# Complexity theory
# chapter zero

# General Introduction

- Complexity theory is part of the theory of computation dealing with the resources required during computation to solve a given problem, predominantly:
  - time (how many steps it takes to solve a problem) and
  - space (how much memory it takes).
- Complexity theory  is an extensive topic that deals with the efficiency of computation.
- Complexity theory forms a basis for the classification and analysis of combinatorial algorithms.
- Complexity theory differs from computability theory, which deals with whether a problem can be solved at all or not, regardless of the resources required.

# Introduction

- Complexity theory helps computer scientists to relate and group problems together into complexity classes. Because:
- Complexity theory specifies the definition of a **meaningful classification of problems according to their computational difficulty** and the derivation of results according to such methods. That is:
- Complexity theory provides an explanation of why certain problems have no practical solutions and provides a way of anticipating difficulties involved in solving problems of certain types.
- The theory relies on robust classification and is not easily altered by minor changes in the model.
- Such theory provides an excellent framework in computer science, which deals with information, its creation & processing and with the systems that perform it.

## Introduction

- The goal of complexity theory is to provide mechanisms for classifying combinatorial problems and measuring the computational resources necessary to solve them.

---

- The classification is quantitative and is intended to investigate what resources are necessary (lower bounds) and what resources are sufficient (upper bounds) to solve various problems.
- This classification should not depend on a particular computational model but rather should measure the intrinsic difficulty of a problem.

- Moreover, complexity theory is interested not in the merely computable but in problems that are efficiently computable.

Introduction

- **Computational complexity theory** attempts to classify computational problems based on the amount of resources required by algorithms to solve them.
- Computational **problems** come in various essences:

I. Decision problem

Example: Is vertex t reachable from vertex s in graph G? (…output is YES/NO)

II. Search problem

Example: Find a satisfying assignment of a Boolean formula, if it exists

IV. Counting problem

Example: Find the number of cycles in a graph

V. Optimization problem

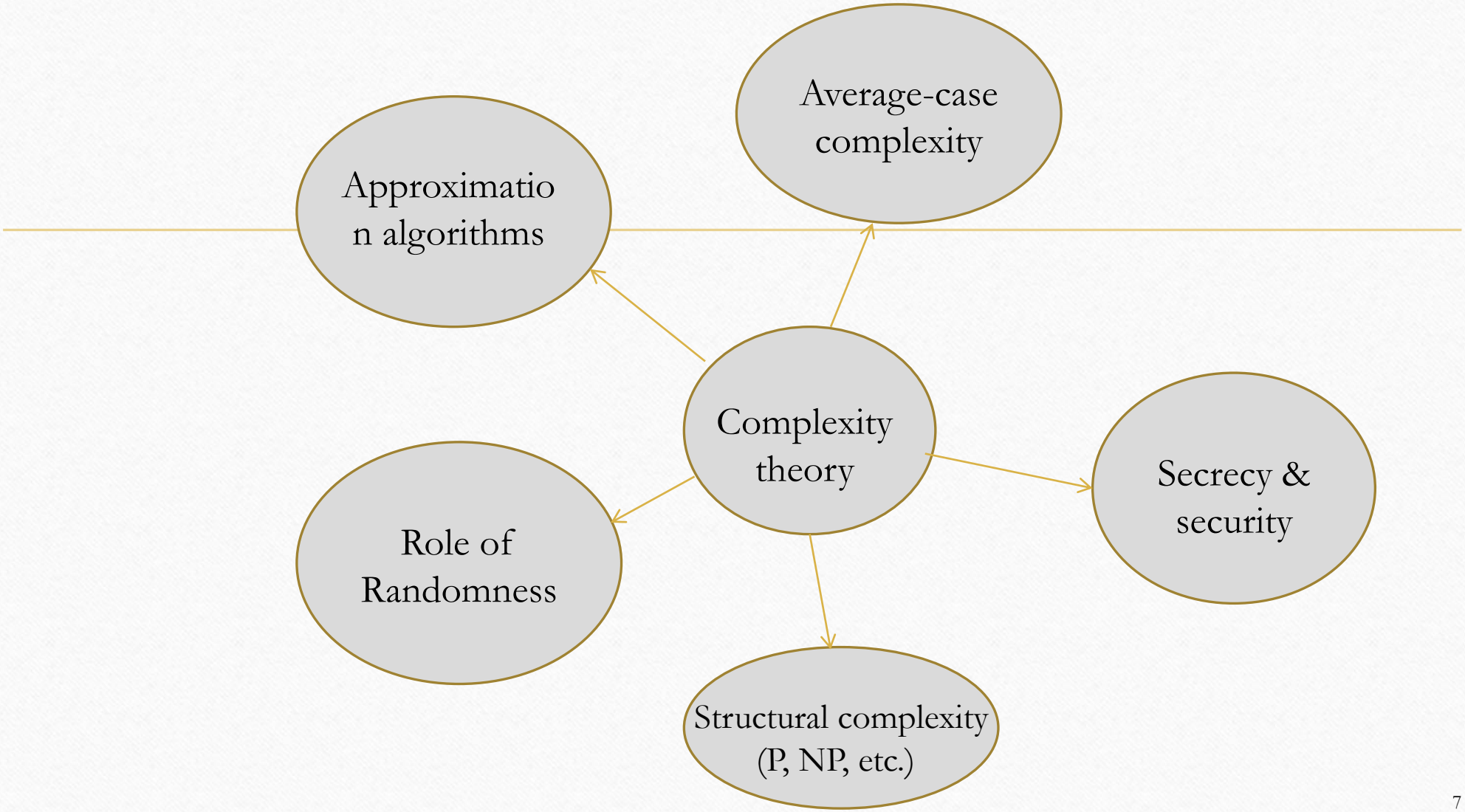
Example: Find a minimum size vertex cover in a graph

Con…

**Algorithms** are methods of solving problems that are studied using formal models of computation. E.g. Turing machines contain a memory with head (like a RAM) & finite control (like a processor)

- **Computational resources** (required by models of computation) can be:
  - Time (bit operations: machines like TM, only permit one bit at a time to be read /written)
  - Space (memory cells)
  - Random bits (algorithms that use random bits are called randomized algorithms)
  - Number of gates in a circuit (used in circuit complexity)
  - Number of processors (used in parallel computing).
  - Amount of communication (used in communication complexity i.e. bit exchanges)

# Related topics in complexity theory



# Structural Complexity

- Classes P, NP, NP-completeness.
  - How hard is it to check satisfiability of a Boolean formula that has exactly one or no satisfying assignment?
- Space bounded computation.
  - How much space is required to check s-t connectivity?
- Counting complexity.
  - How hard is it to count the number of perfect matchings in a graph?
- Polynomial Hierarchy.
  - How hard is it to find a minimum size circuit computing the same Boolean function as a given Boolean circuit?
- Boolean circuits and circuit lower bounds.
  - A central topic in classical complexity theory; Proving P≠NP boils down to showing circuit lower bounds.

# Role of Randomness in Computation

- Probabilistic complexity classes.
  - Does randomization help in improving efficiency?
  - Quicksort has O(n log n) expected time but O(n^2) worst case time.
  - Can SAT be solved in polynomial time using randomness?
- Probabilistically Checkable Proofs (PCPs).
  - Unconditional hardness of approximation results

- Average-case Complexity
- Distributional problems.
  - How hard is it to solve the clique problem on inputs chosen from a "real-life" distribution?
- Hardness amplification: From weak to strong hardness.
  - In cryptographic applications, we need hard functions for secure encryptions.

# Complexity Classes

- A complexity class is a set of problems of related resource based complexity. or
- A complexity class is a set of functions that can be computed within a given resource. It means, it is a collection of sets that can be accepted by Turing machines with the same resources.
- Complexity classes are defined by the following factors:
  - The type of computational problem:
    - The most commonly used problems are decision problems.
    - However, complexity classes can be defined based on function problems, counting problems, optimization problems, promise problems, etc.
  - The model of computation: The most common model of computation is the deterministic TM, but many complexity classes are based on non-deterministic TMs
  - The resource (or resources) that are being bounded and the bounds: These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth", etc.

# Complexity measures

- A complexity measure quantifies the use of a particular computational resource during execution of a computation.
- The two most important and common measures are time, the time it takes a program to execute, and space, the amount of storage used during a computation. i.e.
- Time Complexity: It is a measure of how long a computation takes to execute.
- As far as Turing machine is concerned, this could be measured as the number of moves which are required to perform a computation.
- In the case of a digital computer, this could be measured as the number of machine cycles which are required for the computation.
- Space Complexity: It is a measure of how much storage is required for a computation.
- In the case of a Turing machine, the obvious measure is the number of tape squares used, for a digital computer, the number of bytes used.

# Complexity measures

- For a precise definition of what it means to solve a problem using a given amount of time and space, a computational model such as the deterministic Turing machine is used.
- The following are three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size.
  - Best-case complexity: This is the complexity of solving the problem for the best input of size *n*.
  - Worst-case complexity: This is the complexity of solving the problem for the worst input of size *n*.
  - Average-case complexity: This is the complexity of solving the problem on an average

# Chapter one

- Turing Machine (TM)
  - Standard TM
  - Construction of TMs

- The theory of computing provides computer science with concepts, models, and formalisms for reasoning about both the resources needed to carry out computations and the efficiency of the computations that use these resources.
- The basic model of computation for our study is the Turing machine, and for complexity theory, is the multi tape Turing machine.
- However, these subjects should **not depend** too much on the choice of computational model. To this end, we will discuss Church's thesis
- Church's thesis states that every computational device can be simulated by a Turing machine.
- One of the topics in this chapter is the simulation of RAMs by Turing machines.

- Church's Thesis
  - Church's thesis states that every "effective computation," or "algorithm," can be programmed to run on a Turing machine.
  - Every "computational device" can be simulated by some Turing machine.

*Church-Turing hypothesis*:

*Every computable function can be computed by a Turing machine.*

One can formulate a more restrictive version of this hypothesis:

*Every efficiently computable function can be efficiently computed by a Turing machine.*

# Church-Turing Thesis

Every computer algorithm can be implemented as a Turing machine

Therefore, C, C++, Prolog, Lisp, Small talk, and Java programs can be simulated in Turing machines

**Definition**: a programming language is **Turing-complete** if it is equivalent to a Turing machine.

**Discussion**: Are you sure that such a simple model can simulate my C++ program for computing the Minimum Spanning Tree?

# Remember that

- Finite Automata accepts regular languages only

  For example, $\{a^n b^n : n = 0, 1, \ldots\}$ is not regular

- Pushdown Automata accepts context-free languages only

  For example, $\{a^n b^n c^n : w \in \sum^*\}$ is not context-free

- We can easily write an algorithm (in C) recognizing if a sequence of characters have the form $a^n b^n c^n$ or not

# Introduction of Turing machine

- Introduced by Alan Turing in 1936.
- A simple mathematical model of a computer.
- It is capable of performing any calculation which can be performed by any computing machine.
- A Turing machine (TM) is a finite-state machine with an infinite tape and a tape head that can read or write one tape cell and move left or right.
- It normally accepts the input string, or completes its computation, by entering a final or accepting state.

# Introduction of Turing machine

- Is similar to a finite automaton but with an unlimited and unrestricted memory as well as the movement of head is both direction (either left or right)
- Turing machine is a much more accurate model of a general purpose computer.
- A Turing machine can do everything that a real computer can do.
- In a very real sense, these problems are beyond the theoretical limits of computation.
- The Turing machine model uses an infinite tape as its unlimited memory.
- It has a tape head that can read and write symbols and move around on the tape.
- The head is always located on one of the cells of the tape.

# Introduction of Turing machine

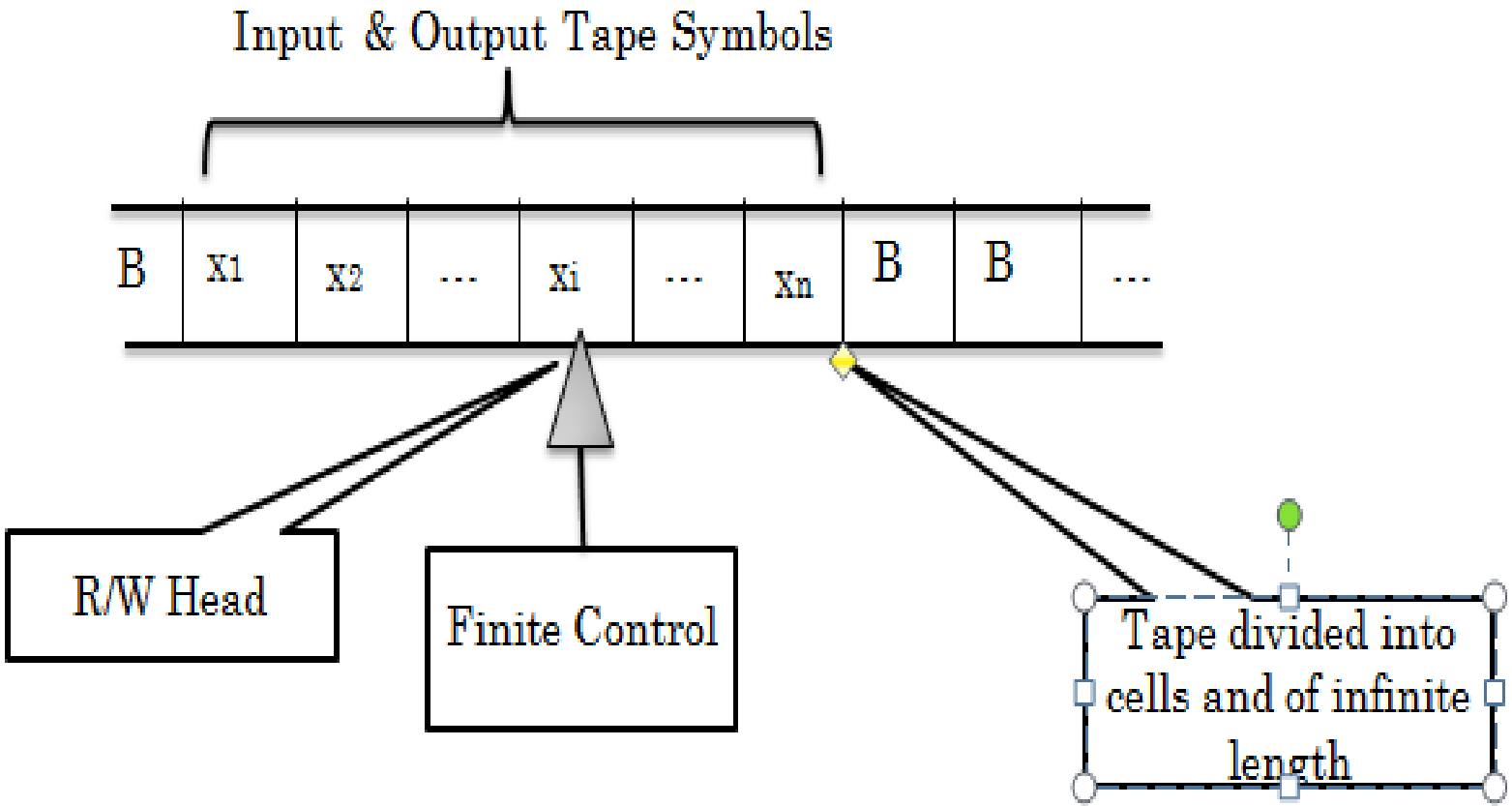
- Initially the tape contains only the input string and is blank everywhere else.
- If the machine needs to store information, it may write this information on the tape.
- To read the information that it has written, the machine can move its head back over it.
- The machine continues computing until it decides to produce an output.
- The outputs accept and reject are obtained by entering designated accepting and rejecting states.

- If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.

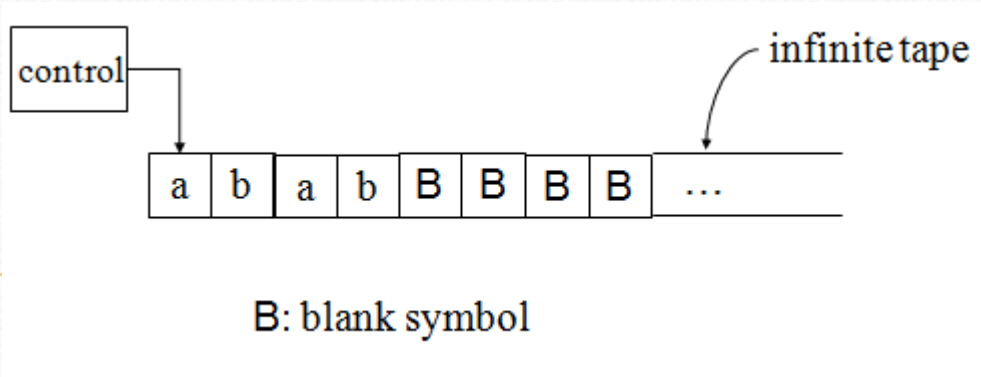
# Introduction of Turing machine

- A transition function $\delta$ tells the head what to do: when being in state $q \in Q$, and reading a symbol $\alpha \in \Sigma$, the transition function specifies:
  - (1) with what to over write the symbol on the tape,
  - (2) whether to move right or left one cell on the tap, and
  - (3) which new state $q'$ to enter .
- The Turing Machine always starts in a predefined start states, and as soon as it reaches a predefined finish state f, it terminates
- Q: Is it possible to design a formal model of a computational device that capture the capabilities of **any** algorithm?
  - Alan Turing, 1940's: **Yes!**

# THE TURING MACHINE MODEL



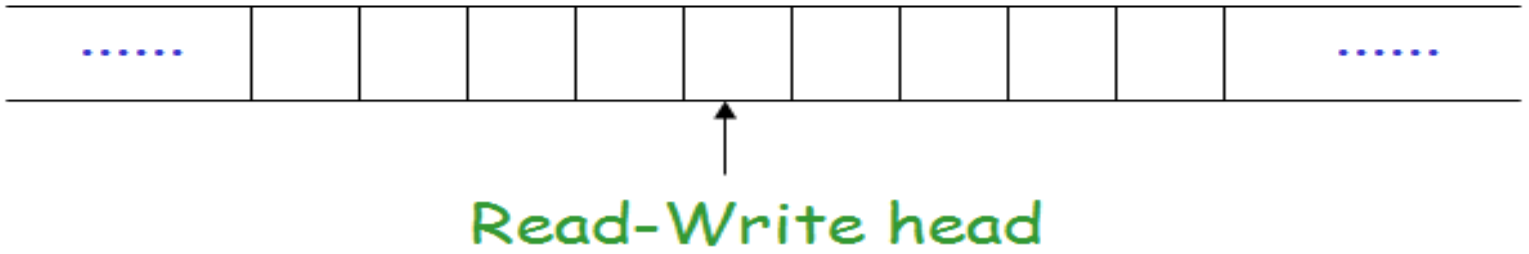
# Description




1. A Turing machine can both write on the tape and read from it.
2. The read-write head can move both to the left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting take immediate effect.

# The Tape

No boundaries -- infinite length

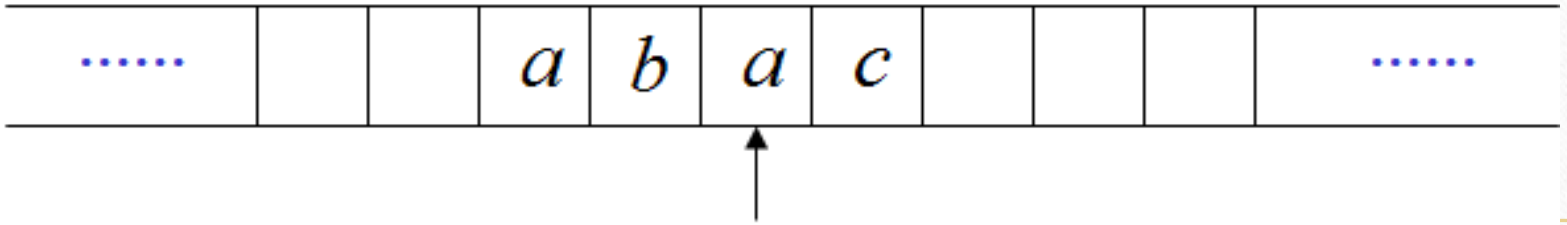



The head at each transition (time step):

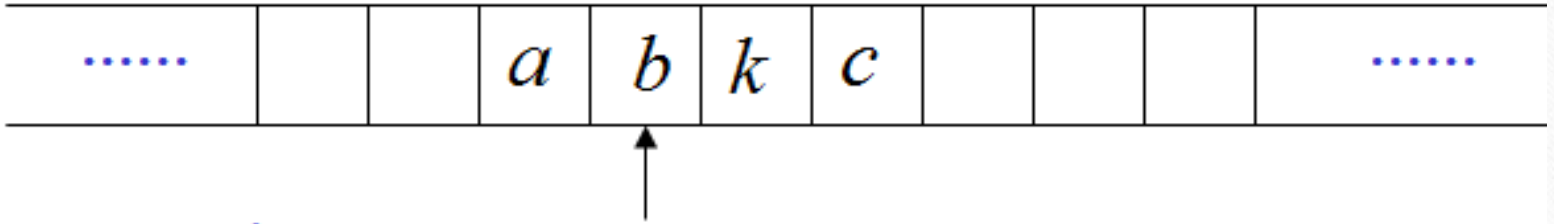
1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

Example:

Time 0

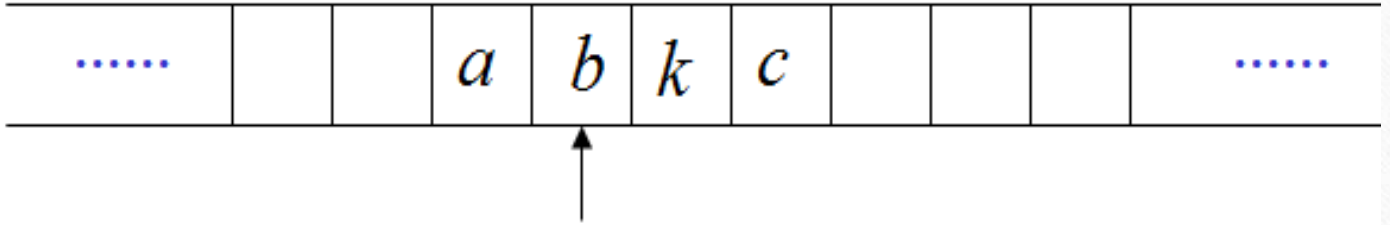



Time 1

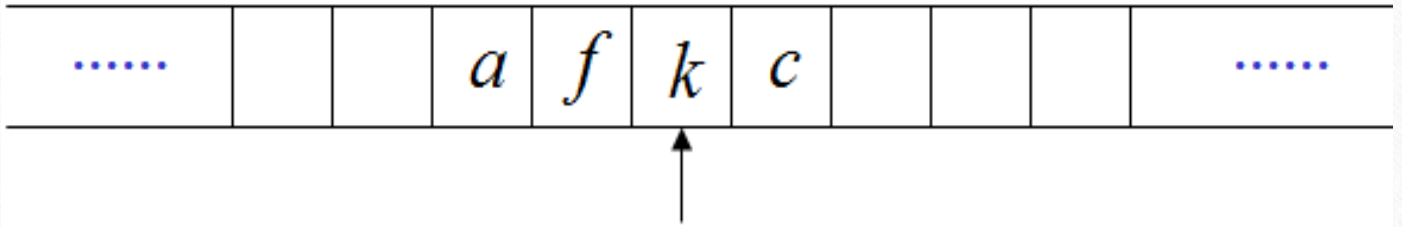



1. Reads $a$
2. Writes $k$
3. Moves Left

Time 1




Time 2




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

# Formal definition of a Turing machine

- The heart of the definition of a Turing machine is the transition function $\delta$ because it tells us how the machine gets from one step to the next.
- For a Turing machine, $\delta$ takes the form: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$. That is, when the machine is in a certain state q and the head is over a tape square containing a symbol a, and if $\delta(q, a) = (r, b, L)$, the machine writes the symbol b replacing the a, and goes to state r.
- The third component is either L or R and indicates whether the head moves to the left or right after writing.
- In this case, the L indicates a move to the left.

# Formal definition of a Turing machine

## Representation of Turing Machine

- **Turing Machine is represented by- $M=(Q, \Sigma, \Gamma, \delta, q0, q_{accept}, q_{reject})$,**
- Where

---

- **Q** is the finite state of states
- **Σ** a set of τ not including B, is the set of input symbols,
- **Γ** is the finite state of allowable tape symbols & contains blank symbol **B**,
- **δ** is the next move function, a mapping from $Q \times \tau$ to $Q \times \tau \times \{L,R\}$
- $q0$ in Q is the start state,
- $q_{accept} \in Q$ is the accept state, and
- $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

# Computation of Turing machine

- As a Turing machine computes, it may halt with ‘accept’ or ‘reject’, or it may never halt!
- During computation of Turing machine, changes occur in

$$\delta(q, X) = (p, Y, R/L)$$

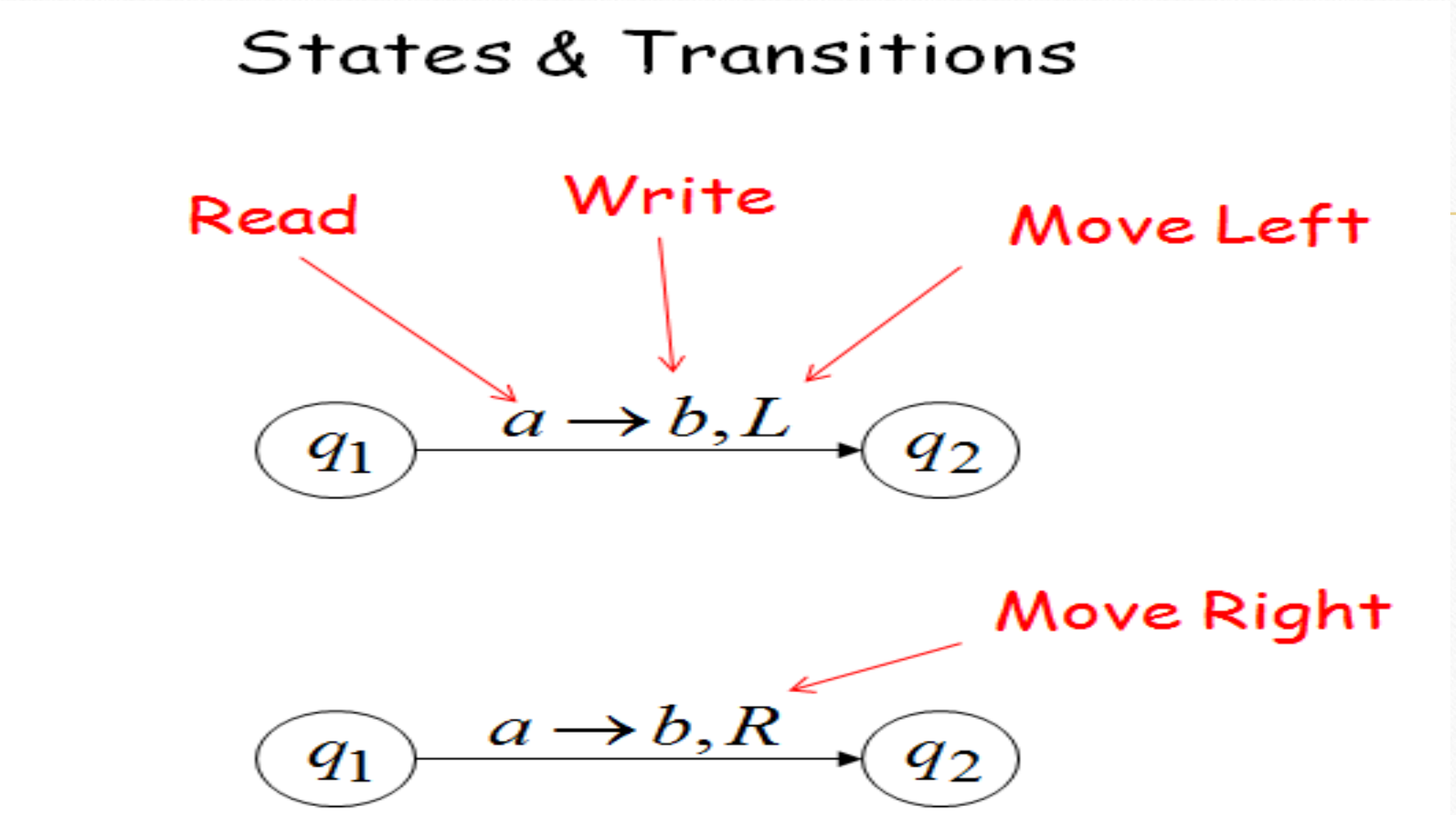
1. the current state,
2. the current tape contents,
3. the current head location.

- A setting of these above three items is called a “configuration” of the Turing machine.

# More on Computation of Turing machine

- A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q0, q_{accept}, q_{reject})$ computes as follows.
- Initially, M receives its input $w = w1\ w2 \ldots wn \in \Sigma*$ on the leftmost n squares of the tape, and the rest of the tape is blank (i.e., filled with blank symbols).
- The head starts on the leftmost square of the tape. Note that $\Sigma$ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input.
- Once M has started, the computation proceeds according to the rules described by the transition function.
- If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L.
- The computation continues until it enters either the accept or reject states, at which point it halts.
- If neither occurs, M goes on forever.

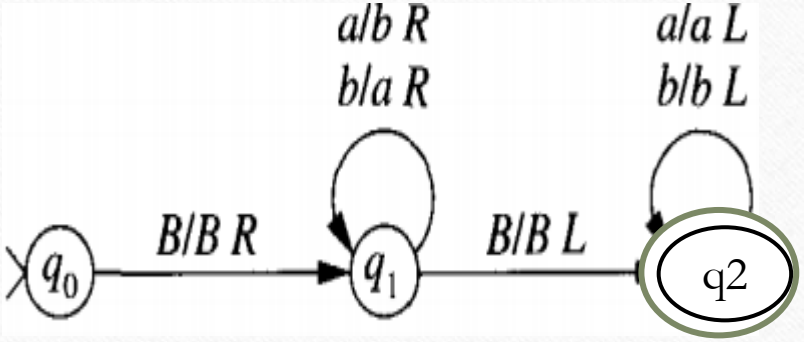
# More on Computation of Turing machine



- Example**:** The transition function of a standard Turing machine with input alphabet {a, b) is given below. The transition from state qo moves the tape head to position one to read the input.
- The transitions in state q1 read the input string and interchange the symbols a and b.
- The transitions in q2 return the machine to the initial position.
- Solution:

| $\delta$ | $B$ | $a$ | $b$ |
|---|---|---|---|
| $q_0$ | $q_1, B, R$ | | |
| $q_1$ | $q_2, B, L$ | $q_1, b, R$ | $q_1, a, R$ |
| $q_2$ | | $q_2, a, L$ | $q_2, b, L$ |

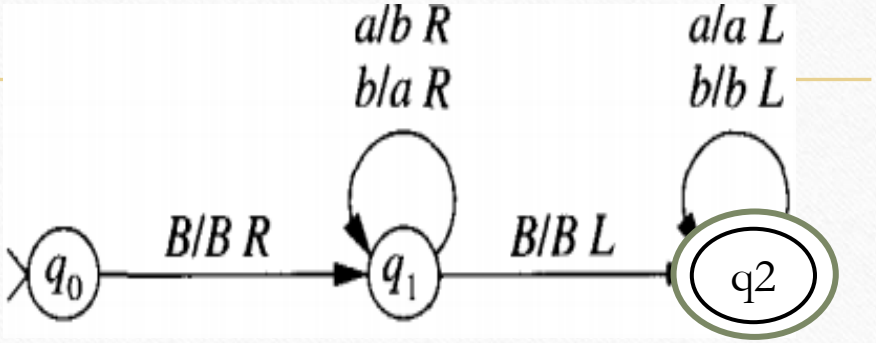
- TM by Transition table



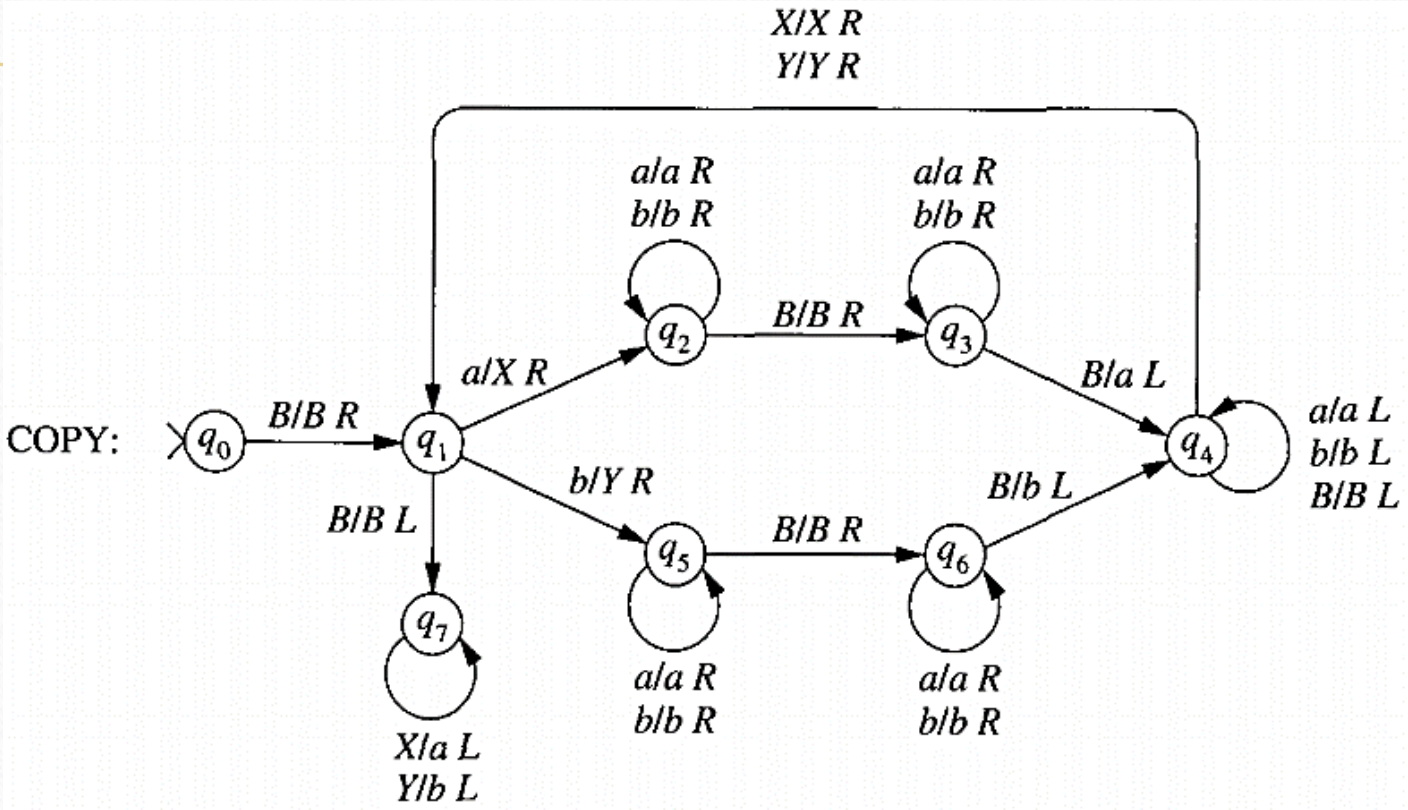

TM by state diagram

- The Turing machine in pervious example interchanges the a's and b's in the input string. Tracing the computation generated by the input string abab yields

$$
\begin{aligned}
& q_0 B a b a b B \\
\vdash\ & B q_1 a b a b B \\
\vdash\ & B b q_1 b a b B \\
\vdash\ & B b a q_1 a b B \\
\vdash\ & B b a b q_1 b B \\
\vdash\ & B b a b a q_1 B \\
\vdash\ & B b a b q_2 a B \\
\vdash\ & B b a q_2 b a B \\
\vdash\ & B b q_2 a b a B \\
\vdash\ & B q_2 b a b a B \\
\vdash\ & q_2 B b a b a B .
\end{aligned}
$$



- Egg2. The Turing machine COPY with input alphabet {a, b} produces a copy of the input string.
- That is, a computation that begins with the tape having the form BuB terminates with tape BuBuB.



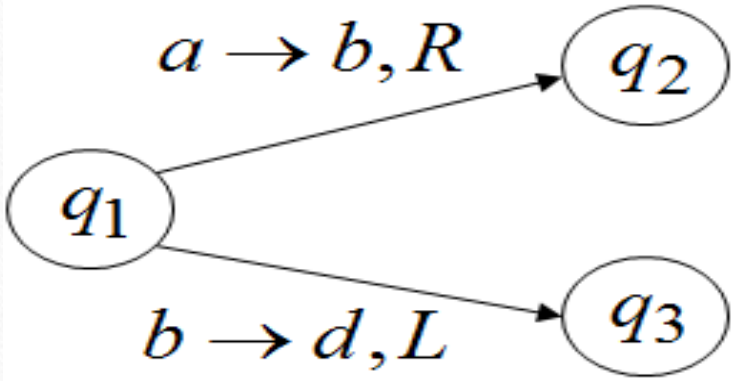

- NB: q7 is final state

- Con…
- The computation copies the input string one symbol at a time beginning with the leftmost symbol in the input. Tape symbols X and Y record the portion of the input that has been copied.
- The first unmarked symbol in the string specifies the arc to be taken from state q1.
- The cycle q1, q2, q3, q4, q1 replaces an a with X and adds an a to the string being constructed.
- Similarly, the lower branch copies a b using Y to mark the input string.
- After the entire string has been copied, the X's and Y's are returned to a's and b's in state q7.

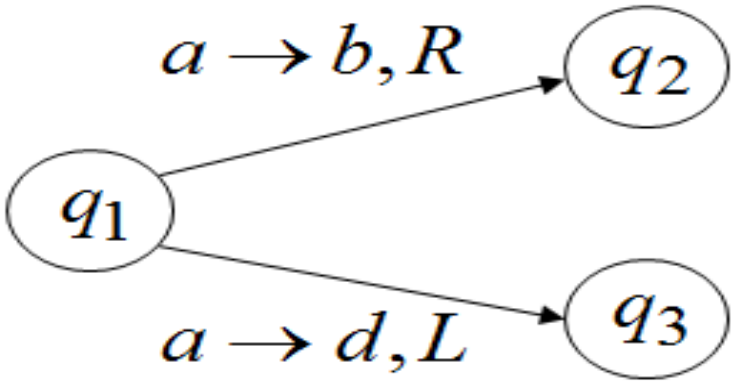
# Determinism

## Turing Machines are deterministic

**Allowed**




**Not Allowed**



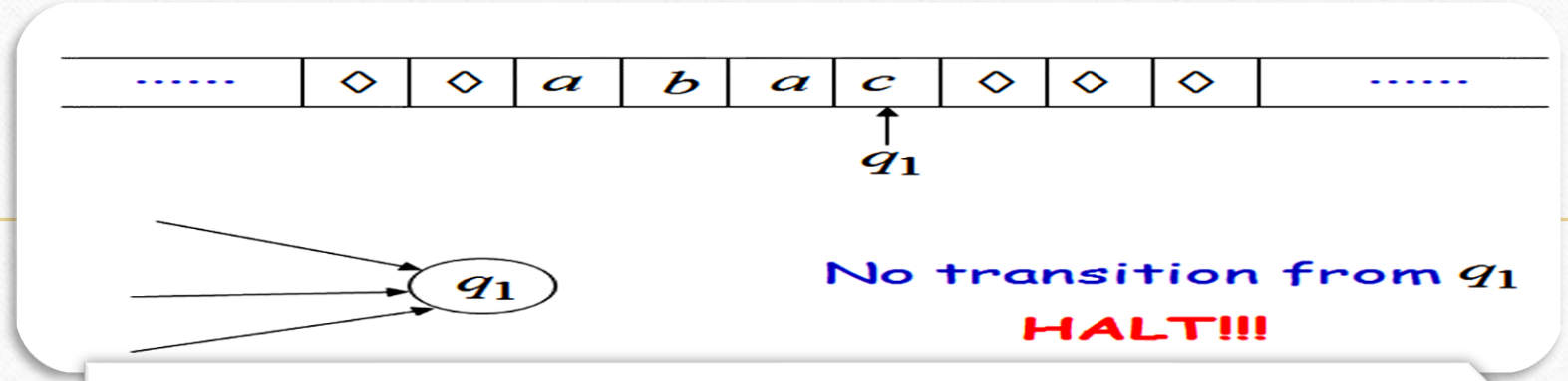

No lambda transitions allowed

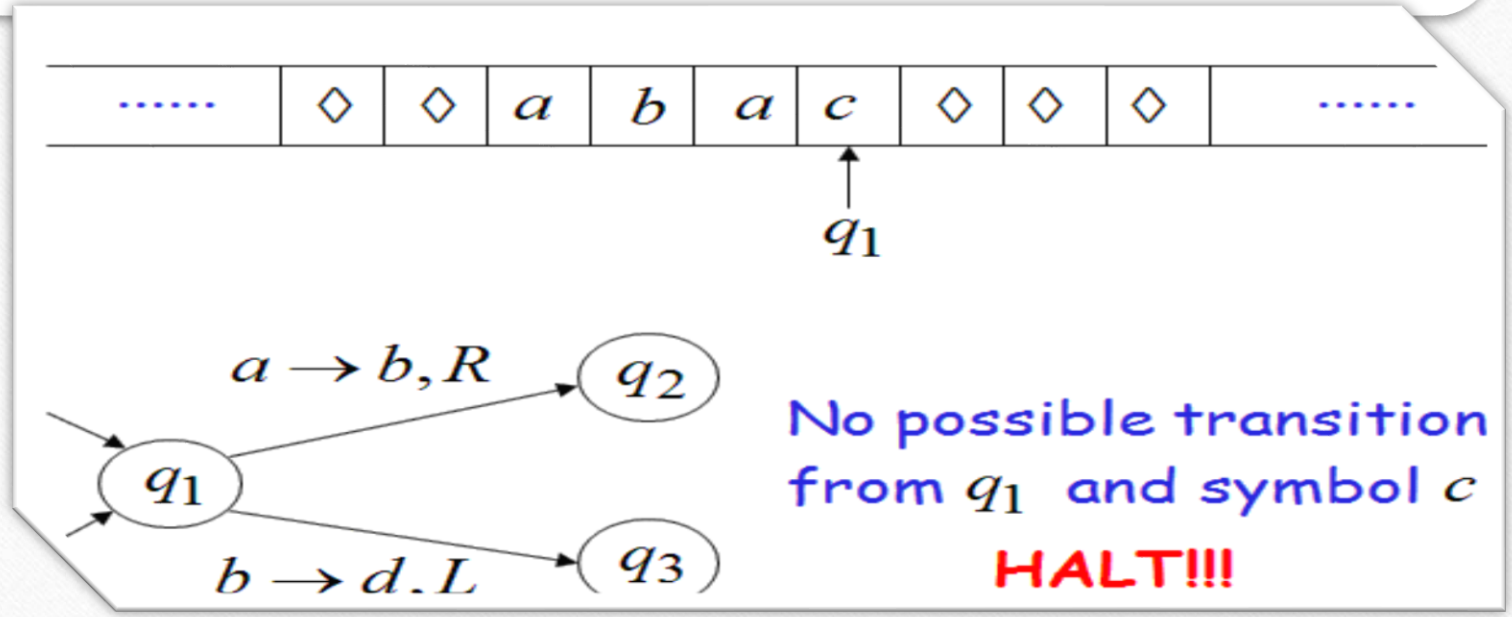
# NON DETERMINISTIC TURING MACHINES

- It is similar to DTM except that for any input symbol and current state it has a number of choices
- A nondeterministic Turing machine allows for the possibility of more than one next move from a given configuration.
- If there is more than one next move, we do not specify which next move the machine makes, only that it chooses one such move.
- A string is accepted by a NDTM if there is a sequence of moves that leads to a final state
- The transaction function $\delta : Q \times \Gamma \longrightarrow 2^{Q \times \Gamma \times \{L, R\}}$
- Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

# Halting

- The machine ***halts*** in a state if there is no transition to follow



# Acceptance

Accept Input string → If machine halts in an accept state

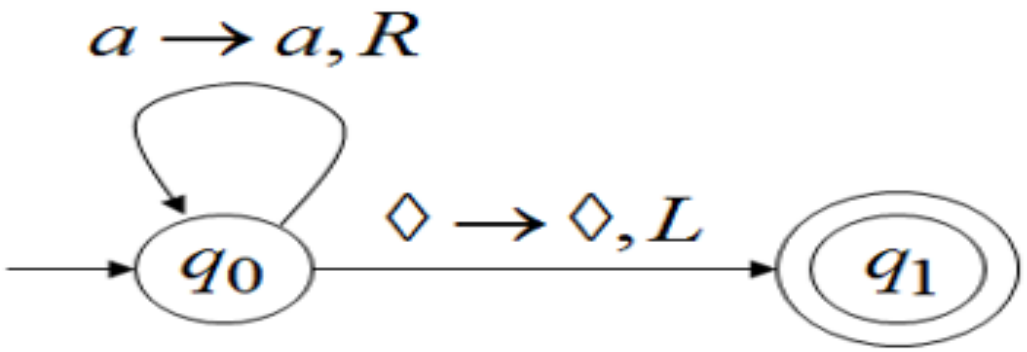
Reject Input string → If machine halts in a non-accept state **or** If machine enters an *infinite loop*

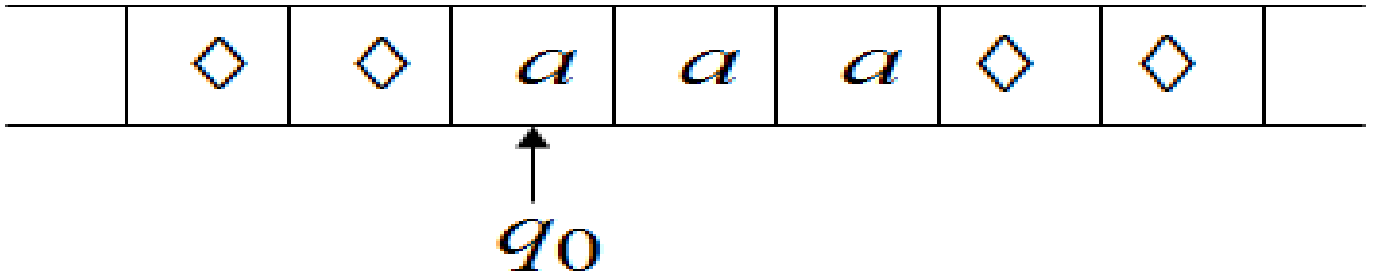
# Turing machine as transducers

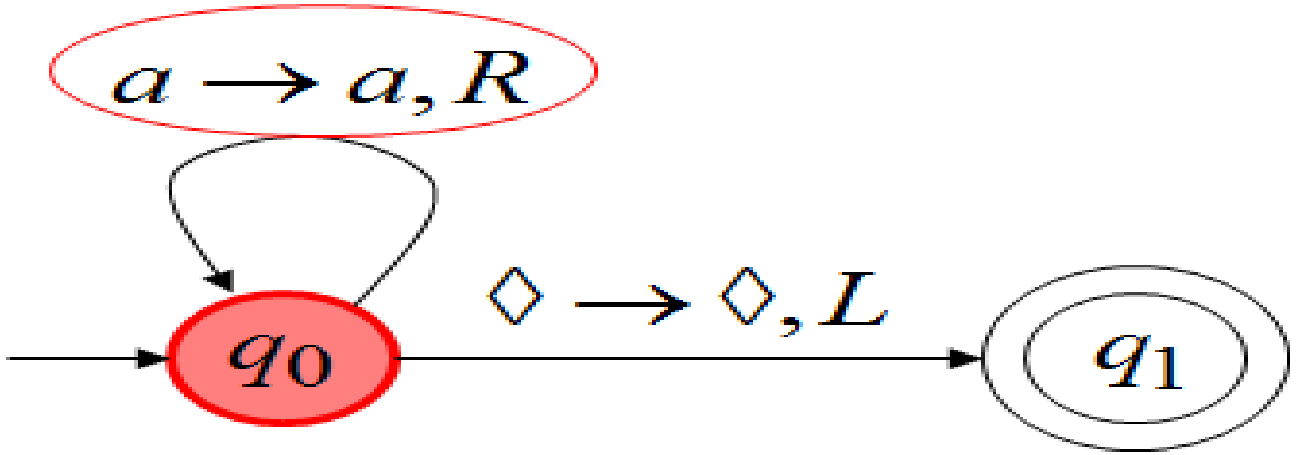
- To use a Turing machine as a transducer, treat the entire nonblank portion of the initial tape as input
- Treat the entire nonblank portion of the tape when the machine halts as output.
- Example:

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

Accepts the language: $a^*$

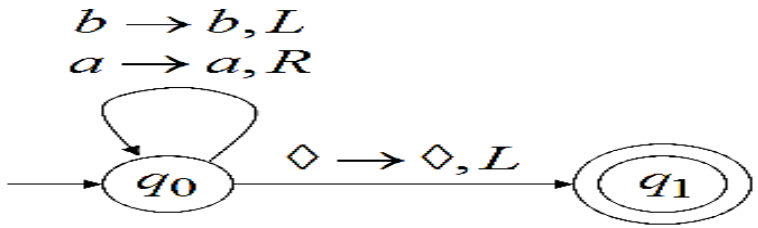



◇
◇
a
a
a
◇
◇
q0

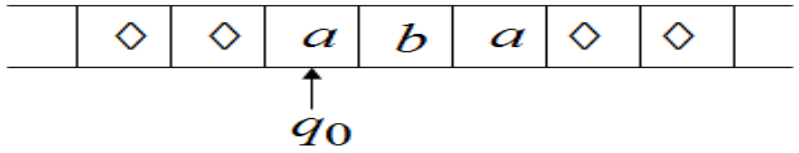

a → a, R
◇ → ◇, L
q0
q1

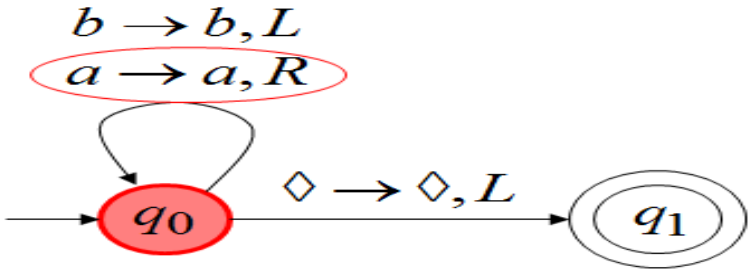
# Infinite Loop Example

A Turing machine
for language $a^* + b(a+b)^*$




Time 0




Because of the infinite loop:

- The accepting state cannot be reached
- The machine never halts
- The input string is rejected

# Another Turing Machine Example

Turing machine for the language $\{a^n b^n\}_{n \geq 1}$

## Basic Idea:

Match a's with b's:

Repeat:

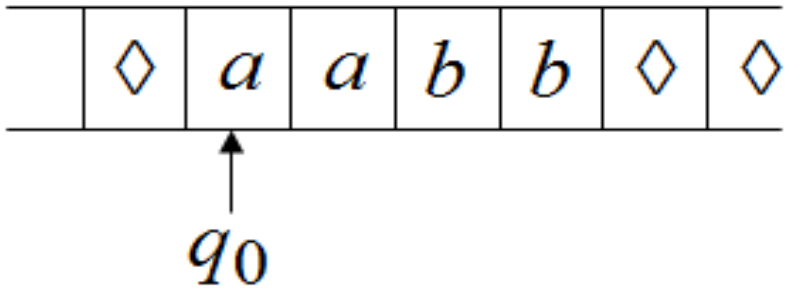
replace leftmost a with x

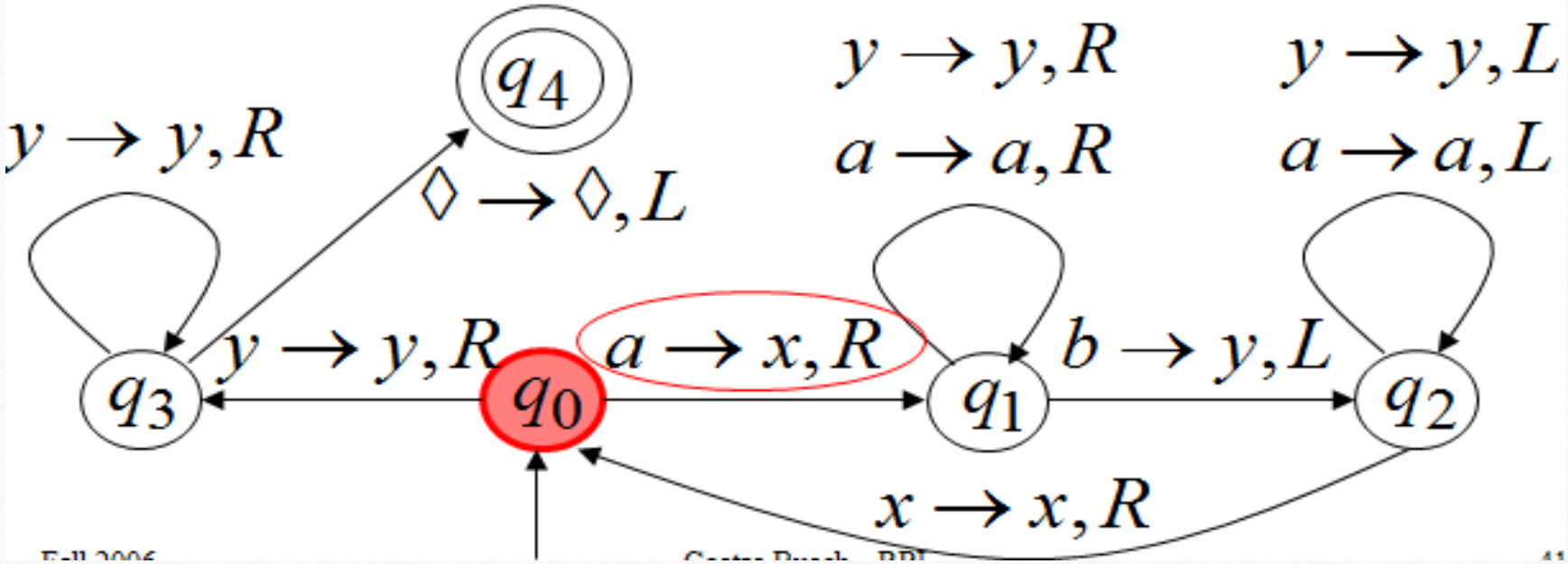
find leftmost b and replace it with y

Until there are no more a's or b's

If there is a remaining a or b reject

Time 0


◊ a a b b ◊ ◊
$q_0$


$q_4$
$y \to y, R$
$\lozenge \to \lozenge, L$
$y \to y, R$
$a \to a, R$
$y \to y, L$
$a \to a, L$
$q_3$
$y \to y, R$
$q_0$
$a \to x, R$
$q_1$
$b \to y, L$
$q_2$
$x \to x, R$

# Exercises

- Design TM for the language $a^n b^n c^n : n>0$
- Construct TM for palindrome strings over alphabet (a, b)
- Construct TM for addition of two positive integers
- Construct TM for addition of two binary number

- **TYPES OF TURING MACHINES**
  - Two way infinite tape
  - Multiple tape Turing Machines
  - Nondeterministic Turing machines
  - Multidimensional Turing machines
  - Multi-head Turing Machines
  - Off-line Turing machines

# MULTITAPE TURING MACHINES

- A Turing Machine with several tapes
- Every Tape's have their Controlled own R/W Head

---

- For N- tape TM $M=(Q,\Sigma, \Gamma,\delta,q0,B,F)$

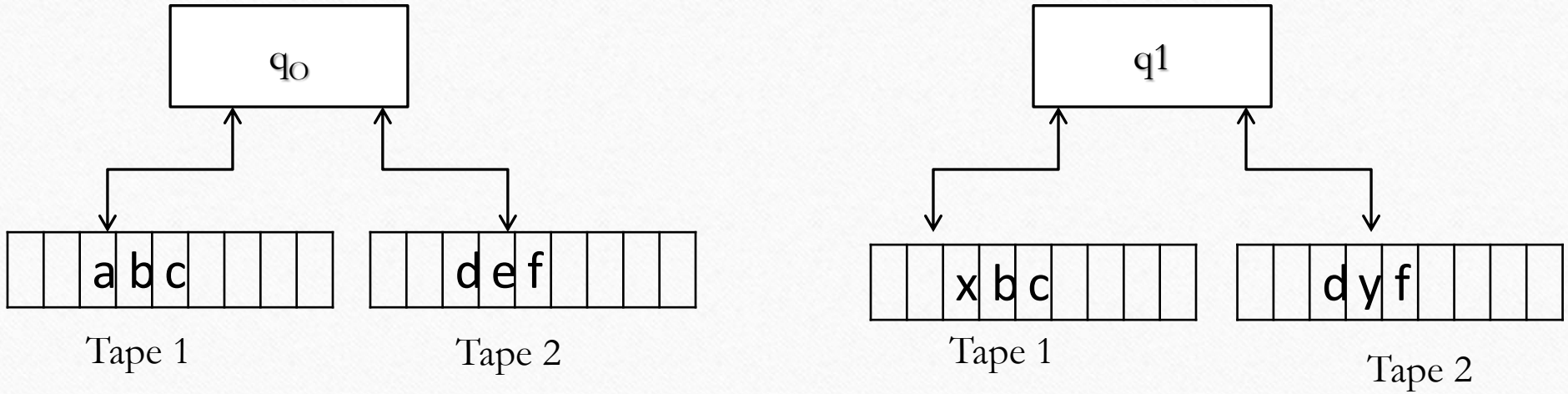
we define $\delta : Q \times \Gamma^N \rightarrow Q \times \Gamma^N \times \{ L , R\}^N$ N is number of tapes

- The multi tape Turing machine is more efficient and easier to program than single-tape Turing machines.
- For every multi tape Turing machine there is a one-tape Turing machine that computes the same partial computable function.

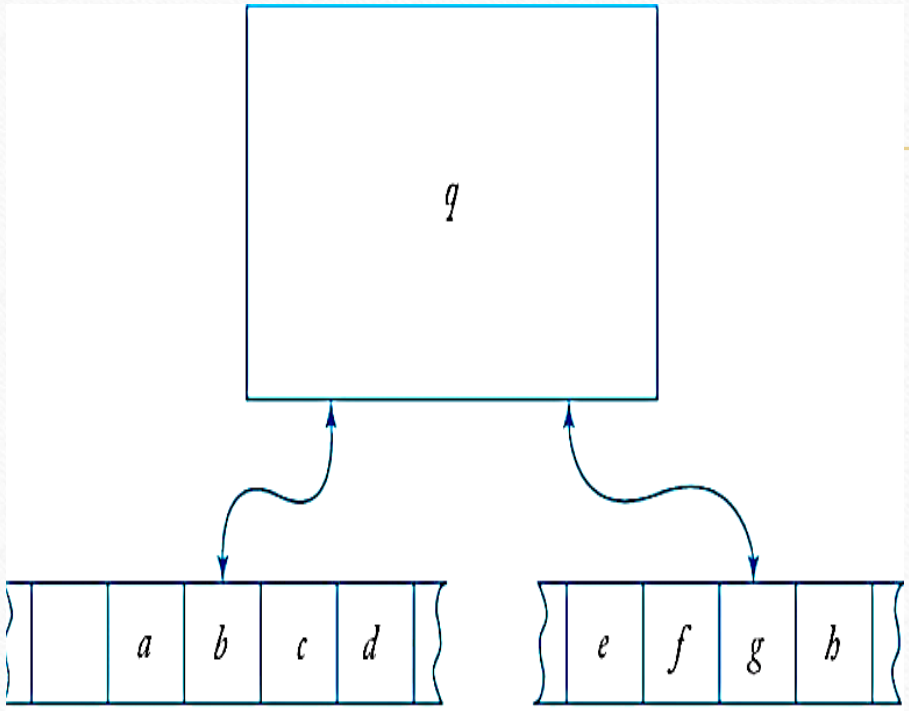
- The expression δ (qi, a1, . . ., aN) = (qj , b1, . . . , bN, L, R, . . . , L) means:
  - if the machine is in state qi and heads 1 through N are reading symbols a1 through aN,
  - the machine goes to state qj, writes symbols b1 through bN,
  - and directs each head to move left or right, or to stay put, as specified.

- For e.g., if N=2 , with the current configuration

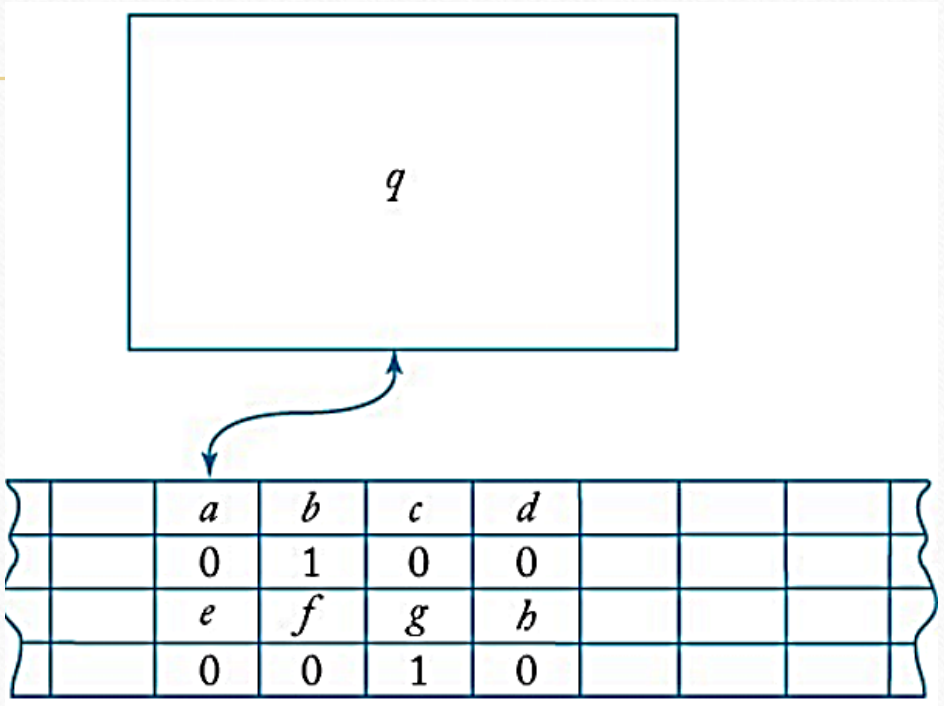
  $\delta(q_O, a, e) = (q_1, x, y, L, R)$



Also we can simulate by standard one-head one-tape Turing machines egg.

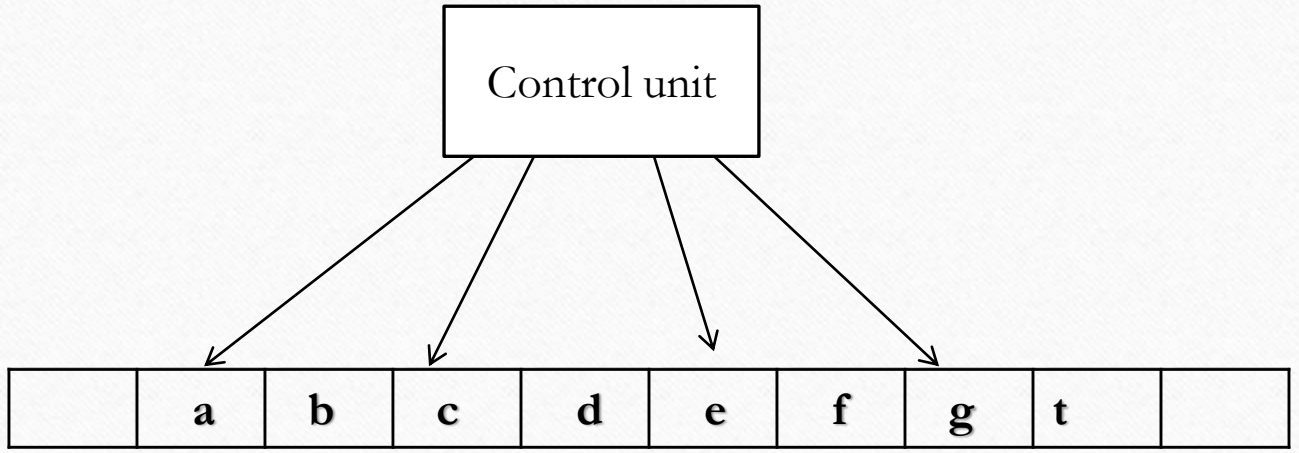



→

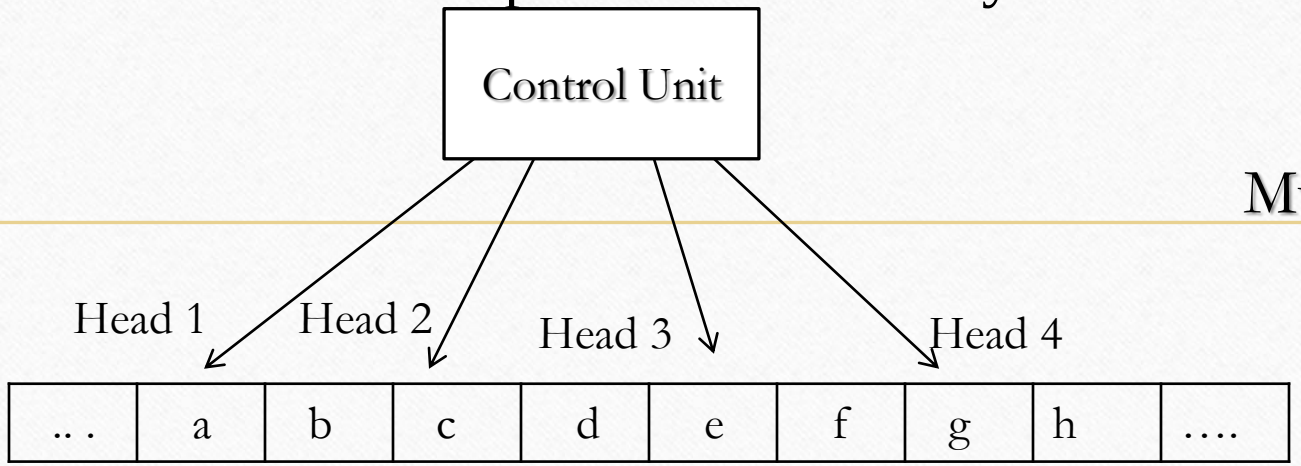


# MULTI-HEAD TURING MACHINE

- Multihead TM has a number of heads instead of one.
- Each head indepently read/ write symbols and move left / right or keep stationery.



- can be simulated by standard one-head one-tape Turing machines. and
- The simulation can be implemented similarly to the case of multi-tape tapes.




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1st track | …. | 1 | B | B | B | B | B | B | B | .. |
| 2nd track | …. | B | B | 1 | B | B | B | B | B | .. |
| 3rd track | .. | B | B | B | B | 1 | B | B | B | .. |
| 4th track | .. | B | B | B | B | B | B | 1 | B | . |
| 5th track | .. | a | b | c | d | e | f | g | h | . |

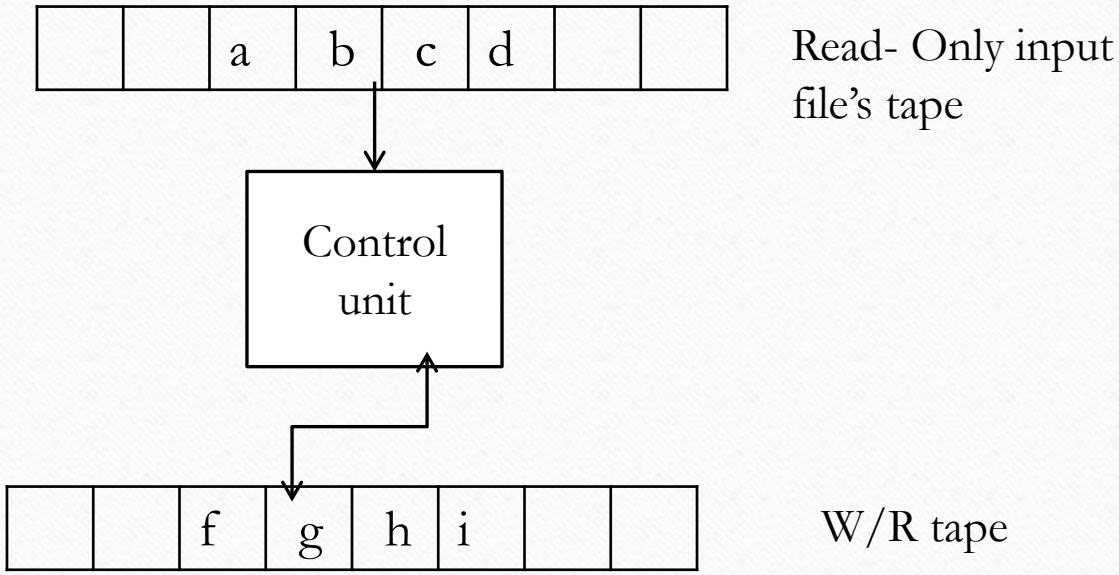
**Multi track TM**

# OFF- LINE TURING MACHINE

- An Offline Turing Machine has two tapes

1. One tape is read-only and contains the input

2. The other is read-write and is initially blank.



# MULTIDIMENSIONAL TURING MACHINE

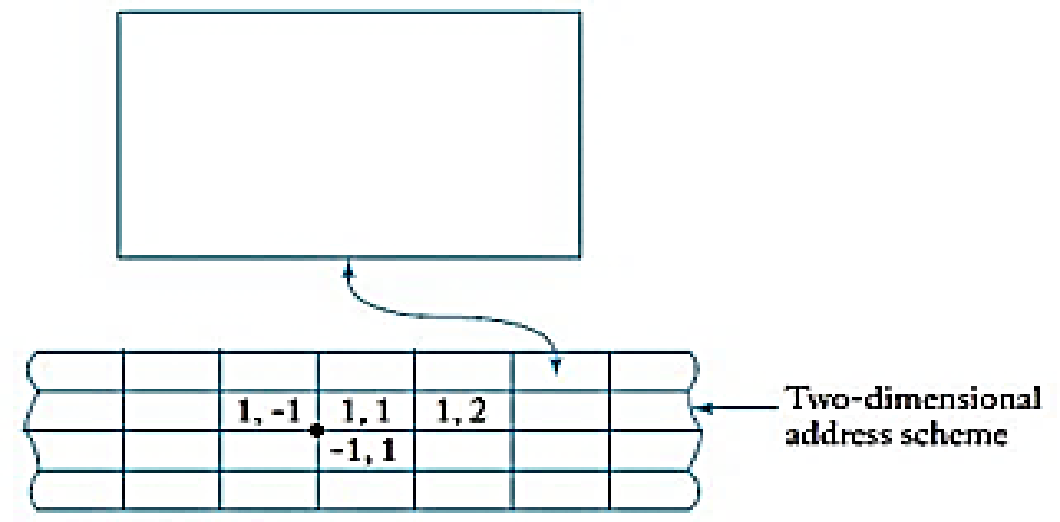
- A Multidimensional TM has a multidimensional tape.

For example, a two-dimensional Turing machine would read and write on an infinite plane divided into squares, like a checkerboard.

---

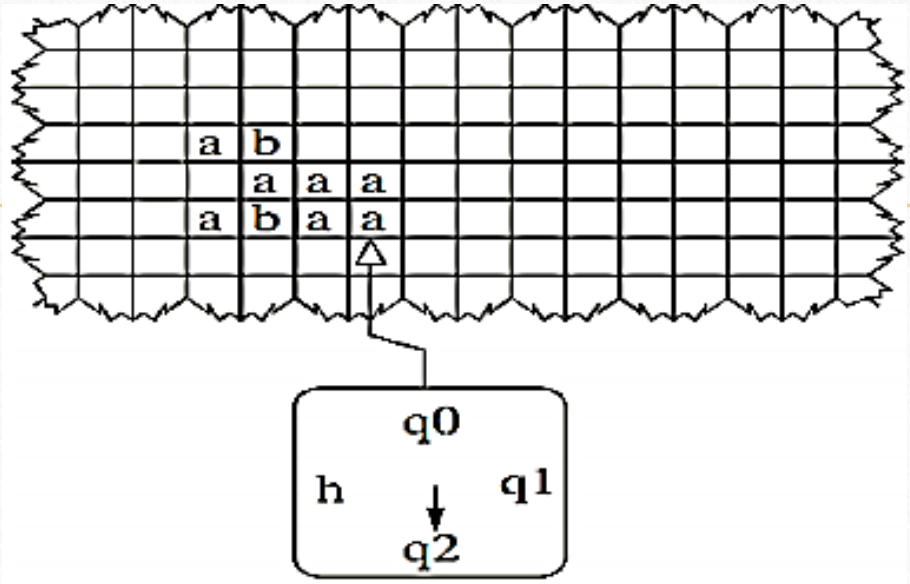
- For a two- Dimensional Turing Machine transaction function define as:

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



# MULTIDIMENSIONAL TURING MACHINE

➢Example-2, a two-dimensional Turing machine using input symbol Σ= {a. b}

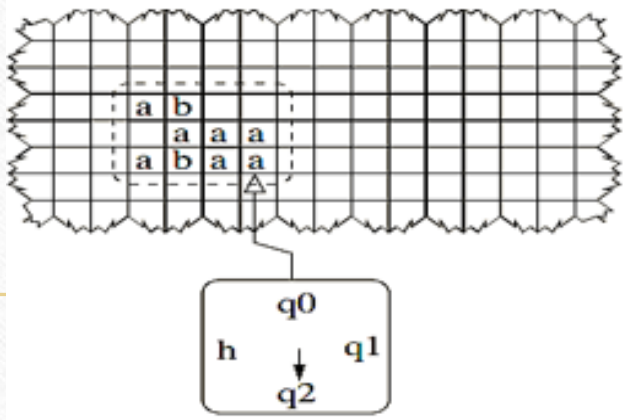


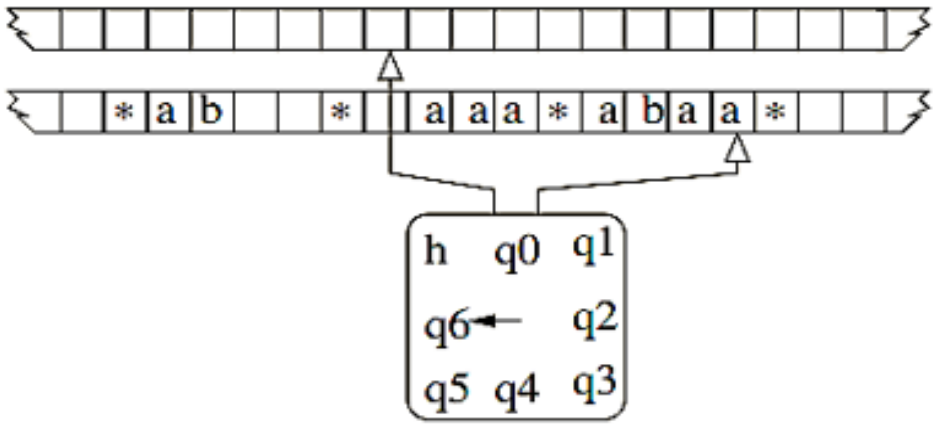

We simulate using a two- tape Turing machine.

We can draw a rectangle around the non-blank cells. Then copy each row of the rectangle onto the second tape, separated by, e.g., *.

# MULTIDIMENSIONAL TURING MACHINE

Con..




Whenever the two- dimensional machine moves its head left or right, the two-tape machine moves tape 2's head left or right.
Whenever the two-dimensional machine moves up (or down), the two-tape machine

- scans left until it finds *;
- as it scans, it writes symbols onto tape 1 to record how many cells it had to scan before it reached the *;

then it scans to the next * to the left (for up) or right (for down);

- then it moves right a certain number of times, as recorded on tape 1.
- If the result of all of this has taken it vertically out of the rectangle, then we add to tape 2 a new line of the same length to the left or right of the current lines.
- If it has moved horizontally out of the rectangle, then we add one cell to the left or right of each line (which will require a lot of left or right shifting).

# Equivalence of TM's and Computers

- What are differences between a Turing machine and computers we use every day?
- In one sense, a real computer has a finite amount of memory, and thus is **weaker** than a TM.
  - But, we can postulate an infinite supply of tapes, disks, or peripheral storage device to simulate an infinite TM tape.
- Computers might store 32 or 64 symbols in one storage location, whereas TM stores only one symbol in each storage location,
- The computer has "random access concept," (egg. contents of memory location 1000 can be accessed by executing one move only).
- Also Turing machine can easily simulate but must make 900 moves in order to move its head 900 cells to the right.
  - i.e. the difference is in efficiency rather than in fundamental computing power.

# Equivalence of TM's and Computers

- Additionally, we can assume there is a human operator to mount disks, keep them stacked neatly on the sides of the computer, etc.
- The model of Turing machines is not meant to provide an accurate (or "tight") model of real-life computers, but rather to capture their inherent limitations and abilities (i.e., a computational task can be solved by a real-life computer if and only if it can be solved by a Turing machine).
- In comparison to real-life computers, the model of Turing machines is extremely oversimplified and abstracts away many issues that are of great concern to computer practice.
- However, these issues are irrelevant to the higher-level questions addressed by Complexity Theory.

# Equivalence of TM's and Computers

- Anything a computer can do, a TM can do, and vice versa
- TM is much slower than the computer, though
- But the difference in speed is polynomial
- Each step done on the computer can be completed in $O(n^2)$ steps on the TM (see book for details of proof).
- While slow, this is key information if we wish to make an analogy to modern computers. Anything that we can prove using Turing machines translates to modern computers with a polynomial time transformation.
- Whenever we talk about defining algorithms to solve problems, we can equally talk about how to construct a TM to solve the problem.
- If a TM cannot be built to solve a particular problem, then it means our modern computer cannot solve the problem either.

# Chapter two

## Decidability

- Turing Decidable
- Turing Acceptable
- Undecidable Problems

# Decidability overview

- Our concern here will be the somewhat simplified setting where the result of a computation is a simple “yes” or “no”. In this case, we talk about a problem being decidable or undecidable.
  - Every question about regular language is decidable
  - AREX is a decidable language
- A decision problem consists of a set of questions whose answers are either yes or no.
- In other words,  it is a general question to be answered, usually possessing several parameters, or free variables, whose values are left unspecified.
- An *instance* of a problem is obtained by specifying particular values for all of the problem parameters.
- Example: *The Hamiltonian Circuit problem.*
  **instance** A *graph G = (V, E)*
  **question** *Does G contain a Hamiltonian circuit?*

# Decidability overview

- Definition: Let $\Sigma$ be an alphabet and let $A \subseteq \Sigma^*$ be a language.
- We say that A is decidable, if there exists a Turing machine M, such that for every string $w \in \Sigma^*$, the following holds:
- 1. If $w \in A$, then the computation of the Turing machine M, on the input string w, terminates in the accept state.
- 2. If $w \notin A$, then the computation of the Turing machine M, on the input string w, terminates in the reject state.
- In other words, the language A is decidable, if there exists an algorithm that
  - (i) terminates on every input string w, and
  - (ii) correctly tells us whether $w \in A$ or $w \notin A$.
- A language A that is not decidable is called undecidable. For such a language, there **does not exist an algorithm** that satisfies rules (i) and (ii) above.

# Decidability overview

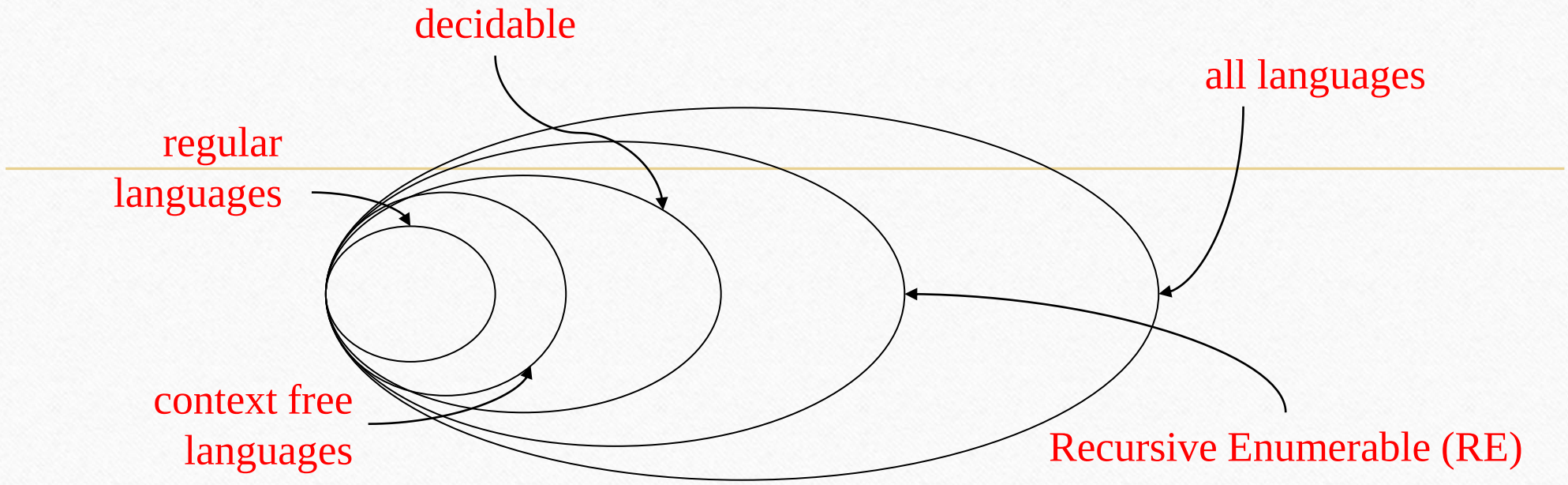
- A solution to a decision problem is an algorithm that answers the question that results from each instance.
- A decision problem is decidable if a solution exists and is undecidable otherwise.
- In other setting, a decision problem is undecidable if there is no algorithm that solves the problem otherwise it is decidable.
- We will be concerned with the question of whether certain decision problems are decidable.
- According to Church's thesis, every computational device can be simulated by some Turing machine that solves the problem.
- To show that a decision problem is undecidable, we take the point of view that it serves to show that no Turing machine solves the problem.

- Turing decidability
- The language L is *Turing decidable* (or just decidable) if there exists a Turing machine M that accepts all strings in L and rejects all strings not in L. i.e.
  - (i.e. A TM *decides* a language L iff it accepts all strings in L and rejects all strings not in L)
- Note that by rejection we mean that the machine halts after a finite number of steps and announces that the input string is not acceptable.
- Acceptance, as usual, also requires a decision after a finite number of steps.
- A language L is called decidable or recursive iff some TM decides L.

# Decidability

- **Turing Recognizability**
- A language L is *Turing recognizable* if there is a Turing machine M that recognizes L, that is, M should accept all strings in L and M should not accept any strings not in L. (i.e. **A TM *recognizes* a language iff it accepts all and only those strings in the language**)
- This is not the same as decidability because recognizability does not require that M actually reject strings not in L.
- M may reject some strings not in L but it is also possible that M will simply "remain undecided" on some strings not in L; for such strings, **M's computation never halts**.
- **A language L is called Turing-recognizable or recursively enumerable iff some TM recognizes L.**

# Decidability




decidable $\subset$ RE $\subset$ all languages

# Decidability overview

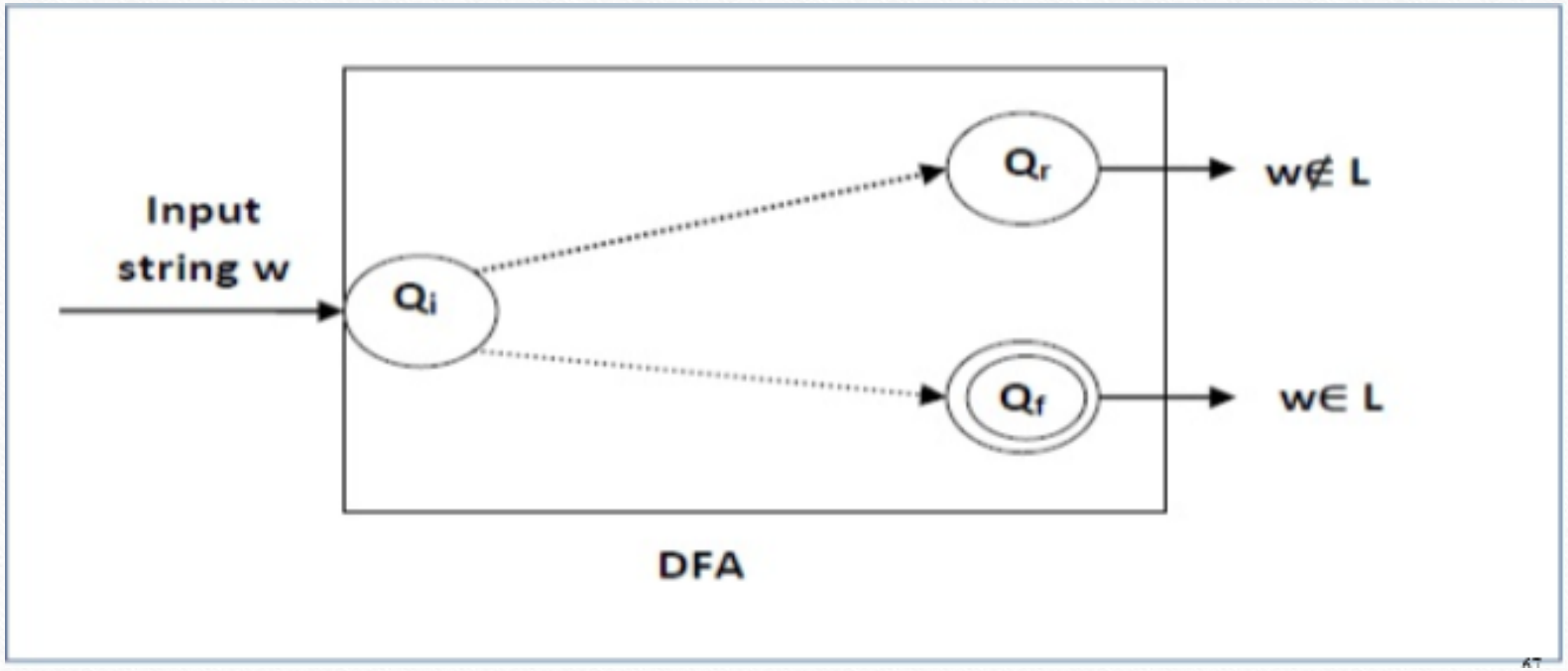
- **Example-1:** Find out whether the following problem is decidable or not.
  Is a number ‘m’ prime?
- Solution

---

- Prime numbers = {2, 3, 5, 7, 11, 13, ……..}
- Divide the number ‘m’ by all the numbers between ‘2’ and ‘√m’ starting from ‘2’.
- If any of these numbers produce a remainder zero, then it goes to the “Rejected state”, otherwise it goes to the “Accepted state”.
- So, here the answer could be made by ‘Yes’ or ‘No’.
- Hence, it is a decidable problem.

# Decidability overview

- Example 2
  Given a regular language L and string w, how can we check if w ∈ L?
  Solution
  Take the DFA that accepts L and check if w is accepted



# Other examples of decidable problems:

- **Equivalence of two regular languages:** Given two regular languages, there is an algorithm and Turing machine to decide whether two regular languages are equal or not.
- **Finiteness of regular language:** Given a regular language, there is an algorithm and Turing machine to decide whether regular language is finite or not.
- **Emptiness of context free language:** Given a context free language, there is an algorithm whether CFL is empty or not.

# Computational problems

A computational problem can be viewed as an infinite collection of instances together with a solution for every instance. The input string for a computational problem is referred to as a problem instance, and should not be confused with the problem itself.

1. Decision problems: answered by "yes" or "no“, “True” or “False”
2. Search problems
   - In a search problem we not only want to know if a solution exists, but find the actual solution as well.
3. Optimization problems
   - Optimization problems ask for the best possible solution to a problem
   - A decision or search problem can have several optimization variants
4. Counting problems: ask the number of solutions of a given instance

# Computational problems

- **Computable functions**
  - Functions whose output values can be determined algorithmically from their input values
- **Non-computable functions**
  - Functions that are so complex that there is no well-defined, step-by-step process for determining their output based on their input values
  - The computation of these functions lies beyond the abilities of any algorithmic system
  - Examples: can you get integer values for x, y and z to solve the problem?
    - $x^2y + 5y^3 = 3$
    - $x^2 + z^5 – 3y^2 = 0$
    - $y^2 – 4z^6 = 0$

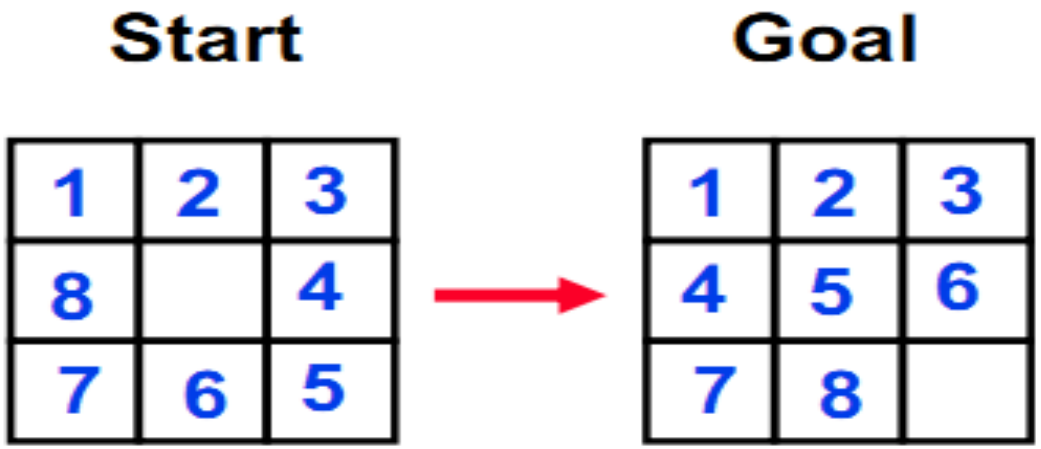
# Decision problems example: Consider problems with answer YES or NO

- Does Machine $M$ have three states ?
- Is string $w$ a binary number?
- Does DFA $M$ accept any input?

A problem is decidable if some Turing machine decides (solves) the problem

# Search problems

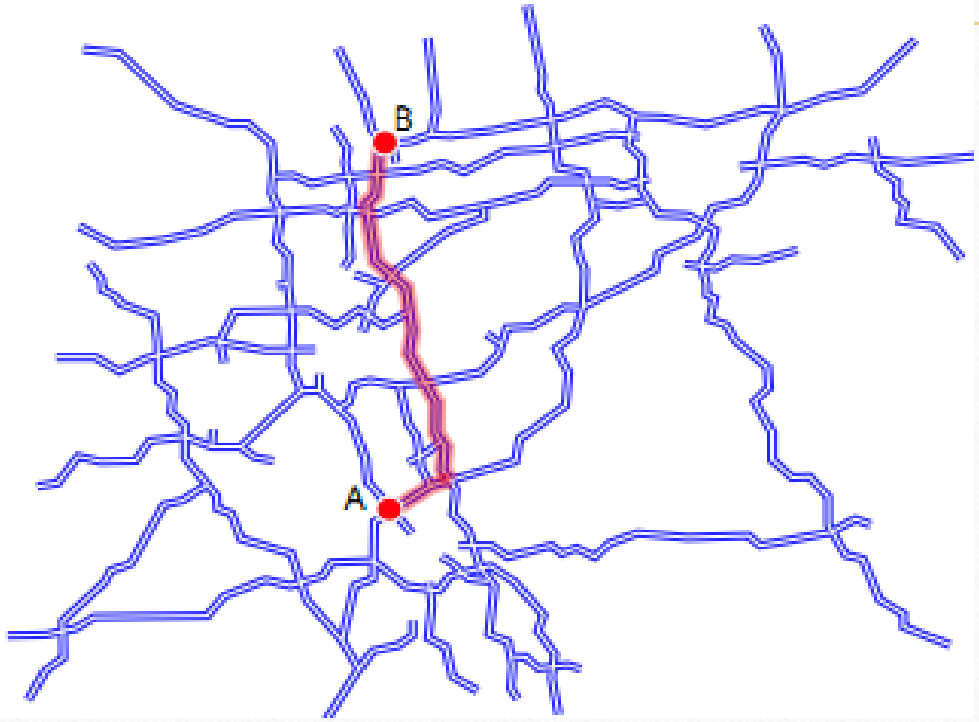
## Example: The 8-puzzle



# Examples for Search Problems

Road Map Problem

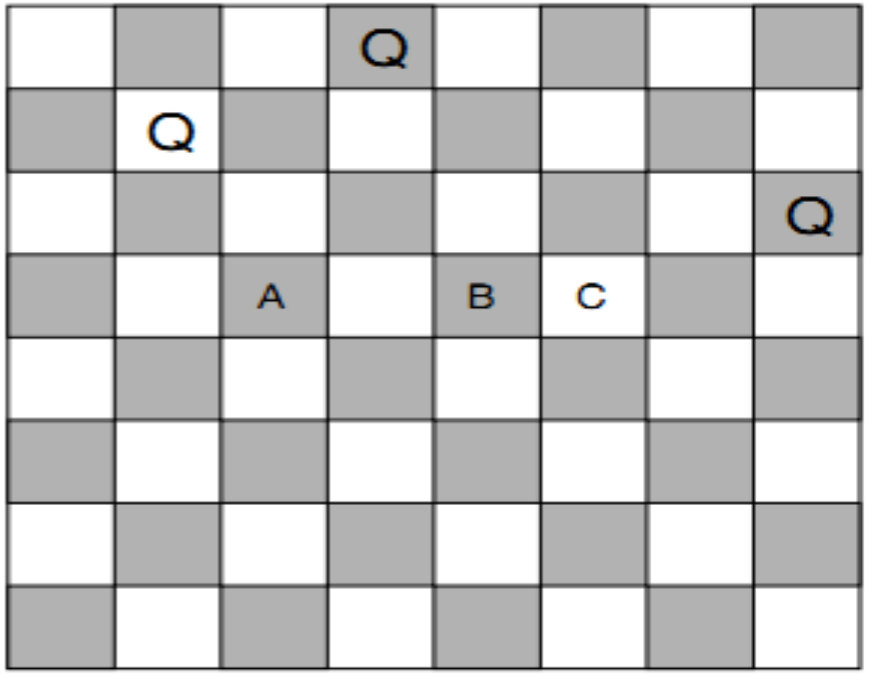
Search for a route from place $A$ to $B$:



# Examples for Search Problems

## 8-Queens Problem

Incremental placement of queens:



# Examples for Search Problems

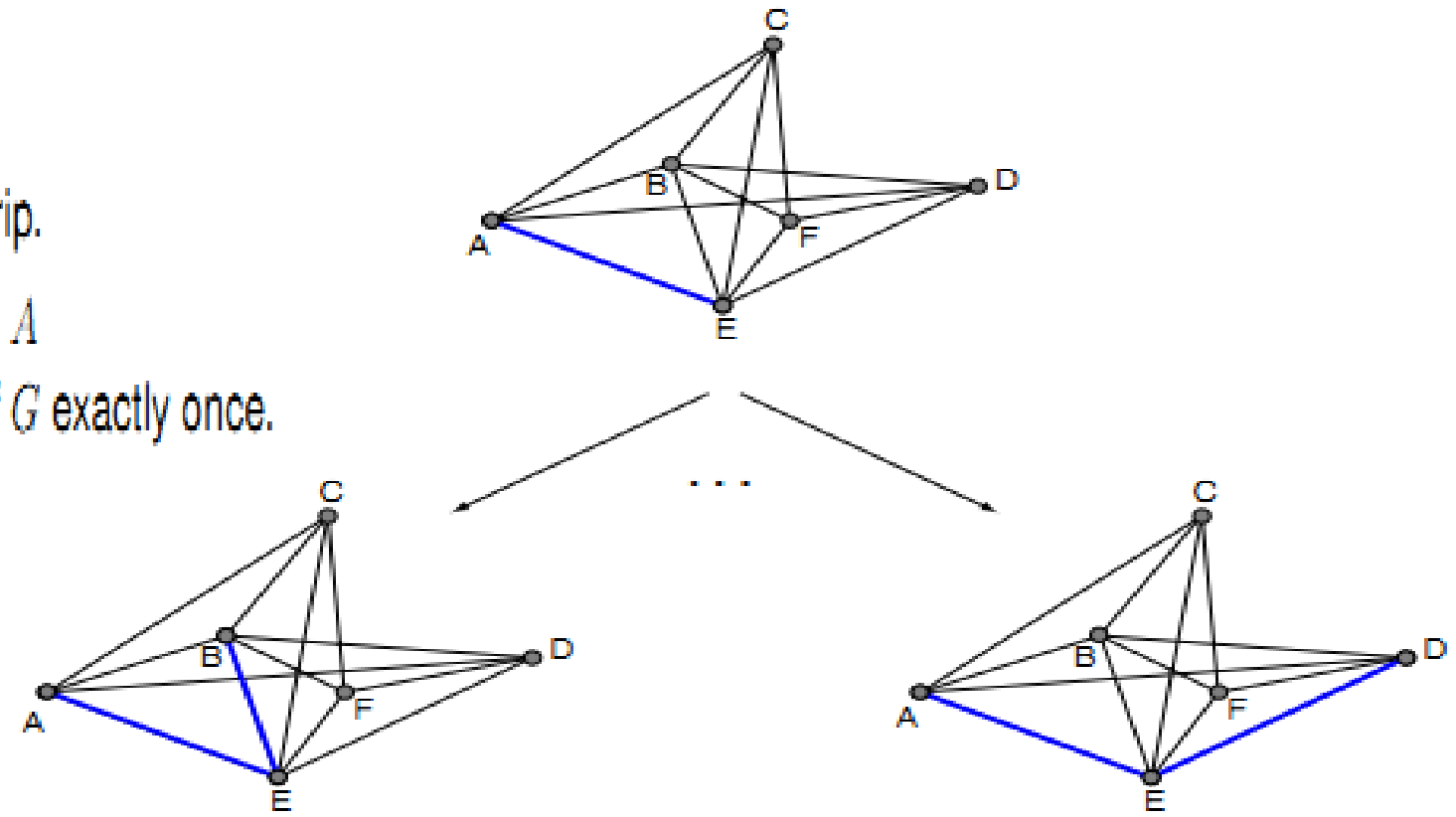
## Traveling Salesman Problem (TSP)

Problem: Traveling Salesman Problem

Instance: $G$. A weighted finite graph.

$A$. Start node for the round-trip.

Solution: A path starting and ending in $A$ that visits each other node of $G$ exactly once.



Here are some examples of decision problems involving Turing machines. Is it decidable whether a given Turing machine

(a) has at least 481 states?

(b) takes more than 481 steps on input $\epsilon$?

(c) takes more than 481 steps on *some* input?

(d) takes more than 481 steps on *all* inputs?

(e) ever moves its head more than 481 tape cells away from the left endmarker on input $\epsilon$?

# Solutions

- problem (a) is easily decidable, since the number of states of M can be read off from the encoding of M.
- we can build a TM that, given the encoding of M written on its input tape, counts the number of states of M and accepts or rejects depending on whether the number is at least 481.
- problem (b) is decidable, since we can simulate M on input Ɛ with a universal machine for 481 steps (counting up to 481 on a separate track) and accept or reject depending on whether M has halted by that time.
- problem (c) is decidable: we can just simulate M on all inputs of length at most 481 for 481 steps.
- If M takes more than 481 steps on some input, then it will take more than 481 steps on some input of length at most 481, since in 481 steps it can read at most the first 481 symbols of the input.

# Solutions

- The argument for problem (d) is similar.
  - If M takes more than 481 steps on all inputs of length at most 481, then it will take more than 481 steps on all inputs.
- For problem (e), if M never moves more than 481 tape cells away from the left end marker, then it will either halt or loop in such a way that we can detect the looping after a finite time.
- This is because if M has k states and m tape symbols, and never moves more than 481 tape cells away from the left end marker, then there are only 482 km $^{481}$ configurations it could

# Introduction - Undecidability

- A **decision problem (DP)** is a problem that requires a yes or no answer.
  - i.e DP = {(x, yes) : $x \in L$} U {(x, no) : $x \notin L$).
- A decision problem may or may not be decidable.
- A problem that cannot be decided by algorithmic means even after giving an unlimited resource and infinite amount of time, is termed as undecidable.
- Means that, a decision problem that admits *no algorithmic solution* is said to be undecidable. (if a decision problem proves to be noncomputable then it is said to be undecidable)
- No undecidable problem can ever be solved by a computer or computer program of any kind. In particular, there is no Turing machine to solve an undecidable problem.
- We have not said that undecidable means we don't know of a solution today but might find one tomorrow. It means we can never find an algorithm for the problem.

# Introduction - Undecidability

- In computability theory and computational complexity theory, an undecidable problem is a decision problem for which it is impossible to construct a single algorithm that always leads to a correct yes-or-no answer.

---

- In computability theory, the halting problem is an undecidable problem which can be stated as follows:
  - Given a description of a program and a finite input, decide whether the program finishes running or will run forever.
- Alan Turing proved in 1936 that a general algorithm running on a Turing machine that solves the halting problem for all possible program-input pairs necessarily cannot exist. Hence, the halting problem is undecidable for Turing machines.

# Halting problem

- Halting means that the program on certain input will accept it and halt or reject it and halt and it would never go into an infinite loop. (i.e. halting means terminating)
- So can we have an algorithm that will tell that the given program will halt or not?
- In terms of Turing machine, will it terminate when run on some machine with some particular given input string.
- The answer is no we cannot design a generalized algorithm which can properly say that “a given program will ever halt or not”
- This is an undecidable problem because we cannot have an algorithm which will tell us whether a given program will halt or not.
- The only way is to run the program and check whether it halts or not.

# Halting problem

**Proof by Contradiction**

**Problem statement:** Can we design a machine which if a given program can find out if that program will always halt or not halt on a particular input?

**Solution**: Assume that we can design that a machine called HM(P, I)

- where HM =machine, P =program and I =input. On taking input, both arguments the machine HM will tell that the program P either halts or not.

- We write another program call CM(X) where X =any program(taken as argument)
- Egg.

```
HM(P, I)
{.... Halt;
or may not Halt }
```

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

- The function HM(X) in CM(X) and we pass the arguments (X, X) to HM(), according to the definition of HM() it can take two arguments (i.e. one is program and another is input)
- Now in the second program we pass X as a program and X as input to the function HM().

# Halting Problem- Example

- Let: int x=1;

```
do {
    if (x%2==0) {
    x = x/3;
    } else {
    x=3x+1;
    }
    x++;
    }
while (x>=1); //(terminates when ??)
```

# Turing Machine Halting Problem

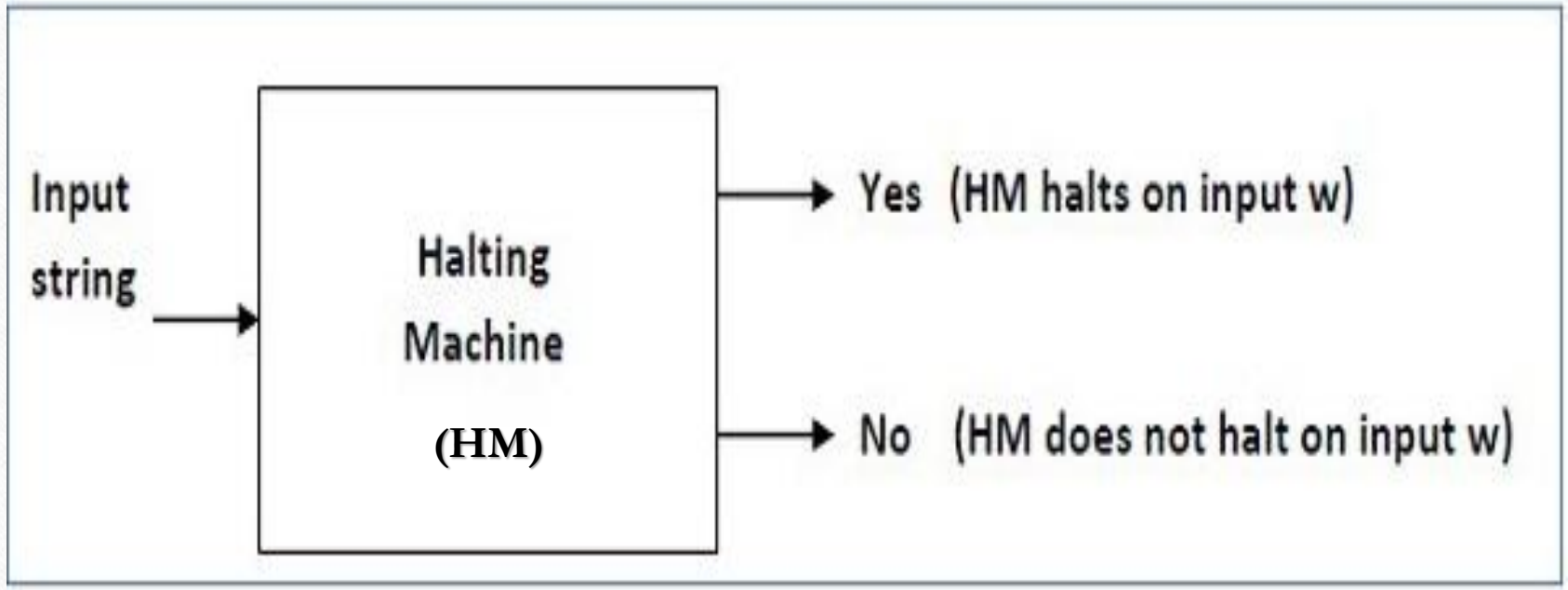
- The most famous of the undecidable problems is concerned with the properties of Turing machines themselves.
- **Assume that:**

---

- **Input** – A Turing machine M and an input string w.
- **Problem** – Does the Turing machine finish computing of a string w in a finite number of steps? The answer must be either yes or no.
- **Proof** – At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself.
- We will call this Turing machine as a Halting machine that produces a 'yes' or 'no' in a finite amount of time.

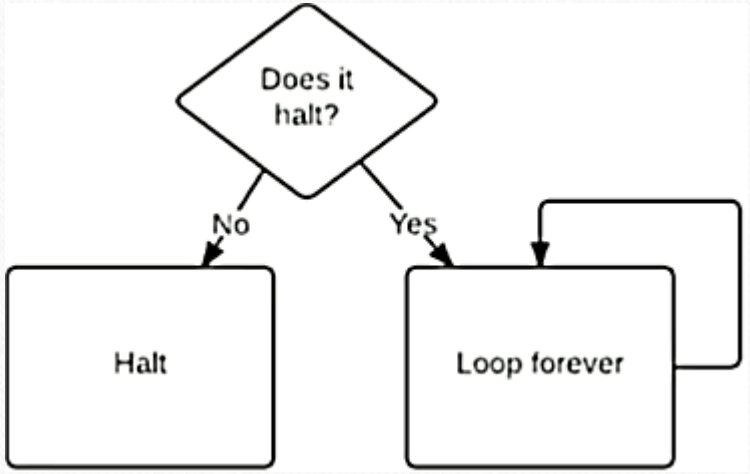
# Turing Machine Halting Problem

- If the halting machine HM finishes in a finite amount of time, the output comes as ‘yes’, otherwise as ‘no’.
- The following is the block diagram of a Halting machine

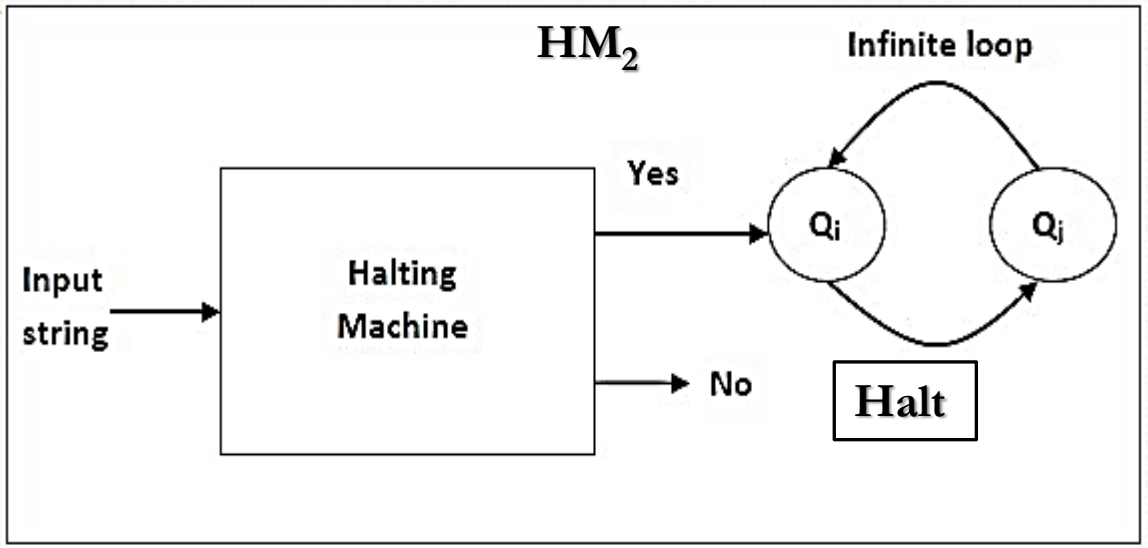


# Turing Machine Halting Problem

- Now we will design an inverted halting machine $HM_2$ as:
  - If HM returns YES, then loop forever. If HM returns NO, then halt.
- The following is the block diagram of an 'Inverted halting machine' $HM_2$




**i.e.**




- Here, we have got a contradiction. Hence, the halting problem is undecidable.

# Other examples of undecidability problems

- **Ambiguity of context-free languages:** Given a context-free language, there is no Turing machine which will always halt in finite amount of time and give answer whether language is ambiguous or not.

- **Equivalence of two context-free languages:** Given two context-free languages, there is no Turing machine which will always halt in finite amount of time and give answer whether two context free languages are equal or not.

- **Everything or completeness of CFG:** Given a CFG and input alphabet, whether CFG will generate all possible strings of input alphabet ($\sum^*$) is undecidable.

# Semidecidability

- A computable function is one that always halts and gives an answer either yes or no.
- Definition: A function *f* with Dom(f) $\subseteq \Sigma^*$ is called **partial Turing computable** if there exists a Turing machine M that computes it. We called it partial, because, the Turing machine may not halt on some inputs w [those w that are not in Dom(f)]. Or
- A **partially computable function** halts and gives a 'yes' on those inputs for which 'yes' is the correct solution, but never halts on other inputs.
- Example: the halting problem.
- To determine Halts(P, D), simply call P(D); then, Halts(P, D) halts and outputs yes if P(D) halts, and loops otherwise.

# Semidecidability

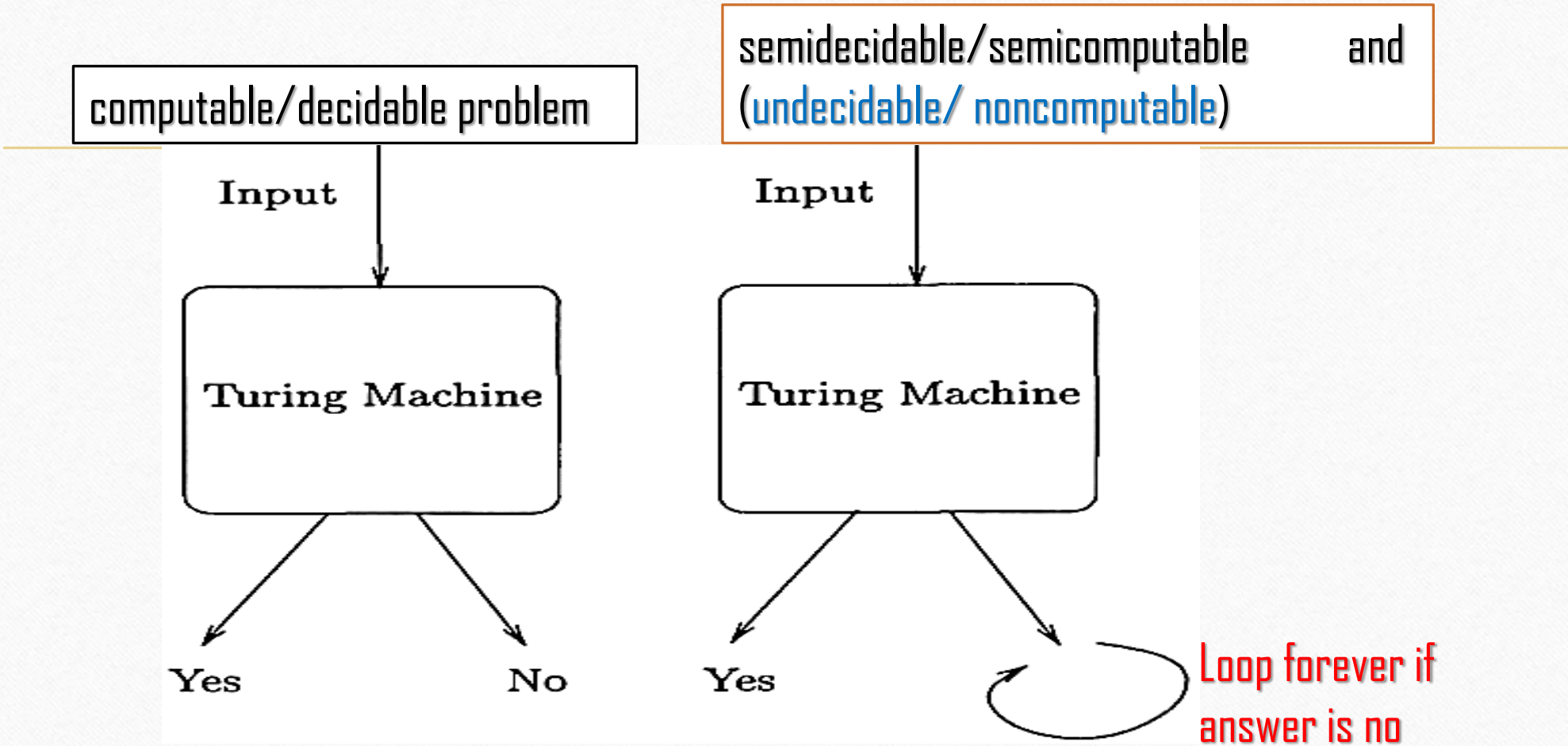
- **Less decidable**: These are cases for which it's not possible to check the validity of either a yes answer or a no answer in **a finite amount of time**.
- Egg. Totality problem, which instead of asking if a program R halts on a given input X asks if it halts on all inputs X. i.e.
- A function (or program) F is said to be **total** if F(x) is defined for all x (or similarly, if F(x) halts for all x).
- There are problems that share with the totality problem the property of not having any sort of answers (yes or no) but are still harder: these can generically be described as highly undecidable problems.
- Example: recurring unbounded tiling problem, where a specified tile from the set is required to occur infinitely often.
- This problem certainly lacks an answer in either direction.

# Semidecidability

- A **language** L is called **semidecidable** if there exists a Turing machine M that outputs 1 on any w $\in$ L and does not halt on any w $\notin$ L [in other words, for the f computed by M, Dom(f) = L]. It means,
- A Turing machine semidecides a language if it recognizes every string in the language and does not halt on the strings that are not in the language. Hence, we can define it with slightly different approach as:
- A **language** L is **semidecidable** if there exists a Turing machine M that halts on every string w$\in$L and does not halt on any string w $\notin$ L.

# Semidecidability

- The difference between computable/decidable and partially computable/decidable



# Reducibility

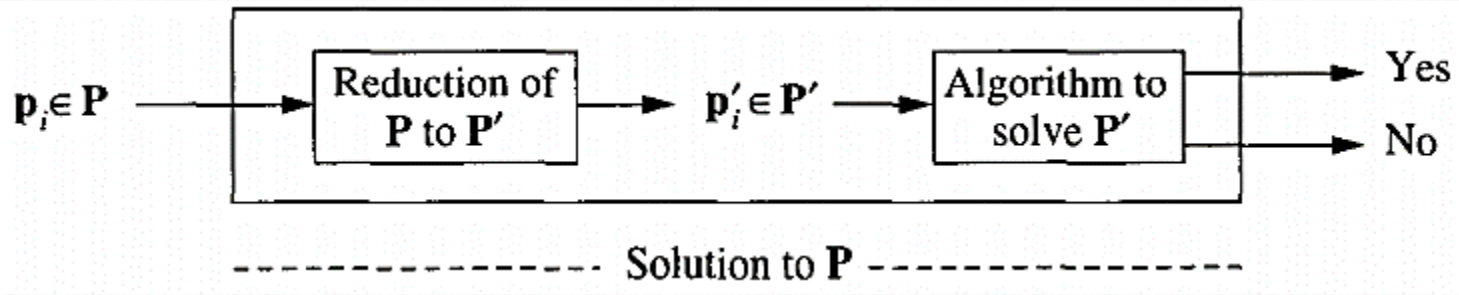
- Showing that a problem is decidable, we can construct a Turing machine there is a decider to decide that language.
- *But, to show that a problem is undecidable*, we should show that there is no Turing machine which can decide a corresponding language.
- Therefore, given a problem to show that it is undecidable it may not be very easy to show that the problem is undecidable.
- Now, there is another way to show that a given problem has to be decidable or undecidable is called Reducibility
  - Reducibility plays an important role in classifying problems by decidability or/and undecidability.

# Reducibility

- A **reduction** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.
- Reducibility always involves two problems ( call them A and B),
- for example, suppose that you want to find your way around a new city (i.e. A).
- You know that doing so would be easy if you had a map (i.e. B).
- Thus you can reduce the problem of finding your way around the city to the problem of obtaining a map of the city.
- If *A* reduces to *B*, we can use a solution to *B* to solve *A*.
- Note that reducibility says nothing about solving A or B alone, but only about the solvability of A in the presence of a solution to B.

# Reducibility

- In other explanation,
- Assume that a decision problem P is many-to-one reducible to a problem **P',** if there is a Turing machine that takes any problem $\mathbf{p_i} \in P$ as input and produces an associated problem $\mathbf{P'_i} \in P'$, where the answer to the original problem $\mathbf{p_i}$ can be obtained from the answer to **p'.**
- As the name implies, the mapping from P to **P'** need not be one-to-one: Multiple problems in **P** may be mapped to the same problem in **P'.**
- If a decision problem **P'** is decidable and **P** is reducible to **P'**, then **P** is also decidable.
- A solution to **P** can be obtained by combining the reduction with the algorithm that solves **P'**

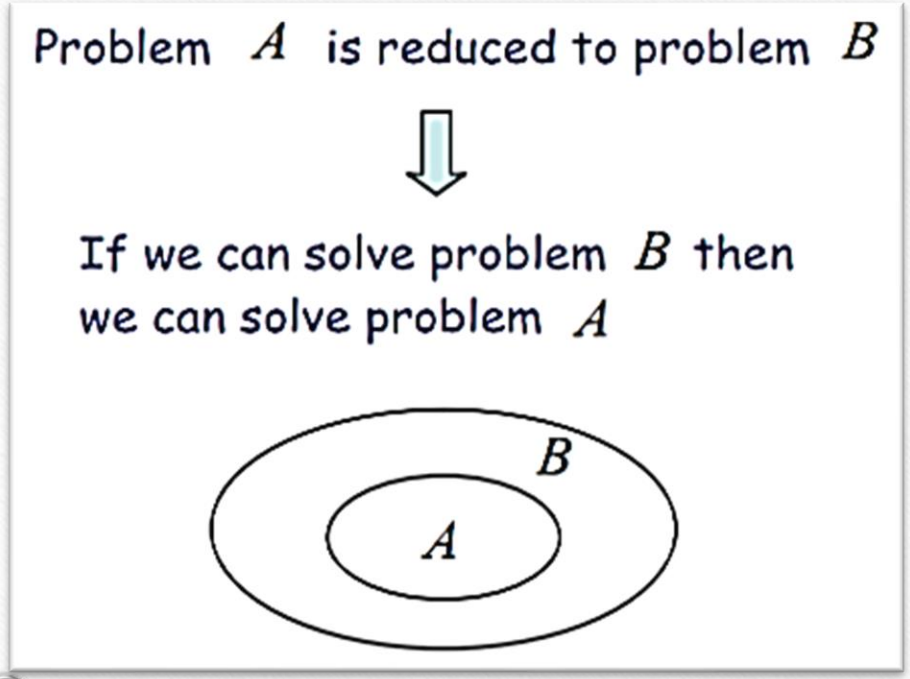


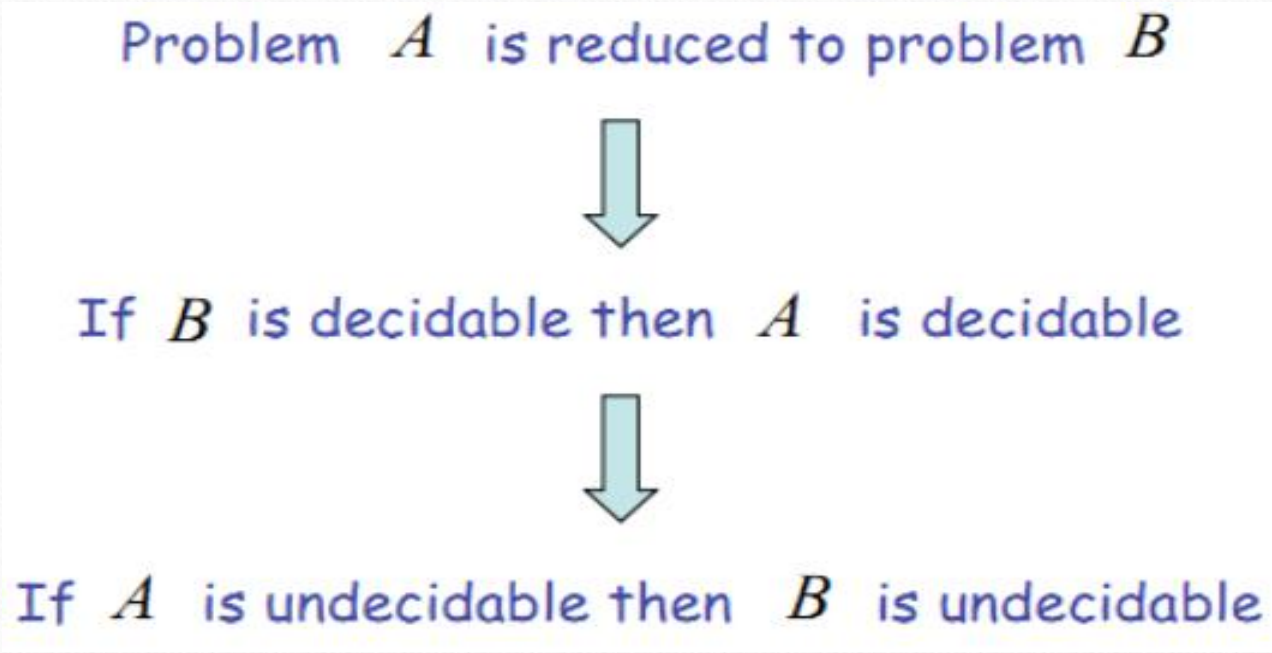
# Reducibility

- We say that a problem **A** is reduced to a problem **B** if the decidability of **A** follows from the decidability of **B**.
- Then, if we know that **A** is undecidable, we can conclude that **B** is also undecidable.

---

- i.e.



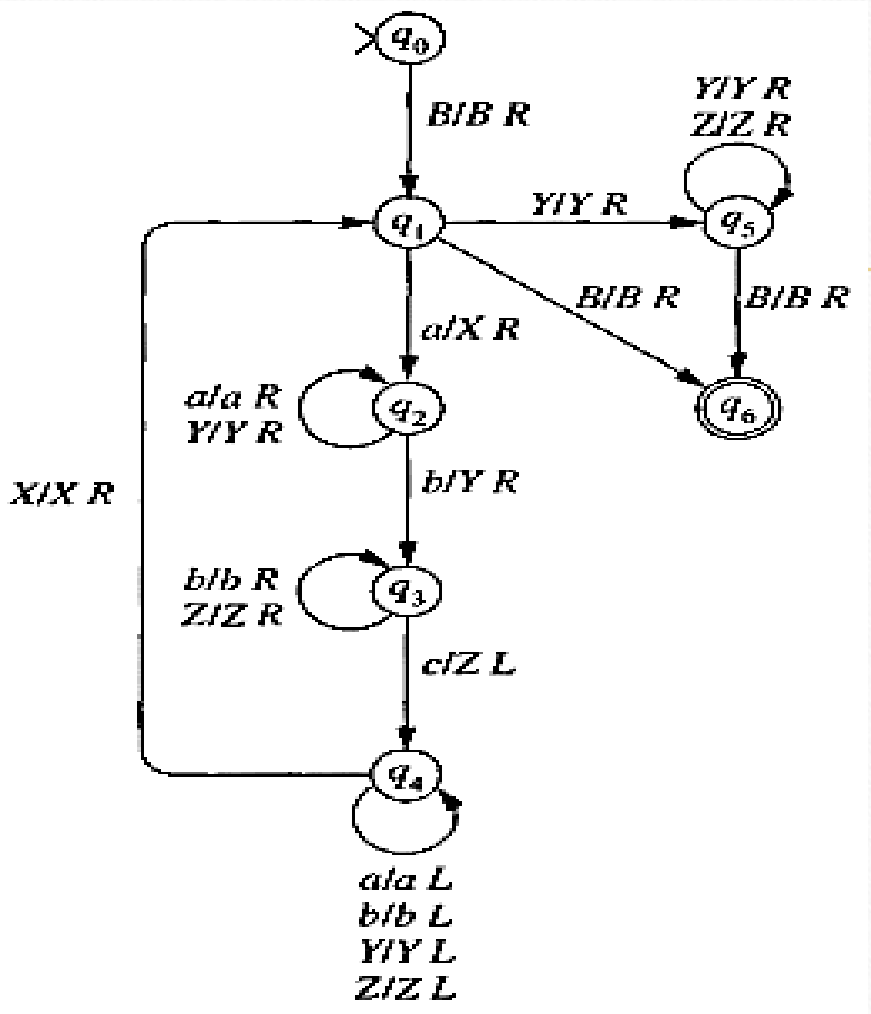
# Reducibility example

- Consider the problem P of accepting strings in the language L = {uu | u = $a^i b^i c^i$ for some i ≥ 0}.
- We will reduce the problem P to that of recognizing a single instance of $a^i b^i c^i$
- The original problem can then be solved using the reduction and the machine M.
- The reduction is obtained as follows:
  1. The input string w is copied.
  2. The copy of *w* is used to determine whether w = *uu* for some string u $\in \{a, b, c\}^*$
  3. If w ≠*uu*, then the tape is erased and a single a is written in the input position.
  4. If w =*uu*, then the tape is erased, leaving u in the input position.
- If the input string w has the form *uu*, then $w \in$ L if, and only if, $u = a^i b^i c^i$ for some *i*.
- The machine M has been designed to answer precisely this question.
- On the other hand, if $w \neq uu$, the reduction produces the string a. This string is subsequently rejected by M, indicating that the input $w \in L$.

# Reducibility example con…

The machine M accepts L= {$a^i b^i c^i \mid i \geq 0$}.

- NB:
- Reduction has important implications for undecidability as well as decidability. That is:
- if A is reducible to B and B is decidable, A also is decidable.
- Equivalently, if A is undecidable and reducible to B, B is undecidable.
- This last version is key to proving that various problems are undecidable.



# Example-2: **state-entry problem/dead code problem**

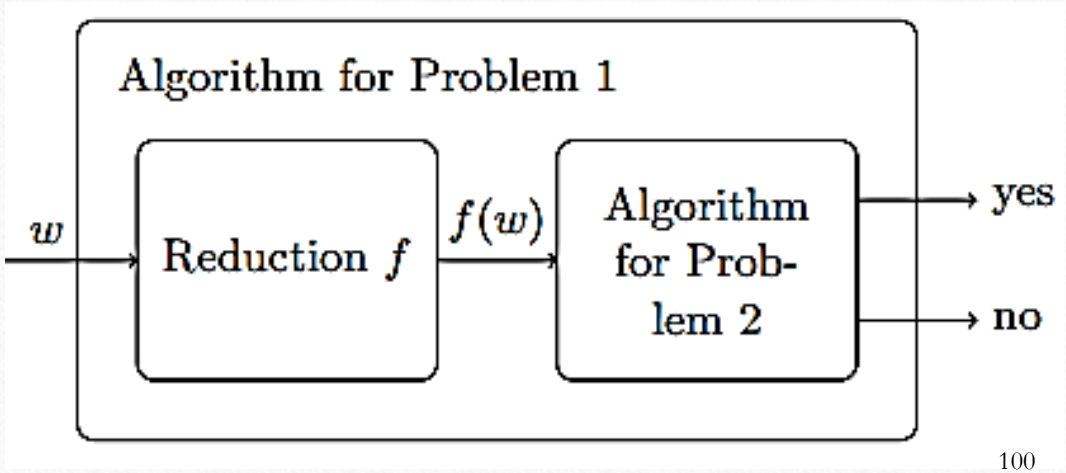
- The **state-entry problem** is as follows: Given any Turing machine M = (Q, Σ, Γ, δ, q0, □, F) and any q ∈ Q, w ∈ $\Sigma^+$, decide whether or not the state **q** is ever entered when **M** is applied to w. This problem is undecidable. (i.e. is state q can entered/halted when M is applied to w? )
- To reduce the halting problem to the state-entry problem, suppose that we have an algorithm **A** that solves the state-entry problem. We could then use it to solve the halting problem. For example, given any **M** and **w**, we first modify **M** to get **M'** in such a way that **M'** halts in state **q** if and only if **M** halts.
- We can do this simply by looking at the transition function **δ** of **M**. If **M** halts, it does so because some **δ (qi, a)** is undefined. To get **M',** we change every such undefined **δ to *δ* (*qi, a*) = (*q, a, R*)**, where ***q*** is a final state. We apply the state-entry algorithm *A* to ( M', *q*, *w*).
- If *A* says yes, the state ***q*** is entered, then (*M*, *w*) halts. If *A* says no, then (*M*, *w*) doesn't halt.
- Thus, the assumption that the state-entry problem is decidable gives us an algorithm for the halting problem. Because the halting problem is undecidable, the state-entry problem must also be undecidable

# Proving Undecidability via Reduction

- To prove a problem P is undecidable, we reduce a known undecidable problem Q to it. Proof by reduction entails:
- Assume there is a program or algorithm Decide_P that decides the problem P.
- Next, show that we can use Decide_P to decide Q, by translating an instance of the Q problem to an instance of the P problem and then applying Decide_P.
- The translation must itself halt.
- This yields a contradiction, since Q is known to be undecidable.
- Q is typically the halting problem.

# Proving Undecidability via Reduction

- *More amplification:*
- Suppose language **L1** reduces to **L2** and **L1** is undecidable.
- Then **L2** is undecidable.
- Proof Sketch: Suppose for contradiction **L2** is decidable.
- Then there is a M that always halts and decides **L2**. Then the next algorithm decides **L1**
  - On input **w**, apply reduction to transform **w** into an input **w'** for problem 2
  - Run M on **w'**, and use its answer.



# Proving Undecidability via Reduction- example

- $HALT_{TM}$ = {(M, w) | M is a TM and M halts on input w} is undecidable
- PROOF:
  - Use the idea that "If A is undecidable and reducible to B, then B is undecidable."
  - Suppose R decides $HALT_{TM}$ . We construct S to decide $A_{TM}$ .
  - S = "On input (M, w)
    - Run R on input (M, w).
    - If R rejects, reject
    - If R accepts, simulate M on w until it halts.
    - If M has accepted, accept; If M has rejected, reject."
  - Since $A_{TM}$ is reduced to $HALT_{TM}$, $HALT_{TM}$ is undecidable.

# Proving Undecidability via Reduction- example-2

- example-2 : emptiness of Turing machine
- $E_{TM} = \{(M) \mid M \text{ is a TM and } L(M) = \Phi\}$ is undecidable.
- Suppose R decides $E_{TM}$ .
- We try to construct **S** to decide $A_{TM}$ using R.
  - Note that S takes (M, w) as input.
- One idea is to run R on (M) to check if M accepts some string or not; but that does not tell us if M accepts w.
- Instead we modify M to $M_1$.
- $M_1$ rejects all strings other than w but on w, it does what M does.
- Now we can check if $L(M_1) = \Phi$.

# Proving Undecidability via Reduction- example-2

- PROOF:
- For any w define $M_1$ as $M_1$ = "On input x:
  1 If x ≠ w, reject.
  2 If x = w, run M on input w and accept if M does."
- Note that $M_1$ either accepts w only or nothing!
- Assume R decides $E_{TM}$ . S defines below uses R to decide on $A_{TM}$
- S = "On input (M, w)
  1 Use (M, w) to construct $M_1$ above.
  2 Run R on input ($M_1$)
  3 If R accepts, reject, if R rejects, accept.
- So, if R decides $M_1$ is empty,
- then M does NOT accept w,
- else M accepts w.
- If R decides $E_{TM}$ then S decides $A_{TM}$ – Contradiction.