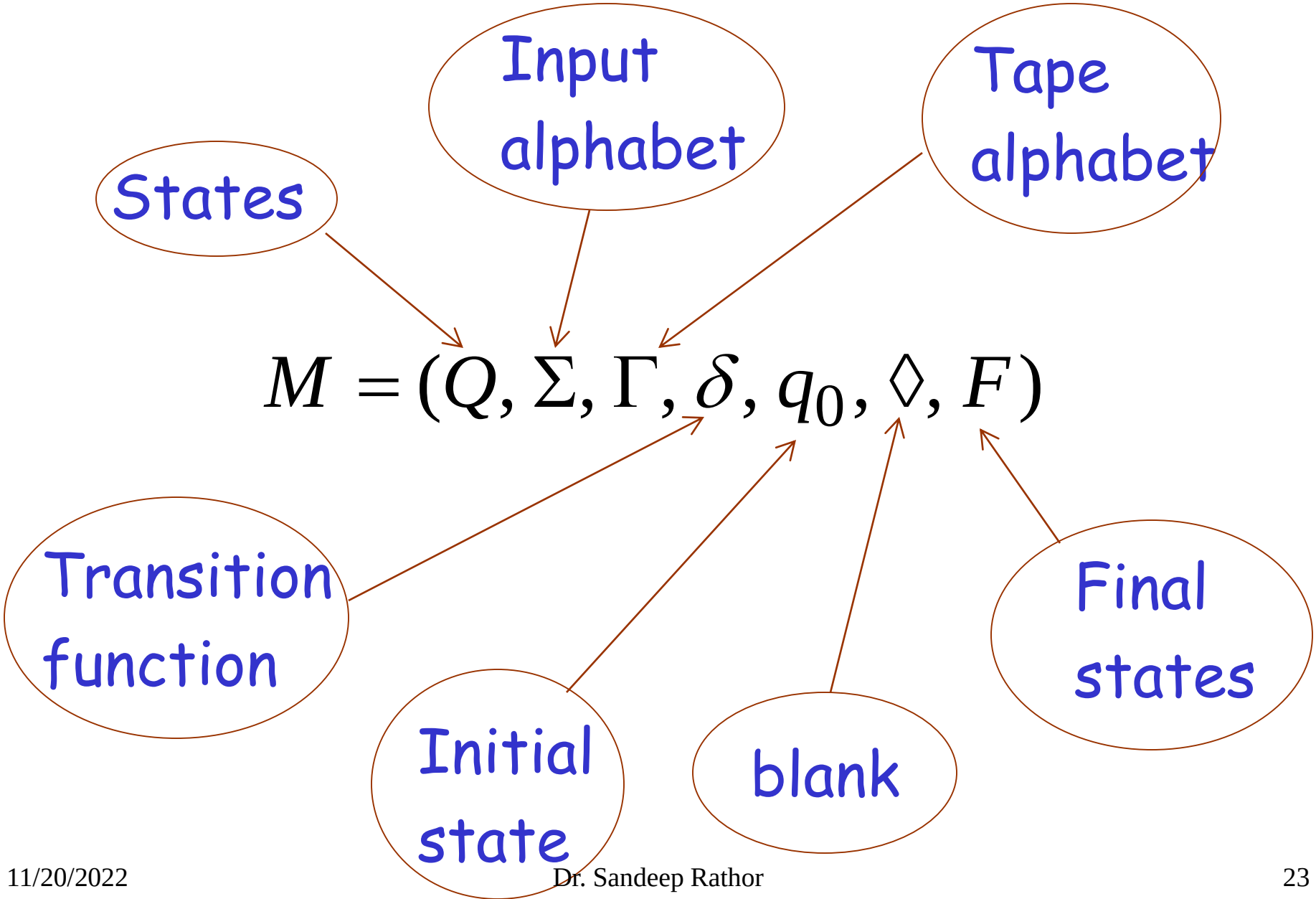
Accept Input  If machine halts in a final state

Reject Input  If machine halts in a non-final state

or

If machine enters an *infinite loop*

Turing Machine:



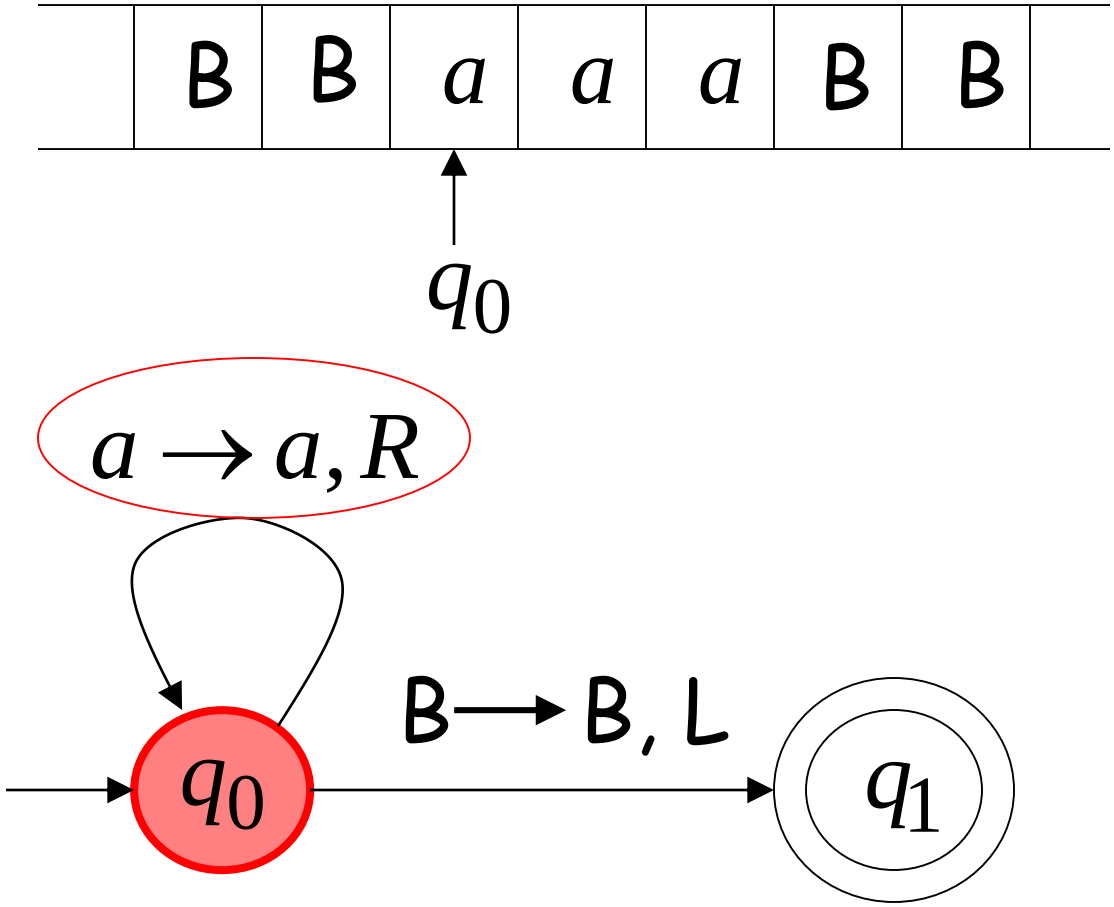
Designing of Turing Machine: Basics

By:
Dr. Sandeep Rathor

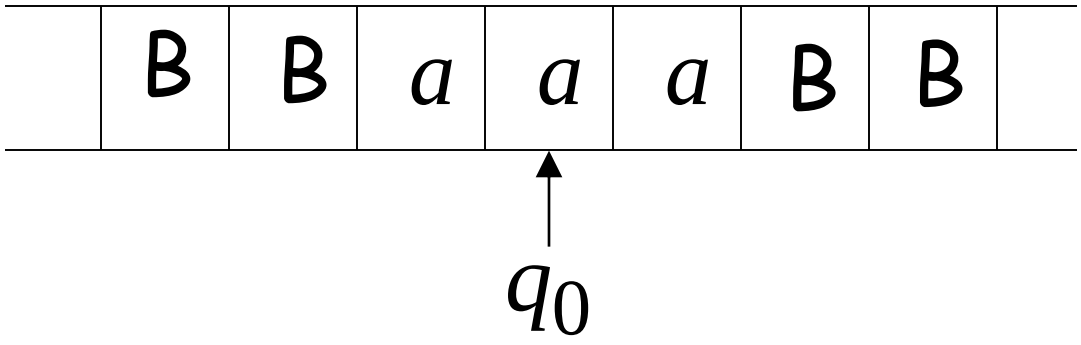
Turing Machine Example

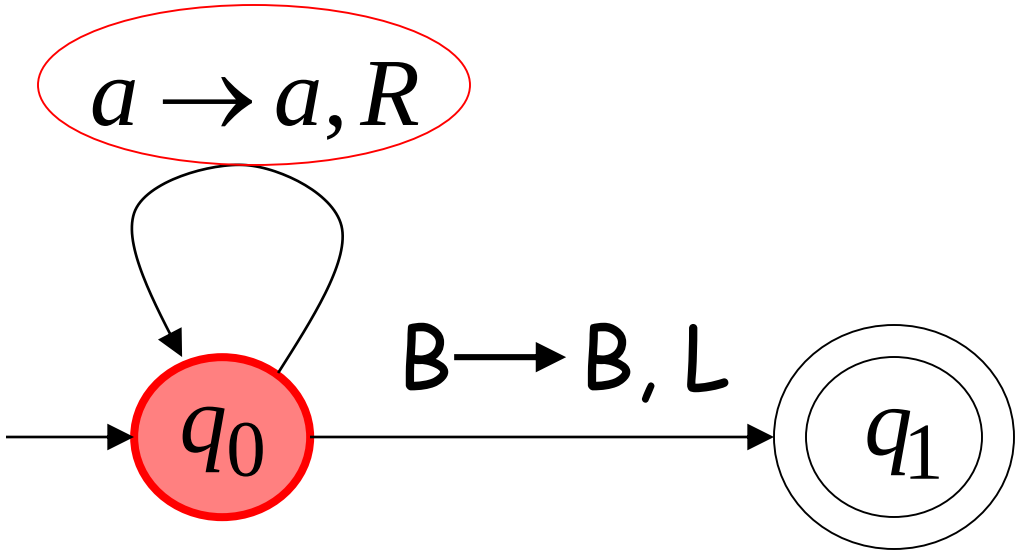
A Turing machine that accepts the language: a^*

Time 0

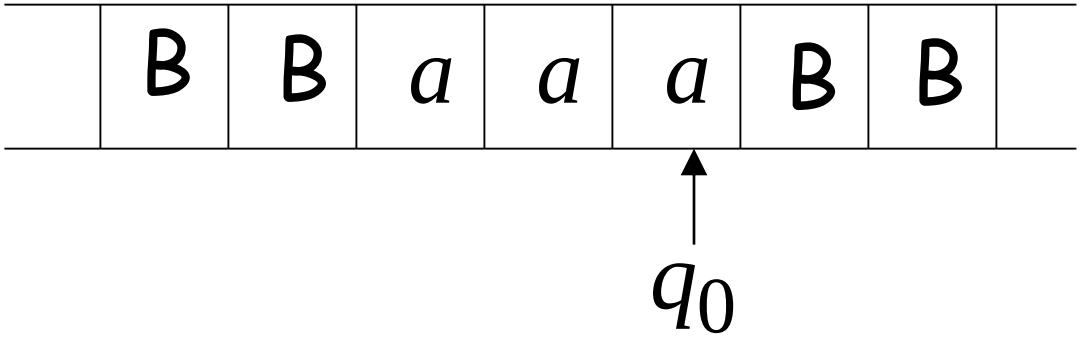


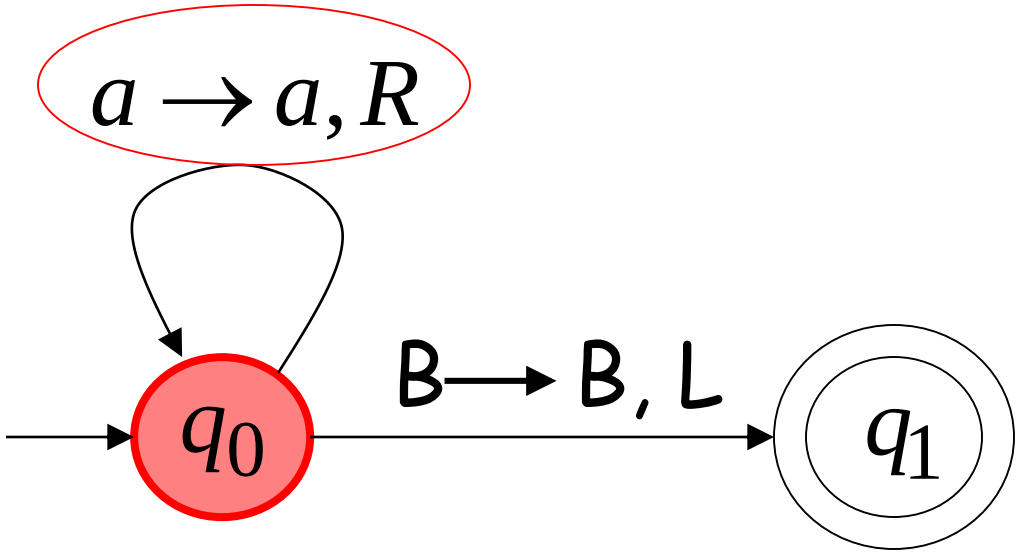
Time 1



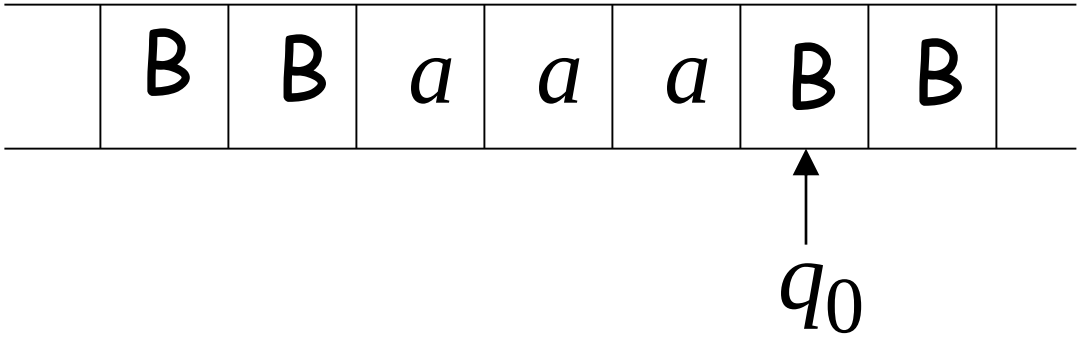


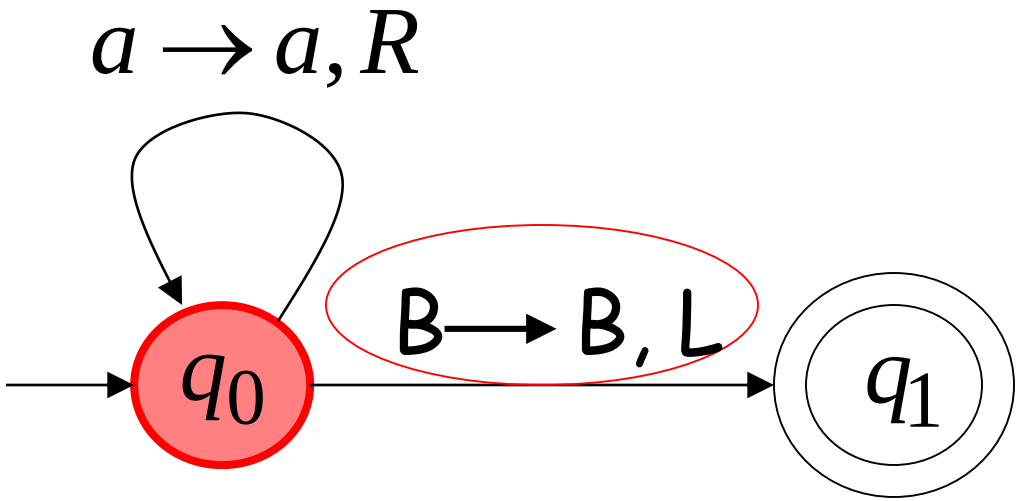
Time 2



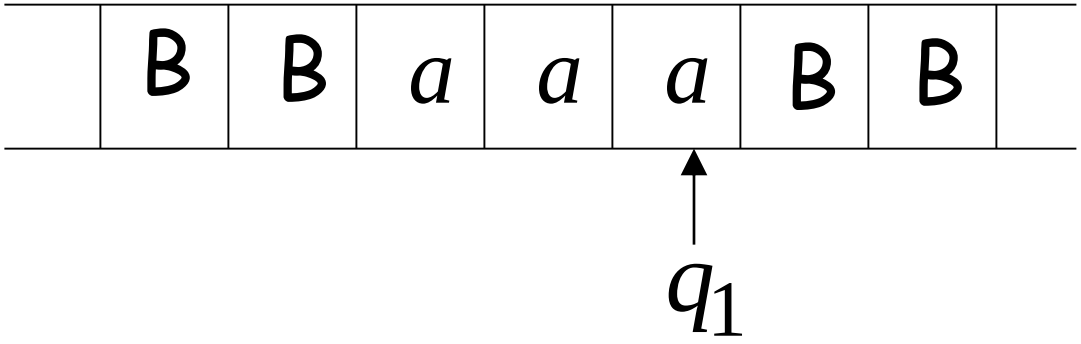


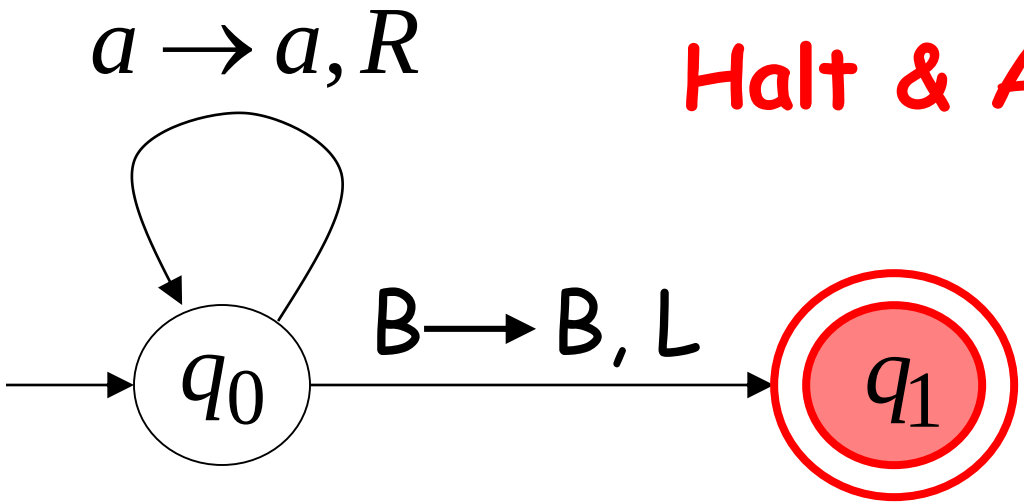
Time 3





Time 4

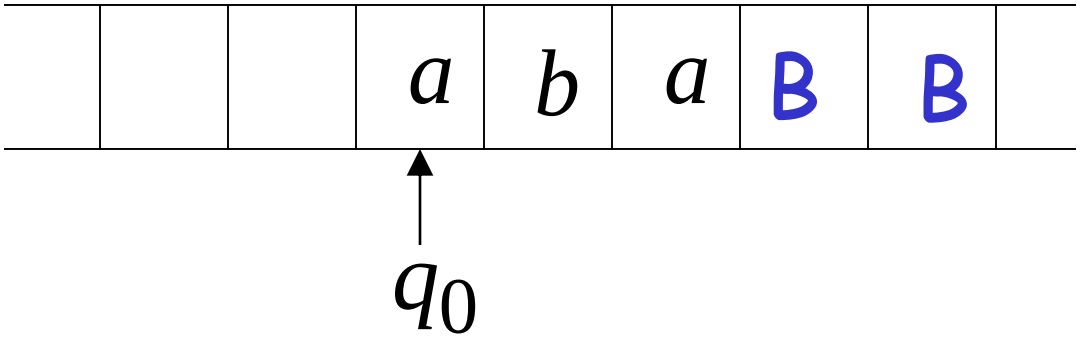


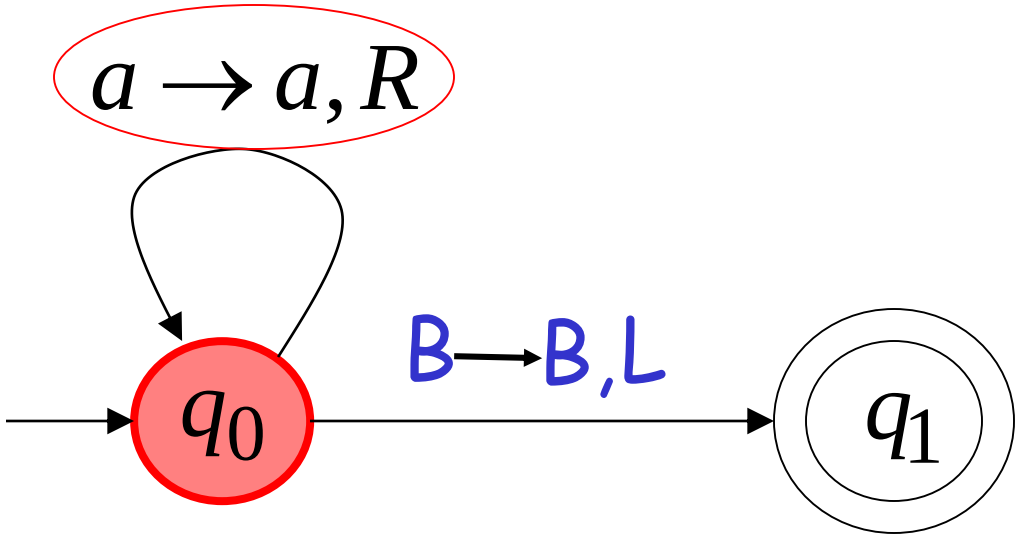


Halt & Accept

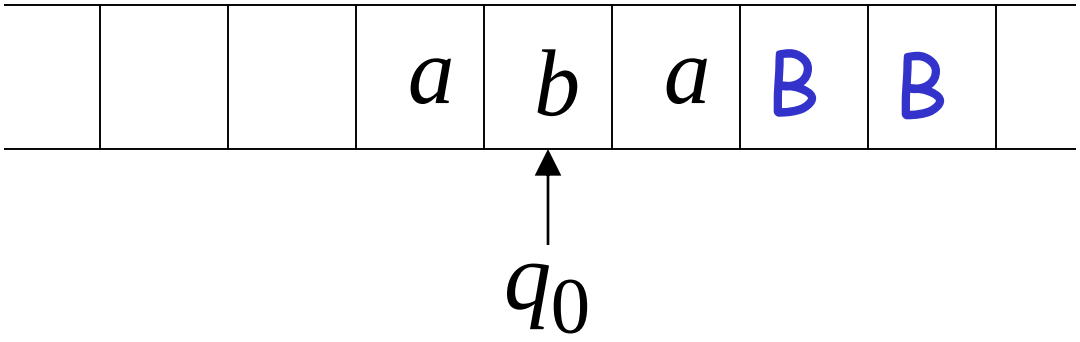
Rejection Example

Time 0



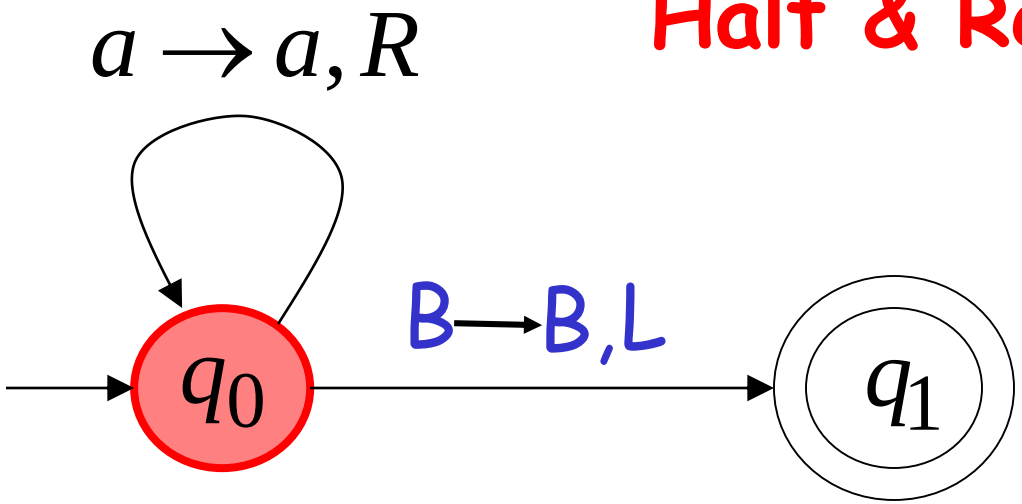


Time 1



No possible Transition

Halt & Reject



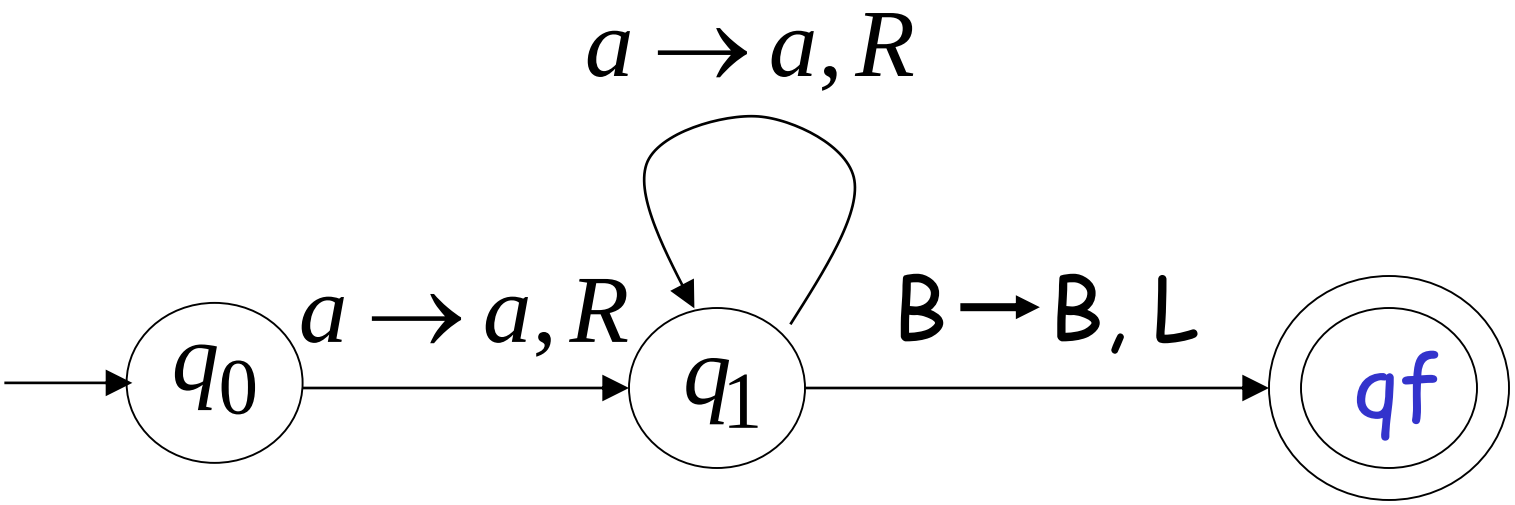
Because of the infinite loop:

- The final state cannot be reached
- The machine never halts
- The input is not accepted

Turing Machine Example

A Turing machine that accepts the language:

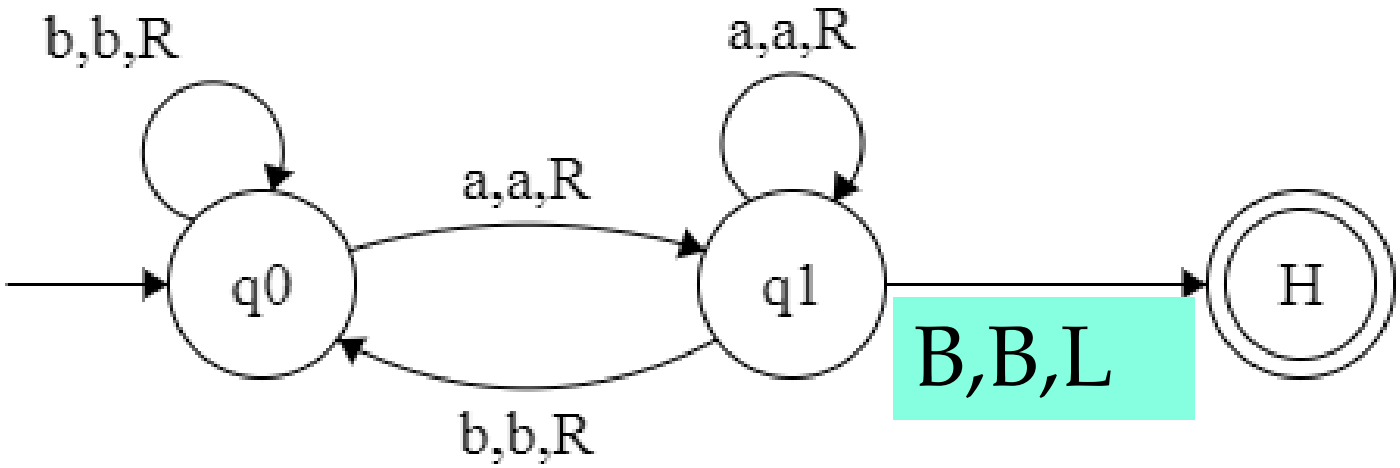
$$aa*$$



Construction of Turing Machines

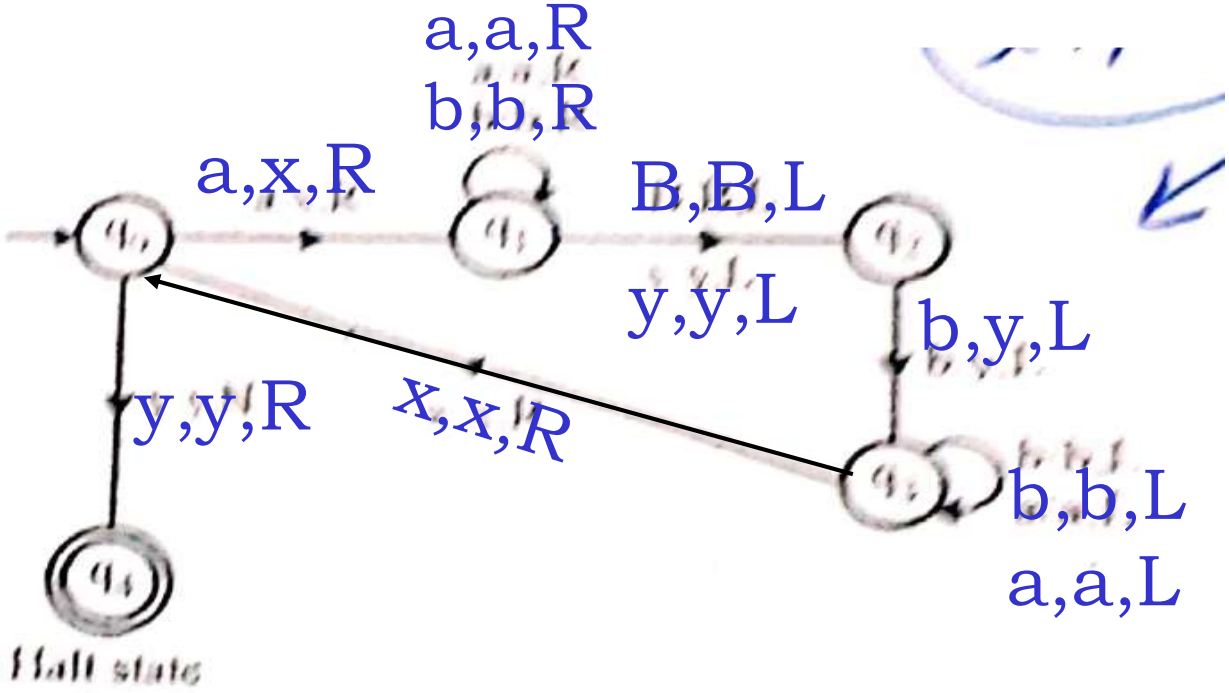
By:
Dr. Sandeep Rathor

Design a TM to recognize L = {w | w ends with a}



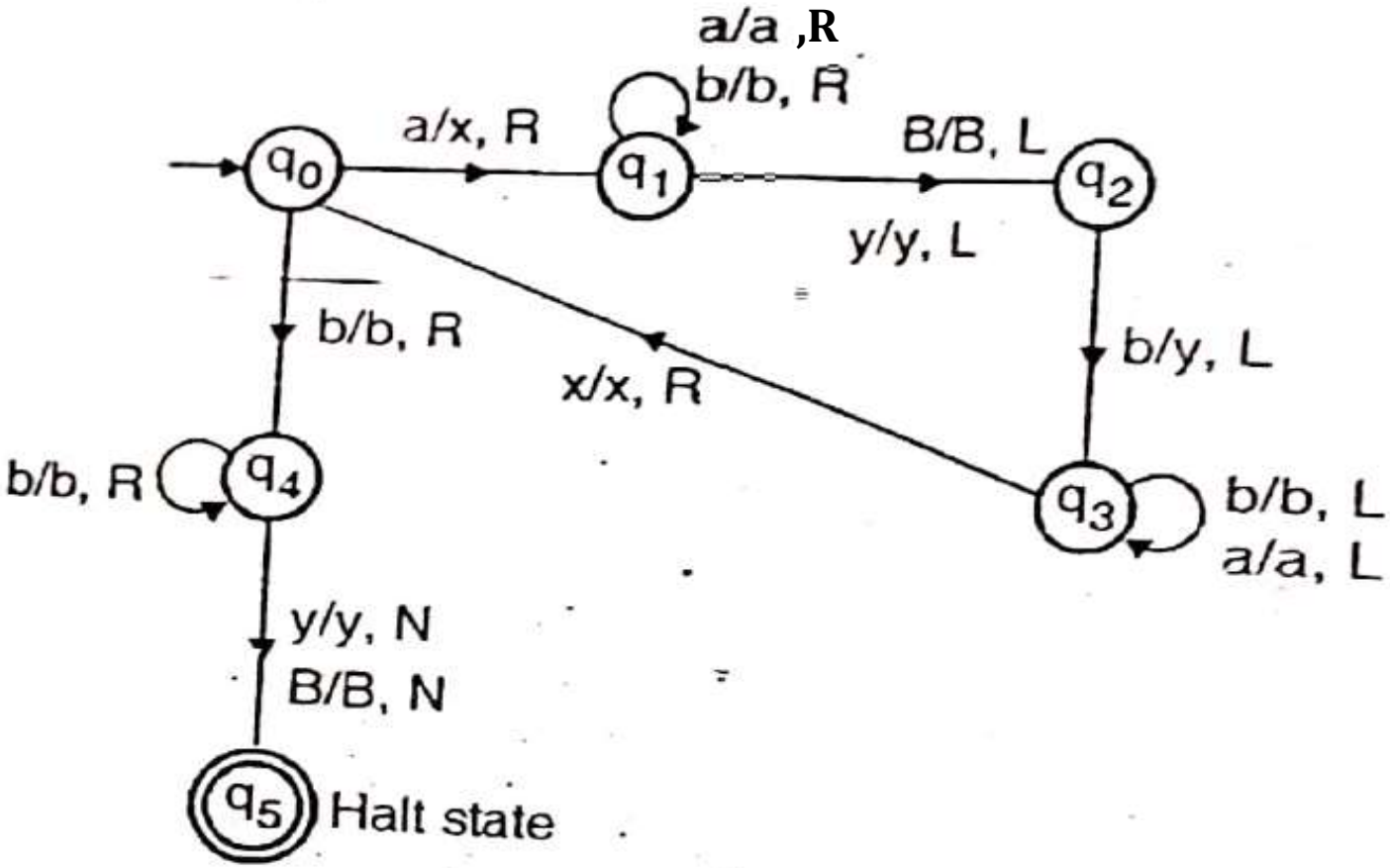
a	a	b	a	B	B	…

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

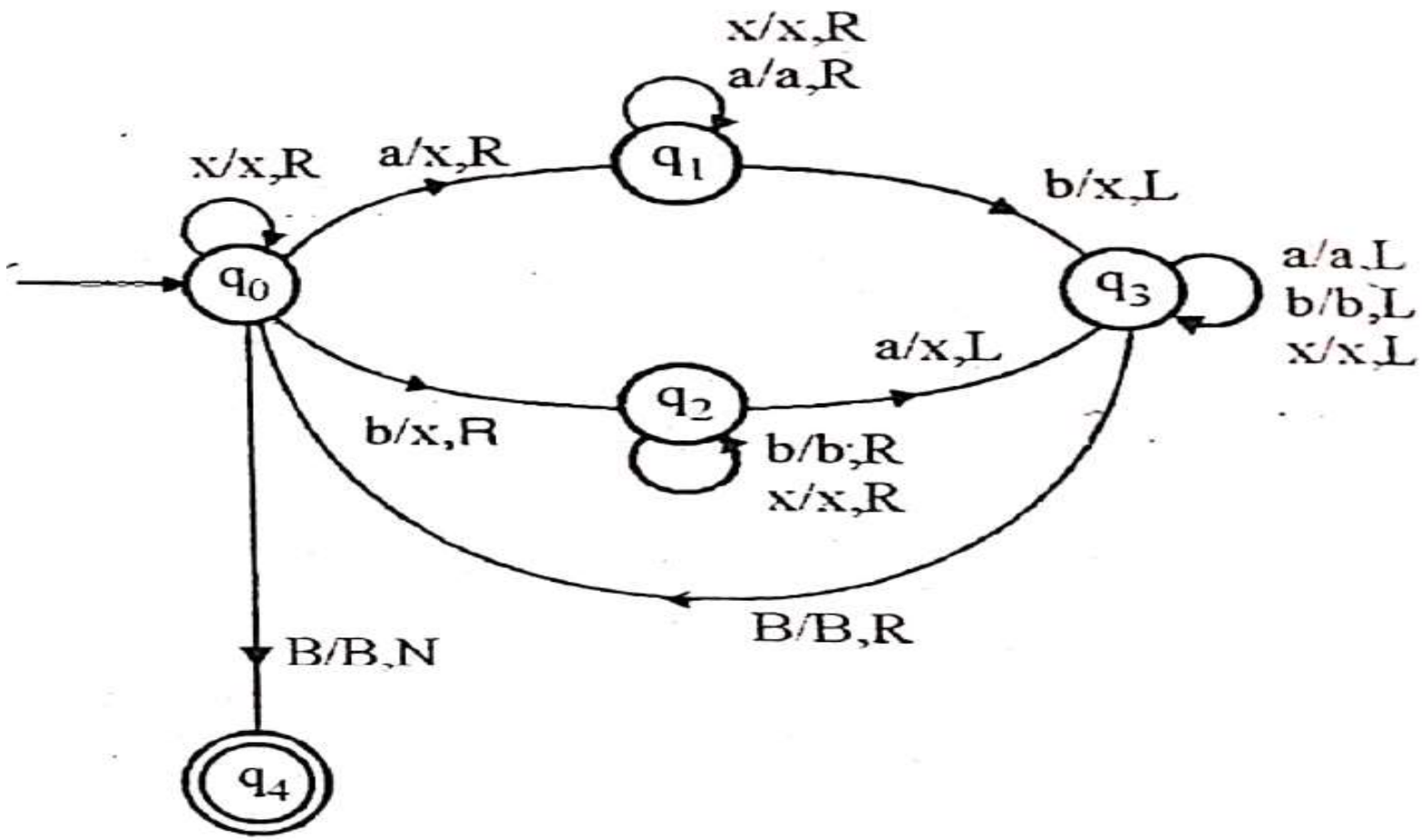


	a	b	x	y	B
$\rightarrow q_0$	(q_1, x, R)	–	–	(q_4, y, N)	–
q_1	(q_1, a, R)	(q_1, b, R)	–	(q_2, y, L)	(q_2, B, L)
q_2	–	(q_3, y, L)	–	–	–
q_3	(q_3, a, L)	(q_3, b, L)	(q_0, x, R)	–	–
$*q_4$	q_4	q_4	q_4	q_4	q_4

$L = \{ a^n b^m \mid n < m \}$


a/a ,R
b/b, R
a/x, R
B/B, L
q0
q1
q2
y/y, L
b/b, R
b/y, L
x/x, R
b/b, R
q4
q3
b/b, L
a/a, L
y/y, N
B/B, N
q5
Halt state

Equal no of a's and b's over (a,b)



Construction of Turing Machines

By:
Dr. Sandeep Rathor

Building a TM for a Simple Language

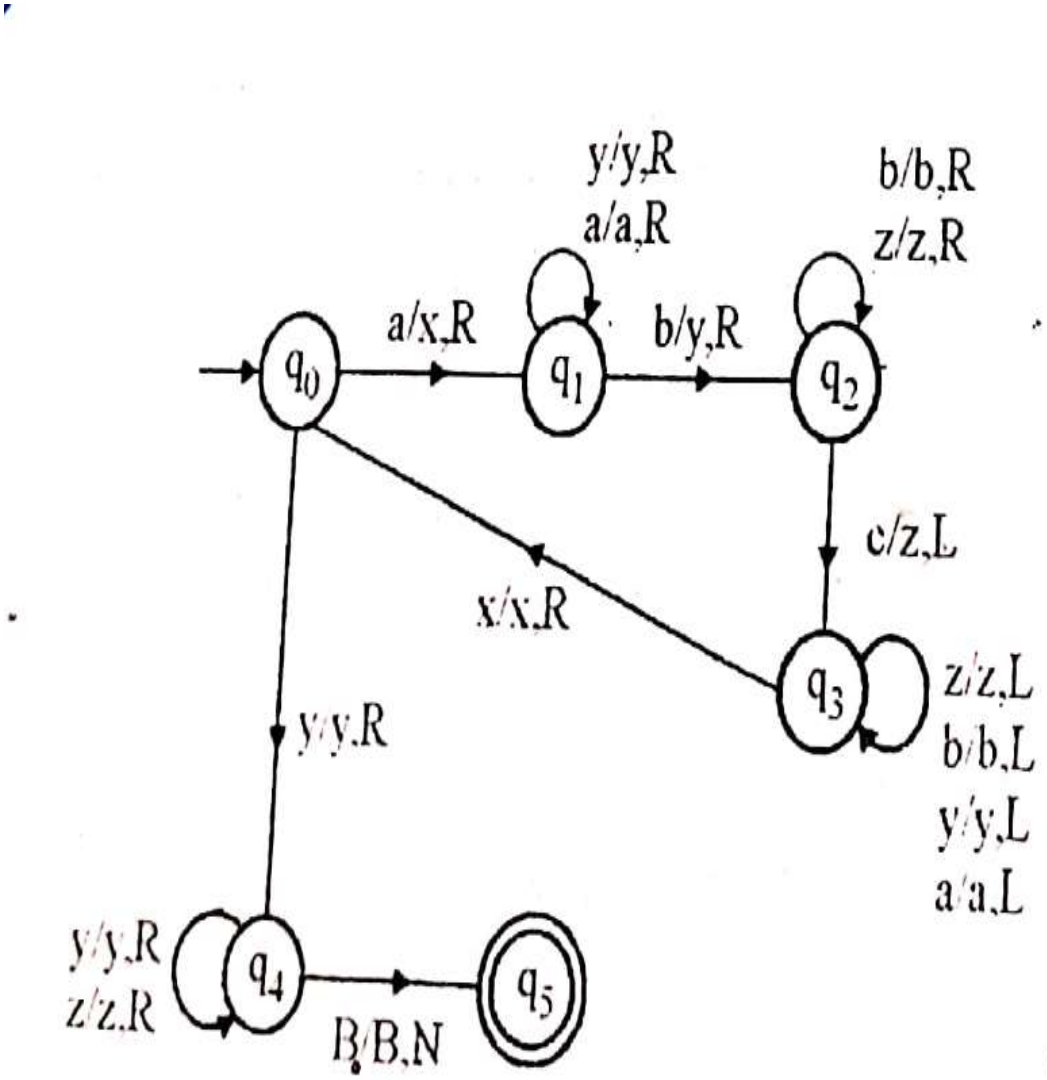
$$L = \{ a^n b^n c^n \mid n > 0 \}$$

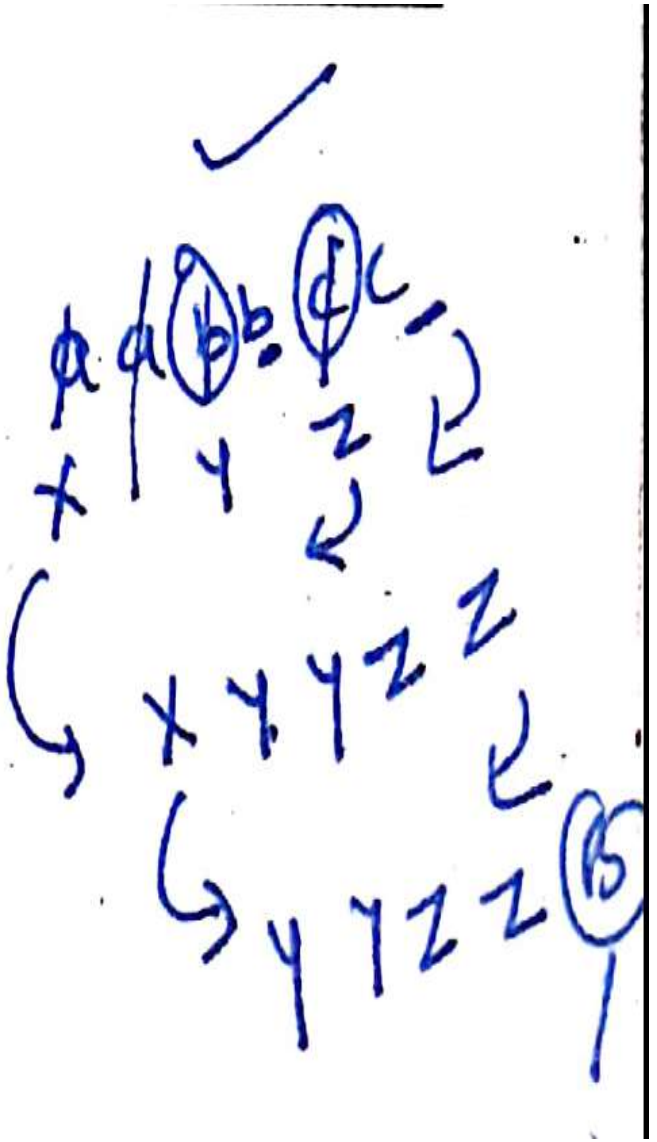
Examples:

Member of L: aaabbbccc

Non-Member of L: aaabbcccc

$L = \{ a^n b^n c^n \mid n > 0 \}$



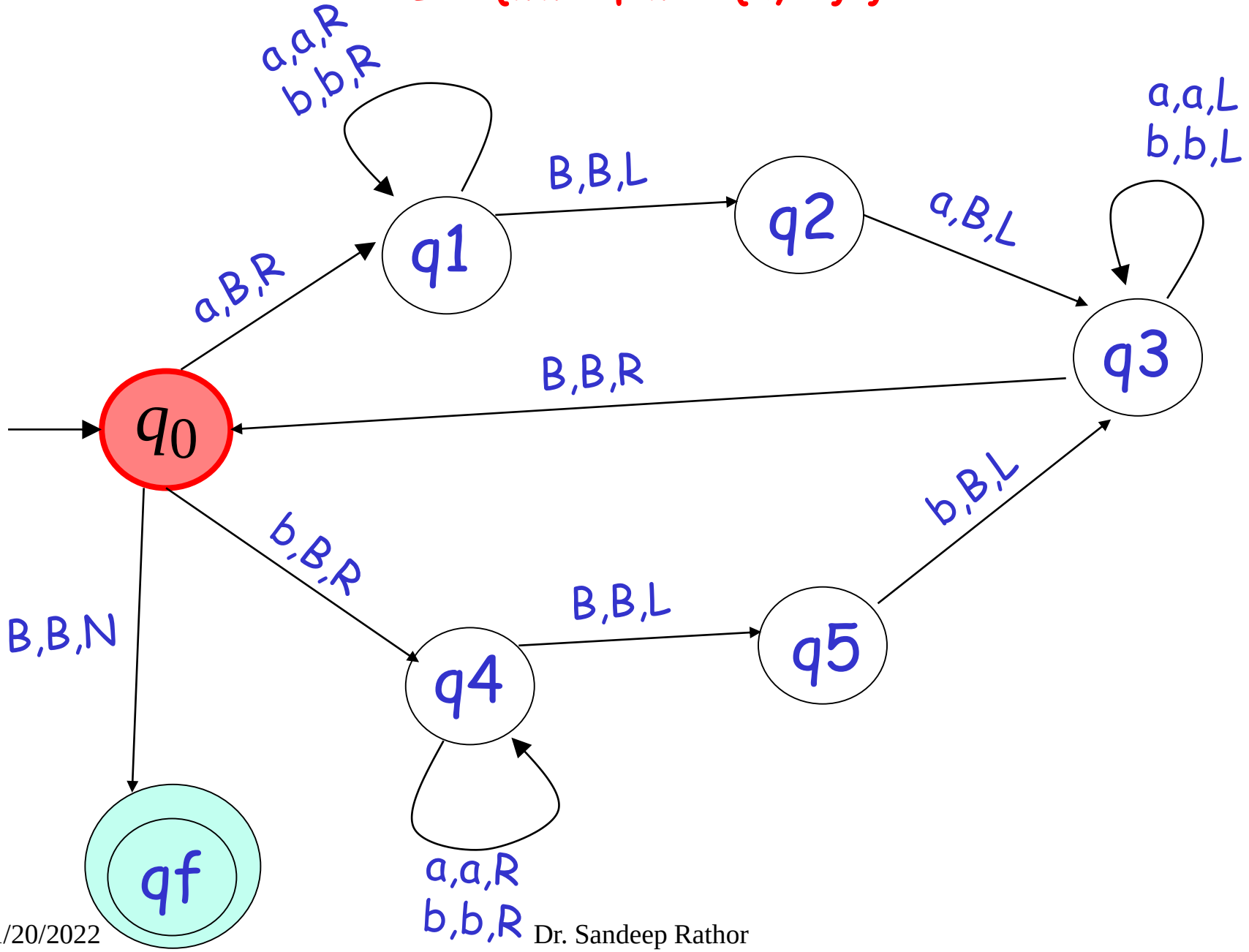


Construction of Turing Machines

By:
Dr. Sandeep Rathor

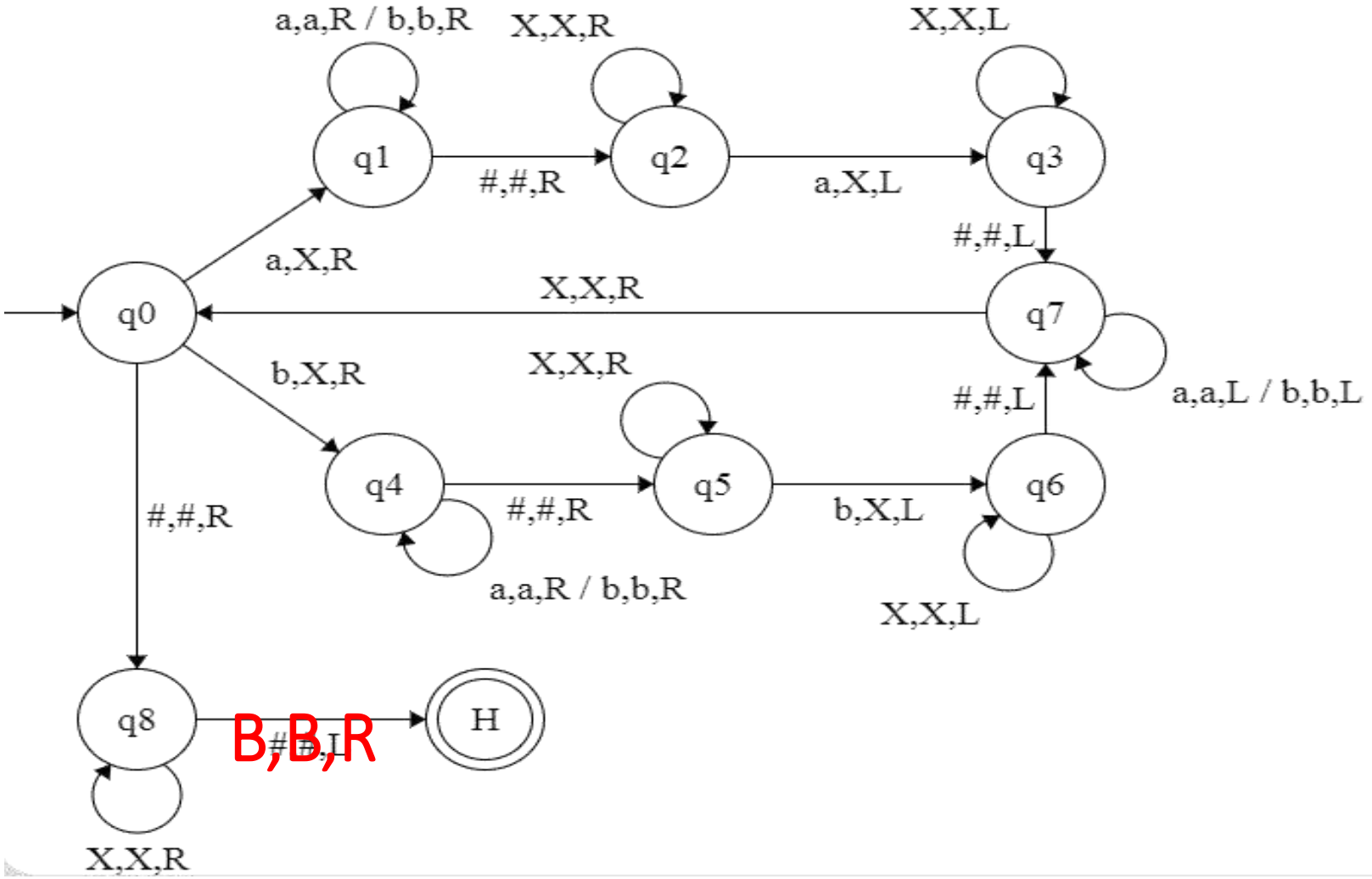
Describe a TM that decides the language

$L = \{wwR \mid w \in \{a, b\}^*\}$



Describe a TM that decides the language

L = {w#w | w ∈ {a, b}*}



Computing Functions with TM

By:
Dr. Sandeep Rathor

Design a TM to compute f(n) = n + 1

Hint

$S111 \vdash 1S11 \vdash 11S1 \vdash 111SB \vdash 1111HB$

Design a TM to compute One's Complement when i/p is given in binary

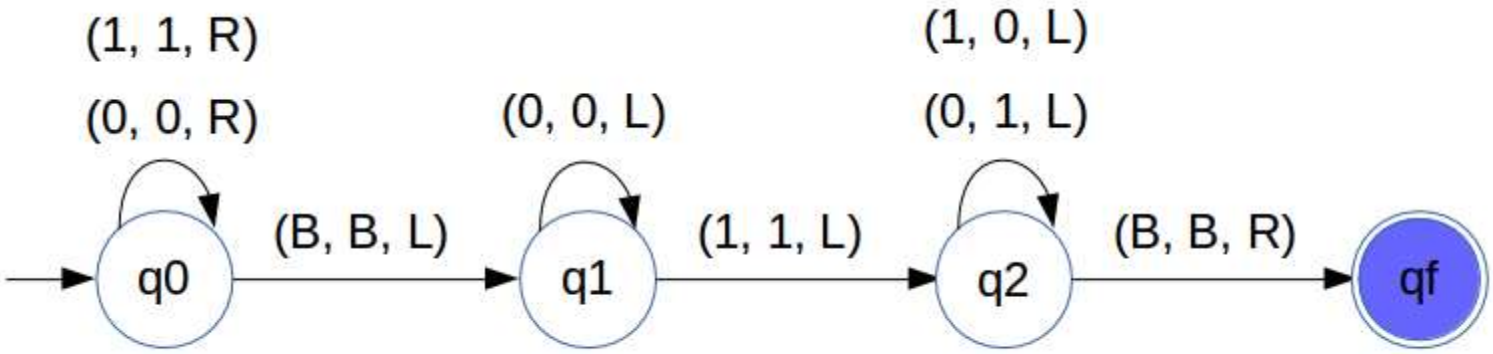
M = (Q, Σ, T, δ, S, B, H)

- Q = {S, H}
- Σ = {0, 1}
- T = {0, 1, B}
- δ = {(S, 0, 1, S, R), (S, 1, 0, S, R), (S, #, #, H, R)}

Consider f(100). Write its Computation History

S100 ⊢ 0S00 ⊢ 01S0 ⊢ 011SB ⊢ 011HB

Computing 2,s Complement



Hint:

MSB LSB

1 1 1 0 1 1 0 0

0 0 0 1 0 0 1 1 1's Complement

1 2's

0 0 0 1 0 1 0 0 2's

Computing Functions with TM

By:
Dr. Sandeep Rathor

A function may have many parameters:

Example: Addition function

$$f(x, y) = x + y$$

Integer Domain

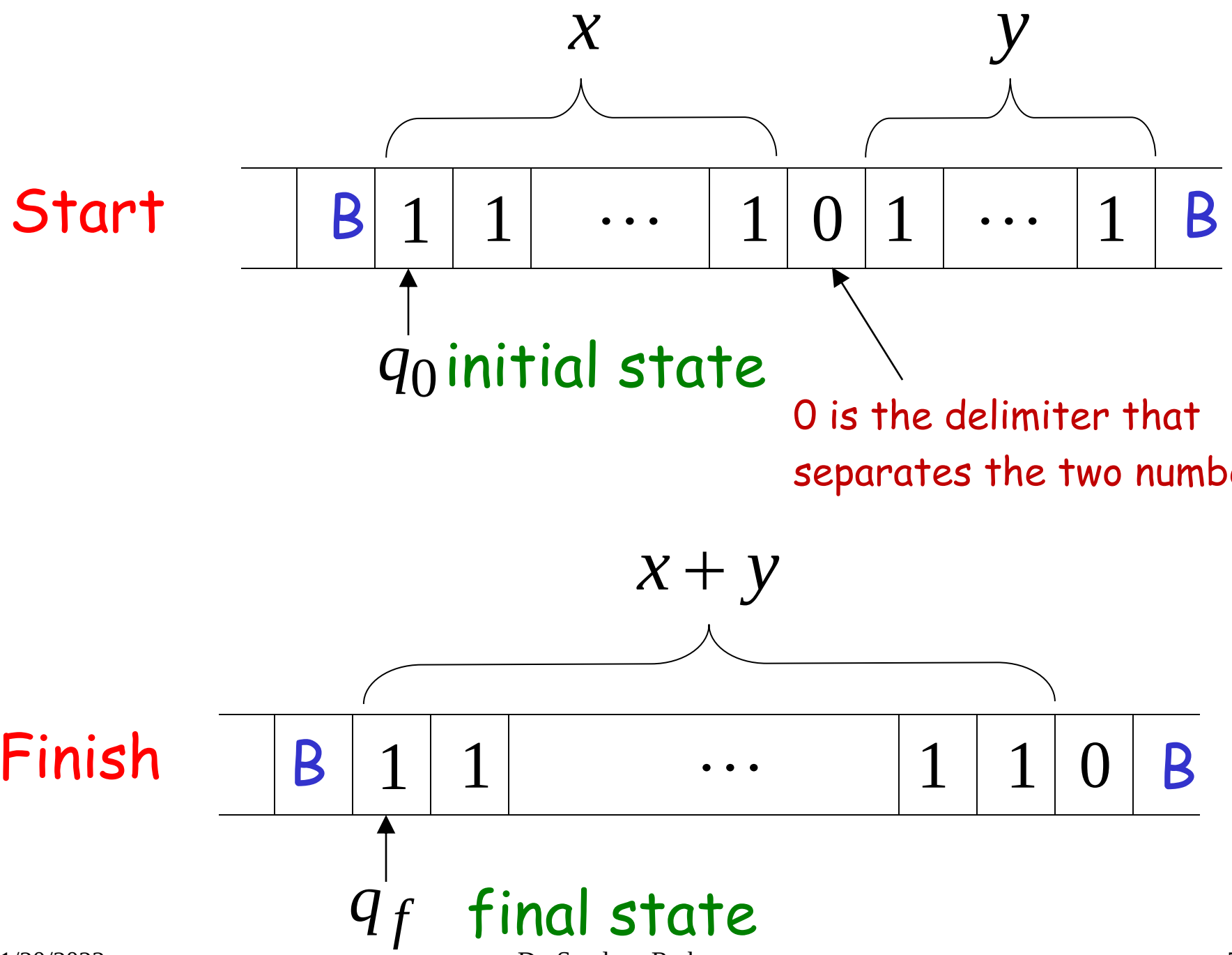
Decimal: 5

Binary: 101

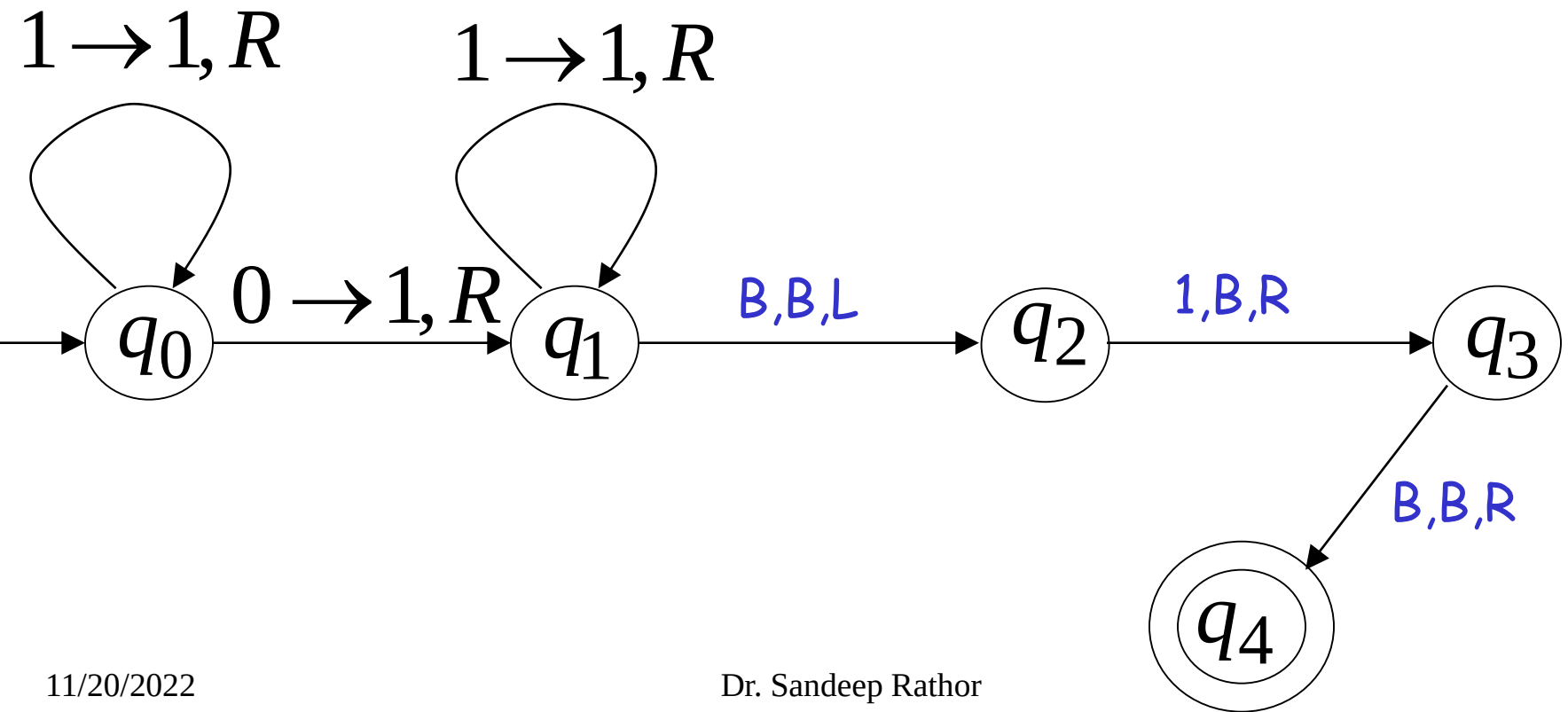
Unary: 11111

We prefer **unary** representation:

easier to manipulate with Turing machines


Start
x
y
B
1
1
⋯
1
0
1
⋯
1
B
q_0 initial state
0 is the delimiter that
separates the two numbers
Finish
x + y
B
1
1
⋯
1
1
0
B
q_f final state

Turing machine for function $f(x, y) = x + y$

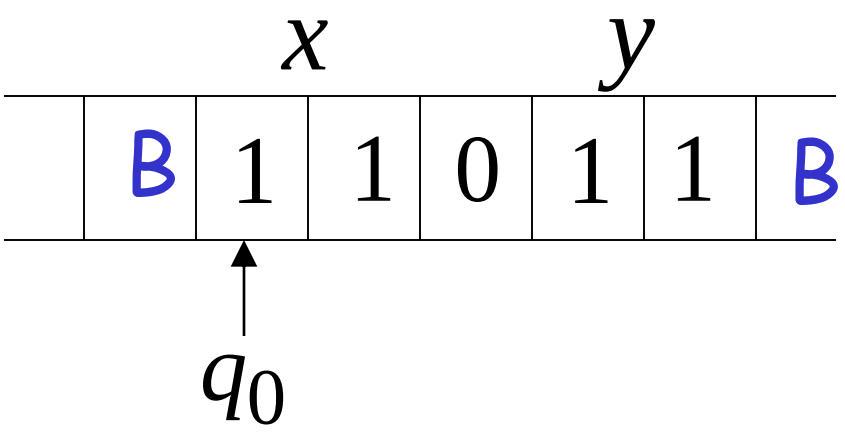


Execution Example:

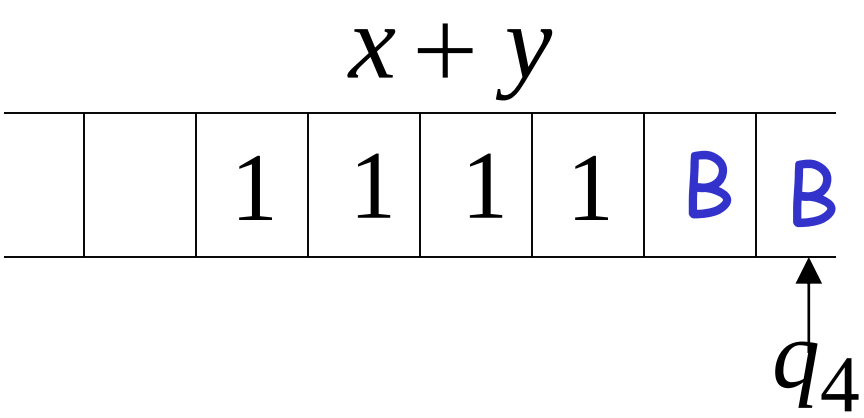
$x = 11$ (2)

$y = 11$ (2)

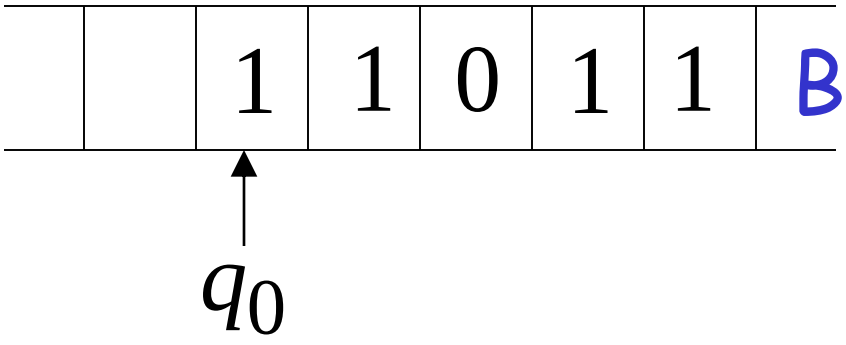
Time 0

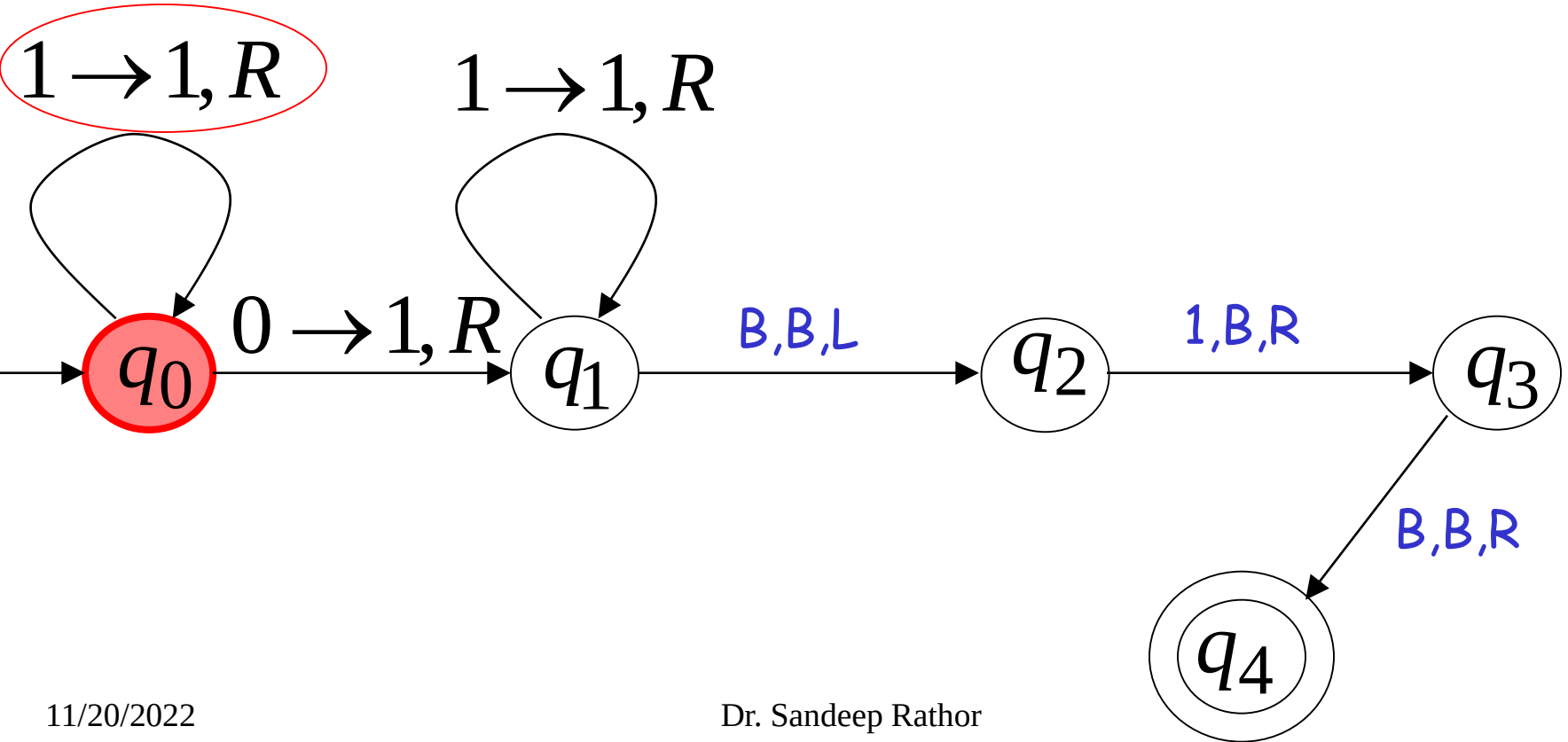


Final Result

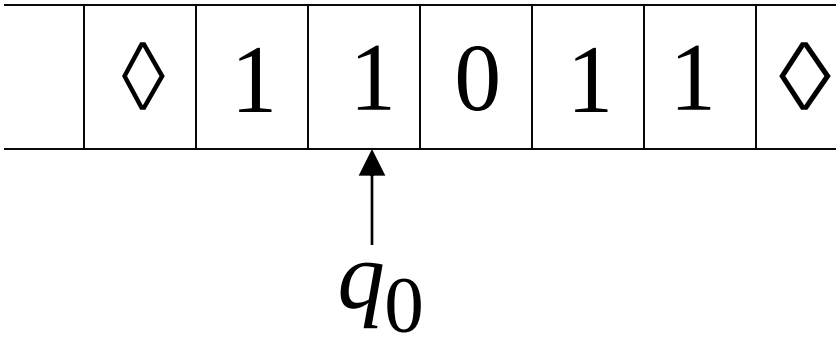


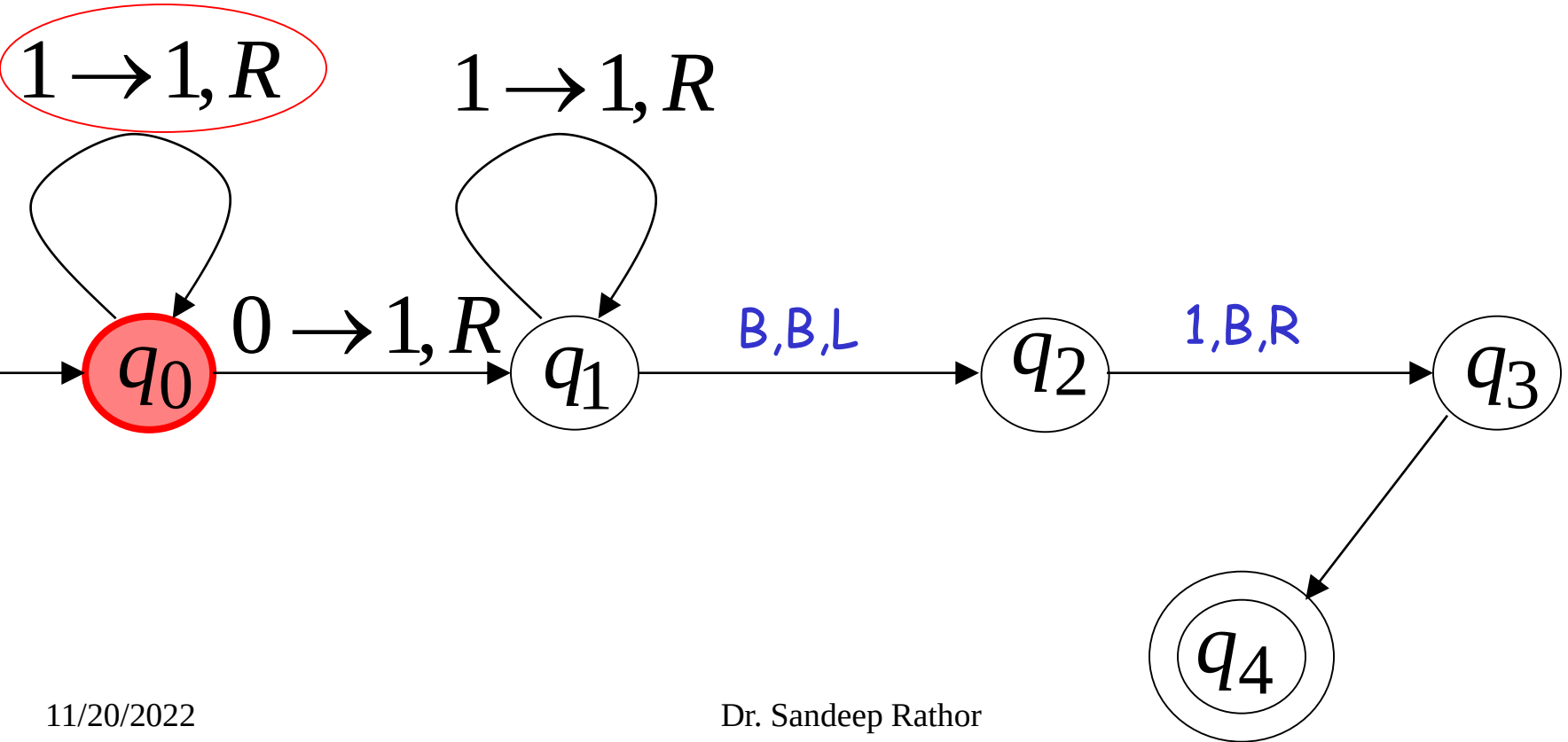
Time 0



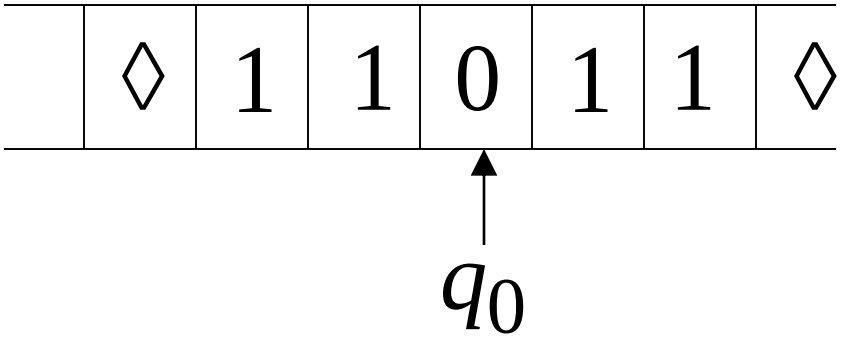


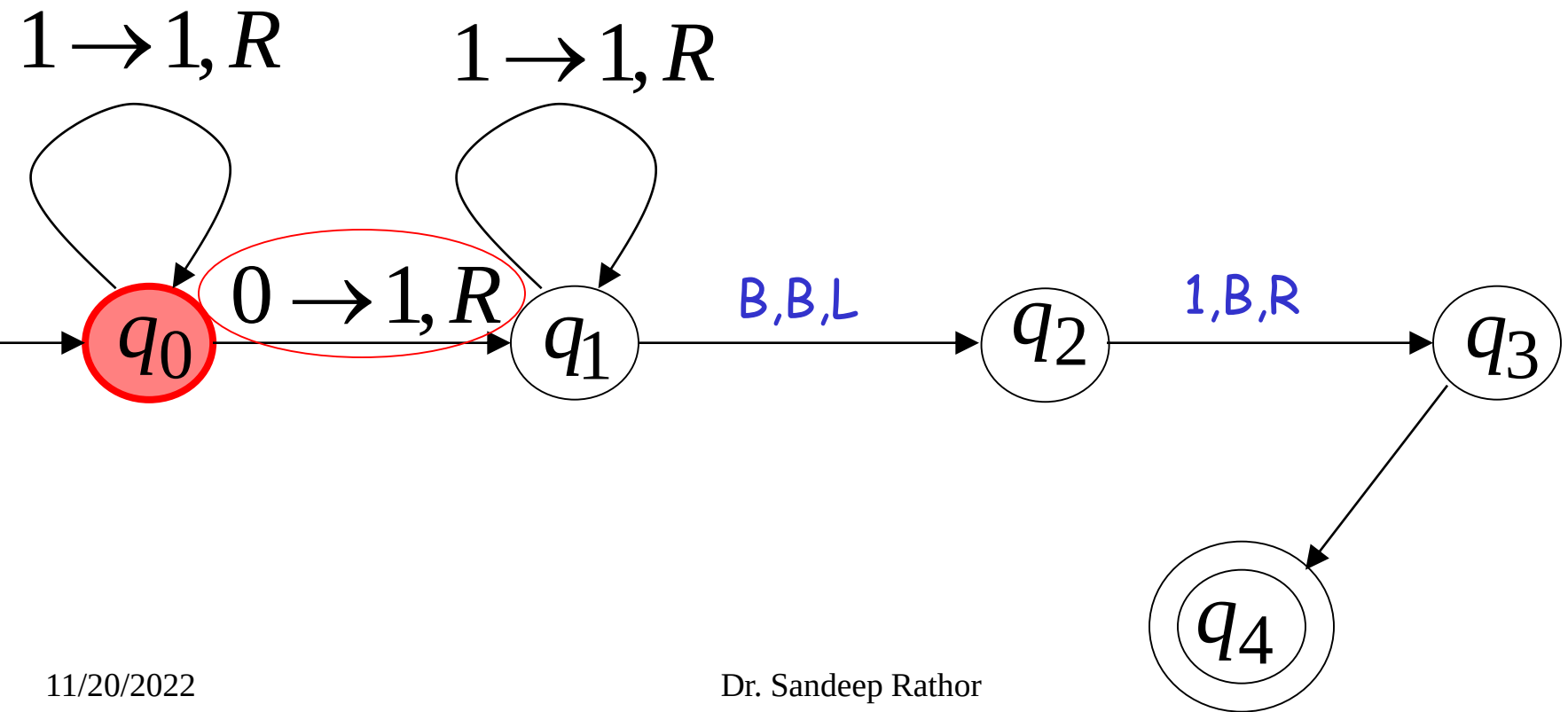
Time 1



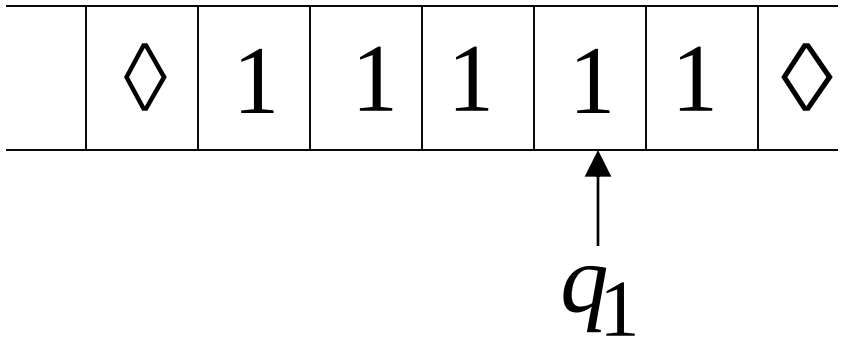


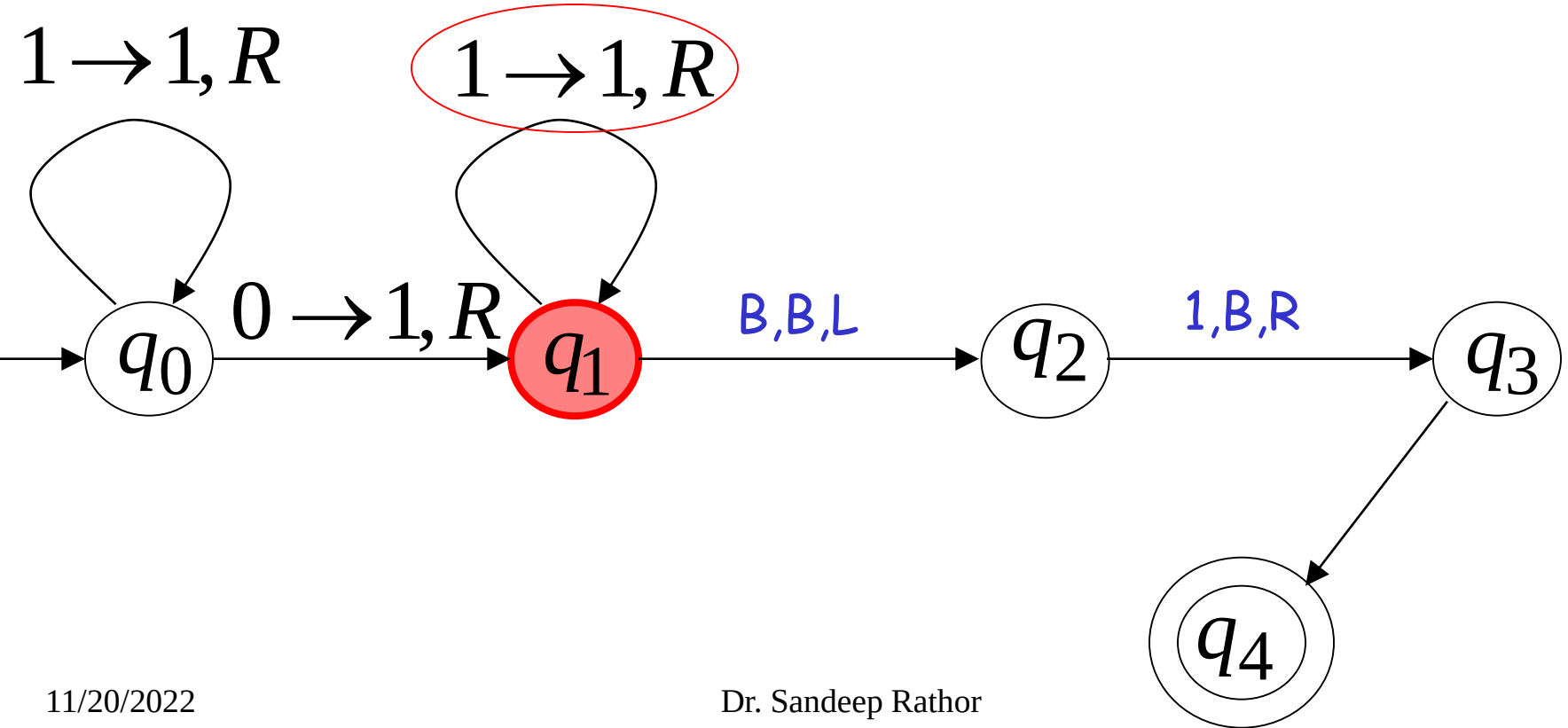
Time 2



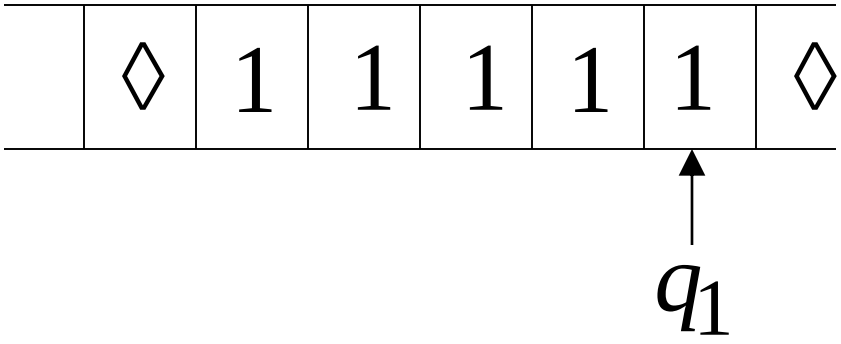


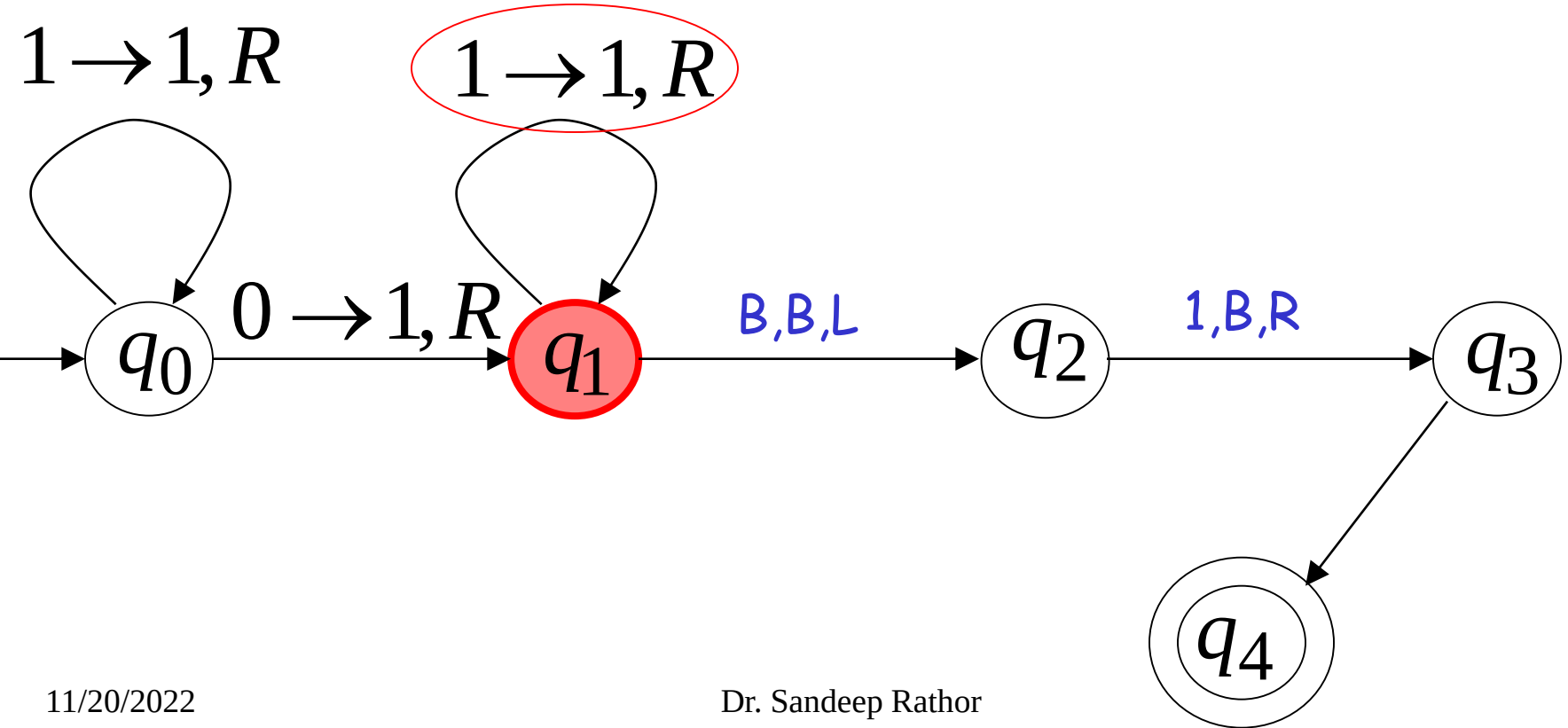
Time 3



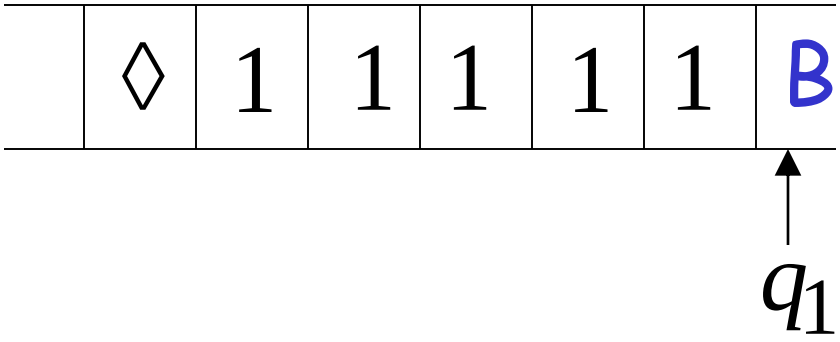


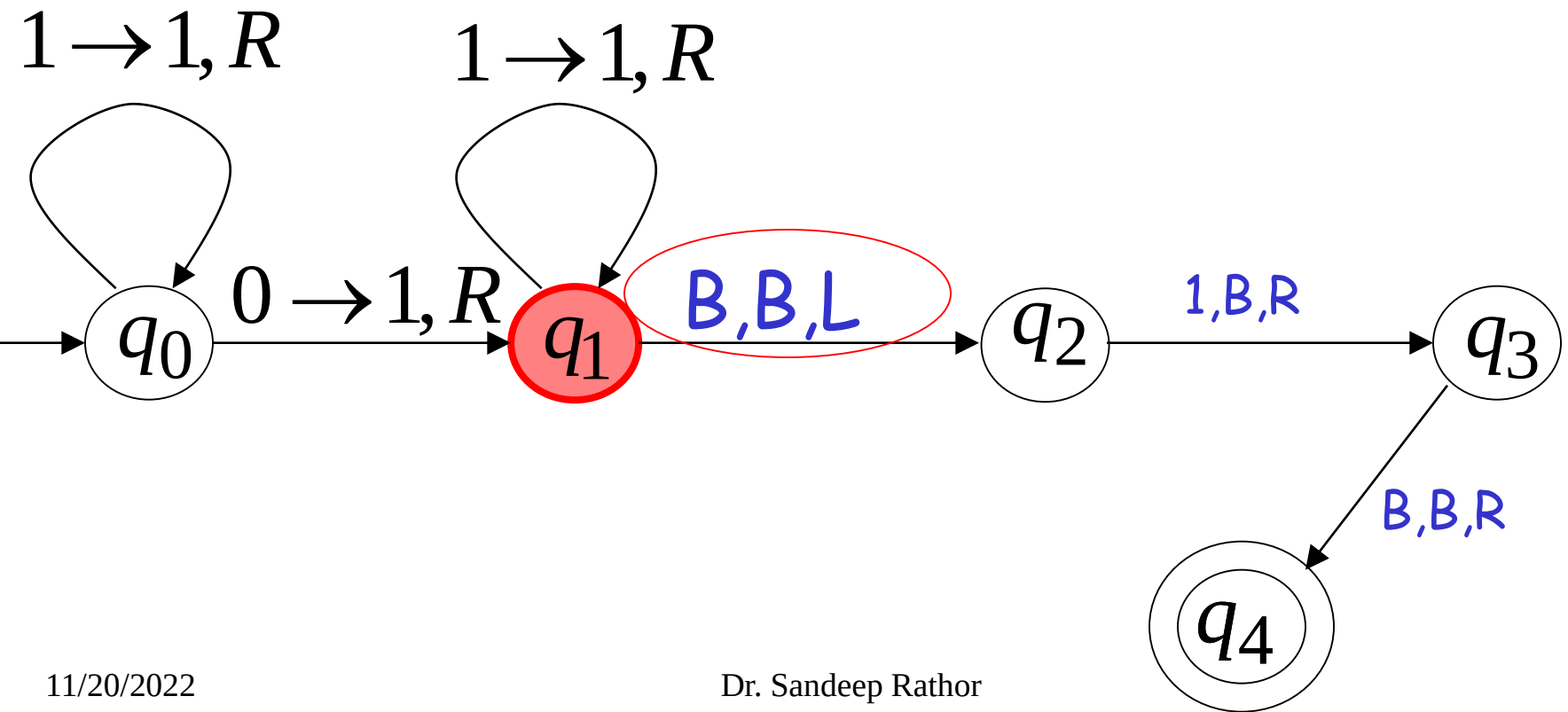
Time 4



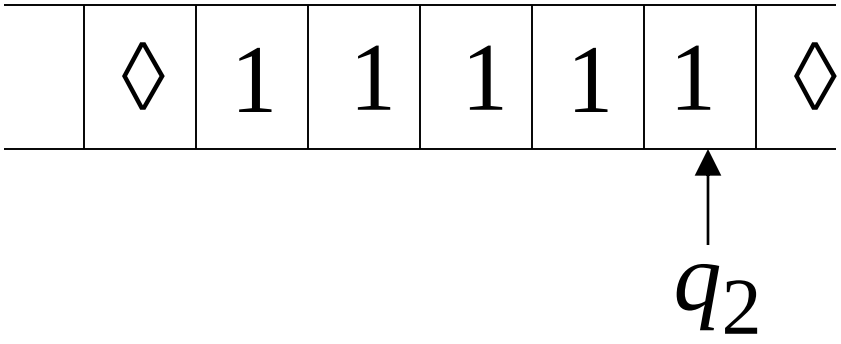


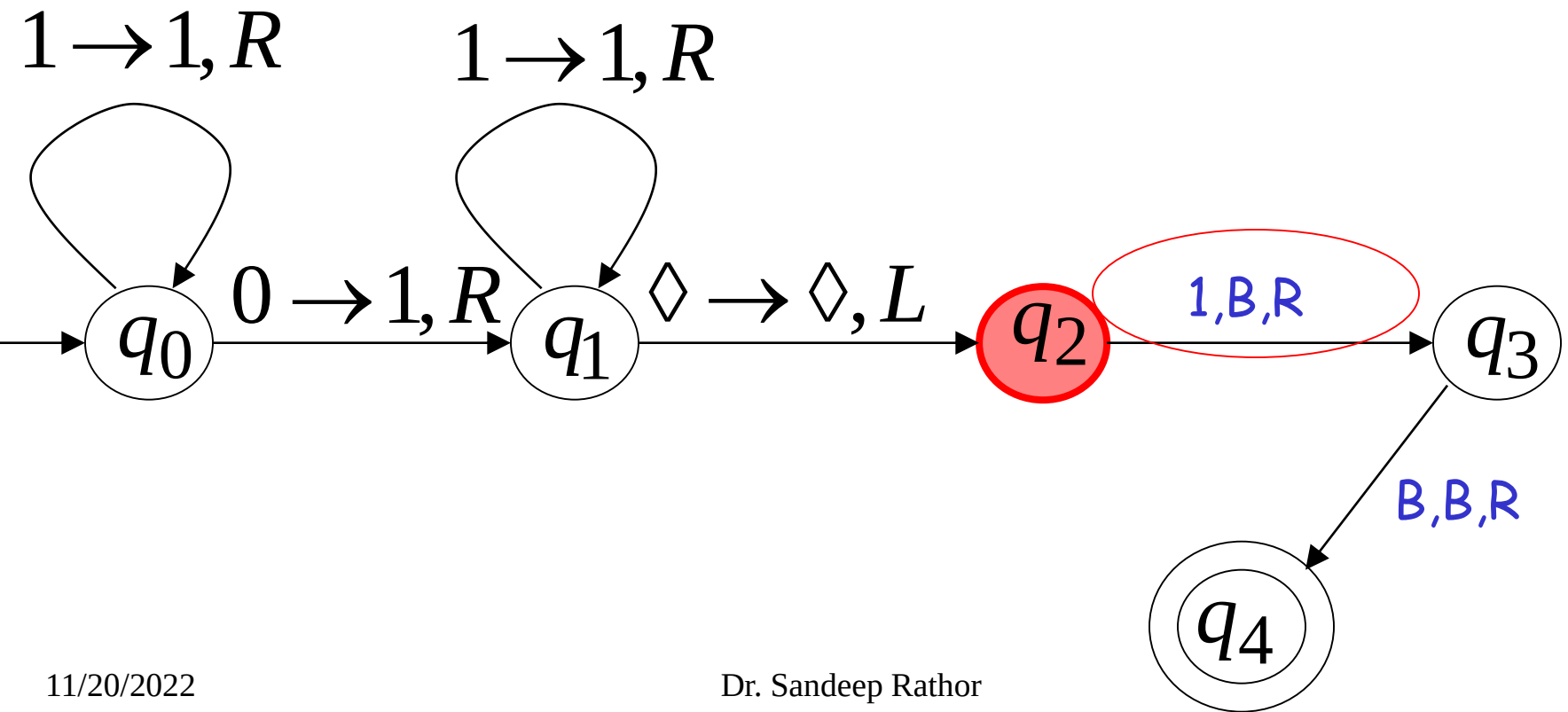
Time 5



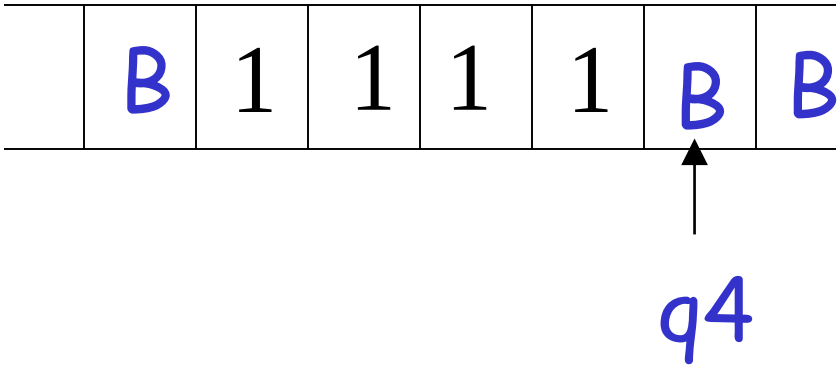


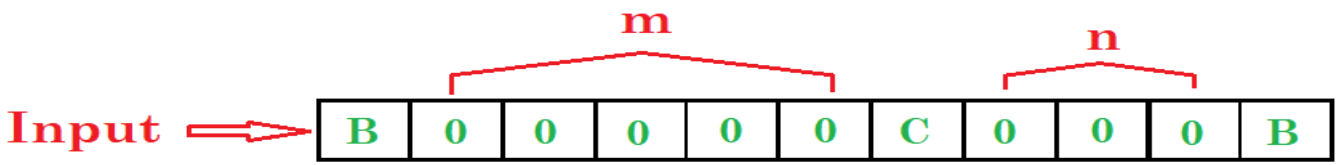
Time 6

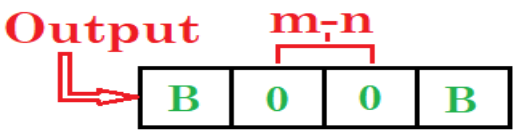

◊ 1 1 1 1 1 ◊
q_2

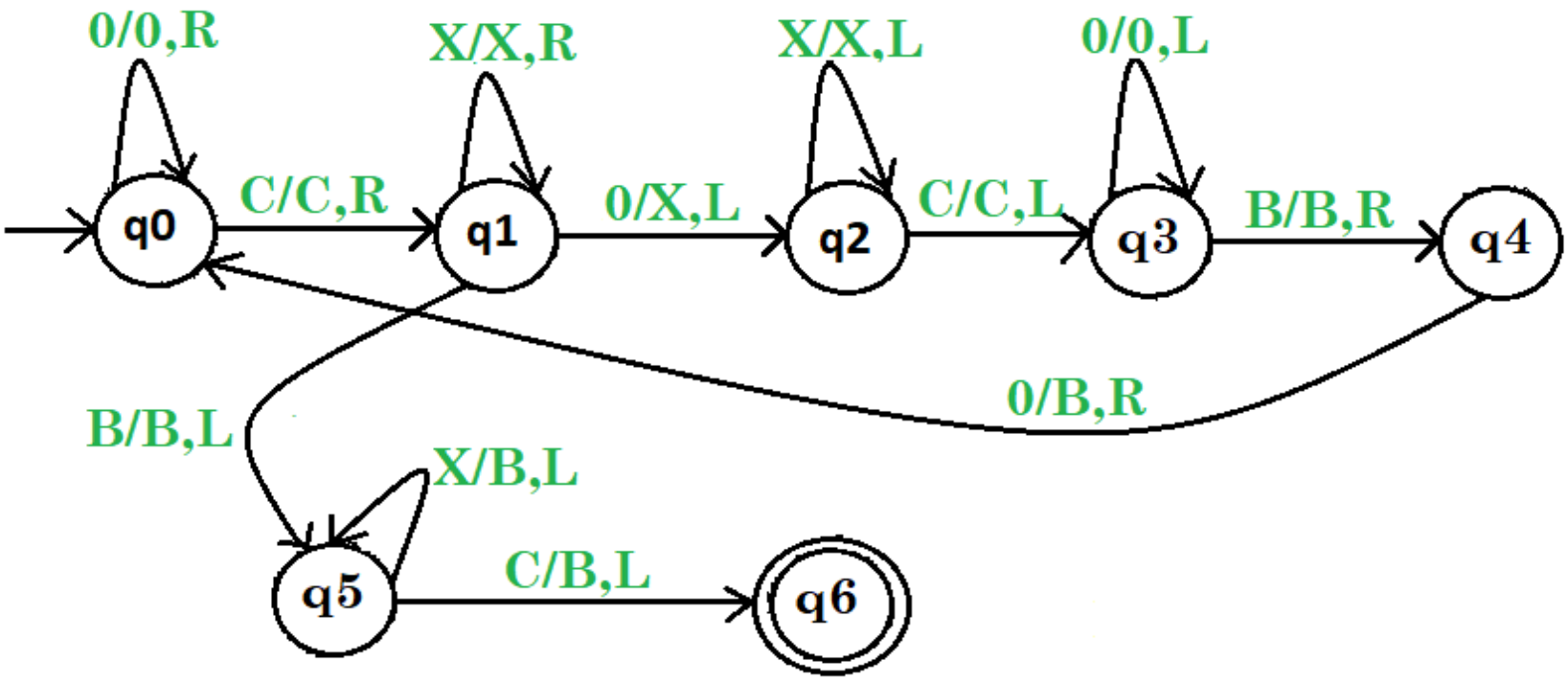

$1 \to 1, R$
$1 \to 1, R$
$0 \to 1, R$
$\Diamond \to \Diamond, L$
1,B,R
B,B,R
q_0
q_1
q_2
q_3
q_4

Time 7




m
n
Input
B
0
0
0
0
0
C
0
0
0
B


Output
m-n
B
0
0
B


0/0,R
X/X,R
X/X,L
0/0,L
C/C,R
0/X,L
C/C,L
B/B,R
q0
q1
q2
q3
q4
0/B,R
B/B,L
X/B,L
C/B,L
q5
q6

Another Example

The function $f(x) = 2x$ is computable

x is integer

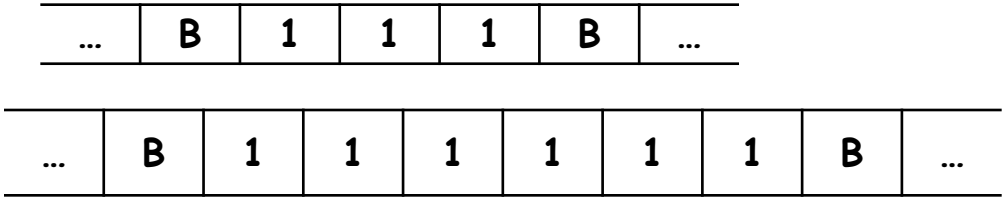
Turing Machine:

Input string: x unary

Output string: xx unary

Design a TM that copies strings of 1
f(n) = 2n

$$q_0 w \overset{*}{\rightarrow} Hww$$



First write an algorithm.

1. Replace every 1 by x
2. Find the rightmost x & replace it by 1
3. Travel to the right end of the current non-blank region & write a 1 there
4. Repeat steps 2 & 3 until there are no more x's

δ (q0, 1) = (q0, x, R)

δ (q0, B) = (q1, B,L)

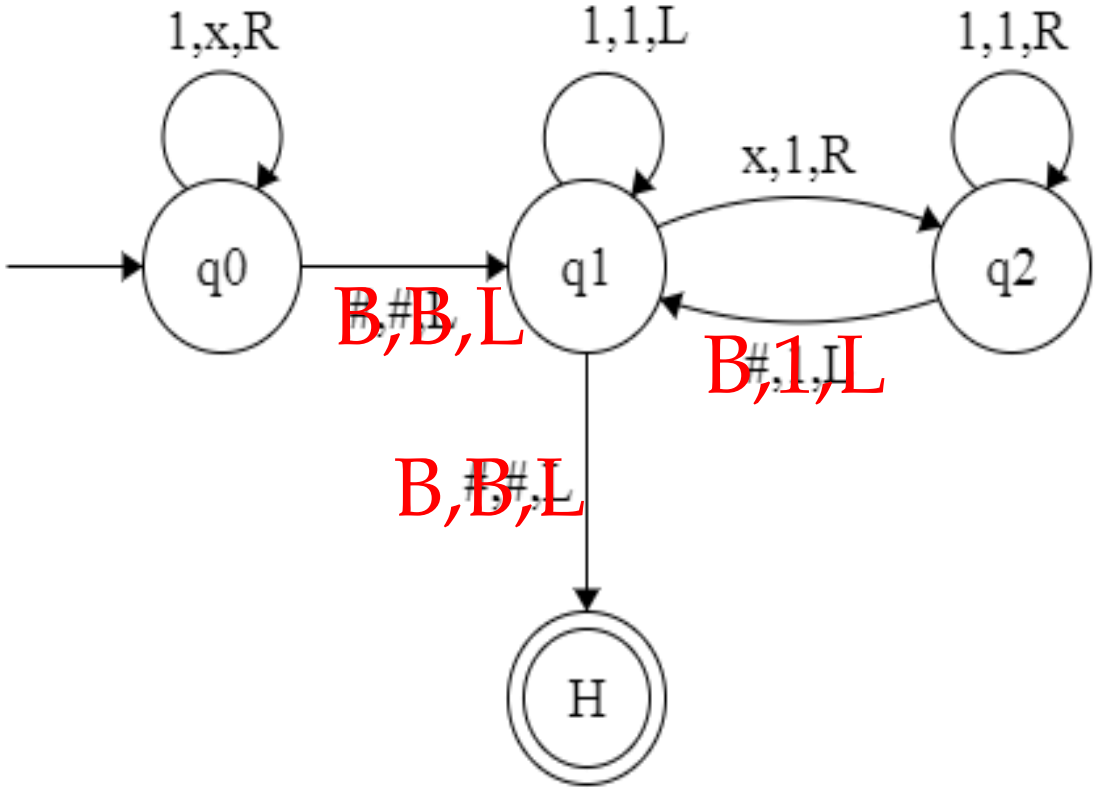
δ (q1, 1) = (q1, 1, L)

δ (q1, x) = (q2, 1, R)

δ (q2, 1) = (q2, 1, R)

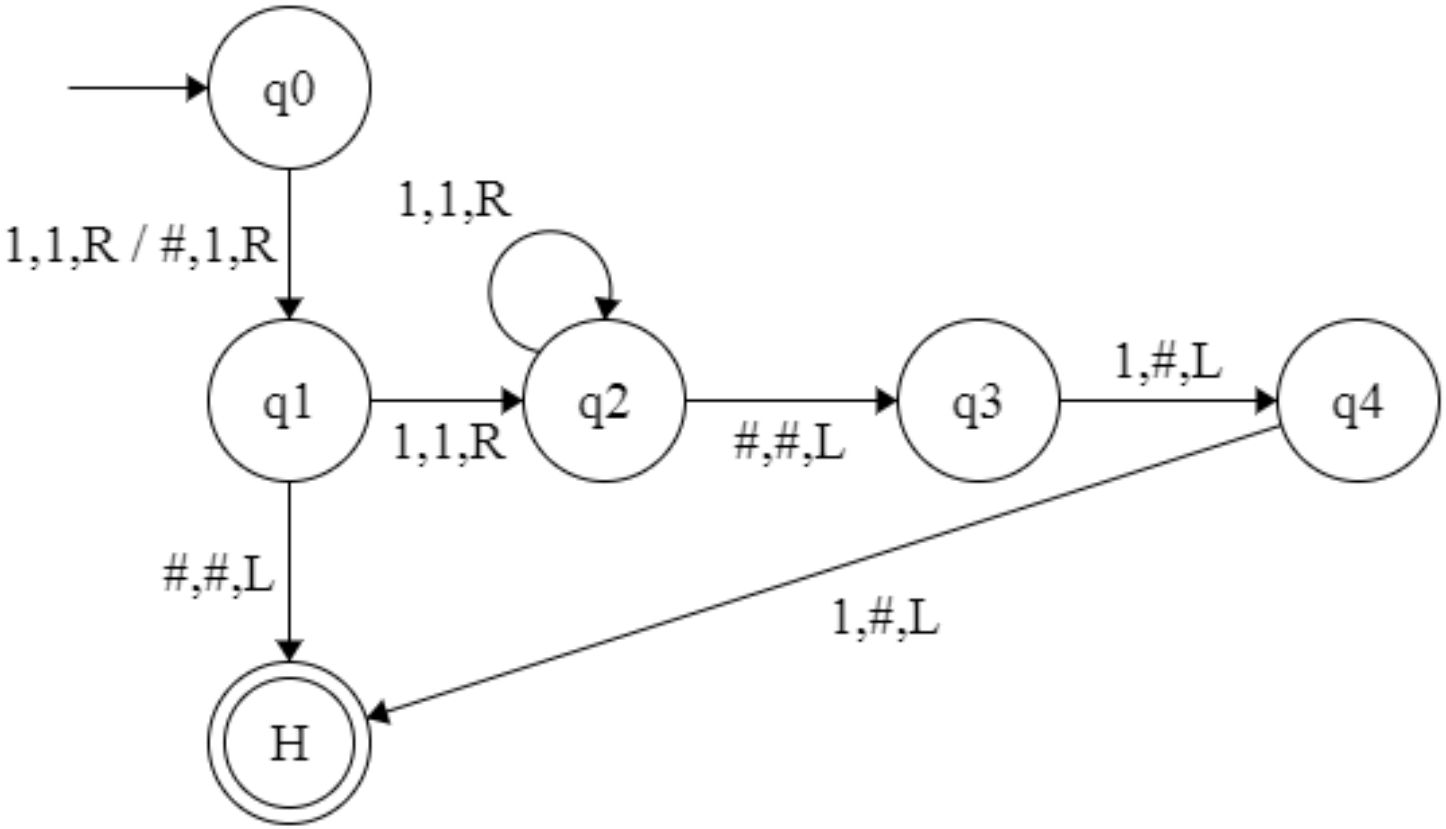
δ(q2, B) = (q1, 1, L)

δ (q1, B) = (H, B, L)



Design a TM to compute

$$\mathbf{f(n)} = \begin{cases} \mathbf{n-2\ if\ n \geq 2} \\ \mathbf{1\ \ if\ \ n < 2} \end{cases}$$



Variants of Turing Machine

By:
Dr. Sandeep Rathor

Variants of TM

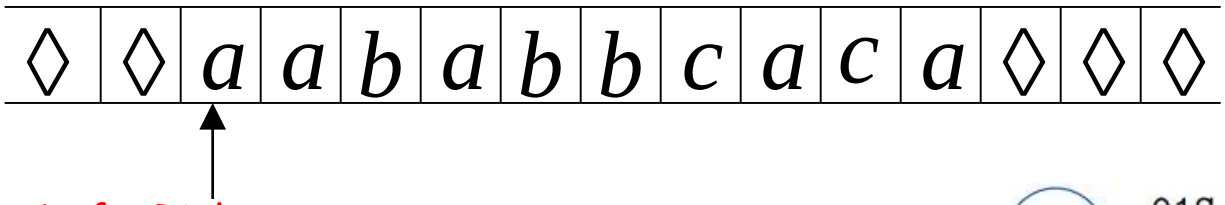
1. Turing Machine with Stay Option

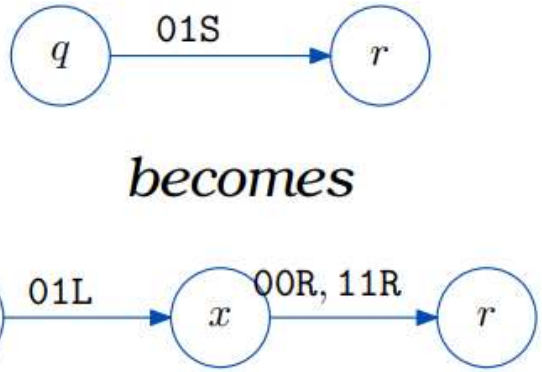
- In standard TM, the R/W head must move either Left or right
- We can define TM with Stay-option

$$\delta: Q \times T \rightarrow Q \times T \times \{L, R, S\}$$

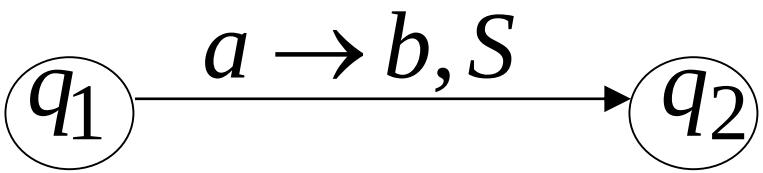
- One can achieve the net effect of stay-in place by moving the head off the cell and immediately moving it back!

The head can stay in the same position

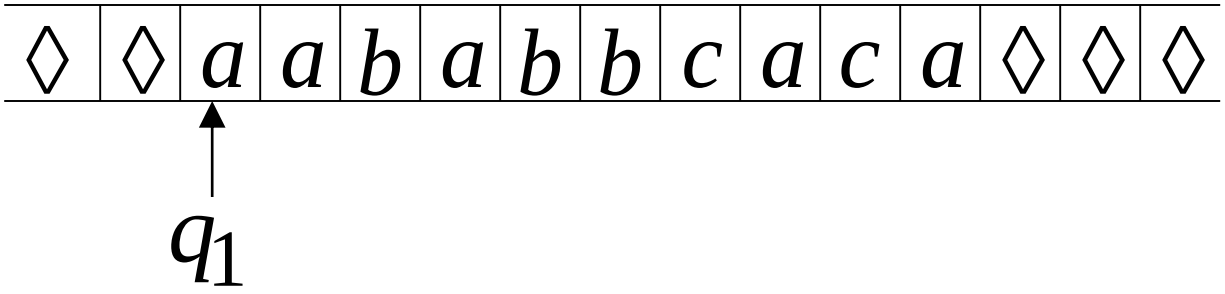




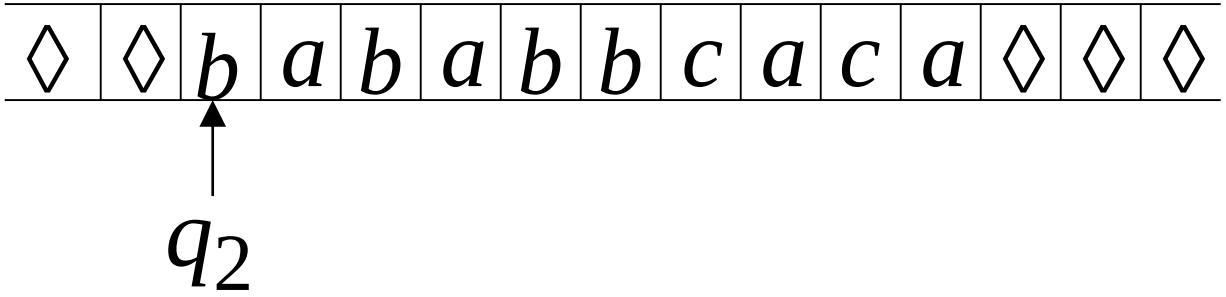
Example:



Time 1



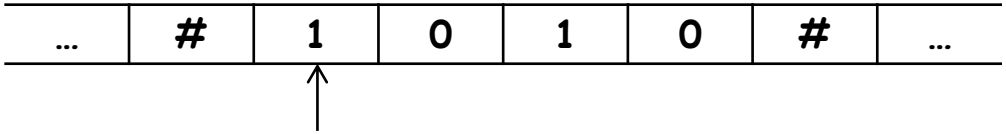
Time 2

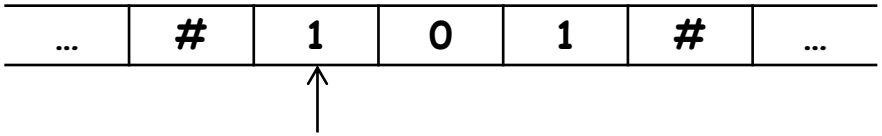


2. Two-way infinite Tape Turing Machine

- Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.
- Let M_0 with one-way infinite tape TM imitating one-way infinite tape TM, M
- The first thing that M_0 needs to do is to mark the start of its tape.
- It can move the whole string to the right every time M moves left to the marker.

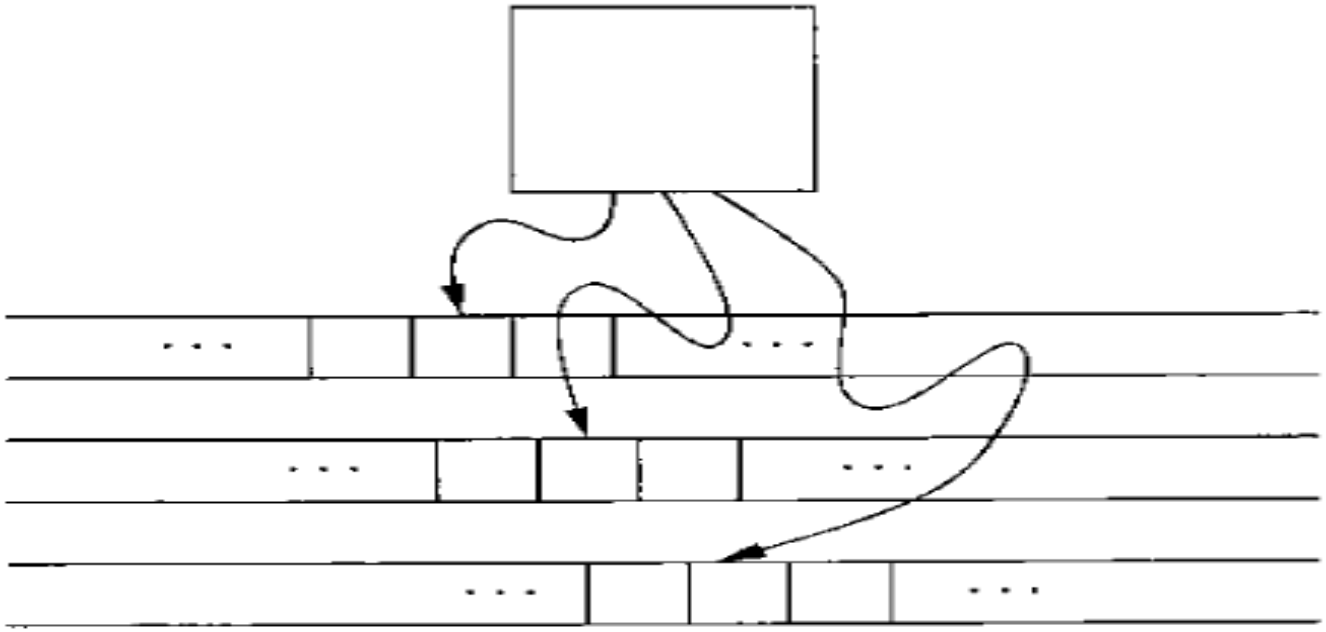
What is the behaviour when started on the tape



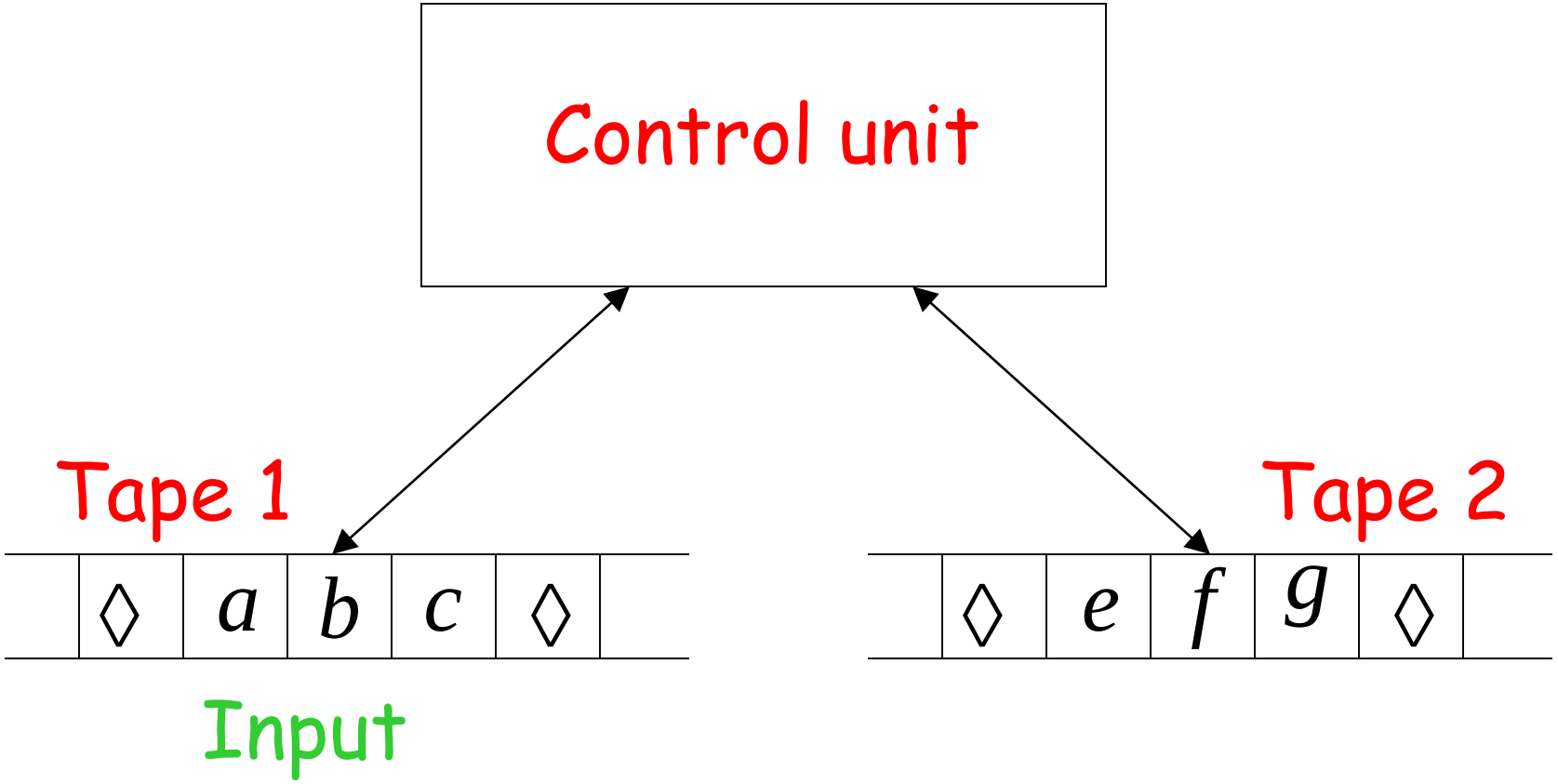


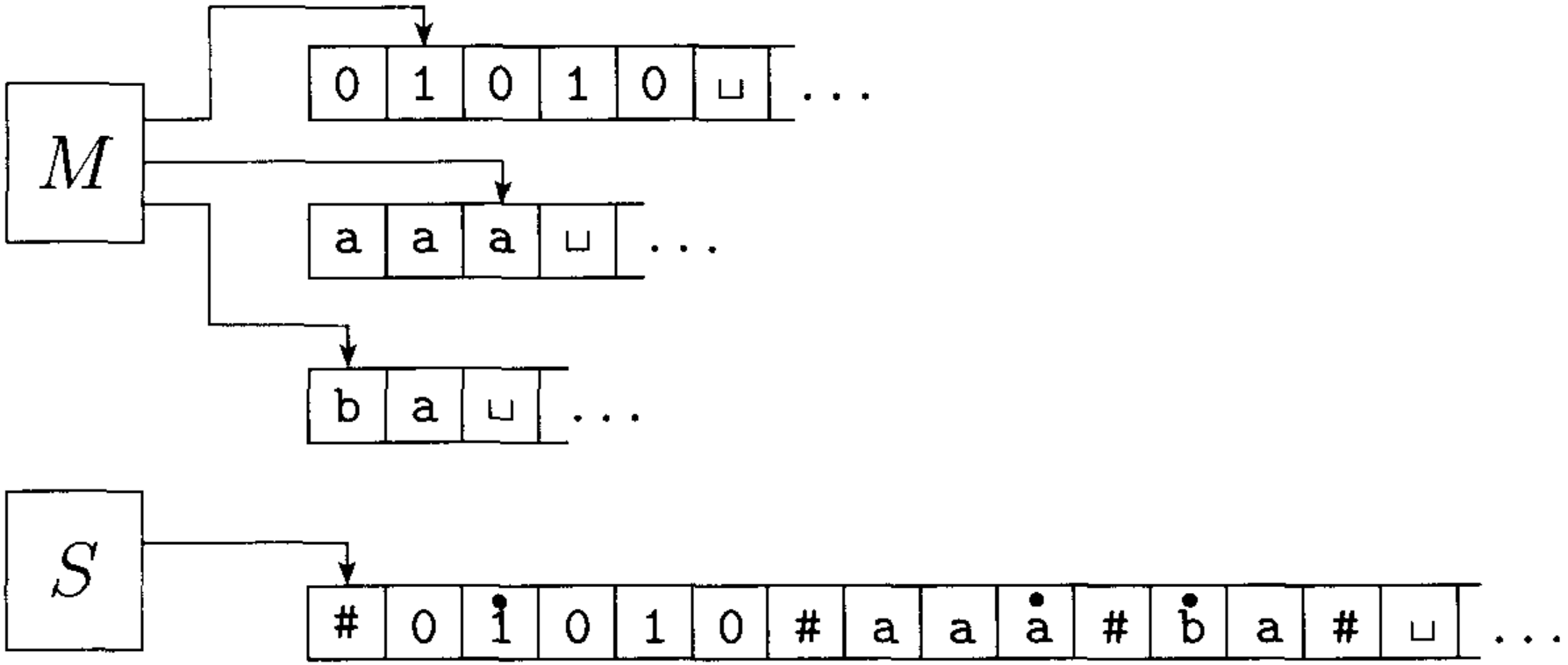
3. Multi Tape Turing Machine

- TM with k tapes and k heads.
- Each tape has its own head for reading and writing.
- Initially the input appears on tape 1, and the others start out blank.
- δ: $Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L,R\}^k$
- $\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$



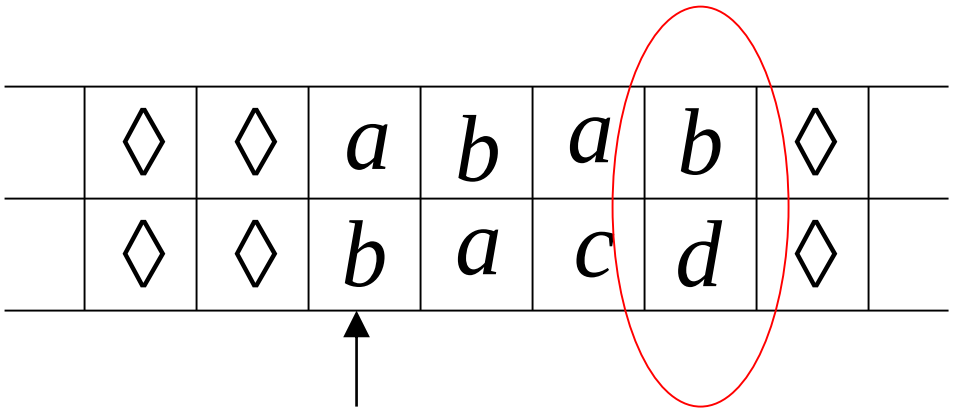
Model of Multitape Turing Machines





Representing three tapes with one

4. Multi Track Turing Machine



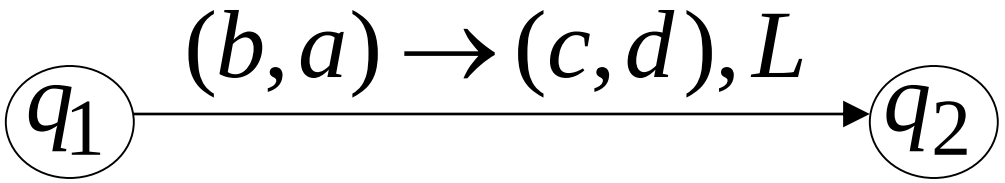
track 1

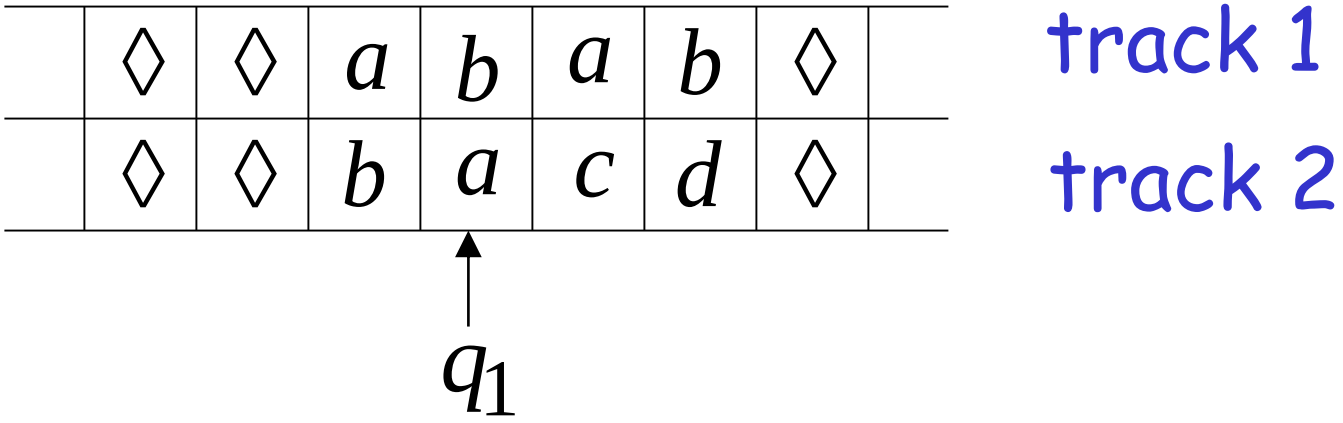
track 2

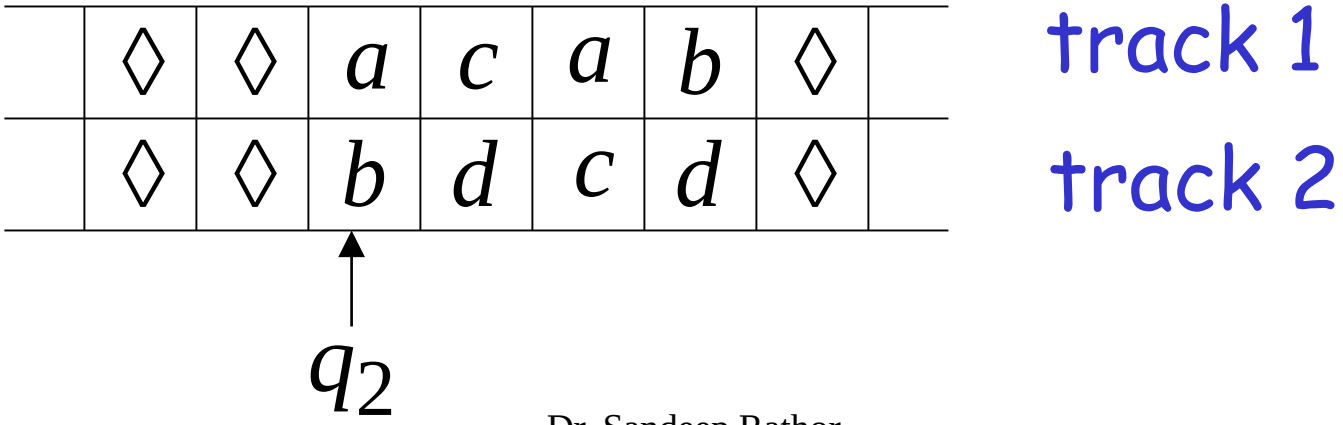
one symbol

Transition on multi track TM…..contd…

Transition on multi track TM…

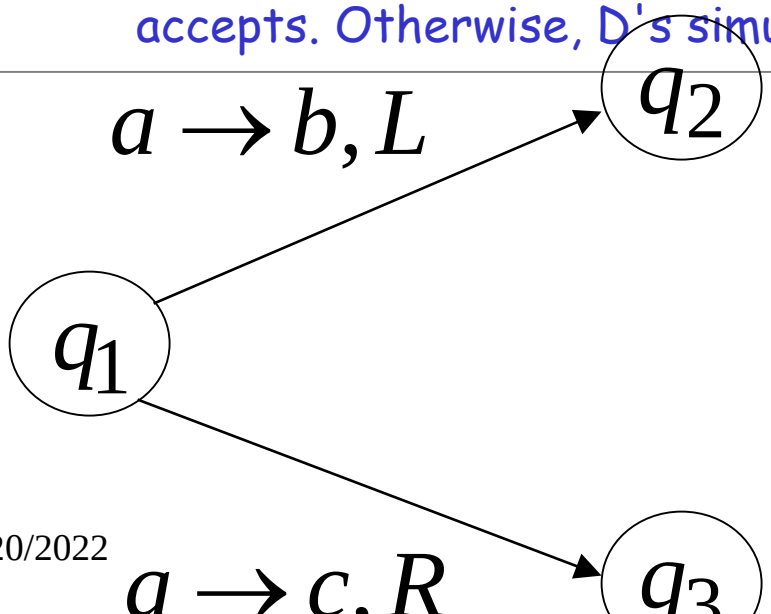


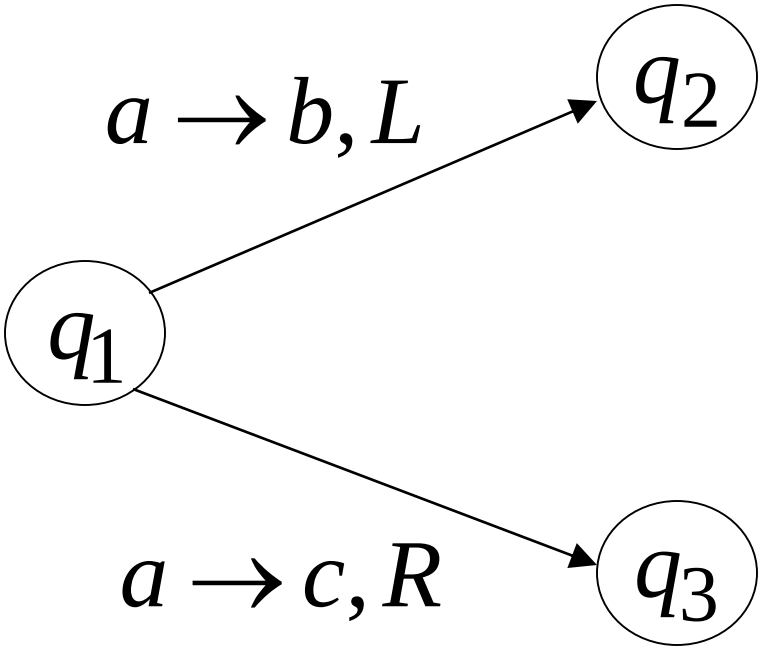




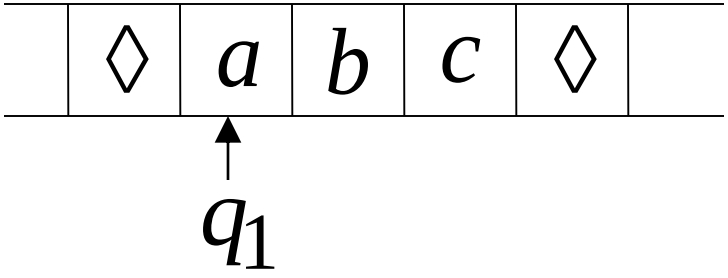
5. Nondeterministic Turing Machine

- At any point in a computation the machine may proceed according to several possibilities.
- $\delta: Q \times \Gamma \rightarrow 2^{(Q \times \Gamma \times \{L,R\})}$
- $\delta(q_i, a) = \{(q_1, b, R), (q_2, c, L), \ldots, (q_m, d, R)\}$
- We can simulate any nondeterministic TM N with a deterministic TM D.
- The idea behind the simulation is to have D try all possible branches of N's nondeterministic computation.
  - If D ever finds the accept state on one of these branches, D accepts. Otherwise, D's simulation will not terminate.



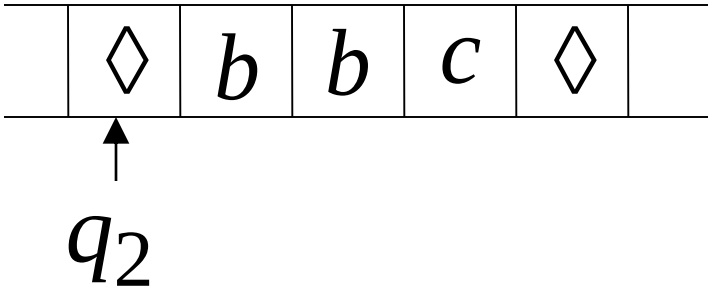


Time 0

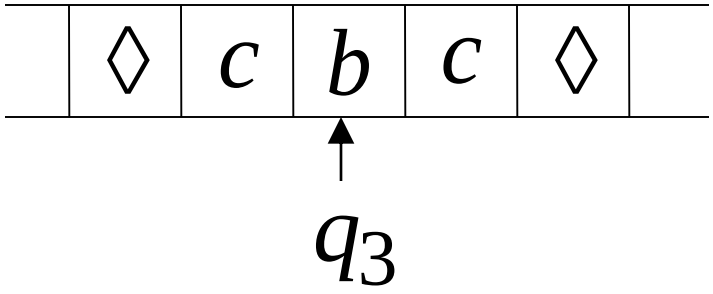


Time 1

Choice 1



Choice 2



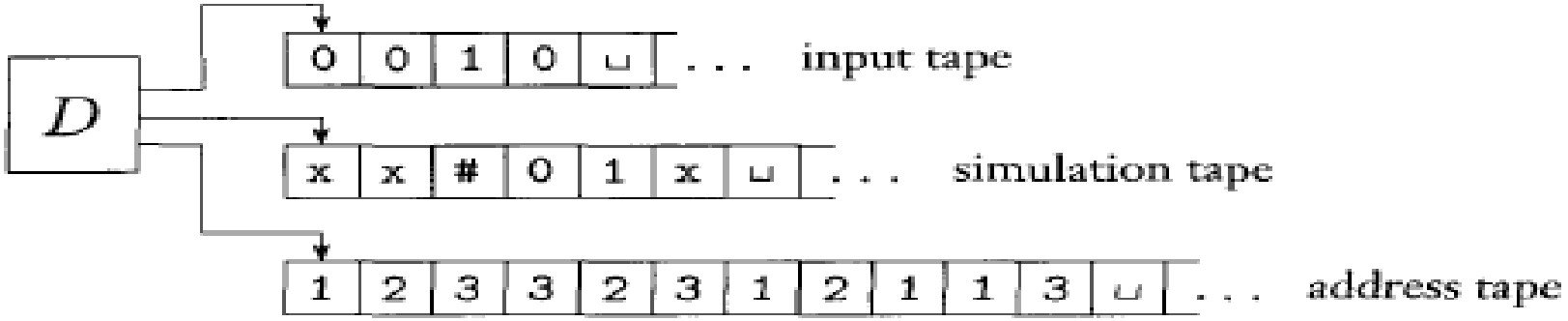
- We view N's computation on an input w as a tree. Each branch of the tree represents one of the branches of the non-determinism.
- Each node of the tree is a configuration of N.
- The root of the tree is the start configuration.
- The TM D searches this tree for an accepting configuration.

Can we use the depth-first search strategy that goes all the way down one branch before backing up to explore other branches?

- We use a strategy that explores all branches to the same depth before going on to explore any branch to the next depth.
- This method guarantees that *D will* visit every node in the tree until it encounters an accepting configuration.

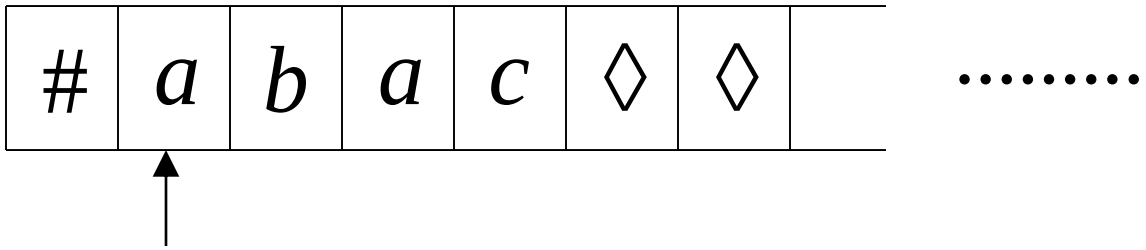
The simulating deterministic TM D has three tapes. (We know that this arrangement is equivalent to having a single tape.)

- Tape 1 always contains the input string and is never altered.
- Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation.
- Tape 3 keeps track of D's location in N's nondeterministic computation tree.



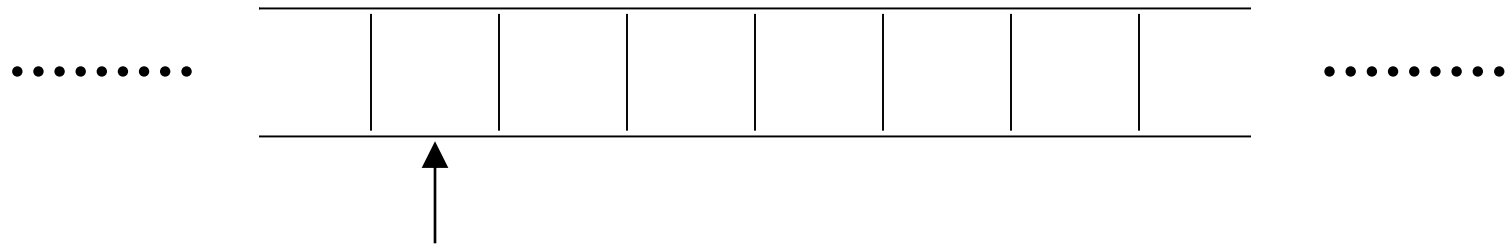
Remark* Non-Deterministic Machines have the same power with Deterministic machines

6. Semi Infinite Tape Turing Machine

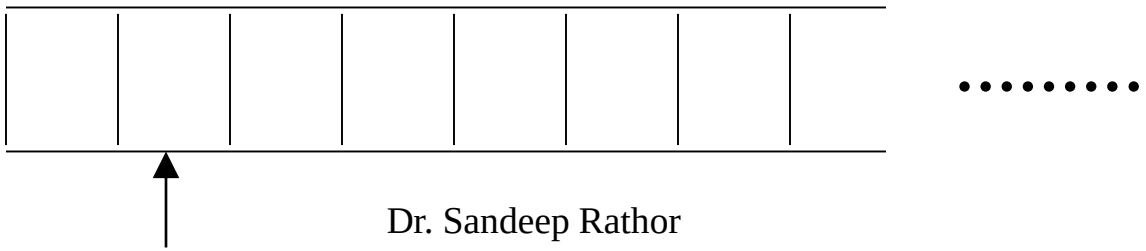


*Standard Turing machines simulate Semi-infinite tape machines.

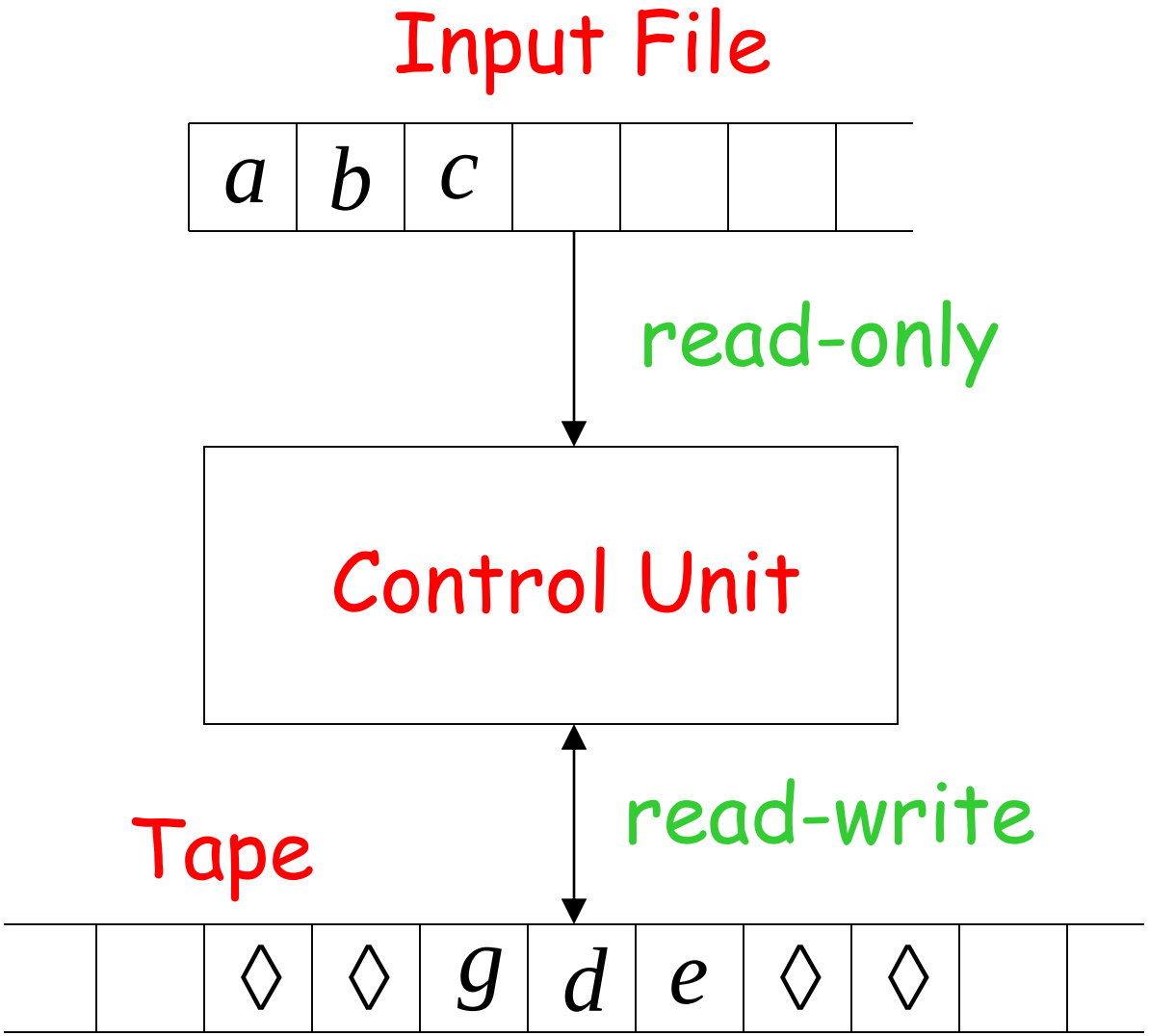
Standard Turing machine



Semi-infinite tape Turing machine



7. OFF- Line Turing Machine



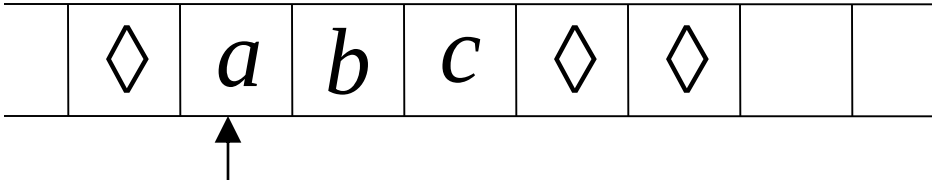
Off-line machines simulate Standard Turing Machines:

Off-line machine:

1. Copy input file to tape

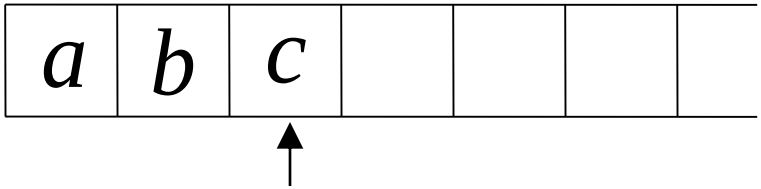
2. Continue computation as in Standard Turing machine

Standard machine

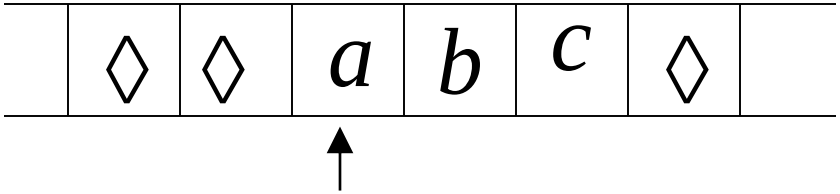


Off-line machine

Input File

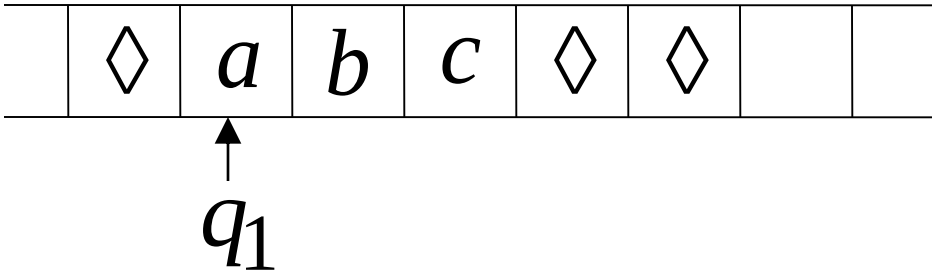


Tape



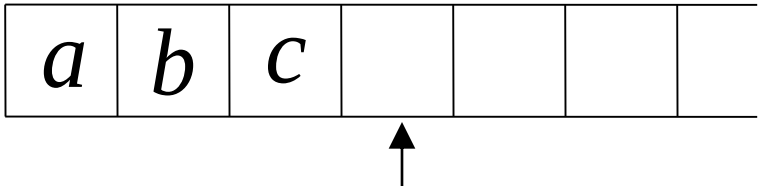
1. Copy input file to tape

Standard machine

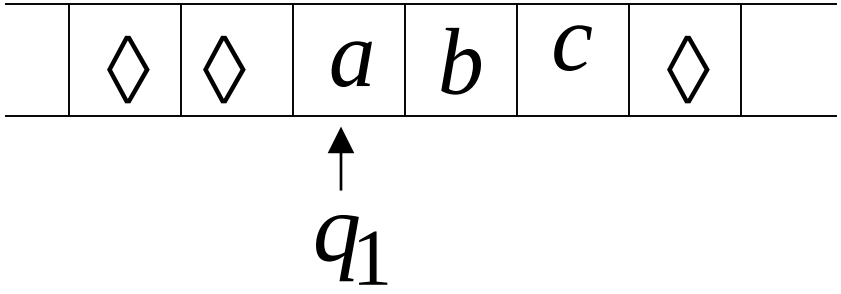


Off-line machine

Input File



Tape



2. Do computations as in Turing machine

Church Turing Thesis

By:
Dr. Sandeep Rathor

Church - Turing Thesis

Alonzo Church
(1903-1995)

Alan Turing
(1912-1954)

Some common interpretations of the thesis are

- **Every algorithmically computable function is TM-computable.**
- **A function is computable iff it can be solved by a Turing Machine**

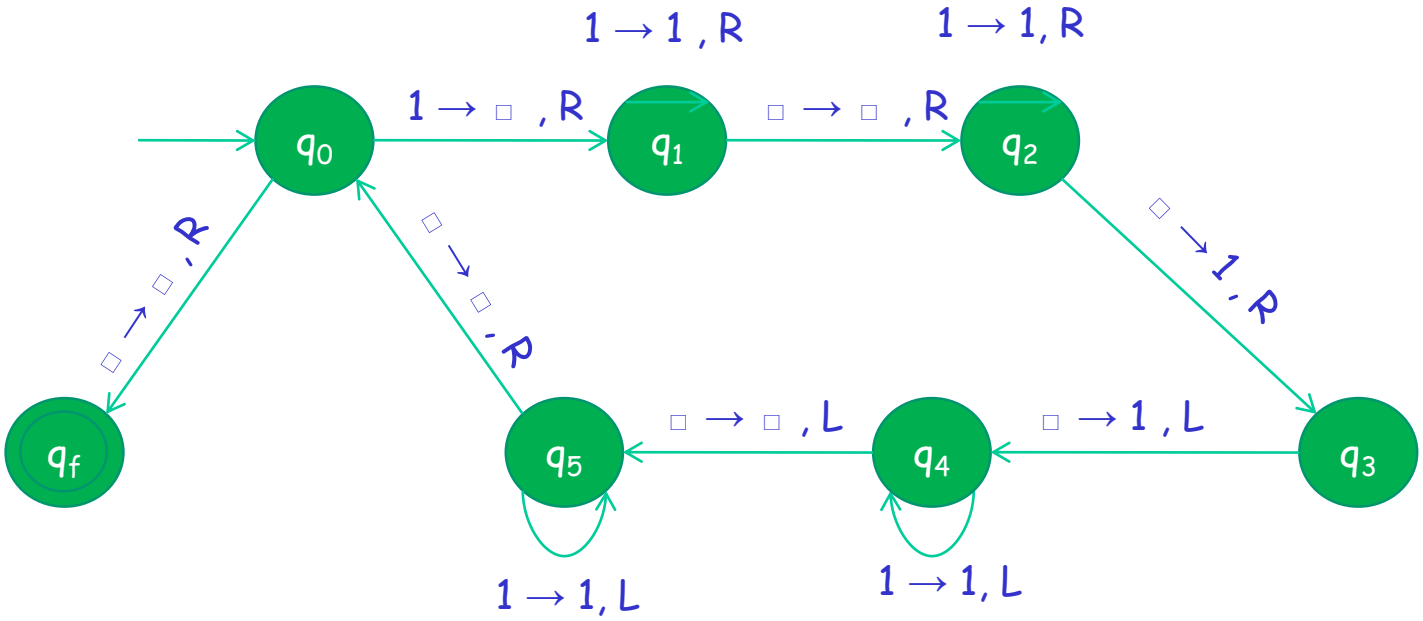
Thesis not Theorem: because we cannot prove this.. with a counter example we could disprove it (but this has not been done yet).

TM is a general model of computation, is simply to say that any algorithmic procedure that can be carried out at all (by the human or team of human or a computer) can be carried out by a TM. This statement was first formulated by Alenzo church in 1930. it was usually refer to as Church Thesis or Church Turing Thesis.

- It is mathematically precise statement b/c we do not have a precise definition of the term "Algorithmic procedure". Therefore, it is not something that we can prove. (that's why it is not a theorem)

- Since invention of TM however enough evidence has accumulated to cause the church Turing thesis to be generally accepted.

What does the following TM compute?



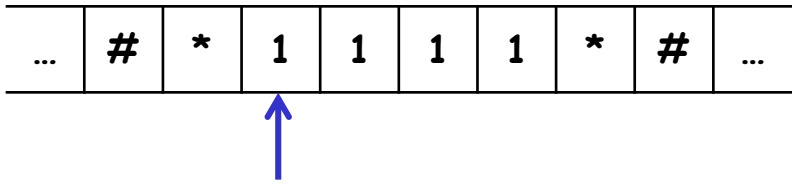
Test for

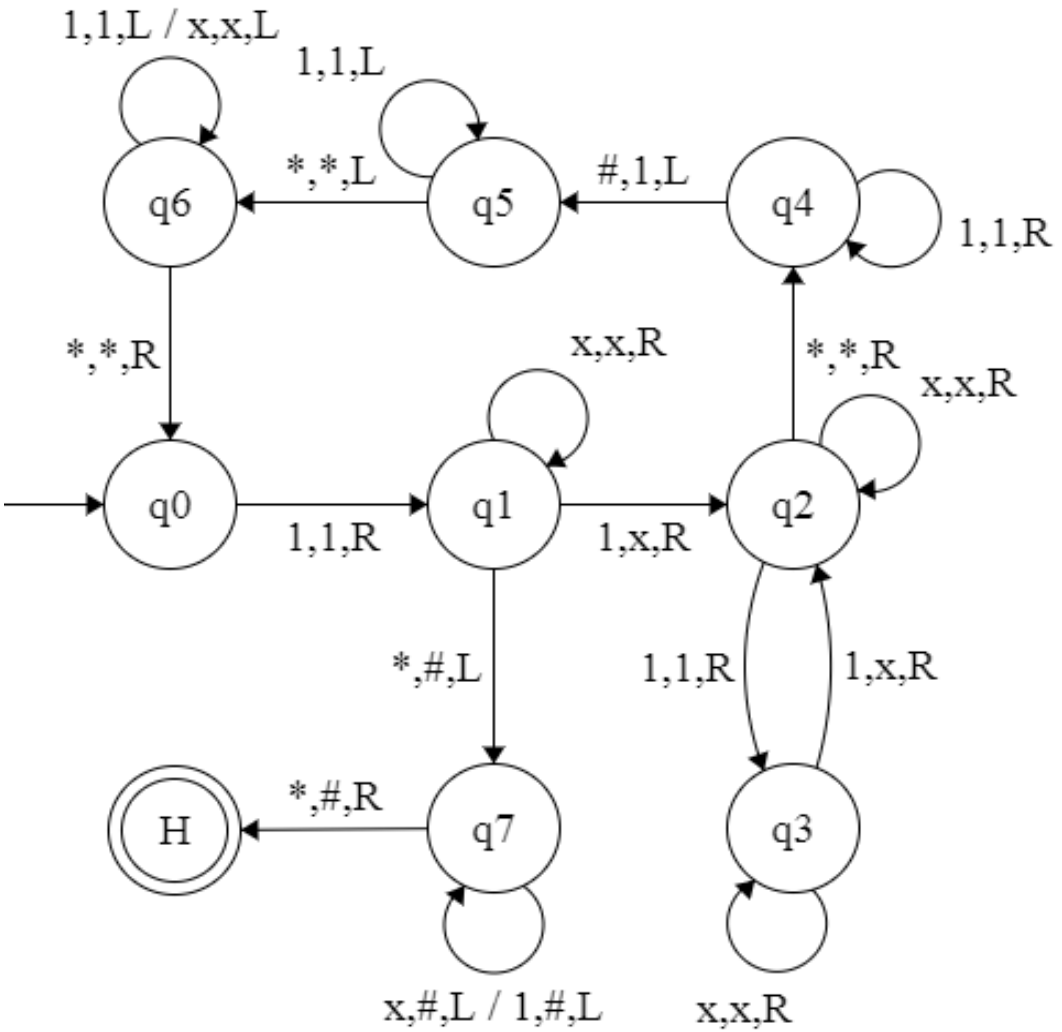
n = ε

n = 11

Design a TM for a partially computable function

$f(n) = \log_2 n$





Try to run the machine by hand on Test inputs:

- ε (shouldn't accept)
- 1 (should accept)
- 11 (should accept)
- 111 (shouldn't accept)
- 111111 (shouldn't accept)
- 11111111 (should accept)

Basis of Computation

Software or Hardware

Why?

Turing Machines – the formalism of algorithms (that we have studied so far)

Is an "unprogrammable" piece of hardware

A Limitation of Turing Machines:

Turing Machines are "hardwired"

they execute only one program

Real Computers are re-programmable

We will see TMs are also software

- There is a generic TM that can be programmed,
- in about the same way as a general purpose computer can,
- to solve any problem that can be solved by a TM.

A modern computer is also "hard wired"

But it is completely flexible

Its job is to execute the instructions stored in its memory, & these can represent any conceivable algorithm.

Universal Turing Machine &

Halting Problem of Turing Machine

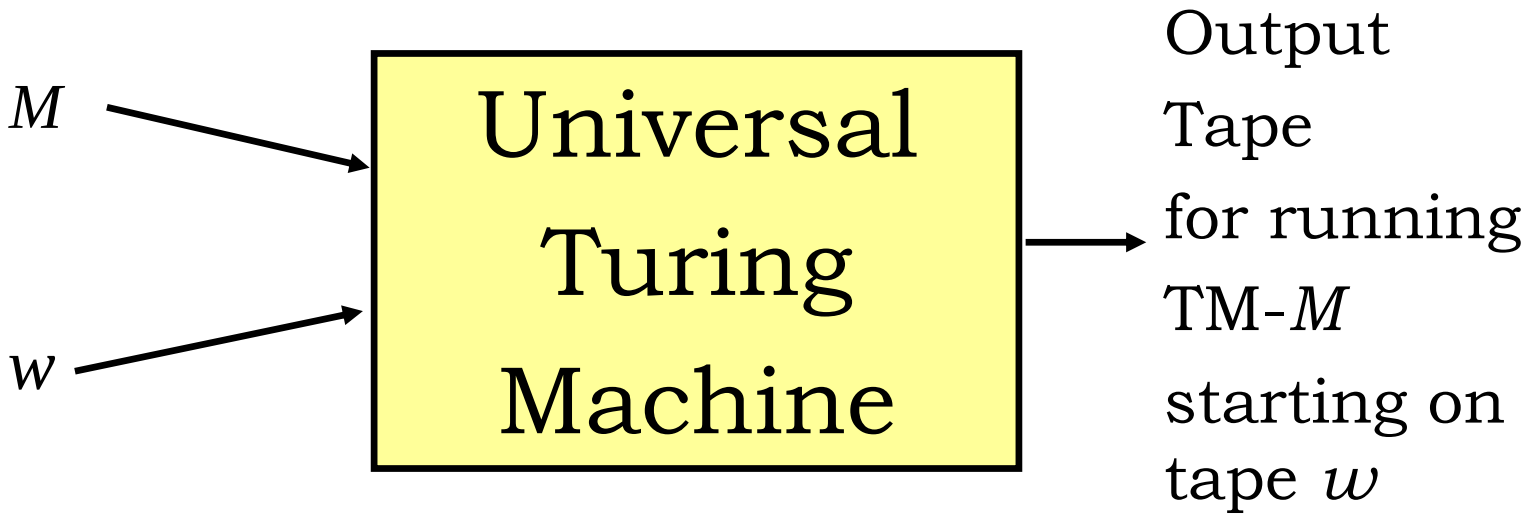
By:

Dr. Sandeep Rathor

Universal TM

Input : < Description of some TM <M, w >

Output: result of running M on w



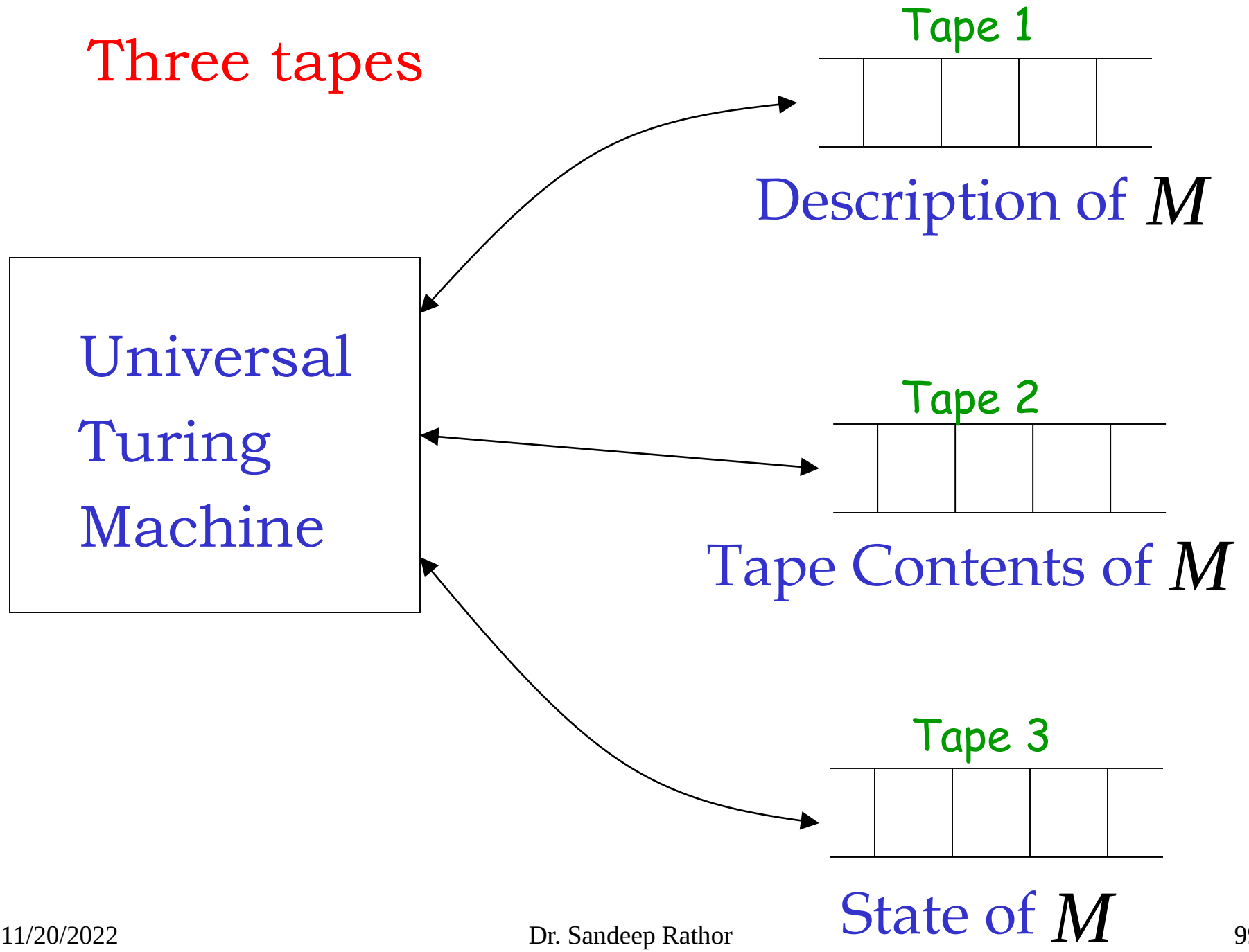
Universal Turing Machine

simulates any other Turing Machine M

Input of Universal Turing Machine:

Description of transitions of M

Initial tape contents of M


Three tapes
Tape 1
Description of M
Universal Turing Machine
Tape 2
Tape Contents of M
Tape 3
State of M

Halting Problem of TM

- Halting means *Terminating*
- Unsolvable Problem

Given a program and an input to the program, determine if the program will eventually stop when it is given that input.

- Given a TM M and input string w.
- Is there an "Algorithm" to decide whether M halts on input w or not?
- It is not recursive and it is undecidable.

- Halting means that, program on certain input will accept it & halt or reject it & halt and never go into an infinite loop.
- So, can we have an algorithm that will tell that the given program will halt or not?
- In terms of TM, will it terminate when run on some machine with some particular given input string.

- There is no *algorithm/ procedure* that can determine whether an arbitrary TM, will halt on an arbitrary input string.

Recursively Enumerable
&
Recursive Languages

By:
Dr. Sandeep Rathor

Definition:

A language is **recursive,** if some Turing machine accepts it and halts on any input string

In other words:

A language is recursive if there is a membership algorithm for it

Let L be a recursive language

and M the Turing Machine that accepts it

For string w

if $w \in L$ then M halts in a final state

if $w \notin L$ then M halts in a non-final state

Remark:

- A Problem/ Language with two answers (yes/no) is decidable.
- A Problem / Language is undecidable if it is not decidable.

Definition:

A language is **recursively enumerable** if some Turing machine accepts it

Remark:
Only acceptable case is in RE

Let L be a recursively enumerable language

and M the Turing Machine that accepts it

For string w:

if $w \in L$ then M halts in a final state

if $w \notin L$ then M halts in a non-final state

or loops forever

Complexity and Computability

By:
Dr. Sandeep Rathor

Complexity and Computability

- A formal "measure" of how efficient algorithm or program, is called Complexity.

- A formal and intuitive description of computation is called computability.

- A complexity class is a collection of problems which can be solved by some computational model.

- Complexity class P is defined to be the set of problems solved by a TM in polynomial time. (Decidable)
- Ex: Postman problem
- Searching shortest path
- Searching a matching in a graph

Time Complexity

- The **time complexity** **of a TM *M is a function*** denoting the *worst-case number of steps M takes* on any input of length *n*.
  - By convention, *n denotes the length of the input.*
  - Assume we're only dealing with deciders, so there's no need to handle looping TMs.

The Class P

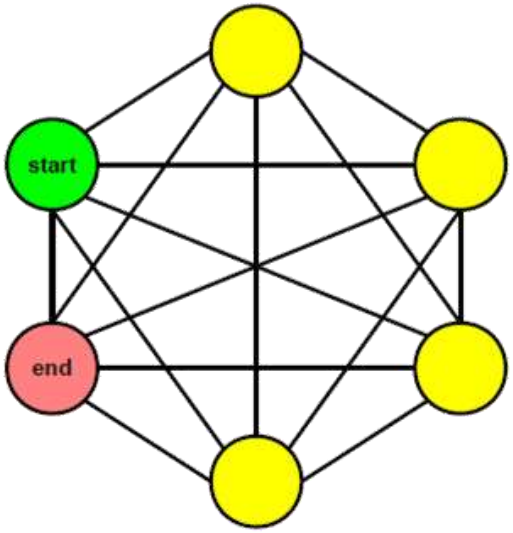
P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine.

$$P = \bigcup_k TIME\left(n^k\right)$$

Class P is important because:

- P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine.
- P is a mathematically robust class. It isn't affected by the particulars of the model of computation that we are using.
- P roughly corresponds to the class of problems that are realistically solvable on a computer.
- When a problem is in P, we have a method of solving it that runs in time n^k ***for some*** **constant** k. Once a polynomial time algorithm has been found for a problem that formerly appeared to require exponential time, some key insight into it has been gained, and further reductions in its complexity usually follow, often to the point of actual practical utility.

What you can't do in polynomial time



How many simple paths are there from the start node to the end node?

How many subsets of this set are there?

An Interesting Observation

- There are (at least) exponentially many objects of each of the preceding types.
- However, each of those objects is not very large.
  - Each simple path has length no longer than the number of nodes in the graph.
  - Each subset of a set has no more elements than the original set.
- This brings us to our next topic...

The Class NP

We can avoid brute-force search in many problems and obtain polynomial time solutions.

However, attempts to avoid brute force in certain other problems, including many interesting and useful ones, haven't been successful, and polynomial time algorithms that solve them aren't known to exist.

Why have we been unsuccessful in finding polynomial time algorithms for these problems?

We don't know the answer to this important question.

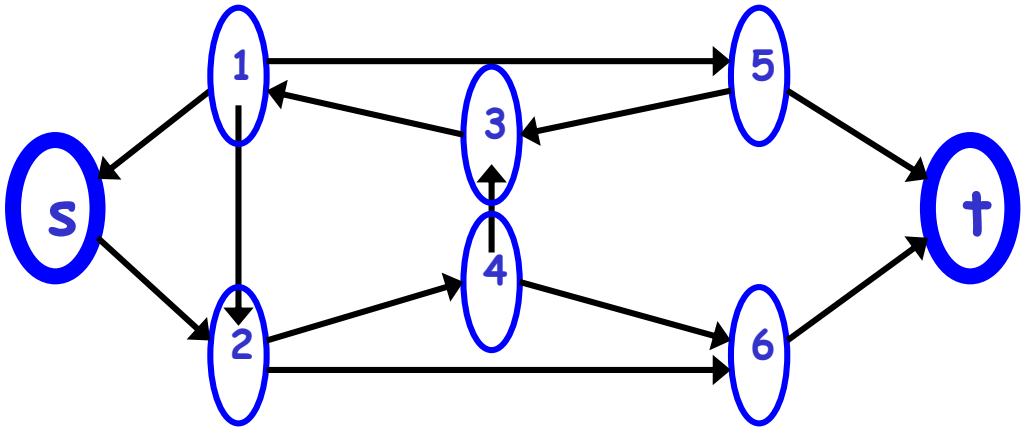
One remarkable discovery concerning this question shows that the complexities of many problems are linked.

A polynomial time algorithm for one such problem can be used to solve an entire class of problems.

A Hamiltonian path in a directed graph G is a directed path that goes through each node exactly once.

We consider a special case of this problem where the start node and target node are fixed.

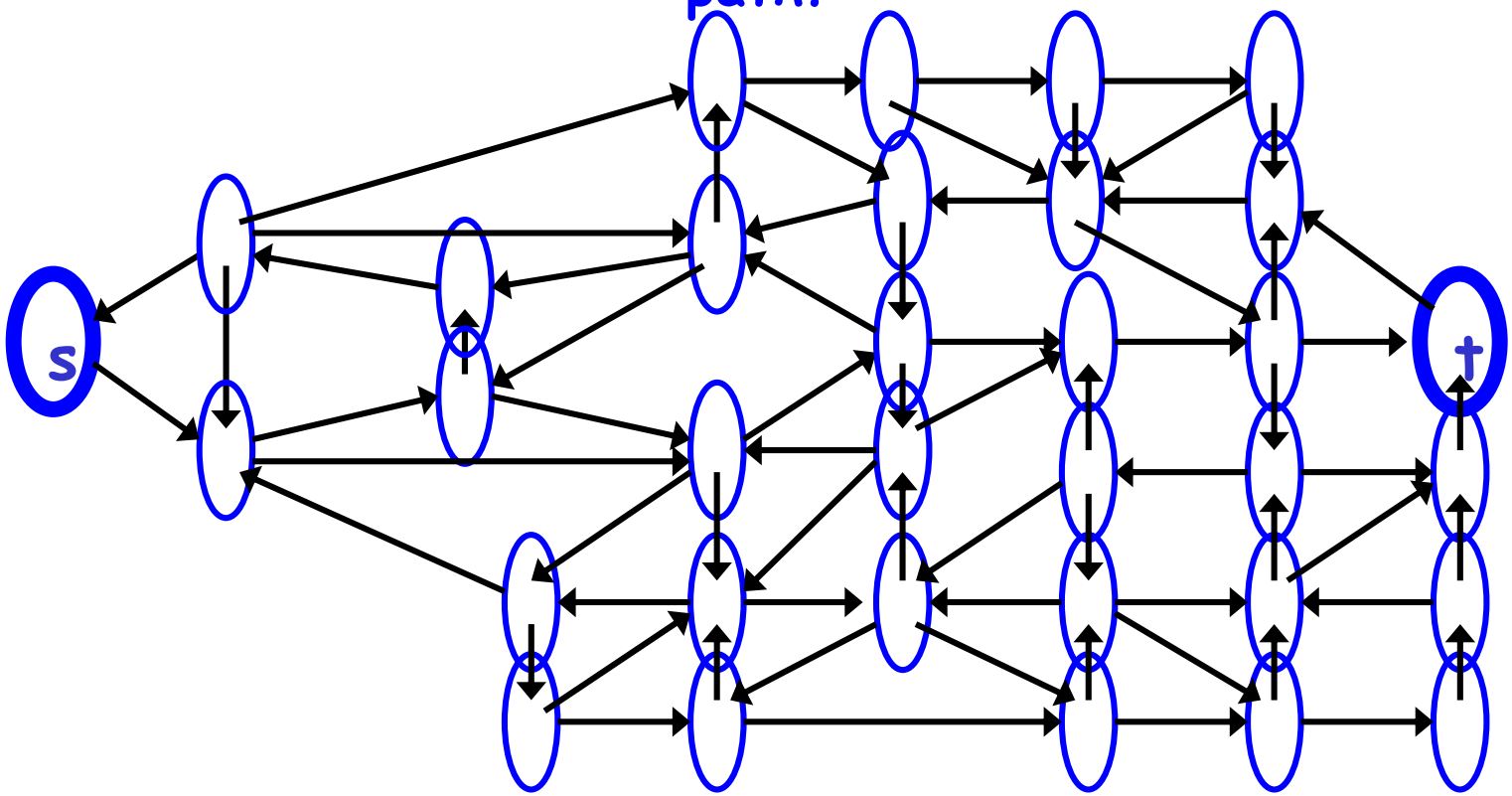
HAMPATH = {<*G*,**s**,**t**> | *G* is a directed graph with a Hamiltonian path from **s** to **t**}



∈ **PATH?**

∈ **HAMPATH?**

Does this graph have a Hamiltonian path?



No fast algorithm for HAMPATH

But

If I somehow discovered such a path (perhaps using the exponential time algorithm), I could easily convince you of its existence, simply by presenting it.

Polynomial Verifiability

Verifying the existence of a Hamiltonian path may be much easier than determining its existence.

Does a Sudoku grid have a solution?

M = "On input ⟨S⟩, an encoding of a Sudoku puzzle:

- **Non deterministically guess how to fill in all the squares.**
- **Deterministically check whether the guess is correct.**
- **If so, accept; if not, reject."**

		7		6		1		
					3		5	2
3			1		5	9		7
6		5		3		8		9
	1						2	
8		2		1		5		4
1		3	2		7			8
5	7		4					
		4		8		7		

P is the class of decision problems that can be efficiently solved.

We now define the complexity class NP that aims to capture the set of problems whose **solutions can be efficiently verified**.

NP is the class of languages that have polynomial time verifiers.

The class NP is important because it contains many problems of practical interest.

The term NP comes from nondeterministic polynomial time and is derived from an alternative characterization by using nondeterministic polynomial time Turing Machines.

Some problems may not be polynomially verifiable!!!

The complement of the *HAMPATH problem.*

Even if we could determine (somehow) that a graph did *not have a Hamiltonian path, we don't* know of a way for someone else to verify its nonexistence without using the

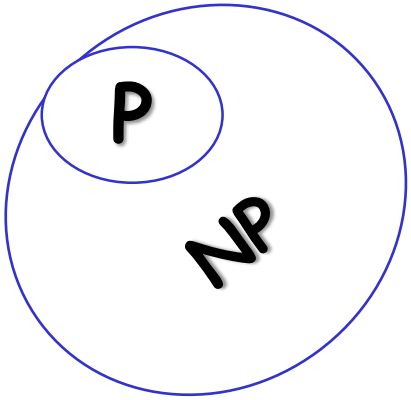
same exponential time algorithm for making the determination in the first place.

Problems in Class NP

- $P \subseteq NP$
- The **decision version of the travelling salesman problem** is in NP. Given an input matrix of distances between n cities, the problem is to determine if there is a route visiting all cities with total distance less than k. A proof can simply be a list of the cities. Then verification can clearly be done in polynomial time.
- The **decision version of the integer factorization problem**: given integers n and k, is there a factor f with $1 < f < k$ and f dividing n?
- The **Subgraph isomorphism** problem of determining whether graph G contains a subgraph that is isomorphic to graph H.

The question of whether P = NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics.

Most researchers believe that the two classes are not equal because people have invested enormous effort to find polynomial time algorithms for certain problems in NP, without success.



OR

P = NP

Researchers also have tried **proving that the classes are unequal**, but that would entail showing that no fast algorithm exists to replace brute-force search. Doing so is presently beyond scientific reach.

An important advance on the P versus NP question came in the early 1970s with the work of Stephen Cook and Leonid Levin.

They discovered certain problems in NP whose individual complexity is related to that of the entire class.

If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable.

These are the NP Complete Problems

"NP-Complete" comes from:

- **Nondeterministic Polynomial**
- **Complete - "Solve one, Solve them all"**

There are more NP-Complete problems than provably intractable problems.

A language B is NP-complete if it satisfies two conditions:

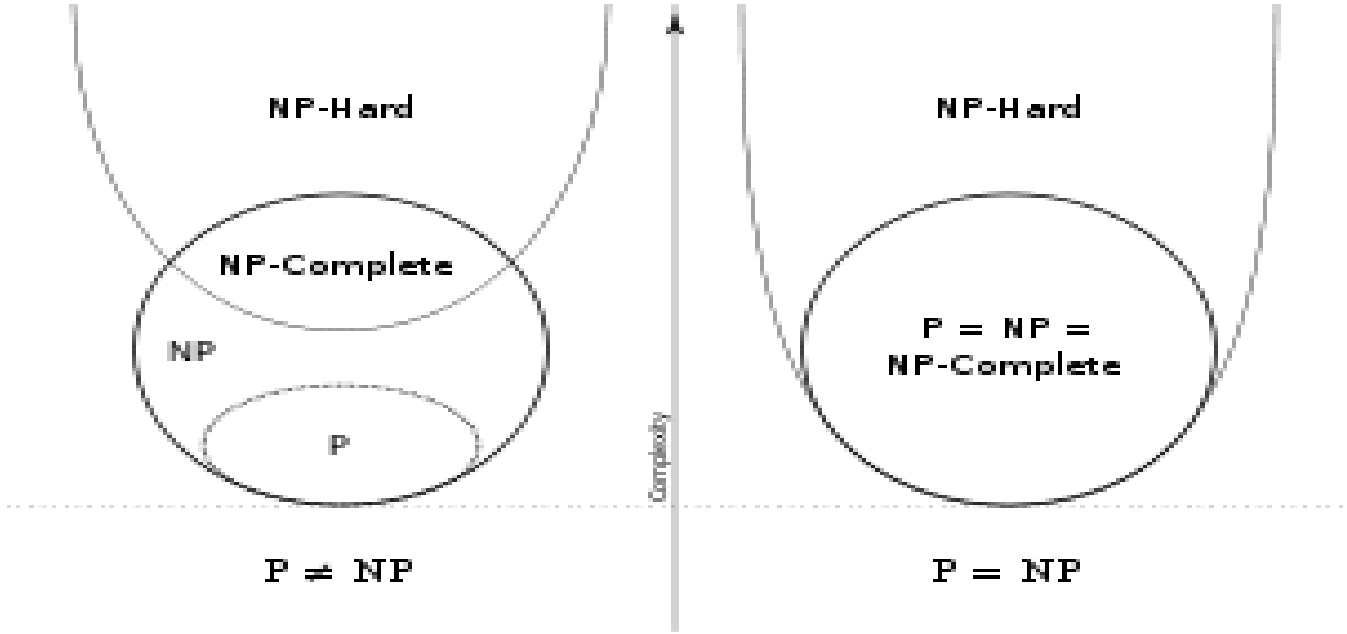
1. B is in NP, and

2. Every A in NP is polynomial time reducible to B.

NP Hard

NP-hard (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, **"at least as hard as the hardest problems in NP".**

A problem H is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H (i.e., $L \leq_P H$).



All NP-complete problems are NP-hard but some NP-hard problems are known not to be NP-complete

NP-complete ⊂ NP-Hard

When

- **a decision version of a combinatorial optimization problem is proved to belong to the class of NP-complete problems,**
- **then the optimization version is NP-hard.**

For example,

- **"is there a Hamiltonian cycle with length less than k" is NP-complete**: it is easy to determine if a proposed certificate has length less than k.

- **The optimization problem, "what is the shortest tour?", is NP-hard**, since there is no easy way to determine if a certificate is the shortest.

Examples of NP Hard Problems

- The decision **subset sum problem**, which is this: given a set of integers, does any non-empty subset of them add up to zero? That is a NP Hard decision problem, and happens to be NP-complete.
- The optimization problem of finding the least-cost cyclic route through all nodes of a weighted graph. This is commonly known as the traveling salesman problem.

- There are decision problems that are NP-hard but not NP-complete, for example the **halting problem**.

How ?

It is easy to prove that the halting problem is *NP-hard* but not *NP-complete*.

Prove Halting problem is not *NP-complete*.

When do we say that a problem is NP Complete ?

For a problem to be NP complete, it must be decidable – which halting problem is not.

Let G be a CFG in Chomsky Normal form (CNF). In order To derive a string of terminals of length n , the number of productions to be used is

(a) 2n + 1
(b) 2n - 1
(c) 2n
(d) None of these

Recursively enumerable languages are not closed under

A. Complementation
B. Union
C. Intersection
D. none of these

Which of the following statement is wrong?

A. Every recursive language is recursively enumerable.
B. A language is accepted by FA if and only if it is context free.
C. Recursive languages are closed under intersection
D. A language is accepted by FA if and only if it is right linear.

Which of the following is true?
A. The complement of a recursive language is recursive.
B. The complement of a recursively enumerable language is recursively enumerable.
C. The complement of a recursive language is either recursive or recursively enumerable.
D. The complement of a context-free language is context-free.

If there exists a language L, for which there exists a TM, T,
that accepts every word in L and either rejects or loops for every word that is not in L, is called
A. Recursive
B. Recursively enumerable
C. NP-HARD
D. none of these

Universal TM influenced the concept of
A. stored program computers.
B. interpretative implementation of programming language.
C. computability.
D. all of these.

Which of the following statements is/are true?
I. Recursive languages are closed under complementation.
II. Recursively enumerable languages are closed under union.
III. Recursively enumerable languages are closed under complementation.
A. I only
B. I and II
C. II and III
D. III only

Type 0 Grammar is accepted by
A. Turing Machine
B. PDA
C. LBA
D. DFA

Closure Properties of Languages

Property	Regular	CFL	DCFL	CSL	Recursive	RE
Union	Yes	Yes	No	Yes	Yes	Yes
Intersection	Yes	No	No	Yes	Yes	Yes
Set Difference	Yes	No	No	Yes	Yes	No
Complementation	Yes	No	Yes	Yes	Yes	No
Intersection with a regular lang.	Yes	Yes	Yes	Yes	Yes	Yes
Concatenation	Yes	Yes	No	Yes	Yes	Yes
Kleen Closure	Yes	Yes	No	Yes	Yes	Yes
Kleen Plus	Yes	Yes	No	Yes	Yes	Yes
Reversal	Yes	Yes	No	Yes	Yes	Yes
Homomorphism	Yes	Yes	No	No	No	Yes
ϵ-free Homomorphism	Yes	Yes	No	Yes	Yes	Yes
Inverse Homomorphism	Yes	Yes	Yes	Yes	Yes	Yes
Substitution	Yes	Yes	No	No	No	Yes

Thank You